EFFECTIVE SUBGOAL DISCOVERY AND OPTION GENERATION IN REINFORCEMENT LEARNING

A THESIS SUBMITTED TO
THE GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCES
OF
MIDDLE EAST TECHNICAL UNIVERSITY

ALPER DEMİR

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR
THE DEGREE OF MASTER OF SCIENCE
IN
COMPUTER ENGINEERING

AUGUST 2016

Approval of the thesis:

EFFECTIVE SUBGOAL DISCOVERY AND OPTION GENERATION IN REINFORCEMENT LEARNING

submitted by **ALPER DEMİR** in partial fulfillment of the requirements for the degree of **Master of Science in Computer Engineering Department, Middle East Technical University** by,

Prof. Dr. Gülbin Dural Ünver
Dean, Graduate School of **Natural and Applied Sciences** _____

Prof. Dr. Adnan Yazıcı
Head of Department, **Computer Engineering** _____

Prof. Dr. Faruk Polat
Supervisor, **Computer Engineering Department, METU** _____

Examining Committee Members:

Prof. Dr. H. Altay Güvenir
Computer Engineering Department, Bilkent University _____

Prof. Dr. Faruk Polat
Computer Engineering Department, METU _____

Prof. Dr. Kemal Leblebicioğlu
Electrical - Electronics Engineering Department, METU _____

Prof. Dr. İsmail H. Toroslu
Computer Engineering Department, METU _____

Assoc. Prof. Sinan Kalkan
Computer Engineering Department, METU _____

Date: _____

I hereby declare that all information in this document has been obtained and presented in accordance with academic rules and ethical conduct. I also declare that, as required by these rules and conduct, I have fully cited and referenced all material and results that are not original to this work.

Name, Last Name: ALPER DEMİR

Signature :

ABSTRACT

EFFECTIVE SUBGOAL DISCOVERY AND OPTION GENERATION IN REINFORCEMENT LEARNING

Demir, Alper

M.S., Department of Computer Engineering

Supervisor : Prof. Dr. Faruk Polat

August 2016, 56 pages

Subgoal discovery is proven to be a practical way to cope with large state spaces in Reinforcement Learning. Subgoals are natural hints to partition the problem into sub-problems, allowing the agent to solve each sub-problem separately. Identification of such subgoal states in the early phases of the learning process increases the learning speed of the agent. In a problem modeled as a Markov Decision Process, subgoal states possess key features that distinguish them from the ordinary ones. A learning agent needs a way to reach an identified subgoal, and this can be achieved by forming an *option* to reach it. Most of the studies in the literature focus on finding useful subgoals by employing statistical methods and graph-based methods. On the other hand, there are few studies working on how to improve the process of forming options.

In this thesis, an efficient subgoal discovery making use of local information is proposed. Unlike other methods, it has lower time complexity and does not

require additional problem specific parameters. Furthermore, a better heuristic for forming options is proposed. It focuses on collecting a set of states that an option is really useful to employ from, leading to more effective options.

Keywords: reinforcement learning, subgoal discovery, option framework

ÖZ

PEKİŞTİRMELİ ÖĞRENMEDE ETKİLİ ALT HEDEF BULMA VE OPSİYON OLUŞTURMA

Demir, Alper
Yüksek Lisans, Bilgisayar Mühendisliği Bölümü
Tez Yöneticisi : Prof. Dr. Faruk Polat

Ağustos 2016, 56 sayfa

Alt hedef bulma, pekiştirmeli öğrenmede, problem büyüklüğüyle başa çıkma konusunda kendini kanıtlamış önemli bir yaklaşımdır. Alt hedefler problemin, alt problemlere bölünmesi konusunda ipucu verir. Bu alt hedeflerin öğrenmenin erken aşamalarında bulunmaları, öğrenen ajanın her bir alt problemi ayrı ayrı çözmesine olanak sağlar ve öğrenme hızını arttırır. Markov Karar İşlemi olarak modellenmiş bir problemde, alt hedefler, ajanın bulunduğu diğer durumlara göre farklı özellikler taşır ve bu özellikler keşfedilmelerine olanak sağlar. Ajanın, bulunmuş bir alt hedefe yönelmesi için, opsiyon sistemi ortaya atılmıştır. Opsiyon sistemi, öğrenen ajana bulunmuş bir alt hedefe gitmek için bir yetenek kazandırır. Alandaki birçok çalışma, bu opsiyonların gideceği alt hedefleri bulma konusuna odaklanmış olup çalışmalar istatistik tabanlı ve grafik tabanlı olmak üzere ikiye ayrılır. Opsiyon oluşturma aşamasını geliştirme konusunda görece daha az çalışma bulunmaktadır.

Bu çalışmada, problem hakkında kısmi bilgi ile çalışan verimli bir alt hedef bulma yöntemi sunulmuştur. Bu yöntem, alandaki diğer yöntemlerin aksine, daha düşük

zaman karmaşıklığına sahiptir ve problem ile alakalı fazladan bir parametreye ihtiyaç duymamaktadır. Ayrıca bu tezde, opsiyon oluşturma aşaması için daha gelişmiş bir yaklaşım ortaya atılmıştır. Bu yaklaşım, opsiyon tanımını, opsiyonun kullanılmasının faydalı olacağı durumlar ile sınırlandırıp opsiyonları ana hedefe yönlendirir. Bu sayede, daha etkili opsiyonlar üretilir.

Anahtar Kelimeler: pekiştirmeli öğrenme, alt hedef bulma, opsiyon sistemi

Dedicated to my dear family.

ACKNOWLEDGMENTS

First of all, I would like to thank my thesis supervisor Prof. Dr. Faruk Polat. He led me to this field and always provided the support that a student may ever require. His leadership motivated me during this whole work. Finally, I would like to thank Erkin Çilden as he contributed to the study in many cases and his counselling helped me to solve the problems that I face.

This work is partially supported by the Scientific and Technological Research Council of Turkey under Grant No. 215E250.

TABLE OF CONTENTS

LIST OF TABLES

TABLES

LIST OF FIGURES

FIGURES

CHAPTER 1

INTRODUCTION

In our daily lives, we always use *abstractions* to avoid thinking about the already learned notions. To drive a car, we learn how to use the steering wheel and pedals in an order so that the car can move to the direction as we intended to. After the learning process is finished, we acquire a *skill* of driving a car and do not rethink the same set of actions as we already made an abstraction over them. This "driving a car" abstraction helps us planning our day without considering the primitive actions of using the pedals and the wheel.

Making abstractions enables any learning agent to plan in higher levels. In Reinforcement Learning (RL) [33], such abstractions save the agent's learning time by employing macro actions in addition to primitive ones. Since the aim is to increase overall cumulative reward, these abstractions may help in solving the task more efficiently. As the state space of a problem may be huge, having abstractions plays an important role in the learning phase.

As usual, a classical approach to cope with the effect of the problem size is the *divide and conquer* strategy. The idea is that, if one can divide a problem into sub-problems, it may be easier to solve the sub-problems first, and then combine the sub-solutions to achieve the answer to the overall problem. From the solution point of view, these sub-solutions are handled as abstractions.

Humans employ divide-and-conquer strategy frequently. Our brain constantly recognizes the sub-problems and solves them separately. For instance, if we need to go to a location in some other room, we realize that we need to reach

the door first, so we solve the problem of reaching the door and making an abstraction about it, avoiding the unnecessary exploration of the whole room. The bottleneck here is to recognize the door in order to partition the problem. In this example, the door may represent a *subgoal* where the *goal* is the location that we want to reach in another room.

A subgoal in the problem is a natural hint to partition it into sub-problems. Identification of such subgoals in the early phase of learning is proven to increase the learning performance. If an agent is capable of identifying a subgoal and making an abstraction to reach it, it may skip the exploration of the sub-problem that the subgoal partitions.

There are a number of different approaches that attack the subgoal discovery problem in RL. Some of the methods are based on graph theory [15, 24, 30, 35], some use statistical methods [4, 21, 29, 31], while others employ data mining approach [16, 19].

Obviously, since the intrinsic focus of RL is on *on-line* performance, it is quite reasonable to expect that the identification of subgoals should better be confluent with the underlying learning procedure. That is, a subgoal must be discovered during learning so that the agent can benefit from it. While some methods natively support this paradigm [11, 29, 30], some others may require additional setup.

Among the approaches for making abstractions, *Options* framework drew attention due to its generality and ease of implementation, where an *option* is nothing but a time extended *abstract action* (or *macro action*). Definition of an option involves an initiation set (states at which an option may start) and a termination condition (how an option terminates). Subgoal discovery, usually deals with how to accurately identify the termination condition, since its quality determines effectiveness of the partitioning.

1.1 Definition of a Subgoal

A subgoal in a problem is defined in different ways in the literature. Some definitions [21, 24] are task dependent, meaning that a subgoal is the state that is visited frequently on the way to a possible goal state. These definitions require that the agent recognizes the goal states in the problem. Other definitions [28, 15, 35, 4, 31, 19, 16] are based on the being a bottleneck idea. A subgoal acts as a bottleneck state allowing the agent to travel between different regions of the state space.

In this thesis, the latter definition is adopted as the proposed subgoal identification method focuses on finding bottleneck candidates in the early phases of learning.

1.2 Contributions

The thesis work proposes two main contributions as a novel subgoal discovery method [6] and an improvement of the option generation process [5].

The proposed subgoal identification method is coherent with RL paradigm as it can be employed on-line and it works with local information. Having these features, it resembles with the methods proposed by Simsek [28] and it uses the same filtering rule as theirs. The main prominent characteristics of the proposed method are as follows.

- Compared to existing methods, the proposed method requires less CPU time with an average time complexity of $O((\log_b(n))^2)$.
- It maintains the partial information about the problem in a more compact way as it leads to low memory usage.
- It achieves the same quality in finding useful subgoals as the other subgoal discovery methods and it identifies less number of subgoals ignoring the similar subgoals having no extra effect on the learning process.

- It does not require any additional parameters besides the ones used in the filtering mechanism.

The proposed option generation method focuses on the initiation set part of an option by creating *directed options*. It aims to form more effective options in the sense of learning performance.

- It significantly improves the learning performance when the same set of subgoals are provided.
- It forms better options in terms of quality measured as usefulness and compactness.
- It requires reasonable computation time.

1.3 Outline

The organization of the thesis is as follows. Chapter 1 introduces the problem at hand and the existing approaches to solve it. Chapter 2 summarizes the necessary background and related works in the literature. Chapter 3 and Chapter 4 present a novel subgoal discovery method and a new way of generating options, respectively, along with the experimental results and discussions. Chapter 5 concludes the research work and proposes future research directions.

CHAPTER 2

BACKGROUND AND RELATED WORK

This chapter gives the necessary background to understand the addressed problem and the proposed improvements in this thesis. It introduces the formalisms that the problem is modeled, summarizes existing methods in the literature.

2.1 Markov Decision Processes

Formally, a Markov Decision Process (MDP) is a tuple $\langle S, A, T, R \rangle$, consisting of

- a finite set of *states* S,
- a finite set of *actions* A,
- a transition function $T : S \times A \times S \rightarrow [0, 1]$ where $\forall s \in S$, $\forall a \in A$, $\sum_{s' \in S} T(s, a, s') = 1$,
- a reward function $R : S \times A \rightarrow \Re$.

$T(s, a, s')$ is the probability of being in state s' if action a is performed in state s. $R(s, a)$ gives the immediate reward from the environment after taking action a in state s.

T states whether the MDP is deterministic or not. In the nondeterministic (stochastic) case, the actions become noisy, i.e., the same action may not lead the agent to the same state all the time.

A policy

$$\pi : S \times A \rightarrow [0, 1] \tag{2.1}$$

is a mapping defining the probability of selecting an action in a state. The aim is to find the optimal policy π^* which maximizes the total expected reward received by the agent. If reward and transition functions are unknown, the optimal policy π^* needs to be found by estimating the *value function* (i.e. function giving the value of being in a state on the way to goal) incrementally. *Incremental estimation* approach makes use of the average cumulative rewards over different trajectories obtained by following a policy to calculate the value function and gives rise to the central idea of most RL algorithms, called the *temporal difference* (TD) [32].

2.2 Semi Markov Decision Processes

A *Semi-Markov Decision Process* (SMDP) extends the MDP model with transitions of stochastic time duration. An SMDP is a tuple $\langle S, A, T, R, F \rangle$, where S, A, T and R define an MDP, and $F(t|s, a)$ is the probability that starting at s, action a completes within time t. MDP is clearly a specialization of SMDP, where a step function has a jump at 1. In the SMDP model, a policy is still a mapping from states to actions, thus the Bellman equations [1] still hold for an optimal policy [2].

2.3 Reinforcement Learning

Reinforcement Learning (RL) defines a family of machine learning methods that try to solve a decision problem by utilizing the feedback received on-line while experiencing the problem space. It also differs from supervised learning techniques in the sense that there is no presentation of correct decisions, yet the agent has to derive them by itself through processing of environmental feedback [33].

