Hierarchical Reinforcement Learning for Spatio-temporal Planning

Shambhuraj Sawant

Master of Science Thesis

TU Delft Delft University of Technology

Hierarchical Reinforcement Learning for Spatio-temporal Planning

Master of Science Thesis

For the degree of Master of Science in Mechanical Engineering at Delft University of Technology

Shambhuraj Sawant

September 23, 2018

Faculty of Mechanical, Maritime and Materials Engineering (3mE) · Delft University of Technology

Delft University of Technology
Department of

The undersigned hereby certify that they have read and recommend to the Faculty of Mechanical, Maritime and Materials Engineering (3mE) for acceptance a thesis entitled

Hierarchical Reinforcement Learning for Spatio-temporal Planning

by

Shambhuraj Sawant

in partial fulfillment of the requirements for the degree of

Master of Science Mechanical Engineering

Dated: September 23, 2018

Supervisor(s):

dr. ir. Matthijs Spaan

Dr. Neil Yorke-Smith

Prof. dr. ir. Martijn Wisse

dr. ir. Joris Sijs

Abstract

Reinforcement learning (RL) is an area of Machine Learning (ML) concerned with learning how a software-defined agent should act in an environment to maximize the rewards. Similar to many ML methods, RL suffers from the *curse of dimensionality*, the exponential increase in solution space with the increase in problem dimensions. Learning the hierarchy present in underlying problems, formulated using the Markov Decision Processes (MDPs) framework, may exploit inherent structure in the environment. Using the hierarchical structure, an MDP can be divided into several simpler semi-MDPs (SMDPs) having temporally extended actions. The solutions of smaller SMDPs can then be re-combined to form a solution for the original MDP. The methods for Hierarchical Reinforcement Learning (HRL) explore ways to break down the original problem into SMDPs while providing several opportunities for state and temporal abstractions. A novel algorithm for learning this hierarchical structure of a discrete-state goal-oriented Factored-MDP (FMDP) is proposed in the thesis work taking into account the causal structure of the problem domain with the use of Dynamic Bayesian Network (DBN) model. The proposed method autonomously learns the state and temporal abstractions in the problem domain and constructs a hierarchy of SMDPs using them. Such a decomposition results in decreasing the problem state dimensions to be considered for solving each SMDP and, hence, reducing the computational complexity induced due to increased dimensionality.

Contents

List of Figures

Acknowledgements

I would like to acknowledge many individuals without whom this thesis would not have materialized. I would like to first express my sincere gratitude to my supervisors, dr. ir. Matthijs Spaan and Dr. Neil Yorke-Smith, for trusting me with the freedom to explore various ideas and showed dedicated interest in my progress. I would like to thank them for their patience, understanding, and supervision. I would also like to thank Prof. dr. ir. Martijn Wisse and dr. ir. Joris Sijs for their support, encouragement, and supervision.

My master's journey in Delft, for the most part, has been shaped by my friends. I would like to thank Janani Venkatasubramanian without whose support and encouragement my work definitely would not have transpired. I will always cherish the discussions we had that helped me in getting to the core of my thesis work. I would like to thank Anant Semwal and Abbas Jhabuawala for the fun times we had together. I would also like to thank Aishwarya Karthikeyan, Sparsh Sharma, Apourva Parthasarathy and Malvika Dixit for helping me get through this journey especially in the last few months.

Finally, and most importantly, I would like to thank my parents and my sister for their consistent support and tireless encouragement throughout my entire academic pursuit. They have stood by me in my lows and highs and have always tried to inspire me to do better.

Delft, University of Technology
September 23, 2018

Shambhuraj Sawant

"Somewhere, something incredible is waiting to be known."

— *Carl Sagan*

Chapter 1

Introduction

The field of Artificial Intelligence is about constructing agents which can act rationally in any given context [2]. Some of the key challenges for AI agents are about learning and representing knowledge of the problem domain at multiple levels of abstractions. In planning problems, a software-defined agent takes actions to maximize it's own notion of reward. An agent is thought to act rationally when it tries to maximize a performance measure given a set of observations and, hence, it has to take decisions to guide itself along the highest reward path. For such sequential decision-making problems, the paradigm of Reinforcement Learning (RL) deals with learning policies in a problem domain using limited feedback to reinforce the optimal behavior. However, RL suffers from the *curse of dimensionality*, the exponential increase in the size of value space required for finding an optimal solution with an increase in problem dimensions.

Most real-world planning problems involve a large set of states and their complex interactions to define dynamics needed to be considered for planning, resulting in an impasse. Fortunately, the real world is structured with many underlying constraints resulting in independent states and systems. Such sets of states can be decoupled and solved separately than through sheer enumeration. In such systems, a hierarchy represents the inter-connected sub-systems which may have their own sub-sub-systems, with everything together constituting the original problem. Such a hierarchical system can be understood as a nearly decomposable system, as its intra-component relations are generally stronger than inter-component relations [3], constituting the property of near-decomposability. Using the near-decomposability property, a *divide-and-conquer* strategy can be implemented on any complex problem to decompose it into smaller sub-problems and so on. These sub-problems may be easily solved due to their smaller problem dimensions and solutions can then be recombined to compute policies for the original problem. Such a decomposition may result in reduced time and space complexity in both learning and execution phases of the policy.

Hierarchical Reinforcement Learning (HRL) is one of the emerging areas of research in RL which takes advantage of this *divide-and-conquer* strategy, by using the inherent hierarchy or by constructing an *artificial* hierarchy, to learn policies effectively using the rewards given by the problem environment. It provides opportunities for state and temporal abstractions for scaling up RL. In HRL, low-level policies invoke *primitive* actions, actions extending for one time-step available in the regular RL setting, which solve only a few parts of the overall task, while, higher-level policies solve the overall task using only higher level observations and abstract actions. Such a method causes the search space for a solution at each level in the hierarchy to be greatly reduced as only a small subset of state variables needs to be considered for decision making. Markov Decision Processes (MDPs) provide a mathematical framework for modeling

such a decision making in scenarios with stochastic outcomes and are solved using reinforcement learning or dynamic programming techniques. When solving a planning task formulated as an MDP using HRL methods, the agent effectively divides the problem at hand into smaller sub-tasks which are then solved as semi-MDPs (SMDPs), thus avoiding the problem of increased dimensionality. Furthermore, such a decomposition provides a possibility of re-using the learned hierarchies and low-level policies in various related problems, thus allowing knowledge transfer.

HRL methods, however, require the agent to know spatial and temporal abstractions in the problem domain and a value decomposition scheme for solving the credit assignment problem that follows with these abstractions. Hence, building on the current HRL methods, the thesis aims to propose a novel algorithm for learning the inherent structure in the problem domain using the causal information of the underlying MDP and exploiting it through planning. The following sections elaborate on the motivation for this thesis work and discuss ideas that stem from related works. This is followed by the research question and an overview of contributions. Finally, the structure of the thesis is provided.

1-1 Motivation

The main challenges associated with reinforcement learning methods are about scalability and sample requirements in different problem domains arising due to the *curse of dimensionality*. In RL literature, model-based and deep RL techniques are gaining a lot of attention for addressing these challenges. However, HRL methods also address these challenges by introducing abstractions to the problem domain. Hence, it is advantageous to use both spatial and temporal abstractions in tandem to make the most out of HRL paradigm. Furthermore, HRL methods take one step closer to addressing the problem of learning and representing knowledge by introducing some independence and modularity in the constructed state hierarchies. Such modular hierarchies can be used to represent the knowledge learned by an agent wherein any new concepts learned by an agent can be added on as modules.

There are two ways to construct hierarchies using these abstractions: a) by learning temporal abstractions in terms of temporally extended actions, referred as abstract actions, and reorganizing the problem domain using them, or b) by re-structuring the state space using a known environment model and learn abstract policies to navigate in it. The first approach discovers useful sub-policies through some predefined heuristics and, using it, the original problem is decomposed. But, such an approach may result in constructing an *artificial* hierarchy, governed implicitly by the heuristics employed for learning sub-policies. The second approach considers the inherent hierarchies and causal relations between state variables present in the problem domain. The construction of hierarchies using the second approach seems more easier to generalize for different problem domains when compared to the first which requires predefined heuristics suitable for the current problem domain. This approach also seems natural and more similar to how human brain learns, by learning various interactions between variables. It agrees with the conclusions of Simon [3], Utgoff and Stracuzzi [4] that the knowledge hierarchy of an agent evolves starting from simpler building blocks to a more complex structure. Agents build their knowledge by moving their *frontier of receptivity* as they acquire new concepts by building on earlier ones in a bottom-up manner. Once such a knowledge hierarchy is built by the agent, it then can learn the ways to navigate this structure. It seems analogous to how humans solve the problem of navigation in an unknown space. A map of the new environment is first learned, albeit partially, and then efforts are aimed to travel to different regions for finding a solution. When the intricacies and constraints of the new domain are understood, the learned knowledge is exploited for finding optimal solutions.

The problems of scalablity and sample requirements can further be addressed using model-

based RL techniques and integration of planning with learning. Model-based techniques in RL learn the transition probabilities and reward distributions in the problem domain and use it to converge faster to the optimal solution. The integration of planning with learning is interesting problem to solve as planned trajectories can take into account both intrinsic and extrinsic rewards representing exploration and exploitation [5]. After sufficiently exploring the problem domain, the planning phase introduces a goal directed behavior in an agent resulting in faster convergence. In case of multi-task RL problems, with all the required low-level policies learned in the previous problems, a planning scheme can be thought of as agent strategizing and stitching together its known information for solving the current problem.

The agent is required to know or learn hierarchies in the problem domain for breaking down the original task. Various methods have been proposed to learn these hierarchies using graph-based techniques [6–8], or by identifying sub-goals with visits and rewards statistics [9–12], or by extracting commonalities in the learned policies [13–15], or by understanding causal structure of the problem domain [16]. Learning the structure using sub-goal identification methods may result in an *artificial* hierarchy as such a hierarchy may not capture the causal interactions in the problem domain. Similarly, hierarchies constructed using reward or visit statistics or graph-based techniques may not reflect the underlying causal relations and constraints. Most of these methods take decisions using the *full* state information and, hence, not taking a complete advantage of the introduced state abstractions. This inefficiency needs to be addressed for accurate decomposition of the original task. HEXQ [17], VISA [16] and Skill-Symbol loop [18,19] methods make use of state and temporal abstractions together and plan their actions with higher level observations. Furthermore, skill-symbol loop also combines abstract actions with planning techniques exploiting an agent's current knowledge of the problem domain. However, HEXQ implements a variable change frequency heuristics which results in construction of a linear hierarchy, VISA method requires a Dynamic Bayesian Network (DBN) model of the environment along with the conditional probabilities describing the transition and reward functions, and Skill-Symbol loop method assumes available of abstract actions for learning new state representations. The heuristics used by HEXQ fails with increase in the problem state dimension as it ignores the parallelism present in the inherent hierarchy. VISA extensively makes use of the given conditional probability distributions which get harder to define with increase in the number of state variables in any given problem domain. Hence, the need to address hierarchy construction with minimal dependence on a priori knowledge.

1-2 Research Question

For planning problems, an agent using hierarchical reinforcement learning can effectively break down the original task into smaller sub-tasks creating a hierarchy of tasks. Construction of such a hierarchy has been achieved in previous HRL methods (viz. HEXQ [17], VISA [16], etc.) by assuming a predefined heuristics or the knowledge of dynamics of the problem domain in terms of conditional probability distribution. The main research question is: *How should we construct hierarchies in the problem domain without extensively depending on any given heuristics or probabilities while still addressing the increase in problem dimensions?* The research question can further be divided into following questions:

- How should we construct state hierarchies and abstractions for a generic problem domain with minimal dependence on a priori knowledge?
- How should we efficiently learn policies to navigate between the learned abstractions in states and effectively decompose the value function to obtain a solution? Furthermore, the consequences of such a value decomposition on the optimality of this solution need to be addressed.

- How do we bootstrap value learning using the learned knowledge of the environment? How can classical planning techniques be combined with learning abstract actions for faster convergence of learned policies?

1-3 Contributions

The thesis work explores incrementally building up the hierarchy using the agent-environment interactions and then inducing a goal-directed behavior in an agent via planning. A novel algorithm combining HRL approach with planning techniques, to address scalability and sample requirements of RL solutions, is proposed and evaluated in different problem domains. The proposed algorithm learns a hierarchy in discrete-time factored MDP domain by inferring causal relations from a Dynamic Bayesian Network model and makes use of model-based RL techniques for faster convergence to a solution.

In an MDP with discrete states, state hierarchies with associated abstract actions can be constructed using the knowledge learned from interacting with the environment and (*one-step*) causal relations given by the DBN model without the use conditional probabilities. The thesis assumes a priori knowledge available to an agent to be the causal relations between the state variables. For constructing state hierarchies in a discrete state domain, a method inspired from HEXQ [17] and VISA [16] is proposed which identifies state abstractions and hierarchies using the causal relations and learns abstract actions to navigate around while interacting with the environment. An abstract action discovery scheme is devised to construct abstract actions for state variables present higher in the hierarchy, i.e., having a lot of dependence on other variables, and a complementary value update scheme similar to HEXQ [17] is used for learning the policies for the discovered abstract actions. A simple abstract action selection scheme is proposed for effectively combining action selection using the learned value function and classical planning techniques at each level of the constructed hierarchy.

In the proposed method, the underlying principle is that hierarchies are learned bottom-up by interacting with the environment while reasoning is introduced in an agent's behavior through planning in a top-down manner. *Knowledge builds from bottom-to-top and reasoning flows from top-to-bottom.* The main contribution of the research work lies with the proposed algorithm for constructing state hierarchies and discovering abstract actions along with the hybrid action selection scheme. The experimentation in both deterministic and stochastic environments for comparing the proposed method with HEXQ, *flat* and Model-Based RL is carried out and discussed.

1-4 Structure of the Thesis

The structure of the thesis is as follows. Chapter 2 discusses the relevant mathematical preliminaries that form an appropriate background of work related to Hierarchical Reinforcement Learning. The topics covered in this chapter include an overview of MDP and SMDP theory along with an introduction to Reinforcement Learning and Hierarchical Reinforcement Learning. Chapter 2 ends with a brief summary of the related works in HRL for hierarchy construction. Chapter 3 provides a detailed exposition of the proposed algorithm along with its pseudo-code. The experimentation carried out for comparing the performance of the proposed algorithm with other methods is discussed in chapter 4. Chapter 5 concludes the work discussing the research findings and provides future research directions.

Chapter 2

Technical Preliminaries

The chapter discusses the required background for Hierarchical Reinforcement Learning. An introduction to Markov Decision Processes, Semi-Markov Decision Processes, and Reinforcement Learning are provided in along with a summary of the components of Hierarchical Reinforcement Learning. The chapter concludes with a discussion on related works about learning hierarchy in Hierarchical Reinforcement Learning methods and challenges present in them.

2-1 Markov Decision Processes

In the field of machine learning, Markov Decision Processes (MDPs) are widely used for modeling interactions between an agent and the environment. It provides a framework for modeling decision making in scenarios where outcomes are partly stochastic due to transitional probabilities and partly under the control of an agent aiming to maximize the rewards. A discrete-time MDP M can be formalized as a tuple $M = (S, A, T, R)$ with a finite discrete state space S, a finite action space A, a transition function $T : S \times A \times S \to [0, 1]$ as a distribution $P(s'|s, a)$ over next states where s' is the new state reached after action a is performed at state s and a reward function $R : S \times A \times S \to \mathbb{R}$ indicating the reward $R(s, a, s')$ received by an agent after reaching state s' when action a is executed in state s. The outcome of a planning problem formulated as an MDP is a policy defined by a mapping from states to actions ($s \to a$) and can be either deterministic or stochastic in nature. For computing optimal policies to solve an MDP, a value function representing an estimate of how good it is for an agent to be in a certain state is learned. It is evaluated as an expected return that can be achieved in future when a certain policy is followed from that state. The value function $V^\pi(s)$ is the value of state s under the policy π i.e. the expected return when starting in state s, following policy π thereafter with discount factor γ. The value function $V^\pi(s)$ for an infinite-horizon MDP is given in equation 2-1 where E_π denotes the expected value under policy π.

$$\begin{aligned} V^\pi(s) &= E_\pi\{\sum_{k=0}^{\infty} \gamma^k r_{t+k} | s_t = s\} \\ &= \sum_{s'} T(s, \pi(s), s')(R(s, a, s') + \gamma V^\pi(s')) \end{aligned} \tag{2-1}$$

Similarly, a state-action value function, also referred as Q-function $Q : S \times A \to \mathbb{R}$, gives the expected reward when an agent starts from state s taking an action a and follows policy π

thereafter. Q-function is given in equation 2-2. Q-function associates value to the state-action pair while value function $V(s)$ associates with the state only.

$$Q^{\pi}(s,a) = E_{\pi}\{\sum_{k=0}^{\infty}\gamma^{k}r_{t+k}|s_t = s, a_t = a\} \tag{2-2}$$

An optimal policy π^* is defined as $V^{\pi^*}(s) \geq V^{\pi}(s)$ for all $s \in S$ and for all policies π. Hence the optimal solution V^* of any MDP is V^{π^*} and it satisfies,

$$V^*(s) = \max_{a\in A}\sum_{s'\in S}T(s,a,s')(R(s,a,s') + \gamma V^*(s')) \tag{2-3}$$

$$\pi^*(s) = \arg\max_{a\in A}\sum_{s'\in S}T(s,a,s')(R(s,a,s') + \gamma V^*(s')) \tag{2-4}$$

where, equation 2-3 is called the *Bellman optimality* equation. The *Bellman optimality* equation states that the value of a state under an optimal policy is the expected reward for the best action in that state. The optimal action to be selected using the optimal value function is given by equation 2-4. This policy for greedily selecting the best action using a value function V is called *greedy* policy $\pi_{greedy}(V)$. The optimal state-action value function is given in equation 2-5 and the greedy policy defined over it is given in equation 2-6.

$$Q^*(s,a) = \sum_{s'\in S}T(s,a,s')(R(s,a,s') + \gamma V^*(s')) \tag{2-5}$$

$$\text{where } V^*(s) = \max_{a} Q^*(s,a)$$

$$\pi^*(s) = \arg\max_{a} Q^*(s,a) \tag{2-6}$$

The advantage of using Q-function over value function V for learning an optimal policy is that no model of the underlying MDP needs to be considered when defining the policy. As seen in equation 2-6, a greedy policy $\pi_{greedy}(Q)$ can solely be defined using Q-function without needing any information of the transition and reward function probabilities. Such probabilities are required when a policy is defined using value function V as given in equation 2-4.

