Reinforcement Learning Approaches in Dynamic Environments

Miyoung Han

To cite this version:

Miyoung Han. Reinforcement Learning Approaches in Dynamic Environments. Databases [cs.DB]. Télécom ParisTech, 2018. English. NNT : . tel-01891805

HAL Id: tel-01891805
https://inria.hal.science/tel-01891805v1

Submitted on 10 Oct 2018

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

ParisTech
INSTITUT DES SCIENCES ET TECHNOLOGIES
PARIS INSTITUTE OF TECHNOLOGY

TELECOM ParisTech

2018-ENST-????

EDITE - ED 130

Doctorat ParisTech

T H È S E

pour obtenir le grade de docteur délivré par

TÉLÉCOM ParisTech

Spécialité « Informatique »

présentée et soutenue publiquement par

Miyoung HAN

le 19 juillet 2018

Approches d'apprentissage par renforcement dans les environnements dynamiques

Directeur de thèse : **Pierre Senellart**

Jury

Mme AMER-YAHIA Sihem, Directrice de Recherche, CNRS	Examinatrice
M. CAUTIS Bogdan, Professeur, Université Paris-Sud	Rapporteur
M. GROSS-AMBLARD David, Professeur, Université de Rennes 1	Rapporteur
M. SENELLART Pierre, Professeur, ENS, Université PSL	Directeur de thèse
M. WUILLEMIN Pierre-Henri, Maître de Conférences, Sorbonne University	Examinateur

T H È S E

TÉLÉCOM ParisTech
école de l'Institut Mines-Télécom - membre de ParisTech
46 rue Barrault 75013 Paris - (+33) 1 45 81 77 77 - www.telecom-paristech.fr

Contents

1 Introduction **5**
1.1 Motivation and Objective . . . 5
1.2 Reinforcement Learning . . . 6
1.3 Overview and Contributions . . . 7

I Literature Review **9**

2 Reinforcement Learning **11**
2.1 Markov Decision Processes . . . 11
2.2 Dynamic Programming . . . 13
2.2.1 Policy Iteration . . . 13
2.2.2 Value Iteration . . . 15
2.2.3 Interaction between Policy Evaluation and Policy Improvement. 16
2.3 Temporal-Difference Methods . . . 16
2.4 Function Approximation Methods . . . 19
2.5 Exploration . . . 23
2.6 Model-Based and Model-Free Methods . . . 24
2.7 Priority-Based Value Iteration . . . 26
2.8 Non-Stationary Environment . . . 29

II Applications **33**

3 Model-Free and Model-Based Methods **35**
3.1 Introduction . . . 35
3.2 Learning without Models . . . 35
3.2.1 Background and Related Work . . . 36
3.2.2 Q-learning for Taxi Routing . . . 36
3.2.3 Performance Evaluation . . . 38
3.2.4 Demonstration Scenario . . . 40
3.3 Learning Models . . . 42
3.3.1 Background: Factored MDP . . . 42
3.3.2 Related work . . . 44
3.3.3 Algorithm for Structure Learning . . . 45

3.3.4 Experiments . . . 47
3.4 Discussion and Future Research . . . 50
3.5 Conclusion . . . 51

4 Focused Crawling **53**
4.1 Introduction . . . 53
4.2 Background . . . 54
4.3 Focused Crawling and Reinforcement Learning . . . 55
4.3.1 Markov Decision Processes (MDPs) in Crawling . . . 56
4.3.2 MDPs with Prioritizing Updates . . . 59
4.3.3 Linear Function Approximation with Prioritizing Updates . . . 59
4.4 Experimental Results . . . 61
4.5 Related Work . . . 65
4.6 Future Work . . . 67
4.7 Conclusion . . . 68

5 Influence Maximization **73**
5.1 Introduction . . . 73
5.2 Background . . . 74
5.3 Topic-Based Influence Maximization Algorithm for Unknown Graphs 76
5.3.1 Problem Statement and our Method . . . 76
5.3.2 Modeling and Algorithm . . . 77
5.4 Related Work . . . 81
5.5 Future Work . . . 83
5.6 Conclusion . . . 84

6 Conclusion **87**
6.1 Future Work . . . 87
6.2 Conclusion . . . 88

Bibliography **91**

Appendices **101**

A Résumé en français **103**

Chapter 1

Introduction

In this thesis, we apply reinforcement learning to sequential decision making problems in dynamic environments. This chapter presents the motivation, objective, and an overview of this thesis. We begin by presenting the motivation and objective of this thesis. Then we introduce briefly reinforcement learning that is used as a fundamental framework throughout this thesis. Finally we provide a preview of each chapter including contributions.

1.1 Motivation and Objective

Reinforcement learning [94] is based on the idea of trial-and-error learning and it has been commonly used in robotics, with applications such as robot soccer [88], robot helicopters [1], etc.

It has also been used in various applications that concern sequential decision making problems in dynamic environments such as power management [95], channel allocation [91], traffic light control problems [19], etc. Power management in data centers is a rapidly growing concern in economic and environmental issues. In [95], a reinforcement learning approach is presented to learn effective management policies of both performance and power consumption in web application servers. In cellular telephone systems, an important problem is to dynamically allocate the communication channels to maximize the service provided to mobile callers. This problem is tackled in [91] using a reinforcement learning method to allocate the available channels to calls in order to minimize the number of blocked calls and the number of calls that are dropped when they are handed off to a busy call. A reinforcement learning method is also applied to the traffic lights control problem [19] that adjusts traffic signal according to real-time traffic in order to reduce traffic congestion. The agent learns a traffic signal control policy in which vehicles do not wait too long for passing through the intersection.

These problems have explicit goals to achieve and they require making an optimal decision for a given environment in order to achieve the goals. Environments change in reaction to some control behaviors. However, it is difficult to design optimal policies in advance because environment models are not available. In such problems, reinforcement learning can be used to find the optimal policies. It learns the policies by interacting with the environment in order to achieve a goal. The learned policies

take into account long-term consequences of individual decisions.

In this thesis, we solve several sequential decision making problems using reinforcement learning methods. For example, in a focused crawling problem, a crawler has to collect as many Web pages as possible that are relevant to a predefined topic while avoiding irrelevant pages. Many crawling methods use classification for unvisited links to estimate if the links point to relevant pages but these methods do not take into account long-term effects of selecting a link. In the influence maximization problem, the agent aims to choose the most influential seeds to maximize influence under a certain information diffusion model. The problem already takes into account long-term values but not necessarily the planning dimension that reinforcement learning introduces.

To solve such sequential decision making problems, we first formulate the problems as Markov decision processes (MDPs), a general problem formulation of reinforcement learning. Then we solve these problems using appropriate reinforcement learning methods for corresponding problems and demonstrate that reinforcement learning methods find stochastic optimal policies for each problem that are close to the optimal.

1.2 Reinforcement Learning

Reinforcement learning is similar to the way of learning of humans and animals. In fact, many of the algorithms of reinforcement learning are inspired by biological learning systems [94].

In reinforcement learning, an agent learns from continuing interaction with an environment in order to achieve a goal. Such interaction produces lots of information about the consequences of the behavior, that helps to improve its performance. Whenever the learning agent does an action, the environment responds to its action by giving a reward and presenting a new state. The agent's objective is to maximize the total amount of reward it receives. Through experience in its environment, it discovers which actions stochastically produce the greatest reward and uses such experience to improve its performance for subsequent trials. That is, the agent learns how to behave in order to achieve goals. In reinforcement learning, all agents have explicit goals and learn decisions by interacting with their environment in order to achieve the goals.

Reinforcement learning focuses on learning how good it is for the agent to be in a state over the long run, called a *value of state*, or how good it is to take an action in a given state over the long run, called a *value of action*. A reward is given immediately by an environment as a response of the agent's action and a learning agent uses the reward to evaluate the value of a state or action. The best action is selected by values of states or actions because the highest value brings about the greatest amount of reward over the long run. Then the learning agent can maximize the cumulative reward it receives.

A model represents the environment's dynamics. A learning agent learns value functions with or without a model. When a reinforcement learning algorithm constructs a model of the environment and learns value functions from the model, it is called a *model-based method*. Reinforcement learning algorithms can learn value

functions directly from experience without any environment models. If an algorithm learns values of states or actions from trial-and-error without a model, we call it a *model-free method*. Since a model mimics the behavior of the environment, it allows to estimate how the environments will change in response to what the agent does. However, learning a complete and accurate model requires more complex computation than model-free methods. We study a model-free method and a model-based method in Chapter 3.

These value functions can be represented using tabular forms but, in large and complicated problems, tabular forms cannot efficiently store all value functions. In this case, the functions must be approximated using parameterized function representation for large problems. In Chapters 4 and 5, we study a focused crawling problem and an influence maximization problem using a function approximation method.

1.3 Overview and Contributions

In Chapter 2, we review the main concepts of reinforcement learning that we have used as a fundamental framework throughout this thesis. We start with the notion of Markov decision process (MDP), that is the general problem formulation of reinforcement learning. Then, we describe the fundamental methods for solving MDP problems, such as dynamic programming (DP) and temporal-difference (TD) methods. These methods based on tabular forms can be extended into function approximation methods that can be applied to much larger-scale problems. We also present some important topics or improvements presented in the reinforcement learning literature.

In Chapter 3, we study two main approaches for solving reinforcement learning problems: model-free and model-based methods. First, we study a model-free method that learns directly from observed experiences without a model. We present a *Q-learning* [98] based algorithm with a customized exploration and exploitation strategy to solve a real taxi routing problem. We demonstrate that a reinforcement learning algorithm is able to progressively learn optimal actions for routing an autonomous taxi to passenger pick-up points. In experiments, we quantify the influence of two important parameters of Q-learning – the step size and discount rate – on effectiveness. We also investigate the influence of trade-off between exploration and exploitation on learning. We published that work in the industry track of the CIKM 2016 conference [50].

Then, we turn to a model-based method that learns transition and reward models of the environment. We address the factored MDP problem [7] where a state is represented by a vector of n variables, in a non-deterministic setting. Most model-based methods are based on Dynamic Bayesian Network (DBN) transition models. We propose an algorithm that learns the DBN structures of state transitions including synchronic parents. Decision trees are used to represent transition functions. In experiments, we show the efficiency of our algorithm by comparing with other algorithms. We also demonstrate that factorization methods allow to learn effectively complete and correct models to obtain the optimal policies and through the learned models the agent can accrue more cumulative rewards.

In Chapter 4, we extend our discussion to a very large and continuous domain, in particular, a focused crawling problem. Focused crawling aims at collecting as many Web pages relevant to a target topic as possible while avoiding irrelevant pages, reflecting limited resources available to a Web crawler. We improve on the efficiency of focused crawling by proposing an approach based on reinforcement learning that learns link scores in an online manner. Our algorithm evaluates hyperlinks most profitable to follow over the long run, and selects the most promising link based on this estimation. To properly model the crawling environment as an MDP, we propose new feature representations of states (Web pages) and actions (next link selection) considering both content information and the link structure. A number of pages and links are generalized with the proposed features. Based on this generalization, we use a linear function approximation with gradient descent to estimate value functions, i.e., link scores. We investigate the trade-off between synchronous and asynchronous methods to maintain action values (link scores) in the frontier that are computed at different time steps. As an improved asynchronous method, we propose moderated update to reach a balance between action-values updated at different time steps. We compare the performance of a crawling task with and without learning. Crawlers based on reinforcement learning show better performance for various target topics. Our experiments demonstrate that reinforcement learning allows to estimate long-term link scores and to efficiently crawl relevant pages. The work presented in that chapter is published at the ICWE 2018 conference [51].

In Chapter 5, we continue our discussion with another very large domain, an influence maximization problem. Given a social network, the influence maximization problem is to choose an optimal initial seed set of a given size to maximize influence under a certain information diffusion model such as the independent cascade (IC) model, the linear threshold (LT) model, etc. We extend the classical IM problem with incomplete knowledge of graph structure and topic-based user's interest. Assuming that the graph structure is incomplete or can change dynamically, we address a topic-based influence maximization problem for an unknown graph. In order to know a part of the graph structure and discover potentially promising nodes, we probe nodes that may have a big audience group. Then, we find the most influential seeds to maximize topic-based influence by using reinforcement learning. As we select seeds with a long-term impact in the influence maximization problem, action values in the reinforcement learning signify how good it is to take an action in a given state over the long run. Thus we learn action values of nodes from interaction with the environment by reinforcement learning. For this, nodes are generalized with some features that represent a node's proper information and relation information with respect to surrounding nodes. We define states and actions based on these features, and we evaluate action value for each probed node and select a node with the highest action value to activate.

Finally, in Chapter 6, we discuss various interesting directions for future work. Then, we conclude this thesis with some additional remarks.

Part I

Literature Review

Chapter 2

Reinforcement Learning

Reinforcement learning [94] is learning from interaction with an environment to achieve a goal. It is a powerful framework to solve sequential decision-making problems. The agent discovers which actions produce the greatest reward by experiencing actions and learns how good it is for the agent to be in a state over the long run, called the *value of state*, or how good it is to take a certain action in a given state over the long-term, quantified by the *value of action*. Reinforcement learning aims to maximize the total reward in the long run. Rewards are given immediately by selecting an action but values of states (or actions) must be estimated from an agent's experience. Since states (or actions) with the highest values can bring about the greatest amount of reward over the long run, we are most concerned with the value of state (or action) when making decisions.

In this chapter, we start with Markov decision processes (MDPs) which are a key formalism for reinforcement learning. Then, we describe the fundamental methods for solving MDP problems, such as dynamic programming (DP) and temporal-difference (TD) methods. These methods based on tabular forms can be extended into function approximation methods that can be applied to much larger problems. The remaining sections present some important topics or improvements from the literature.

2.1 Markov Decision Processes

The notion of Markov Decision Process (MDP) underlies much of the work on reinforcement learning. An MDP is defined as a 4-tuple $\mathrm{M} = \langle S, A, R, T \rangle$ where S is a set of states, A is a set of actions, $R : S \times A \to \mathbb{R}$ is a reward function, and $T : S \times A \times S \to [0, 1]$ is a transition function. The reward function returns a single number, a reward, for an action selected in a given state. The transition function specifies the probability of transition from state s to state s' on taking action a (denoted $T(s, a, s')$ or, simply, $\Pr(s' \mid s, a)$). A finite MDP is an MDP in which the sets of states S, actions A, and rewards R have a finite number of elements.

An entity that learns and makes decisions is called the *agent* and everything outside the agent is called the *environment*. The agent learns from continual interaction with an environment to achieve a goal. The agent selects actions and the environment responds to these actions by giving a reward and presenting new state.

The objective of the agent is to maximize the total amount of reward it receives in the long run.

We usually describe the interaction between the agent and the environment with a sequence of discrete time steps. At time step t, the agent is given a state s_t and selects an action a_t on the basis of the current state. At time step $t+1$, the agent receives a reward r_{t+1} and new state s_{t+1} as a result of taking action a_t.

In the MDP environment, a state should retain all relevant information, though we are not concerned with the complete history of states that led to it. We say that the state has the *Markov property*: if the state is a Markov state, then the environment's response at time $t+1$ depends only on the state and action representations at time t. This Markov property enables to predict the next state and reward given the current state and action. Relevant information about states and actions are typically summarized in a compact form.

A *policy* $\pi : S \times A \to [0, 1]$ maps states to probabilities of selecting an action. The policy π represents the agent's action selection in a certain state s. In any MDP, there is a policy that is better than or equal to all other policies for all states. This is called an optimal policy, denoted π^*. The goal of the agent is to find an optimal policy π^* that maximizes the total reward in the long run.

Rewards and Values. At each time step, the agent receives a reward from the environment as a consequence of its behavior. The objective of the agents is to maximize the total amount of reward it receives in the long run. To achieve the objective, the agent has to estimate the expected total amount of reward starting from a state, called the *value of state*. While a reward characterizes how good the action is in an immediate sense, a value function of state measures how good it is for the agent to be in a given state over the long run. A reward is given directly from the environment but a value function must be learned from the agent's experience. In reinforcement learning, when making decisions, the agent does not focus on immediate rewards but on values, i.e., cumulative reward in the long run. A state might yield a low immediate reward but it does not mean that the following state also brings about a low reward. If a state with a low reward is followed by some states that yield high rewards, it has a high value. Thus, the agent has to follow states with the highest values not the highest immediate rewards because those states bring about the greatest amount of reward over the long run.

Value Functions. For each time step, the agent selects an action and then learns from its experience. By repeated action-selection behavior, the agent learns the value of being in a state and taking an action in a state. The value functions are defined with respect to policies. The value of a state s under a policy π, denoted $v_\pi(s)$, is defined as the expected future rewards when starting from state s and following policy π, using a *discount factor* $0 \leq \gamma \leq 1$ (usually, $0 < \gamma < 1$):

$$v_\pi(s) \doteq \mathbb{E}_\pi \left[\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \mid s_t = s \right] \tag{2.1}$$

The discount factor γ determines the present value of future rewards. If $\gamma = 0$, the agent is only concerned with the immediate reward. The agent's action influences

only the current reward. If γ approaches 1, the agent considers future rewards more strongly on its action.

The value function can be computed recursively:

$$
\begin{aligned}
v_\pi(s) &= \mathbb{E}_\pi \left[r_{t+1} + \gamma v_\pi(s_{t+1}) \mid s_t = s \right] && (2.2)\\
&= \sum_a \pi(a \mid s) \sum_{s'} \Pr(s' \mid s, a) \left[R(s,a) + \gamma v_\pi(s') \right] && (2.3)
\end{aligned}
$$

Similarly, the optimal state-value function, denoted v_*, is defined as:

$$
v_*(s) = \max_a \sum_{s'} \Pr(s' \mid s, a) \left[R(s,a) + \gamma v_*(s') \right] \tag{2.4}
$$

The value of taking an action a in a state s under a policy π, denoted $q_\pi(s,a)$, is also defined in the same way and computed recursively:

$$
\begin{aligned}
q_\pi(s,a) &\doteq \mathbb{E}_\pi \left[\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \mid s_t = s, a_t = a \right] && (2.5)\\
&= \mathbb{E}_\pi \left[r_{t+1} + \gamma q_\pi(s_{t+1}, a_{t+1}) \mid s_t = s, a_t = a \right] && (2.6)\\
&= \sum_{s'} \Pr(s' \mid s, a) \left[R(s,a) + \gamma \sum_{a'} \pi(s', a') q_\pi(s', a') \right] && (2.7)
\end{aligned}
$$

The optimal action-value function, denoted q_*, is similarly described with a recursive definition:

$$
q_*(s,a) = \sum_{s'} \Pr(s' \mid s, a) \left[R(s,a) + \gamma \max_{a'} q_*(s', a') \right] \tag{2.8}
$$

2.2 Dynamic Programming

Dynamic programming (DP) is a collection of algorithms that assume a complete and perfect model of the environment's dynamics as an MDP and compute optimal policies using the model. The model of the environment's dynamics means the transition function and the reward function. Since DP requires a prior knowledge of a complete model and a great computational expense, DP is of limited applicability but it is an essential foundation of reinforcement learning. DP algorithms use value functions to obtain optimal policies. We present two fundamental DP methods, policy iteration and value iteration, in the following subsections.

2.2.1 Policy Iteration

Policy iteration consists of two processes, policy evaluation and policy improvement. Policy evaluation computes the value functions consistent with a given policy and policy improvement makes the policy greedy with respect to the value function obtained in policy evaluation.

Policy Evaluation In policy evaluation, the state-value function v_π is computed for an arbitrary policy π. We assume that the environment's dynamics are completely known. The state-value function v_π can be obtained iteratively by using the Bellman equation.

$$v_{k+1}(s) \doteq \mathbb{E}_\pi \left[r_{t+1} + \gamma v_k(s_{t+1}) \mid s_t = s \right] \tag{2.9}$$

$$= \sum_a \pi(a \mid s) \sum_{s'} \Pr(s' \mid s, a) \left[R(s,a) + \gamma v_k(s') \right] \tag{2.10}$$

where $\pi(a \mid s)$ is the probability of taking action a in state s under policy π. The sequence (v_k) converges to v_π as $k \to \infty$ if $\gamma < 1$ or eventual termination is guaranteed from all states under the policy π [94]. The iteration is stopped when the changes of value functions are lower than a threshold θ. The iterative policy evaluation algorithm is shown as follows:

Algorithm 1 Iterative policy evaluation

1: Input: π, the policy to be evaluated
2: Initialize an array $V(s) = 0$, for all $s \in S$
3: **repeat**
4: $\quad \Delta \leftarrow 0$
5: $\quad$ **for** each $s \in S$ **do**
6: $\quad\quad v \leftarrow V(s)$
7: $\quad\quad V(s) \leftarrow \sum_a \pi(a \mid s) \sum_{s'} \Pr(s' \mid s, a) \left[R(s,a) + \gamma V(s') \right]$
8: $\quad\quad \Delta \leftarrow \max(\Delta, |v - V(s)|)$
9: $\quad$ **end for**
10: **until** $\Delta < \theta$ (a small positive number)
11: Output $V \approx v_\pi$

Policy Improvement We obtained the value function v_π under an arbitrary deterministic policy π from the policy evaluation step. In the policy improvement method, we consider whether it is better to change the current policy π to the new policy π'. For example, for some state s, we keep following the current policy $\pi(s)$ or select an action $a \neq \pi(s)$. If it is better to select a in s and thereafter follow the existing policy π than it would be to follow π all the time, we have to change the current policy π to new policy π' with $\pi'(s) \neq \pi(s)$. This is generalized in the policy improvement theorem.

Theorem 1 (policy improvement theorem)**.** *Let π and π' be any pair of deterministic policies such that, for all $s \in S$, $q_\pi(s, \pi'(s)) \geq v_\pi(s)$. Then the policy π' must be as good as, or better than, π. That is, it must obtain greater or equal expected return from all states $s \in S$: $v'_\pi(s) \geq v_\pi(s)$.*

According to the policy improvement theorem, we can define the new greedy

policy π' by

$$\pi'(s) \doteq \arg\max_a q_\pi(s,a) \tag{2.11}$$
$$= \arg\max_a \mathbb{E}\left[r_{t+1} + \gamma v_\pi(s_{t+1}) \mid s_t = s, a_t = a\right] \tag{2.12}$$
$$= \arg\max_a \sum_{s'} \Pr(s' \mid s,a)\left[R(s,a) + \gamma v_\pi(s')\right] \tag{2.13}$$

The new policy π' improves on an original policy π by greedily taking actions according to value function v_π. The deterministic policy π we have seen above can be extended to the general form, a stochastic policy that is specified by probabilities $\pi(a \mid s)$ for taking each action a in each state s.

Policy Iteration In the policy iteration method, we repeat policy evaluation and policy improvement until convergence to an optimal policy and optimal value function. Given an arbitrary policy π, we compute value function v_π and improve the policy π with respect to value function v_π to yield a better policy π'. Then we compute again v'_π and improve π' to get a better policy π'' and so on. A complete algorithm is as follows:

Algorithm 2 Policy Iteration (using iterative policy evaluation)

1: $V(s) \in \mathbb{R}$ and $\pi(s) \in A(s)$ arbitrarily for all $s \in S$
2:
3: // 1. Policy Evaluation
4: **repeat**
5: $\quad \Delta \leftarrow 0$
6: $\quad$ **for** each $s \in S$ **do**
7: $\quad\quad v \leftarrow V(s)$
8: $\quad\quad V(s) \leftarrow \sum_{s'} \Pr(s' \mid s, \pi(s))\left[R(s,a) + \gamma V(s')\right]$
9: $\quad\quad \Delta \leftarrow \max(\Delta, \mid v - V(s) \mid)$
10: $\quad$ **end for**
11: **until** $\Delta < \theta$ (a small positive number)
12:
13: // 2. Policy Improvement
14: *policy-stable* $\leftarrow$ *true*
15: **for** each $s \in S$ **do**
16: $\quad$ *old-action* $\leftarrow \pi(s)$
17: $\quad \pi(s) \leftarrow \arg\max_a \sum_{s'} \Pr(s' \mid s,a)\left[R(s,a) + \gamma V(s')\right]$
18: $\quad$ If *old-action* $\neq \pi(s)$, then *policy-stable* $\leftarrow$ $false$
19: **end for**
20: If *policy-stable*, then stop and return $V \approx v_*$ and $\pi \approx \pi_*$; else go to 4

2.2.2 Value Iteration

In policy iteration, each iteration involves policy evaluation that repeatedly sweeps through the state space. The policy evaluation step can be truncated without losing

the convergence guarantees of policy iteration [94]. When policy evaluation is used just once, we call this value iteration. Then we combine the policy evaluation and the policy improvement steps in a simple update operation:

$$v_{k+1}(s) \doteq \max_a \mathbb{E}\left[r_{t+1} + \gamma v_k(s_{t+1}) \mid s_t = s, a_t = a\right] \tag{2.14}$$

$$= \max_a \sum_{s'} \Pr(s' \mid s, a)\left[R(s,a) + \gamma v_k(s')\right] \tag{2.15}$$

In Eq. (2.15), an action is greedily selected with respect to the current value function and the selected greedy action is used to update the value function. A complete algorithm of value iteration is as follows:

Algorithm 3 Value Iteration

1: Initialize array V arbitrarily (e.g., $V(s) = 0$ for all $s \in S$)
2: **repeat**
3: $\quad \Delta \leftarrow 0$
4: $\quad$ **for** each $s \in S$ **do**
5: $\qquad v \leftarrow V(s)$
6: $\qquad V(s) \leftarrow \max_a \sum_{s'} \Pr(s' \mid s, a)\left[R(s,a) + \gamma V(s')\right]$
7: $\qquad \Delta \leftarrow \max(\Delta, \mid v - V(s) \mid)$
8: $\quad$ **end for**
9: **until** $\Delta < \theta$ (a small positive number)
10: Output a deterministic policy, $\pi \approx \pi_*$, such that
11: $\quad \pi(s) \leftarrow \arg\max_a \sum_{s'} \Pr(s' \mid s, a)\left[R(s,a) + \gamma V(s')\right]$

2.2.3 Interaction between Policy Evaluation and Policy Improvement.

In policy iteration, two processes, policy evaluation and policy improvement, alternate. When one process completes then the other process begins. Policy evaluation makes the value function consistent with the current policy. Then the updated value function is used to improve the policy. In policy improvement, the policy is changed with respect to the current value function. As the interaction of two processes continues, the policy and the value function move toward the optimal. When the policy and the value function are not changed by either process then they are optimal. The interaction between the policy evaluation and policy improvement processes underlies almost all reinforcement learning methods [94]. Whereas policy iteration separates them as two different processes, value iteration merges them in one process.

2.3 Temporal-Difference Methods

While dynamic programming (DP) in section 2.2 needs a complete and accurate model of the environment, temporal-difference (TD) methods do not require prior

knowledge about the environment's dynamics. They compute value functions using raw experience in an on-line, fully incremental manner. For example, at time t, if the agent takes action a_t in state s_t under policy π, the action causes a transition to s_{t+1} with reward r_{t+1}. With this experience, the TD method updates the value function of s_t by:

$$V(s_t) \leftarrow V(s_t) + \alpha \left[r_{t+1} + \gamma V(s_{t+1}) - V(s_t) \right] \tag{2.16}$$

The quantity in brackets in the update is called the *TD error*.

$$\delta_t \doteq r_{t+1} + \gamma V(s_{t+1}) - V(s_t) \tag{2.17}$$

It is the difference between the current estimated value of s_t and the better estimated value based on the actual observed reward and the estimated value of the next state, s_{t+1}, i.e., $r_{t+1} + \gamma V(s_{t+1})$. As value functions are repeatedly updated, the errors are reduced. Here, α is a positive fraction such that $0 < \alpha \leq 1$, the step-size parameter that influences the rate of learning. When $\alpha = 1$, the agent considers only the most recent information for learning. If α is properly reduced over time, the function converges [94].

TD methods are the most widely used methods due to their several advantages such as computational simplicity, on-line learning approach, and learning directly from experience generated from interaction with an environment.

In TD methods, each iteration of value updates is based on an *episode*, a sequence of state transitions from a start state to the terminal state (or, in some cases, till some other condition has been reached, for example, the limited number of time steps, etc.). For example, at time t, in state s, the agent takes an action a according to its policy, which results in a transition to state s'. At time $t+1$ in the successor state of s, state s', the agent takes its best action a' followed by a transition to state s'' and so on until the terminal state. Each episode starts in a starting state or any randomly selected state and ends in the terminal state.

A complete algorithm for the TD method is shown in Algorithm 4.

Algorithm 4 Tabular TD for estimating v_π

```
1:  Input: the policy π to be evaluated
2:  Initialize V(s) arbitrarily (e.g., V(s) = 0, for all s ∈ S)
3:  repeat for each episode
4:      Initialize s
5:      repeat for each step of episode
6:          a ← action given by π for s
7:          Take action a, observe r, s'
8:          V(s) ← V(s) + α [r + γV(s') − V(s)]
9:          s ← s'
10:     until s is terminal
11: until
```

We now turn to action-value functions rather than state-value functions. There are two main approaches for learning q_π in TD methods: on-policy and off-policy.