A famous TD algorithm using action-values (i.e. Q-values) instead of state-values is named Q-Learning [36], and is widely used due to its simplicity and ease of use. Q-values represent the value of taking an action a in a state s. The update rule for Q-Learning is

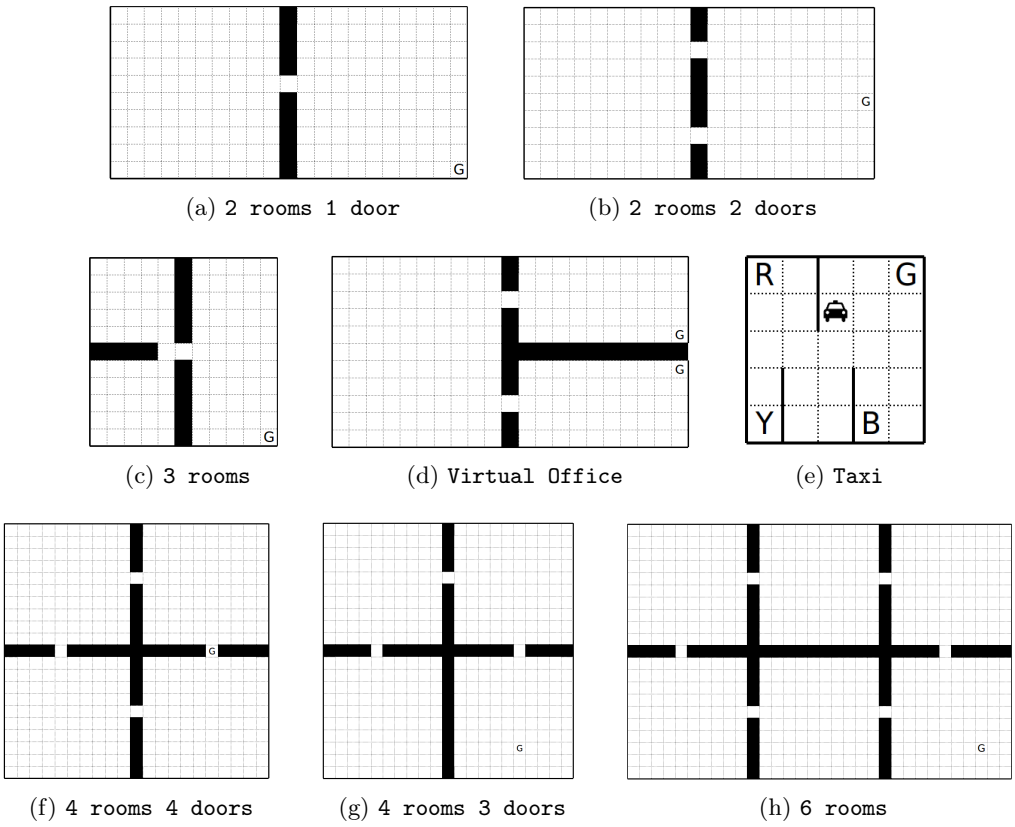
$$Q(s,a) \leftarrow (1-\alpha) \times Q(s,a) + \alpha \times [r + \gamma \times \max_{a' \in A} Q(s',a')], \tag{2.2}$$

where $\alpha \in [0,1)$ is the *learning rate*, $\gamma \in [0,1)$ is the *discount factor*, s' is the reached state when action a is fired at state s and a' is the action leading to the maximum Q-value in state s'. Q-Learning has been shown to converge to the optimal action-value function denoted by Q^*, under standard stochastic approximation assumptions.

2.4 Options Framework and Macro-Q Learning

An implicit assumption for the MDP model is that an action lasts for a single time step. However, there are acceptable rationales to relax this assumption. An obvious one would be the convenience in the reuse of a behaviour pattern (i.e. skill) in different situations within the problem space. This *abstraction* idea took attention by various researchers, and a few different mainstream approaches emerged [7, 26, 34].

As a prominent abstraction formalism based on the SMDP model, *options framework* [34] devices a way to define and invoke timed actions via incorporation of composite actions on top of an MDP model. It allows to create and use abstract actions (*options*) by using primitive actions, lasting for a finite number of discrete time steps. Briefly, an option is defined by three components $< I, \pi, \beta >$: (1) a set of states that the option can be initiated at, called the *initiation set*, I, (2) option's local policy, π, and (3) a probability distribution induced by the *termination condition* β.



(a) `2 rooms 1 door` (b) `2 rooms 2 doors`

(c) `3 rooms` (d) `Virtual Office` (e) `Taxi`

(f) `4 rooms 4 doors` (g) `4 rooms 3 doors` (h) `6 rooms`

Figure 2.1: Sample problem domains

A natural extension of Q-Learning to include options is Macro-Q Learning [22], where the value of each primitive action is again updated according to regular Q-Learning (as given in the update rule 2.2), while the value of an option is updated according to the following rule:

$$Q(s_t, o_t) \leftarrow Q(s_t, o_t) + \alpha \times (\gamma^n \times \max_{o'} Q(s_{t+n}, o') - Q(s_t, o_t) \\ + r_{t+1} + \gamma r_{t+2} + ... + \gamma^{n-1} r_{t+n}), \tag{2.3}$$

where s_t is the starting state of the option o_t, n is the number of steps taken while the option is employed, s_{t+n} is the state that the option terminates at, o' is the option from s_{t+n} that has the maximal value and r_{t+i} is the reward received at time $t + i$. The reward is discounted by the time it is received.

Table2.1: Problem sizes and reference publications (problems marked with * sign are modified versions of their originals)

Problem	|S|	|A|	Reference
`2 rooms and 1 door`	201	4	[30]
`2 rooms and 2 door`	202	4	[24]*
`3 rooms`	106	4	-
`Virtual Office`	212	4	[8]*
`4 rooms and 3 doors`	404	4	[21]*
`4 rooms and 4 doors`	403	4	[21]
`6 rooms`	605	4	[24]
`Taxi`	500	6	[7]

2.5 Sample Problem Domains

Commonly used sample domains include grid world navigation problems (Figure 2.1). They are well-known benchmark problems in the related literature. Table 2.1 include sizes of the state space and action space of the experimented domains. New grid world domains and different versions of existing ones are also included to further analyze the behaviour of the proposed improvements.

In the grid world problems, except `Taxi`, the agent can perform four movement actions, move *north*, *east*, *south* and *west*. The environment is non-deterministic, and the agent moves to the intended direction with probability 0.9 and moves randomly in any of the movement directions otherwise. The agent is given a reward of +1 if it reaches the goal state and transitions to nongoal states yield 0 reward. For 2,3 and 4 rooms problems and `Virtual Office`, the agent starts from any cell in the left room(s). The agent starts from any cell in the upper left room in the `6 rooms` domain.

The last domain is the famous `Taxi` domain (Figure 2.1e, [7]), in which a taxi tries to pick up a passenger from its location and transfer it to a destination location in a 5×5 grid world with designated locations. The taxi agent can perform 6 actions: movement actions *north*, *east*, *south*, *west*; a *pickup* action to get the passenger, and a *putdown* action to drop the passenger. The passenger is initially located in one of the four different cells marked as Red (R), Yellow (Y),

Green (G) and Blue (B), and the destination of the passenger is one of these four designated cells. The actions are noisy, leading the agent to its intended direction with probability of 0.8 and randomly moving it to the left or the right of the intended direction with probability of 0.1 each. The agent is punished for wrong pickups and putdowns with -10 and it is rewarded with $+20$ when it puts down its passenger in the desired location. Any other transition is given the penalty of -1. A state in this problem is made up of the cell that the taxi is currently located (25 cells), the destination of the passenger (4 designated cells) and the position of the passenger (4 designated cell or taxi).

In `Taxi` domain, as the destination of the passenger is constant during an episode, the number of states that can be visited in an episode is limited to 125. On the contrary, the agent can visit all the states in an episode in the other problems. This causes the learning to take more episodes in `Taxi` domain than the other ones. However, the agent constantly receives rewards in `Taxi` domain while in other grid world domains, it receives a reward only when it reaches to the goal state. Again this difference leads agent to take less steps in an episode in `Taxi` domain than the others.

2.6 Related Work

2.6.1 Automatic Subgoal Discovery

Automatic discovery of subgoals deals with the problem of identifying a set of intermediate points or regions within an MDP, that are "subgoals" or "bottlenecks", naturally partitioning the problem in hand. Due to the vagueness of the concept, a number of different approaches had been developed for subgoal discovery in RL context.

Some of the methods transform the experience history to a transition graph and analyze it to find most suitable bottleneck regions that partitions the problem [9, 15, 24, 25, 30, 35]. Some other methods rely on state visitation statistics to find frequently used states, based on the observation that frequently visited states are more likely to be a bottleneck on the way to goal [4, 12, 21, 29, 31]. A

yet different approach interprets the same matter as a clustering problem, trying to find separate regions in state space using experiences and then identify access points between regions as subgoals [16, 19]. Although not explicitly subgoal-based, a related family of methods focuses on the sequence analysis on episode histories, under the assumption that the subgoals are signaled via reward peaks [11, 20].

However, it is not straightforward to determine whether a state is a subgoal or not. In the ideal case, one needs the complete transition function T in order to make an accurate decision, which is practically not possible. Nevertheless, majority of subgoal discovery methods rely on the assumption that an approximate T can be gathered throughout the RL experience. A drawback of this approach is that it may not be possible to decide that approximation of T is accurate enough to be used for subgoal discovery. Alternatively, few other methods use a hybrid approach that brings together locally collected transition information and a means to statistically test its sufficiency for subgoal discovery [28].

2.6.1.1 Access States

Simsek [28] puts a definition on a subgoal, called an *access state*, as a state that connects two or more connected regions having few transitions in between. The key idea is that, a method searching for an access state is allowed to possess only local statistics throughout the experience, to classify a state as either a target state (an access state) or not. Observations that are collected for a state this way are then used in the following decision rule:

$$\frac{n_+}{n} > \frac{\ln \frac{1-q}{1-p}}{\ln \frac{p(1-q)}{q(1-p)}} + \frac{1}{n} \frac{\ln(\frac{\lambda_{fa}}{\lambda_{miss}} \frac{p(N)}{p(T)})}{\ln \frac{p(1-q)}{q(1-p)}}, \tag{2.4}$$

where n is the total number of observations for a state, n_+ is the total number of positive observations for a state, p is the probability of a positive observation given a target state (an access state), q is the probability of a positive observation given a non-target state, λ_{fa} is the cost of a false alarm, λ_{miss} is the cost of a miss, $p(N)$ is the prior probability of non-target states and $p(T)$ is the prior probability of target states.

If the inequality holds, then the state is classified as an access state. This decision rule pinpoints the time step when the collected observations are enough to make a decision about the label of a state. This two-level mechanism enables the methods to avoid the time cost of traversing the whole problem domain. On the other hand, the parameters are problem specific and needed to be either guessed or estimated with the help of prior experiments. The analysis of the parameters can be found in [28].

The main reason behind having a second level decision rule is that the subgoal identification methods proposed by Simsek can lead to noisy results as they work on partial information. It is possible that the agent walk in a way so that the resulted interaction may show a different problem structure, suggesting a regular state as a subgoal. Such noises are eliminated with inequality 2.4.

In the same study, Simsek proposes three access state identification methods (Local Betweenness, Local Cuts, Relative Novelty) to generate the observations for the states that are required for the inequality 2.4. Two of them are graph-based methods working on *local interaction graphs* and one of them is a frequency based method collecting statistics about the states.

A *local interaction graph* is a weighted and directed graph constructed with the interactions between the agent and the environment. As the name suggests, the graph possesses only local information. For example, a local interaction graph created at the end of an episode may not have all the states in the problem domain as those states may remain unvisited within that episode.

2.6.1.2 Relative Novelty

Relative Novelty (RN) focuses on the idea that access states are more likely to allow the agent to go from a highly visited region to another new region in the state space. The method employs a statistical approach. The agents keeps a history of states that it visits and calculates the relative novelty metric on the states.

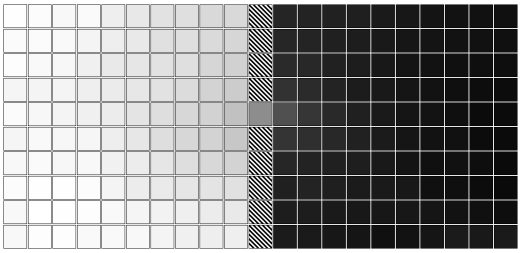


Figure 2.2: `2 rooms 1 door` grid world domain colored according to state visitations where brighter color means higher visitation.

Novelty of a set of states is defined as $n^{-\frac{1}{k}}$, where n is the mean number of visits to states in the set and $k > 0$ is a parameter called *novelty exponent*. With this parameter, the novelty of a set decays with the increasing number of visits.

Relative novelty of a state is the ratio of the novelty of following states (including itself) to the novelty of preceding states. The number of states to consider in these sets is determined with a parameter called *novelty lag* (l_n). Since calculating relative novelty of a state needs a set of following states, the calculation is done when the history of states allows the state of interest to have a following set.

A state is said to generate relative novelty when its relative novelty value is bigger than an estimated threshold called *relative novelty threshold* (t_{RN}). If the value of a state is greater than the threshold, the algorithm generates a positive observation for the state, it generates a negative observation otherwise. Then, the inequality 2.4 is checked and the state is classified accordingly.

The method resets the visitation counts periodically, since only the recent experience is important to the classification. A very suitable time for resetting is the end of each episode. Also, self transitions are ignored, that is consecutive visitations for the same state are skipped.

The algorithm has $O(1)$ time complexity as it consists of a basic calculation. However, RN requires to be employed at each step. In an episodic view, the complexity of it becomes $O(m)$ where m represents the number of steps taken in that episode.

To give a further idea, `2 rooms 1 door` grid world domain with state visitation frequencies is shown in Figure 2.2 where a cell with brighter color is visited more often. Due to the early exploration, the agent spends more time on the left room than the right one. The method depends on this difference in the visitation counts and with the relative novelty metric, it identifies the states around the doorway as access states. On the other hand, RN is highly open to generating noise as it depends on the idea that the agent explores the problem with an unbiased way. That is why, it performs best when random action selection is employed. If the agent spends some extra time on some set of regular states, possibly due to a pitfall, the method may consider that set as a connected region and it may mark the state that the agent exits the set as an access state.