The optimal solution of an MDP is obtained by computing the optimal value functions given in equations 2-3 and 2-5. The equation 2-3, used for obtained the optimal value function V^*, is a system of n equations in n unknowns with n being the dimension of the state space. Solving equation 2-3 becomes possible only when dynamics ($T(s,a,s')$ and $R(s,a,s')$) of the environment is known and, hence, it requires a prior knowledge of the world. The equation 2-5 used for solving optimal Q-function is a system of $n \times n_a$ equations in $n \times n_a$ unknowns where n_a is the number of actions in the action set. Hence, in case of Q-function, the ease of finding a greedy policy comes at the cost of increased computation.

2-1-1 Factored Markov Decision Processes

Factored MDPs [20] are one of the approaches for representing large, structured MDPs compactly. In the FMDP representation, a *state* is implicitly described by an assignment to some set of state variables. In a factored MDP, the set of states is described via a set of random variables $\mathbf{X} = \{X_1, ..., X_n\}$, where each X_i takes on values in some finite domain $domain(X_i)$. A state x defines a value $x_i \in domain(X_i)$ for each variable X_i. A Dynamic Bayesian Network [21] can then allow a compact representation of the transition model, by exploiting the fact that the transition of a variable often depends only on a small number of other variables. A Dynamic Bayesian Network (DBN) is a Bayesian network which relates variables to each

other over adjacent time steps and is often referred to as a *Two-Time slice* Bayesian Network (2TBN) because it says that at any point in time T, the value of a variable can be calculated from the internal regressors and the immediate prior value (time $T-1$), complimenting the Markov property of MDPs. The momentary rewards can often also be decomposed as a sum of rewards related to individual variables or small clusters of variables. There are two main types of structures that can simultaneously be exploited in factored MDPs: *additive* and *context-specific* structures [22]. Additive structure captures the fact that typical large-scale systems can often be decomposed into a combination of locally interacting components. Such additive structure can also be present in the reward function. Context-specific structure encodes a different type of locality of influence: Although a part of a large system may, in general, be influenced by the state of every other part of this system, at any given point in time only a small number of parts may influence it directly. These structures directly signify the inter-connected and intra-connected systems in the given problem domain. As factored MDPs exploit these structures, learning a hierarchy in a problem domain represented using a factored MDP becomes simpler.

2-2 Semi-Markov Decision Processes

Semi-Markov Decision Processes (SMDPs) are MDPs with extended actions [23]. In an SMDP, an abstract action can take a random number of time-steps to terminate, hence a new variable representing the duration of execution of an abstract action is required to be introduced in the MDP formulation. The model of an SMDP with added random variable $N \geq 1$ representing the number of time-steps an abstract action a takes to complete, starting in state s and terminating in state s', can be defined using the updated state transition probability and the expected reward function. The transition function $T : S \times A \times S \times N \rightarrow [0,1]$ gives the probability of an abstract action a terminating in state s' after N time-steps, after initiating from state s, and is given as,

$$T(s,a,s',N) = Pr\{s_{t+N} = s' | s_t = s, a_t = a\} \tag{2-7}$$

The reward function $R : S \times A \times S \times N \rightarrow \mathbb{R}$ gives the expected discounted-sum of future rewards when starting in state s with an abstract action a and terminating in state s' after N steps, and is given as,

$$R(s,a,s',N) = E\{\sum_{n=0}^{N-1} \gamma^n r_{t+n} | s_t = s, a_t = a, s_{t+N} = s'\} \tag{2-8}$$

Similar to MDPs, the value of a state s under a policy π of an SMDP is the expected return after starting in state s at time t and taking next abstract action according to the policy π. Consider an abstract action a that continues for N time-steps when starting from state s, then the value function $V^\pi(s)$ can be given as the sum of the rewards accumulated for the first N steps and remainder of the series.

$$\begin{aligned} V^\pi(s) &= E_\pi\{\sum_{k=0}^{\infty} \gamma^k r_{t+k} | s_t = s\} \\ &= E_\pi\{(r_t + \gamma r_{t+1} + ... + \gamma^{N-1} r_{t+N-1}) + (\gamma^N r_{t+N} + ...) | s_t = s\} \\ &= \sum_{s',N} T(s,\pi(s),s',N)[R(s,\pi(s),s',N) + \gamma^N V^\pi(s')] \end{aligned} \tag{2-9}$$

where, summation is taken over s' and N using the joint transition probability and reward function given in equation 2-7 and 2-8 respectively. Similarly, Q-function $Q^\pi(s,a)$ representing the value of taking action a in state s and following the policy π thereafter is given by,

$$Q^\pi(s,a) = \sum_{s',N} T(s,a,s',N)[R(s,a,s',N) + \gamma^N Q^\pi(s',\pi(s'))] \tag{2-10}$$

The optimal value and Q-function for SMDPs can be given using equations 2-9 and 2-10 as,

$$V^*(s) = \max_a \sum_{s',N} T(s,a,s',N)[R(s,a,s',N) + \gamma^N V^*(s')] \quad (2\text{-}11)$$

$$Q^*(s,a) = \sum_{s',N} T(s,a,s',N)[R(s,a,s',N) + \gamma^N V^*(s')] \quad (2\text{-}12)$$

$$\text{where, } V^*(s') = \max_{a'} Q^*(s',a')$$

In equations 2-11 and 2-12, $T(s,a,s',N)$ represents a joint transitional probability defined over both the variables s' and N, given by $P(s',N|s,a)$. With this joint probability distribution, backing up of a value over the duration of execution of an abstract action becomes difficult. This problem constitutes a temporal credit assignment problem in HRL methods and requires a different value update scheme based on the SMDP theory when compared with the value update scheme in MDPs.

2-3 Reinforcement Learning

The field of reinforcement learning is mainly concerned with how a software-defined agent ought to take actions in an environment so as to maximize its notion of cumulative reward [24]. Among the two ways of solving MDPs, the classical dynamic programming methods assume knowledge of the environment in terms of an exact mathematical model while reinforcement learning methods target MDPs where such exact methods become infeasible. RL methods make use of the MDP theory and define the value function and Q-function as specified in equations 2-1 and 2-2 respectively. As discussed previously, the outcome of a planning problem is a policy defined by a mapping from states to actions ($s \rightarrow a$) and can be either deterministic or stochastic in nature. In most RL methods, the policy is deterministic and is computed from optimal value and Q-function using equation 2-4 or 2-6. There are two ways of solving equation 2-3: a) Policy Iteration and b) Value Iteration [24]. Policy iteration consists of performing a policy evaluation, wherein value function is iteratively computed using Bellman equation until it converges to V^π, and a policy improvement step which computes a new policy using the converged value function V^π. The policy evaluation step follows the update rule give by,

$$V_{n+1}(s) = \sum_a \Pi(s,a) \sum_{s'} T(s,a,s',N)[R(s,a,s',N) + \gamma^N V^*(s')]$$

where π denotes the deterministic policy currently being followed and $\Pi(s,a)$ is a matrix with $\Pi(s,\pi(s)) = 1$ and zeros elsewhere. The sequence $\{V_n\}$ is guaranteed to converge to V^π as $n \rightarrow \infty$ and $\gamma < 1$. In policy improvement step, V^π is used to compute the new policy greedily,

$$\begin{aligned} \pi'(s) &= \arg\max_a Q^\pi(s,a) \\ &= \arg\max_a \sum_{s'} T(s,a,s',N)[R(s,a,s',N) + \gamma^N V^*(s')] \end{aligned}$$

Value iteration performs only one sweep over the state-space for the current policy in policy evaluation step and then performs policy improvement similar to policy iteration. The update rule for policy evaluation is derived from the *Bellman optimality* equation and is given as,

$$V_{n+1}(s) = \max_a \sum_{s'} T(s,a,s',N)[R(s,a,s',N) + \gamma^N V^*(s')]$$

Policy and value iteration methods require a model of the world to be known a priori and, hence, are studied as a part of *model-based* RL techniques. If no such model is known, then

an agent is first required to build it and then learn a value function over it. Another way of solving such a scenario is by using Temporal Difference (TD) methods. TD methods do not require a model of the environment to be known and fall under the umbrella of *model-free* RL methods [24]. In TD methods, an agent processes the immediate rewards it receives at each time step, and thereby learning from each action. The three widely used implementations of the TD method are Watkins' Q-learning [25], SARSA [24, 26] and actor-critic RL [27]. The thesis work makes use of Watkins' Q-learning method for learning the optimal Q-function. In simplest form, 1-*step* Q-learning is defined by,

$$Q(s_k, a_k) \leftarrow Q(s_k, a_k) + \alpha * (r_{k+1} + \gamma * \max_a Q(s_{k+1}, a) - Q(s_k, a_k)) \tag{2-13}$$

The Curse of Dimensionality

The curse of dimensionality refers to phenomena observed when analyzing and organizing data in high dimensional spaces wherein, as the problem dimensionality increases, the volumes of the search space for solutions increases so fast that the available data becomes sparse resulting in more efforts for finding an optimal solution. It can clearly be seen manifesting in RL methods through the MDP theory. As discussed previously in section 2-1, the optimal value function and Q-function are computed using equations 2-3 and 2-5 respectively and have n and $n \times n_a$ unknowns for a problem domain having a single variable with n being its dimension and n_a representing the number of actions available to an agent. As the number of state variables increase, the sizes of value function and Q-function grow as $(n_1 \times n_2 \times ... \times n_k)$ and $(n_1 \times n_2 \times ... \times n_k \times n_a)$ where k is the number of state variables and n_i, $i = 1, ..., k$ is the state space dimension for i_{th} variable. As the problem dimensions increase, size of each value function increases exponentially and rewards get sparser resulting in more computational effort for finding optimal solution. In RL, even if few variables in the problem domain are independent from each other, the values corresponding to them are repeatedly learned in different scenarios. With HRL, such repetitive values collapse together and are computed only once for dependent variables. Furthermore, value functions in HRL are learned in sections corresponding to states which depend extensively on each other, reflecting the intra-component relations in the problem domain. Such sections are collapsed together and further connected to other sections, reflecting inter-component relations. A hierarchy can be seen to emerge in the ways of interaction between these sections in the value spaces.

2-4 Components of Hierarchical Reinforcement Learning

The use of abstract actions with SMDPs lead to a hierarchical decomposition of original MDP as abstract actions may invoke a set of different abstract actions creating levels of hierarchy. The newly formed SMDPs, after dividing the original MDP, may have abstract actions which are policies of smaller SMDPs such that higher SMDPs call for policies of lower ones. This relationship between SMDPs lead to task-hierarchies [28], a directed acyclic graph where the root node is a top-level SMDP which invokes its child nodes i.e. the smaller SMDPs. The child nodes recursively invoke their child sub-tasks till only primitive actions are remaining. Task hierarchies give a general flow of control that takes place when following solutions of HRL methods i.e. hierarchical policies. With such a hierarchical structure, various state and temporal abstraction opportunities are presented along with credit assignment problems. Hence, HRL methods can be studied with respect to the state and temporal abstractions induced in the problem domain, their value decomposition scheme and its consequences on the optimality of the solution. The following sections discuss these components briefly.

2-4-1 State and Temporal Abstraction

Any real-world planning problem formulated as an MDP is typically defined over a large state space wherein few state variables might be independent of each other. For example, the navigation policy to leave a room is independent of the color of the walls of the room. Hence, such cases provide the opportunities for state abstractions resulting in a reduced complexity of finding an optimal policy for the original MDP. Abstractions can be introduced in a problem domain when: (a) there are irrelevant state variables, or (b) available abstract actions funnel the agent through small sub-sets of states [29]. The first approach results in an *organized* state space resulting in state abstraction and can be constructed using the causal structure of the problem domain. In the first approach, abstract actions are then learned over the *organized* state space. The second approach gives an opportunity to directly form abstract actions, resulting in temporal abstraction. Both state and temporal abstractions can be used to reduce the search space of a solution.

- **Eliminating irrelevant state variables**: If a learned skill or an abstract action is independent of some state variables, then the skill can be learned without consideringsuch variables and hence reducing the solution search space. In case an irrelevant variable changes, instead of re-learning the policies every-time, they could be reused.
- **Funneling**: Abstract actions may guide agents from large number of initial states to a small number of termination states. For example, in room leaving policies, an agent beginning from any state in the room will end up at the doorway state. Hence, at the root task, the whole room can be abstracted to just these funnel states, irrespective of agent's initial position.

Abstract Actions

In SMDPs, actions are allowed to extend for multiple time steps. Such temporally extended action sequences are grouped together into a single operator referred to as an abstract action. Such temporally extended actions result in agent skipping over some subset of the state space. Abstract actions introduce temporal abstraction as until it terminates, the current SMDP doesn't need to take any decision i.e. the control exists with the current abstract action. Once the abstract action is terminated, the control is given back to the current SMDP. They are similar to *macros* in computer science which make available a sequence of instructions as a single programming statement. In HRL literature, there are many terms having notions similar to temporally extended actions like skills, options, sub-policies, behaviors, partial programs or macros.

The options formulation [30] is a widely used framework for representing abstract actions. In the framework, an option O is formed using a tuple (I, μ, β), with the initiation set I where option O can be initiated, a policy function μ for option O where $\mu(s, a)$ gives the preference value given to an action a in state s and a termination function β such that $\beta(s)$ gives the probability of option O terminating in state s. The options framework makes use of the SMDP theory for updating defined value functions. In the thesis work, options formulation is used for representing learned abstract actions. The original actions in an MDP which extend for one time-step are referred as *primitive actions* in the SMDP theory. These primitive actions can also be included into the set of abstract actions.

By adding abstract actions to the formulation, it may seem that the complexity of the problem has increased as the agent has more choices. However, abstract actions help in *skipping* over the state space quickly without taking random actions and backing-up the value function over

greater distances [30]. With an appropriate abstract action, a policy of an agent is more constrained (sub-goal-oriented), and hence it can skip over a large state space, without much exploration, and terminates into a smaller section of the state space. Though, if the goal is located at a certain state which is not a termination state for any abstract action, then inclusion of primitive actions is a necessity. In such cases, inclusion of primitive actions guarantees global optimality, but with increased storage and exploration. Furthermore, such a combination of primitive and abstract actions lead to increase in search space of the solution. Hence, this problem of defining the action set for an agent needs further investigation.

2-4-2 Value Function Decomposition

With a hierarchy of SMDPs, it is necessary to decompose value functions to compute optimal value of a state at each level. In RL methods, as rewards are handled immediately, a temporal credit assignment problem is raised because a *good* action taken at the beginning of a learning episode needs to be rewarded appropriately. Furthermore, in HRL, with the introduction of abstract actions, the value needs to be appropriately backed-up to the beginning and the *good* decisions taken at the start of a learning episode. In a planning problem with a goal to reach an exit state of a house with several rooms, the decision of which doorway to target in a particular room depends on the current state of an agent inside the room and overall goal location with respect to each doorway. The success of an agent in reaching the overall goal is founded in the decision taken in the beginning and hence needing an appropriate reward.

For addressing the temporal credit assignment problem, the MAXQ approach [28] decomposed the value function in two parts with one representing value due to termination of the current sub-task and another one for termination of the current abstract action invoked by the current sub task. The thesis work makes use of value decomposition similar to MAXQ, explained further in detail, with the use of *pseudo* rewards. A three part decomposition of value function was proposed by Andre and Russell [31] which also takes into account the context of an abstract action, making solutions hierarchically optimal. It involves an added component of the value due to termination of the overall task after the current sub-task terminates. An important concept, here, is that the way value function is decomposed over a task hierarchy determines the optimality of resulting solutions along with the properties of learned sub-policies. In MAXQ approach, value decomposition only considers values until completion of the current sub-task and neglects values due to completion of further sub-tasks or the original task. Thus, the sub-policies learned are specific to the sub-task at hand. Such sub-policies are recursively optimal (or optimal if only primitive actions invoked) and are *context-free* i.e. independent of their parent task. Thus, such policies can be reused easily in multi-task RL problems. While, in case of the three part decomposition by Andre and Russell, due to inclusion of value associated with the overall task completion, learned policies are hierarchically optimal, but at the expense of less state abstraction [1], and are context dependent i.e. specific to the parent sub-task.

The MAXQ approach uses SMDP theory and breaks down the value of a state into a sum of sub-task completion values and the expected reward for the immediate primitive action. The completion value is the expected cumulative reward on completion of a sub-task after executing an abstract action. For a specific sub-task m, the state-action value function Q for an SMDP can be given as,

$$Q^{\pi}(m,s,a) = \sum_{s',N} T(m,s,a,s',N)[R(m,s,a,s',N) + \gamma^{N} Q^{\pi}(m,s,\pi(s'))]$$

where, the hierarchical policy π is a set of policies defined for each sub-task, an abstract action a which for the sub-task m invokes a child sub task m_a.

The expected value for completing sub-task m_a is expressed as $V^\pi(m_a, s)$. The completion function $C^\pi(m, s, a)$ is the expected cumulative reward after completing an abstract action a from state s in sub-task m and backed-up to the beginning of a. It represents the expected reward that will be obtained by following hierarchical policy π till the end of the sub-task m.

$$\begin{aligned} C^\pi(m, s, a) &= \sum_{s', N} T(m, s, a, s', N)\gamma^N Q^\pi(m, s', \pi(s')) \\ Q^\pi(m, s, a) &= V^\pi(m_a, s) + C^\pi(m, s, a) \end{aligned} \tag{2-14}$$

Hence, Q-function can be expressed as a summation of the expected value of a sub-task m_a and the completion function as expressed in equation 2-14. The value of a sub-task m_a is given as,

$$V^\pi(m_a, s) = \begin{cases} Q^\pi(m_a, s, \pi(s)) & \text{if } a \text{ is abtract} \\ \sum_{s'} T(s, a, s')R(s, a, s') & \text{if } a \text{ is primitive} \end{cases} \tag{2-15}$$

For any such sub-task, the expected value can be further decomposed till a primitive action a is executed. For example, for a root task m_0, if the sub-tasks invoked are $m_1, ..., m_k$ with m_k executes a primitive action and a_i is an abstract action invoked in a sub-task m_i governed by the hierarchical policy $\pi(s)$ then by recursively decomposing the value function $V^\pi(m_i, s)$, the state-action value for the root task m_0 can be given as,

$$Q^\pi(m_0, s, \pi(s)) = V^\pi(m_k, s) + C^\pi(m_{k-1}, s, a_{k-1}) + ... + C^\pi(m_1, s, a_1) + C^\pi(m_0, s, a_0) \tag{2-16}$$

2-4-3 Optimality

HRL methods in general can't guarantee the optimality of resulting solutions after the problem decomposition. The solution depends highly on the available abstract actions, the used value decomposition scheme and the constructed hierarchy. Hence, the optimal solutions using HRL methods can qualitatively be described as,

- **Hierarchically optimal**: Such policies optimize the overall value function consistent with the constraints imposed by the task hierarchy.
- **Recursively optimal**: Recursively optimal policies [28] are *context-free* policies i.e. they try to reach their goal states ignoring the needs of their parent tasks.
- **Hierarchical-greedy optimal**: An optimal abstract action may become sub-optimal during the course of its execution due to stochastic drift. In such cases, such a sub-optimal action can be constantly interrupted to choose a better one. Such an optimality is called Hierarchical Greedy optimality [28].