On-policy TD method: Sarsa First we consider an on-policy TD method called Sarsa [94]. In on-policy method, we estimate $q_\pi(s, a)$ for the current behavior policy π and change π toward greediness with respect to q_π. This method learns action-values based on transitions from a state-action pair to a state-action pair. Since it uses a tuple of transition experience $(\boldsymbol{s_t, a_t, r_{t+1}, s_{t+1}, a_{t+1}})$for each update, it is named **Sarsa** and an action value is updated by:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \left[r_{t+1} + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)\right] \quad (2.18)$$

The pseudocode of Sarsa is given in Algorithm 5. An action can be greedily selected all the time, called *greedy policy*. Alternatively, most of the time, the agent selects an action with the highest estimated value, but with small probability ϵ selects an action uniformly at random, called ϵ-*greedy policy* that is one of the most commonly used methods (see Section 2.5 for other action selection methods).

Algorithm 5 Sarsa (on-policy TD) for estimating $Q \approx q_*$

1: Initialize $Q(s, a)$ for all $s \in S, a \in A(s)$, arbitrarily, and $Q(\textit{terminal-state}, \cdot) = 0$
2: **repeat** for each episode
3: Initialize s
4: Choose a from s using policy derived from Q (e.g., ϵ-greedy)
5: **repeat** for each step of episode
6: Take action a, observe r, s'
7: Choose a' from s' using policy derived from Q (e.g., ϵ-greedy)
8: $Q(s, a) \leftarrow Q(s, a) + \alpha \left[r + \gamma Q(s', a') - Q(s, a)\right]$
9: $s \leftarrow s'$
10: $a \leftarrow a'$
11: **until** S is terminal
12: **until**

If all state-action pairs are visited infinitely often, Sarsa converges with probability 1 to an optimal policy and action-value function.

Off-policy TD method: Q-learning Now we consider an off-policy TD method called Q-learning [97]. This method learns the optimal action-value, regardless of the policy being followed. This method is defined by:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \left[r_{t+1} + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t)\right] \quad (2.19)$$

A minimal requirement of convergence to the optimal policy is that all state-action pairs are visited an infinite number of times. Under this assumption and a variant of the usual stochastic approximation conditions on the sequence of step-size parameters, Q has been shown to converge with probability 1 to the optimal action-values q_* [94].

The pseudocode of Q-learning is given in Algorithm 6.

On-policy and Off-policy. An important challenge of reinforcement learning is the exploration–exploitation dilemma. The agent has to learn the optimal policy

Algorithm 6 Q-learning (off-policy TD) for estimating $\pi \approx \pi_*$

1: Initialize $Q(s, a)$ for all $s \in S, a \in A(s)$, arbitrarily, and $Q(\textit{terminal-state}, \cdot) = 0$
2: **repeat** for each episode
3: Initialize s
4: **repeat** for each step of episode
5: Choose a from s using policy derived from Q (e.g., ϵ-greedy)
6: Take action a, observe r, s'
7: $Q(s, a) \leftarrow Q(s, a) + \alpha \left[r + \gamma \max_a Q(s', a) - Q(s, a)\right]$
8: $s \leftarrow s'$
9: **until** S is terminal
10: **until**

while behaving non-optimally, i.e., by exploring all actions. This dilemma brings about two main approaches for learning action values: on-policy and off-policy.

In on-policy methods, the agent learns the best policy while using it to make decisions. On the other hand, off-policy methods separate it into two policies. That is, the agent learns a policy different from what currently generates behavior. The policy being learned about is called the *target* policy, and the policy used to generate behavior is called the *behavior* policy. Since learning is from experience "off" the target policy, these methods are called off-policy learning. The on-policy methods are generally simpler than off-policy methods but they learn action values not for the optimal policy, but for a near-optimal policy that still explores [94]. The off-policy methods learn the optimal policy and they are considered more powerful and general but they are often of greater variance and are slower to converge [94].

While on-policy methods learn policies depending on actual behavior, off-policy methods learn the optimal policy independent of agent's actual behavior, i.e., the policy actually used during exploration. In Sarsa, it updates action values using a value of the current policy's action a' in next state s'. In Q-learning, it updates its action-value using the greedy (or optimal) action a' of next state s' but the agent selects an action by ϵ-greedy policy. In Q-learning, the target policy is the greedy policy and the behavior policy is ϵ-greedy policy.

2.4 Function Approximation Methods

Many classical reinforcement-learning algorithms have been applied to small finite and discrete state spaces and value functions are represented using a tabular form that stores the state(-action) values in a table. In such small and discrete problems, a lookup table represents all state-action values of a learning space. However, in many realistic problems with large and continuous state spaces, there are many more states than could possibly be entries in a table. In such problems, a major challenge is to represent and store value functions. Thus, the tabular methods typically used in reinforcement learning have to be extended to apply to such large problems, for example, using function approximation methods. The approximate value function is represented as a parameterized functional form with weight vector $\mathbf{w} \in \mathbb{R}^d$. $\hat{v}(s, \mathbf{w}) \approx v_\pi(s)$ denotes the approximate value of state s given weight

vector $\mathbf{w}$.

First we consider the squared difference between the approximate value $\hat{v}(s, \mathbf{w})$ and the true value $v_\pi(s)$ over the state space, i.e., the Mean Squared Value Error. Then, we study gradient-descent methods to minimize the error. Finally, we introduce linear function approximation based on the gradient-descent method.

Mean Squared Value Error. In tabular methods, learning at a certain state yields an update to the state's value function, though the values of all other states are left unchanged. That is, an update is applied only to the current state and it does not affect value functions of the other states. Each state-action value from a lookup table represents the true value function of one state-action pair. However, in approximation methods, the number of states is larger than the number of weights, the dimensionality of weight vector $\mathbf{w}$. Thus, an update at one state affects the estimated values of many other states and it is not possible that all state values are correctly estimated [94]. Updating at one state makes its estimated value more accurate, but it may make values of other states less correct because the estimated values of other states are changed as well. Hence, in the function approximation, rather than trying to make zero error of value functions for all states, we aim to balance the errors in different states [94]. For that, it is necessary to specify a state weighting or distribution $\mu(s) \geq 0, \sum_s \mu(s) = 1$ in order to represent how much we care about the error in each state s [94]. For example, the fraction of time spent in s may be used as $\mu(s)$. The squared difference between the approximate value $\hat{v}(s, \mathbf{w})$ and the true value $v_\pi(s)$ is averaged with weighting over the state space by μ. The Mean Squared Value Error, denoted $\overline{VE}$, is obtained:

$$\overline{VE}(\mathbf{w}) \doteq \sum_{s \in S} \mu(s) \left[v_\pi(s) - \hat{v}(s, \mathbf{w})\right]^2 \tag{2.20}$$

By the square root of $\overline{VE}$, we can measure roughly how much the approximate values differ from the true values [94].

Gradient-Descent Methods. We consider gradient-descent methods to minimize the mean squared error (Eq. (2.20)) on the observed data. The gradient-descent methods are commonly used in function approximation. The approximate value function $\hat{v}(s, \mathbf{w})$ is a differentiable function of the weight vector $\mathbf{w} \doteq (w_1, w_2, \ldots, w_d)$ for all $s \in S$. We assume that all the states are encountered equally in learning. At each time step, we update the weight vector $\mathbf{w}$. By the gradient descent method, the weight vector $\mathbf{w}$ is changed by a small amount in the direction that minimizes the $\overline{VE}$, the error between true value function under policy π, $v_\pi(s)$ and the approximate value function $\hat{v}(s, \mathbf{w})$.

$$\begin{aligned} \mathbf{w}_{t+1} &\doteq \mathbf{w}_t - \frac{1}{2}\alpha \nabla \left[v_\pi(s_t) - \hat{v}(s_t, \mathbf{w}_t)\right]^2 \\ &= \mathbf{w}_t + \alpha \left[v_\pi(s_t) - \hat{v}(s_t, \mathbf{w}_t)\right] \nabla \hat{v}(s_t, \mathbf{w}_t) \end{aligned} \tag{2.21}$$

where α is a positive step-size parameter and $\nabla \hat{v}(s_t, \mathbf{w}_t)$ is the vector of partial derivatives with respect to the elements of the weight vector:

$$\nabla \hat{v}(s_t, \mathbf{w}_t) \doteq \left(\frac{\partial \hat{v}(s_t, \mathbf{w}_t)}{\partial w_1}, \frac{\partial \hat{v}(s_t, \mathbf{w}_t)}{\partial w_2}, \ldots, \frac{\partial \hat{v}(s_t, \mathbf{w}_t)}{\partial w_d} \right). \tag{2.22}$$

If the update is done on a single example, the update is called a 'stochastic' gradient-descent update. When more than one example is used for an update, the gradient-descent method is called 'batch' and the batch update is obtained as follows:

$$\mathbf{w}_{t+1} \doteq \mathbf{w}_t + \alpha \sum_i \left[\left[v_\pi(s_i) - \hat{v}(s_i, \mathbf{w}_t) \right] \nabla \hat{v}(s_i, \mathbf{w}_t) \right] \tag{2.23}$$

where s_i is i^{th} state among all input states.

The pseudocode of gradient TD is given below.

Algorithm 7 Gradient TD

1: Input: the policy π to be evaluated
2: Input: a differentiable function $\hat{v} : S \times \mathbb{R}^d \to \mathbb{R}$ such that $\hat{v}(terminal, \cdot) = 0$
3: Initialize value-function weights $\mathbf{w}$ arbitrarily (e.g., $\mathbf{w} = 0$)
4: **repeat** for each episode:
5: Initialize s
6: **repeat** for each step of episode:
7: Choose $a \sim \pi(\cdot \mid s)$
8: Take action a, observe r, s'
9: $\mathbf{w} \leftarrow \mathbf{w} + \alpha \left[r + \gamma \hat{v}(s', \mathbf{w}) - \hat{v}(s, \mathbf{w}) \right] \nabla \hat{v}(s, \mathbf{w})$
10: $s \leftarrow s'$
11: **until** s' is terminal
12: **until**

Linear Function Approximation based on Gradient-Descent Method. The approximate value function of state s, $\hat{v}(s, \mathbf{w})$, is commonly represented in a linear function with the weight vector $\mathbf{w} \in \mathbb{R}^d$. State s is represented with a real-valued vector of features, called a feature vector, $\mathbf{x}(s) \doteq (x_1(s), x_2(s), \ldots, x_d(s))$. Each $x_i(s)$ is the value of a function $x_i : S \to \mathbb{R}$ and the value is called a feature of s. The functions x_i are also called basis functions in a linear function because they form a linear basis for the set of approximate functions [94]. Thus, $\hat{v}(s, \mathbf{w})$ is a linear function of features of the state s, with the weight vector $\mathbf{w}$. In linear methods, if the feature vector, $\mathbf{x}(s)$ is a d-dimensional vector, the weight vector, $\mathbf{w}$ is also a d-dimensional vector. The feature vector, $\mathbf{x}(s)$, and the weight vector, $\mathbf{w}$, have the same number of elements. Then the state-value function is approximated by the inner product between $\mathbf{w}$ and $\mathbf{x}(s)$:

$$\hat{v}(s, \mathbf{w}) \doteq \mathbf{w}^\intercal \mathbf{x}(s) \doteq \sum_{i=1}^{d} w_i x_i(s) \tag{2.24}$$

This approximate value function is linear in the weights and we refer to it as a linear function approximator.

The gradient-descent methods above are commonly used in linear function approximation. The gradient-descent–based update in state s_t is:

$$\mathbf{w}_{t+1} \doteq \mathbf{w}_t + \alpha \left[r_{t+1} + \gamma \hat{v}(s_{t+1}, \mathbf{w}_t) - \hat{v}(s_t, \mathbf{w}_t) \right] \nabla \hat{v}(s_t, \mathbf{w}_t) \tag{2.25}$$

where $\nabla\hat{v}(s,\mathbf{w})$ (Eq. (2.21)), the gradient of the approximate value function with respect to $\mathbf{w}$, is $\mathbf{x}(s)$.

We can extend the state-value function approximation, $\hat{s}(s,\mathbf{w})$, to action-value function approximation, $\hat{q}(s,a,\mathbf{w}) \approx q_*(s,a)$. Like the state-value function, the action-value function is approximated by linearly combining feature vector $\mathbf{x}(s,a)$ and weight vector $\mathbf{w}$:

$$\hat{q}(s,a,\mathbf{w}) \doteq \mathbf{w}^\intercal \mathbf{x}(s,a) \doteq \sum_{i=1}^{d} w_i x_i(s,a) \tag{2.26}$$

and the gradient-descent update based on Sarsa method is:

$$\mathbf{w}_{t+1} \doteq \mathbf{w}_t + \alpha \left[r_{t+1} + \gamma \hat{q}(s_{t+1}, a_{t+1}, \mathbf{w}_t) - \hat{q}(s_t, a_t, \mathbf{w}_t) \right] \nabla \hat{q}(s_t, a_t, \mathbf{w}_t). \tag{2.27}$$

Pseudocode for the complete algorithm is given as below.

Algorithm 8 Gradient Sarsa for Estimating $\hat{q} \approx q_*$

1: Input: a differentiable function $\hat{q} : S \times A \times \mathbb{R}^d \to \mathbb{R}$
2: Initialize value-function weights $\mathbf{w} \in \mathbb{R}^d$ arbitrarily (e.g., $\mathbf{w} = 0$)
3: **repeat** for each episode:
4: $\quad s, a \leftarrow$ initial state and action of episode (e.g.,ϵ-greedy)
5: $\quad$ **repeat** for each step of episode:
6: $\quad\quad$ Take action a, observe r, s'
7: $\quad\quad$ **if** s' is terminal: **then**
8: $\quad\quad\quad \mathbf{w} \leftarrow \mathbf{w} + \alpha \left[r - \hat{q}(s,a,\mathbf{w}) \right] \nabla \hat{q}(s,a,\mathbf{w})$
9: $\quad\quad\quad$ Go to next episode
10: $\quad\quad$ **end if**
11: $\quad\quad$ Choose a' as a function of $\hat{q}(s', \cdot, \mathbf{w})$ (e.g., ϵ-greedy)
12: $\quad\quad \mathbf{w} \leftarrow \mathbf{w} + \alpha \left[r + \gamma \hat{q}(s',a',\mathbf{w}) - \hat{q}(s,a,\mathbf{w}) \right] \nabla \hat{q}(s,a,\mathbf{w})$
13: $\quad\quad s \leftarrow s'$
14: $\quad\quad a \leftarrow a'$
15: $\quad$ **until**
16: **until**

Feature Construction. In linear function approximation, as we have seen before, the value is obtained by sums of features times corresponding weights. Its computation relies on features. Appropriate features help to correctly estimate values, but, if the features are selected improperly, it may cause poor performance.

Features should represent the state space of the environment and convey the information necessary to learn the environment's dynamics. Selecting appropriate features, i.e., feature engineering, remains a challenge because it requires domain-specific knowledge and a great engineering effort. Representational design is based only on the system designer's knowledge and intuition. In addition to the engineering problem, the linear form itself has a limitation that it cannot take into account any interactions between features [94]. For example, feature i can be good or bad depending on feature j. Linear methods assume that each feature is linearly independent of other features. Even with careful engineering, it is not possible for a

system designer to choose features with considering all interaction between features. Several works [82, 35, 36] have addressed this problem to construct features automatically. These methods are based on errors of the value function and add features that help improve the value estimation.

Geramifard et al. [35] introduce incremental Feature Dependency Discovery (iFDD) as an online feature expansion method in the context of a linear function approximation. Their method gradually creates features that help eliminate error of the value function approximation. The process begins with an initial set of binary features. Their method identifies all conjunctions of existing features as potential features and increases the relevance of each potential feature by the absolute approximation error. If a potential feature's relevance exceeds a predefined threshold, the feature is added to the pool of features used for future approximation.

The authors extend iFDD to the batch setting in [36] and prove that Batch-iFDD is a Matching Pursuit (MP) algorithm with its guaranteed rate of error-bound reduction. Like iFDD, Batch-iFDD does not require a large pool of features at initialization but expands the pool of potential features incrementally that are the conjunction of previously selected features. Batch-iFDD runs the least-squares TD (LSTD) algorithm to estimate the TD-error over all samples and then adds the most relevant feature to the feature set. Their empirical results show that Batch-iFDD outperformed the previous state of the art MP algorithm.

Even though features are constructed in an online manner and these methods overcome an imperfect initial selection of features, it is still crucial to provide good initial features because their feature constructions are based on the initial features.

2.5 Exploration

To maximize total reward, the agent must select the action with highest value (exploitation), but to discover such action it has to try actions not selected before (exploration). This exploration enables to experience other actions not taken before and it may increase the greater total reward in the long run because we would discover better actions. The trade-off between exploitation and exploration is one of the challenges in reinforcement learning [94]. We present three well-known exploration methods.

ϵ-greedy. The ϵ-greedy strategy is one of the most commonly used method. Most of the time, the agent selects an action with the highest estimated value, but with small probability ϵ selects an action uniformly at random. The drawback of ϵ-greedy is to choose equally among all actions.

Softmax. One alternative is the softmax (or Boltzmann) method. It gives weights to actions according to their value estimates. The good actions have exponentially higher probabilities of being selected. An action a is chosen with probability:

$$p = \frac{\mathrm{e}^{Q(s,a)/\tau}}{\sum_{a'} \mathrm{e}^{Q(s,a')/\tau}} \tag{2.28}$$

where $\tau > 0$, the temperature, is used for the degree of exploration. When $\tau \to \infty$, actions are selected randomly. When τ approach 0, actions are selected greedily.

It is not clear whether softmax action selection or ϵ-greedy action selection is better [94]. It depends on the task and on heuristics. ϵ parameter is easy to set with confidence but setting τ requires knowledge of the likely action values and of powers of e [94].

Optimistic Value Initialization. Another method commonly used in model-based learning (see Section 2.6) is optimistic value initialization, such as in the R-max method that gives the maximum reward $r_{\max}$ to unknown state-actions [9, 11, 28]. It encourages the agent to explore all states. Known and unknown state-actions are classified by the number of visits. For each time step, the agent behaves greedily. Another similar method is to add exploration bonuses for states with higher potential of learning [5] or with higher uncertainty [62, 63, 68, 69].

2.6 Model-Based and Model-Free Methods

In reinforcement learning, there are two main approaches to estimate state-action values: model-based and model-free. Model-based methods require a model of the environment such as dynamic programming and model-free methods learn without a model such as temporal-difference (TD) methods. A model simulates the environment's dynamics and it allows to inference how the environment will behave. The model signifies the transition function and the reward function in an MDP.

Model-based methods. Model-based methods learn the transition and reward models from interaction with the environment, and then use the learned model to calculate the optimal policy by value iteration. That is, a model of the environment is learned from experience and value functions are updated by value iteration over the learned model. By learning a model of the environment, an agent can use it to predict how the environment will respond to its actions, i.e., predict next state and next reward given a state and an action. If an agent learns an accurate model, it can obtain an optimal policy based on the model without any additional experiences in the environment. Model-based methods are more sample-efficient than model-free methods but exhaustive exploration is often necessary to learn a perfect model of the environment [53]. However, learning a model allows the agent to perform targeted exploration. If some states are not visited enough or uncertain to learn a model correctly, this insufficiency of information drives the agent explore more those state. Thus, optimistic value initialization is commonly used for exploration method (see Section 2.5). If the agent take an action a in state s, the action value $Q(s, a)$ is updated by the Bellman equation:

$$Q(s,a) \doteq R(s,a) + \gamma \sum_{s'} \Pr(s' \mid s,a) \max_{a'} Q(s',a') \tag{2.29}$$

where $0 \le \gamma < 1$ is the discount rate that determines the present value of future rewards. If $\gamma = 0$, the agent is only concerned with the immediate reward. The agent's action influences only the current reward. If γ approaches 1, the agent

considers future rewards more strongly on its action. The optimal value function Q^* can be obtained by iterating on the Bellman equation until it converges [94].

Transition and reward models are commonly learned by a maximum likelihood model. Suppose $C(s, a)$ is the number of times that action a is taken in state s and $C(s, a, s')$ is the number of times that taking action a in state s transitions to state s'. The transition probability from (s, a) pair to state s' is obtained by:

$$\Pr(s'|s, a) = T(s, a, s') = C(s, a, s')/C(s, a) \tag{2.30}$$

The reward model for (s, a) pair is also computed in a similar way:

$$R(s, a) = RSUM(s, a)/C(s, a) \tag{2.31}$$

where $RSUM(s, a)$ is the sum of rewards that the agent receives when taking action a in state s. The most well-known model-based algorithm is R-MAX [9]. Algorithm 9 shows the pseudocode of R-MAX.

Algorithm 9 R-MAX

```
1: s_r : an absorbing state
2: for all a ∈ A do
3:     R(s_r, a) ← R_max
4:     T(s_r, a, s_r) ← 1
5: end for
6: repeat
7:     Choose a = π(s)
8:     Take action a, observe reward r and next state s'
9:     C(s, a, s') ← C(s, a, s') + 1
10:    C(s, a) ← C(s, a) + 1
11:    RSUM(s, a) ← RSUM(s, a) + r
12:    if C(s, a) ≥ m then
13:        R(s, a) ← RSUM(s, a)/C(s, a)
14:        for all s' ∈ C(s, a, ·) do
15:            T(s, a, s') ← C(s, a, s')/C(s, a)
16:        end for
17:    else
18:        R(s, a) ← R_max
19:        T(s, a, s_r) ← 1
20:    end if
21:    Update Q-values using VI
22:    s ← s'
23: until converge
```

Most model-based methods assume a Dynamic Bayesian Network (DBN) transition model, and that each feature transitions independently of the others [54]. In Section 3.3, we study the factored MDP problem [7, 8, 23] that uses the DBN formalism and present an algorithm that learns the DBN structure of transition model.

Model-free methods. Model-free methods improve the value function directly from observed experience and do not rely on the transition and reward models. Value functions are learned by trial and error. These methods are simple and can have advantages when a problem is complex so that it is difficult to learn an accurate model. However, model-free methods require more samples than model-based to learn value functions. When an action a is taken in state s, the agent receives a reward r, and moves to the next state s', the action value $Q(s, a)$ based on Q-learning [98] is updated as follows:

$$Q(s, a) \doteq Q(s, a) + \alpha \left[r + \gamma \max_{a'} Q(s', a') - Q(s, a) \right] \tag{2.32}$$

where $0 < \alpha \leq 1$ is the step-size parameter and $0 \leq \gamma < 1$ is the discount rate. The step-size parameter α influences the rate of learning. When $\alpha = 1$, the agent considers only the most recent information for learning.

In Section 3.2, we study a model-free method and apply Q-learning algorithm to a real taxi routing problem. In Section 3.3, we study model-based methods for a factored MDP problem.

2.7 Priority-Based Value Iteration

Value Iteration (VI) is a dynamic programming algorithm that performs complete sweeps of updates across the state space until convergence to optimal policies (see Section 2.2.2). VI is a simple algorithm but computationally expensive. If the state space is large, the computational cost of even one single sweep is also immense and multiple sweeps for convergence can cause extreme computational cost. This is due to some inefficiencies of updates in VI. First, some updates are useless. VI updates the entire state space at each iteration even though some updates do not change value functions. In fact, if a state value is changed after its update, then the values of its predecessor states are also likely to be changed but the values of other states that do not lead into the state remain unchanged. Updating these states have no effect. Second, updates are not performed in an optimal order. Performing updates on a state after updating its successors can be more efficient than that in an arbitrary order. That is, it is better to propagate backward from any state whose value has changed to states that lead into the state. If the updates are not performed in a good order, some states may need to be updated redundantly to converge.

When the DP algorithms do not sweep the state space for each iteration, we call them asynchronous (or sweepless) DP algorithms [94]. In these algorithms, the values of states can be updated in any order. Asynchronous DP methods give great flexibility in selecting states to update [94]. Algorithms do not need to get locked into long sweeps over the entire state space and by the flexibility of selecting the states we can improve the convergence rate. Eventual convergence of asynchronous DP algorithms is guaranteed as long as the algorithms continue to update the values of all the states. It is shown in the MDP literature that the performance of value iteration can be significantly improved by ordering updates intelligently rather than by arbitrary order updates.

Algorithm 10 Prioritized sweeping algorithm

1: Initialize $Q(s,a)$, $Model(s,a)$, for all s,a, and $PQueue$ to empty
2: **loop**
3: $\quad s \leftarrow$ current (non-terminal) state
4: $\quad a \leftarrow policy(s,Q)$
5: $\quad$ Execute action a, observe resultant reward, r, and state, s'
6: $\quad Model(s,a) \leftarrow r, s'$
7: $\quad p \leftarrow |r + \gamma \max_a Q(s',a) - Q(s,a)|$
8: $\quad$ **if** $p > \theta$ **then**
9: $\quad\quad$ insert s,a into $PQueue$ with priority p
10: $\quad$ **end if**
11: $\quad$ **repeat**
12: $\quad\quad s,a \leftarrow \text{first}(PQueue)$
13: $\quad\quad r,s' \leftarrow Model(s,a)$
14: $\quad\quad Q(s,a) \leftarrow Q(s,a) + \alpha\,[r + \gamma \max_a Q(s',a) - Q(s,a)]$
15: $\quad\quad$ **for** all $\bar{s},\bar{a}$ predicted to lead to s **do**
16: $\quad\quad\quad \bar{r} \leftarrow$ predicted reward for $\bar{s},\bar{a},s$
17: $\quad\quad\quad p \leftarrow |\bar{r} + \gamma \max_a Q(s,a) - Q(\bar{s},\bar{a})|$
18: $\quad\quad\quad$ **if** $p > \theta$ **then**
19: $\quad\quad\quad\quad$ insert $\bar{s},\bar{a}$ into $PQueue$ with priority p
20: $\quad\quad\quad$ **end if**
21: $\quad\quad$ **end for**
22: $\quad$ **until** n times while $PQueue$ is not empty
23: **end loop**

Prioritized Sweeping (PS) is a well-known prioritization method introduced by Moore et al. [78]. The principal idea of PS is to update backwards from states which values have changed to the states that lead into them, i.e., its predecessors. Prioritizing updates that are expected to cause large value changes, it has an effect of propagating values backwards from states with a relatively high state-values and it enables to reduce the number of updates. PS maintains a priority queue to order updates and Bellman error is commonly used as the priority metric. The Bellman error is the difference between the current estimated value and the estimated value after applying the Bellman operator. PS can start from any state not just a goal state. If the priority of the state-action pair is greater than a certain threshold, then the pair is stored in the queue with its priority. For each iteration, a state-action pair (s,a) with the highest priority is removed from the queue and its value function is updated. For each predecessor of (s,a), its priority value is computed. If the priority greater than some threshold then the predecessor is inserted in the priority queue. Algorithm 10 shows the Prioritized Sweeping algorithm.

Prioritized methods help to improve considerably the performance of value iteration by reducing the number of updates, but maintaining the priority queue may result in an excessive overhead. It is tackled in a different ways in the literature such as prioritizing updates without priority queue [21], stationary update orders [20, 25], partitioning the state space [102, 101], etc.

Dai et al. [21] show that the overhead for maintaining a priority queue can be

greater than its advantages. The authors introduce a prioritized value iteration algorithm that does not require a priority queue. The algorithm starts with goal state and performs a breadth-first (or depth-first) traversal of the transpose of the state space graph. States are backed up in the order they are encountered. The order of updates is sub-optimal but its smaller overhead allows to converge faster than other existing methods.

Topological value iteration (TVI) [20] performs updates in the order of a causal relation among layers of states. While previous prioritized algorithms dynamically updating state priorities, TVI uses stationary update order. Dibangoye et al. [25] propose an improved version of TVI, iTVI. iTVI builds mapping of states space S into a metric space (S, d) and performs updates in the order of metric d, the distance from the start state to a state, which is causal relations among states measured by standard shortest-path techniques like Dijkstra.

McMahan et al. [71] merge some features of Dijkstra's algorithm and value iteration. Their algorithm called improved Prioritized Sweeping (IPS) reduces to Dijkstra's algorithm when given a deterministic MDP. IPS is suitable to solve short path lengths problems.

[34] also proposes a prioritized value iteration algorithm based on Dijkstra's algorithm to solve stochastic-shortest-path problems. Different from IPS, their method can deal with multiple start and goal state and it has guaranteed convergence to the optimal solution.

Wingate et al. [102] present three enhancements such as prioritizing updates, partitioning, and variable reordering to accelerate value and policy iteration. As a priority metric, they introduce a new metric, $H2$, defined as the value of the state plus the Bellman error and compare it with Bellman error. The proposed prioritized and partitioned algorithm selects a high-priority partition p, update state values in the partition, and then reprioritizes any partition which depends upon anything in p. It may reorder the states in the partition such that for each sweep. The state ordering is computed during initialization.

In the Dyna [93] framework, planning process is performed in addition to learning. While learning updates the value function by interacting with the real environment, planning performs some fixed number of value-function updates in a model of environment that simulates a real environment. Queue-Dyna [83] improves Dyna [93] by prioritized value-function updates in planning process. The priority is determined by the prediction difference of Q-values between a state-action pair and its successor state. If the difference is larger than a predefined threshold, then the state-action pair is placed in the queue. In planning process, a state-action pair with the highest priority is removed from the queue and updated with simulated experiences. If the state-value estimates of its predecessors or successors change, its all transitions become update candidates.