2.6.1.3 Local Betweenness

The method of local betweenness (LoBet) uses a graph-based approach to identify bottlenecks. It uses local interaction graphs to identify bottleneck states.

Betweenness of a vertex in a graph is defined as the ratio of shortest paths on the graph, between all possible sources and targets, that pass through the vertex of interest to the total number of shortest paths. Local betweenness is the betweenness of a state computed in a local interaction graph.

In addition to the betweenness metric, Simsek also considers the path weights which are calculated according to the rewards taken in a path and the number of transitions between the states in a path.

Hence, the betweenness value of a vertex v is calculated as,

$$\sum_{s \neq t \neq v} \frac{\sigma_{st}(v)}{\sigma_{st}} w_{st}, \tag{2.5}$$

where σ_{st} is the number of shortest paths from vertex s to t, $\sigma_{st}(v)$ is the number of such paths that pass through v, w_{st} is the weight of the path from s to t.

This metric directly shows the important bottleneck nodes in a graph, corre-

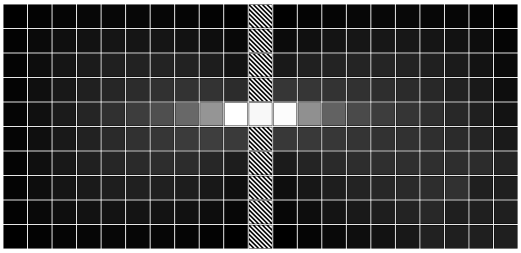


Figure 2.3: `2 rooms 1 door` grid world domain colored according to betweenness values of the states where brighter color represents higher betweenness.

sponding to access states in a problem domain. For the same example of `2 rooms 1 door` grid world domain given in Figure 2.3, the states with higher betweenness values are easily detectable. The algorithm generates a positive observation for a state if its betweenness is local maxima among its immediate neighbors in the graph, and a negative observation otherwise. Finally, those observations are fed to the inequality 2.4.

With Brandes' algorithm [3], a local interaction graph can be processed for betweenness of each node in $O(n \cdot m)$ time with unweighted edges, $O(n \cdot m + n^2 \cdot \log n)$ with weighted edges where n is the number of nodes and m is the number of edges in the graph.

One problem about LoBet is that it may miss a set of access states when they are neighbors to each other. In a version of the `2 rooms 1 door` grid world domain with 2 cells in the doorway, the method will miss both cells when the local interaction graph is symmetric around the doorway, since LoBet marks a state as access state if its betweenness is local maxima around its neighbors. In the symmetric case, both states in the doorway may have the same betweenness, causing the method to miss both of them. However, this problem does not occur as much because of partial information on domain structure.

2.6.1.4 Local Cuts

Local Cuts (L-Cut) is another graph based approach proposed by Simsek [28]. Basically, it aims to partition the local interaction graph into two according to a metric called *normalized cut* (*NCut*) introduced by Shi and Malik [27].

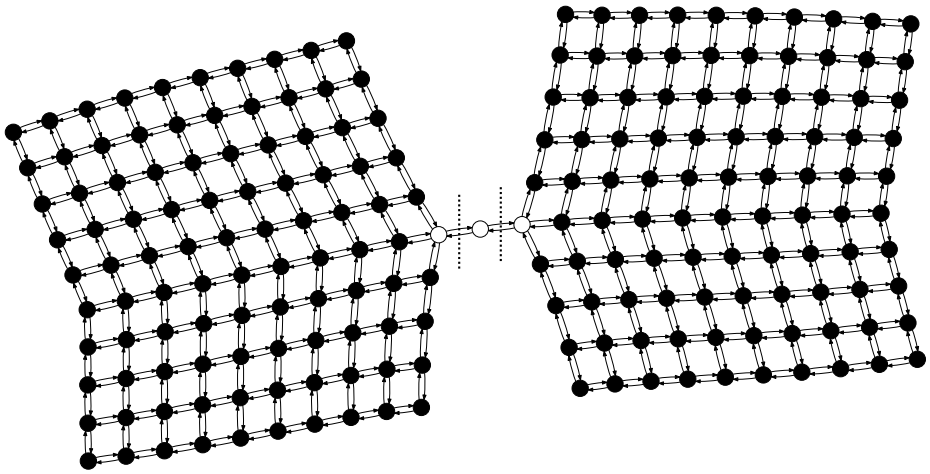


Figure 2.4: `2 rooms 1 door` grid world domain interaction graph where each node represents a state and each edge represents a transition. Possible cuts are shown with dashed lines and access states that may be identified by L-Cut is shown with white coloring.

NCut is defined as,

$$NCut(A, B) = \frac{cut(A, B)}{vol(A)} + \frac{cut(B, A)}{vol(B)}, \tag{2.6}$$

where A and B are two partitions, $cut(A, B)$ is defined as the sum of the weights of the edges that start in A and end in B and $vol(A)$ is defined as the sum of the weights of all edges that start in A.

The algorithm employs spectral clustering of Shi and Malik [27] for the partitioning of the local interaction graph. Then, it calculates *NCut* value of the partitioning for determining the cut quality. If *NCut* value is lower than a predetermined threshold called *cut threshold* (t_c), indicating that the partitions are well separated, then the border states of this cut get a positive observation while the others get a negative one. Again, these observations are given to the inequality 2.4 for further elimination.

In the example of `2 rooms 1 door` grid world domain with complete interaction graph given in Figure 2.4, possible cut positions are given with dashed lines. The method finds two partitions, namely the left and the right room. Having *NCut*

value lower than t_c, generates positive observations to the possible border states marked with white color. As the method works on each local interaction graph, inequality 2.4 separates each of them from the remaining states.

The partitioning of the graph requires $O(n^3)$ time where n is the number of states in the local interaction graph. Since the local interaction graph may not have all the states in the problem domain, this high time complexity is not a problem.

Unlike the other two methods, L-Cut always points two candidates for being access states as the cut always has two ends. This may lead to having more access states compared to the other methods.

2.6.2 Option Generation and Experience Replay

Option framework does not impose any strategy for designing meaningful or useful options. Although there are "guidelines" giving some insight to the dynamics of the framework, the formalism itself lacks procedural ways for defining a valuable option. A general rule of thumb would be, "the initiation set and termination condition of an option together restrict its range of application in a potentially useful way" [34].

However, many of the existing automatic option generation algorithms focus on the termination condition, mostly due to the fact that (as an obvious strategy) a partitioning attempt should find splitting points or regions in the state space and let the termination criteria of options be derived by using those bottleneck states.

Initiation sets, on the other hand, are usually defined via a simpler premise, like "all states other than termination states," hoping for the action selection mechanism to restrict choices during learning later on. "All other states" heuristic is probably one of the simplest ideas for construction of the initiation sets [13]. Some methods improve this idea by introducing additional –direct or indirect– information about the domain. Restrictive parameters like "option lag" can be used to provide a problem specific limit for the option length [29]. Some meth-

ods incorporate a *reachability* criterion for states to the ones in the termination condition as a heuristic [31].

In a relatively less explored approach, sub-sequence commonalities are explored instead of bottlenecks [23, 10]. This flexibility allows the methods to generate useful options from common parts of histories. While [23] stores a history database for this purpose, [10] introduces shortcut history procedure and a compact tree structure eliminating the excessive memory requirement, which inspired our work. Unlike [10] which focuses on sequences including actions, our work uses the similar history tree heuristic with only state information in order to identify the set of states that an option can be useful to be employed from.

There are also studies that expand the notion of option to handle problems that do not fit to classic MDP model. For problems with continuous state space, initiation set of discovered skills can be the terminal set of the next skill, forming a skill chain [17]. Reachability heuristic is also studied, although in a different form, for factored MDP setting [14]. These methods apply to the relevant extended MDP models, and are not directly related to our work.

As the third component of an option, a common way of forming the *option policy* is the *Experience Replay* (ER) mechanism [18]. ER makes use of a repository of past episodes and replays them repeatedly as if the agent experiences them again. In option generation, a policy for reaching a state can effectively be formed by using ER via artificial *internal* rewards rather than the ones yielded by the environment. Traditionally, the ER rewards are designed in such a way that reaching the state gets a very high reward, while reaching the states of the initiation set are punished.

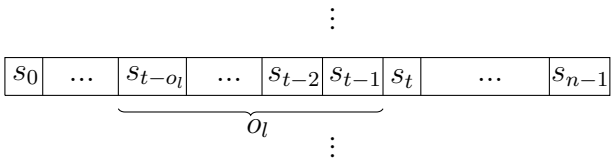


Figure 2.5: Greedy selection of initiation set states with the option lag heuristic. A state history with length l is given with terminal state s_t and option lag o_l.

2.6.2.1 Option Lag Heuristics

Figure 2.5 illustrates one of the popular greedy strategies for forming an initiation set for a given terminal state (possibly a subgoal). It selects the initiation set states among the ones visited before the terminal state occurrences within an episode. As the agent explores different paths to the terminal state in each episode, the method collects the states in these paths.

The number of transitions to check before the terminal state of the option is an externally supplied parameter called the *option lag*. An episode starts with the state s_0 and ends with the state s_{n-1} with a total history length n, and the number of transitions checked before the terminal state s_t (with an option lag of o_l) are indicated in the figure.

This strategy is based on the idea that there is a path reaching the terminal state s_t from each state visited before it, and this time ordering is likely to occur again in the future episodes, meaning that the agent will visit the same states in a similar order to reach the terminal state. As this strategy depends on a fragile intuition, there is a room for improvement which is proposed in Chapter 4.

CHAPTER 3

TREE BASED SUBGOAL IDENTIFICATION

The aim of reinforcement learning is to find an optimal policy (π^*) to solve a planning task where π^* represents an optimal set of actions to reach a goal state from any other state in the problem. Following this policy, the agent can form a shortest path from any state to a goal state. Such a shortest path is an ordering of states according to their values leading to a goal state.

Even though the optimal policy is not given to the agent, it is possible to derive such an ordering after a successful episode. A *successful episode* is the one that ends with a reward peak, possibly reaching to a goal state. With such an information at hand, the agent can form shortcuts for reaching the reward peak from any state visited in the episode.

By eliminating loops and choosing the best states to go from any state in the episode history, such shortcuts may provide further information about how the states in the episode may have been visited efficiently to reach to a possible goal state. This idea is influenced from Girgin's study of shortcut history procedure [10]. As these shortcut paths merge, they form a tree that is called a *history tree*. In a history tree, some states may act as bottleneck states in the way to a goal state. Being the main advantage of automatic subgoal discovery methods, reaching to such states in the early phases of learning may accelerate it in a significant way.

In this study, a *subgoal* state is defined as a state that serves as a junction point or a region of the derived shortcut paths from each state to a goal state. Based

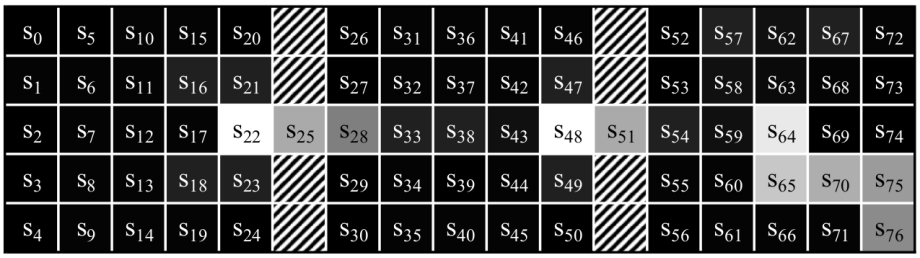
on this intuition, the algorithm requires the episode to end with a peak reward so that it can form a history tree to reach it.

3.1 Local Roots

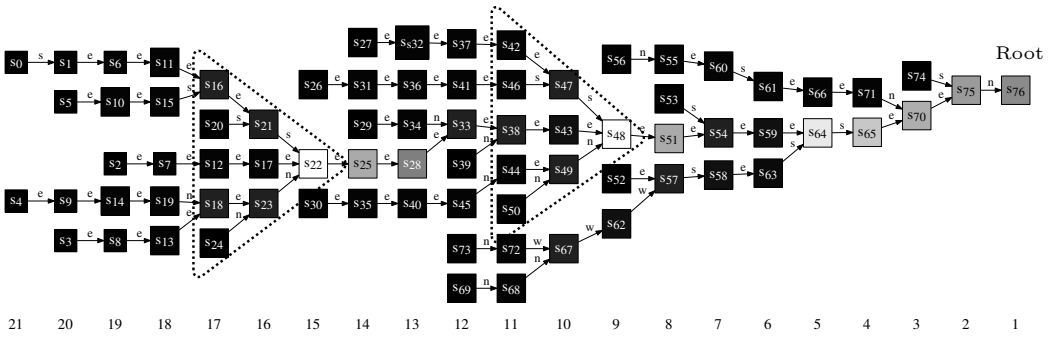
The proposed method, named *Local Roots* [6], generates positive and negative observations for visited states and feeds them to the inequality 2.4. As it is mentioned in Chapter 2, this is a common pattern in local approaches in subgoal discovery, since the local information gathered from the episode history can be highly dependent on the particular way of state visitations, and thus, may give rise to noisy results without a high level decision filter (especially false positives). Inequality 2.4 is calculated for each visited state, aiming to distinguish the subgoals more accurately.

Local Roots method records the transition history for each episode (Algorithm 1, line 3). Upon completion of an episode, it first checks whether the last transition yields the maximum reward for that episode, or not. If so, it calculates the average number of distinct transitions made through a state (nt_{avg}) and creates a history tree using shortcut paths derived using state equivalences, to serve as a collection of the best "memorized" trajectories starting from every visited state in the episode up to the goal state (Algorithm 1, line 6). As this history tree consists of the states only visited in the episode, it is called *local history tree*.