Consider that an MDP M is broken down into several SMDPs M_i with a hierarchical policy defined as $\pi = \{\pi_1, \pi_2, ..., \pi_n\}$ where π_i is the policy of M_i. Then for a recursively optimal hierarchical policy, a policy $\pi_i \in \pi$ is optimal for an SMDP M_i given the policies of all the child sub-tasks of M_i. Instead, hierarchically optimal policy is a hierarchical policy that is optimal policy among all the policies that can be specified given the constraints imposed by the hierarchy.

In simpler words, a hierarchical optimal policy for a given task hierarchy is a hierarchical policy that has the best possible reward among all possible hierarchical policies. A hierarchical policy π is recursively optimal if each policy π_i, defined in π, is optimal given the descendants' policies of π_i in the task hierarchy. The important distinction is that the optimality of π or π_i is

defined with respect to their child tasks only without any consideration for their parent tasks. The recursively optimal solution is arbitrarily worse than hierarchical optimal solution and the later arbitrarily worse than global optimal solution. However, abstract actions with weaker recursively optimality can be reused in different scenarios, making them the building blocks for other hierarchies. Hierarchical greedy optimality is guaranteed to be no worse than hierarchical or recursive optimality and may be considerably better. But no guarantees can be provided about it being globally optimal as it may sometimes suffer from the constraints induced by the task hierarchy and available abstract actions.

2-5 Related Works

The section presents the previously proposed HRL methods and discusses the challenges associated with these methods. Representing a problem in a hierarchical structure requires the knowledge of how to decompose the problem and how to introduce abstractions to balance the complexity against the loss in optimality. Automating this decomposition is more difficult, but is more desirable. In real world problems, the complexity of finding an optimal solution can be reduced by abstracting the regularities in the problem state space. The choice of variable representation plays a large part in providing opportunities to decompose the problem and hence many HRL methods make use of FMDP representation. Similarities in different state space regions provide a chance of state abstraction while extended action make temporal abstraction possible. However, for temporal abstraction, abstract actions leading only to useful sub-goals should be learned to reduce the problem complexity. Hence, many researchers have tried to learn the hierarchical structure by first learning the sub-goals to partition the problem into near independent reusable sub-problems and then learning abstract actions to reach such sub-goals. For automatically decomposing the problem, few methods look for sub-goals or landmark states, while others look for common trajectories or region in learned policies.

The field of HRL was started with H-DYNA proposed by Singh [32] as an extension to Sutton's DYNA [33] wherein a hierarchy of abstract models was learned using the agent-environment interactions. H-DYNA constructs a hierarchy of variable temporal resolution models (VTRMs), learned over multiple related tasks having a policy module at each level of the hierarchy with a single evaluation function module. The evaluation function modules maintains an estimate of optimal value function for each task and gives payoff to one of the abstract model after completion of a task. A method using high reward gradient and bottleneck states to abstract features was used by Digney [10]. But this method is highly dependent on the specified reward function. A simpler approach for automatically determining potentially useful sub-goals by detecting regions that agent visits often on successful trajectories but not on unsuccessful trajectories was given by McGovern [34] and McGovern and Barto [11] wherein an agent learns abstract actions to the regions which appear early in learning and persist throughout.

Learning the hierarchical decomposition of a complex task evolves from the bottom-up while a control behavior is employed from the top-down. According to Simon [3], *complex systems will evolve from simple systems much more rapidly if there are stable intermediate forms.* The compression inherent in the progression of learning from simple to more complex tasks was pointed out by Utgoff and Stracuzzi [4]. Their work suggests a building block approach, designed to eliminate replication of knowledge structures. With this as motivation, the thesis work proposes ways to construct hierarchies by building from the bottom-up by learning abstractions present in the state space while the later part introduces reasoning from the top-down using simple planning techniques. The following section presents the previously proposed method for learning hierarchies.

2-5-1 Learning Structure

Various methods for autonomously learning hierarchy have been proposed which use different interaction information to infer the structure. Some methods learn hierarchy by identifying bottleneck or gate-way states in the problem domain using visit statistics of a certain state in successful trials [6, 11] or relative novelty introduced by a state [9]. Using such heuristics, problem state space is reduced and sequences of actions are learned. Some approaches build a linear hierarchy using predefined frequency of variable change heuristics [17,35,36], while others leverage the model of environment by using condition probabilities [16] to plan sets of actions beforehand. Few other methods learn useful sequences of actions satisfying predefined criteria and chain them together to form a solution for the underlying problem [37]. In general, the two opportunities for abstraction or creating the building blocks for hierarchy construction are mentioned previously in section 2-4-1: Variable elimination and Funneling. Based on these two approaches, the methods for autonomously learning hierarchy can be loosely classified in two groups: (a) methods which start with temporal abstraction i.e. by constructing temporally extended actions to useful states, and (b) methods which structure the state space to begin with and then learn actions to navigate these structures. The thesis work mainly focuses on the second approach and hence, the related methods are discussed ahead.

A simple way of modeling a state space is by using the constraints present in it. Such constraints are further reflected in the way state variables interact with each other i.e. their causal relations. Such constraints lead to a hierarchy and a variable order such that variables higher in hierarchy dependent on the lower ones. For identifying such ordering of state variables, few heuristics and assumptions are used in HRL methods discussed further in detail.

HEXQ

The HEXQ (Hierarchical exit Q-function) [17] approach is motivated from the MAXQ approach. It constructs a hierarchy starting from base level states and primitive actions by identifying the regions in the state space which are similar or strongly connected and learns the *exit* abstract actions for such regions. Using the *exit* abstract actions, it recursively formulates reduced SMDPs at higher levels. The approach relies on having a discrete state space with finitely number of states. It exploits a factored state representation of MDPs and makes use of variable change heuristic to order the states in a hierarchy. The reasoning behind this heuristics is that the variables lower in hierarchy should be frequently changing. It mainly employs funnel type state abstractions. States are included in the same regions when transitions do not change any other variables. Whenever this condition is violated for a state transition, an *exit* represented by a state-action pair (s, a) is created. If this exit cannot be reached from all states in a region, then the region is further divided and extra *exits* are created, such that these exits can be reached with probability 1. The learned exit policies are considered as abstract actions at higher levels in hierarchy. The hierarchical policy learned by HEXQ algorithm is recursively optimal [17]. Several updates for HEXQ algorithm have been proposed to address concurrent learning and self-repairing of hierarchies [35], partial order planning [36], safe state abstraction with updated value decomposition formulation [38].

The value update scheme implemented in HEXQ is inspired from the MAXQ approach. HEXQ includes the expected primitive reward immediately after the sub-task exits and the hierarchical value of next state, but does not include any accumulated rewards while executing the sub-tasks and hence the name: Hierarchical exit Q-function. Instead, in MAXQ, the value of a sub-task is composed of the accumulated reward during execution of invoked sub-sub-policies, represented using the value of its child task, and completion value of sub-task. The motivation for such a scheme in HEXQ stems from how rewards are assigned for primitive actions wherein the value

is composed of the reward of an action obtained immediately after its execution and the value of the next state. HEXQ extends this scheme to abstract actions resulting in recursive optimality of the obtained solutions. In HEXQ, the recursively optimal hierarchical Q-function $Q^*_{em}(s,a)$ at level e in sub-MDP m is the expected value after completing the execution of abstract action a starting in state s and following the optimal hierarchical policy thereafter and is given by,

$$Q^*_{em}(s,a) = \sum_{s'} T(s,a,s',N)[R(s,a,s',N) + V^*_{em}(s')] \tag{2-17}$$

$$V^*(s) = \max_a [V^*_{e-1,m^{e-1}(a)} + Q^*_{em}(s,a)] \tag{2-18}$$

where, s' is the next state, $m^{e-1}(a)$ is the sub-MDP implementing action a at the lower level and $R(s,a,s',N)$ is the primitive reward after execution of action a.

VISA

The heuristics of state variable ordering used in HEXQ may have an adverse effect on learning important abstract actions in large problem domains. In any such MDP, when an action is executed, the resulting values of states depend on a subset of the state variables, referred to as their contexts. In order to address the heuristics used by HEXQ, VISA (Variable Influence Structure Analysis) algorithm by Jonsson and Barto [16] made use of the causal structure in the problem domain, introduced using Dynamic Bayesian Networks (DBNs) [21]. VISA algorithm makes use of a compact DBN model of a factored MDP to build a *state variable influence graph* (SVIG) indicating the causal relations between state variables. In SVIG, the algorithm determines if there are any incoming edges to a certain state variable and, if so, then defines a set of exits. These exits are represented using their context c and an primitive action a which causes the concerned state variable to change. VISA identifies these exits by searching in the conditional probability trees (CPTs) of the DBN model and then introduces an abstract action o for each exit (c,a). Any abstract action o is admissible in states from which it is possible to reach its context c and if its associated exit causes at least one state variable to change. The termination condition for option o is defined as when the associated context is reached or can no longer be reached. Once an abstract action is successfully terminated i.e. context is reached then associated action a is always executed.

Similar to HEXQ and VISA, HI-MAT (Hierarchy Induction via Models and Trajectories) algorithm proposed by Mehta *et al.* [39] learns MAXQ task hierarchies using solved RL problems. HI-MAT applies DBN models to a successful trajectory from the problem domain to construct a *causally annotated trajectory* (CAT). In the CAT of a pruned successful trajectory, actions are annotated with relevant state variables that are checked or changed while action execution. Guided by the causal and temporal associations between actions in the CAT, HI-MAT recursively parses the successful trajectories to define MAXQ task hierarchy.

Issues with the Current HRL Methods

A brief overview of proposed HRL methods along with recent advancements is provided in [40], [1, ch. 9]. Among these methods, a very few combine temporal abstraction with state abstraction to construct a hierarchy. The approaches for temporal abstraction [13, 37, 41–43] mainly learn sequences of actions and collapse them together to create single operators. However, the decision making still makes use of the *full* state information. Most of the sub-goal identification algorithms [6–12, 15] work along similar lines as they identify bottleneck states with respect to a particular task and learn abstract actions to reach such states. This results in structuring the problem domain along with temporal abstraction. But the decision making

still takes into account the *full* state information. Furthermore, such identification of bottleneck states lead to creating an *artificial* structure which may not reflect the inherent nature of the problem domain and may not be transferable to other related tasks.

The HEXQ [17] and VISA [16] methods assume a factored state MDP and order the state variables in the given FMDP for hierarchy construction. They successively construct options to change each state higher in hierarchy and differ primarily in the method used to order the state variables. The resulting options are hierarchical in the sense that the options for modifying higher level state variables can execute those for modifying lower-level variables. The agent has access to all options at the same time, and at the same level, when learning to solve the task. However, these methods require some specific necessities. Among these methods, the updated version of HEXQ addresses partial order planning and self-repairing, but still construct a linear hierarchy not reflecting the causal relations in the domain. VISA algorithm takes the causal structure of the problem domain into account, but requires to have a DBN model of the environment explaining the problem domain in great details with the use of conditional probabilities. The HI-MAT algorithm makes use causal structure to construct compact hierarchies but requires a successful RL trajectory to parse. The hierarchy is constructed in reference of the provided successful trajectories and may not reflect the exact causal nature of the domain with no avenues for exploration. Furthermore, the issue of defining an action set available to an agent at each level of a hierarchy, discussed in section 2-4-1, is not addressed as in these methods, all the learned abstract actions are made available to the agent further increasing the complexity of finding an optimal solution. Skill-Symbol loop method proposed by Konidaris *et al.* [18, 19, 44] combines temporal and state abstraction simultaneously with skill-symbol loops and extends the use of learned abstract actions in multi-task RL problem domains by executing both learning and planning to solve similar tasks. However, the work doesn't explicitly consider any structure inherently present in the problem domain which is exploited, to a certain extent, in previous methods. In order to make the most of the HRL paradigm, state and temporal abstractions need to be used in tandem. Hence, addressing the discussed concerns in previous algorithms, the thesis work aims to investigate and propose an algorithm which makes use of the inherent hierarchy present in the state space, without having a detailed model of it, and builds temporally extended actions for navigating the learned hierarchy. The underlying motivation of the thesis work is derived from this need to have a novel method for learning hierarchy in most problem domains which tie in together both, state and temporal abstractions.

Chapter 3

A Generic Method for HRL

There mainly are two ways of constructing hierarchies: a) by learning temporal abstractions in terms of abstract actions and using them to reorganize the state space, or b) by re-structuring the state space using a known or learned environment model and learning abstract actions to navigate in it. The second approach considers the causal relations present in the problem domain and seems more similar to how human brain functions. Among the current available HRL methods, only a few method make use of both state and temporal abstractions to reduce the problem complexity. Sub-goal identification and skill learning methods result in an *artificial* state hierarchy which may not capture the causal structure of the problem domain and may fail to learn optimal policies appropriate for reaching goal. HEXQ and its updated versions use a predefined heuristics resulting in a linear state hierarchy. VISA algorithm extensively makes use of the defined transition and reward functions given by the DBN model to construct the hierarchies and abstract actions. HI-MAT parses successful trajectories to construct task hierarchies; thus requiring an agent to first reach the goal before starting the hierarchy construction.

For the hierarchy construction, inferring constraints from causal structure is very useful. However, instead of using conditional probabilities, defining which gets harder with increase in the problem dimension, we propose to use of DBN model in a minimalist sense to define only the causal relations found in the problem domain. The thesis assumes a priori knowledge available to an agent to be the causal relations between the state variables. Then by interacting with the environment, an agent can simultaneously discover abstract actions and their probabilities. Furthermore, such causal relations can also be defined by a designer with a minimal knowledge of the problem domain. There are three different phases of the proposed algorithm: a) learning hierarchies and abstract actions, b) selecting actions, and c) updating values of the followed policies. The assumptions used in the thesis work and the problem definition in discussed initially. The following sections give an overview of the different phases and a pseudo-code of the proposed algorithm.

3-1 Problem Definition and Assumptions

The thesis work assumes a usual formulation of finite goal-oriented MDPs with discrete time-steps, states and action. A real-world MDP is naturally described in a factored form. Hence, the FMDP representation of an underlying MDP is used and exploited for the hierarchy construction. It is assumed that the state s is defined by a vector of n state variables, $\mathbf{s} = (s_1, s_2, ..., s_n)$.

In general, the state variables are defined using upper case letters S while its value at a particular time instance is represented using lower case letter s with a boldfaced $\mathbf{s}$ representing vectors. The goal variable is assumed to be the topmost variable in the hierarchy. The learned abstract actions are representing using the options framework. The SMDP theory [23] which generalizes MDPs to models with variable time between decisions is used for updating values learned abstract actions. As the problem of constructing hierarchies and discovering abstract actions is largely orthogonal to the employed value learning scheme. Hence, a simple one-step Q-learning value back-up is used for abstract actions defined over primitive actions while a similar Q value update scheme from the SMDP theory is used for updating high-level abstraction actions. The transition and reward functions in the problem domain are assumed to stay constant with time and the provided DBN model representing the transition model of the problem domain is assumed to be accurate.

The objective of the thesis work is to find a solution aimed at being recursively optimal for a single goal-oriented MDP with discrete time-steps and states by maximizing the expected value of the future discounted rewards. The value decomposition used is similar to that of HEXQ, however, with the use of pseudo-rewards. The value of an abstract action is defined using only the pseudo-reward received on successful termination. With the used value decomposition scheme, each options are learned using defined pseudo-rewards while the extrinsic reward defined by a designer is only given to the goal policy after the completion of the task. Hence, the hierarchy and options are learned independent of the goal task resulting in a goal independent representation of the problem domain. However, the value decomposition used in thesis can only accommodate single extrinsic rewards as any intermediate rewards obtained for a variable lower in hierarchy are lost and not reflected at the goal node.

The proposed algorithm only makes use of an available DBN model for inferring the causal relations present in the problem domain of an FMDP with a single goal. Such causal relations can even be given by a designer without having exact model of the domain. The proposed algorithm is generic in the sense that, it does not assume or build an any environmental features. For example, the sub-goal identification methods assume presence of certain environmental features, skill-learning methods assume predefined heuristics are sufficient to break down the problem. The proposed algorithm exploits the causal relations to order the variables and then, abstract actions are learned by interacting with the environment. Such causal relations are inherent in all the domains and if no such relations exist, then the proposed algorithm falls back to a flat RL method.

3-2 Hierarchy Construction

The approach used in the thesis work for the hierarchy construction is to first structure the state space and then learn the abstract actions for navigating it. The hierarchy construction scheme is further divided into schemes for: a) constructing state hierarchy and b) discovering options. The following subsections give a brief summary of devised state hierarchy construction and option discovery schemes.

3-2-1 State Hierarchy Construction Scheme

The algorithm starts with inferring a variable order using the provided DBN model for the current task. A term, *influence graph*, is used to represent the inferred causal relations learned for the DBN model. Influence graph (IG) indicates which state variables affect changes in a particular state variable and is used for understanding contexts and other relevant attributes.

A special case of *IG* is also learned called State Variable Influence Graph (*SVIG*) similar to *SVIG* from VISA algorithm. The learned *SVIG* is *IG* without the self-dependence of state variables. The main distinction of the learned *IG* and *SVIG* from that of in VISA is in terms of the availability of conditional probability trees. Furthermore, *IG* serves as a basis for all the learned hierarchies and policies further differentiating it from *SVIG* form VISA.

Influence Graph, State Variable Influence Graph and Variable Order

IG captures a complete context of a state variable i.e. its dependence on itself and other state variables. *SVIG* only considers dependence between state variables and is used for understanding variable order without getting stuck in a loop due to self-dependence. The nodes of *IG* are the state variables and the edges between them are the actions for which corresponding variables relate to each other. The variable order *VO* is used for representing a variable order when constructing hierarchy. It is learned by first identifying the variable without any outgoing edges in *SVIG* and is considered as the root node (goal node) in hierarchy with $level = 0$. The variables further down in *SVIG* are ordered by computing their farthest distance from the root node. This farthest distance is considered as their level in the hierarchy and is used for computing *VO*. In any general problem domain, following cases of variable ordering and *IG* can be obtained:

- **Linear variable ordering**: In a linear variable ordering, *VO* does not have more than one variable at each level.
- **Interdependence of variables**: In *IG*, variables may depend on each other and may not be present at the same level in hierarchy. Such variables can be identified using strongly connected components in *IG* and can be merged together to obtained a linear variable order.
- **Partial order planning**: A state variable S may depend on two or more variables which are independent of each other resulting in a having independent variables in its context.