PAC(Probably Approximately Correct)-MDP learning is one of the approaches to exploration in RL. Its exploration strategy guarantees that with high probability the algorithm performs near optimally for all but a polynomial number of time steps [46]. The well-known algorithms are E^3 [59] and R-max [9]. In PAC-MDP algorithms, when a state-action pair is visited sufficiently many times, it is considered as known by the agent. Whenever a new state-action pair becomes known to the agent, planning step is executed. In [46], the authors propose several approaches

to improve the planning step in PAC-MDP learning. They reduce the number of planning steps by extending the notion of known state-action pairs by a notion of a known state. In action value updates, BAO updates are proposed, updating only the best action of each state instead of updating all actions of a given state. An extension to the prioritized sweeping algorithm is presented which adds only policy predecessors to the priority queue instead of all predecessors of a given state. In [47], they analyze theoretically and empirically on prioritization of Bellman updates. They show empirical evidence that ordering of updates in standard value iteration can significantly influence its performance.

2.8 Non-Stationary Environment

The environment we have seen so far is assumed to be stationary, that is, the environment dynamics does not change over time. However, this is not realistic in many real-world problems. For example, in a traffic lights control problem, traffic patterns vary over time. We often encounter reinforcement learning problems that are effectively non-stationary [94]. Non-stationary problems are the most common in reinforcement learning. In non-stationary problems, transition and reward probabilities are time-dependent. The true values of the actions and the agent's policy change over time. In such cases, since the estimates continue to vary in response to the most recently received rewards, one of the most popular methods is to use a constant step-size parameter so that it can give more weight to recent rewards than to long-past rewards [94].

In fact, even in the stationary environment, the problems are non-stationary in the early stage of learning because the value of actions are undergoing learning. As the agent interacts with its environment, it learns incrementally the good policies and value functions from each observed experience. As we have discussed in Section 2.2.3, when the value function is changed for the agent's current policy, the policy is also improved with respect to the current value function. Notwithstanding the non-stationarity in the early learning, as the interaction between two processes continues, value functions and policies become optimal.

When we use the traditional RL methods in non-stationary environment, it can cause an inefficiency in learning. The agent keeps track of dynamics changes and learns good policies corresponding to the current dynamics. When the environment dynamics changes, the knowledge that has been learned becomes useless and the agent has to learn the new dynamics. The problem is that even if the environment reverts to the previously learned dynamics, the agent has to learn it again because the learned knowledge in the past is not laid aside. Some previous works have been addressed non-stationary problems. In most proposed methods, it is assumed some degree of regularity in dynamics changes, and that the changed dynamics last long enough so that the agent can learn the changes.

Choi et al. [18] introduce hidden-mode Markov decision process (HM-MDP) that are a subclass of partially observable Markov decision processes (POMDP) to solve reinforcement learning problems in non-stationary environments. A hidden-mode model is defined as a finite set of MDPs that share the same state space and action space, with possibly different transition functions and reward functions. The authors

assume that the environment is in exactly one of the modes at any given time. Based on the hidden-mode model, a variant of the Baum–Welch algorithm is proposed to capture environmental changes and learn different modes of the environment.

RL-CD [19] is also a method for solving reinforcement learning problems in non-stationary environments. The authors assume that the environment is non-stationary but it is divided into partial models estimated by observing the transitions and rewards. The algorithm evaluates how well the current partial model can predict the environment using a quality of a model that is a value inversely proportional to its prediction error. For each time step, the model with the highest quality is activated. If there is no model with quality higher than minimum quality threshold, a new model is created. Each environment dynamics is called a context. Whenever the currently active model is replaced, they consider a context change is detected. The algorithm starts with only one model and then incrementally creates new ones as they become necessary. In experiments, two non-stationary environments are used: ball catching and traffic lights control. In ball catching scenario, the movement of the ball change over time and in traffic scenario, three traffic patterns with different car insertion distributions are used to build the non-stationarity of the environment. The experimental results show that the performance of RL-CD performs better than two traditional RL algorithms, Q-Learning and Prioritized Sweeping (PS).

TES [79] is an online framework that transfers the world dynamics in heterogeneous environments. The agent learns world models called views and collects them into a library to reuse in future tasks. A view is a decomposition of the transition function that consists of a structure component and a quantitative component. The structure component picks the features relevant to an action and the quantitative component defines how these features should be combined to approximate the distribution of action effects. When the agent encounters a new task in a new environment, it selects a proper view from the library or adapts to new tasks or environments with completely new transition dynamics and feature distributions. In experiments, it is shown that TES is adapted to multi-dimensional heterogeneous environments with a small computational cost.

Rana et al. [85] apply two model-free approach, Q-learning and Q-learning with eligibility trace $Q(\lambda)$, to solve the dynamic pricing problem with finite inventory and non-stationary demand. The agent aims to maximize the revenue for selling a given inventory by a fixed deadline. The agent learns the demand behavior and, based on that, it optimizes their pricing strategies. In experiments, it is shown that the $Q(\lambda)$ outperforms the standard Q-learning algorithm. If the initial Q-values can be set to the best estimated demand function, the learning converges faster than when no prior knowledge of demand is assumed. $Q(\lambda)$ performs particularly well in situations where the demand between successive days exhibits self-correlation.

Bayesian policy reuse (BPR) [89] is a Bayesian framework to determine quickly the best policy for a novel task by reusing a policy from a library of existing policies.

BPR+ [52] extends BPR in a multi-agent setting to deal with non-stationary opponents. Thus, the tasks in BPR are opponent strategies in BPR+ and the policies in BPR are optimal policies against those stationary strategies in BPR+. While BPR assumes knowledge of all possible tasks and optimal policies is given a priori, BPR+ learns new models in an online fashion without prior knowledge. The learning agent detects that the current policies do not perform optimally and then learns

incorporating models of new strategies. BPR+ computes the probability of the rewards under the known models, and if this probability is lower than a threshold for some number of rounds, BPR+ considers that a new model is detected. The authors assume that the opponent will not change of strategies during a number of rounds. Then, the new model, i.e., opponent strategy, is learned by value iteration and its performance models are also updated accordingly to be able to detect switches to either a new or a previously known strategy. Experimental results show that BPR+ is capable of efficiently detecting opponent strategies and reacting quickly to behavior switches.

Part II

Applications

Chapter 3

Model-Free and Model-Based Methods

3.1 Introduction

In this chapter, we study two main approaches for solving reinforcement learning problems: model-free and model-based methods.

We first study a model-free method that learns directly from observed experiences without a model. We apply Q-learning [98], one of the widely-used model-free methods, to a real taxi routing problem with a customized exploration and exploitation strategy. In experiments, we investigate two important parameters of Q-learning – the step size and discount rate. We also investigate the influence of trade-off between exploration and exploitation on learning.

Then, we turn to a model-based method that learns transition and reward models of the environment. In particular, we address the factored MDP problem [7] where a state is represented by a vector of n variables. Most model-based methods are based on Dynamic Bayesian Network (DBN) transition models. Our algorithm learns the DBN structure including synchronic arcs and uses decision trees to represent transition functions.

3.2 Learning without Models

In this section, we study a model-free method. One of the widely-used model-free methods is Q-learning [98]. We apply Q-learning algorithm to a real taxi routing problem. We demonstrate that a reinforcement learning algorithm is able to progressively learn optimal actions for the routing to passenger pick-up points of an autonomous taxi in a real scenario at the scale of the city of Singapore. To improve action selection strategy, we present a customized exploration and exploitation strategy for the taxi problem. While model-free methods do not learn transition and reward models, they use two important parameters such as step size α and discount rate γ that influence the learning. We quantify the influence of the parameters on effectiveness: step size, discount rate, and trade-off between exploration and exploitation.

3.2.1 Background and Related Work

Most studies addressing the taxi routing problem focus on providing the fastest route and a sequence of pick-up points [84] by mining historical data [105, 103, 104, 84]. Yuan et al. [103] uses road segments and travel time clustering to find the fastest driving route. The authors build a landmark graph to model the traffic pattern and provide the time-dependent fastest route to a given destination. They then present in [104] a recommendation system for taxi drivers and passengers based on detecting parking places by clustering road segment extracted from GPS trajectories. The system recommends a parking place with high probability to get a passenger and suggests parking places or road segments where passengers can find vacant taxis. Qu et al. [84] propose a method to recommend an entire driving route for finding passengers instead of a sequence of pick-up points. They develop a graph representation of a road network by mining the historical taxi GPS traces and generate a cost-effectively optimal driving route for finding passengers. Those models rely on the availability of accurate historical data and trajectories. They might not be suitable for dynamical environments such as that of an autonomous taxi looking for optimal passenger pick-up points.

Reinforcement learning [94] has the potential to continuously and adaptively learn from interaction with the environment. Q-learning [98] is a widely used method because of its computational simplicity. In Q-learning, one does not require a model of transition functions and reward functions but learns directly from observed experience. In this study, we apply Q-learning algorithm to a real taxi routing problem.

While taxi routing has often been used as the example application for reinforcement learning algorithms, it often remained relegated to toy or small scale examples, as it is the case of the seminal 5×5 grid introduced in [26] and used for experimental purposes in [37, 53, 38, 29]. Learning pick-up points is a somewhat new application of reinforcement learning.

3.2.2 Q-learning for Taxi Routing

We assume an autonomous taxi agent does not know about the city and that the car moves completely depending on the estimated action values of reinforcement learning. The aim of this application is that the autonomous taxi decides where to go in order to pick up a passenger by learning both the values of actions given a state and the existence probability of passengers.

The learning agent takes an action a in state s, receives a reward r, and moves to the next state s'. With Q-learning (see Section 2.3), the estimated value of taking action a in state s, denoted $Q(s,a)$, is updated as:

$$Q(s,a) \doteq Q(s,a) + \alpha \left[r + \gamma \max_{a'} Q(s',a') - Q(s,a) \right]. \tag{3.1}$$

We call an episode a series of steps until the agent finds a passenger. For the first episode, the taxi located at a random position moves according to its policy. The episode ends when the taxi finds a passenger. Then, it moves to the passenger's destination and starts a new episode. As the taxi moves it receives rewards and updates its action-value and the existence probability. The road network is discretized

Algorithm 11 Taxi Routing for Learning Pick-up Points

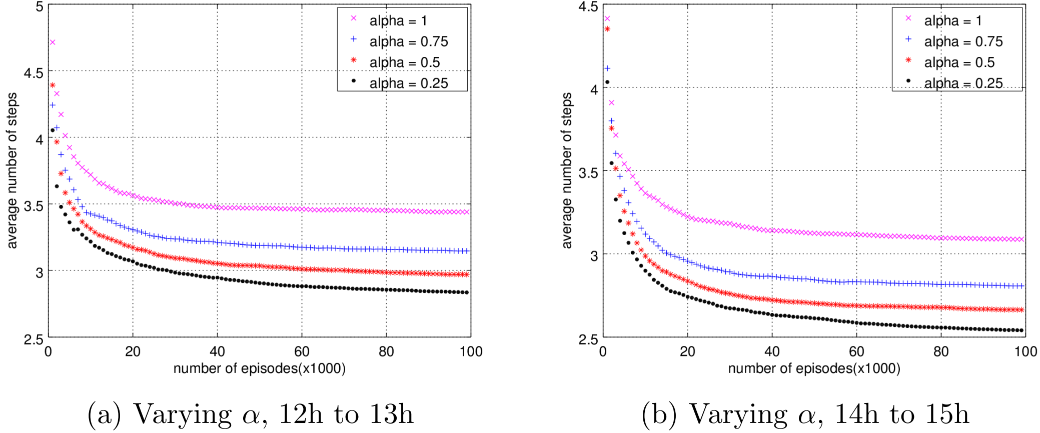
```
1: Initialize Q(s, a), existence probability of passengers p
2: repeat
3:     repeat
4:         if greedy then
5:             V ≐ { a ∈ A | Q(s, a) ≥ max_{a'} Q(s, a') − η }
6:             if |V| > 1 then
7:                 Select action a with highest probability p
8:             end if
9:         else /* not greedy */
10:            Select action a uniformly at random
11:        end if
12:        Take action a, obtain reward r, observe next state s'
13:        Q(s, a) ≐ Q(s, a) + α [r + γ max_{a'} Q(s', a') − Q(s, a)]
14:        Increment visit count on s'
15:        Update existence passenger probability p(s')
16:        if passenger found in s' then
17:            Increment found count on s'
18:            s becomes the end of the passenger route from s'
19:        else
20:            s ≐ s'
21:        end if
22:    until a passenger is found
23: until algorithm converges
```

and the movements correspond to steps in the discretized network. At each step, the taxi learns where passengers are likely to be located.

The Taxi Routing algorithm for learning pick-up points is outlined in Algorithm 11. According to the ϵ-greedy policy, an action a is selected in a given state s.

The action selection rule selects the action with the maximum estimated action value (greedy action). However, with this rule, the algorithm ignores other actions that, although they have slightly lesser value, may lead to a state having a higher chance to pick up a passenger. Hence, instead of selecting one greedy action, we loosen the selection condition by setting a lower bound below the maximum value in order to choose from more potentially valuable candidate actions (Line 5). The candidate actions are compared with existence probabilities of passengers in their corresponding states (Line 7). We later refer to the algorithm with this selection strategy as *Q-learning using LB/Proba*. This probability comparison is very effective when actions share the same value ($Q(s, a_1) = \ldots = Q(s, a_n)$). In this case, we originally select one action at random because we consider all actions are the same. In fact, they may not be the same if one causes to move to a state with very high existence probability of passengers. Comparing probabilities reduce this kind of mistake.

After taking an action, we update the action-value in the current state s with reward r and next state s'. As we visit a new state s', the visit and found counts are incremented and the existence probability of passengers is also recalculated. We



(a) Varying α, 12h to 13h (b) Varying α, 14h to 15h

Figure 3.1 – Average number of steps with different step-size α

repeat this procedure until we find a passenger.

3.2.3 Performance Evaluation

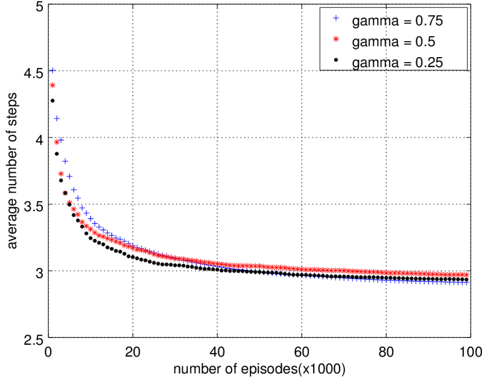
For the sake of simplicity, in this work, we present the results for a map discretized into cells of 0.01 degree longitude $\times$ 0.01 degree latitude (about 1.1km $\times$ 1.1km) forming a 38×20 grid. At each cell of the grid, eight actions are possible: up, down, right, left, and diagonally. A step is the movement from one cell to an adjacent one. Although such a representation does not capture several natural constraints on the traffic, it is sufficient, with limited loss of generality, to evaluate the effectiveness of the algorithm.

Since popular pick up points generally depend on the time of the day, we run the experiments for selected time intervals. Here, we present the results for two off-peak hours (12h to 13h and 14h to 15h), but we obtain comparable results for other time slots. At each episode we select 300 passengers according to actual geographical distribution in the given time interval in a dataset of taxi pickups and drop-offs for a fleet of one thousand taxis for one month in Singapore.

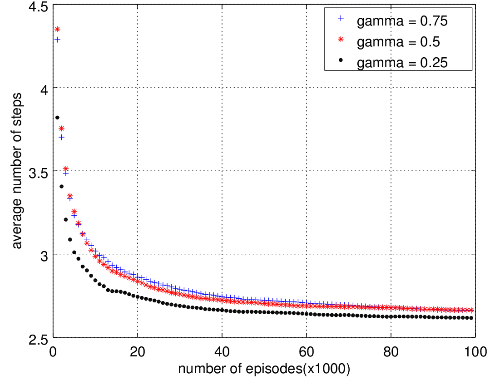
We first look into the impact of the step-size parameter α, the discount rate γ, and the probability of exploration ϵ. We evaluate how these parameters influence the learning performance with ordinary Q-learning. We compare the average number of steps. The average steps are calculated at every 100 episode by dividing the total steps from the first episode to the last by the total number of episodes.

Figures 3.1a–3.1b show the average number of steps with different step-size parameter α values for different time intervals. We compare four different α with a fixed γ $(= 0.5)$ and ϵ $(= 0.3)$. For all the time intervals, as the α is smaller, the average number of steps also decreases. Lower step-size values perform better. This indicates that accumulated experience affects value estimation more significant than recent experience, i.e., that the problem is indeed stochastic.

For the discount rate γ experiment, we fixed α $(= 0.5)$ and ϵ $(= 0.3)$ and changed the γ. The average number of steps with different γ values for different time intervals are shown in Figures 3.2a–3.2b. In Figure 3.2b, the lowest γ $(= 0.25)$ performs better. This means that immediate rewards are more important than future rewards.

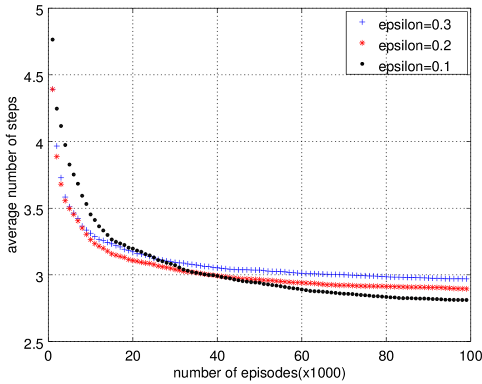


(a) Varying γ, 12h to 13h

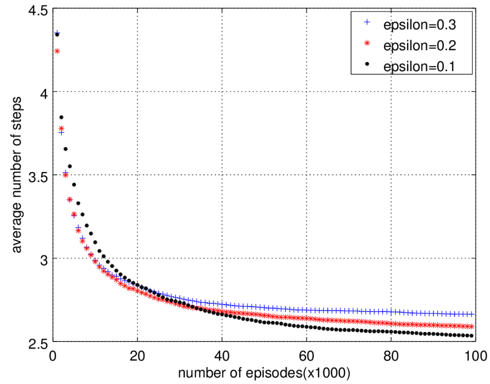


(b) Varying γ, 14h to 15h

Figure 3.2 – Average number of steps with different discount rate γ



(a) Varying ϵ, 12h to 13h



(b) Varying ϵ, 14h to 15h

Figure 3.3 – Average number of steps with different ϵ

In Figure 3.2a, as episodes continue, a higher γ (= 0.75) is slightly better. In Figure 3.2a, relatively longer step counts than those of the other time intervals are needed to achieve a goal. In this case, future rewards are more significant than current rewards.

Figures 3.3a–3.3b show the average number of steps with different ϵ values for different time intervals, given $\alpha = 0.5$ and $\gamma = 0.5$. The average number of steps for three cases first decreases dramatically and then converges gradually. For all the time intervals, when ϵ is 0.1, the average number of steps is bigger than the other cases in early episodes but it dominates after about 30,000 episodes. At the beginning, exploration is more effective and relatively inexpensive. Eventually, sufficient knowledge is accumulated and exploitation is worthy.

In the experiments, we saw how parameters α, γ, and ϵ behave in learning. While the optimal value of the parameter ϵ does not depend on the domain, the values of parameters α and γ depend on intrinsic properties of the state space.

We now compare Q-learning using LB/Proba (our algorithm) with ordinary Q-learning. For experiments, we select three parameter values shown in the previous section. The step-size parameter α is set to 0.25 because low learning rate is appro-

priate to our problem. The probability of exploration ϵ is set to 0.1. Since loosened selection for maximum action has the effect of exploration, high ϵ is not needed. We take $\eta = 0.01$ to set the lower bound on the maximum action value per state. Two algorithms are compared by varying the discount rate γ value.

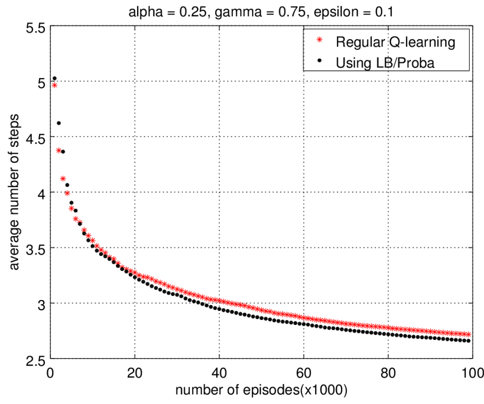
In Figures 3.4a–3.4d, when $\gamma = 0.75$ or 0.5, Q-learning using LB/Proba converges faster than Q-learning. On the other hand, when γ is 0.25 (Figures 3.4e–3.4f), Q-learning performs similarly well or slightly better than Q-learning using LB/Proba. These experiments show that when the learning rate α is low and the discount rate γ approaches 1, Q-learning using LB/Proba outperforms Q-learning. In other words, it has to accumulate much experience for value prediction and it considers future rewards more strongly. The reason is that Q-learning using LB/Proba depends on existence probability of passengers that requires enough experience and that is more related to long-term high rewards.

3.2.4 Demonstration Scenario

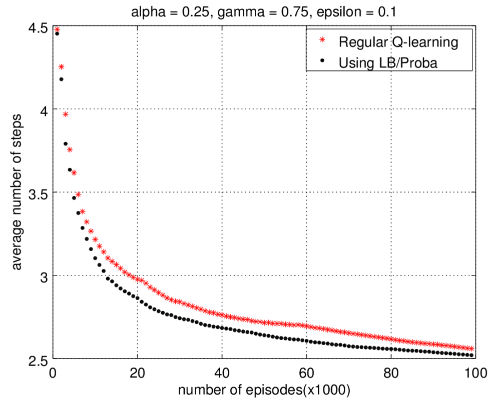
Figure 3.5 shows the initial screen of our pick-up point learning system. The red dot is a taxi and flags are passengers. Passengers' positions are based on a real dataset of Singaporean taxi trajectories.

In Figure 3.6, the user interface for simulation consists of two parts. The upper one is for the configuration of simulation and the lower one is for displaying the learning on the map. Before starting the simulation, the user can configure the properties such as episode count and exploration percentage (ϵ). The episode count is used for consecutive executions. The exploration percentage defines how often we choose a random selection. The user starts simulation in three ways: manually move one step by 'One Step' button, automatically execute with a fixed number of episodes by 'Start Driving' button, and repeat 100 times the fixed number of episodes by 'Experiment' button. 1) Manually: Every time the user clicks the button, the taxi moves on the next cell according to the learning policy. The taxi moves are traced on the map by a green line that connects the current position and the next position (Figure 3.6). The next taxi position is displayed with a green dot. 2) Automatically: It enables continuous learning with the fixed number of episodes. If the user sets to 100 episodes, the taxi does the pick-up learning 100 times. One episode means that the taxi finds a passenger. With 100 episodes, the taxi finds 100 passengers. 3) Simple experiment: We repeat 100 times the automatic learning explained above. If the user sets to 100 episodes and executes this experiment, it executes 100 (episode) $\times$ 100 (times), i.e., in total 10,000 episodes are executed. Experiment results are shown by 'Average Steps'.

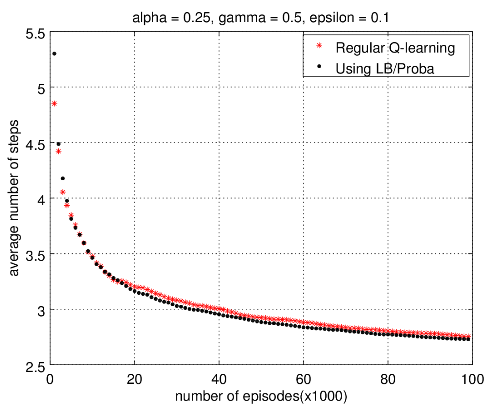
The user verifies the experiment result in the interface such as the average steps and the existence probability of passengers. The average steps is calculated at every 100 episodes and it is obtained by dividing the total steps from the first episode to the last by the total number of episodes. The user can see the average steps on the left of the 'Experiment' button (Figure 3.6). By clicking 'Average Steps', the user verifies a list of average steps calculated at every 100 episodes. With this list, the user can also visualize a chart of average steps. The existence probability of passengers is also an experiment result that the user can see on the map. The probability is calculated every time the taxi moves and it is displayed on each cell



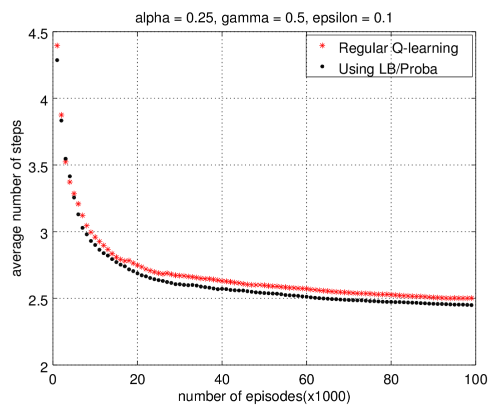
(a) Regular Q-learning vs LB/Proba, $\gamma = 0.75$, 12h to 13h



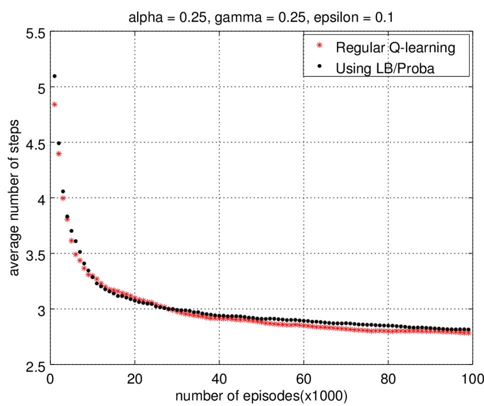
(b) Regular Q-learning vs LB/Proba, $\gamma = 0.75$, 14h to 15h



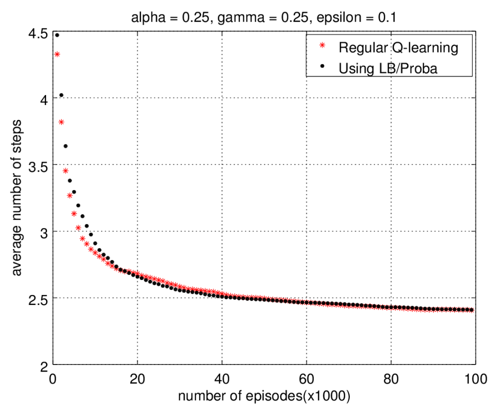
(c) Regular Q-learning vs LB/Proba, $\gamma = 0.5$, 12h to 13h



(d) Regular Q-learning vs LB/Proba, $\gamma = 0.5$,14h to 15h



(e) Regular Q-learning vs LB/Proba, $\gamma = 0.25$, 12h to 13h



(f) Regular Q-learning vs LB/Proba, $\gamma = 0.25$, 14h to 15h

Figure 3.4 – Regular Q-learning vs LB/Proba: Average number of steps as the number of episodes increases.

of the map. After experiment, the user can visualize the learned probability by a heatmap. Depending on the probability, the cells are filled in red, yellow, green, or blue color. The most probable places are in red color, less probable places in yellow or green, and the least probable places are in blue.

Through the simulation, the user can see an interesting behavior that the taxi moves inside road network areas (Figure 3.6). As experiments are repeated, the taxi traces draw features of Singaporean geography. That is obtained by reinforcement learning, not deliberately programmed in the system.

3.3 Learning Models

In the previous section, an agent improves the value function directly from observed experience and does not rely on the transition and reward functions. In contrast, model-based methods learn the transition and reward models and use these models to update value functions (see Section 2.6). Most model-based methods are based on Dynamic Bayesian Network (DBN) transition models and each feature's transition is assumed to be independent from that of the others [7, 8, 23, 54]. In this section, we study a model-based method. In particular, we address the factored MDP problem [7, 8, 23] whose state is represented by a vector of n variables. As the size of the state spaces increases, representing MDPs with large state spaces is challenging in reinforcement learning. Factored MDPs using the DBN formalism is one approach to represent large MDPs compactly. We propose an algorithm that learns the DBN structure including synchronic arcs and uses decision trees to represent transition functions. We evaluate the efficiency of our algorithm by comparing with other algorithms.

3.3.1 Background: Factored MDP

A factored MDP first proposed by Boutilier et al. [7] is an MDP where the state is represented by a vector of n variables. The transition function in the factored MDP is described by a DBN. Learning the structure of the DBN transition function is called structure learning [54]. Dynamic Bayesian Networks (DBN) are Bayesian Networks (BN) for time series modeling. An example of DBN is illustrated in Figure 3.7. Like in a BN, nodes represent variables and edges represent dependencies between two variables but DBNs include a temporal dimension: a state at time $t+1$ depends only on its immediate past, i.e., states at time t.

The DBN model determines which features are relevant or not for the predictions of certain features. It represents transition functions compactly and reduces the computation complexity. It is also effective in exploration to the unvisited states. Instead of exploring every state-action, the agent can make reasonable predictions about unseen state-action pairs.