Best trajectories are calculated by traversing from a leaf of the tree to the root, iteratively updating the transitions with the best values. The path from a vertex to the root in the tree forms the shortest path from the corresponding state to (possibly) the last state in the episode. The tree generation procedure is given in Algorithm 2. Starting from the last transition reaching to the reward peak going to the beginning of the episode, each transition is used to calculate a value $\mathcal{V}$ for each visited state. Such values, are formed with the rewards taken during these transitions and the discounted values of their end states. If there is a transition reaching to a better state s' from a state s, then the *best* state to go for s is set to s'. Collecting these relations, a directed edge from s to its best state is added



(a)



(b)

Figure 3.1: (a) A sample grid world with two consecutive subgoals, colored according to rooting factor values of the states. Shaded cells represent walls. (b) The generated tree, using the same coloring scheme. Actions are noted on the edges. The numbers at the bottom are corresponding levels of the tree.

to the tree. The resulting local history tree is a collection of shortcut paths (i.e. free of loops) from every visited state in the episode to the last state (with the reward peak), based on the local history derived from experiences.

The core idea of the proposed method lies in a state metric, that is called the *rooting factor*, due to the visual resemblance to a root structure of a tree in the nature fringing underground. To clarify the idea, Figure 3.1a illustrates a sample grid world domain made up of three rooms with passageways between adjacent rooms, and Figure 3.1b is a tree generated for the problem by using an episode history. The goal state is s_{76} which is located in the south-east corner of the room and the agent starts from the north-west corner, namely s_0. The agent can move to any one of the four compass directions at each time step, except that after a move attempt to the walls and the boundaries of the room, it stays still.

The grid world and the tree instance given in Figure 3.1 are colored according to the rooting factor values of states scaled from black to white, where brighter color means a higher value. Note that the states in the doorways have high rooting factor values. In the case of state s_{48} located at level 9, the rooting factor metric focuses on the sub-tree having state s_{48} as the root. To calculate the rooting factor of state s_{48}, one should first find the *widest level* for the sub-tree rooted by state s_{48}, which is the level possessing the first peak in sub-tree width. In this sub-tree, the widths of each level are 1,3,5,5,5,5,3 consecutively, and the first peak value in terms of width is 5. The method considers level 11 as the widest level and ignores the second peak with value 5 starting in level 17 since level 11 has the first peak. This way, even if there is a wider level below, it is not taken into account for state s_{48}. Thus, each bottleneck state in the tree possesses its relative sub-tree, as is the case for another subgoal s_{22} in level 15 where the widest level of its relative sub-tree is level 17.

Upon identification of the widest level for state s_{48}, one can think of an imaginary triangle (i.e. the dotted triangle in Figure 3.1b) where the vertices in the widest level compose its base, and its topmost corner is s_{48} (w.r.t. a portrait orientation of the tree, where the root is at the top). The shape of this triangle is an indication of the "importance" of state s_{48} in the tree. A wider triangle suggests that, for relatively more states, the agent should pass through state s_{48} in order to reach the root state (i.e. goal). The height of the triangle, on the other hand, pinpoints a state which is the "first" junction point of the merging paths. That is why, the rooting factor of state s_{48} is higher than its parent's, state s_{51}.

As a mathematical interpretation of the above characteristics, the rooting factor of a state s can be defined as follows:

$$r_s = \frac{(n_{widest})^{nt_{avg}}}{d_{widest} - d_s}, \tag{3.1}$$

where d_s is the depth of s in the tree, nt_{avg} is the average number of distinct transitions of states in the tree, n_{widest} is the number of vertices in the widest level and d_{widest} is the depth of the states in that level. In order to strengthen the effect of possible connections that a vertex can have, number of vertices in the widest level is powered by the average number of distinct transitions (nt_{avg}).

Algorithm 1 *LOCAL_ROOTS*

Require: p, q, $\frac{\lambda_{fa}}{\lambda_{miss}}$, $\frac{p(N)}{p(T)}$

1: $o_i \leftarrow 0$, $o_i^+ \leftarrow 0$
2: **for** each episode **do**
3: $h \leftarrow$ Interact with the environment ▷ record episode history
4: **if** h ended with a peak reward **then**
5: $nt_{avg} \leftarrow$ calculate average number of distinct transitions in h
6: $T \leftarrow CREATE\_TREE(h)$
7: $CALCULATE\_ROOTING\_FACTORS(T, nt_{avg})$
8: **for** $s \in V_T$ **do**
9: $o_s \leftarrow o_s + 1$
10: **if** s is a local maximum on T **then**
11: $o_s^+ \leftarrow o_s^+ + 1$
12: **end if**
13: **if** the decision rule is satisfied **then** ▷ use inequality 2.4
14: Classify s as a subgoal
15: **end if**
16: **end for**
17: **end if**
18: **end for**

After the tree is constructed by using Algorithm 2, the rooting factor calculation takes place for every vertex (Algorithm 1, line 7). Algorithm 3 is employed for this purpose, where a tree traversal (via breadth first search, BFS) is employed first, to find the depth of each vertex in the tree. An additional traversal is run afterwards, from the level with the deepest state(s) to the root, to find the number of vertices below each vertex classified by their depths (Algorithm 3, lines 3-7). Using this information, the rooting factor of each state can be calculated by traversing from the state under consideration to the deeper levels.

Having calculated the rooting factor values for every visited state, each state is checked whether it is a local maximum or not, in terms of rooting factors, among its children and parent in the tree. The state gets a positive observation if that is the case, or a negative observation otherwise. The root of the entire

Algorithm 2 *CREATE_TREE*

Require: a successful episode trajectory h

Ensure: a tree T representing shortcut histories to goal from each state

$t \leftarrow length(h) - 2$
$\mathcal{V}_{s_{t+1}} \leftarrow 0$, $best_{s_{t+1}} \leftarrow null$
while $t \geq 0$ **do**
    if $\mathcal{V}_{s_t}$ is undefined or $(r_{t+1} + \gamma * \mathcal{V}_{s_{t+1}}) > \mathcal{V}_{s_t}$ **then**
        $\mathcal{V}_{s_t} \leftarrow r_{t+1} + (\gamma * \mathcal{V}_{s_{t+1}})$
        $best_{s_t} \leftarrow s_{t+1}$
    end if
    $t \leftarrow t - 1$
end while
$V \leftarrow \{s_l\}, E \leftarrow \emptyset$
for each state $s \neq s_l$ **do**
    $V \leftarrow V \cup \{s\}$
    $E \leftarrow E \cup (s, best_s)$
end for
return (V, E)

tree does not get a positive observation since it is a possible goal state and is obviously not a subgoal. The observations made for each state are fed to the inequality 2.4 for a further classification.

The most time consuming portion of the Local Roots algorithm is the part where the rooting factor value is calculated for each state which has $O(n^2)$ worst time complexity. However, the worst case happens when the tree is linear (although it is usually unlikely at the initial states of learning due to the exploration component, the agent might execute a policy that visits each state only once) and the branch factor (b) is 1, which means, by the definition, there is no new subgoal to identify. Thus, the worst case can be avoided by a heuristic check on the shape of the generated tree. On the other hand, the algorithm performs $O((\log_b(n))^2)$ on the average, where n is the number of nodes in the tree. Since the algorithm makes use of local episode trajectories, the number of nodes in the tree (n) does not directly relate to the number of states in the whole domain.

Algorithm 3 *CALCULATE_ROOTING_FACTORS*

Require: a successful history tree T, nt_{avg}

calculate the depth of each state in T ▷ use BFS
$d_{max} \leftarrow \max_{s \in V_T}(\text{depth}(s))$ ▷ find maximum depth
for every $s \in V_T$ **do**
    $n_i(s) \leftarrow$ number of nodes at depth $i \geq \text{depth}(s)$ in the subtree rooted at s
end for
for each state s in V_T **do** ▷ rooting factor calculation for every vertex
    $d_s \leftarrow \text{depth}(s)$
    $i \leftarrow d_s, n_{widest} \leftarrow 1, d_{widest} \leftarrow d_s$
    while $i \leq d_{max}$ and $n_i(s) \geq n_{widest}$ **do**
        if $n_i(s) > n_{widest}$ **then**
            $n_{widest} \leftarrow n_i(s)$
            $d_{widest} \leftarrow i$
        end if
        $i \leftarrow i + 1$
    end while
    $r_s \leftarrow$ calculate the rooting factor of s using Equation 3.1
end for

3.2 Experiments

3.2.1 Settings

We compared Local Roots method (LoRoots) with RN, LoBet and L-Cut, since they use the same decision rule and can be employed on-line. The decision rule parameters were optimized separately for each method and problem so that they find subgoals in the early stages of learning and they eliminate noise properly. The cost ratio ($\lambda_{fa}/\lambda_{miss}$) and the prior ratio ($p(N)/p(T)$) parameters of inequality 2.4 were set to 100 for all experiments like in the Simsek's study [28]. The visitation counts used by RN were reset at the end of each episode. The remaining parameters used by the subgoal identification methods are given

Table3.1: Parameter values used in subgoal identification

Problem	**Parameters used**						
	Method	p	q	t_c	t_{RN}	k	l_n
`2 rooms 1 door`	RN	0.06	0.01	-	2.0	2	7
	L-Cut	0.3	0.01	0.05	-	-	-
	LoBet	0.7	0.07	-	-	-	-
	LoRoots	0.6	0.06	-	-	-	-
`3 rooms`	RN	0.05	0.01	-	2.0	2	7
	L-Cut	0.1	0.01	0.05	-	-	-
	LoBet	0.6	0.06	-	-	-	-
	LoRoots	0.6	0.06	-	-	-	-
`6 rooms`	RN	0.5	0.008	-	2.0	2	7
	L-Cut	0.2	0.01	0.05	-	-	-
	LoBet	0.5	0.05	-	-	-	-
	LoRoots	0.75	0.05	-	-	-	-
`Taxi`	RN	0.712	0.01	-	2.0	2	7
	L-Cut	0.04	0.002	0.05	-	-	-
	LoBet	0.3	0.03	-	-	-	-
	LoRoots	0.24	0.03	-	-	-	-

in Table 3.1. Unfortunately, currently, there is no practical way to find the correct values other than a number of trial-and-error experimentation sessions. Specifically, a heuristic we used to set p and q values is, to examine the outputs of the used subgoal discovery methods and calibrate them according to the subgoals that we manually identified. Other parameters are mostly inherited from [28] where a further analysis can be found.

When a subgoal is found, the agent generates an option to reach that subgoal. The initiation set of the new option contained the states before the first occurrence of the subgoal in each previous episode. *Option lag* (l_o), the number of time steps to look for states to add the initiation set, was 10. Termination probability for each state in the initiation set was set to 0.0, while 1.0 was used for the subgoal.

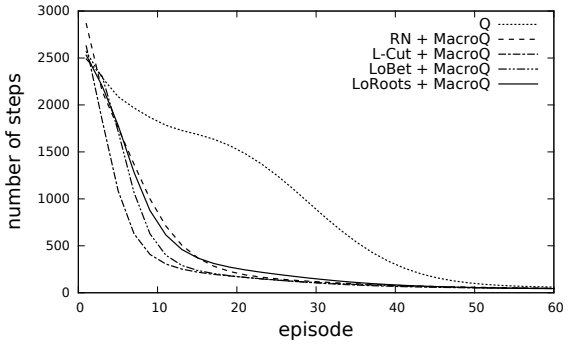
The policy of the option was formed through ER by giving 100 reward upon reaching the subgoal, -10 punishment for leaving the initiation set and -1 punishment for any other transition. For the policy learning part of ER, $\alpha = 0.125$ and $\gamma = 0.9$ were used as the learning parameters. The replay is repeated 10 times for fast convergence.

The agent incorporated Macro-Q learning algorithm, where Q values of an option were updated according to Macro-Q learning while Q values of primitive actions were updated according to regular Q learning. ϵ-greedy was used as option selection strategy, with $\epsilon = 0.1$, and $\alpha = 0.05$ and $\gamma = 0.9$ were set as learning parameters. The same γ value is used in Algorithm 2. All of the results are averaged over 200 experiments.

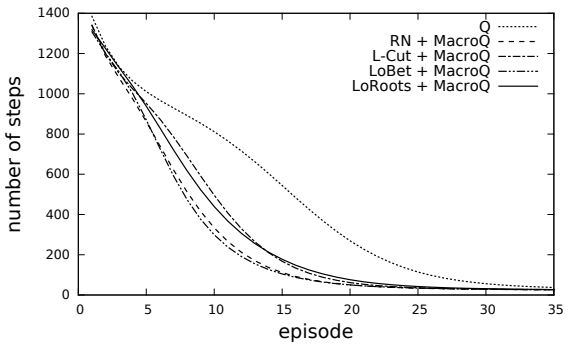
3.2.2 Results and Discussion

Average number of steps to reach the goal state are compared among methods, and the results are sketched in Figure 3.2. Plots are smoothed for visual clarity. All of the subgoal identification methods, including Local Roots, improve the learning speed of the agent by leading it to the goal state earlier and our proposed method matches the performance of the other methods. In general, subgoals discovered by Local Roots seem to be as useful as the ones found by L-Cut, LoBet and RN. We can conclude that the solution quality of Local Roots method is not worse than others on the average.