Without any loss in generality, for further discussions, a linear variable ordering case is considered and extensions for other two cases have been proposed in section 3-6.

Influence graph serves as a *knowledge* base for the further routines in the proposed scheme. As each node in *IG* represents a state variable from the problem domain, a modularity is added to the learned knowledge. Hence, in related problem domains, with addition or deletion of any state variable, the rest of *IG* can still be reused. Each node in *IG* has following attributes:

- *State set*: The set of states which are discovered for the current variable. Its filled in as new states are discovered.
- *Change action set*: The set of primitive actions which causes the variable to change.
- *Action set*: The set of abstract actions defined for the current state variable.
- *Primitive action set*: The set of primitive actions associated with the current state variable.
- *Sub action set*: The set of abstract actions which are supplementary to the abstract actions in *Action set* defined for the current state variable.
- *Primitive sub action set*: The set of abstract actions encapsulating a particular primitive action and are supplementary to abstract actions in *Action set* defined for the current state variable.

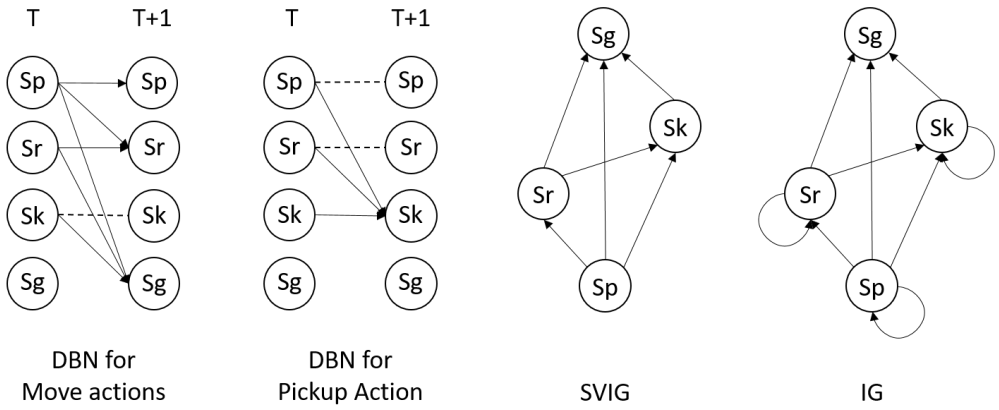

Figure 3-1: The DBN model, state variable influence graph ($SVIG$) and influence graph (IG) for the key-and-lock problem

Consider a simple key-and-lock problem wherein an agent needs to collect an exit-door key from some locations and exit the problem domain for completing the task. Figure 3-1 shows the given DBN model of a key-and-lock problem for *move* actions: $\{north, south, east, west\}$ and key *pickup* action along with the inferred state variable influence graph and influence graph. In figure 3-1, *Sg* indicates the goal variable representing whether the goal is reached by an agent or not, *Sk* stands for the key variable representing whether an agent has a key or not, *Sr* indicates room variable representing the current room occupied by an agent and *Sp* indicates position variable representing the position of an agent in the current room. An example of IG and $SVIG$ for such a key-and-lock problem is shown in figure 3-1 while an influence graph for an FMDP with n state variables is shown in figure 3-2c. The variable order inferred in this problem domain would be $\{0 : [Sg], 1 : [Sk], 2 : [Sr], 3 : [Sp]\}$ with the numerical values showing which *level* of each variable in the hierarchy. In some problem instances, *Sr* and *Sp* may depend on each other as shown in figure 3-2a or *Sk* might be independent of *Sr* as shown in figure 3-2b. Such scenarios constitute other cases of variable ordering discussed previously. Initially, among the state attributed of IG, only *change action set* and *primitive action set* are assigned while others are filled when interacting with the environment.

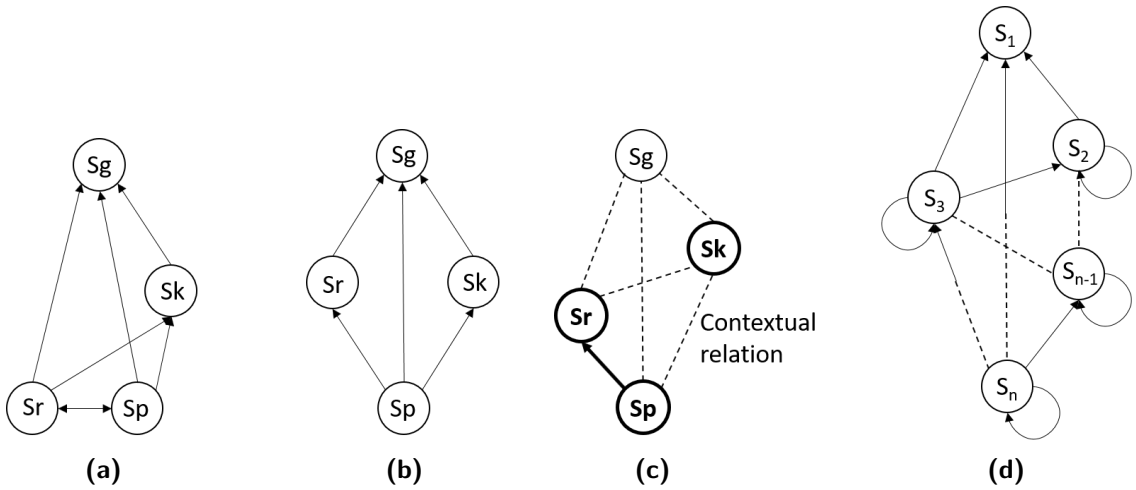

Figure 3-2: The state variable influence graph for key-and-lock problem for (a) variable interdependence and (b) partial order planning scenarios. c) Independent state hierarchies in the key-and-lock problem. d) A general influence graph for a problem domain with n state variables.

Independent State Hierarchies

Similar to influence graphs, independent state hierarchies are also inferred from the DBN model. The term *independent state hierarchies* is coined to represent a hierarchy of state variables affected by a primitive action. Such independent hierarchies have a contextual relation due to the current definition of the problem domain. Furthermore, the advantage of defining such hierarchies is that the corresponding primitive actions can be tied with the bottom-most state variables leading to a concise definition of abstract actions over only required primitive actions.

For example, any key-and-lock problem has two independent hierarchies: location hierarchy and key hierarchy corresponding to *move* and *pickup* actions, shown in figure 3-2c. Sp and Sr together define a location hierarchy and Sk defines a key hierarchy. These hierarchies are contextually related to each other under the problem definition i.e. the goal variable Sg. The independence between hierarchies is as: any changes in Sp may change Sr, but will never change Sk. Sk only changes due to *pickup* action which does not affect Sp and Sr. With such an independence, any room leaving policy for Sr is defined using only the actions associates with Sp i.e. *move* actions. Furthermore, only *move* actions are added to *change action set* of Sr.

Both, influence graph and state variable influence graph, represent the state hierarchy in terms of state variables and is directly inferred from the DBN model of an FMDP, similar to VISA algorithm [16]. However, unlike VISA algorithm, use of the DBN model is limited to constructing this state hierarchy. A state hierarchy in terms of in terms of individual state variable instantiations $\mathbf{s}$ is represented using a state transition graph G, discussed in the following section.

State Transition graph

A *state transition* graph G, defined as a multi-directional graph, is constructed using causal information from IG and the agent-environment interaction. G captures the transition information in the problem domain along with the contextual information for each transition. Figure 3-3 shows an example of state transition graph for a key-and-lock problem. All the discovered states are added to G with connecting edges to their contexts from IG. Each newly added state is connected to its context variables' corresponding instantiations. Such connecting edges are used for defining initiation set and value space of learned abstract actions and are shown as the black colored edges in figure 3-3. Furthermore, within each state variable domain, transitions are recorded with edges between states, shown using green or blue color in figure 3-3. These edges further used in recording transitions for learning probabilities. With such stored transition information, a model of the problem domain is learned and used for bootstrapping value function, discussed in detail in section 3-4-1.

3-2-2 Option Discovery Scheme

In the thesis work, abstract actions are learned to navigate the constructed state hierarchy and are represented using the options framework [30]. These options are learned using value update scheme of the SMDP theory, discussed in detail in section 3-4. This sections gives a brief overview of the devised option discovery scheme for learning options in terms of a generic influence graph (figure 3-2c) along with few examples.

Option Representation

For learning options to navigate between abstract states while avoiding replication, new options are built only when a variable with some context changes. The reasoning is that when a

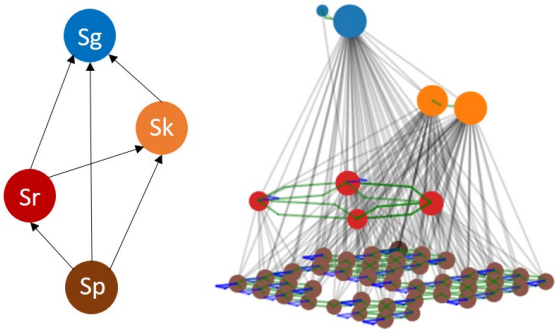

Figure 3-3: Learned state hierarchy in terms of state variable influence graph and state transition graph for a key-and-lock task (color-coded to represent corresponding state variables and their instantiations)

state variable having some context changes, these transitions depend on other variables and, hence, the need to create an option. Furthermore, to avoid replication, the learned options are uniquely identified using the changes induced in the problem domain. In options framework [30], an option O is formed using a tuple (I, μ, β), with the initiation set I where option O can be initiated, a policy function μ for option O where $\mu(s, a)$ gives the preference value given to an action a in state s and a termination function β such that $\beta(s)$ gives the probability of option O terminating in state s. In the proposed algorithm, building on the options formulation, each option is defined using following attributes:

- *Variable*: The state variable for which the option is defined.
- *Next variable*: The state variable over which the option is defined.
- *Change*: The change caused by the option in its *variable*.
- *Initiation state*: A set of initiation state of the option.
- *Termination state*: A set of termination states of the option.
- *Termination Initiation state*: A set of states from which a terminating (abstract) action can be initiated.
- *Actions*: Actions (primitive and abstract) over which the option is defined.
- *Value*: Option's value space used for defining its policy.

For an IG, an option O for a state variable S is defined over a state variable S_i using its contexts $\mathbf{S}^{IG}_{cont} = \{S, S_1, S_2, ..., S_k\}$. Such an option O is defined with $variable = S$ and $next\ variable = S_i$. Its *actions* are obtained either from the set of primitive actions available for S_i in IG if $S_i \in \mathbf{S}^{IG}_{cont}$ and $context(S_i) = \phi$ or from the set of abstract actions available for S_i in IG if $S_i \in \mathbf{S}^{IG}_{cont}$ and $context(S_i) \neq \phi$. Its *value* is defined over a cross product of states s such that $s \in State\ set$ of S_i and s connected to *initiation set* of option O in G i.e. $s \in context(initiation\ set) \cap domain(S_i)$, and its *actions*. The previous associations of state variables and primitive actions become important when defining *actions* for an option.

For example, in a key-and-lock problem, an option O for changing room variable Sr is defined over Sp. If option O is defined over all the primitive actions, then it would result in the number of actions $n_a = 5$ (*move* and *pickup*). However, due to the associations between state variables and primitive actions, option O can be defined using only *move* primitive actions i.e. $n_a = 4$. Furthermore, if *value* of option O is defined using the five primitive actions, then the result will no longer be *modular* as its value function involves a component associated with key variable. In the next problem instance, if instead of key variable, another state variable is introduced then option O will no longer remain viable.

A Generic Scheme for Option Discovery

A general scheme for iteratively constructing an option O defined for a state variable S undergoing a transition with context $\mathbf{S}^{IG}_{cont}$ is as follows:

1. Set the bottom-most variable in the context $\mathbf{S}^{IG}_{cont}$ as current variable S_{cur}. initiate previous option O_p, intermediate option O_i and intermediate variable S_i to *null*.
2. Learn an option O over S_{curr} for reaching the current instantiation of S_{curr}. If an option O_p exists, then add O_p to *sub action set* of S and to *actions* of option O.
3. If the primitive action a which caused the change in S belongs to *primitive action set* of S_{cur} and $context(S_{cur}) \neq \phi$, then define option $PO_i(S_{cur})$ encapsulating a. Add PO_i to *actions* of option O and *primitive sub action set* of S.
4. Set option O_p, if exists, as $O_i(S_i)$.
5. Set option O as O_p and current variable S_{cur} as S_i.
6. Set the next variable according to variable order VO in $\mathbf{S}^{IG}_{cont}$ as S_{cur} and go to step 2. If no next variable exists from S_{cur} then go to step 7.
7. Add option O to *action set* of S.

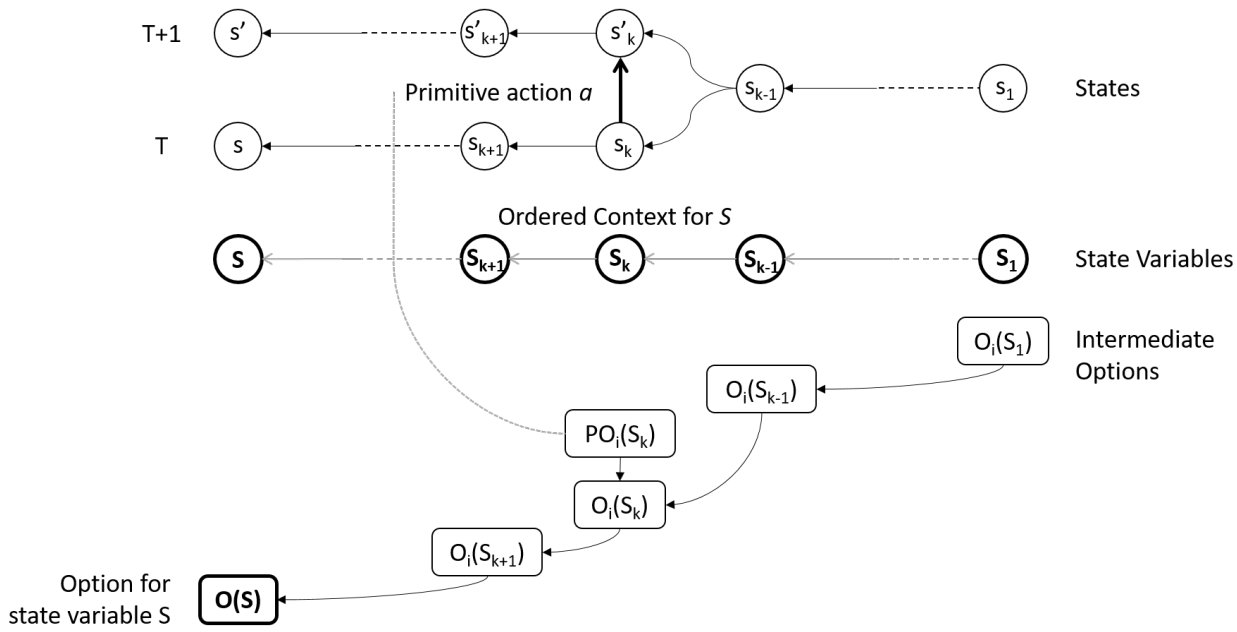

Figure 3-4: A graphical overview representing the construction of an option O for state variable S

A graphical overview of this iterative scheme is given in figure 3-4 wherein the final option $O(S)$ represents the abstract action available to an agent for changing variable S and all intermediate options $O_i(S_i)$ are the supplementary options created for option O. All the options defined in this process have *variable* $= S$ signifying their association with variable S while *next variable* attribute to be corresponding S_i over which they are defined. The option discovery happens while interacting with the environment and, hence, these newly discovered options need to be added to *action* of all options defined over variables Sh higher in hierarchy than S if $S \in \mathbf{Sh}^{IG}_{cont}$. For all the options constructed in the iterative process, *initiation set*, *termination set* and *termination initiation set* need to be defined and a simple rule is discussed ahead.

Initiation, Termination and Termination-Initiation conditions

In a general case, for defining *initiation set*, *termination set* and *termination initiation set* of an option O for a transition of state variable S_c from $s(c)$ to $s'(c)$, let the ordered context $\mathbf{Sc}^{IG}_{cont}$ of S_c to be $\{S_c, ..., S_{k+1}, S_k, S_{k-1}, ..., S_1\}$ such that $level(S_k) < level(S_{k-1})$ for all $S_k, S_{k-1} \in \mathbf{Sc}^{IG}_{cont}$. Let s' and s be the values of $\mathbf{Sc}^{IG}_{cont}$ at $T+1$ and T time-steps respectively with primitive action a which causes the change in S_c. The primitive action a may not belong to *primitive action set* of S_c and is assumed to be belonging *primitive action set* of S_k with $S_k \in S_{cont}$. This general case is shown in figure 3-4 and 3-5. For each intermediate option O_i defined over S_i,

- *Initiation set* is the values of all S_j such that $level(S_j) \leq level(S_i)$ at time instance T.
- *Termination set* is defined as the value of all S_j such that $level(S_j) \geq level(S_i)$ at time instance $T+1$.
- *Termination initiation set* is defined as the value of all S_j such that $level(S_j) \geq level(S_i)$ at time instance T.

Time	S_c	...	S_{k+1}	S_k	S_{k-1}	...	S_n
T+1	s'(c)	...	s'(k+1)	s'(k)	s'(k-1)	...	s'(n)
				↑ Primitive action *a*			
T	s(c)	...	s(k+1)	s(k)	s(k-1)	...	s(n)

Figure 3-5: A general case of a state transition for defining *initation set*, *termination set* and *termination initiation set* for an option O

A primitive action a is removed from a state variable's *change action set* when a transition is observed in the variable due to a. Furthermore, as a room can have more than one door to go to an adjacent room, more than one termination states may need to be defined for an option. When a transition same as the one defined in an option O is observed but for a different termination state, this new termination condition s^{new}_t is added to *termination set* of option O if s^{new}_t satisfies initiation conditions of any of the intermediate options of option O. Such a check is carried out from the intermediate option defined over the lowest to the top-most variable in the context of *variable* of option O. At a certain level in context of *variable* of option O, if s^{new}_t satisfies the initiation condition but doesn't satisfy the termination condition, then it is added to the termination set of the corresponding intermediate option and all other options constructed after it. If the initiation condition of an intermediate option is not satisfied, then a new intermediate option is created using a scheme similar to previous one except that the resulting option O_i is added to *actions* of next intermediate option and correspondingly termination states for all next intermediate options and option O are updated. Finally, initial definition of option O is updated to include this new way to change its *variable*.