In factored MDPs, a state is characterized by a finite set of random variables $s = \{x_1, x_2, \ldots, x_n\}$. We use x_i^t to denote the variable x_i at time t. The transition function from s^t to s^{t+1} after taking action a is defined by the conditional probability $\Pr(s^{t+1} \mid s^t, a)$. To simplify notation, we omit action a and use the notation $\Pr(s^{t+1} \mid$

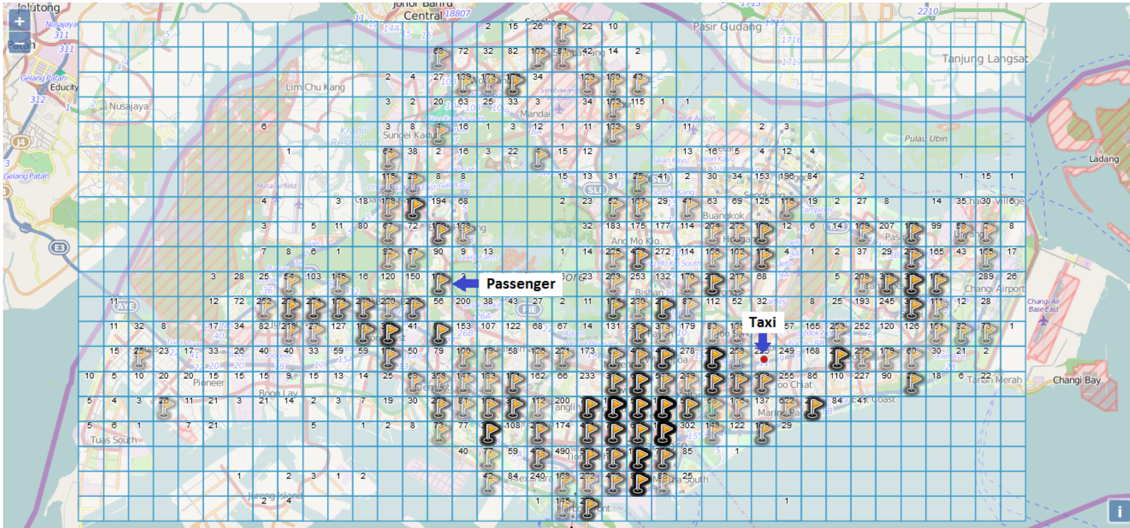


Figure 3.5 – The user interface

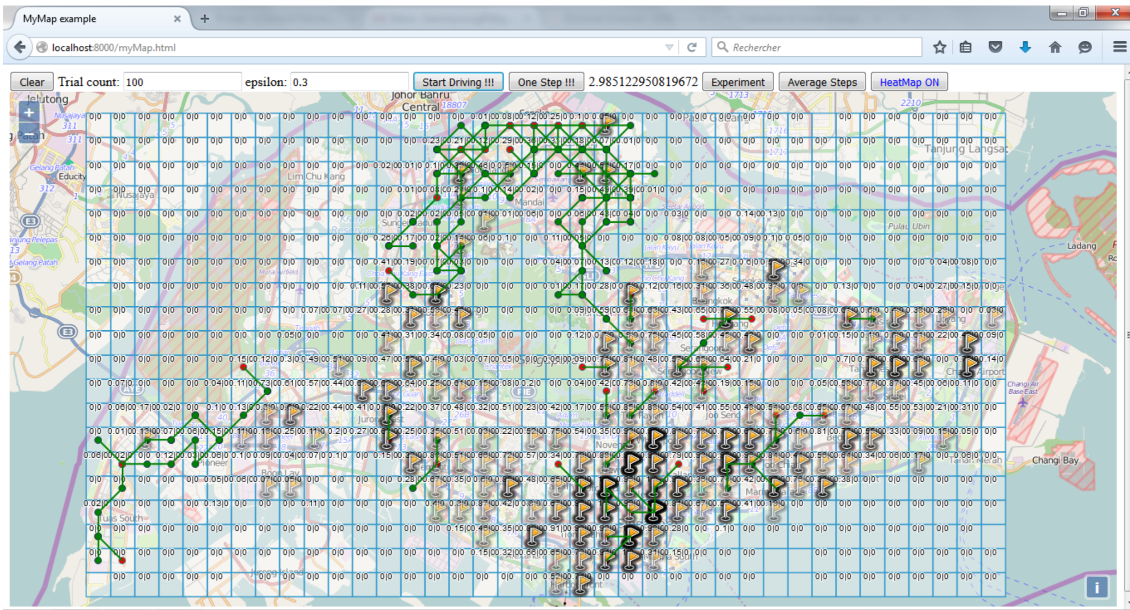


Figure 3.6 – Traces of taxi moves

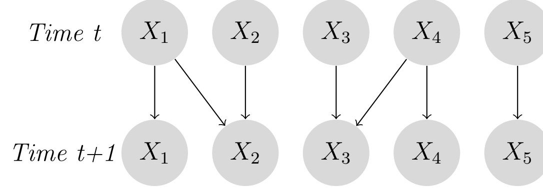


Figure 3.7 – Example of Dynamic Bayesian Network (DBN)

s^t). By the Bayes rule, the probability is decomposed as follows:

$$\begin{aligned}\Pr(s^{t+1} \mid s^t) = \Pr(x_1^{t+1} \mid s^t, x_2^{t+1}, x_3^{t+1}, \dots x_n^{t+1}) \\ \Pr(x_2^{t+1} \mid s^t, x_3^{t+1}, x_4^{t+1}, \dots x_n^{t+1}) \\ \dots \Pr(x_n^{t+1} \mid s^t)\end{aligned} \tag{3.2}$$

In many cases, it is assumed that there is no synchronic arc, i.e., an arc from x_i to x_j at time $t+1$ [8]. When the variable s^{t+1} depends on only the variable s^t, then the transition function (Eq. (3.2)) satisfies the independence criterion as follows:

$$\begin{aligned}\Pr(s^{t+1} \mid s^t) &= \Pr(x_1^{t+1} \mid s^t) \Pr(x_2^{t+1} \mid s^t) \dots \Pr(x_n^{t+1} \mid s^t) \\ &= \prod_i \Pr(x_i^{t+1} \mid s^t)\end{aligned} \tag{3.3}$$

In a DBN without synchronic arcs, this independence assumption is valid but such model may not be realistic. Thus, in our research, we focus on structure learning with synchronic arcs [8, 23]. We suppose that correlation between state features can exist at time $t+1$ as an effect of taking action a at time t. When the DBNs have synchronic arcs, each factor is dependent on its parents in the previous time step as well as other factors in the same time step as seen in Eq. (3.2). To simplify the notation, we use $\mathit{Parents}(x_i)$ to denote the parent set of variable x_i. $\mathit{Parents}(x_i)$ consists of $\mathit{Parents}^t(x_i^{t+1})$, parents set at time t and $\mathit{Parents}^{t+1}(x_i^{t+1})$, parents set at time $t+1$. We can rewrite the transition function with $\mathit{Parents}(x_i)$ as follows:

$$\Pr(s^{t+1} \mid s^t) = \prod_i \Pr(x_i^{t+1} \mid \mathit{Parents}(x_i^{t+1})) \tag{3.4}$$

3.3.2 Related work

A factored MDP was first proposed by Boutilier et al.[7] and the transition and reward functions are represented with Dynamic Bayesian Networks. In [8], the authors use DBNs with decision trees representing transition functions and rewards. Based on this representation, structured value iteration (SVI) and structured policy iteration (SPI) algorithms are proposed. They also consider synchronic constraints in a problem and extend their algorithm to deal with the synchronic constraints.

DBN-E^3 [58] and Factored-Rmax [48] assume the DBN is known in advance and learn the transition probabilities from the structure of the DBN. In [92, 28, 11, 24, 53], the DBN structure is not given. The agent learns the structure of the DBN and then learns the transition probabilities from the DBN. In several methods [92, 28],

the maximum in-degree (the maximum number of parents of any factor) of the DBNs is given as a prior knowledge.

SLF-Rmax [92] learns the conditional probabilities of DBNs when given the maximum in-degree of the DBNs. The algorithm enumerates all possible combinations of factors as elements and keeps statistics for all pairs of the elements.

Diuk et al. [28] propose Met-Rmax which improves SLF-Rmax's sample complexity. Met-Rmax [28] is based on k-Meteorologists Problems. For n binary factors and maximum in-degree D, all $\binom{n}{k}$ subsets of factors are considered as possible parents. Each parent set corresponds to a hypothesis sub-class in Adaptive k-Meteorologists and it predicts the outcome. The squared prediction error of meteorologist is used to improve the efficiency of the algorithm.

Chakraborty et al. [11] present a similar approach called LSE-RMax but the algorithm does not require knowledge of the in-degree of the DBNs. Instead, LSE-RMax uses a planning horizon that satisfies a certain conditions.

Another approach is to use decision trees for building structured representations of the problem.

Degris et al. [24] use the decision tree induction algorithms called ITI [96] to learn the reward function and the DBN of the transition function. The generalization property of the decision trees improves the policy faster than tabular representations.

RL-DT [53] improves Degris et al.'s algorithm with the relative effects of transitions and a different exploration policy. The algorithm uses decision trees to generalize the relative effects of actions across similar states in the domain. As exploration policy, the agent first explores the environment to learn an accurate model. When it takes the actions it believes to be optimal, it switches into exploitation mode.

3.3.3 Algorithm for Structure Learning

We present an algorithm to learn the structure of the DBN transition functions with synchronic arcs, shown in Algorithm 12. We use decision trees to represent transition functions instead of tabular representations.

Similar to R-max [9], all unknown state-action values are initialized by a constant $Rmax$ in order to encourage the agent to explore. Each time, the agent takes a greedy action.

To build decision trees, actions have to be visited sufficiently often. We set a predefined parameter m, the minimum number of visits required to unknown actions, to decide if actions are known or not. Whenever an action is taken, the visit count of the action is incremented (Line 9). If the number of visit for an action is equal to m, decision trees for the action are created (Line 13).

Generally, given an action, each factor $s'(i)$ has its own decision tree to estimate $\Pr(s'(i) \mid \cdot, a)$, i.e., one decision tree represents $\Pr(s'(i) \mid \cdot, a)$. We reduce the number of decision trees by choosing some relevant factors whose values are constantly changed whenever the action is taken. Transition functions of non-changed factors are identity functions. Since decision trees of those non-changed factors do not affect the estimation of transition from state s to s', we do not create their decision trees. We collect all value-changed factors in F_a whenever action a is selected (Line 10). Then, for each factor $s'(i)$ in F_a, we create a decision tree of the DBNs.

LearnTransitionFunction estimates $\Pr(s'(i) \mid s, a)$ from the corresponding decision tree and updates the tree with s and $s'(i)$. The action value is computed with the obtained transition functions (Line 23). If there is any factor that is still not predictable, we update action value with *Rmax* (Line 21) to make learn more the state-action value.

Algorithm 12 Learning the structure of the DBN with synchronic arcs

1: Input: initial action value *Rmax*, minimum visit count on action m
2: // Initialization
3: $\forall a \in A, \forall s \in S, \quad Q(s, a) \leftarrow Rmax$
4: **repeat**
5:     **repeat**
6:         $a \leftarrow argmax_{a' \in A} Q(s, a')$
7:         Execute a, obtain reward r and observe next state s'
8:         **if** $C(a) < m$ **then**
9:             $C(a) \leftarrow C(a) + 1$
10:             $F_a \leftarrow RecordChangedFactors(a)$
11:         **else**
12:             **if** $C(a) == m$ **then**
13:                 $BuildDecisionTrees(a)$
14:             **end if**
15:             // Estimate transition function
16:             **for** each factor $s'(i)$ **do**
17:                 $\Pr(s'(i) \mid s, a) \leftarrow LearnTransitionFunction(s, a, s'(i))$
18:             **end for**
19:             // Update action-values
20:             **if** $\exists i, \Pr_i(s'(i) \mid s, a) = \perp$ **then**
21:                 $Q(s, a) \leftarrow Rmax$
22:             **else**
23:                 $Q(s, a) \leftarrow R(s, a) + \gamma \sum_{s'} \Pr(s' \mid s, a) max_{a' \in A} Q(s', a')$
24:             **end if**
25:         **end if**
26:         $s \leftarrow s'$
27:     **until** reaching the terminal state
28: **until** algorithm converges

BuildDecisionTrees is shown in Algorithm 13. To build decision trees, we first select parents factors that will be used as nodes of the decision tree (Line 3). For each factor $s'(i)$ of F_a, $FindParents$ applies χ^2 test to all other factors of time t and to all other factors of F_a at time $t+1$ to find its parents, $Parent(s'(i))$, shown in Eq. (3.4). *par* is its parents at time t and par_{sync} is its parents at time $t+1$. However, it is difficult to conclude that par_{sync} is a parents set of factor $s'(i)$ because χ^2 test determines whether there is a significant relation between two variables but it does not determines which one causes the other. To decide which one is a parent, we predefine the order of features. Suppose there are two features x_i and x_j and they are related to each other at time $t+1$ by χ^2 test. If x_i precedes x_j in order, then we consider x_j a synchronic parent of x_i. For feature x_i, we

place its potential parent features after x_i. All following features are candidates for synchronic parents. Using this verification, $FindRealSyncParents(i)$ determines which factors are real synchronic parents and returns par_{sync}. *CreateDecisionTree* builds a decision tree whose nodes are elements of *par* and par_{sync}. In our algorithm, we use HoeffdingTree [30, 55] that is an incremental decision tree induction algorithm that is capable of learning from massive data streams[1].

Algorithm 13 BuildDecisionTrees

```
1: Input: action a
2: for each factor s'(i) of F_a do
3:     (par, par_sync) ← FindParents(i)
4:     par_sync ← FindRealSyncParents(i)
5:     CreateDecisionTree(par, par_sync)
6: end for
```

3.3.4 Experiments

We evaluate our algorithm with respect to three different algorithms: Q-learning [98], R-max [9], and LSE-RMax [11]. Q-learning is a model-free algorithm and R-max is a model-based algorithm. LSE-RMax is a factored model-based algorithm with tabular representations. We apply our algorithm to the coffee delivery task [8, 23].

Coffee Delivery Task A robot goes to a coffee shop to buy coffee and delivers the coffee to its owner in his office. It may rain on the way to the coffee shop or the office. The robot will get wet if it does not have an umbrella.

The state is described by six Boolean variables:

- **HRC:** the robot has coffee
- **HOC:** the robot's owner has coffee
- **W:** the robot is wet
- **R:** it is raining
- **U:** the robot has an umbrella
- **O:** the robot is at the office

As discussed in Algorithm 13, we have to order state features to decide which features are synchronic parents. The feature order used in our experiments is the same order as the list above. A synchronic arc exists between HRC and HOC when selecting DelC action. After the delivering action, the robot loses the coffee depends on whether the owner successfully gets the coffee or not. The probability setting about value changes of HRC and HOC features is explained below.

The robot has four actions:

[1] http://weka.sourceforge.net/doc.dev/weka/classifiers/trees/HoeffdingTree.html

- **Go:** Moves to office or coffee shop with success probability 0.9.
- **BuyC:** Buy coffee if it is in the coffee shop with success probability 0.9.
- **DelC:** Deliver coffee to its owner if it is in the office
- **GetU:** Get an umbrella if it is in the office with success probability 0.9.

All actions can be noisy, i.e., tasks are non-deterministic. In our experiments, we set the success probabilities of Go, BuyC, and GetU actions to 0.9. It rains on the way to the coffee shop or the office with probability 0.3. For DelC action, we set the probability of value changes for HRC and HOC features as follows: if the owner receives the coffee, the robot loses it. This happens with probability 0.8. Otherwise, the owner fails to receive the coffee. Then, the robot keeps holding it with probability 0.8 or loses it with probability 0.2. The robot gets a reward of 0.8 if its owner has coffee and an additional reward of 0.2 if it is dry. The start state is that the robot is at the office without a coffee. When the robot delivers a coffee to its owner, an episode finishes.

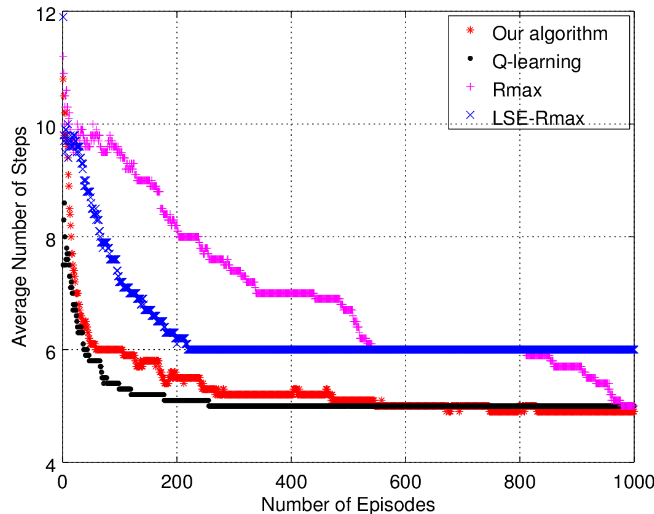


Figure 3.8 – Average Step Counts

Experimental Results Figure 3.8 shows the average step counts for different algorithms. The average steps are calculated at every 100 episode by dividing the total steps from the first episode to the last by the total number of episodes. The optimal policy is "(GetU→)Go→BuyC→Go→DelC". The goal has to be achieved in four or five steps in the best case. Q-learning converges to the optimal policy faster than the others. Our algorithm is also relatively fast. R-max converges late in comparison to Q-learning and our algorithm. R-max has to learn transition functions for each state-action pair, so it explores more to visit every state-action pair. Unlike the others, LSE-Rmax does not learn effectively the optimal policy.

The average rewards for different algorithms are shown in Figure 3.9. For each episode, the agent gets either 0.8 or 1.0 reward. If the agent gets wet, -0.2 penalty

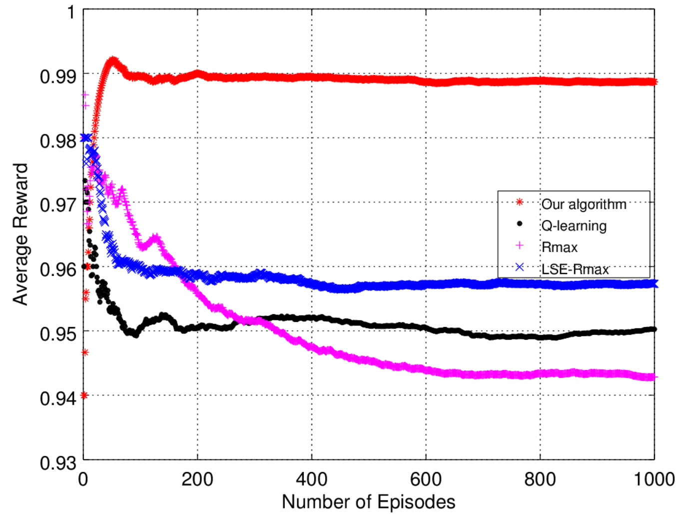


Figure 3.9 – Average Reward

is given at the terminal state, i.e., the agent gets 0.8 as total reward. When the agent notices this penalty, it will learn an alternative way to avoid the penalty. Our algorithm learns well this penalty case. This is because it learns more accurately the transition models by factoring states. LSE-Rmax is lower than our algorithm but it is slightly better than Q-learning and R-max. By this experiment, we can see that factorization methods learn effectively correct models that lead to increase rewards.

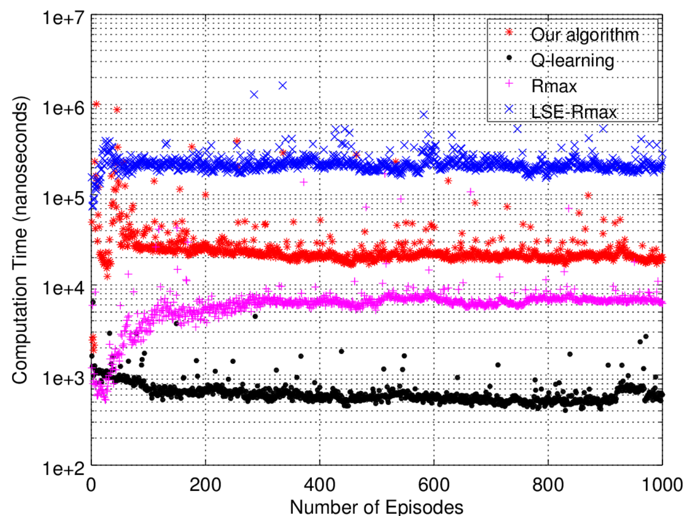


Figure 3.10 – Average Computation Time

Figure 3.10 shows the average computation time for each episode. Computation time is measured on action-value update of each algorithm. Q-learning and R-max are faster than LSE-Rmax and our algorithm that use factorized state spaces. In non-factored models, updating a value is quite simple because they just find and update the corresponding state-action pair from a tabular, however, factored models

have to learn a transition function for each factor. It means the computation cost to learn transition functions by factoring states is greater than that of non-factored methods.

3.4 Discussion and Future Research

In the taxi application, we selected two off-peak hours (12h to 13h and 14h to 15h) and we investigated the learning behavior during the fixed time periods. In a real problem, passenger behavior changes over time. For example, locations of passengers change suddenly because of unexpected events. In the taxi problem, goals, transition and reward probabilities of the environment can change over time. Effectively, its environment is highly non-stationary (see Section 2.8).

In this chapter, we applied Q-learning algorithm, a model-free method, to the taxi application. Since reinforcement Learning is basically applicable to non-stationary environment, the learning agent can adapt continually to dynamics changes. However, we cannot expect a more responsive adaptation to the changes and when the environment reverts to the previously learned dynamics, the learned knowledge in the past becomes useless.

The Q-learning algorithm is fast but it does not consider a model of the environment. Since the taxi problem is non-stationary, it may be better to apply a model-based method and make it detect the environment changes. In addition to model-based methods' advantages such as sample-efficiency and prediction about the next state and next reward, we can expect to solve the non-stationarity of the taxi problem.

Some previous works have addressed non-stationary problems. Choi et al. [18] introduce hidden-mode Markov decision process (HM-MDP) and assume that the environment is in exactly one of the modes at any given time. In RL-CD [19], the environment is divided into partial models estimated by observing the transitions and rewards. For each time step, the model with the highest quality is activated. The algorithm starts with only one model and then incrementally creates new ones as they become necessary. In TES [79], the agent learns world models called views and collects them into a library to reuse in future tasks. When the agent encounters a new task in a new environment, it selects a proper view from the library or adapts to the new task with completely new transition dynamics and feature distributions. BPR+ [52] is in a multi-agent setting to deal with non-stationary opponents. The learning agent detects that the current policies do not perform optimally and then learns incorporating models of new strategies.

In the taxi problem, we can differentiate the passenger behaviors with various time periods such as morning-rush hour, evening-rush hour, off-peak hours and holidays. Then, the agent can learn models depending on the different time periods. However, even in the same period the movement may be varied dynamically. We will need a more flexible method to adapt to environment dynamics. Like those existing methods [19, 79, 89, 52], the environment dynamics of the taxi problem can be divided into partial models that are stored in a library. For each time, the agent use a partial model that predicts well the environment. If the prediction error of the current model is larger than a threshold, the agent selects another model from

the library. If the environment dynamics is completely different from the existing models, it creates a new model. This will be more flexibly adaptable to a non-stationary environment than selecting pre-defined modes by a system designer. In addition, the taxi application we have discussed so far is based on a single agent but it has to be extended to a multi-agent setting. In a multi-agent environment, it will be important for the agent to have an ability to detect non-stationary opponents and learn optimal policies against changed opponent strategies. When opponent strategies are not known a priori, the agent has to adapt to the new environment. Instead of fixed models, the flexible models proposed above will be able to deal with such non-stationary problems.

3.5 Conclusion

In this chapter, we discussed two learning approaches – learning without models and learning models to estimate action values.

One of the well-known model-free methods is Q-learning. We apply Q-learning algorithm to a real taxi routing problem. We also investigate the influence of the step size, discount rate and trade-off between exploration/exploitation on learning. To improve action selection strategy, we proposed a customized exploration and exploitation strategy for the taxi problem.

In model-based methods, we address the factored MDP problem in a non-deterministic setting. Most model-based methods are based on DBN transition models. We proposed an algorithm that learns the DBN structure including synchronic arcs. Decision trees are used to represent transition functions. In experiments, we show the efficiency of our algorithm by comparing with other algorithms. We also demonstrate that factorization methods allow to learn effectively complete and correct models to obtain the optimal policies and through the learned models the agent can accrue more cumulative rewards.

Chapter 4

Focused Crawling

In this chapter, we extend our discussion to a very large and continuous domain, in particular, a focused crawling problem.

4.1 Introduction

Focused crawlers are autonomous agents designed to collect Web pages relevant to a predefined topic, for example to build a search engine index. Given a start page (the *seed*), a crawler browses Web pages by exploiting hyperlinks of visited Web pages to find relevant pages. Usually, crawlers maintain a priority queue of new URLs, called the *frontier*. Each new URL is assigned a priority value and URLs are fetched from the queue in decreasing order of priority. Since the focused crawler aims to collect as many relevant pages as possible while avoiding irrelevant pages, the key success factor of crawling systems is how good the scoring policy is.

The priority score is initially based on contextual similarity to the target topic [10, 27], on link analysis measures such as PageRank and HITS [61, 80, 2], or on a combination of both [3, 12]. However, links that look less relevant to the target topic but that can potentially lead to a relevant page in the long run may still be valuable to select. *Reinforcement learning* (RL) enables the agent to estimate which hyperlink is the most profitable over the long run. A few previous studies have applied reinforcement learning to crawling [86, 44, 74, 75], but they require an off-line training phase and their state definitions do not consider the link structure; for example, states are represented with a vector which consists of the existence or frequency of specific keywords.

Hence, we propose a reinforcement learning based crawling method that learns link scores in an online manner, with new representations of states and actions considering both content information and the link structure. Our method assumes that the whole Web graph structure is not known in advance. To properly model the crawling environment as a Markov decision process (MDP), instead of considering each individual page as a state and each individual hyperlink as an action, we generalize pages and links based on some features that represent Web pages and the next link selection, thus reducing the size of the state–action space. To allow efficient computation of a link score, i.e. *action-value*, we approximate it by a linear combination of the feature vector and a weight vector. Through this modeling, we

can estimate an action value for each new link, to add it to the frontier. As action values computed at different time steps are used in the frontier, we investigate a synchronous method that recalculates scores for all links in the frontier, along with an asynchronous one that only compute those of all outlinks of the current page. As an improved asynchronous method, we propose moderated update to reach a balance between action-values updated at different time steps. In experiments, we compare our proposed crawling algorithms based on reinforcement learning and an algorithm without learning.

This chapter is organized as follows. Section 4.2 presents some important background. Section 4.3 introduces our algorithm, focused crawling with reinforcement learning. Section 4.4 describes the details of our experiments and shows the performance evaluation of our algorithm. Section 4.5 presents prior work in the literature. Section 4.6 discuss some possible extensions. Section 4.7 concludes with some further work.

4.2 Background

Web Crawling. A Web crawler is an agent which autonomously browses Web pages and collects all visited pages. The fetched pages may be stored and indexed in a repository. Web crawlers are in particular used to index Web pages by search engines in order to provide users with fast search. Starting with a set of seed URLs, the crawler visits these URLs, retrieves all hyperlinks from the pages and adds them to the queue of unvisited URLs, called the frontier. The URLs in the queue are visited according to some priority policy. The process repeats until a certain number of pages are collected or some other objective is achieved. The priority of URLs in the queue depends on which crawling strategy is used. In a breadth-first crawl, the frontier can be implemented as a first-in-first-out (FIFO) queue. The best-first crawler assigns a priority to each unvisited URL based on an estimate value of the linked page and the frontier is implemented as a priority queue. Most crawling algorithms in the literature are variations of best-first [73].

Focused Crawler, Topical Locality, and Tunneling. A focused crawler selects from the frontier the links that are likely to be most relevant to a specific topic(s). It aims to retrieve as many relevant pages as possible and avoid irrelevant pages. This process consequently brings considerable savings in network and computational resources. While general-purpose crawlers may follow breadth-first search, focused crawlers are best-first search with their own priority strategies.

Focused crawlers are based on topical locality [22, 72]. That is, pages are likely to link to topically related pages. Web page authors usually create hyperlinks in order to help users navigate, or to provide further information about the content of the current page. If hyperlinks are used for the latter purpose, the linked pages may be on the same topic as the current page and hyperlinks can be useful information for topic-driven crawling. Davison [22] shows empirical evidence of topical locality on the Web. He demonstrates that linked pages are likely to have high textual similarity. Menczer [72] extends the study and formalizes two general conjectures, link–content conjecture and link–cluster conjecture, representing connections from

the Web's link topology to its lexical and semantic content. The measurement results of these conjectures confirm the existence of link–content conjecture that a page is similar to the pages that link to it and that of link–cluster conjecture that two pages are considerably more likely to be related if they are within a few links from each other. The author shows that the relevance probability is maintained within a distance of three links from a relevant page, but then decays rapidly.