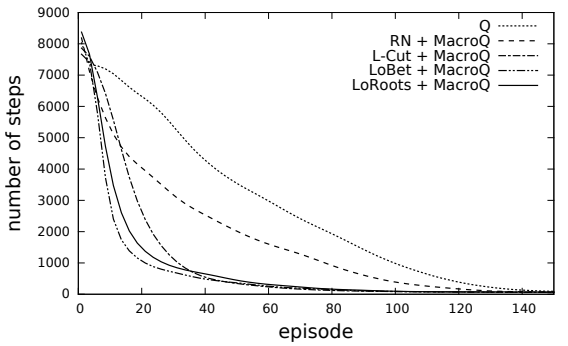
LoBet algorithm has $O(n \cdot m)$ and $O(n \cdot m + n^2 \cdot \log n)$ time complexities on unweighted and weighted graphs, respectively, while L-Cut algorithm requires $O(n^3)$ time when the local interaction graph has n vertices and m edges. On the other hand, RN algorithm is $O(1)$. The time complexity of Local Roots, depends on the maximum depth of a state in the tree it creates. It requires $O((\log_b(n))^2)$ time where b is the average branching factor and n is the number of vertices in the tree.



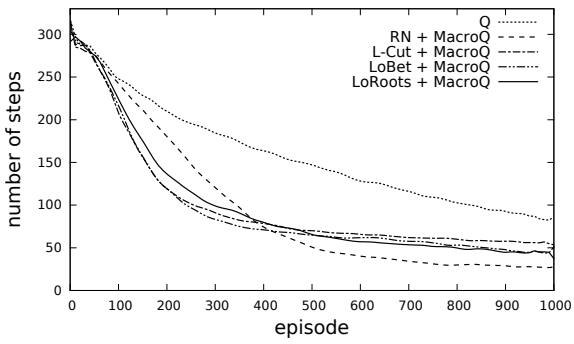
(a) `2 rooms 1 door`



(b) `3 rooms`



(c) `6 rooms`



(d) `Taxi`

Figure 3.2: Average number of steps to goal for each problem

Table3.2: Average CPU time overhead per episode (msec) for subgoal identification

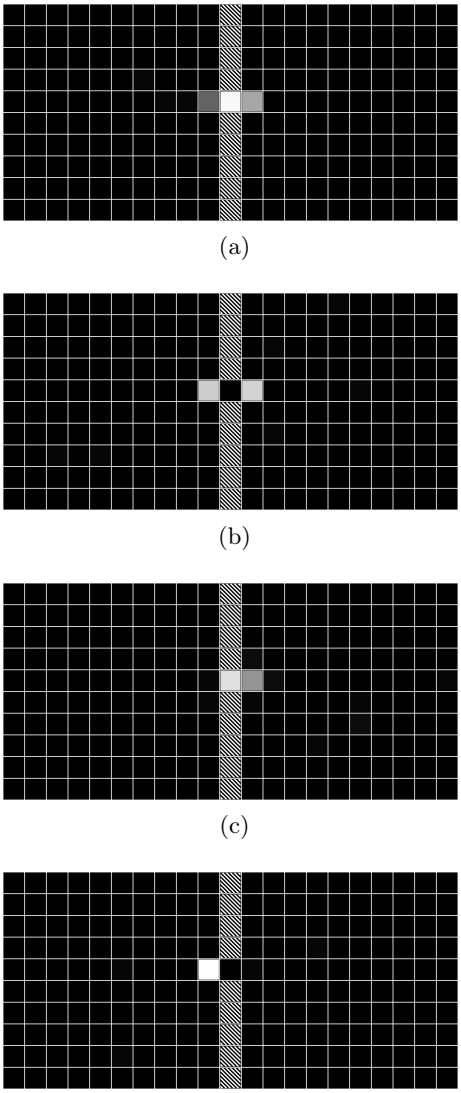
Problem	**RN**	**L-Cut**	**LoBet**	**LoRoots**
`2 rooms 1 door`	0.63	5.18	0.65	**0.45**
`3 rooms`	0.33	1.78	0.32	**0.24**
`6 rooms`	11.00	289.30	7.10	**4.08**
`Taxi`	0.85	1.68	**0.79**	0.81

CPU time measurements, indicating the CPU times used by each subgoal discovery method excluding the underlying Macro-Q algorithm, are given in Table 3.2. The results shows that both LoBet and L-Cut require much more time than Local Roots because of their higher time complexities. The only exception is LoBet for `Taxi` problem, whose time consumption seems to be nearly the same as in Local Roots. On the other hand, although RN is $O(1)$, its time complexity is in fact associated with the number of steps taken, since it is invoked at every time step, unlike the other methods waiting for the episode end. Longer episodes in the earlier stages of an experiment causes RN to generally take more time than Local Roots. Table 3.2 implies that Local Roots algorithm shows a significant advantage in terms of CPU time compared to other methods.

Figure 3.3 shows the subgoals discovered by all four methods for `2 rooms 1 door` problem, marked with brighter color showing high frequency of identification. L-Cut, LoBet and RN finds more than one subgoals including the doorway and states one step near to it. On the other hand, our proposed method finds only the state before the doorway as it is the first merging point of the shortest paths of the states in the left room to the goal state in the right room. This characteristic causes Local Roots to find less number of subgoals than the other algorithms, especially in `2, 3` and `6 rooms` domains. However, as seen Figure 3.4, effectiveness of subgoals discovered are usually higher in Local Roots method compared to the others. We define the *subgoal effectiveness* as

$$(100 \times n_{steps(option)}/n_{steps(episode)})/n_{subgoals}, \tag{3.2}$$

where $n_{steps(option)}$ is the total number steps passed within option sequences, $n_{steps(episode)}$ is the total number of steps taken during the episode, and $n_{subgoals}$ is the number of subgoals identified at the end of an episode.



(a)

(b)

(c)

(d)

Figure 3.3: Subgoals found in `2 rooms 1 door` domain by (a) L-Cut (b) LoBet (c) RN (d) Local Roots.

Subgoal effectiveness can be interpreted as the ability of a subgoal to trigger a useful option. Contribution of some of the additional subgoals found by the other three methods are not as significant as that are found by Local Roots in general.

The average memory usage of Local Roots does not exceed the graph based methods (i.e. LoBet and L-Cut) in general, showed in Figure 3.5. It is worth noting that, memory usage metrics also include ER repositories. Since LoRoots works on a more compact representation of the episode history, namely local

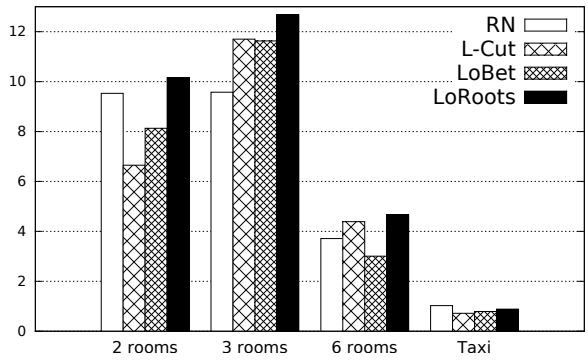


Figure 3.4: Average subgoal effectiveness (average option trace % per subgoal)

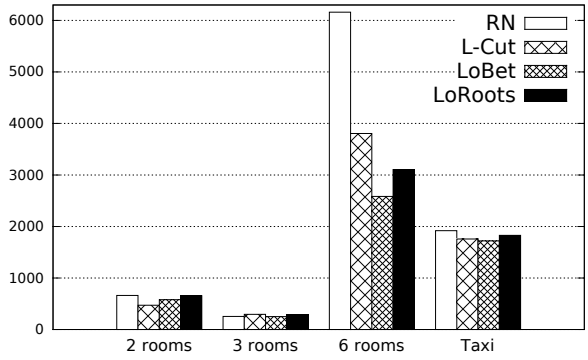


Figure 3.5: Average memory usage per episode in kilobytes

Table3.3: Average number of vertices ($|V|$) and edges ($|E|$) of the local interaction graphs

Problem	**Sizes**	**L-Cut**	**LoBet**	**LoRoots**
`2 rooms 1 door`	$\|V\|$	**36.898**	37.786	39.196
	$\|E\|$	52.194	52.921	**38.196**
`3 rooms`	$\|V\|$	25.272	**24.803**	25.591
	$\|E\|$	37.618	35.577	**24.591**
`6 rooms`	$\|V\|$	85.506	**78.478**	84.098
	$\|E\|$	158.724	131.138	**83.101**
`Taxi`	$\|V\|$	21.915	21.014	**20.819**
	$\|E\|$	36.504	34.078	**19.821**

history tree, it contains less number of edges compared to the other graph based methods. This phenomenon is supported by the results given in Table 3.3 and it leads to the low memory usage.

In addition to the parameters of the inequality 2.4, L-Cut requires one and RN requires four more parameters while LoBet and Local Roots require none. These extra parameters determine the quality of the subgoals found by L-Cut and RN and make them dependent on the structure of the domain. In that sense, Local Roots, like LoBet, is less dependent on the problem characteristics compared to L-Cut and RN. Moreover, as seen in Figure 3.4, Local Roots outperforms LoBet in terms of subgoal quality.

CHAPTER 4

DIRECTED OPTIONS

Identification of a subgoal is of no use unless there is a mechanism that directs the agent to that subgoal. Option framework provides such a formalism by having a definition with an initiation set, I, a local policy, π, and a probability distribution induced by the termination condition, β. With option framework, the agent can employ an option o from a state in I_o, follow the local policy π_o and end it when β_o is satisfied which will cause the agent to reach a possible subgoal.

Most of the studies in the literature focuses on the local policy and the termination condition part of an option, leaving the impact of the initiation set. As I consists of the states that the option can be employed from, it possesses an important role.

Definition of a subgoal inherently depends on being a bottleneck state in the state space since such a subgoal is important on making transitions between the different regions of the problem domain. However, reaching to a subgoal is useful only on the way to a goal state. That is, starting from a state, there is no need to visit a subgoal if there is a better path to a goal state without the subgoal. Reaching to a subgoal is *relatively useful* only for a set of states in the state space.

In the example of `2 rooms 1 door` domain, a bottleneck state is the one at the doorway. However, the agent does not need to visit it when it is located in the right room (the same room with the goal state). It is only useful for the locations

at the left room to reach the bottleneck state. This intuition demonstrates the importance of an initiation set of the option reaching to that bottleneck state.

Common approach in generating an initiation set for the option reaching to an identified subgoal is *option-lag*. Based on the time ordering, it includes the states visited before the subgoal to the initiation set. This method depends on the idea that there is a path from those states to the subgoal and it will be followed again in the next episodes.

However, this is not always the case. It is quite possible that some states visited before the subgoal have a better route to the goal by-passing the subgoal, i.e. have a higher utility in the long term. However, the greedy strategy with option lag heuristic does not take this into account, just putting those states into the initiation set, and depending on the action selection mechanism to perform the necessary distinction in the future. Unfortunately, this unnecessary option transaction grabs learning time until the utility values saturate, which negatively effects the overall performance of learning. Moreover, checking several steps before the terminal does not take state repetitions (or loops) into account. Repetition of states in the observed history usually shortens the initiation set generated, violating an indirect intention of the option lag heuristics to keep track of more or less similar initiation set sizes.

This thesis proposes a better heuristics for building an initiation set for an option. It is again influenced by the history tree approach and it follows the same intuition of ordering states according to their utilities. It focuses on collecting the states that a possible subgoal is relatively useful for reaching.

Algorithm 4 *HISTORY_TREE_HEURISTIC*

Require: a history H

Require: a terminal state s_t

Ensure: initiation set I

$s_r \leftarrow$ state yielding a reward peak and highest reachability through s_t
$parent(s_r) \leftarrow null$, $\mathcal{V}_{s_r} \leftarrow 0$,
for each sub-history $e \in H$ ending with s_r **do**
    $i \leftarrow length(e) - 2$
    while $i \geq 0$ **do** ▷ calculate state values backwards in history
        if $\mathcal{V}_{s_i}$ is undefined or $(r_{i+1} + \gamma * \mathcal{V}_{s_{i+1}}) > \mathcal{V}_{s_i}$ **then**
            $\mathcal{V}_{s_i} \leftarrow r_{i+1} + (\gamma * \mathcal{V}_{s_{i+1}})$
            $parent(s_i) \leftarrow s_{i+1}$
        end if ▷ get rid of loops
        $i \leftarrow i - 1$
    end while
end for
$V \leftarrow \{s_r\}, E \leftarrow \emptyset$
for each state $s \neq s_r$ **do**
    $V \leftarrow V \cup \{s\}$ ▷ vertices of the tree
    $E \leftarrow E \cup (s, parent(s))$ ▷ edges of the tree
end for
$I \leftarrow$ traverse sub-tree of (V, E) rooted by s_t down to depth o_d (excluding s_t)
return I

4.1 History Tree Heuristics

The proposed method [5], given in Algorithm 4, is a heuristic making use of a history tree in order to generate an initiation set. It aims to construct options with goal orientation, meaning that it restricts the initiation set of a new option with the states that are more likely to construct it with an implicit "direction". The method also aims to incorporate a larger set of states as the initiation set compared to the option lag heuristic, due to the elimination of the redundant loops.

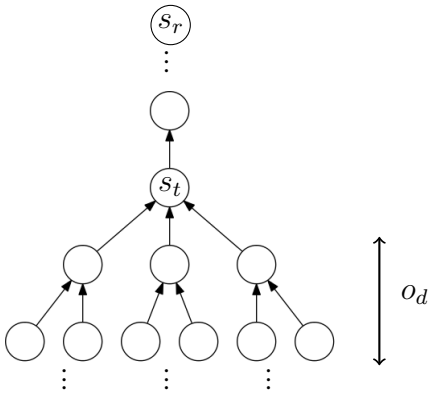


Figure 4.1: History tree heuristic for initiation set generation. The tree is rooted at state s_r and the maximum depth of BFT starting from the terminal state s_t is given as option depth (o_d).