Examples

In a key-and-lock problem, a room leaving option O_r is defined for Sr when a transition is occurred from room 0 to room 1 and having context $\{Sr, Sp\}$. Assume that this transition occurs due to primitive action *north* when Sp changes from 5 to 6. Hence, in the discussed option discovery scheme beginning with Sp, an intermediate option $O_i(Sp)$ is defined with *variable* as Sr, *next variable* as Sp, *initiation set* as $\{Sr : 0\}$, *termination set* as $\{Sr : 1, Sp : 6\}$, *termination initiation set* as $\{Sr : 0, Sp : 5\}$, *actions* as $\{north, south, east, west\}$ and *value* as a cross product of *actions* of $O_i(Sp)$ and *State set* of Sp. Now, as primitive action *north* belongs with Sp, O_i is assigned to the room leaving option O_r for Sr and is added to *action set* of Sr. The option O_r will have *change* attribute to be $[0, 1]$ as it changes Sr from 0 to 1.

For key collecting policy O_k with change of $\{Sk : 0, Sr : 1, Sp : 7\}$ to $\{Sk : 1, Sr : 1, Sp : 7\}$ for *pickup* action, starting with Sp, an intermediate option $O_i(Sp)$ is defined with *variable* as Sk, *next variable* as Sp, *initiation set* of $\{Sk : 0, Sr : 1\}$, *termination set* of $\{Sr : 1, Sp : 7\}$, *termination initiation set* of $\{Sr : 1, Sp : 7\}$, *actions* as $\{north, south, east, west\}$ and *value* as a cross product of *actions* of $O_i(Sp)$ and *state set* of Sp. Similarly, an intermediate option $O_i(Sr)$ is defined with *variable* as Sk, *next variable* as Sr, *initiation set* of $\{Sk : 0\}$, *termination set* of $\{Sr : 1, Sp : 7\}$, *termination initiation set* of $\{Sr : 1, Sp : 7\}$, *actions* as *action set* of Sr along with $O_i(Sp)$ and *value* as a cross product of *actions* of $O_i(Sr)$ and *state set* of Sr. Now, as *pickup* action belongs to *primitive action set* of Sk, a special primitive option PO_k^s need to be defined encapsulating *pickup* action. Hence, $PO_i(Sk)$ is defined with *variable* as Sk, *next variable* as Sk, *initiation set* of $\{Sk : 0, Sr : 1, Sp : 7\}$, *termination set* of $\{Sk : 1, Sr : 1, Sp : 7\}$, *termination initiation set* of $\{Sk : 0, Sr : 1, Sp : 7\}$, *actions* as *pickup* and *value* as 10 (predefined pseudo-reward) for *pickup* action for the corresponding state in *termination initiation set*. Finally, O_k is defined with *variable* as Sk, *next variable* as Sk, *initiation set* of $\{Sk : 0\}$, *termination set* of $\{Sk : 1, Sr : 1, Sp : 7\}$, *termination initiation set* of $\{Sk : 0, Sr : 1, Sp : 7\}$, *actions* as $\{O_i(Sr), PO_i(Sk)\}$ and *value* as a cross product of *actions* of O_k and *state set* of Sk. Here, as the primitive action belong with Sk, all the state variable before Sk are unchanged and only participate in the option discovery process as context for Sk. Thus, *termination set* and *termination initiation set* are the same for both intermediate options $O_i(Sp)$ and $O_i(Sr)$.

3-3 Action Selection Scheme

The state hierarchy construction and option discovery schemes work in a bottom-up manner while action selection scheme works in a top-down manner. In hierarchy construction, the goal variable is not given any extra significance. It can be considered as building up the knowledge for all the state variables in the problem domain. On the contrary, in action selection scheme, actions are taken to discover or reach the goal location. In the proposed scheme, it is assumed that changing the topmost variable in IG will get the agent to the goal location. If the goal is associated with any other state variable lying anywhere below the topmost variable in IG, then agent will still discover an extra part of the hierarchy not necessary for reaching the goal. Hence, in the thesis work, the goal variable is assumed to be the topmost variable. In the proposed algorithm, following action selection scheme is used:

1. If no option is currently being followed, then set the topmost node in IG as S_{cur} and go to step 2. If an option O is currently being followed then actions are to be chosen according to the policy of O. In such a case, set S_{cur} as *variable* of option O, current action a as option O and go to step 3.

2. Using a selection criterion defined with probability w, choose whether to slide the control down or to choose an action using available actions for S_{cur}. The probability of selecting to pass the control down is defined to be $(1-w)$ and that of selecting an action is w wherein w linearly increases from 0 to 1 as learning progresses. The choice of action selection is only possible if an action is available to be executed from S_{cur}.

   (a) The action selection pool for choosing an action is obtained from *primitive action set*, *action set* and *change actionset* of S_{cur}. An action a is randomly chosen from this pool. If the selected action is an abstract action, then go to step 3 else stop as primitive action is chosen.

   (b) For sliding the control down, the control of deciding an action is passed to the next variable in VO and S_{cur} is set to be the next variable. After passing the control to the next variable, repeat step 2. If no state variable exists after S_{cur} in VO, then select an action randomly from the primitive action set of the problem domain and stop.

3. For an abstract action a for state variable S_{cur}, apply a selection criterion defined using probability p for selecting whether the next action is to be selected using value function of a or using planning. The probability of selecting an action using value function of abstract action a is defined to be p and that of using planning is defined to be $(1-p)$ wherein p is kept constant throughout the learning phase.

   (a) When selecting next action using value function of action a, either ϵ greedy or Boltzmann exploration criterion is used. The selected action is assigned to a. If a is a primitive action then stop else go to step 1.

   (b) When selecting next action using planning, first carry out value iteration over value space of action a. If the termination state of action a is reachable from the current state, then select a_{next} using greedy policy (without any exploration) and set $a = a_{next}$, else select next action using value function of action a. If the selected action a_{next} is a primitive action then stop else go to step 1.

Hierarchical Policy

A *hierarchical* policy is a set of all policies for each sub-task. At a particular time instance T, an instantiation of hierarchical policy is represent using *Hpolicy* which details about abstract actions currently being followed at each level in hierarchy. Any option O for state variable S creates n intermediate options where n is the size of $context(S)$ such that each such option is defined over only one state variable. Due to the way options are created, at a particular time instance, at the most one option is being followed at each level in the hierarchy which simplifies the value update scheme. An option O is ended either successfully if an agent reaches the desired termination state or unsuccessfully if an action taken by an agent inducing an undesired change in the problem domain or violates the initiation conditions of option O or if duration of execution of option O exceeds a predefined limit. Once an option O is terminated, successfully or unsuccessfully, then option O and all other options invoked by O are ended and removed from $Hpolicy$. The control of deciding action is given to the lowest continued option in $Hpolicy$. If no such policy exists, then the action selection scheme is restarted from the root node.

When selecting action in RL methods, it is necessary to balance the trade off between exploration and exploitation. In the presented action selection scheme, the schemes implemented for addressing exploration and exploitation are discussed in the following sections.

3-3-1 Exploration

Exploration implemented in the current scheme is carried out for: a) discovering new states and transitions and b) learning an optimal policy. When following an option, action selection under the influence of its policy results in learning an optimal value surface. Furthermore, it may result in discovering new states due to exploration in the followed policy (ϵ greedy or Boltzmann exploration). In the current scheme, directed exploration is implemented for discovering new states by choosing an action from *change action set* for a state variable with some probability. As learning progresses, probability assigned for the directed exploration increases. As the goal is considered to be the topmost variable, as learning progresses, agent starts to frantically look for changing the top variable by executing primitive actions from its *change action set*. Once a transition caused due a primitive action a is observed in the topmost variable, action a is removed from its *change action set* and a goal policy is formed. In beginning phases of learning, with the discussed selection criterion, control is passed down to lower variables. As learning progresses, variables higher in hierarchy start executing primitive actions from their respective *change action set*. In later stages of learning, agent focuses more on changing the top variable if the goal is not yet reached else the goal policy is executed.

3-3-2 Exploitation

Similar to exploration, exploitation is addressed either: a) by following value function or b) through classical planning methods. Action selection can be done using value function (*value*) of the current option O with some exploration criteria (ϵ greedy or Boltzmann exploration). Such action selection scheme exploits already learned value function. Another way for exploiting the learned knowledge is by using classical planning techniques. Here, exploitation is achieved using value iteration method defined over the learned model (State transition graph G). The model learned through state transition graph is used to bootstrap the value function of option O. Starting from termination state of option O in the domain of its *variable* S, breadth first search (BFS) algorithm is implemented. As each option is defined over only one state variable and connecting edges between nodes in G encode the context transitions between states, considering only $domain(S)$ in G is sufficient for planning and selecting abstract actions. With the graph traversal, the corresponding Q values of states are updated in *value* for option O. This results in bootstrapping the value function and helping in faster convergence. An action is chosen according to the updated value function without any exploration probability resulting in a (sub) goal directed behavior.

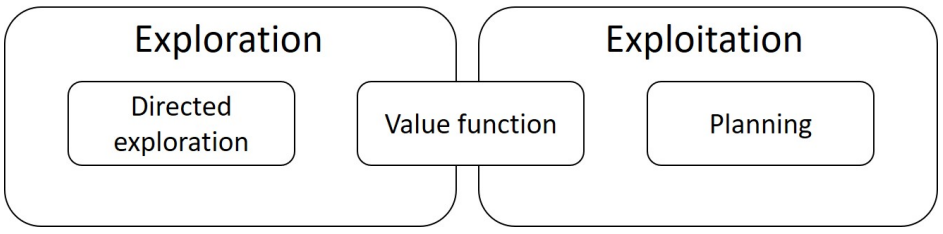

Figure 3-6: An abstract overview of exploration-exploitation trade off in implemented action selection schemes

Figure 3-6 gives an abstract overview of how exploration-exploitation trade off is handled in the implemented action selection scheme. The probability of directed exploration increases linearly as learning progresses and ends when a change is discovered. The probability of exploration when selecting actions based on value function decreases linearly with learning while no exploration takes place when selecting actions based on a plan. For exploitation, as probabilities of selecting actions using value function and planning are kept constant throughout the learning

phase, exploitation of learned value function increases as learning progresses. The probability of selecting actions using value function should be kept higher than that of using planning. The reasoning is that, with an intermittent planning a goal-directed behavior would be introduced without depriving the agent of exploration. Furthermore, with planning, values for all states of a policy are bootstrapped which further aid action selection at the next time instance. Another important reasoning behind having intermittent planning is that the environment model used may not be accurate enough and hence solely relying on it could be detrimental to the agent's performance.

3-4 Value Update Scheme

In the thesis work, value update for an option O is carried out using the SMDP theory. Furthermore, a value decomposition scheme inspired from the HEXQ and MAXQ methods is used. All the options except for the goal option are learned using pseudo-rewards obtained after their termination. The Q value of a certain state s and an abstract action a following a hierarchical policy π is given in the SMDP theory (equation 2-10) as,

$$Q^{\pi}(s,a) \quad = \quad \sum_{s',N} T(s,a,s',N)[R(s,a,s',N) + \gamma^{N} Q^{\pi}(s',\pi(s'))] \tag{3-1}$$

In the implemented value decomposition scheme, $R(s,a,s',N)$ represents only the reward obtained after the termination and does not include any accumulated rewards obtained during execution of abstract action a, similar to HEXQ. The value update scheme in the HEXQ approach is inspired from the MAXQ approach but differs on in the regards of how $R(s,a,s',N)$ is defined. However, Q values in an option are learned using the concept of pseudo-rewards from MAXQ approach. In the implemented value decomposition, the value of a next state s' used for update is defined to be $\max_a Q(s',a)$ when compared with that of the HEXQ method (equation 2-18). Thus, each option is learned independently without any consideration for its child or parent tasks. When an option is successfully terminated, a pseudo-reward of $+10$ is given and if it is unsuccessfully terminated because of violation of its initiation condition, then a pseudo-reward of -20 is given. The value function of an option O is only defined over its *next variable* such that its initiation conditions are satisfied.

Consider an option O constructed for manipulating a state variable S and defined over state variable S_n. A simple Q update scheme, similar to Q-learning, is used for updating a policy π of an option O and is given as,

$$Q^{\pi}(s,a) = (1-\alpha) * Q^{\pi}(s,a) + \alpha * (R(s,a,s',N) + \gamma^{N} * \max_{a} Q^{\pi}(s',a)) \tag{3-2}$$

where a represents an abstract action in *actions* of an option O, s and s' represents agent's states at T and $T+1$ time instance respectively, γ is the discount factor and α is the learning rate. For updating options defined over primitive actions, a simple Q value update scheme given in equation 2-13 is used. In the implemented Q update, the discount factor γ is kept constant through out the learning phase, however, the learning rate α is decreased linearly with the number times an option is invoked.

As discussed previously, *Hpolicy* maintains a list of all policies being followed at each time instance and is used for updating their value function. With each transition, termination conditions are checked for all options in *Hpolicy*. If a termination condition of an option satisfied, then the corresponding option is deemed to be successfully terminated. If an option's initiation conditions are no longer valid or the intended change has occurred but without satisfying its recorded termination conditions or its duration of execution exceeds a predefined limit, then

the option is terminated and the termination is considered to be unsuccessful. If an option is terminated, successfully or unsuccessfully, then all the options invoked by it are also terminated. All the terminated options are removed from *Hpolicy*. At each time instance, options which invoke primitive actions are updated. If an option is terminated successfully terminated then its parent option is also updated. Such a value update is carried out after the execution of a primitive action. However, value iteration is also carried out when selecting an action using planning which updates Q values of an option. It requires a model of the problem domain and is discussed in the following section.

3-4-1 Model-based Techniques

When selecting actions based on planning, value iteration is carried out over the value space of an option O currently being executed. In any planning problem, transition functions can either be deterministic or stochastic. Hence, agents needs to have an estimate of transitional probabilities for primitive and abstract actions. For value iteration, the transition probabilities of the domain of *next variable* of option O are learned using the state transition graph G. In the current scheme, the probability of a certain transition is obtained by counting the number of times a particular outcome occurs. As the policy of an option keeps on evolving in the learning phase, a discounting is implemented using discount factor γ to weigh the current information more than previous. This value iteration is used only as a mean to bootstrap value function and hence, an exact estimation of probabilities is not required. In the SMDP theory, for value update given in equation 3-1, a transition function defined using a joint probability over both next state s' and time of execution N is required for updating value function. For estimating probabilities, an approximate method, discussed in next section, is devised for both deterministic and stochastic domains. The approximate method directly updates Q values using the estimated joint probabilities.

Approximate Method for Value Iteration

In deterministic scenario, the estimation of a value associated with a state-action pair for an option O for state variable S is carried out by finding the shortest path between a state s and termination state s_t of option O and is implemented using a breadth first search (BFS) traversal of the domain of S. The value update scheme used for a state s is as follows:

$$V_{n+1}(s) = (1 - w) * V_n(s) + w * \gamma^N * P(s', N|a, s) * V_{n+1}(s') \tag{3-3}$$

where w is the weight parameter which linearly decreases with the number of times option O is invoked, γ is the discount factor, $P(s'|a, s)$ gives the probability of reaching next state s' in N time-steps after executing an action a in a state s. As the domain of S is traversed from termination state to other states, $V_{n+1}(s')$ is used for updating $V_n(s)$ wherein $V_n(s) = \max_a Q_n(s, a)$. Extending this scheme for stochastic problem domain requires an additional step. In stochastic domain, first, a shortest path to the termination state s_t is found using the most probable outcome of actions. Figure 3-7b shows an example of a simple domain with 9 states wherein dark edges represent most probable outcomes of an action and grey edges show outcomes with low probability. On the obtained shortest path, value update is carried out using equation 3-3. In the second step, the domain of S is traversed again using BFS along the most probable action outcomes, but the value update is only carried out for low probable incoming edges using the following update scheme:

$$V_{n+1}(s) = V_{n+1}(s) + w * \gamma^N * P(s', N|a, s) * V_{n+1}(s') \tag{3-4}$$

Figure 3-7 shows an example both deterministic and stochastic domains and the final optimal policy obtained. It can be observed that, in the approximate scheme, the second step only adds value in the stochastic domain and hence, it can be used for both the domains.

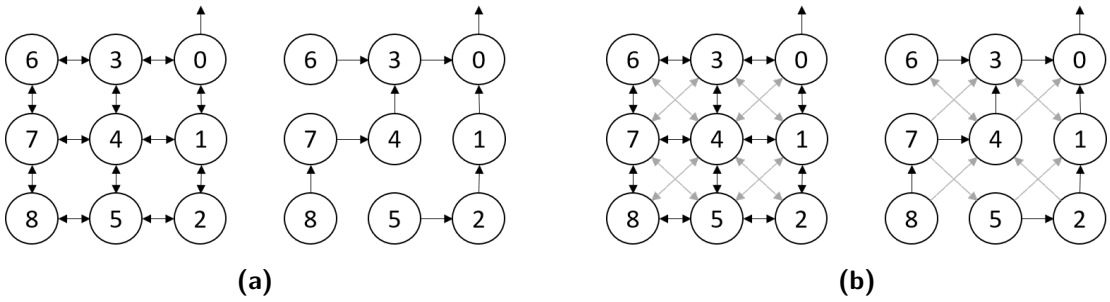

Figure 3-7: The domain of state variable S in state transition graph G with optimal policy for a) deterministic transitions and b) stochastic transitions (grey edges show the transitions with low probability of occurrence)

3-5 Pseudo-code

The proposed algorithm can be divided into: a) HRL scheme (Algorithm 1), b) Action selection scheme (Algorithm 2), c) Option discovery scheme (Algorithm 3) and d) Value update scheme (Algorithm 4). A brief overview and a pseudo-code of the proposed algorithm are discussed in the following sections.

HRL Scheme

The HRL scheme represents the main algorithm wherein the required preliminaries: *Influence graph IG*, *state variable influence graph SVIG*, *variable order VO* and *state transition graph G*, are computed and rest of the scheme are called upon. *IG* serves as the grounding framework for all the learned exit options and their value functions. *VO* is computed to have a loosely defined linear hierarchy for sliding the control down from the root node. The state-transition graph G is used primarily for constructing a compact hierarchy of the state variables in terms of their instantiations and to record their transitions. For both deterministic and stochastic transitions, G serves as a model of state space and can be used directly for planning.