To selectively retrieve pages relevant to a particular topic, focused crawlers have to predict whether an extracted URL points to a relevant page before actually fetching the page. Anchor text and surrounding text of the links are exploited to evaluate links. Davison [22] shows that titles, descriptions, and anchor text represent the target page and that anchor text is most similar to the page to which it points. Anchor text may be useful in discriminating unvisited child pages.

Although a focused crawler depends on the topical locality, pages on the same topic may not be linked directly and it can be necessary to traverse some off-topic pages to reach a relevant page, called tunneling [6]. When going through off-topic pages, it is needed to decide if the crawl direction is good or not. Bergmark et al. [6] propose a tunneling technique that evaluates the current crawl direction and decides when to stop a tunneling activity. They show tunneling improves the effectiveness of focused crawling and the crawler should be allowed to follow a series of bad pages in order to get to a good one. Ester et al. [32] also propose a tunneling strategy that reacts to changing precision. If precision decreases dramatically, the focus of the crawl is broaden. Conversely, if precision increases, the focus goes back to the original user interest.

4.3 Focused Crawling and Reinforcement Learning

The goal of focused crawling is to collect as many pages relevant to the target topic as possible while avoiding irrelevant pages because the crawler is assumed to have limited resources such as network traffic or crawling time. Thus, in a sequence of crawling, link selection should not be a random choice.

To achieve the crawling goal, given a page, the agent selects the most promising link likely to lead to a relevant page. Even though a linked page looks less relevant to the target topic, if it can potentially lead to a relevant page in the long run, it might be valuable to select it. At each time step, the agent has to estimate which hyperlink can lead to a relevant page. It will be a key success factor in crawling if the agent has an ability to estimate which hyperlink is the most profitable over the long run.

Reinforcement learning finds an optimal action in a given state that yields the highest total reward in the long run by the repeatedly interaction with the environment. With reinforcement learning, the optimal estimated value of hyperlinks (actions) are learned as pages (states) are visited. The agent can evaluate if a link selection can yield a long-term optimal reward and selects the most promising link based on the estimation. In this section, we discuss how to model a focused crawling with Reinforcement Learning. Like most focused crawlers, we assume that pages with similar topics are close to each other. Our crawling strategy is based on the

topical locality and tunneling technique. We also assume that the whole Web graph structure is not known to the crawling agent in advance.

4.3.1 Markov Decision Processes (MDPs) in Crawling

To model the crawling environment in an MDP $M = \langle S, A, R, T \rangle$, we define Web pages as states S and direct hyperlinks of a page as actions A. When the crawling agent follows a hyperlink from the current page, a transition from the current page to the linked page occurs and a relevance to the target topic is computed for the linked page to evaluate if the selected hyperlink leads to a page relevant to the target topic or not. The transition function T is the probability of transition from the current page to the linked page on taking the hyperlink. The reward $r \in R$ is a relevance value of the linked page to the given topic. For the next crawling step, the agent selects a hyperlink with the highest estimation value from the newly visited page, and so on.

Before applying the model above to solve our crawling problem, we must consider two issues: first, scalability of state-action space in a reinforcement learning, second, applicability to a crawling task without loss of its inherent process property. For the scalability problem, we reduce a state-action space by generalization presented in this section and update value functions with linear function approximation discussed in Section 4.3.3. For the applicability issue, in order to preserve the original crawling process we prioritize updates, see Section 4.3.2 and 4.3.3.

In this section, we discuss modeling a crawling task as an MDP. As we mentioned above, we define Web pages as states and direct hyperlinks of a page as actions. However, Web pages are all different, there are a huge amount of pages on the Web, and they are linked together like the threads of a spider's Web. If each single Web page is defined as a state and each direct hyperlink as an action, it makes learning a policy intractable due to the immense number of state-action pairs. Furthermore, in reinforcement learning, optimal action-values are derived after visiting each state-action pair infinitely often. It is not necessary for a crawler to visit the same page several times. Thus, our MDPs can not be modeled directly from a Web graph. Instead, we generalize pages and links based on some features that represent Web pages and the next link selection. By this generalization, the number of state-action pairs is reduced and Web graph is properly modeled in an MDP. Some pages with the same feature values are in the same state. Some hyperlinks with the same feature values also can be treated as the same action. The features extracted from pages and hyperlinks are presented in the following.

States. A Web page is abstracted with some features of Web pages in order to define a state. The features of a state consists of two types of information. The first one is proper information about the page itself. The second is relation information with respect to surrounding pages. Page relevances to the target topic and to some categories are the current pages' own information. Relevance change, average relevance of parent pages, distance from the last relevant page represent the relation with the pages surrounding the current page. In order to properly obtain the relation information, each unvisited link should retain parent links. The crawling agent is assumed not to know the whole Web graph in advance, thus each link initially

does not know how many parents they have but parent information is progressively updated as pages are crawled. When a page is visited, the URL of the current page is added to all outlinks of the page as their parent. Each link has at least one parent link. If a link has many parents, it means that the link is referenced by several pages.

Most features are continuous variables, which we specified with two different indexes discretized into 5 and 6 buckets according to value ranges: 1) the range $[0.0, 0.2]$ by 0, $[0.2, 0.4]$ by 1, ..., $[0.8, 1.0]$ by 4. 2) the range $[0.0, 0.1]$ by 0, $[0.1, 0.3]$ by 1, ..., $[0.9, 1.0]$ by 5. The relevance value is also discretized according to value ranges as above but occasionally the value has to be converted to a Boolean to specify if a page is relevant to a given topic or not. For example, two features, the Average Relevance of Relevant Parents and Distance from the Last Relevant Page, require true/false value regarding to the relevance. To avoid an arbitrary threshold for relevance, we simplify the definition of relevant page as follows: if a crawled page has a tf-idf based relevance greater than 0.0 or simply contains the target topic word, the page is defined to be relevant to the topic.

- **Relevance of Target Topic:** The target topic relevance based on textual content is computed by cosine similarity between a word vector of target topic and that of the current page and it is discretized according to value ranges.

- **Relevance Change of Target Topic:** The current page's relevance to the target topic is compared to the weighted average relevance of all its ancestors on the crawled graph structure. The weighted average relevance of all its ancestors are computed in an incremental manner by applying an exponential smoothing method on the parents pages.

  Before we explain how to calculate the weighted average relevance of its all ancestors, we simply note that in the exponential smoothing method, the weighted average y at time i with an observation x_i is calculated by: $y_i = \beta \cdot x_i + (1-\beta) \cdot y_{i-1}$, where $\beta (0 < \beta < 1)$ is a smoothing factor. The exponential smoothing assigns exponentially decreasing weights on past observations. In other words, recent observations are given relatively more weight than the older observations.

  In our crawling example, if the relevance of a page x is denoted $rl(x)$, then the weighted average relevance of x, $w_{rl}(x)$ is obtained by $w_{rl}(x) = \beta \cdot rl(x) + (1-\beta) \cdot \max_{x' \to x} w_{rl}(x')$.

  If the current page has many parents, i.e. many path from its ancestors, the maximum average among them, $\max_{x' \to x} w_{rl}(x')$, is used for the update. $w_{rl}(x)$ is the weighted average relevance over x and all its ancestors on the crawled graph structure.

  Then, we can calculate the relevance change between current page p and $w_{rl}(x)$ where x is a parent of p: $change \leftarrow rl(p) - \max_{x \to p} w_{rl}(x)$.

  The change helps to detect how much the relevance of the current page is increased or decreased than the average relevance of its ancestors.

  The relevance change to the current page from its ancestors is discretized according to value ranges. With predefined parameters δ_1 and δ_2, the difference

within δ_1 is indexed by 0, the increase by δ_2 is indexed by 1, increase more than δ_2 is indexed by 2, decrease by δ_2 is indexed by 3, and decrease more than δ_2 is indexed by 4.

- **Relevances of Categories:** Given a target topic, its related categories in a category hierarchy such as the Open Directory Project (ODP, `https://www.dmoz.org/`, `http://curlie.org/`) are properly selected by the designer of the system. For each category, its relevance is calculated by cosine similarity between a word vector of the category and that of the current page. It is discretized according to value ranges.

- **Average Relevance of All Parents:** The average of all parents' relevance is calculated and discretized according to value ranges.

- **Average Relevance of Relevant Parents:** The average of relevant parents' relevance is calculated and discretized according to value ranges.

- **Distance from the Last Relevant Page:** The distance on the crawl path from the last relevant ancestor page to the current page is simply calculated by adding 1 to the parent's distance. If there are many parents, the minimum distance among them is used. The distance value is capped at 9 to keep it within a finite range.

$$distance = \begin{cases} 0 & \text{if it is a relevant page} \\ 1 + \text{parent's distance} & \text{otherwise} \end{cases} \tag{4.1}$$

Actions. In order to define actions, all hyperlinks in a Web page are also abstracted with some features in a similar way as pages are. Relevances to the target topic and to some categories are used to predict the relevance of the page that a hyperlink points to. Different from pages, hyperlinks do not have sufficient information to calculate the values. Thus, the URL text, the anchor text and surrounding text of a hyperlink are used to compute. Here, the relevance is not a true relevance but a prediction because it is not possible to know which page will be pointed by a hyperlink before following the link. In order to support the relevance prediction, the average relevances of parent pages are also used as features that represent the relation with the pages surrounding the link. Each hyperlink has at least one parent. If the link is referenced by several pages, it can have many parents. As mentioned above, parent information is progressively updated as pages are crawled and each unvisited link retains parent links. Then, the parent information is used to compute average relevance of parent pages. The features for action are a subset of those of states, namely:

- **Relevance of Target Topic**

- **Relevances of Categories**

- **Average Relevance of All Parents**

- **Average Relevance of Relevant Parents**

The size of a discretized state space is $(10)^4 \cdot (10)^{\text{num of categories}} \cdot 5$ and the size of action space is $(10)^3 \cdot (10)^{\text{num of categories}}$. For example, if there is just one category, the size of the state space is $5 \cdot 10^5$ and the size of the action space is 10^4.

4.3.2 MDPs with Prioritizing Updates

In a focused crawl, the agent visits a Web page and extracts all hyperlinks from the page. The hyperlinks are added to the priority queue, called frontier. A link with the highest priority is selected from the frontier for the next visit. The frontier plays a crucial role in the crawling process. The agent can take the broad view of the crawled graph's boundary, not focusing on a specific area of the whole crawled graph. Unvisited URLs are maintained in the frontier with priority score and therefore, for each iteration, the most promising link can be selected from the boundary of the crawled graph. Thus, the Web crawler can consistently select the best link regardless of its current position.

We use a temporal difference (TD) method of reinforcement learning in order to make crawling agents learn good policies in an online, incremental manner as crawling agents do (see Section 2.3). In most TD methods, each iteration of value updates is based on an episode, a sequence of state transitions from a start state to the terminal state. For example, at time t, in state s the agent takes an action a according to its policy, which results in a transition to state s'. At time $t+1$ in the successor state of s, state s', the agent takes its best action a' followed by a transition to state s'' and so on until the terminal state. While crawling, if the agent keeps going forward by following the successive state transition, it can fall into crawling traps or local optima. That is the reason why a frontier is used importantly in crawling. It is necessary to learn value functions in the same way as crawling tasks.

To keep the principle idea of crawling tasks, we model our crawling agent's learning with prioritizing the order of updates that is one of value iteration methods to propagate the values in an efficient way (see Section 2.7). With a prioritized update method, the crawling agent does not follow anymore the successive order of state transitions. Each state-action pair is added to the frontier with its estimated action value. For each time, it selects the most promising state-action pair among all pairs as the traditional crawling agent does.

4.3.3 Linear Function Approximation with Prioritizing Updates

We have modeled our crawling problem as an MDP and defined features of the states and the actions in Section 4.3.1. Then, we have presented prioritized updates in reinforcement learning to follow the original crawling process in Section 4.3.2. In this section, we discuss how to represent and update action-value functions based on the state and action features defined in Section 4.3.1.

As discussed in Section 4.3.2, the crawling frontier is a priority queue. Each URL in the frontier is associated with a priority value. The links are then fetched from the queue in order of assigned priorities. In our crawling model, we estimate an action value for each unvisited link and add it to the frontier with its action value.

In reinforcement learning, if a state space is small and discrete, the action value functions are represented and stored in a tabular form. But, this method is not suitable for our crawling problem with a large state-action space. Thus, we use a function approximation method, in particular the linear function approximation, to represent action values (see Section 2.4). The action value function is approximated by linearly combining the feature vector $\mathbf{x}(s, a)$ and the weight vector $\mathbf{w}$ with Eq. (2.26). State and action features defined in Section 4.3.1 are used as the components of a feature vector $\mathbf{x}(s, a)$. At each time step, the weight vector $\mathbf{w}$ is updated using a gradient descent method, as in Eq. (2.27). The approximated action-value obtained from Eq. (2.26) is used as the priority measure.

When we calculate action-values only for the outlinks of the current page with newly updated weights and add them to the frontier, an issue can arise in the scope of state-action pairs regarding computation of action value. This problem is caused from the prioritized order of selecting a link from the frontier. If the agent keeps going forward by following the successive state transition, it is correct that calculating action values is applied only to the direct outlinks because the next selection is decided from one of the all outlinks. However, in the prioritized order selecting from the frontier, when the weight vector $\mathbf{w}$ is changed, action values of all links in the frontier also have to be recalculated with the new $\mathbf{w}$. We call this *synchronous* method. Recalculating for all links is the correct method but it involves an excessive computational overhead. Otherwise, we can calculate action-value only for outlinks of the current page and/or recalculate all links(actions) in the frontier that are from the current state. The action values of all other links in the frontier are left unchanged. We call this *asynchronous* method. This method does not incur computational overhead but action values of all links in the frontier are calculated at different time steps and make it difficult to choose the best action from the frontier. In experiments, we compare the performance of the two methods.

Since the asynchronous method has an advantage that does not need to recalculate action values of all unvisited links in the frontier, we try to improve the asynchronous method. The problem of asynchronous method is that action values computed in different time steps exist together in the frontier and it can cause a noise in selection. Thus, we reduce the action value differences in the frontier by manipulating weight updates. The TD error is the difference between the estimates at two successive time steps, $r + \gamma\hat{q}(s', a', \mathbf{w})$ and $\hat{q}(s, a, \mathbf{w})$. Updating the error to weights signifies that the current estimate $\hat{q}(s, a, \mathbf{w})$ is adjusted toward the update target $r + \gamma\hat{q}(s', a', \mathbf{w})$. In order to moderate the TD error, we adjust the estimate $\hat{q}(s', a', \mathbf{w})$ by the amount of the TD error when updating weights as follows:

$$\mathbf{w} \leftarrow \mathbf{w} + \alpha \left[r + \gamma(\hat{q}(s', a', \mathbf{w}) - \delta) - \hat{q}(s, a, \mathbf{w})\right] \nabla\hat{q}(s, a, \mathbf{w}) \tag{4.2}$$

where $\delta = r + \gamma\hat{q}(s', a', \mathbf{w}) - \hat{q}(s, a, \mathbf{w})$. We call this *moderated* update. In fact, this moderated update can be shown to have same effect as reducing the step-size α of the original update by $1 - \gamma$.

$$\begin{aligned}
\mathbf{w} &= \mathbf{w} + \alpha \left[r + \gamma(\hat{q}(s', a', \mathbf{w}) - \delta) - \hat{q}(s, a, \mathbf{w})\right] \nabla\hat{q}(s, a, \mathbf{w}) \\
&= \mathbf{w} + \alpha\delta\nabla\hat{q}(s, a, \mathbf{w}) - \alpha\gamma\delta\nabla\hat{q}(s, a, \mathbf{w}) \\
&= \mathbf{w} + \alpha(1 - \gamma)\delta\nabla\hat{q}(s, a, \mathbf{w})
\end{aligned}$$

The idea behind the moderated update is to decrease an overestimated action value or to increase an underestimated action value of the update target in order to make a balance between action-values updated at different time steps.

In experiments, we compare the performance of synchronous, asynchronous methods and asynchronous method with moderated update.

Our reinforcement learning for crawling is outlined in Algorithm 14. The crawling task is started with seed pages (lines 5–13). The frontier is filled with (s, a) pairs of all outlinks from the seed pages. A link is extracted from the frontier according to the ϵ-greedy policy (lines 16–20). With small probability ϵ, the agent selects a link uniformly at random. Otherwise, it selects greedily a link from the frontier. The agent fetches the page corresponding to the selected link and defines feature values of the newly visited state as described in Section 4.3.1 (line 24). All outlinks in the fetched page are retrieved (line 25). For each outlink, action feature values are defined as described in Section 4.3.1 (line 30). The weight vector $\mathbf{w}$ of linear function approximation is updated based on a reward and feature vectors of the new state returned from the fetch in line 24 (lines 32–39). With the updated weight vector, an estimated action value of each outlink is computed and added to the frontier with the estimated value. If we use the synchronous method, action values of all hyperlinks in the frontier $(l, \cdot, \cdot)$ are recalculated (lines 40–43). With the asynchronous method, hyperlinks $(l', s', \cdot)$ that are from the state s' are updated with new action values (lines 44–47). This process repeats until the visit counter reaches the predefined visit limit.

4.4 Experimental Results

A crawling task starts with a seed page and terminates when the visit counter reaches the predefined visit limit. In our experiments, the limit of the page visit is set to 10,000. For each time step, the agent crawls a page and obtains a reward based on two values: cosine similarity based on tf-idf, and cosine similarity with word2vec vectors (w2v) pre-trained from `https://nlp.stanford.edu/projects/glove/`. If a crawled page has a tf-idf based relevance greater than 0.0 or simply contains the target topic word, the page is relevant to the target topic and then the agent receives a reward of 30. If a page has a tf-idf based relevance lower than 0.0 but it has a w2v based relevance greater than 0.5 or 0.4, the agent receives a reward of 30 or 20 respectively because the content of such page is rather related to the target topic and could eventually lead to a relevant page. Otherwise, the agent receives a reward -1 per time step.

As a crawling environment, we use a database dump of Simple English Wikipedia provided by the site `https://dumps.wikimedia.org/`. As target topics to use in our experiments, we choose three topics, Fiction, Olympics, and Cancer, of which relevant pages are fairly abundant and another three topics, Cameras, Geology, and Poetry, of which relevant pages are sparse in our experimental environment. For each target topic, a main page corresponding to the topic is used as a seed page. In all experiments, parameter settings for learning are $\epsilon = 0.1$, discount rate $\gamma = 0.9$, and step size $\alpha = 0.001$. For topic Olympics and Fiction, step size α is set to 0.0005.

Algorithm 14 Focused Crawling based on Reinforcement Learning

1: Input: seed links *Seeds*, maximum number of pages to visit *LIMIT_PAGES*
2: Initialize value-function weights $w \in \mathbb{R}^d$
3: $B \leftarrow \emptyset$ # contains (s, a) pairs
4:
5: **while** *Seeds* is not empty **do**
6: Select a link l from *Seeds*
7: $s \leftarrow$ Fetch and parse page l
8: $L' \leftarrow$ Extract all outlinks of l
9: **for** each $l' \in L'$ **do**
10: $(l', s', a') \leftarrow$ Get action features a' of l'
11: Add (l', s', a') to (s', a') pair of B with initial Q-value
12: **end for**
13: **end while**
14:
15: **while** visited_pages $<$ *LIMIT_PAGES* **do**
16: **if** With probability ϵ **then**
17: Select a (s, a) pair uniformly at random from B and select a link (l, s, a) from the pair
18: **else**
19: Select a (s, a) pair from B with highest Q-value and select a link (l, s, a) from the pair
20: **end if**
21: **if** l is visited **then**
22: continue
23: **end if**
24: $r, s' \leftarrow$ Fetch and parse page (l, s, a)
25: $L' \leftarrow$ Extract all outlinks of l
26: **for** each $l' \in L'$ **do**
27: **if** l' is visited **then**
28: continue
29: **end if**
30: $(l', s', a') \leftarrow$ Get action features a' of l'
31: **end for**
32: **if** visited page is relevant **then**
33: $w \leftarrow w + \alpha\left[r - \hat{q}(s, a, w)\right] \nabla\hat{q}(s, a, w)$
34: **else**
35: Choose a' as a function of $\hat{q}(s', \cdot, w)$ with ϵ-greedy policy
36: $\delta \leftarrow r + \gamma\hat{q}(s', a', w) - \hat{q}(s, a, w)$
37: $w \leftarrow w + \alpha\left[r + \gamma\hat{q}(s', a', w) - \hat{q}(s, a, w)\right] \nabla\hat{q}(s, a, w)$ #*original* update
38: $w \leftarrow w + \alpha\left[r + \gamma(\hat{q}(s', a', w) - \delta) - \hat{q}(s, a, w)\right] \nabla\hat{q}(s, a, w)$ #*moderated* update
39: **end if**
40: **for** each $(\cdot, \cdot)$ pair $\in B$ **do** #*synchronous* method
41: Calculate Q-value of $(\cdot, \cdot)$
42: Update $(\cdot, \cdot)$ to B with Q-value
43: **end for**
44: **for** each $(s', \cdot)$ pair $\in L'$ **do** #*asynchronous* method
45: Calculate Q-value of $(s', \cdot)$
46: Add $(l', s', \cdot)$ to $(s', \cdot)$ pair of B with Q-value
47: **end for**
48: visited_pages $\leftarrow$ visited_pages + 1
49: **end while**

Each feature of state and action is specified with two indexes discretized 5 and 6 buckets. In the case of 'Relevance Change of Target Topic' feature, it is discretized into 5 buckets. The parameter δ_1 and δ_2 used for discretizing its value are set to 0.1 and 0.3 respectively and smoothing factor β is set to 0.4. The number of features are different depending on a target topic because categories vary according to the target topic. In our experiments, categories are empirically pre-selected based on the Open Directory Project (ODP, `http://www.dmoz.org`, `http://curlie.org/`), an open directory of Web links. The ODP is a widely-used Web taxonomy that is maintained by a community of volunteers. Among the target topics of our experiments, for Cancer, four related categories, Disease, Medicine, Oncology and Health, are chosen from the category hierarchy. For Fiction, there are two related categories, Literature and Arts. For Olympics, one related category, Sports, is selected, for Cameras, three categories, Photography, Movies, and Arts, for Geology, two categories, Earth and Science, and for Poetry, two related categories, Literature and Arts. A state is a eight + $2 \cdot \alpha$ dimensional vector where α is the number of categories selected from a category hierarchy. Like a state, an action is represented as a six + $2 \cdot \alpha$ dimensional feature vector.

In this section, we compare our three proposed crawling algorithms based on reinforcement learning (synchronous method, asynchronous method, and asynchronous method with moderated update), and an algorithm without learning. The no-learning algorithm is served as a baseline of performance. It uses w2v based cosine similarity as a priority in the crawling frontier and does not use any features or learning update formulas presented in Section 4.3. In no learning algorithm, when crawling a page, each outlinks is added to the frontiers with its w2v based cosine similarity based on the URL text, the anchor text and surrounding text of the hyperlink. Then, a link with the highest priority is selected for the next crawling.

Given a crawling task that visit 10,000 pages, Figure 4.1 shows the accumulated number of relevant pages per time step during the crawling task. The x axis represents time step of a crawling task and the y axis marks the accumulated number of relevant pages per time step. Each curve is the average of 100 tasks. For each task, all data obtained during a crawling task is initialized, for example, hyperlinks in the frontier and parent information of hyperlinks, etc., but the weight vector learned in the precedent task is maintained. For all target topics, the algorithm without learning finds relevant pages progressively as time steps increase. For some topics such as Olympics, Cameras, Geology, and Poetry, we can see a sharp increase in early time steps. This is because a given seed page is a main page corresponding to each target topic, thus, the agent has more chance to meet relevant pages in early time steps. Compared to no learning with monotonous increase, reinforcement learning algorithms speed up finding relevant pages. In particular, for topic Cancer, Fiction, Geology, and Poetry, the accumulated number of relevant pages is increased abruptly. It means that reinforcement learning effectively helps find relevant pages as time steps increase. For topic Olympics and Cameras, the agent based on reinforcement learning follows a similar curve as no learning but finds more relevant pages.

Figure 4.2 displays the quality of the experimental results above. The x axis marks w2v-based relevance discretized into 10 intervals and the y axis represents the number of relevant pages per relevance level. Each bar is the average of 100

tasks. In lower or higher levels of relevance, there is no significant difference among all algorithms because there are not many pages corresponding to those relevances on the environment. Meanwhile, it is apparent that learning and no learning algorithms have a big difference of performance for 3rd to 6th relevance levels depending on the distribution of relevant pages on the Web graph for each topic. Among learning algorithms, their performance results are similar or slightly different depending on topics. For topic Cancer, Geology, and Poetry, learning algorithms find similar number of relevant pages per relevant level. For topic Fiction, Olympics and Cameras, we can see a bit difference of performance between learning algorithms.

Figure 4.3 shows learning curves of three algorithms on the different target topics. Each curve is the average of 10 independent trials. The x axis represents the number of crawling tasks and the y axis marks the number of relevant pages per task. A crawling task consists of visiting 10,000 pages. The learning curves show how the learning is improved as a task repeats. For each task, the weight vector learned in the precedent task is maintained and all other data obtained during a crawling task is initialized, for example, hyperlinks in the frontier and parent information of hyperlinks, etc. The same seed pages are given for each task. Thus, each crawling task starts in the same condition except the weight vector. By the learning curves, we can see how a crawling task is improved given the same condition. We compare reinforcement learning algorithms with no learning algorithm. Since each task is executed under the same condition, no learning algorithm's performance is the same regardless of the number of crawling tasks. For all target topics, reinforcement learning algorithms have better performance than the algorithm without learning. Those performances are generally 1.2 to 1.6 times, in particular, for topic Cancer, 2.5 times better than that of no learning algorithm. In Figure 4.3(a)–(c), among all reinforcement learning algorithms, the synchronous method has the highest performance. In asynchronous methods, the moderated update outperforms the original update. In Figure 4.3(d)–(f), the moderated update finds relevant pages more than the other algorithms. Relevant pages of the three topics exist sparsely on the environment and thus those topics need more exploration. Since action values of unvisited links in the frontier are calculated at different time steps, those different values can hinder a good selection. By the moderated method, we can reduce the action value differences between time steps and effectively explore promising links while being less influenced by time steps.

From Figure 4.3, we see that the synchronous method is better in general but the overhead of updating all action values cannot be ignored. For example, the computation time of synchronous method is 654 seconds while that of asynchronous and no learning are 55 and 28 seconds respectively for one crawling task of topic Olympics. Thus, if we consider the overhead of updates, the asynchronous method with moderated update can be a good alternative and may be even better in the environment in which the agent needs more exploration and in which action value differences are influenced by time steps.

4.5 Related Work

Chakrabarti et al. [10] first introduced focused crawling to selectively seek out pages that are relevant to a pre-defined set of topics, using both a classifier and a distiller, to guide the crawler. The classifier is based on naive Bayesian method and evaluates the relevance of a page with respect to the given topics. The distiller identifies if a page is a great access point to many relevant pages within a few links.

Diligenti et al. [27] introduced a context-focused crawler that improves on traditional focused crawling. Their classifier is trained by a context graph with multiple layers and used to estimate the link distance of a crawled page from a set of target pages.

Basically, the relevance is measured based on the textual content but the Web graph structure is also exploited to evaluate relevance in many crawling methods. PageRank and HITS are two famous algorithms that rely on the link structure of the Web to calculate the relevance to the target pages.

Kleinberg [61] proposes the HITS algorithm that discovers relevant authoritative pages by the link structure analysis. He introduces the notion of authority and hub based on the relationship between relevant pages. An authority is a prominent page related to the query topic. A hub is a page that has links to many relevant authoritative pages.

Page et al. [80] introduce PageRank, a method for rating Web pages based on the Web graph structure. The importance of the pages, PageRank is measured by counting citations or backlinks to a given page.

The intelligent crawler [2] proposed by Aggarwal et al. statistically learns the characteristics of the link structure of the Web during the crawl. Using this learned statistical model, the crawler gives priorities to URLs in the frontier. The crawler computes the interest ratios for each of the individual factors such as page content, URL token, link, and sibling. Then, the interest ratios are combined by a linear combination of the weighted logarithms of individual factors. The combined ratio is used to estimate the probability of a candidate URL satisfying the user needs.

Almpanidis et al. [3] propose a latent semantic indexing classifier that combines link analysis and content information. Chau et al. [12] focus on how to filter irrelevant documents from a set of documents collected from the Web, through a classification that combines Web content analysis and Web structure analysis. For each page, a set of content-based and link-based features is defined and used for input data of the classification.

Most crawling approaches use classification methods to evaluate priority but a few previous works have applied reinforcement learning to focused crawling.

Rennie et al. [86] first use reinforcement learning in Web crawling. Their algorithm calculates Q-values for hyperlinks in training data, then learns a value function that maps hyperlinks to future discounted reward by using naive Bayes text classifiers. It performs better than traditional crawling with breadth-first search but the training is performed off-line. The authors define that the state is the bit vector indicating which pages remain to be visited and the actions are choosing a particular hyperlink in the frontier.