Given a terminal state s_t, similar to Algorithm 2, Algorithm 4 creates a tree having s_r as the root node, which is a assumed to be a possible goal state. A state can be decided as a root if it yields a peak reward within history and is the most frequently reached target from the given terminal state s_t. The aim here, is to determine an accurate direction for the new option.

When the state s_r is selected as the root node, the algorithm traverses all the sub-histories (or episodes) ending with s_r and generates a tree of shortest paths from each visited node to s_r. Each state possesses a value $\mathcal{V}$ determined according to the discounted rewards taken in the path from the state to the root s_r. Using these values, every episode ending with s_r is traversed from the last visited state to the first, setting the parent states for each state as a representative of the best state to go in order to reach the root state s_r (lines 3-12). As a result, a history tree is generated having the vertex set V including all states observed during the traversal and the edge set E having transitions from each state to its parent (lines 13-17).

Figure 4.1 illustrates a portion of a tree generated by the algorithm. Algorithm 4 then employs a traversal (which we implemented as a breadth-first traversal) starting from the terminal state s_t to add every visited state (except s_t) to the initiation set (line 18). With terminal state s_t in depth 0, the maximum depth of the traversal is provided as a parameter called *option depth* (o_d).

Table4.1: Initiation set size settings for both option lag and history tree heuristics

Problem	option lag		history tree	
	o_l	size	o_d	size
`2 rooms 1 door`	15	75.47	10	75.00
`2 rooms 2 doors`	10	45.22	10	44.02
`Virtual Office`	20	69.52	10	66.28
`4 rooms 4 doors` (2 subgoals)	18	63.76	10	64.47
`4 rooms 4 doors` (3 subgoals)	11	50.98	10	49.38
`4 rooms 3 doors`	11	75.53	10	75.00
`Taxi`	20	35.98	10	34.70

4.2 Experiments

4.2.1 Settings

The experiments are designed to compare the performance of history tree heuristic with the greedy option generation strategy using the option lag heuristic. For a fair comparison, the terminal states are provided to the agent beforehand. For the rooms domains, the terminal states are the doorways. For the `Taxi` domain, they are the navigational bottleneck states and the states right after the taxi agent picks up the passenger. The agent generates an option to reach a terminal state (inclusive) when sufficient number of episodes are experienced. This number is called the *episode threshold*, which is set as 10 for all experiments.

Unlike the approach of using only the first occurrence in Chapter 3, here, the option lag heuristic is employed at all occurrences of a subgoal in an episode. There is no distinct trend on the approach for this choice. However, to show the real effect of option lag, all occurrences of a subgoal are examined.

After the formation of the initiation sets is complete, ER is employed by the agent to generate the policy through the states in the initiation set to reach the terminal state. A transition is rewarded 1000 upon reaching the terminal state, and punished with -100 for leaving the initiation set, and given a -1 punishment for any other transition. The learning parameters of ER are $\alpha = 0.125$ and $\gamma = 0.9$. An option is terminated upon reaching the terminal state

with probability 1.0, while it is terminated at any state in the initiation set with probability of 0. The learning process is repeated 10 times to achieve a fast convergence of the Q function.

The parameters *option lag* and *option depth* are set in such a way that the resulting options have initiation sets with approximately the same number of states on the average for both approaches (Table 4.1). Macro-Q learning is used with learning parameters $\alpha = 0.05$, $\gamma = 0.9$ and $\epsilon = 0.1$. Both methods are compared against the regular Q-Learning using the same learning parameters without options. All test results are averaged over 200 experiments.

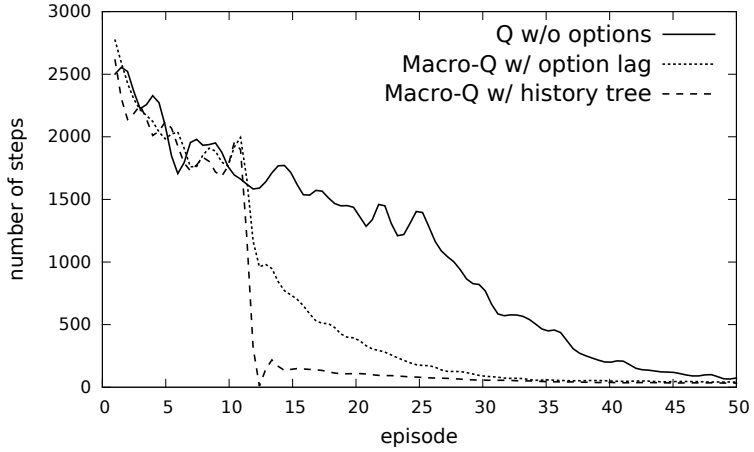
4.2.2 Results and Discussion

It is clear from the results given in Figures 4.2 and 4.3 that, under fair conditions, Macro-Q learning with options generated using the history tree heuristic for initiation set clearly outperforms the one with option lag heuristic, in terms of average number of steps to reach the goal state. For every problem in the set, history tree provides an advantage to the Macro-Q learning algorithm, especially at the early stages of the learning process.

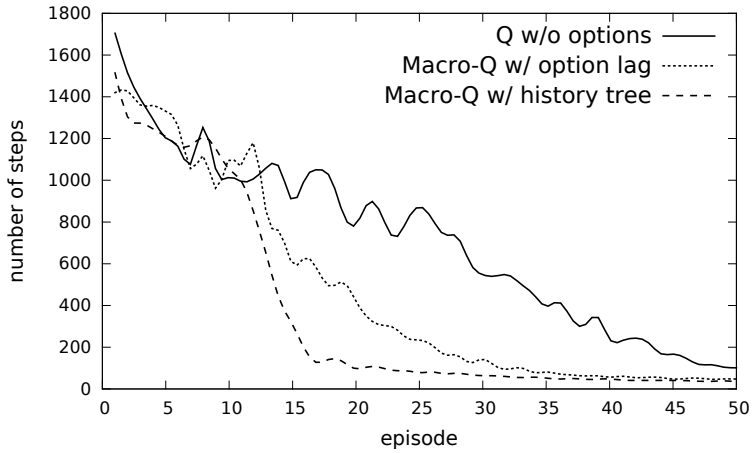
In order to further analyze the quality of generated options, two metrics are introduced: One of them is the *option overlap* which is defined as

$$100 \times \frac{|\cap_{o'} I_{o'}|}{|\cup_{o'} I_{o'}|}, \tag{4.1}$$

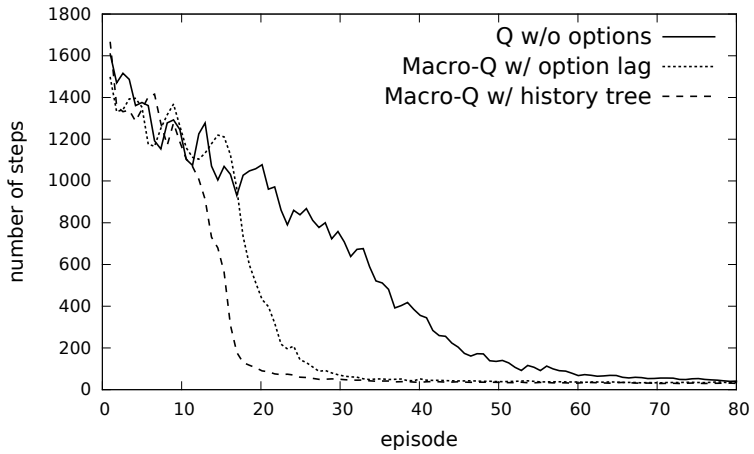
where $|\cap_{o'} I_{o'}|$ represents the cardinality of the intersection of initiation sets of all options and $|\cup_{o'} I_{o'}|$ represents the cardinality of the union of initiation sets of all options. Higher values of this metric is an indication of potentially unnecessary repetitions among options.



(a) `2 rooms 1 door`

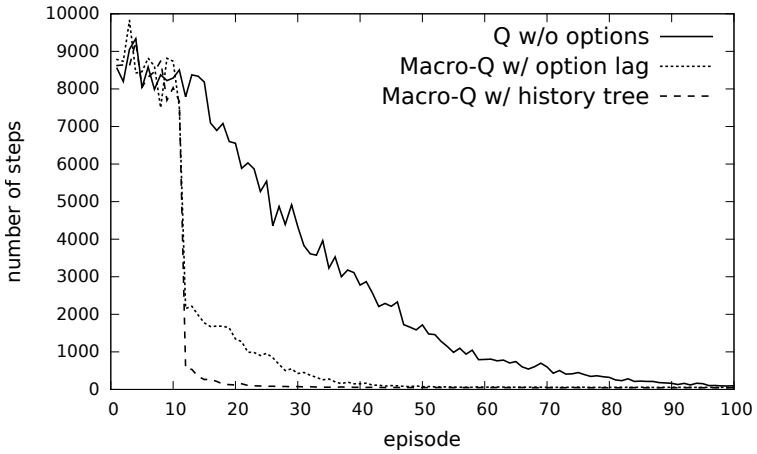


(b) `2 rooms 2 doors`

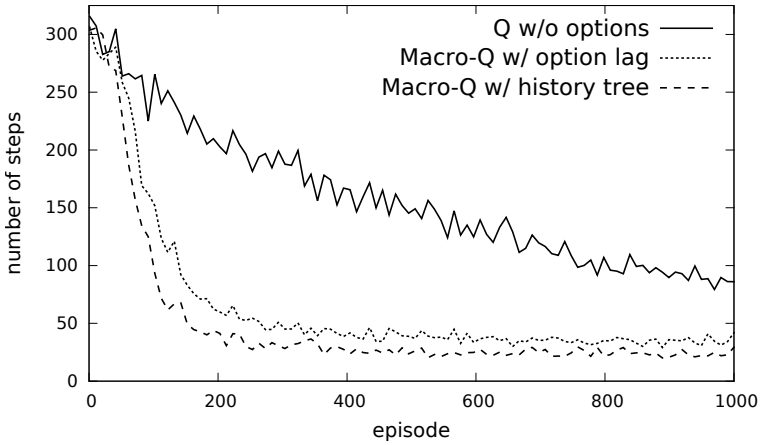


(c) `Virtual Office`

Figure 4.2: Average number of steps to goal for `2 rooms 1 door`, `2 rooms 2 door` and `Virtual Office` problems



(a) `4 rooms 3 doors`



(b) `Taxi`

Figure 4.3: Average number of steps to goal for `4 rooms 3 doors` and `Taxi` problems

Table4.2: Results in option quality and time consumption for both of the initiation set heuristics

Problem	**Option overlap (%)**		**Average time overhead (msec)**	
	option lag	**history tree**	**option lag**	**history tree**
`2 rooms 1 door`	0.00	0.00	15.72	**15.58**
`2 rooms 2 doors`	13.99	**0.38**	21.82	**20.69**
`Virtual Office`	9.43	**1.15**	11.83	**10.39**
`4 rooms 4 doors` (2 subgoals)	0.00	0.00	**22.89**	23.75
`4 rooms 4 doors` (3 subgoals)	8.08	**0.00**	**33.83**	38.18
`4 rooms 3 doors`	12.85	**0.00**	**103.79**	105.31
`Taxi`	52.59	**51.24**	25.53	**23.31**

The other metric is the *option preferability* indicating how desirable it is to employ the option at the states in its initiation set. It is defined as

$$P(o) = \frac{\sum_{s \in I_o} p(s, o)}{|I_o|}, \tag{4.2}$$

and $p(s, o)$ is defined as

$$p(s, o) = \begin{cases} 1, & \max_{o'} Q(s, o') = Q(s, o), \\ 0, & \text{otherwise} \end{cases} \tag{4.3}$$

where o is the option at hand, s represents a state in I_o, o' is any option that can be employed from s and I_o is the initiation set of the option o.

Table 4.2 lists the option overlap percentages and average CPU time used by the heuristic. In terms of option overlap, history tree heuristics provides a significant improvement for the majority of problems. This means that history tree heuristics can generate an initiation set for an option in such a way that redundant repetitions are reduced. Moreover, the method achieves this using more or less the same CPU time.

Options generated via the history tree method are preferred more than the ones generated via the option lag method, on the average, as can be seen in Figure 4.4.