The main HRL scheme shown in algorithm 1 gives an overall flow of the proposed algorithm. In the beginning, all the essential parameters are computed. Each trial can be considered to be divided in four steps: 1) Primitive action selection, 2) Action execution, 3) Inferring from resulting environmental changes which involves option discovery, the hierarchy construction etc. and 4) Value function update. Each of these steps, except action execution step, calls for a routine, given in-detail in later algorithms. Step 3 checks for transitions in state variables and compares them with *IG* for building or updating exit options, while step 4 considers active options for updating their value function. Step 3 also updates the state-transition graph G along with the addition of newly formed options to higher state variables. The *initiation set* of an option O is updated over the newly discovered states in step 3 (line 10 and 34 in algorithm 1). Similarly, *value* of an option O is updated to include newly discovered states and the new state values are initiated to zero (line 18 in algorithm 3).

Algorithm 1: General Scheme for HRL+Planning

Data: Env, Dynamic Bayesian Network (DBN) Model

Initialization
Compute state-variable influence graph ($SVIG$) using DBN model
Compute influence graph (IG) using DBN model
Compute variable order (VO)
Initiate multi-directed state-transition graph (G)

while $N\_episodes > 0$ **do**
    Environment reset
    $S_{cur} = Sg$; `// Control is given to the root node`
    Initiate policy value function over (newly discovered) states
    while *($N\_steps > 0$) and (goal not reached)* **do**
        Compute the graph representation g_{rep} for current state s

        `// Action selection`
        Compute weight w for exploration
        $a = 0$; `// Initiate action` a
        while *a is not in primitive action set* **do**
            if *Any option a being followed* **then**
                $(a, S_{cur}, Hpolicy) = Action\ selection(g_{rep}, S_{cur}, w, Hpolicy, a)$
            else
                if *(Selection criterion) and (action is available)* **then**
                    $(a, S_{cur}, Hpolicy) = Action\ selection(g_{rep}, S_{cur}, w, Hpolicy)$
                else
                    S_{cur} = Variable next to S_{cur} in VO
                end
            end
            if *No next variable in VO after S_{curr}* **then**
                Random action selection from the primitive action set
            end
        end

        `// Action execution`
        $sn = take\ action(s, a)$
        Compute the graph representation gn_{rep} for current state sn

        `// Checks and updates`
        Initiate policy value function over (newly discovered) states
        $G\ graph\ update(gn_{rep})$; `// Adding new state to` G
        $Option\ discovery(g_{rep}, gn_{rep}, a)$

        `// Value Update for HRL`
        $(S_{cur}, Hpolicy) = Value\ update(g_{rep}, gn_{rep}, a, S_{cur}, Hpolicy)$
    end
end

Algorithm 2: Action selection scheme

Data: current state var S_{cur}, state g_{rep}, weight w, *Hpolicy*, current policy *pol*
Result: S_{cur}, primitive action, *Hpolicy*

Identify the executable action set A from current state g_{rep} for S_{cur} in IG
$no\_action\_flag = False$; `// Availability of action for current state`
if *No action available* **then**
  $no\_action\_flag = True$
end
$no\_plan\_flag = Flase$; `// Availability of plan to termination`
if *pol* **then**
  a = pol
else
  $a = random\ choice(A)$
end

while *(a is not in primitive action set) and (not no_action_flag)* **do**
  $a_{next} = []$
  if *(Selection criterion) or (no_plan_flag)* **then**
    `// Action selection based on value function`
    $a_{next} = \epsilon$ greedy action selection for g_{rep} using value function of a
    if *No action available* **then**
      $no\_action\_flag = True$
      Break
    end
  else
    `// Action selection based on planned path in current value function space to the termination state`
    Value iteration on value space of a using G
    if *termination reachable from* g_{rep} *in* G **then**
      $no\_plan\_flag = True$
    else
      $a_{next} = \arg\max_a(Q^a(g_{rep}, a))$ where $a \in A$
    end
  end
  $a_{next} = a$
  Update *policy_count* for a ; `// Number of times a policy is invoked`
end
if *no_action_flag* **then**
  $a = Nothing$
else
  Update IG, , S_{cur}, *Hpolicy* with actions being followed at each var level
end

Action Selection Scheme

In the proposed action selection scheme (line 14-29 in algorithm 1), if any option is being followed, then actions are selected according to it. If no option is currently being followed then the selection is handled by breaking decision making process as: *do nothing* and *do something* at a current state variable S_{cur}. Initially, S_{cur} is set as the goal variable or root node. When *do nothing* is chosen, S_{cur} is updated to the variable next in VO and if there are more than one variable at the next level, then selection is done randomly among them. If *do something* is selected, then an action is selected randomly (line 10 in algorithm 2) or with some heuristics from the *action set*, *primitive action set* and *change action set* of state variable S_{cur}. For the goal variable, when the goal is reached for the first time, a goal option is created and the corresponding primitive action is removed from *change action set*. Hence, in *do something* case, algorithm always selects either the goal option or a primitive action from *change action set* at the root node. Furthermore, when the goal option is selected, it will be terminated only when the goal is reached, hence giving more emphasis to the goal directed behavior in the later stages of learning. The lowest state variable in VO will not have any abstract actions and, hence, when S_{cur} is the lowest variable, then an action is chosen randomly from its *primitive action set*.

Option Discovery Scheme

The action selection scheme depends highly on how options are put together. Each variable higher in hierarchy has some preconditions which need to be satisfied when trying to change the variable. These are tagged as *context* for the changed state variable Sc and are obtained using predecessors of the changed variable in IG. When building an option O_{Sc} for changing a state variable Sc, options are build to reach *context* and are combined to form O_{Sc}.

The implemented option discovery scheme provides an ability to independently learn options, resulting in modularity. Hence, for the next task, if the goal variable is different, policies defined for a state variable lower in hierarchy can directly be used as they are learned without having any dependence on previous goal variable. In the proposed scheme, any option is uniquely defined by the change it induces on its state variable. If a certain change is not dependent of a variable but is shown to be in the provided DBN model, then, the counter instance, when observed, is still added to the same option (line 22-37 of algorithm 3). This routine is also useful in identifying new termination states for an option.

Value Update Scheme

In the implemented value update scheme given in algorithm 4, Q values are updated according to the checks of termination conditions. Options may be terminated successfully or successfully as discussed in section 3-4. Parent options of successfully terminated options are updated. Any option which invokes primitive actions is updated at each time instance. A simple Q update is carried for value update (line 15 of algorithm 4) where N is the duration of execution of an option O ($N = 1$ for primitive actions).

Algorithm 3: Option discovery scheme

Data: state g_{rep}, next state gn_{rep}, primitive action a

Check changes in any state variable from g_{rep} to gn_{rep} using IG
Order changed variables sc in ascending order according to VO
Identify context for each changed variable

for *Changed variables* Sc **do**
  if *Change in* Sc *is new* **then**
    $context$ = context of Sc in IG
    Initiate exit option O_{Sc} for observed change in Sc
    Initiate option O_p to *null*
    while *context is not empty* **do**
      $Sd = pop(context, 0)$
      Create option O for Sd to reach the associated context
      Assign initiation and termination criterion with option O
      Define value function using *state set* and *action set* of Sd
      Add O_p to the action set of the newly formed option O
      $O_p = O$
    end
    $O_{Sc} = O_p$
    Update *action set* of Sc
    Update value function of higher level options using the updated *action set* of Sc
  else
    O_p = Option having same change in Sc
    Initiate option $O_{S_{cur}}$ to *null*
    for S_{cur} *from ordered context(Sc)* **do**
      a = Intermediate option corresponding to S_{cur} defined in O_p
      if g_{rep} *satisfies initiation set of* a **then**
        if gn_{rep} *does not satisfy termination set for* a **then**
          Add $O_{S_{cur}}$ to *action set* of O_p and set option $O_{S_{cur}}$ to *null*
          Update *termination set* of O_p using gn_{rep}
          Update *termination Initiation set* of O_p using gn_{rep}
        end
      else
        Define option $O_{S_{cur}}$ for S_{cur} to reach the associated context
        Assign initiation and termination criterion with option $O_{S_{cur}}$
        Define value function using *state set* and *action set* of S_{cur} for option $O_{S_{cur}}$
      end
    end
  end
end

Algorithm 4: Value Update scheme

Data: state g_{rep}, next state gn_{rep}, primitive action a, *Hpolicy*

1 $O_{up} = null$; `// set of options eligible for value update and corresponding terminating child options`
2 **for** O *in Hpolicy* **do**
3 **if** *Termination criteria are satisfied for* O **then**
4 Terminate O and options invoked by O
5 Update *Hpolicy* to reflect the termination
 `// If` O `is terminated, then abstract action which invoked` O `is eligible for value update`
6 $O_{up}.append(parent(O), O)$
7 **end**
8 **if** *a belongs to action set of* O **then**
9 $O_{up}.append(O, a)$
10 **end**
11 **end**
12 **for** Op, O *in* O_{up} **do**
13 Compute learning rate α using *policy_count* of O
 `//` Q`-function update for option` Op
14 $Q^{Op}(g_{rep}, O) \leftarrow \alpha * Q^{Op}(g_{rep}, O) + (1 - \alpha) * (reward + \gamma^N \max_o Q^{Op}(gn_{rep}, O))$
15 **end**

3-6 Extensions for Different Variable Ordering

The proposed scheme is discussed for a problem domain with a linear variable ordering. But it can be easily extended to two other cases of variable ordering: a) interdependence of state variables and b) partial order planning case. The extensions required are as follows:

- **Interdependence of variables case**: A set of interdependent state variables $\mathbf{S} = \{S_1, S_2, ..., S_k\}$ in *IG* can be identified using strongly connected component algorithm. A new state variable V can be constructed by merging the interdependent variables $\mathbf{S}$ such that each different $\mathbf{S}$ vector is mapped onto a discrete number space $\mathbb{Z}$. In *IG*, the new variable V replaces $\mathbf{S}$ inheriting their connections with remaining state variables. With the new variable V replacing $\mathbf{S}$, a linear variable ordering can be obtained.
- **Partial order planning case**: In partial order planning case, two or more state variables from context of a state variable are independent of each other. In such a case, variable order *VO* obtained from *IG* will have more than one state variable at a particular level. The proposed extension for the implemented action selection scheme is that a state variable is to be chosen randomly from next level (line 10 in algorithm 2) to give the control to when sliding down in *VO*. For option discovery scheme, when option is to be constructed using independent state variables, a cross product of their combined states and actions should be taken to define the value function.

With these extensions, any general variable ordering and influence graph can be solved using the proposed algorithm. Furthermore, with the extension proposed for partial order planning case, an agent can even start without any knowledge of causal relations by defining value function a cross product of all states and actions, similar to a *flat* RL agent, and then learn the causal structure by interacting with the environment. Hence, in the worst case scenario, the proposed algorithm will perform equivalent to a *flat* RL learner.

Chapter 4

Evaluations and Discussion

The chapter briefly discusses the problem domains and experimental setup used for evaluating and comparing the performance of the proposed algorithm with that of HEXQ, *flat* and Model-based RL algorithm. The simulation results from the problem domains are presented and discussed in the following sections.

4-1 Problem Domains

In HRL literature, various different problem domains like taxi domain, four-room, playroom or robot navigation task are used for evaluating the performance of the proposed methods. In the thesis work, two tasks: *four-room* and *key-and-lock* task, are used for evaluating and comparing the proposed algorithm with other RL methods. The following section introduces these two tasks in brief.

4-1-1 Four-Room Task

In the four-room task shown in figure 4-1, the goal is to leave the room by the doorway present in North-West direction (indicated by an arrow), while the black oval gives one of the agent's starting locations. The problem environment is fully observable and agent can sense the room it is in and its position inside that room (i.e. factored state representation). The agent has four *move* actions: $\{north, sount, east, west\}$ to move along the four directions. The underlying MDP is representing using state variables $\{Sg, Sr, Sp\}$ where Sg indicates the goal variable representing whether the goal is reached by an agent or not, Sr indicates room variable representing the current room occupied by an agent and Sp indicates position variable representing the position of an agent in the current room. In a deterministic domain, an agent moves in the corresponding direction after execution of an action with probability 1. In stochastic domain, an actions moves an agent with 0.8 probability of moving in intended direction and 0.1 probability to slip into adjacent states. It receives 0 reward for each action and 10 for achieving the goal. Also, if the agent is to move into a wall, then it will remain in the same state. Figure 4-1 shows the DBN model and the problem domain for a four-room task.

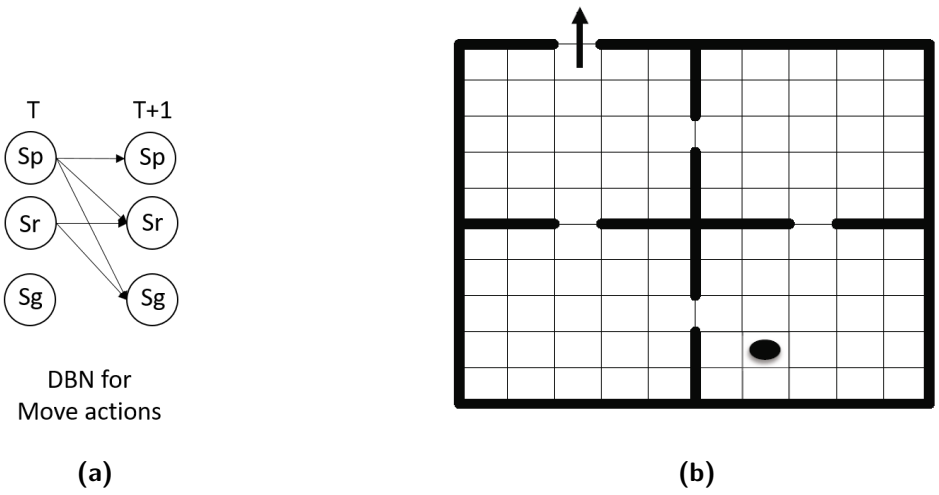

Figure 4-1: The four-room task with: a) its DBN model and b) its problem domain [1, ch. 9]

4-1-2 Key-and-Lock Task

The key-and-lock task shown in figure 4-2b is an extension of the four-room task. In this domain, few keys are added to a four-room problem domain. A key-and-lock problem domain has an extra variable *Sk* representing whether an agent has a key or not and an added primitive action: key *pickup*. The goal location is defined randomly in the problem domain. In this problem domain, for reaching the goal, an agent has to first collect a key available at certain locations in the domain and then navigate to the goal location. The transition probabilities are the same for the four *move* actions. For the added *pickup* action, the probability of collecting key at key locations is given to be 1 in deterministic scenario and 0.9 in stochastic scenario with 0.1 probability of an agent failing to collect the key. The reward function is kept the same as that of the four-room task wherein an agent receives a reward of 10 for successfully reaching the goal and 0 for taking actions. Figure 4-2 shows the DBN model and the problem domain for a key-and-lock problem wherein locations where keys are represented by small signs. The problem domain can easily be changed to include more number of rooms or keys resulting in increase of problem state dimensions and, thus, is used for comparing performance of RL and HRL methods. Figure 4-2 shows one of such problem instance with four rooms and 3 keys with its DBN model.

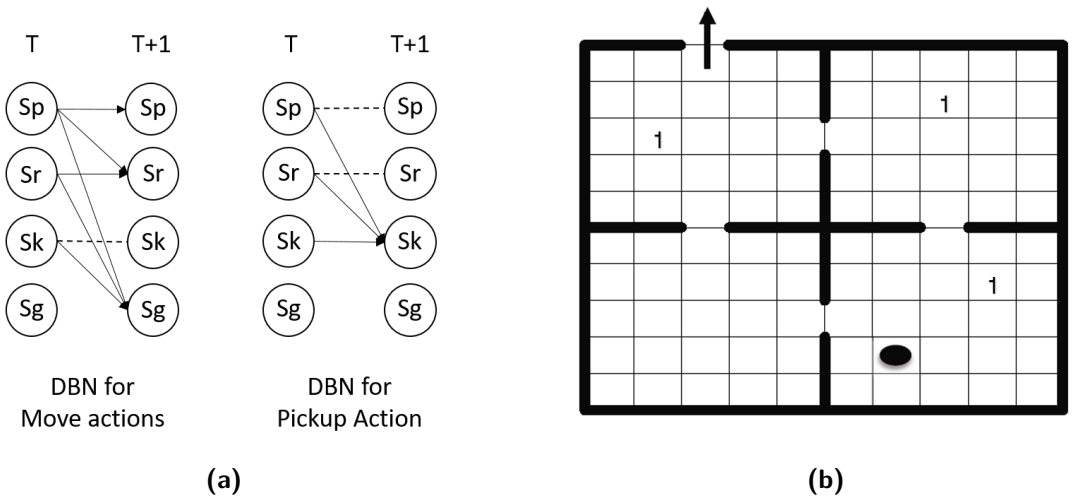

Figure 4-2: A key-and-lock task with: a) its DBN model and b) its problem domain

Both the methods are used for comparing the performance of the proposed algorithm with other RL methods. The experimental setup used in for carrying out the simulations is discussed in the following section.

4-2 Experimental Setup

In this section, the performance measure used to quantify the convergence of value surface and the experimental procedure is discussed and finally, hypotheses are presented.

Performance Measure

The performance of any RL method for a goal-directed MDP is evaluated by quantifying the convergence of the value surface. This convergence can be quantified in two ways: a) by measuring the values of certain *test* states in the problem domain at the end of each episode or b) by measuring the number of time-steps taken by an agent to reach the goal starting from certain *test* states in the problem domain when following the learned policy at the end of each episode. Here, a learning episode is considered to be made up of a certain number of time-steps and the performance of an agent is evaluated at the end of each such learning episode.

In the thesis work, the implement value decomposition scheme does not consider extrinsic rewards for variables lower in hierarchy. The values of states in an option are defined using the pseudo-rewards with respect to their own termination condition without any consideration of the overall goal task. Furthermore, a value of a state s in the goal policy learned using given extrinsic rewards represent abstracted or averaged values of all the state underlying state s and hence, such values cant be used to compare the evolution of the goal policy. Hence, for evaluating the performance of the proposed algorithm, the second measure is used wherein at the beginning of experiment few test states are randomly selected from the problem domain. At the end of each learning episode, the time-steps taken by an agent starting from these test states to reach the goal state following the learned goal policy is computed and used to measure the convergence of the learned policy. As the learning progresses, the time-steps taken by an agent for reaching the goal are expected to decrease and converge to a certain value.