Grigoriadis et al. [44] propose a focused crawling that uses reinforcement learning

to estimate link score. The algorithm is composed of two modes such as training and crawling. In training, an agent visits pages by selecting randomly hyperlinks until it finds a relevant page. As the agent visits pages, a neural network gradually learns states' estimate values. In crawling, the agent evaluated all outlinks using the trained neural network but in fact these link scores inherit their parent's score. These hyperlinks with their scores are added to the queue. The crawler selects the hyperlink with the highest score. In [44], every page represents a state that consists of a feature vector of 500 binary values and actions are the hyperlinks in each page. Each binary value in a state represents the existence of a specific keyword and the binary vector of a state is used as input data of neural network to estimate its state-value. State values are approximated with gradient descent function approximation based on neural network.

InfoSpiders [74, 75] is a distributed adaptive online crawler based on genetic algorithms. It is inspired by ecological models in which a population of agents lives, learn, reproduce and die in an environment. Each agent consists of the genotype, that determines its searching behavior, and a neural network used to evaluate links. The neural net learns to estimate the Q values of links extracted from a source page. The cosine similarity between the agent's keyword vector and the page containing the link is calculated and it is used as a reward. Then, the neural net's link scores and the cosine similarity are combined. This procedure is similar to the reinforcement learning algorithm. Based on the combined score, the agent selects one of the links in the frontier. In [74, 75], all hyperlinks in the frontier are considered as actions. Each input unit of the neural net receives a weighted count of the frequency with which the keyword occurs in the vicinity of the link to be traversed.

Those methods require an off-line training phase and their state definitions do not consider the link structure; for example, states are represented with a vector which consists of the existence or frequency of specific keywords. Our method learns link scores in an online manner, with new representations of states and actions considering both content information and the link structure.

Meusel et al. [76] combine online classification and bandit-based selection strategy. To select a page to be crawled, they first use the bandit-based approach to choose a host with the highest score. Then, a page from the host is taken using the online classifier. Similarly, Gouriten et al. [42] use bandits to choose estimators for scoring the frontier.

Like reinforcement learning, crawling involves a trade-off between exploration and exploitation of information: greedily visiting URLs that have high estimate scores vs exploring URLs that seem less promising but might lead to more relevant pages and increase overall quality of crawling. Pant et al. [81] demonstrate that the best-N-first outperforms the naive best-first. The best-N-first algorithm picks and fetches top N links from the frontier for each iteration of crawling. Increasing N results in crawlers with greater emphasis on exploration and consequently a reduced emphasis on exploitation [81].

4.6 Future Work

We have seen how to model a crawling task using the MDP formalism. The crawling agent effectively learns link scores based on reinforcement learning. In this section, we discuss some possible extensions to improve our method.

First, the dataset we used is sufficiently good to verify the effectiveness of reinforcement learning based crawler but it should be evaluated in larger and various datasets, such as full English Wikipedia and dataset from the site `http://commoncrawl.org/`, etc.

Second, the state and action representation is based on both content information and the link structure. Among features, categories related to a target topic is required to be pre-selected by a system designer. As we have seen in section 2.4, approximated value functions rely on feature. Thus, poor feature selection may result in poor performance. Instead of manually selecting features based on domain-specific knowledge, it is necessary to build up an efficient mechanism for category selection.

Finally, the method we have proposed is based on a single agent. In a real problem, the number of outlinks of a page may be much larger and the size of the frontier will grow fast. An agent does a crawling task in a large environment, it will be a time-consuming. To accelerate crawling performance, we can consider multiple crawling agents. That is, executing multiple crawling agents in parallel. Each agent crawls Web pages and learns action values independently. The agents will explore different parts of the environment. They will have different experience because they will visit different pages (states) and receive different rewards. As a result, they will also have distinct scoring policies with respect to their own value functions. We can consider each agent to be completely independent, i.e. they do not share any information including the frontier. If the frontier is shared by all agents, scoring policies also have to be merged in some way. A few research works [45, 100, 33] show that combining the different policies outperforms a single agent.

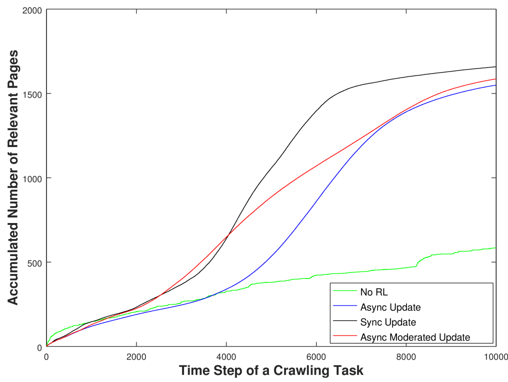
Grounds et al. [45] present an approach of parallelization to solve single-agent RL problems more quickly. The value functions are represented using linear function approximators and updated by SARSA(λ) algorithm. Each agent learns independently in a separate simulation. During learning, agents update feature weights and maintain visit counts for each features. In every predefined time step, a synchronous merging operation is executed and it calculates a weighted average of the feature value estimates collected from all the agents, favoring features with high probabilities of visitation. To improve the merging process, selective communication of significant information and asynchronous message passing are proposed.

Wiering et al. [100] present several ensemble approaches that combine the policies of multiple independently learned RL algorithms. Among the five RL algorithms used for combining, three algorithms, Q-learning, Sarsa, QV-learning, learn state-action values and two algorithms, actor–critic (AC) and AC learning automaton, learn preference values of policies. The authors combine the different policies with four ensemble methods such as majority voting (MV), rank voting, Boltzmann multiplication (BM), and Boltzmann addition. Their experiments show that the BM and MV ensembles significantly outperform the other ensemble methods and the single RL algorithms.

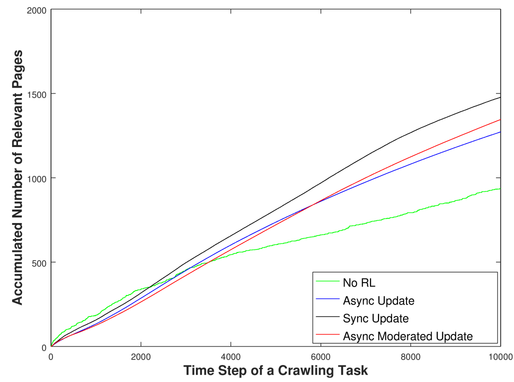
FauBer et al. [33] propose several ensemble methods that combine parameterized state-value functions of multiple agents. Temporal-Difference (TD) and Residual-Gradient (RG) learning are used to update state-value functions. These functions are combined with Majority Voting and Average of the state-values to learn joint policies. Another improvement is an average predicted state-value that explicitly combines the state-values of all agents for the successor state. Their experiments show that ensemble methods outperforms a single agent.

4.7 Conclusion

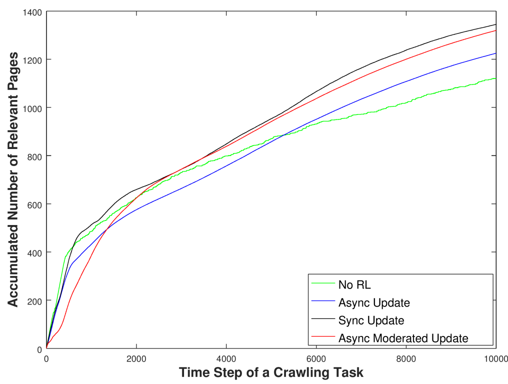
In this chapter, we applied reinforcement learning to focused crawling. We propose new representations for Web pages and next link selection using contextual information and the link structure. A number of pages and links are generalized with the proposed features. Based on this generalization, we used a linear function approximation with gradient descent to score links in the frontier. We investigated the trade-off between synchronous and asynchronous methods. As an improved asynchronous method, we propose moderated update to reach a balance between action-values updated at different time steps. Experimental results showed that reinforcement learning allows to estimate long-term link scores and to efficiently crawl relevant pages. In future work, we hope to evaluate our method in larger and various datasets, such as full English Wikipedia and dataset from the site `http://commoncrawl.org/`, etc. Another challenging possibility is to build up an efficient mechanism for categories selection to avoid a system designer pre-selecting proper categories for each target topic. We also want to investigate other ways to deal with exploration/exploitation. Finally, extending single agent based method to multiple crawlers will be an interesting future work.



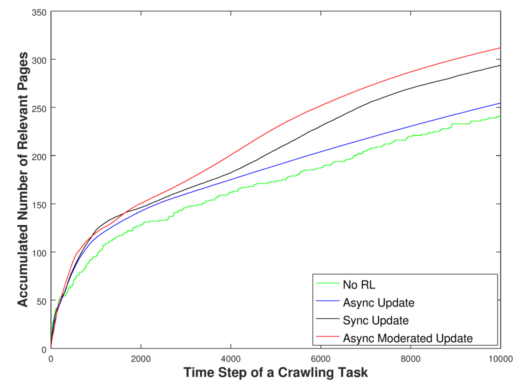
(a) Cancer



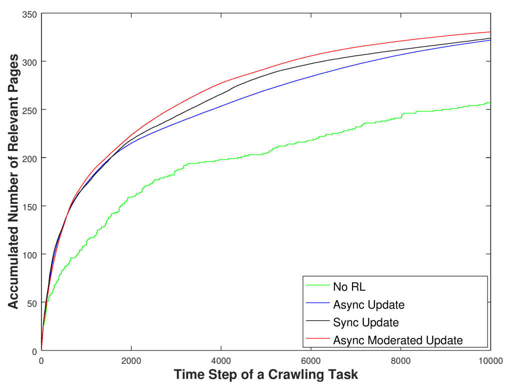
(b) Fiction



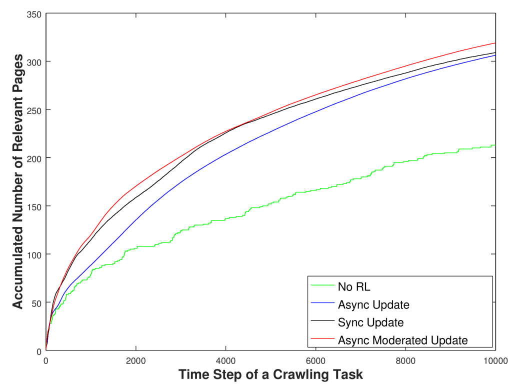
(c) Olympics



(d) Cameras

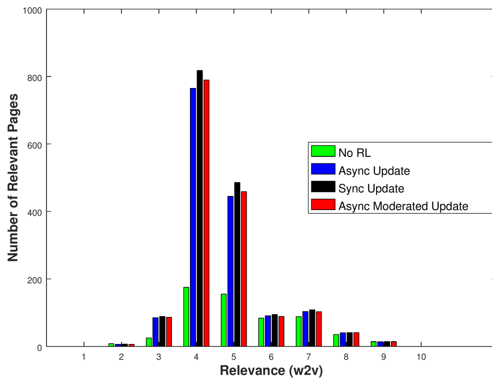


(e) Geology

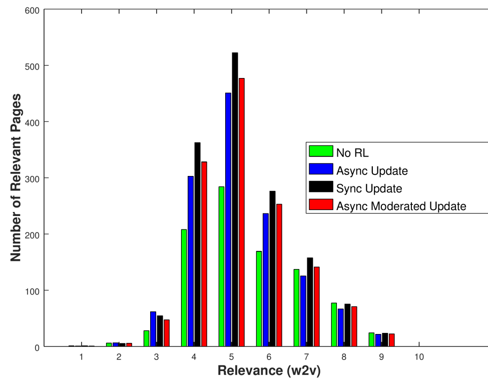


(f) Poetry

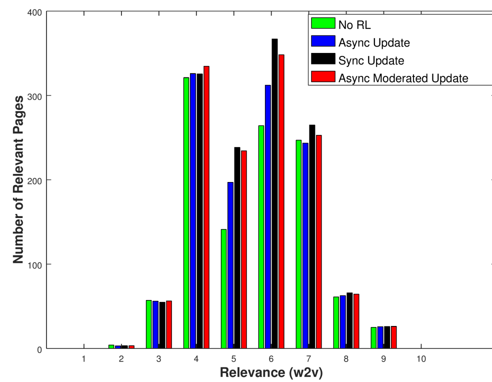
Figure 4.1 – Accumulated Number of Relevant Pages per Time Step



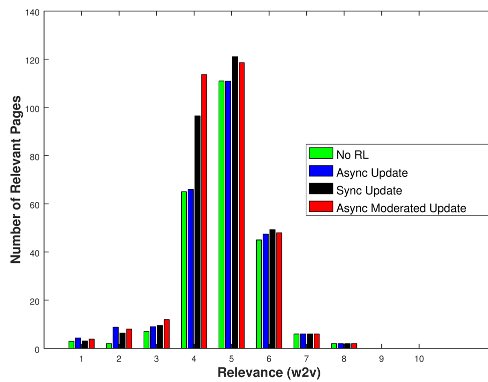
(a) Cancer



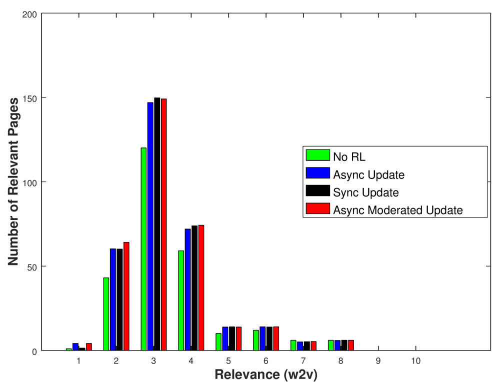
(b) Fiction



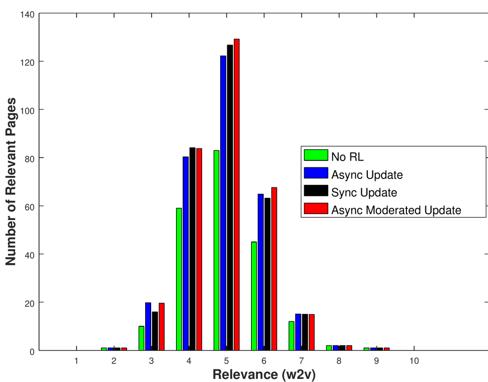
(c) Olympics



(d) Cameras

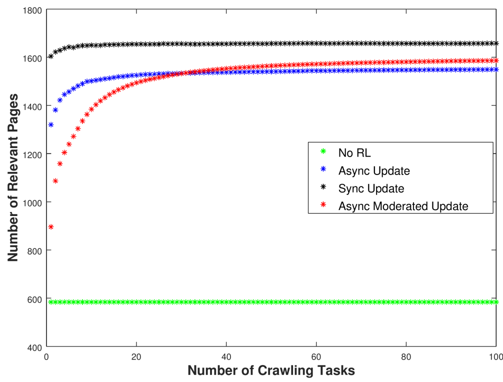


(e) Geology

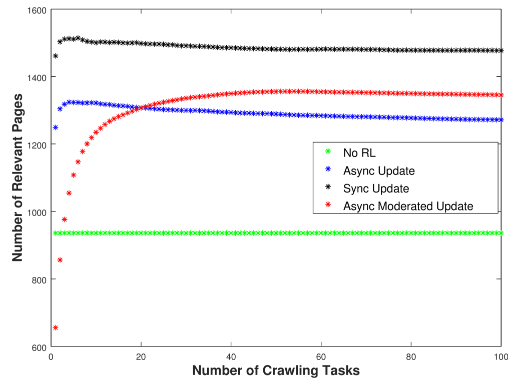


(f) Poetry

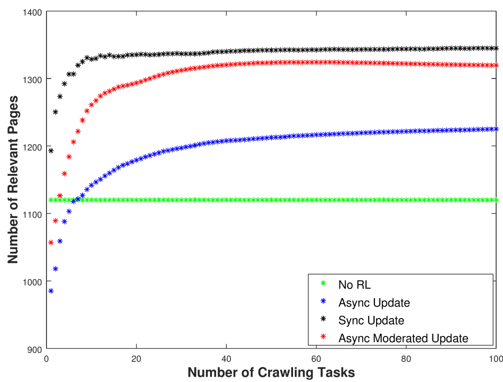
Figure 4.2 – Number of Relevant Pages per Relevance Interval



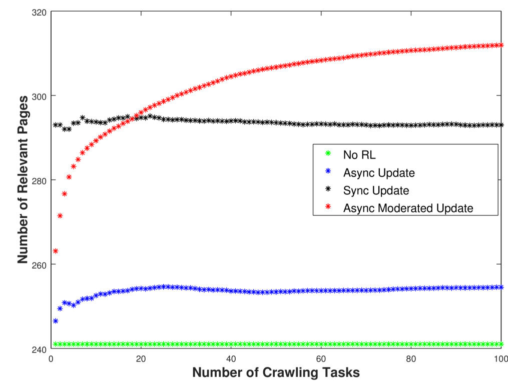
(a) Cancer



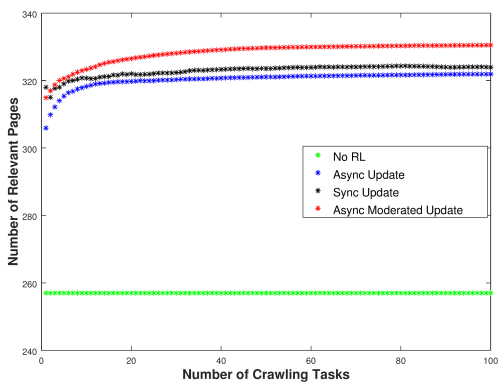
(b) Fiction



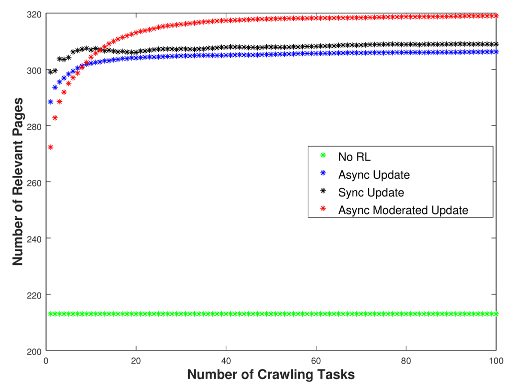
(c) Olympics



(d) Cameras



(e) Geology



(f) Poetry

Figure 4.3 – Number of Relevant Pages as Tasks Repeat

Chapter 5

Influence Maximization

In this chapter, we continue our discussion with another domain with rich applications, the influence maximization problem.

5.1 Introduction

Word-of-mouth, buzz marketing, and viral marketing have been used as effective marketing strategies traditionally conducted in offline networks. Offline social network activities have been extended to online social networks such as Facebook and Twitter, etc., and the popularity of such social media has rapidly increased over the last decade. Social networking sites are good platforms not only for communication among users but also for information diffusion. Some information is disseminated to many other users through the network. Since these social networks can play an important role in the spread of information at a very large scale, they have attracted interest in the area of online viral marketing. Detecting influential users is an important problem for efficient online viral marketing.

Suppose that a company develops a new product and hopes to market the product to a large number of people on an online network. The company would like to choose some users of the network to give free samples of the product while expecting they spread information after use, recommend it, or ultimately lead to purchase it. When we want to advertise the product efficiently with a limited budget for giving samples, a problem that can arise is to determine who the most influential users are. The problem assumes that a few influential users, i.e., seeds, can trigger a large diffusion of information via a network.

Given a social network, the influence maximization problem is to choose an optimal initial seed set of a given size to maximize influence under a certain information diffusion model such as the independent cascade (IC) model, the linear threshold (LT) model, etc. It was first proposed by Domingos and Richardson [31, 87] and formulated as an optimization problem by Kempe et al. [60]. The IM problem has been actively studied in the literature [31, 87, 60, 57, 77, 14].

In many existing algorithms, the whole topological structure of a social network is given in advance. However, it is known that the complete knowledge of the topological structure of a social network is typically difficult to obtain [107, 70, 39, 77]. Even though the complete graph is given, the graph may change dynamically [107].

Mihara et al. [77] address influence maximization problem for unknown graphs and show that a reasonable influence spread can be achieved even when knowledge of the social network topology is limited and incomplete.

Another unrealistic aspect of many existing methods is that these methods do not take into account topical interests of users. In fact, users have their own interests and are more likely to be influenced by information that is related to their interests. That is, the spread of information varies depending on the topic of a post. There are some works that study the topic-based influence maximization problem [49, 15, 13, 66]. Their methods consider multiple topic distributions on nodes and a query but we focus on one target topic and study influence maximization for a given topic.

In this study, assuming that the graph structure is incomplete or can change dynamically, we address a topic-based influence maximization problem for an unknown graph and show how it can be phrased, again, as a Markov decision process. In order to know a part of the graph structure and discover potentially promising nodes, we probe nodes that may have a big audience group. Then, we find the most influential seeds to maximize topic-based influence by using reinforcement learning [94]. As we select seeds with a long-term impact in the influence maximization problem, action values in the reinforcement learning signify how good it is to take an action in a given state over the long run. Therefore, we learn action values of nodes from interaction with the environment by reinforcement learning. For this, nodes are generalized with some features that represent a node's proper information and relation information with respect to surrounding nodes and we define states and actions based on these features. Then, we evaluate action value for each probed node and select a node with the highest action value to activate.

In the following section, we review the influence maximization problem. In Section 5.3, we present topic-based influence maximization problem for unknown graphs and our method for the problem. Section 5.4 presents prior work in the literature. Section 5.5 discuss several opportunities for future work and we conclude in Section 5.6.

5.2 Background

The influence maximization problem is to choose an optimal initial seed set of a given size in a given social network that maximizes the total amount of influence under a certain information diffusion model.

The influence maximization problem is formally defined as follows:

Problem [Influence Maximization Problem]. We fix a graph $G = (V, E)$ and a parameter (called budget) $k \leq |V|$ where $v \in V$ are nodes and $e \in E$ are edges between nodes. Let $\sigma(S)$ the expected number of active nodes through a seed set S under a given diffusion model. The *influence maximization problem* is to select a seed set $S \subseteq V$ with $|S| = k$ that maximizes the influence $\sigma(S)$.

The influence function is *monotone* if $\sigma(S) \leq \sigma(T)$ for all $S \subseteq T \subseteq V$ and it is *submodular* if $\sigma(S \cup \{v\}) - \sigma(S) \geq \sigma(T \cup \{v\}) - \sigma(T)$ for all $S \subseteq T \subseteq V$ and for all $v \in V$.

Algorithm 15 shows the greedy algorithm of the influence maximization problem.

Algorithm 15 Greedy(G, k, p)

1: Input: Graph G, budget k, influence probabilities p
2: $S_0 \leftarrow \emptyset$
3: **for** $i = 1, 2, \ldots, k$ **do**
4: $\quad v \leftarrow \arg\max_{v \notin S_i} \; [\sigma(S_{i-1} \cup \{v\}) - \sigma(S_{i-1})]$
5: $\quad S_i \leftarrow S_{i-1} \cup \{v\}$
6: **end for**
7: return S_k

In each round, the algorithm computes the marginal influence of each node $v \notin S_i$ and adds the maximum one to the seed set S_i until $|S| = K$.

The influence maximization problem was first proposed by Domingos and Richardson [31, 87] and formulated as an optimization problem by Kempe et al. [60]. It is NP-hard to determine the optimum for influence maximization but the greedy algorithm (Algorithm 15) provides a $(1 - 1/e)$ approximation ratio for a non-negative, monotone, and submodular influence function $\sigma(S)$ [60].

There are two well-known diffusion models: the independent cascade (IC) model and the linear threshold (LT) model. In the IC model, each active node tries to activate each inactive neighbor with a predefined influence probability. In the LT model, an inactive node becomes active if the ratio of its active neighbors exceeds a predefined threshold. In these models, a node can change its state from inactive to active but it cannot switch in the other direction.

Most existing algorithms for the influence maximization problem are based on the independent cascade (IC) model proposed by Goldenberg et al. [41]. The IC model starts with an initial (or seed) set of active nodes, denoted S_0. For each time step t, each node v activated in step t, i.e., $v \in S_t$, tries to activate each inactive neighbor w. This attempt succeeds with a probability $p_{v,w}$. If there are multiple nodes that try to activate node w, their attempts are sequenced in an arbitrary order. If v succeeds, then w will become active in step $t+1$ and it is added in S_{t+1}. However, whether w is activated or not, v cannot make any further attempts to activate w in subsequent rounds. The process continues until no more activations are possible.

The linear threshold (LT) model is proposed by Granovetter and Schelling [43, 90]. Suppose that a node v is influenced by each neighbor w with a weight $b_{v,w}$ such that $\sum_{w:\text{ neighbor of } v} b_{v,w} \leq 1$. Each node v chooses a threshold θ_v uniformly at random from the interval $[0, 1]$. Given an initial (or seed) set of active nodes, denoted S_0, at each time step t, all nodes that were active in step $t-1$ remain active, and any node v is activated if the total weight of its active neighbors is at least θ_v: $\sum_{w:\text{ active neighbor of } v} b_{v,w} \geq \theta_v$. This diffusion process ends when no more node is to be activated.

In influence maximization problems, we find the most influential seeds to maximize influence under a certain information diffusion model such as the IC model, the LT model, etc.

5.3 Topic-Based Influence Maximization Algorithm for Unknown Graphs

We first define our problem and briefly explain our method. Then, we present two main parts of our algorithm: selecting seeds and probing nodes.

5.3.1 Problem Statement and our Method

In many existing algorithms, the whole topological structure of a social network is assumed to be provided and the complete knowledge is used to find the optimal seed sets. However, it is known that the complete knowledge of the topological structure of a social network is typically difficult to obtain [107, 70, 39, 77]. Even though the complete graph is given, the graph may change dynamically [107]. Thus, in this study, we assume that the graph structure is incomplete or can change dynamically. We find the most influential seeds for an unknown graph while probing nodes in order to know a part of the graph structure and discover potentially promising nodes. The most related work is influence maximization for unknown graphs proposed by Mihara et al. [77]. Their work shows that a reasonable influence spread can be achieved even when knowledge of the social network topology is limited and incomplete.

Another unrealistic thing in many existing methods is that these methods do not take into account topical interests of users. In fact, users have their own interests and are more likely to be influenced by information that is related to their interests. That is, the spread of information varies depending on the topic of a post. For example, a post about cars will be spread though users who are interested in cars. It will be different from the information spread of a post about dogs. There are some works that study topic-based influence maximization problems [49, 15, 13]. Their methods consider multiple topic distributions on nodes and a query but we will focus on one target topic and study influence maximization for a given topic.

In this study, we address a topic-based influence maximization problem for an unknown graph. Assuming that a social graph $G = (V, E)$ is directed, V is known but E is not known, we find the most influential seeds to maximize topic-based influence while probing nodes that may have a big audience group. For selecting a seed, instead of differentiating all individual nodes, we first choose some features that represent a node's proper information and relation information with respect to surrounding nodes. We will call the generalized form with the features about relation information the *state*. The generalized form with the features about a node's proper information is called *action*. Then, we evaluate a node based on its action and state. An *action value* will signify how valuable it is to choose an action (a node) to activate in a given state in order to maximize the influence spread. The agent chooses a node based on its action value to activate. Since it is similar to the concept of action value in reinforcement learning [94], we use the reinforcement learning methodology to learn action values. In short, we probe nodes to discover the graph structure and choose nodes with the highest action value as seeds.

Before we move to the next subsection, we discuss the influence maximization problem and the focused crawling problem to help understand our modeling. Recall that in the focused crawling, the agent collects Web pages relevant to the target

topic using a frontier. The problem itself does not consider long-term effects, but a reinforcement learning approach allows to estimate long-term link scores, as we have seen in the previous chapter. In the influence maximization, the agent aims to choose the most influential seeds to maximize influence. This problem already takes into account long-term values but not necessarily the planning dimension that reinforcement learning introduces.

Those two problems are based on different objectives and have been studied in different ways, but they have some similarities caused from structural characteristics of web graphs and the nature of tasks.

In the focused crawling, Web pages are connected by hyperlinks but they are not linked randomly. Pages are likely to be linked to topically related pages (see Section 4.2). In the influence maximization, users are also likely to be friends of other users who have similar interests. The feature selection in the following subsection is inspired by the features used in the focused crawling problem (see Section 4.3.1).

In addition, selecting seeds with the highest action values is similar to link-selections from the frontier in prioritized order in the focused crawling problem. Thus, it can have the similar issue discussed in the previous chapter and action values can be balanced in the same way as we did in the focused crawling (see Section 4.3.3). However, while the crawling agent selects a link from a frontier for each time step, the agent in the IM problem selects one probed node with the highest action values, and then depends on information diffusion from the selected node.

We continue the details of our modeling for the influence maximization problem in the following subsection while considering such similarities and differences.

5.3.2 Modeling and Algorithm

We first explain how to define states and actions and to compute the value of actions in order to select seeds and then discuss how to probe nodes. The whole algorithm is shown as Algorithm 16.

5.3.2.1 Selecting Seeds

As we mentioned above, a node is generalized with some features that represent a node's proper information and relation information with respect to surrounding nodes, called action and state, respectively. Then, we evaluate a node based on its state and action. The features of states and actions are presented in the following.