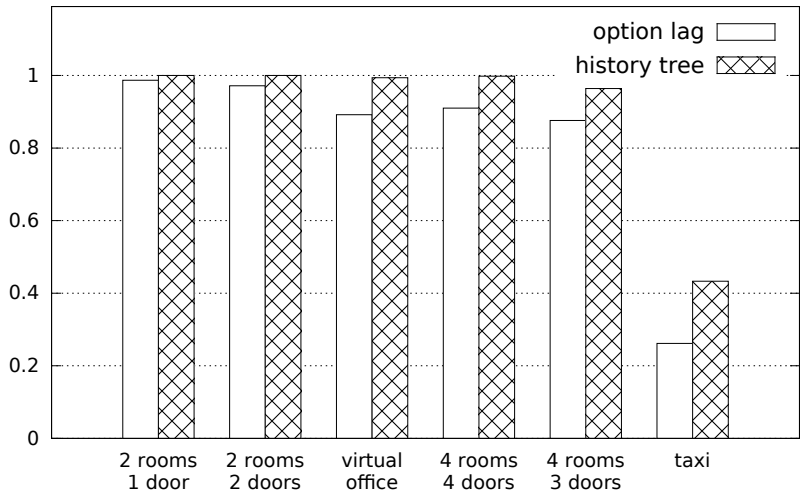
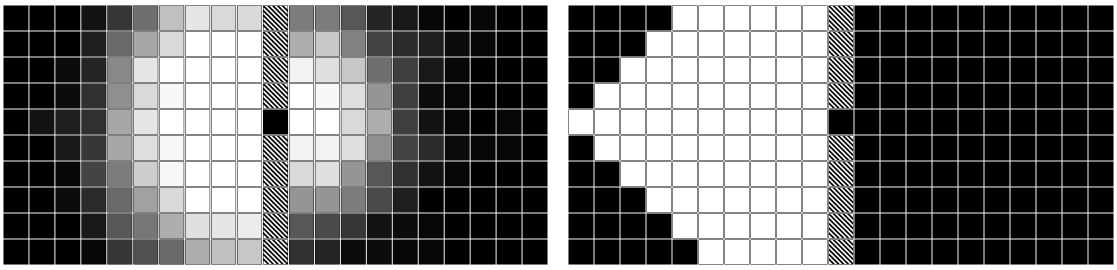
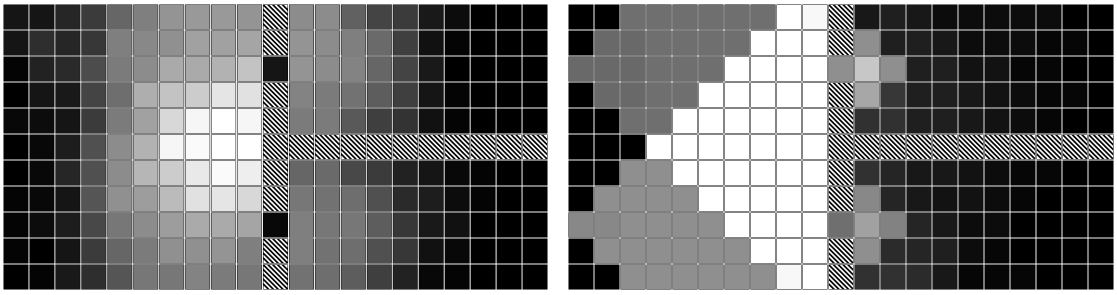


Figure 4.4: Average option preferabilities



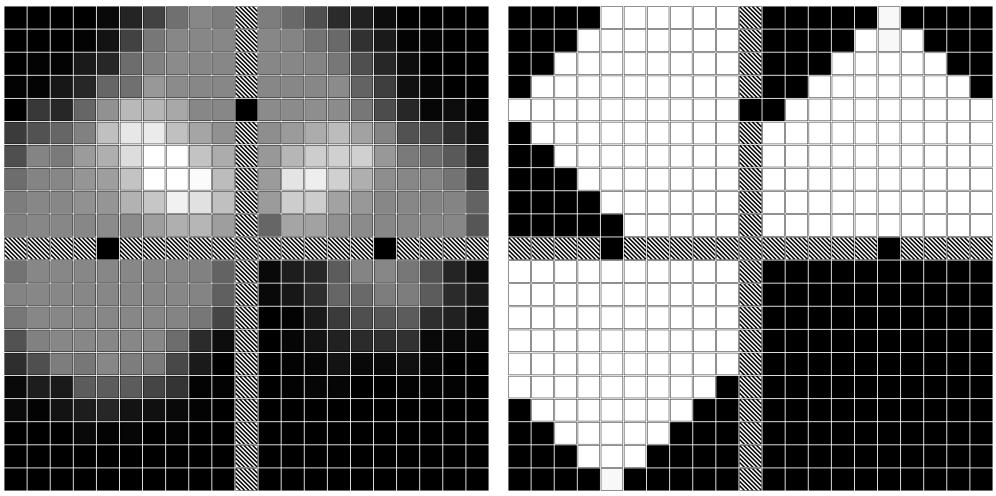
(a) option lag (b) history tree

Figure 4.5: The number of occurrences of each state within an initiation set for `2 rooms 1 door` problem



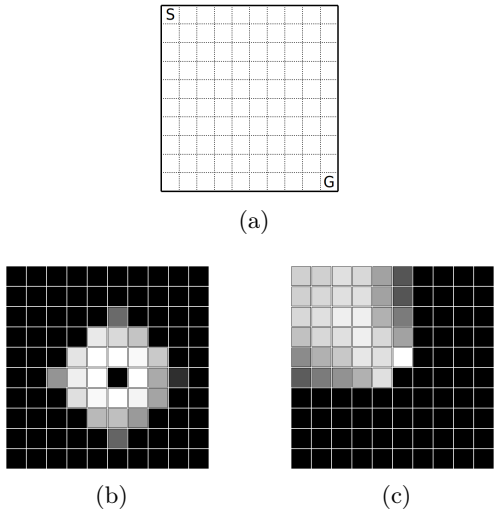
(a) option lag (b) history tree

Figure 4.6: The number of occurrences of each state within an initiation set for `Virtual Office` problem



(a) option lag (b) history tree

Figure 4.7: The number of occurrences of each state within an initiation set for `4 rooms 3 doors` problem



(a) (b) (c)

Figure 4.8: The number of occurrences sketch for a simple domain (a) 10x10 grid domain definition (b) number of occurrences with option lag (b) number of occurrences with history tree

The relative difference seems to be more significant with increasing problem size and number of terminal states, supporting our intuition that history tree based initiation sets drive the action selection process through macros by encouraging generation of goal oriented options.

In order to have a deeper insight on how our method realizes this improvement, Figures 4.5, 4.6 and 4.7 visualize the number of occurrences of each state within an initiation set on the average. Brighter color means that the state has been selected more as an initiation set element, while darker means less.

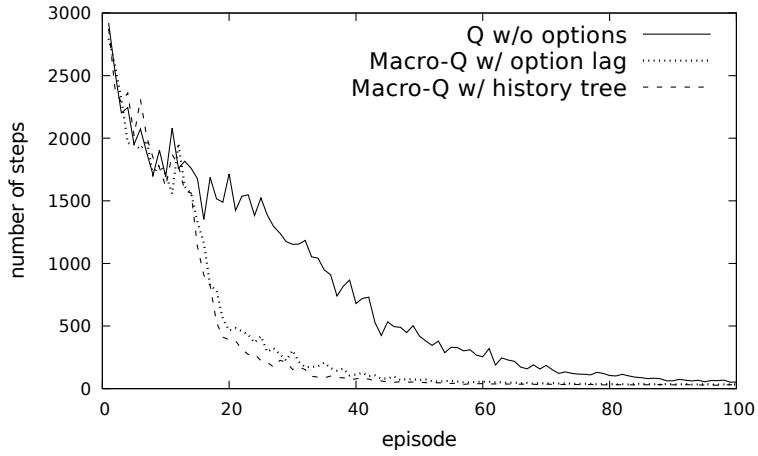
But first, it might be useful to use a simple 10x10 grid world domain (Figure 4.8a) with the same action, transition and reward semantics with the rooms domains, but lacking any inherent bottleneck states. Start state is S and the goal state is G. This means that if a terminal state is randomly chosen in this problem, it is unlikely to improve learning performance, since the problem can not be naturally partitioned. However, still, the position of the goal state may imply an option orientation with respect to a randomly selected terminal state. A state is selected somewhere in the middle of the grid world, as an artificial terminal (or subgoal) state, and the both option lag and history tree methods are invoked to generate initiation sets, and then options are generated. The

initiation set occurrence frequencies for each state are illustrated in Figures 4.8b and 4.8c. Both methods ended up with having approximately 17 states in their initiation sets. As depicted in the figures, the option lag approach generates an option aiming to reach the given terminal state from its surrounding states. On the other hand, history tree approach includes only the states between the start state and the terminal state. Since the goal state is located at the south-east corner, the states selected as initiation set elements are clearly shapes the option to have an implicit "direction" towards the goal state. The eliminated states have shorter paths to reach the goal state without any need to visit the terminal state.

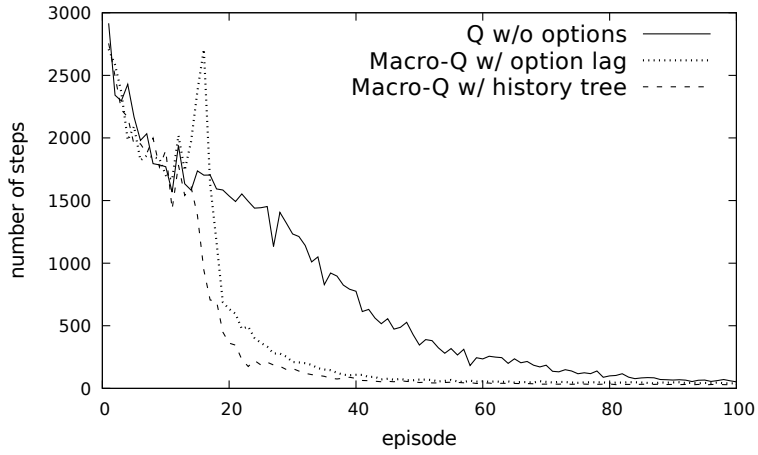
Similarly, Figure 4.5 shows that, although both methods have nearly the same amount of states in the initiation sets (Table 4.1), since the agent can circle in loops leading back to the doorway, the option lag mechanism tends to choose many of the states on the right side of the doorway and add them to the initiation set. However, the initiation set generated by history tree heuristic consists only of the states of the room on the left. This differentiation prevents the agent to explore an option among the states at the right side and results in improved learning performance (Figure 4.2a).

The `2 rooms 2 doors` domain (Figure 2.1b) offers two alternative ways for the agent to reach the single goal state. In the generated options, the history tree heuristic adds a state in the left room to only one of the initiation sets of the options leading to the doorways, resulting in a very low option overlapping compared to the one of the option lag heuristic (Table 4.2). This is simply because a state in the western room can belong to only one of the sub-trees below a door state within the generated tree. This allows the history tree method to skip the unnecessary exploration on an equally useful option and improves the performance of it in the learning speed (Figure 4.2b).

When multiple goal states are introduced, like in the `Virtual Office` domain, history tree heuristic still outperforms the current option lag heuristic in terms of learning speed (Figure 4.2c). Since an episode including a bottleneck state usually ends with the goal state in the corresponding room, history tree heuristic



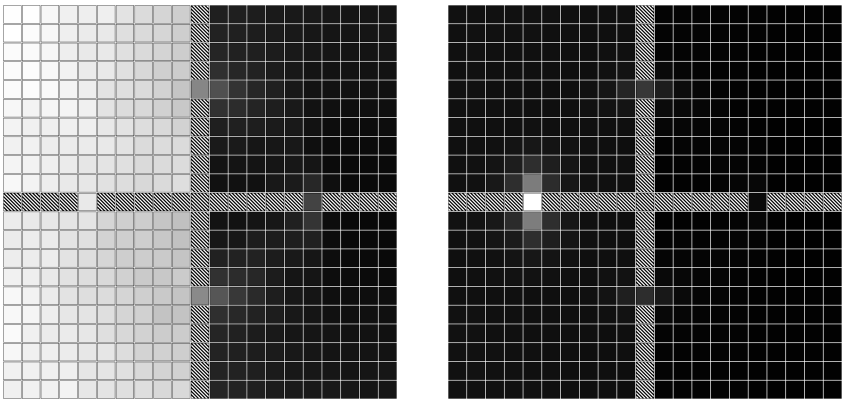
(a) `4 rooms 4 doors` (2 subgoals)



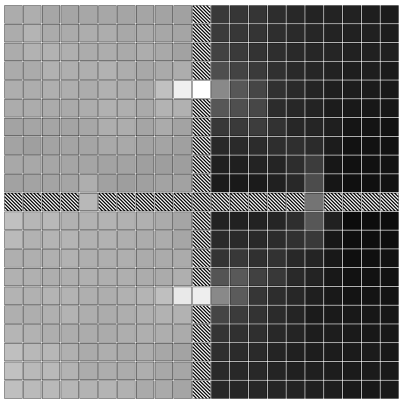
(b) `4 rooms 4 doors` (3 subgoals)

Figure 4.9: Average number of steps to goal for `4 rooms 4 doors` domain with (a) 2 subgoals and (b) 3 subgoals

selects that goal state as the root, resulting in generating an option for entering that room. This selection mechanism prevents our method to create an option leaving a room with a goal state. The occurrence frequencies of states in the initiation sets are sketched in Figure 4.6, indicating that history tree heuristic tends to derive options more eager in leading the agent to the goal states in the eastern rooms. It is important to note that the initiation set states at the right rooms in `Virtual Office` are actually included to the initiation sets of the options reaching to the other doorway, not to the one close to them.



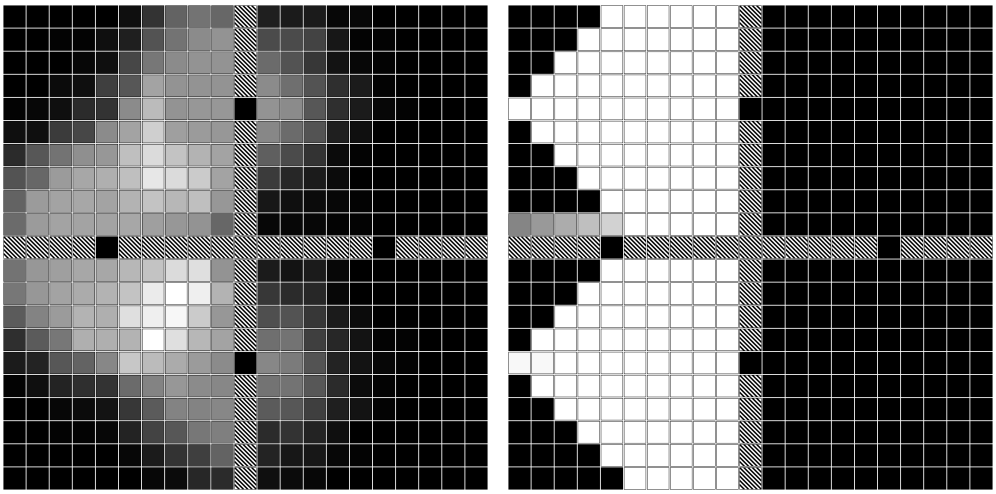
(a) Q without options (b) option lag



(c) history tree

Figure 4.10: Visitation frequencies of each state for `4 rooms 4 doors` problem with 3 subgoals provided. Brighter color represents higher visitation.