Experimental Procedure

This section discusses the experimental procedure used for simulations along with used problem domain and algorithms. For evaluating the performance of the RL algorithms, the problem domains discussed above: the four-room task and the key-and-lock task are used with both stochastic and deterministic transition functions. An implemented learning routine is defined using $N_{episodes}$ number of learning episodes each having T time-steps. As the actions selected by an agent are partly random, to get an estimate of variance in the evolution of the goal policy, 25 such learning routines are used each with $N_episodes$ episodes and T time-steps. For each such episode the performance measure is obtained using a fixed set of test states. The time-steps taken for different test states is averaged, representing an overall score obtained by the policy. In the experiments carried out, the number of test states are assigned to be the dimension of position variable Sp in both the problem domains and the test states are chosen randomly. Hence, this score represents how effectively the goal policy is learned is different regions of the state space. Median of this score is taken across the learning routines to represent the most probable policy score and outliers present are removed by using only data lying between 25^{th} and 75^{th} percentile. All the experiments are carried with a discount factor γ of 0.98.

The proposed algorithm is compared against HEXQ, flat and model-based RL. In HEXQ, initially random actions are taken to obtain a variable ordering and using it, a hierarchy is constructed. As the proposed algorithm starts with a DBN model of the problem domain, for *fairer* comparison, a linear variable ordering is given to HEXQ. The implementation of HEXQ is carried out by giving the proposed algorithm without planning this linear variable ordering resulting in an augmented HEXQ implementation. However, for a simple linear hierarchy, both the algorithm are theoretically similar. Also, as key-and-lock task involves a parallel hierarchy, such a comparison is carried out in a four-room task only. Furthermore, in HEXQ, only abstract actions are available to the agent at higher level and these abstract actions are mainly the room leaving policies. Hence, for successful completion of task, the goal is set as one of terminating states of these abstract actions i.e. doorway states. A simple implementation of *flat* RL is used wherein an agent follows ϵ greedy policy when learning with ϵ linearly decreasing as the learning progresses. Model-based RL is a modified version of *flat* RL where the result each primitive action is simulated and primitive action which leads to a predicted state having highest Q value is chosen. For the proposed algorithm, initially various different values of p, probability of taking actions based on planning, are chosen and their performance was compared. The goal policy for all the methods during testing is obtained by following greedy policy.

Hypotheses

The thesis work is aimed to propose a novel scheme for learning inherent hierarchies in a generic problem domain to address the increased problem dimensions. The expected outcomes of the comparison of performance for the above mentioned algorithm are as follows:

Hypothesis 1. A lower value of probability $(1 - p)$ for selecting actions based on planning is better due to increased exploration in different problem domains.

As discussed in section 3-3, lesser depends on planning results in more exploration and faster goal discovery along with reduced impact of an insufficiently learned model of the problem domain. However, in both stochastic and deterministic problem domains, intermittent planning induces a goal-directed behavior leading to faster convergence of value surface.

Hypothesis 2. Without any planning, the proposed algorithm performs similar to HEXQ learner. The proposed algorithm with planning outperforms HEXQ learner.

As the in a simple linear variable case, the proposed scheme and HEXQ are theoretically similar and hence are expected to perform alike. However, with planning, the proposed algorithm should converge faster.

Hypothesis 3. The performs of flat and model-based RL deteriorates with increase in problem state dimension while the proposed algorithm outperforms both the methods.

As the problem state dimensions increase, the search space for solutions using RL methods exponentially increases. With the introduction of hierarchy, the proposed scheme is expected to outperform both the RL methods.

Hypothesis 4. The proposed algorithm handles increase in state dimensions with greater ease than RL methods and works with a generic problem domain.

As the proposed algorithm uses an influence graph as its basis and in such a case, added state dimensions directed get added to the domain of corresponding state variable. Extra efforts are only required to learn the additional abstract actions over its context rather than multiplying with all the other state variables. As an influence graph is used as a basis for all the routines in the proposed algorithm and defining such an influence graph using the DBN model of an FMDP is fairly simple, the proposed algorithm is expected to work with any generic problem domain with discrete states and represented using an FMDP.

In the following section, the results of simulations are presented and discussed. Initially findings regarding the probability $(1-p)$ for selecting actions based on planning are argued and then, comparison is carried out between the proposed algorithm and HEXQ. Finally the results of increased dimensionality on the proposed algorithm and flat RL methods in both deterministic and stochastic problem domains are evaluated.

4-3 Evaluations and Discussion

The following simulations were used to test out the hypotheses:

Experiment 1

For evaluating the impact of the probability $(1-p)$ for selecting actions based on planning, a comparison of the proposed algorithm with various different value of p was performed. The probability values used were: $(1-p) = \{0.9, 0.8, 0.65, 0.5, 0.25, 0\}$ where $p = 1$ results in the proposed algorithm without any planning phase. A simulation was carried out for a one-key three-room key-and-lock problem with deterministic transitions and the size of grid-world (state dimension of Sp) being 10×10 for 250 episodes each going for 400 time-steps, unless the goal is reached. The result obtained is shown in figure 4-3 wherein the 25^{th} to 75^{th} percentile data is shown in shaded region.

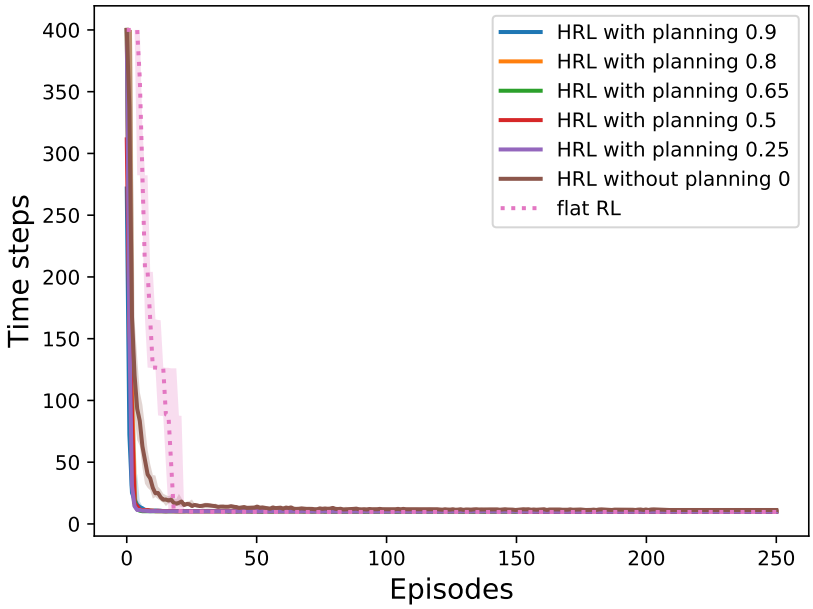

Figure 4-3: A comparison plot of the performance of the proposed algorithm with different values of the probability $(1-p)$ for selecting actions based on planning in one-key three-room key-and-lock problem (with the numeric values in plot legends showing different probability $(1-p)$ values)

No significant difference in the performance was observed in figure 4-3. Hence, further experimentation was carried out for different size of the same problem and obtained results are shown in figure 4-4. No significant performance difference was observed below the probability value of $p \leq 0.75$ and hence the hypothesis 1 was rejected. Thus, in the next experiments, the choice between action selection using value function or planning was carried out randomly i.e. with a probability value of 0.5.

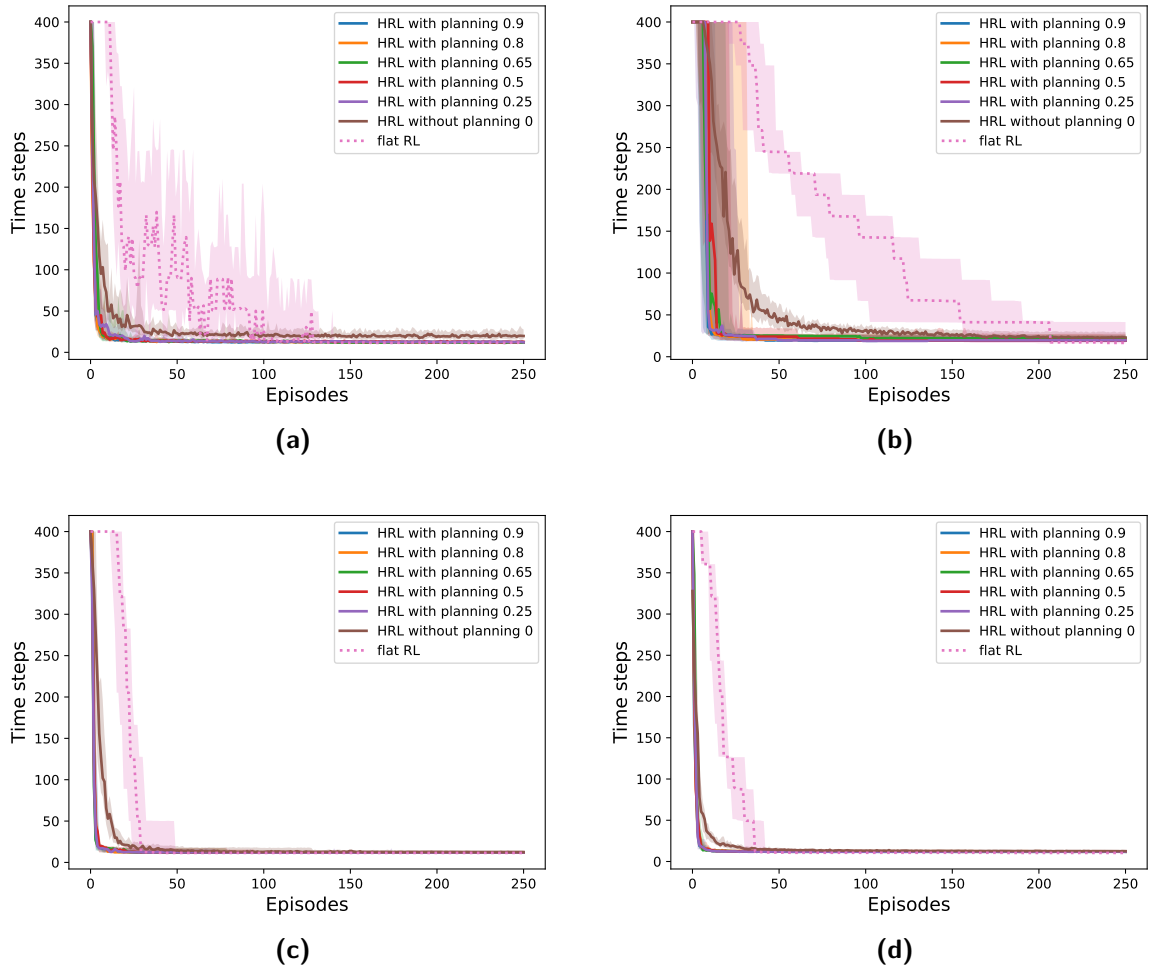

Figure 4-4: A comparison plot of the performance of the proposed algorithm with different values of the probability p for selecting actions based on planning in: a) one-key three-room key-and-lock problem with 10×10 grid-world having stochastic transitions, b) one-key three-room key-and-lock problem with 15×15 grid-world having deterministic transitions, c) two-key three-room key-and-lock problem with 10×10 grid-world having deterministic transitions and d) one-key five-room key-and-lock problem with 10×10 grid-world having deterministic transitions (with the numeric values in plot legends showing different probability p values)

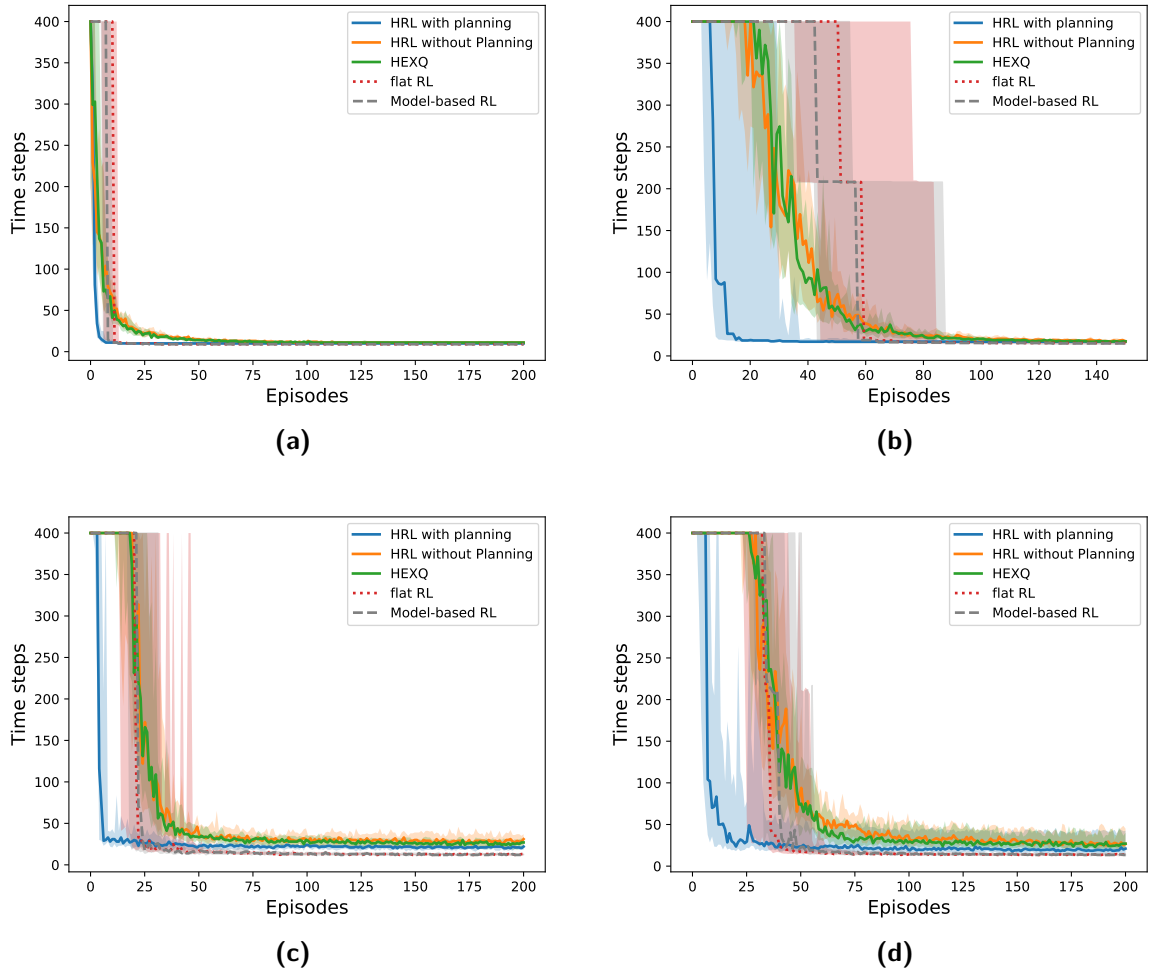

Figure 4-5: A comparison plot of the performance of the proposed algorithm with HEXQ, flat and model-based RL learner in: a) a four-room problem with 15×15 grid-world having deterministic transitions, b) a four-room problem problem with 20×20 grid-world having deterministic transitions, c) a four-room problem problem with 15×15 grid-world having stochastic transitions and d) a four-room problem problem with 20×20 grid-world having stochastic transitions

Experiment 2

The comparison of the proposed algorithm with HEXQ was carried out in different four-room environments and the obtained results are shown in figure 4-5 (All experiments carried out for $N_{episodes} = 200$ and 400 time-steps). Figure 4-5 shows that the performance of the proposed algorithm without planning is similar to that of HEXQ learner in both deterministic and stochastic domains and the proposed algorithm with planning outperforms both the methods. Hence, hypothesis 2 is accepted.

Experiment 3

For comparing performance of proposed algorithm and RL methods, a simulation was carried out in a one-key three-room key-and-lock problem with size of the grid-world as 10×10 and results are shown in figure 4-6. Figure 4-6 shows that proposed algorithm with planning outperforms flat and model-based RL learner converging very quickly to its optimal solution (the optimality of solutions is discussed further in-detail). For evaluating the performance of all these algorithms

as the problem dimensions increase, various experiments for increasing each state dimensions were carried out.

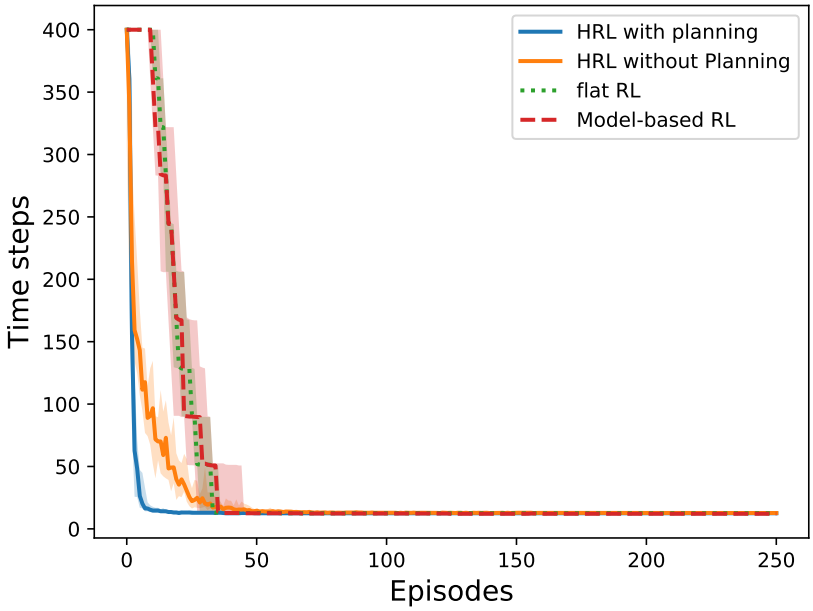

Figure 4-6: A comparison plot of the performance of the proposed algorithm with flat and model-based RL in one-key three-room key-and-lock problem

Various experiments carried out in different problem domains are given in figure 4-7. For all experiments, the number of episodes $N_{episodes} = 250$ and maximum time-steps $T = 400$ was used along with discount factor $\gamma = 0.98$. Each state variable in a key-and-lock problem was increased for both deterministic and stochastic problem domains. Figure 4-7 shows the degradation in the performance of RL methods with increase state dimensions. However, the proposed algorithm with planning still manages to reach the optimal value surface quickly. Hence the hypothesis 3 is accepted. Still, as the problem state dimensions increase, the number of episodes taken by the proposed algorithm to converge increases. Such an increase can be observed in figure 4-7a where the dimensions of variable *Sp* are increased. In other figures, the increase in the time taken for convergence by the proposed algorithm is not so significant because, the increase in the value space for a hierarchical policy is smaller compared to that of RL methods. With the addition of an extra room in figure 4-7b or an extra key in figure 4-7c only adds an extra value space to the hierarchical policy while, in RL methods, an addition of an extra variable resulting in multiplying the previous value space. Hence, the curse of dimensionality is not fully solved but addressed to a degree using the proposed HRL scheme.