State. The state features are based on relation information with respect to surrounding nodes. Since the complete graph structure is not known in advance, each node does not know all actual parents (i.e., incoming nodes) and then we have to progressively update parent information while visiting nodes by probing or tracing activated nodes. When visiting a node, we let its child nodes know who are parents by referencing the current visiting node.

For topic (or category) based features, in order to decide whether a post is relevant to the given topic (or category), we can use a classification method or cosine similarity between a word vector of the given topic (or category) and that of a post. When we use cosine similarity, a threshold θ has to be selected. Then, if

similarity is greater than the threshold θ, we can consider it relevant to the given topic.

Based on this, we can compute a posting rate of a given topic (or category) among all posts generated by a user as follows: for a user, the number of the user's posts that are relevant to the given topic (or category) is divided by the number of all posts generated by the user.

Then, as in the previous chapter, we discretized the posting rate into 10 buckets according to value ranges: the range $[0.0, 0.1]$ by 0, $[0.1, 0.2]$ by 1, ..., $[0.9, 1.0]$ by 9.

- **Average Posting Rate of All Parents for Given Topic:** The average posting rate for the given topic over all parent nodes is calculated and discretized according to value ranges.

- **Average Posting Rate of All Parents for Categories:** First, some categories relevant to the given topic are properly preselected from a category hierarchy such as the Open Directory Project (ODP, `https://www.dmoz.org/`) by the system designer. Then, based on the preselected categories, the average posting rate for each category over all parent nodes is calculated and discretized according to value ranges.

- **Posting Rate Change for Given Topic:** The current node's posting rate for the given topic is compared to the weighted average posting rate for the given topic over all its ancestors on the probed graph structure. The weighted average posting rate over all its ancestors is computed in an incremental manner by applying an exponential smoothing method on its parents nodes.

  We denote posting rate for the given topic in node x as $posting(x)$. The weighted average posting rate from all x's ancestors to x, $w_{posting}(x)$, is obtained by $w_{posting}(x) = \beta \cdot posting(x) + (1-\beta) \cdot \max_{x' \to x} w_{posting}(x')$, where $\beta (0 < \beta < 1)$ is a smoothing factor. If the current node has many parents, i.e., many path from its ancestors, the maximum average among them, $\max_{x' \to x} w_{posting}(x')$, is used for the update. The exponential smoothing assigns exponentially decreasing weights on past observations. In other words, recent observations are given relatively more weight than the older observations.

  Then, we can calculate the posting rate change between current node z and $w_{posting}(x)$ where x is a parent of z: $change \leftarrow posting(z) - \max_{x \to z} w_{posting}(x)$.

  The change helps to detect how much the posting rate of the current node for the given topic is increased or decreased than the average posting rate of its ancestors.

  The posting rate change to the current node from its ancestors is discretized according to value ranges. With predefined parameters δ_1 and δ_2, the difference within δ_1 is indexed by 0, the increase by δ_2 is indexed by 1, increase more than δ_2 is indexed by 2, decrease by δ_2 is indexed by 3, and decrease more than δ_2 is indexed by 4.

- **Distance from the Last Activated Node:** The distance from the last activated node is simply calculated by adding 1 to the parent's distance. If there are many parents, the minimum distance among them is used. The distance value is capped at 9 to keep it within a finite range.

$$distance = \begin{cases} 0 & \text{if it is activated} \\ 1 + \text{parent's distance} & \text{otherwise} \end{cases} \tag{5.1}$$

Action. The action features are based on a node's proper information consisting of two types of information. One is general behaviors of the user on a social network and the other is the user's topic interest. The feature 'Number of Children' is a good indicator to see if a user has a big audience group or not. The feature 'Number of Posts' can be used to predict user's activity. The two other features, such as 'Posting Rate for Given Topic' and 'Posting Rate for Categories', represent user's interest.

- **Number of Children:** This is obtained by simply counting child nodes and the count is discretized according to 10 value ranges.
- **Number of Posts:** This is also obtained by simply counting posts generated by a user and the count is discretized according to 10 value ranges.
- **Posting Rate for Given Topic:** This is the posting rate for the given topic among all posts generated by a user and the rate is discretized according to value ranges.
- **Posting Rate for Categories:** Based on the preselected categories, the posting rate of each category among all posts generated by a user is computed and the rate is discretized according to value ranges.

The size of a discretized state space is $(10)^3 \cdot (10)^{\text{num of categories}}$ and the size of action space is $(10)^3 \cdot (10)^{\text{num of categories}}$. For example, if there is just one category, the size of the state space is 10^4 and the size of the action space is 10^4.

Based on feature values discussed above, we evaluate the action value of each node. As in the previous chapter, we use gradient-descent based linear function approximation (see Section 2.4) to compute action values because the state-action space is very large. We define a weight vector $\mathbf{w}$ that has the same size of feature vector $\mathbf{x}(s,a)$. Recall that the value of action a in state s, $\hat{q}(s,a,\mathbf{w})$, is approximated by linearly combining feature vector $\mathbf{x}(s,a)$ and weight vector $\mathbf{w}$:

$$\hat{q}(s,a,\mathbf{w}) \doteq \mathbf{w}^{\intercal}\mathbf{x}(s,a) \doteq \sum_{i=1}^{d} w_i x_i(s,a) \tag{5.2}$$

and the weight vector $\mathbf{w}$ is updated as follows:

$$\mathbf{w}_{t+1} \doteq \mathbf{w}_t + \alpha \left[r_{t+1} + \gamma \hat{q}(s_{t+1}, a_{t+1}, \mathbf{w}_t) - \hat{q}(s_t, a_t, \mathbf{w}_t) \right] \nabla \hat{q}(s_t, a_t, \mathbf{w}_t). \tag{5.3}$$

Here, α $(0 < \alpha \leq 1)$ is the step-size parameter that influences the rate of learning; r is a reward received for taking action a; the discount rate γ $(0 \leq \gamma < 1)$ determines

the present value of future rewards. s_{t+1} is the next state and a_{t+1} is an action in s_{t+1} under a given policy π. We define reward r as the rate of activated child node, and (s_{t+1}, a_{t+1}) as state-action pair of a child node that has the highest action value among all child nodes.

We update weight vector $\mathbf{w}$ for all activated nodes. After activating a seed, the information influence is spread depending on users' choices. Thus, the update has to be on a node explicitly activated by a learning agent. We extend the scope of action. Activating a node by the learning agent is an *explicit action*. We consider activating nodes by influence spread from the initially selected node as an *implicit action* because they yield rewards and next states in the same mechanism of the explicitly activated nodes.

For each probed node, we use the learned $\mathbf{w}$ to evaluate an action value of the node and select nodes with the highest action values as seeds. However, if action values of the probed nodes are computed at different time steps and they are not synchronized with the same weight vector $\mathbf{w}$, we may have the same issue as we have seen in the focused crawling problem (see Section 4.3.3). In this case, it would be good to use *moderated* update in order to reduce the action value differences at different time steps by manipulating weight updates. The details of the moderated update is presented in Section 4.3.3.

5.3.2.2 Probing Nodes

As a complete knowledge of a social network is not given in advance, we need to probe nodes in order to partially know the graph. An effective method of probing nodes will be to compute action values for all inactive nodes and to choose a node with the highest action value. However, if there are a huge number of nodes, the computational cost can be extremely high. Alternatively, we can use the out-degrees of nodes. In fact, a node with high out-degree means that the node has a big audience. Such node is likely to spread information more widely than nodes with low out-degree. In Sample Edge Count (SEC) [70], a biased sampling method, nodes with the highest expected degree are greedily probed. This method is effective for finding hub nodes of large degree. Thus, in our algorithm, the expected degrees of nodes are initially set to 0 and we update them progressively while probing nodes and select nodes with the highest expected degrees.

The pseudocode of topic-based influence maximization for an unknown graph is shown in Algorithm 16. The expected out-degree d_{out} of each node in V is initialized with 0 (lines 2–4). For each time, m nodes are probed and stored in C (Algorithm 17). With small probability ϵ, a state-action pair is selected uniformly at random from C. Otherwise, a state-action pair with the highest action value is selected. Then, a node from the selected pair is taken (lines 10–14). The seed node is activated (line 16) and all activated nodes from the seed node are collected in A_t (line 17). Since some features are based on parent information, the state and action of each activated node z' are defined in increasing order of distance d (line 21). Then the weight vector w is updated based on the states and actions of activated nodes. Among state-action pairs of all child nodes of z', a state-action pair with the highest action-value is selected for the next state-action pair (line 26) and then

it is used for updating weight vector w (line 27). If the activated node is in C, it is removed from C. If not, the expected out-degree of each parent node not in C is incremented by 1 (lines 28–34). This process repeats until the number of seeds reaches the predefined budget k.

The pseudocode of probing nodes is given in Algorithm 17. For each time, we select a node that is not in C that contains all probed nodes (lines 2–6). With small probability ϵ, an inactive node is selected uniformly at random. Otherwise, a node with the highest expected out-degree d_{out} is selected. Then, the selected node is probed and its actual out-degree is obtained (line 7). The expected out-degree of each parent node not in C is incremented by 1. To compute the action value of the probed node, the state and action of the node are defined, the action value is computed and the node is added to corresponding state-action set of C (lines 11–13). This process is repeated m times.

5.4 Related Work

Some studies [40, 4, 99, 17] analyze diffusion patterns on social medias.

Goel et al. [40] investigate the online diffusion structures of seven diverse domains: Yahoo! Kindness, Zync, Secretary Game, Twitter News Stories, Twitter Videos, Friend Sense, and Yahoo! Voice. In spite of the heterogeneity of data, the distribution of diffusion patterns over all seven cases is striking in its similarity. The authors find that in all domains large cascades are not only rare, but even when present they occur within one degree of a few dominant individuals. In particular, less than 10% of adoptions take place in cascades consisting of more than 10 nodes, and less than 10% occur in trees that extend more than two generations (depths) from the seed.

While the previous study focus on the most influential users, Bakshy et al. [4] consider all individuals and study their impact on the spread of information on Twitter. The authors quantify the influence of a given post by the number of users who subsequently repost the URL through the Twitter follower graph and describe the cascades with user attributes and past influence properties of seed users. They find that a small fraction of posted URLs are reposted thousands of times but most posted URLs do not spread at all. The average cascade size is 1.14. The maximum depth of cascades is 9 and most URLs are not reposted at all. It implies that most events do not spread at all and large cascades are rare. They also find that the number of followers is an informative feature, the number of tweets is also a good feature to predict user's activity, and past performance provides the most informative set of features to predict the cascades. Spreading information using the most influential users is the most cost-effective way. However, the authors find that "ordinary influencers" – individuals who exert average, or even less-than-average influence – are under many circumstances more cost-effective.

In [99], the authors propose a method to identify topics using Twitter data by detecting communities in the hashtag co-occurrence network, and to quantify the topical diversity of user interests and content. They verify which user characteristics make people influential by observing several individual properties: number of retweets, number of followers, number of tweets, content interestingness, and di-

versity of interests. They found that high social influence of an individual can be obtained when a user has a big audience group, produces lots of interesting content, and stays focused on a field.

Cheng et al. [17] examine the problem of predicting the growth of cascades on photo-resharing data from Facebook. To describe the growth and spreading of cascades, five classes of features are used: the content, the original poster, the resharer, the graph structure of the cascade, and temporal characteristics of the cascade. The authors find that the set of temporal features outperforms all other individual feature sets but it is still possible to obtain reasonable performance without the temporal features. The features of the content and the original poster become less important as more of the cascade is observed while the importance of temporal features remains relatively stable. They also find that the greater the number of observed reshares, the better the prediction of the growth of a cascade and that breadth, rather than depth in an initial cascade is a better indicator of larger cascades.

The Influence maximization problem has been actively studied in the literature [31, 87, 60, 57, 77, 14]. The problem was first proposed by Domingos and Richardson [31, 87] and formulated as an optimization problem by Kempe et al. [60].

IRIE [57] integrates influence ranking (IR) and influence estimation (IE) methods for the influence maximization problem. The authors use the independent cascade (IC) model and its extension IC-N model as the information diffusion process. The IR method generates a global influence ranking of the nodes and selects the highest ranked node as the seed. However, IR computes the influence for individual nodes. To overcome this shortcoming, the IR method is integrated with a simple influence estimation (IE) method. After one seed is selected, additional influence impact of this seed to each node in the network is computed and then the result is used to adjust next round computation of influence ranking. In experiments, IRIE is compared with PMIA [16], CELF [64], SAEDV [56], Degree, and PageRank on five real-world social networks such as ArXiv, DBLP, Epinions, Slashdot, and LiveJournal. The authors show that IRIE is much more robust and stable both in running time and memory usage than other algorithms.

In influence maximization problems, most algorithms assume that the entire topological structure of a social network is given. However, complete knowledge of the graph structure is typically difficult to obtain. Mihara et al. [77] introduce an influence maximization problem for unknown graphs and propose a heuristic algorithm called IMUG for the problem. They assume that only the set of nodes is known and the set of links is unknown. The topological structure of a graph is partially obtained by probing a node to get a list of its friends. In each round, IMUG probes m nodes with the highest expected degree, selects k seed nodes with the highest expected degree and then triggers influence spread from the selected seed nodes. IMUG is simulated on five real social networks: NetHEPT, DBLP, Amazon, Facebook-small, and Facebook-large. The IC model is used as an influence cascade model. IMUG achieves 60–90% of the influence spread of the algorithms using the entire social network topology even when only 1–10% of the social network topology is known. The authors show that we can achieve a reasonable influence spread even when knowledge of the social network topology is limited and incomplete.

The probability on edges is usually acquired by learning from the real-world data and the obtained estimates always have some inaccuracy comparing to the

true value. The uncertainty in edge probability estimates may affect the performance of the influence maximization task. Chen et al. [14] propose the problem of robust influence maximization to address the impact of uncertainty in edge probability estimates. Because of the uncertainty, the authors consider that the input to the influence maximization task is not edge influence probability on every edge of a social graph, but an interval in which the true probability may lie with high probability. The authors provide the LUGreedy algorithm that solves this problem with a solution-dependent bound. They also study uniform sampling and adaptive sampling methods based on information cascade to effectively reduce the uncertainty on parameters and increase the robustness of the LUGreedy algorithm. The experimental results validate the usefulness of the information cascade based sampling method, and that robustness may be sensitive to the uncertainty of parameter space, i.e., the product of all intervals on all edges.

5.5 Future Work

This work creates several opportunities for future work.

First, our method is not validated with experiments yet. Experimental evaluation is left for future work. It should be based on different diffusion models : the independent cascade (IC) model, the linear threshold (LT) model, etc., and different social networks.

In the classical IM problem, the most typical application is viral marketing that a company promotes a new product in an online social network. The IM problem aims to maximize influence in such a scenario. In our study, we extended to topic-based influence maximization for an unknown graph. We can extend it again to a more realistic environment. In the real world, there is not just one company that wants to promote its product in an online social network. Many companies may competitively use viral marketing on the same social network. In such a case, the problem is extended to maximize influence in a competitive environment, which is called the competitive influence maximization problem [67, 106, 65]. Then, our concern will be how a company effectively maximizes its information influence in a social media when many companies competitively spread their information in the same social media. The competitive IM problem aims to find a strategy against opponents' strategies. Lin et al. [67] propose a reinforcement learning approach for the competitive IM problem. They define the problem with an MDP. The states are defined through some features that represent the current occupation status as well as the condition of the network, for example, number of free (or non-occupied) nodes, summation of degrees of all free nodes, maximum degree among all free nodes, etc. Actions are four strategies called degree-first, max-weight, blocking, sub-greedy. For example, the degree-first strategy chooses high degree nodes as seeds, the max-weight strategy chooses nodes whose overall weights of adjacent edges are maximal, etc. The learning agent chooses a strategy in a given state. In experiments, the method is tested with two scenarios that the opponent's strategy is known and unknown and the effectiveness of their method is shown. An extension of our method in such environment will be an interesting challenge for future work. Multi-agent reinforcement learning will be a good method for the competitive IM

problem to learn the optimal strategy against opponents' strategies.

5.6 Conclusion

In this chapter, we addressed a topic-based influence maximization problem for an unknown graph. Assuming that the graph structure is incomplete or can change dynamically, we probe nodes that may have a big audience group, in order to know a part of the graph structure and discover potentially promising nodes. Then we find the most influential seeds to maximize topic-based influence by using reinforcement learning. Nodes are generalized with some features and we define states and actions based on these features. Action values of nodes are learned from interaction with the environment by reinforcement learning. We then evaluate action values for each probed node and select a node with the highest action value to activate. Experimental evaluation and extension to a more realistic environment, for example the competitive influence maximization problem, are left for future work.

Algorithm 16 Topic-based Influence Maximization for an Unknown Graph

1: $S \leftarrow \emptyset$, $A \leftarrow \emptyset$, $C \leftarrow \emptyset$ // seed set S, active node set A, probed node set C
2: **for** each node $z \in V$ **do**
3: $\quad d_{out}(z) \leftarrow 0$
4: **end for**
5: **for** $t = 1 \ldots k$ **do**
6: $\quad$ // Probe
7: $\quad$ ProbingNodes(C)
8:
9: $\quad$ // Select a seed node
10: $\quad$ **if** With probability ϵ **then**
11: $\quad\quad$ Select a (s, a) pair uniformly at random from C and select a node (z, s, a) from the pair
12: $\quad$ **else**
13: $\quad\quad$ Select a (s, a) pair from C with highest action-value and select a node (z, s, a) from the pair
14: $\quad$ **end if**
15: $\quad$ $S \leftarrow S \cup \{(z, s, a)\}$
16: $\quad$ Activate node z
17: $\quad$ Create $A_t = \{(z', d) : \text{activated node } z' \text{ at time } t, \text{ distance } d \text{ from } z\}$
18: $\quad$ $A \leftarrow A \cup A_t$
19: $\quad$ **for** $d' = 0 \ldots \max\ d$ **do**
20: $\quad\quad$ **for** each activated node $(z', d') \in A_t$ **do**
21: $\quad\quad\quad$ Define state s' and action a' of z'
22: $\quad\quad$ **end for**
23: $\quad$ **end for**
24: $\quad$ **for** each activated node $(z', d) \in A_t$ **do**
25: $\quad\quad$ Get state-action pair (s', a') from z' and observe r
26: $\quad\quad$ $(s'', a'') \leftarrow \underset{(s'',a'') \in \text{State-Action}(out(z'))}{\arg\max} v(s'', a'', w)$
27: $\quad\quad$ $w \leftarrow w + \alpha \left[r + \gamma \hat{q}(s'', a'', w) - \hat{q}(s', a', w)\right] \nabla \hat{q}(s', a', w)$
28: $\quad\quad$ **if** $z' \in C$ **then**
29: $\quad\quad\quad$ Remove (z', d) from C
30: $\quad\quad$ **else**
31: $\quad\quad\quad$ **for** each parent node $p \notin C$ **do**
32: $\quad\quad\quad\quad$ $d_{out}(p) \leftarrow d_{out}(p) + 1$
33: $\quad\quad\quad$ **end for**
34: $\quad\quad$ **end if**
35: $\quad$ **end for**
36: $\quad$ $t \leftarrow t + 1$
37: **end for**
38: return S

Algorithm 17 ProbingNodes

1: **for** $j = 1 \dots m$ **do**
2:     **if** With probability ϵ **then**
3:         Select inactive node $z \notin C$ uniformly at random
4:     **else**
5:         Select inactive node $z = \arg\max_{z \in V} \{d_{out}(z) \mid z \notin C\}$
6:     **end if**
7:     $d_{out}(z) \leftarrow$ actual out-degree of z
8:     **for** each parent node $p \notin C$ **do**
9:         $d_{out}(p) \leftarrow d_{out}(p) + 1$
10:     **end for**
11:     Extract state s and action a from z
12:     Calculate action-value with w
13:     Add (z, s, a) to s of C with the action-value
14: **end for**

Chapter 6

Conclusion

In this chapter, we discuss the limitations of the proposed methods and some possible directions for future work. Then, we close with some remarks.

6.1 Future Work

In this thesis, we applied reinforcement learning to several applications. For these applications, our work creates multiple possible directions for future work.

In the taxi routing problem, we should take into account the non-stationarity of the taxi problem to make policies adapt to environment dynamics. In a real problem, passenger behavior changes over time. It means that goals, transition and reward probabilities of the environment can change over time.

Since reinforcement learning is basically applicable to non-stationary environment, our learning agent can adapt continually to dynamics changes. However, we cannot expect a more responsive adaptation to the changes and when the environment reverts to the previously learned dynamics, the learned knowledge in the past becomes useless. We will need a method that can explicitly address the non-stationarity. Q-learning algorithm we used is fast but it does not consider a model of the environment. It may be better to apply a model-based method and make it detect the environment changes.

In order to adapt flexibly to environment dynamics, the environment model of the taxi problem may need to be divided into partial models that are stored in a library as shown in [19, 79, 89, 52]. For each time, the agent use a partial model that predicts well the environment. If the prediction error of the current model is larger than a threshold, the agent selects another model from the library. If the environment dynamics is completely different from the existing models, it creates a new model. This will be more flexibly adaptable to a non-stationary environment than selecting pre-defined modes by a system designer. In addition, the taxi application we have discussed so far is based on a single agent but it has to be extended to a multi-agent setting. In a multi-agent environment, it will be important for the agent to have an ability to detect non-stationary opponents and learn optimal policies against changed opponent strategies. When opponent strategies are not known a priori, the agent has to adapt to the new environment. Instead of fixed models, the flexible models proposed above will be able to deal with such non-stationary problems.

In the focused crawling problem, the most obvious is to apply our algorithm in larger and various datasets, such as full English Wikipedia and dataset from the site `http://commoncrawl.org/`, etc. In our work, we used a database dump of Simple English Wikipedia provided by the site `https://dumps.wikimedia.org/`. The dataset was sufficiently good to verify the effectiveness of reinforcement learning based crawler but we have to consider a bit larger and more realistic environments.

Another interesting possibility is to build up an efficient mechanism for category selection. Among state and action features, categories related to a target topic are manually pre-selected by a system designer. Since poor feature selection may result in poor performance, it is very important to select appropriate categories for the target topic. The current system relies on human knowledge and intuition about the specific domain. They should be selected in an intelligent and automatic way.

Finally, we can consider multiple crawling agents in order to accelerate crawling performance. One simple method using multiple agents is that all agents are completely independent and they do not share any information including the frontier. In fact, when the agents explore different parts of the environment, they will have distinct scoring policies with respect to their own value functions because they have different experience. An alternative is to share information between agents such as the frontier and scoring policies. In that case, scoring policies have to be merged in some way. A few research works [45, 100, 33] show that combining the different policies outperforms a single agent.

In the influence maximization problem, our method is not validated with experiments yet. Experimental evaluation is left for future work. It should be based on different diffusion models and different social medias.

In our study, we extended the classical IM problem with incomplete knowledge of graph structure and topic-based user's interest. Assuming that the graph structure is incomplete or can change dynamically, we addressed a topic-based influence maximization problem for an unknown graph. We can extend it again to more realistic environment. In the real world, there is not just one company that wants to promote its product in an online social network. Many companies may competitively use viral marketing on the same social network. We call such problem the competitive influence maximization problem [67, 106, 65]. Then, our concern will be how a company effectively maximizes its information influence in a social media when many companies competitively spread their information in the same social media. The competitive IM problem aims to find a strategy against opponent' strategies. Lin et al. [67] propose an reinforcement learning approach for the competitive IM problem. An extension of our method in such environment will be an interesting challenge for future work. Multi-agent reinforcement learning will be a good method for the competitive IM problem to learn the optimal strategy against opponents' strategies.

6.2 Conclusion

In this thesis, we applied reinforcement learning methods to sequential decision making problems in dynamic environments and explored several different reinforcement learning methods such as a model-free method, a model-based method, and a linear function approximation method. There are many other different methods presented

in the literature. We cannot say which algorithm is truly better than others in general. However, we have to choose a right representation of states and actions and a right method for the given problem and its domain because the performance of learning is influenced by the used representation and method.

We tried to use an appropriate method for each application.

For instance, in the taxi routing problem, a tabular based model-free method is used and it is sufficiently good to learn value functions for the problem but we can also use a model-based method when considering the non-stationarity of the problem. However, a function approximation method will not be necessary for this problem because the problem has only position features. If there are many features, a tabular based model-free method is not sufficient to store all state-action pairs and then it needs to extend with an approximate method.

In the focused crawling problem and the influence maximization problem, we used a linear function approximation method. Since the state and action spaces are large, we cannot store all state-action values in tabular forms. Thus, a tabular based model-free method is not used. A model-based method is also difficult to apply to the problems because actions are very noisy.

Even though we select a proper method for a given problem, there may be some things that do not match well with the nature of task, especially if the problem is under slightly different conditions or assumptions. In that case, the selected method has to be adapted to the problem. For example, in the focused crawling problem and the influence maximization problem, the learning algorithms had to be tuned for their tasks.

Another important factor that influences learning performance is how to represent states and actions. In the taxi routing problem, the state and action spaces are clear to define. However, it may not always be clear beforehand which features to use for a given problem if the problem is complex and hard to model in an MDP or if it is difficult to know what characteristics the environment has. For example, in the focused crawling problem and the influence maximization problem, it was not straightforward to select features that represent states and actions.

As we have seen through this thesis, reinforcement learning is a good method to solve a sequential decision making problem in a dynamic environment. It is important to choose a good representation of states and actions and an appropriate method for a given problem.

Bibliography

[1] P. Abbeel, A. Coates, M. Quigley, and A. Y. Ng. An application of reinforcement learning to aerobatic helicopter flight. In B. Schölkopf, J. C. Platt, and T. Hoffman, editors, *Advances in Neural Information Processing Systems 19*, pages 1–8. MIT Press, 2007.

[2] C. C. Aggarwal, F. Al-Garawi, and P. S. Yu. Intelligent crawling on the World Wide Web with arbitrary predicates. In *WWW*, 2001.

[3] G. Almpanidis, C. Kotropoulos, and I. Pitas. Combining text and link analysis for focused crawling. An application for vertical search engines. *Inf. Syst.*, 32(6), 2007.

[4] E. Bakshy, J. M. Hofman, W. A. Mason, and D. J. Watts. Everyone's an influencer: Quantifying influence on twitter. In *Proceedings of the Fourth ACM International Conference on Web Search and Data Mining*, WSDM '11, pages 65–74, New York, NY, USA, 2011. ACM.

[5] A. Baranes and P. Y. Oudeyer. R-iac: Robust intrinsically motivated exploration and active learning. *IEEE Transactions on Autonomous Mental Development*, 1(3):155–169, Oct 2009.

[6] D. Bergmark, C. Lagoze, and A. Sbityakov. Focused crawls, tunneling, and digital libraries. In *ECDL*, 2002.

[7] C. Boutilier, R. Dearden, and M. Goldszmidt. Exploiting structure in policy construction. In *Proceedings of the 14th International Joint Conference on Artificial Intelligence - Volume 2*, IJCAI'95, pages 1104–1111, San Francisco, CA, USA, 1995. Morgan Kaufmann Publishers Inc.

[8] C. Boutilier, R. Dearden, and M. Goldszmidt. Stochastic dynamic programming with factored representations. *Artif. Intell.*, 121(1-2):49–107, Aug. 2000.

[9] R. I. Brafman and M. Tennenholtz. R-max - a general polynomial time algorithm for near-optimal reinforcement learning. *JMLR*, 3, 2003.

[10] S. Chakrabarti, M. van den Berg, and B. Dom. Focused crawling: A new approach to topic-specific web resource discovery. In *WWW*, 1999.

[11] D. Chakraborty and P. Stone. Structure learning in ergodic factored mdps without knowledge of the transition function's in-degree. In L. Getoor and T. Scheffer, editors, *Proceedings of the 28th International Conference on Machine Learning (ICML-11)*, pages 737–744, New York, NY, USA, 2011. ACM.

[12] M. Chau and H. Chen. A machine learning approach to web page filtering using content and structure analysis. *Decis. Support Syst.*, 44(2), 2008.

[13] S. Chen, J. Fan, G. Li, J. Feng, K.-l. Tan, and J. Tang. Online topic-aware influence maximization. *Proc. VLDB Endow.*, 8(6):666–677, Feb. 2015.

[14] W. Chen, T. Lin, Z. Tan, M. Zhao, and X. Zhou. Robust influence maximization. In *Proceedings of the 22Nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '16, pages 795–804, New York, NY, USA, 2016. ACM.

[15] W. Chen, T. Lin, and C. Yang. Real-time topic-aware influence maximization using preprocessing. *Computational Social Networks*, 3(1):8, Nov 2016.

[16] W. Chen, C. Wang, and Y. Wang. Scalable influence maximization for prevalent viral marketing in large-scale social networks. In *Proceedings of the 16th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '10, pages 1029–1038, New York, NY, USA, 2010. ACM.

[17] J. Cheng, L. Adamic, P. A. Dow, J. M. Kleinberg, and J. Leskovec. Can cascades be predicted? In *Proceedings of the 23rd International Conference on World Wide Web*, WWW '14, pages 925–936, New York, NY, USA, 2014. ACM.