The original version of the 4 rooms domain with 4 doors, possesses an important feature as the doorway between the left rooms is a bottleneck state and can be considered as a subgoal while it is not that useful to reach it. As the goal state is the doorway between the right rooms, an optimal policy may lead the agent to the doorways connecting the left and the right rooms. When only these 2 subgoals are provided to the agent, history tree heuristic results in better performance (Figure 4.9a), even though the difference is close. However, both methods lead to no option overlap (Table 4.2). On the other hand, with the doorway between the left rooms included, the agent spends time to learn that the option reaching to it, is no use with option lag heuristic as it can be seen from Figure 4.9b. Generated by option lag heuristic, such an option acts like a pitfall, making the agent visit the useless doorway more frequently. From Figures



(a) option lag (b) history tree

Figure 4.11: The number of occurrences of each state within an initiation set for `4 rooms 4 doors` problem with 3 subgoals provided

4.10a, 4.10b and 4.10c with state visitation frequencies, it can be seen that the agent spends more time in the left rooms with no options, visits the left doorway often with options with option lag heuristic and it is led to the goal state more sooner with the options generated by history tree heuristic. Moreover, addition of the left doorway causes option overlap for option lag heuristic while history tree method only includes a small set of states to the initiation set of the option reaching to it (4.11b).

For the modified version of 4 rooms domain, without the doorway between the southern rooms (Figure 2.1g), an option to reach the door connecting the left rooms is useful *only if* it initiates from the states in the south-west room, which is in fact a distractor leading nowhere. To be included in the initiation set, history tree method selects only those states which directs the option towards the room exit. The difference between the selection strategies can be clearly seen in Figure 4.7.

Finally, in `Taxi` domain, the destination of the passenger inherently suggests a direction towards the goal state. Since history tree heuristic helps generating goal oriented options, its performance is much better than the option lag approach, as can be seen in Figure 4.3b. On the other hand, the subgoals in the

`Taxi` problem are successive, meaning that, starting from an initial state, more than one subgoal exists on the way to goal. This causes a higher option overlapping compared to the other problems for both of the methods. However, history tree heuristic still outperforms the option lag heuristic in terms of overlapping options.

In general, the goal oriented options generated via the history tree heuristic seem to be more preferable than the ones created through the option lag mechanism on their corresponding initiation sets, outperforming for all of the problems in the problem set. This suggests that history tree heuristic can be a better alternative than the conventional greedy initiation set generation approach.

CHAPTER 5

CONCLUSION AND FUTURE WORK

This thesis proposes a tree based automatic subgoal discovery method called Local Roots that helps the learning agent to identify important states on the way to the goal state in the early stages of learning. Local Roots method can be employed upon reward peaks, which are usually goal states. Using the options framework, the learning agent can devise abstractions to reach the identified subgoals. The method utilizes a tree based metric to locally identify the junction points of the shortcuts directed from each visited state towards the goal state.

In terms of learning speed, Local Roots outperforms the regular Q-Learning for all problem domains experimented. It also keeps up with the performance of the other local methods on the average, showing that subgoals identified by Local Roots are no worse than the ones found by other algorithms. Compared to other graph based methods tested, Local Roots has lower time complexity. On the other hand, when average CPU times per episode are compared, Local Roots outperforms all other methods on the average, including Relative Novelty which has the lowest theoretical time complexity, but should be invoked at every time step. Local Roots is also shown to identify less number of subgoals with higher effectiveness in general. Moreover, it requires no additional parameters unlike Relative Novelty and Local Cuts.

A possible future work on Local Roots can focus on finding an alternative way of discriminating noise from local subgoal information with less domain specific parameters. Also, automatic detection of these parameters can be an important improvement for all the on-line subgoal discovery methods presented here.

Furthermore, a goal oriented option generation method utilizing a history tree heuristic is proposed. It restricts the initiation set of an option to the states from which employing the option would be useful. With this restriction, an option possesses a direction towards a possible goal state. The new approach increases the option quality by eliminating unnecessary states that do not need to visit the terminal state to reach a goal state. Without requiring additional computation time, history tree heuristic increases the learning performance of the agent.

For the initiation set heuristic, a future work can be on the improvement of the selection of the root state which determines the direction of the generated options. It can be reasonable to choose a root state that an identified subgoal is most useful to reach, or choose multiple roots causing multiple options aiming to go to the same terminal state through different directions. Moreover, the generated tree can also directly be used for construction of the option policy. The only missing parts are the actions that lead the agent through the paths in the tree. The artificial rewards used in experience replay mechanism can be modified so that best target for each state corresponds to the parent of that state in the tree.

REFERENCES

[1] R. E. Bellman. *Dynamic Programming*. Princeton University Press, Princeton, NJ, 1957.

[2] S. J. Bradtke and M. O. Duff. Reinforcement learning methods for continuous-time markov decision problems. In G. Tesauro, D. Touretzky, and T. Leen, editors, *Advances In Neural Information Processing Systems*, volume 7 of *NIPS '94*, pages 393–400, Cambridge, MA, 1994. MIT Press.

[3] U. Brandes. A faster algorithm for betweenness centrality*. *Journal of mathematical sociology*, 25(2):163–177, 2001.

[4] F. Chen, S. Chen, Y. Gao, and Z. Ma. Connect-based subgoal discovery for options in hierarchical reinforcement learning. In J. Lei, J. Yao, and Q. Zhang, editors, *Proceedings of the Third International Conference on Natural Computation*, volume 4 of *ICNC '07*, pages 698–702. IEEE, 2007.

[5] A. Demir, E. Çilden, and F. Polat. A history tree heuristic to generate better initiation sets for options in reinforcement learning. European Conference of Artificial Intelligence, ECAI 2016, The Hague, The Netherlands, August 29 - September 2, 2016.

[6] A. Demir, E. Çilden, and F. Polat. Local roots: A tree-based subgoal discovery method to accelerate reinforcement learning. European Conference on Machine Learning and Principles and Practice of Knowledge Discovery, ECML-PKDD 2016, Riva del Garda, Italy, September 19 - 23, 2016.

[7] T. G. Dietterich. Hierarchical reinforcement learning with the MAXQ value function decomposition. *J. Artif. Intell. Res.*, 13(1):227–303, 2000.

[8] L. T. Dung, T. Komeda, and M. Takagi. Solving POMDPs with Automatic Discovery of Subgoals. In M. J. Er and Y. Zhou, editors, *Theory and Novel Applications of Machine Learning*, pages 229–238. InTech, 2009.

[9] M. Ghafoorian, N. Taghizadeh, and H. Beigy. Automatic abstraction in reinforcement learning using ant system algorithm. In *Lifelong Machine Learning: Papers from the 2013 AAAI Spring Symposium*. AAAI, 2013.

[10] S. Girgin, F. Polat, and R. Alhajj. Learning by automatic option discovery from conditionally terminating sequences. ECAI 2006, pages 494–498, Amsterdam, The Netherlands, 2006. IOS Press.

[11] S. Girgin, F. Polat, and R. Alhajj. Improving reinforcement learning by using sequence trees. *Mach. Learn.*, 81(3):283–331, 2010.

[12] S. Goel and M. Huber. Subgoal discovery for hierarchical reinforcement learning using learned policies. In I. Russell and S. M. Haller, editors, *Proceedings of the 16th International FLAIRS Conference*, pages 346–350. AAAI Press, 2003.

[13] N. K. Jong, T. Hester, and P. Stone. The utility of temporal abstraction in reinforcement learning. In *Proceedings of the 7th International Conference on Autonomous Agents and Multiagent Systems*, volume 1 of *AAMAS '08*, pages 299–306, Estoril, Portugal, May 2008. International Foundation for Autonomous Agents and Multiagent Systems.

[14] A. Jonsson and A. Barto. Causal graph based decomposition of factored MDPs. *Journal of Machine Learning Research*, 7:2259–2301, Nov. 2006.

[15] S. J. Kazemitabar and H. Beigy. Automatic discovery of subgoals in reinforcement learning using strongly connected components. In M. Köppen, N. Kasabov, and G. Coghill, editors, *Advances in Neuro-Information Processing: ICONIP '08 Revised Selected Papers, Part I*, volume 5506 of *LNCS*, pages 829–834. Springer-Verlag, 2008.

[16] G. Kheradmandian and M. Rahmati. Automatic abstraction in reinforcement learning using data mining techniques. *Robot. Auton. Syst.*, 57(11):1119–1128, 2009.

[17] G. Konidaris and A. S. Barreto. Skill discovery in continuous reinforcement learning domains using skill chaining. In *Advances in Neural Information Processing Systems*, pages 1015–1023, 2009.

[18] L.-J. Lin. Self-improving reactive agents based on reinforcement learning, planning and teaching. *Mach. Learn.*, 8(3):293–321, 1992.

[19] S. Mannor, I. Menache, A. Hoze, and U. Klein. Dynamic abstraction in reinforcement learning via clustering. In *Proceedings of the Twenty-first International Conference on Machine Learning*, ICML '04, pages 71–78. ACM, 2004.

[20] A. McGovern. acQuire-macros: An algorithm for automatically learning macro-actions. In *The Neural Information Processing Systems Conference Workshop on Abstraction and Hierarchy in Reinforcement Learning*, NIPS '98, 1998.

[21] A. McGovern and A. G. Barto. Automatic discovery of subgoals in reinforcement learning using diverse density. In *Proceedings of the Eighteenth International Conference on Machine Learning*, ICML '01, pages 361–368. Morgan Kaufmann Publishers Inc., 2001.

[22] A. McGovern, R. S. Sutton, and A. H. Fagg. Roles of macro-actions in accelerating reinforcement learning. *Proceedings of the 1997 Grace Hopper Celebration of Women in Computing*, pages 13–18, 1997.

[23] E. A. McGovern. *Autonomous Discovery of Temporal Abstractions from Interaction with an Environment.* Ph.D. thesis, University of Massachusetts Amherst, Amherst, MA, USA, 2002.

[24] I. Menache, S. Mannor, and N. Shimkin. Q-Cut—Dynamic discovery of sub-goals in reinforcement learning. In T. Elomaa, H. Mannila, and H. Toivonen, editors, *Machine Learning: ECML 2002: 13th European Conference on Machine Learning Proceedings*, volume 2430 of *LNCS*, pages 295–306. Springer Berlin Heidelberg, 2002.

[25] P. Moradi, M. E. Shiri, and N. Entezari. Automatic skill acquisition in reinforcement learning agents using connection bridge centrality. In T.-H. K. et al., editor, *Communication and Networking: International Conference, FGCN 2010 Proceedings, Part II*, volume 120 of *CCIS*, pages 51–62. Springer Berlin Heidelberg, 2010.

[26] R. Parr and S. Russell. Reinforcement learning with hierarchies of machines. In M. Jordan, M. Kearns, and S. Solla, editors, *Advances in Neural Information Processing Systems*, volume 10 of *NIPS '97*, pages 1043–1049. MIT Press, 1998.

[27] J. Shi and J. Malik. Normalized cuts and image segmentation. *IEEE Trans. Pattern Anal. Mach. Intell.*, 22(8):888–905, 2000.

[28] O. Simsek. *Behavioral Building Blocks for Autonomous Agents: Description, Identification, and Learning.* Ph.D. thesis, University of Massachusetts Amherst, 2008.

[29] O. Simsek and A. G. Barto. Using relative novelty to identify useful temporal abstractions in reinforcement learning. In *Proceedings of the Twenty-first International Conference on Machine Learning*, ICML '04, pages 95–102. ACM, 2004.

[30] O. Simsek, A. P. Wolfe, and A. G. Barto. Identifying useful subgoals in reinforcement learning by local graph partitioning. In *Proceedings of the Twenty-second International Conference on Machine Learning*, ICML '05, pages 816–823. ACM, 2005.

[31] M. Stolle and D. Precup. Learning options in reinforcement learning. In S. Koenig and R. C. Holte, editors, *Proceedings of the 5th International Symposium on Abstraction, Reformulation, and Approximation*, volume 2371 of *LNCS*, pages 212–223. Springer Berlin Heidelberg, 2002.

[32] R. S. Sutton. Learning to predict by the methods of temporal differences. *Mach. Learn.*, 3(1):9–44, 1988.

[33] R. S. Sutton and A. G. Barto. *Reinforcement Learning: An Introduction.* Adaptive Computation and Machine Learning. MIT Press, 1998.

[34] R. S. Sutton, D. Precup, and S. Singh. Between MDPs and semi-MDPs: a framework for temporal abstraction in reinforcement learning. *Artif. Intell.*, 112(1-2):181–211, 1999.

[35] N. Taghizadeh and H. Beigy. A novel graphical approach to automatic abstraction in reinforcement learning. *Robot. Auton. Syst.*, 61(8):821–835, 2013.

[36] C. Watkins. *Learning from Delayed Rewards.* Ph.D. thesis, Cambridge University, 1989.