Optimality

The solutions obtained from the proposed scheme are recursively optimal and a weak proof for it is discussed ahead. A given policy π is recursive optimal if π is locally optimal given its child policies [28]. The implemented value decomposition scheme defines value using pseudo-rewards. The Q values for options are learned over its *next variable* using equation 3-2 which treats abstract actions simply as extended actions by discount the future reward appropriately. Hence, Q values learned are optimal for reaching the termination state of the corresponding option given its actions i.e. its policy is locally optimal given its child abstract actions. Hence, the obtained solutions are recursively optimal. Furthermore, the solutions of HEXQ are recursive

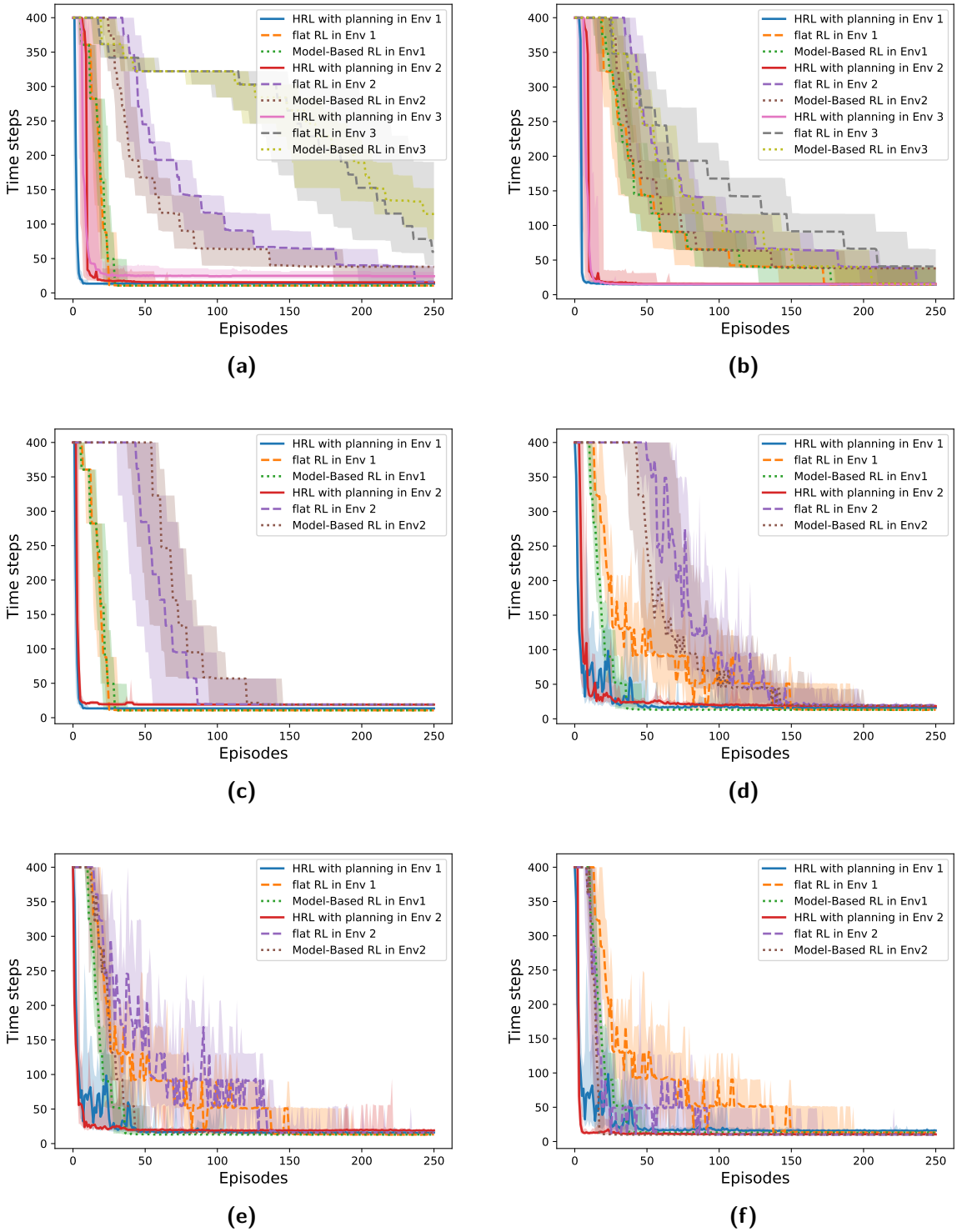

Figure 4-7: A comparison plot of the performance of the proposed algorithm, flat and Model-based RL learner different environments as: a) for one-key three-room key-and-lock problem with grid-world size as 1) 10×10, 2) 15×15 and 3) 20×20 with deterministic transitions, b) for a grid-world with size 15×15 and deterministic transitions in 1) one-key two-room 2) one-key three-room and 3) one-key five-room key-and-lock problem, c) for a grid-world with size 10×10 and deterministic transitions in 1) one-key two-room and 2) two-key three-room key-and-lock problem,
d) for one-key three-room key-and-lock problem with grid-world size as 1) 10×10 and 2) 15×15 with stochastic transitions, e) for a grid-world with size 10×10 and stochastic transitions in 1) one-key three-room and 2) one-key five-room key-and-lock problem, c) for a grid-world with size 10×10 and stochastic transitions in 1) one-key two-room and 2) two-key three-room key-and-lock problem

optimal [17]. As the implemented value decomposition is modeled after that of HEXQ's, hence, the recursive optimality of learned solutions is supported.

As discussed previously, recursive optimality is weaker than that of global optimality. Hence, the converged value of time-steps obtained by RL methods will be smaller than or equal to the proposed scheme as seen in figure 4-7. This is attributed to the abstractions increased in the problem domain, as with state abstraction, some state information is lost.

In the proposed algorithm, with the use of influence graph, the need of having both primitive and abstract actions available to an agent is addressed as all primitive and abstract actions are associated with corresponding state variable in the influence graph. When a certain transition is observed, an option is created for it and added to the influence graph and when selecting an action at a certain level in the hierarchy, action selection pool is formed using both *primitive action set* and *action set*. However, due to the association of actions with state variables, the choice available to an agent is decreased. Furthermore, *IG* can be inferred for any generic problem domain formulated as a goal-oriented MDP with discrete time-steps and states, represented as an FMDP using its DBN model, without any restrictions on its environmental features or predefined heuristics. *IG* serves as a basis for the proposed algorithm, for all such generic problem domain, a solution can be obtained given an accurate DBN model. Hence, the hypothesis 4 is accepted. This is supported by that converged solutions obtained for the discussed problems having different problem domains.

Chapter 5

Conclusion

The main difficulties associated with reinforcement learning are about the scalability of available methods and number of samples required for such methods. Both these problems can be addressed using hierarchical reinforcement learning. To make the most out of HRL paradigm, it is advantageous to use both spatial and temporal abstractions in tandem. There are two ways to construct hierarchies using these abstractions: a) by learning temporal abstractions in terms of abstract actions and then, using it, reorganizing the state space, or b) by re-structuring the state space using a known or learned environment model and learn abstract policies to navigate in it. The second approach considers the inherent hierarchies present in the state space and agrees with the conclusions of Simon [3], Utgoff and Stracuzzi [4] that the knowledge hierarchy of an agent evolves starting from simpler building blocks to a more complex structure. Once the knowledge hierarchy is built by an agent, it then can learn the ways to navigate this structure.

In the thesis work, a novel method for learning a hierarchy in discrete-time factored MDP domain using the DBN model is proposed. For constructing state hierarchies, a scheme inspired from HEXQ [17] and VISA [16] is devised. The state hierarchy is constructed along with a model of the problem domain and then, options are learned to navigate this constructed hierarchy. A generic option discovery scheme is proposed and a complementary value update scheme similar to HEXQ [17] is used. An action selection scheme which efficiently combines actions selection based on the learned value function and classical planning techniques also is formulated. The basic principle followed is that a state hierarchy is built from bottom-up while reasoning flows in a top-down manner. The main contribution of the research work lies with the proposed scheme for the hierarchy construction and the option discovery. The proposed algorithm addresses the research questions using this hierarchy construction and option discovery scheme along with the action selection scheme. The recursive optimality of the obtained solutions is argued, but no guarantees about the global optimality of the final solution can be given.

The hypotheses put forth regarding the performance of the proposed scheme when compared with HEXQ and other RL methods shown to be correct using the discussed experimental results in deterministic and stochastic problem domains. These results also help in concluding that the proposed HRL algorithm with planning outperforms HEXQ and *flat* RL methods. Furthermore, the discussed experimental results show that, with the increase in the size of the state space for the problem domain, the performance of flat and model-based RL deteriorates. The proposed algorithm, however, converges rapidly and hence, easily handles the added state dimension, alleviating the *curse of dimensionality* to some degree.

5-1 Future Directions

The proposed algorithm, as concluded, outperforms HEXQ and *flat* RL learners and also addresses the concerns of increased dimensionality. However, it doesn't take into account any intrinsic rewards given for exploration. A directed exploration can easily be implemented in the proposed algorithm with the use of predefined intrinsic reward formulation, for example, Bayesian Inference Criterion (BIC) [45]. In the action selection scheme, an agent can then plan to explore an unknown region after weighing intrinsic and extrinsic rewards. The implemented value function decomposition leads to a *broken* value function wherein the values across the hierarchies are not related. Such a value function can be further linked together across the hierarchy using a formulation similar to [38]. With an updated value function, a value of a state at any level in the hierarchy can be directly compared to a value learned by a *flat* RL learner. Furthermore, in the problem domain with stochastic transitions, the performance of the proposed algorithm can further be improved using different model learning methods especially by improving the probability estimation scheme.

The main prerequisite for the proposed scheme is to have a DBN model of the underlying FMDP. However, in future research directions, such a DBN model of the problem domain can also be learned simultaneously. Using the extensions proposed for partial-order planning case, an agent can start learning with a value function similar to a *flat* RL agent and then as it discovers the causal relations between variables, this value function can be divided or merged together. In such a scheme, the division of value function imitates the division of the original MDP into SMDPs.

The reasoning implemented in the proposed scheme through planning is of a causal nature. In a certain problem domain, if more than one sequence of abstract actions exist to the goal then temporal constraints can also be implemented taking into consideration the estimated probabilities of the time of execution for each option. Hence, if two or more paths with different reward distribution and time of execution exist to reach the goal, an agent can choose its actions based on causal and temporal constraints of the problem domain. Furthermore, the proposed scheme needs to be evaluated in multi-task RL problems to check the transferability of learned influence graphs and options with a final aim of having a continual learning algorithm.

Bibliography

[1] M. Wiering and M. Van Otterlo, "Reinforcement learning," *Adaptation, learning, and optimization*, vol. 12, 2012.

[2] S. Russell and P. Norvig, "Intelligent agents," *Artificial intelligence: A modern approach*, vol. 74, pp. 46–47, 1995.

[3] H. A. Simon, *The sciences of the artificial*. MIT press, 1996.

[4] P. E. Utgoff and D. J. Stracuzzi, "Many-layered learning," *Neural Computation*, vol. 14, no. 10, pp. 2497–2529, 2002.

[5] J. Menashe and P. Stone, "Monte carlo hierarchical model learning," in *Proceedings of the 2015 International Conference on Autonomous Agents and Multiagent Systems*, pp. 771–779, 2015.

[6] Ö. Şimşek, A. P. Wolfe, and A. G. Barto, "Identifying useful subgoals in reinforcement learning by local graph partitioning," in *Proceedings of the 22nd international conference on Machine learning*, pp. 816–823, ACM, 2005.

[7] I. Menache, S. Mannor, and N. Shimkin, "Q-cut—dynamic discovery of sub-goals in reinforcement learning," in *European Conference on Machine Learning*, pp. 295–306, Springer, 2002.

[8] 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*, p. 71, ACM, 2004.

[9] Ö. Şimşek 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*, p. 95, ACM, 2004.

[10] B. Digney, "Emergent hierarchical control structures: Learning reactive/hierarchical relationships in reinforcement environments," *From animals to animats*, vol. 4, pp. 363–372, 1996.

[11] A. McGovern and A. G. Barto, "Automatic discovery of subgoals in reinforcement learning using diverse density," in *ICML*, vol. 1, pp. 361–368, 2001.

[12] Ö. Şimşek and A. G. Barto, "Skill characterization based on betweenness," in *Advances in neural information processing systems*, pp. 1497–1504, 2009.

[13] S. Thrun and A. Schwartz, "Finding structure in reinforcement learning," in *Advances in neural information processing systems*, pp. 385–392, 1995.

[14] D. Precup, *Temporal abstraction in Reinforcement Learning*. University of Massachusetts Amherst, 2000.

[15] M. Pickett and A. G. Barto, "Policyblocks: An algorithm for creating useful macro-actions in reinforcement learning," in *ICML*, pp. 506–513, 2002.

[16] A. Jonsson and A. Barto, "Causal graph based decomposition of factored mdps," *Journal of Machine Learning Research*, vol. 7, no. Nov, pp. 2259–2301, 2006.

[17] B. Hengst, "Discovering hierarchy in reinforcement learning with hexq," in *ICML*, vol. 2, pp. 243–250, 2002.

[18] G. Konidaris, "Constructing abstraction hierarchies using a skill-symbol loop," in *IJCAI: proceedings of the conference*, vol. 2016, p. 1648, 2016.

[19] G. Konidaris, L. P. Kaelbling, and T. Lozano-Perez, "From skills to symbols: Learning symbolic representations for abstract high-level planning," *Journal of Artificial Intelligence Research*, vol. 61, pp. 215–289, 2018.

[20] C. Boutilier, R. Dearden, and M. Goldszmidt, "Stochastic dynamic programming with factored representations," *Artificial intelligence*, vol. 121, no. 1-2, pp. 49–107, 2000.

[21] T. Dean and K. Kanazawa, "A model for reasoning about persistence and causation," *Computational intelligence*, vol. 5, no. 2, pp. 142–150, 1989.

[22] C. Guestrin, D. Koller, R. Parr, and S. Venkataraman, "Efficient solution algorithms for factored mdps," *Journal of Artificial Intelligence Research*, vol. 19, pp. 399–468, 2003.

[23] M. L. Puterman, *Markov Decision Processes: discrete stochastic dynamic programming*. John Wiley & Sons, 2014.

[24] R. S. Sutton, A. G. Barto, *et al.*, *Reinforcement learning: An introduction*. MIT press, 1998.

[25] C. J. Watkins and P. Dayan, "Q-learning," *Machine learning*, vol. 8, no. 3-4, pp. 279–292, 1992.

[26] G. A. Rummery and M. Niranjan, *On-line Q-learning using connectionist systems*, vol. 37. University of Cambridge, Department of Engineering Cambridge, England, 1994.

[27] V. R. Konda and J. N. Tsitsiklis, "Actor-critic algorithms," in *Advances in neural information processing systems*, pp. 1008–1014, 2000.

[28] T. G. Dietterich, "Hierarchical reinforcement learning with the maxq value function decomposition," *Journal of Artificial Intelligence Research*, vol. 13, pp. 227–303, 2000.

[29] T. G. Dietterich, "State abstraction in maxq hierarchical reinforcement learning," in *Advances in Neural Information Processing Systems*, pp. 994–1000, 2000.

[30] R. S. Sutton, D. Precup, and S. Singh, "Between mdps and semi-mdps: A framework for temporal abstraction in reinforcement learning," *Artificial intelligence*, vol. 112, no. 1-2, pp. 181–211, 1999.

[31] D. Andre and S. J. Russell, "State abstraction for programmable reinforcement learning agents," in *AAAI/IAAI*, pp. 119–125, 2002.

[32] S. P. Singh, "Reinforcement learning with a hierarchy of abstract models," in *Proceedings of the National Conference on Artificial Intelligence*, no. 10, p. 202, John Wiley & Sons Ltd., 1992.

[33] R. S. Sutton, "Dyna, an integrated architecture for learning, planning, and reacting," *ACM SIGART Bulletin*, vol. 2, no. 4, pp. 160–163, 1991.

[34] E. A. Mcgovern and A. G. Barto, *Autonomous discovery of temporal abstractions from interaction with an environment*. PhD thesis, University of Massachusetts at Amherst, 2002.

[35] D. Potts and B. Hengst, "Concurrent discovery of task hierarchies," in *AAAI Spring Symposium on Knowledge Representation and Ontology for Autonomous Systems*, 2004.

[36] B. Hengst, "Partial order hierarchical reinforcement learning," in *Australasian Joint Conference on Artificial Intelligence*, pp. 138–149, Springer, 2008.

[37] G. Konidaris and A. G. Barto, "Skill discovery in continuous reinforcement learning domains using skill chaining," in *Advances in neural information processing systems*, pp. 1015–1023, 2009.

[38] B. Hengst, "Safe state abstraction and reusable continuing subtasks in hierarchical reinforcement learning," in *Australasian Joint Conference on Artificial Intelligence*, pp. 58–67, Springer, 2007.

[39] N. Mehta, S. Ray, P. Tadepalli, and T. Dietterich, "Automatic discovery and transfer of maxq hierarchies," in *Proceedings of the 25th international conference on Machine learning*, pp. 648–655, ACM, 2008.

[40] A. G. Barto and S. Mahadevan, "Recent advances in hierarchical reinforcement learning," *Discrete Event Dynamic Systems*, vol. 13, no. 4, pp. 341–379, 2003.

[41] G. Konidaris and A. G. Barto, "Building portable options: Skill transfer in reinforcement learning.," in *IJCAI*, vol. 7, pp. 895–900, 2007.

[42] G. Konidaris, S. Kuindersma, R. Grupen, and A. G. Barto, "Constructing skill trees for reinforcement learning agents from demonstration trajectories," in *Advances in neural information processing systems*, pp. 1162–1170, 2010.

[43] G. Konidaris, S. Kuindersma, R. Grupen, and A. Barto, "Robot learning from demonstration by constructing skill trees," *The International Journal of Robotics Research*, vol. 31, no. 3, pp. 360–375, 2012.

[44] G. Konidaris, L. P. Kaelbling, and T. Lozano-Perez, "Constructing symbolic representations for high-level planning.," in *AAAI*, pp. 1932–1938, 2014.

[45] C. M. Vigorito and A. G. Barto, "Intrinsically motivated hierarchical skill learning in structured environments," *IEEE Transactions on Autonomous Mental Development*, vol. 2, no. 2, pp. 132–143, 2010.