[18] S. P. M. Choi, D.-Y. Yeung, and N. L. Zhang. An environment model for nonstationary reinforcement learning. In S. A. Solla, T. K. Leen, and K. Müller, editors, *Advances in Neural Information Processing Systems 12*, pages 987–993. MIT Press, 2000.

[19] B. C. da Silva, E. W. Basso, A. L. C. Bazzan, and P. M. Engel. Dealing with non-stationary environments using context detection. In *Proceedings of the 23rd International Conference on Machine Learning*, ICML '06, pages 217–224, New York, NY, USA, 2006. ACM.

[20] P. Dai and J. Goldsmith. Topological value iteration algorithm for markov decision processes. In *Proceedings of the 20th International Joint Conference on Artifical Intelligence*, IJCAI'07, pages 1860–1865, San Francisco, CA, USA, 2007. Morgan Kaufmann Publishers Inc.

[21] P. Dai and E. A. Hansen. Prioritizing bellman backups without a priority queue. In *Proceedings of the Seventeenth International Conference on International Conference on Automated Planning and Scheduling*, ICAPS'07, pages 113–119. AAAI Press, 2007.

[22] B. D. Davison. Topical locality in the web. In *SIGIR*, 2000.

[23] T. Degris and O. Sigaud. *Factored Markov Decision Processes*, pages 99–126. John Wiley & Sons, Inc., 2013.

[24] T. Degris, O. Sigaud, and P.-H. Wuillemin. Learning the structure of factored markov decision processes in reinforcement learning problems. In *Proceedings*

of the 23rd International Conference on Machine Learning, ICML '06, pages 257–264, New York, NY, USA, 2006. ACM.

[25] J. S. Dibangoye, B. Chaib-draa, and A.-i. Mouaddib. A novel prioritization technique for solving markov decision processes. In *FLAIRS Conference*, pages 537–542, 2008.

[26] T. G. Dietterich. The MAXQ method for hierarchical reinforcement learning. In *ICML*, 1998.

[27] M. Diligenti, F. Coetzee, S. Lawrence, C. L. Giles, and M. Gori. Focused crawling using context graphs. In *VLDB*, 2000.

[28] C. Diuk, L. Li, and B. R. Leffler. The adaptive k-meteorologists problem and its application to structure learning and feature selection in reinforcement learning. In *Proceedings of the 26th Annual International Conference on Machine Learning*, ICML '09, pages 249–256, New York, NY, USA, 2009. ACM.

[29] C. Diuk, A. L. Strehl, and M. L. Littman. A hierarchical approach to efficient reinforcement learning in deterministic domains. In *AAMAS*, 2006.

[30] P. Domingos and G. Hulten. Mining high-speed data streams. In *Proceedings of the Sixth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '00, pages 71–80, New York, NY, USA, 2000. ACM.

[31] P. Domingos and M. Richardson. Mining the network value of customers. In *Proceedings of the Seventh ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '01, pages 57–66, New York, NY, USA, 2001. ACM.

[32] M. Ester, M. Groß, and H.-P. Kriegel. Focused web crawling: A generic framework for specifying the user interest and for adaptive crawling strategies. In *VLDB*, 2001.

[33] S. Faußer and F. Schwenker. Ensemble methods for reinforcement learning with function approximation. In *Proceedings of the 10th International Conference on Multiple Classifier Systems*, MCS'11, pages 56–65, Berlin, Heidelberg, 2011. Springer-Verlag.

[34] M. D. Garcia-Hernandez, J. Ruiz-Pinales, E. Onaindia, J. G. Aviña Cervantes, S. Ledesma-Orozco, E. Alvarado-Mendez, and A. Reyes-Ballesteros. New prioritized value iteration for markov decision processes. *Artif. Intell. Rev.*, 37(2):157–167, Feb. 2012.

[35] A. Geramifard, F. Doshi, J. Redding, N. Roy, and J. P. How. Online discovery of feature dependencies. In *Proceedings of the 28th International Conference on International Conference on Machine Learning*, ICML'11, pages 881–888, USA, 2011. Omnipress.

[36] A. Geramifard, T. J. Walsh, N. Roy, and J. P. How. Batch-ifdd for representation expansion in large mdps. In *Proceedings of the Twenty-Ninth Conference on Uncertainty in Artificial Intelligence*, UAI'13, pages 242–251, Arlington, Virginia, United States, 2013. AUAI Press.

[37] M. Ghavamzadeh and S. Mahadevan. A multiagent reinforcement learning algorithm by dynamically merging Markov decision processes. In *AAMAS*, 2002.

[38] M. Ghavamzadeh and S. Mahadevan. Learning to communicate and act using hierarchical reinforcement learning. In *AAMAS*, 2004.

[39] M. Gjoka, M. Kurant, C. T. Butts, and A. Markopoulou. Walking in facebook: A case study of unbiased sampling of osns. In *Proceedings of the 29th Conference on Information Communications*, INFOCOM'10, pages 2498–2506, Piscataway, NJ, USA, 2010. IEEE Press.

[40] S. Goel, D. J. Watts, and D. G. Goldstein. The structure of online diffusion networks. In *Proceedings of the 13th ACM Conference on Electronic Commerce*, EC '12, pages 623–638, New York, NY, USA, 2012. ACM.

[41] J. Goldenberg, B. Libai, and E. Muller. Talk of the network: A complex systems look at the underlying process of word-of-mouth. *Marketing Letters*, 12(3):211–223, Aug 2001.

[42] G. Gouriten, S. Maniu, and P. Senellart. Scalable, generic, and adaptive systems for focused crawling. In *HyperText*, pages 35–45, 2014.

[43] M. Granovetter. Threshold models of collective behavior. *American Journal of Sociology*, 83(6):1420–1443, 1978.

[44] A. Grigoriadis and G. Paliouras. Focused crawling using temporal difference-learning. In G. A. Vouros and T. Panayiotopoulos, editors, *SETN*, 2004.

[45] M. Grounds and D. Kudenko. Parallel reinforcement learning with linear function approximation. In *Proceedings of the 5th , 6th and 7th European Conference on Adaptive and Learning Agents and Multi-agent Systems: Adaptation and Multi-agent Learning*, ALAMAS'05/ALAMAS'06/ALAMAS'07, pages 60–74, Berlin, Heidelberg, 2008. Springer-Verlag.

[46] M. Grześ and J. Hoey. Efficient planning in r-max. In *The 10th International Conference on Autonomous Agents and Multiagent Systems - Volume 3*, AAMAS '11, pages 963–970, Richland, SC, 2011. International Foundation for Autonomous Agents and Multiagent Systems.

[47] M. Grześ and J. Hoey. On the convergence of techniques that improve value iteration. In *The 2013 International Joint Conference on Neural Networks (IJCNN)*, pages 1–8, Aug 2013.

[48] C. Guestrin, R. Patrascu, and D. Schuurmans. Algorithm-directed exploration for model-based reinforcement learning in factored mdps. In *In Proceedings*

of the International Conference on Machine Learning, pages 235–242. Morgan Kaufmann Publishers Inc, 2002.

[49] J. Guo, P. Zhang, C. Zhou, Y. Cao, and L. Guo. Personalized influence maximization on social networks. In *Proceedings of the 22Nd ACM International Conference on Information & Knowledge Management*, CIKM '13, pages 199–208, New York, NY, USA, 2013. ACM.

[50] M. Han, P. Senellart, S. Bressan, and H. Wu. Routing an autonomous taxi with reinforcement learning. In *Proc. CIKM*, Indianapolis, USA, Oct. 2016. Industry track, short paper.

[51] M. Han, P. Senellart, and P.-H. Wuillemin. Focused crawling through reinforcement learning. In *Proc. ICWE*, June 2018.

[52] P. Hernandez-Leal, M. Taylor, B. Rosman, L. E. Sucar, and E. M. de Cote. Identifying and tracking switching, non-stationary opponents: A bayesian approach, 2016.

[53] T. Hester and P. Stone. Generalized model learning for reinforcement learning in factored domains. In *AAMAS*, 2009.

[54] T. Hester and P. Stone. *Learning and Using Models*, pages 111–141. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

[55] G. Hulten, L. Spencer, and P. Domingos. Mining time-changing data streams. In *Proceedings of the Seventh ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '01, pages 97–106, New York, NY, USA, 2001. ACM.

[56] Q. Jiang, G. Song, G. Cong, Y. Wang, W. Si, and K. Xie. Simulated annealing based influence maximization in social networks. In *Proceedings of the Twenty-Fifth AAAI Conference on Artificial Intelligence*, AAAI'11, pages 127–132. AAAI Press, 2011.

[57] K. Jung, W. Heo, and W. Chen. Irie: Scalable and robust influence maximization in social networks. In *Proceedings of the 2012 IEEE 12th International Conference on Data Mining*, ICDM '12, pages 918–923, Washington, DC, USA, 2012. IEEE Computer Society.

[58] M. Kearns and D. Koller. Efficient reinforcement learning in factored mdps. In *Proceedings of the 16th International Joint Conference on Artificial Intelligence - Volume 2*, IJCAI'99, pages 740–747, San Francisco, CA, USA, 1999. Morgan Kaufmann Publishers Inc.

[59] M. Kearns and S. Singh. Near-optimal reinforcement learning in polynomial time. *Mach. Learn.*, 49(2-3):209–232, Nov. 2002.

[60] D. Kempe, J. Kleinberg, and E. Tardos. Maximizing the spread of influence through a social network. In *Proceedings of the Ninth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '03, pages 137–146, New York, NY, USA, 2003. ACM.

[61] J. M. Kleinberg. Authoritative sources in a hyperlinked environment. *J. ACM*, 46(5), 1999.

[62] J. Z. Kolter and A. Y. Ng. Near-bayesian exploration in polynomial time. In A. P. Danyluk, L. Bottou, and M. L. Littman, editors, *Proceedings of the 26th International Conference on Machine Learning (ICML-09)*, page 65, 2009.

[63] G. Konidaris and A. Barto. Building portable options: Skill transfer in reinforcement learning. In *Proceedings of the 20th International Joint Conference on Artificial Intelligence*, pages 895–900, 2007.

[64] J. Leskovec, A. Krause, C. Guestrin, C. Faloutsos, J. VanBriesen, and N. Glance. Cost-effective outbreak detection in networks. In *Proceedings of the 13th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '07, pages 420–429, New York, NY, USA, 2007. ACM.

[65] H. Li, S. S. Bhowmick, J. Cui, Y. Gao, and J. Ma. Getreal: Towards realistic selection of influence maximization strategies in competitive networks. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, pages 1525–1537, New York, NY, USA, 2015. ACM.

[66] Y. Li, J. Fan, D. Zhang, and K.-L. Tan. Discovering your selling points: Personalized social influential tags exploration. In *Proceedings of the 2017 ACM International Conference on Management of Data*, SIGMOD '17, pages 619–634, New York, NY, USA, 2017. ACM.

[67] S.-C. Lin, S.-D. Lin, and M.-S. Chen. A learning-based framework to handle multi-round multi-party influence maximization on social networks. In *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '15, pages 695–704, New York, NY, USA, 2015. ACM.

[68] M. Lopes, T. Lang, M. Toussaint, and P. yves Oudeyer. Exploration in model-based reinforcement learning by empirically estimating learning progress. In F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 25*, pages 206–214. Curran Associates, Inc., 2012.

[69] M. Lopes and P. Y. Oudeyer. The strategic student approach for life-long exploration and learning. In *2012 IEEE International Conference on Development and Learning and Epigenetic Robotics (ICDL)*, pages 1–8, Nov 2012.

[70] A. S. Maiya and T. Y. Berger-Wolf. Benefits of bias: Towards better characterization of network sampling. In *Proceedings of the 17th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '11, pages 105–113, New York, NY, USA, 2011. ACM.

[71] H. B. McMahan and G. J. Gordon. Fast exact planning in markov decision processes. In *ICAPS*, pages 151–160, 2005.

[72] F. Menczer. Lexical and semantic clustering by web links. *J. Am. Soc. Inf. Sci. Technol.*, 55(14), 2004.

[73] F. Menczer. Web crawling. In B. Liu, editor, *Web Data Mining: Exploring Hyperlink, Content and Usage Data*. Springer, 2007.

[74] F. Menczer and R. K. Belew. Adaptive retrieval agents: Internalizing local contextand scaling up to the web. *Mach. Learn.*, 39(2-3), 2000.

[75] F. Menczer, G. Pant, and P. Srinivasan. Topical web crawlers: Evaluating adaptive algorithms. *ACM Trans. Internet Technol.*, 4(4), 2004.

[76] R. Meusel, P. Mika, and R. Blanco. Focused crawling for structured data. In *CIKM*, 2014.

[77] S. Mihara, S. Tsugawa, and H. Ohsaki. Influence maximization problem for unknown social networks. In *Proceedings of the 2015 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining 2015*, ASONAM '15, pages 1539–1546, New York, NY, USA, 2015. ACM.

[78] A. W. Moore and C. G. Atkeson. Prioritized sweeping: Reinforcement learning with less data and less time. *Mach. Learn.*, 13(1):103–130, Oct. 1993.

[79] T. T. Nguyen, T. Silander, and T.-Y. Leong. Transferring expectations in model-based reinforcement learning. In *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 2*, NIPS'12, pages 2555–2563, USA, 2012. Curran Associates Inc.

[80] L. Page, S. Brin, R. Motwani, and T. Winograd. The PageRank citation ranking: Bringing order to the web. Technical Report 1999-66, Stanford InfoLab, 1999.

[81] G. Pant, P. Srinivasan, and F. Menczer. Exploration versus exploitation in topic driven crawlers. In *WWW Workshop on Web Dynamics*, 2002.

[82] R. Parr, C. Painter-Wakefield, L. Li, and M. Littman. Analyzing feature generation for value-function approximation. In *Proceedings of the 24th International Conference on Machine Learning*, ICML '07, pages 737–744, New York, NY, USA, 2007. ACM.

[83] J. Peng and R. J. Williams. Efficient learning and planning within the dyna framework. *Adapt. Behav.*, 1(4):437–454, Apr. 1993.

[84] M. Qu, H. Zhu, J. Liu, G. Liu, and H. Xiong. A cost-effective recommender system for taxi drivers. In *KDD*, 2014.

[85] R. Rana and F. S. Oliveira. Real-time dynamic pricing in a non-stationary environment using model-free reinforcement learning. *Omega*, 47:116 – 126, 2014.

[86] J. Rennie and A. McCallum. Using reinforcement learning to spider the web efficiently. In *ICML*, 1999.

[87] M. Richardson and P. Domingos. Mining knowledge-sharing sites for viral marketing. In *Proceedings of the Eighth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '02, pages 61–70, New York, NY, USA, 2002. ACM.

[88] M. Riedmiller, T. Gabel, R. Hafner, and S. Lange. Reinforcement learning for robot soccer. *Autonomous Robots*, 27(1):55–73, Jul 2009.

[89] B. Rosman, M. Hawasly, and S. Ramamoorthy. Bayesian policy reuse. *Machine Learning*, 104(1):99–127, Jul 2016.

[90] T. C. Schelling. *Micromotives and macrobehavior*. WW Norton & Company, 2006.

[91] S. Singh and D. Bertsekas. Reinforcement learning for dynamic channel allocation in cellular telephone systems. In *Proceedings of the 9th International Conference on Neural Information Processing Systems*, NIPS'96, pages 974–980, Cambridge, MA, USA, 1996. MIT Press.

[92] A. L. Strehl, C. Diuk, and M. L. Littman. Efficient structure learning in factored-state mdps. In *AAAI*, 2007.

[93] R. S. Sutton. Integrated architecture for learning, planning, and reacting based on approximating dynamic programming. In *Proceedings of the Seventh International Conference (1990) on Machine Learning*, pages 216–224, San Francisco, CA, USA, 1990. Morgan Kaufmann Publishers Inc.

[94] R. S. Sutton and A. G. Barto. *Introduction to Reinforcement Learning*. MIT Press, 1998.

[95] G. Tesauro, R. Das, H. Chan, J. Kephart, D. Levine, F. Rawson, and C. Lefurgy. Managing power consumption and performance of computing systems using reinforcement learning. In J. C. Platt, D. Koller, Y. Singer, and S. T. Roweis, editors, *Advances in Neural Information Processing Systems 20*, pages 1497–1504. Curran Associates, Inc., 2008.

[96] P. E. Utgoff, N. C. Berkman, and J. A. Clouse. Decision tree induction based on efficient tree restructuring. *Mach. Learn.*, 29(1):5–44, Oct. 1997.

[97] C. J. C. H. Watkins. *Learning from Delayed Rewards*. PhD thesis, King's College, Cambridge, UK, May 1989.

[98] C. J. C. H. Watkins and P. Dayan. Technical note: Q-learning. *Mach. Learn.*, 8(3-4), 1992.

[99] L. Weng and F. Menczer. Topicality and impact in social media: Diverse messages, focused messengers. *PLOS ONE*, 10(2):1–17, 02 2015.

[100] M. A. Wiering and H. van Hasselt. Ensemble algorithms in reinforcement learning. *Trans. Sys. Man Cyber. Part B*, 38(4):930–936, Aug. 2008.

[101] D. Wingate and K. D. Seppi. P3vi: A partitioned, prioritized, parallel value iterator. In *Proceedings of the Twenty-first International Conference on Machine Learning*, ICML '04, pages 109–, New York, NY, USA, 2004. ACM.

[102] D. Wingate and K. D. Seppi. Prioritization methods for accelerating mdp solvers. *J. Mach. Learn. Res.*, 6:851–881, Dec. 2005.

[103] J. Yuan, Y. Zheng, X. Xie, and G. Sun. T-drive: Enhancing driving directions with taxi drivers' intelligence. *TKDE*, 25(1), 2013.

[104] N. J. Yuan, Y. Zheng, L. Zhang, and X. Xie. T-finder: A recommender system for finding passengers and vacant taxis. *TKDE*, 25(10), 2013.

[105] Y. Zheng. Trajectory data mining: An overview. *ACM Trans. Intell. Syst. Technol.*, 6(3), 2015.

[106] Y. Zhu, D. Li, and Z. Zhang. Minimum cost seed set for competitive social influence. In *IEEE INFOCOM 2016 - The 35th Annual IEEE International Conference on Computer Communications*, pages 1–9, April 2016.

[107] H. Zhuang, Y. Sun, J. Tang, J. Zhang, and X. Sun. Influence maximization in dynamic social networks. In *2013 IEEE 13th International Conference on Data Mining*, pages 1313–1318, Dec 2013.

Appendices

Annexe A

Résumé en français

Dans cette thèse, nous appliquons l'apprentissage par renforcement à des problèmes de décision séquentiels dans des environnements dynamiques.

A.1 Introduction

L'apprentissage par renforcement [94] est basé sur l'idée d'apprentissage par essais et erreurs et il a été couramment utilisé en robotique, avec des applications telles que les robots jouant au football [88], les robots hélicoptères [1], etc.

Il a également été utilisé dans diverses applications qui concernent des problèmes de décision séquentiels dans des environnements dynamiques tels que la gestion de l'énergie [95], l'allocation de canaux [91], le contrôle des feux de signalisation [19], etc. La gestion de l'énergie dans les centres de données est une préoccupation croissante en matière économique et environnementale. Dans [95], une approche d'apprentissage par renforcement est présentée pour apprendre des politiques efficaces de gestion à la fois de la performance et de la consommation d'énergie dans les serveurs d'applications Web. Dans les systèmes de téléphonie cellulaire, un problème important consiste à allouer dynamiquement les canaux de communication pour maximiser le service fourni aux appelants mobiles. Ce problème est abordé dans [91] en utilisant une méthode d'apprentissage par renforcement pour allouer les canaux disponibles aux appels afin de minimiser le nombre d'appels bloqués et le nombre d'appels qui sont abandonné quand ils sont remis à un appel occupé. Une méthode d'apprentissage par renforcement est également appliquée au problème de contrôle des feux de circulation [19] qui ajuste le signal en fonction du trafic en temps réel afin de réduire la congestion du trafic. L'agent apprend une politique de contrôle des feux de circulation dans laquelle les véhicules n'attendent pas trop longtemps pour traverser l'intersection.

Ces problèmes ont des objectifs explicites à atteindre et nécessitent de prendre une décision pour un environnement donné afin d'atteindre les objectifs. Les environnements changent en réaction à certains comportements de contrôle. Cependant, il est difficile de concevoir des politiques optimales à l'avance parce que les modèles d'environnement ne sont pas disponible. Dans de tels problèmes, l'apprentissage par renforcement peut être utilisé pour trouver les politiques optimales. Il apprend les politiques en interagissant avec l'environnement afin d'atteindre un objectif. Les po-

litiques apprises prennent en compte les conséquences à long terme des décisions individuelles.

Dans cette thèse, nous résolvons plusieurs problèmes de décision séquentiels en utilisant des méthodes d'apprentissage par renforcement. Par exemple, dans un problème de parcours du Web ciblé, un agent doit collecter autant de pages Web que possible, pertinentes pour un sujet prédéfini, tout en évitant les pages non pertinentes. De nombreuses méthodes d'exploration utilisent la classification pour les liens non visités afin d'estimer si les liens pointent vers des liens pertinents, mais ces méthodes ne prennent pas en compte les effets à long terme de la sélection d'un lien. Dans le problème de maximisation de l'influence, l'agent vise à choisir les graines les plus influentes pour maximiser l'influence sous un certain modèle de diffusion de l'information. Le problème prend déjà en compte les valeurs à long terme mais pas nécessairement la dimension de planification que l'apprentissage par renforcement introduit.

Pour résoudre de tels problèmes de décision séquentiels, nous formulons d'abord les problèmes en tant que processus de décision de Markov (MDP), une formulation générale de l'apprentissage par renforcement. Ensuite, nous résolvons ces problèmes en utilisant des méthodes d'apprentissage de renforcement appropriées aux problèmes correspondants et démontrons que les méthodes d'apprentissage par renforcement trouvent des politiques stochastiques pour chaque problème qui sont proches de l'optimale.

A.2 Apprentissage par renforcement

L'apprentissage par renforcement est similaire à la façon d'apprendre des humains et des animaux. En fait, de nombreux algorithmes d'apprentissage par renforcement sont inspirés par les systèmes d'apprentissage biologiques [94].

Dans l'apprentissage par renforcement, un agent apprend de l'interaction continue avec un environnement afin d'atteindre un objectif. Une telle interaction produit beaucoup d'informations sur les conséquences de son comportement, ce qui aide à améliorer ses performances. Chaque fois que l'agent d'apprentissage fait une action, l'environnement répond à son action en donnant une récompense et en présentant un nouvel état. L'objectif de l'agent est de maximiser la quantité totale de récompense qu'il reçoit. Grâce à l'expérience dans son environnement, il découvre quelles actions produisent de manière stochastique la plus grande récompense et utilise une telle expérience pour améliorer sa performance pour des essais ultérieurs. Autrement dit, l'agent apprend comment se comporter afin d'atteindre ses objectifs. Dans l'apprentissage par renforcement, tous les agents ont des objectifs explicites et apprennent des décisions en interagissant avec leur environnement afin d'atteindre les objectifs.

L'apprentissage par renforcement vise à apprendre comment il est bon pour l'agent d'être dans un état sur le long terme, caractérisé par une *valeur d'état*, ou comment il est bon de prendre une action dans un état donné sur le long terme, caractérisé par une *valeur d'action*. Une récompense est donnée immédiatement par un environnement en réponse à l'action de l'agent et un agent d'apprentissage utilise la récompense pour évaluer la valeur d'un état ou d'une action. La meilleure action est sélectionnée par les valeurs d'états ou d'actions car la valeur la plus élevée

apporte la plus grande récompense à long terme. Ensuite, l'agent d'apprentissage peut maximiser la récompense cumulative qu'il reçoit.

Un modèle représente la dynamique de l'environnement. Un agent d'apprentissage apprend des fonctions de valeur avec ou sans modèle. Quand un algorithme d'apprentissage par renforcement construit un modèle de l'environnement et apprend des fonctions de valeur à partir du modèle, elle est appelée une *méthode basée sur un modèle*. Les algorithmes d'apprentissage par renforcement peuvent apprendre des fonctions de valeur directement à partir des expérience sans modèles d'environnement. Si un algorithme apprend des valeurs des états ou des actions par essais et erreurs sans modèle, nous l'appelons une *méthode sans modèle*. Comme un modèle imite le comportement de l'environnement, il permet d'estimer comment les environnements vont changer en réponse à ce que fait l'agent. Cependant, l'apprentissage d'un modèle complet et précis nécessite un calcul plus complexe que les méthodes sans modèle. Nous étudions une méthode sans modèle et une méthode basée sur un modèle.

Ces fonctions de valeur peuvent être représentées en utilisant des formes tabulaires mais, pour des problèmes complexes, les formes tabulaires ne peuvent pas stocker efficacement toute les valeurs des fonctions. Dans ce cas, les fonctions doivent être approximées en utilisant une représentation de fonction paramétrée pour les problèmes importants. Nous étudions un problème de parcours du Web ciblé et un problème de maximisation de l'influence en utilisant une méthode d'approximation de fonction.

A.3 Méthodes sans modèle et basées sur des modèles

Nous étudions deux approches principales pour résoudre les problèmes d'apprentissage par renforcement : les méthodes sans modèle et celles basées sur des modèles.

A.3.1 Apprendre sans modèles

Dans cette sous-partie, nous étudions une méthode sans modèle qui apprend directement des expériences sans modèle. L'une des méthodes sans modèle largement utilisées est le Q-apprentissage [98]. Nous présentons un algorithme de Q-apprentissage [98] avec une stratégie d'exploration et d'exploitation personnalisée pour résoudre un vrai problème de routage de taxi.

Nous supposons qu'un agent de taxi autonome ne connaît pas la ville et que la voiture se déplace complètement en fonction des valeurs d'action estimées de l'apprentissage par renforcement. Le but de cette application est que le taxi autonome décide où aller pour prendre un passager en apprenant à la fois les valeurs des actions étant donné un état et la probabilité d'existence des passagers.

L'agent d'apprentissage prend une action a dans l'état s, reçoit une récompense r, et passe à l'état suivant s'. Avec le Q-apprentissage, la valeur estimée de la prise d'action a dans l'état s, notée $Q(s, a)$, est mise à jour comme suit :

$$Q(s,a) \doteq Q(s,a) + \alpha \left[r + \gamma \max_{a'} Q(s',a') - Q(s,a) \right]. \tag{A.1}$$

Algorithm 18 Routage de taxis pour l'apprentissage des points de ramassage

1: Initialise $Q(s, a)$, probabilité d'existence des passagers p
2: **repeat**
3: **repeat**
4: **if** glouton **then**
5: $V \doteq \{ a \in A \mid Q(s, a) \geq \max_{a'} Q(s, a') - \eta \}$
6: **if** $|V| > 1$ **then**
7: Sélectionner l'action a avec la plus haute probabilité p
8: **end if**
9: **else** /* pas glouton */
10: Sélectionner l'action a uniformément au hasard
11: **end if**
12: Prendre l'action a, obtenir la récompense r, observer l'état suivant s'
13: $Q(s, a) \doteq Q(s, a) + \alpha \left[r + \gamma \max_{a'} Q(s', a') - Q(s, a) \right]$
14: Incrémenter le compteur des visites de s'
15: Mise à jour de la probabilité d'existence des passagers $p(s')$
16: **if** passager trouvé en s' **then**
17: Incrémenter le compteur des passagers trouvés en s'
18: s devient la fin de la route des passagers à partir de s'
19: **else**
20: $s \doteq s'$
21: **end if**
22: **until** un passager est trouvé
23: **until** l'algorithme converge

Nous appelons un *épisode* une série d'étapes jusqu'à ce que l'agent trouve un passager. Pour le premier épisode, le taxi situé à une position aléatoire se déplace en fonction de sa politique. L'épisode se termine lorsque le taxi trouve un passager. Ensuite, il se déplace vers le destination du passager et commence un nouvel épisode. Lorsque le taxi se déplace, il reçoit des récompense et met à jour sa valeur d'action et la probabilité d'existence. Le réseau routier est discrétisé et les mouvements correspondent à des étapes du réseau discrétisé. À chaque étape, le taxi apprend où les passagers sont susceptibles d'être localisés.

L'algorithme de routage de taxi pour l'apprentissage des points de ramassage est décrit dans l'Algorithme 18. Selon la politique ϵ-gloutonne, l'action a est sélectionnée dans un état donné s.

La règle de sélection d'action sélectionne l'action avec la valeur d'action estimée maximale (action gloutonne). Cependant, avec cette règle, l'algorithme ignore les autres actions qui, bien qu'elles aient une valeur légèrement inférieure, peuvent conduire à un état ayant une plus grande chance de prendre un passager. Par conséquent, au lieu de sélectionner une action gloutonne, nous relaxons la condition de sélection en plaçant une borne inférieure en dessous de la valeur maximale afin de choisir parmi les actions candidates potentiellement plus intéressantes. Les actions candidates sont comparées aux probabilités d'existence des passagers dans leurs états correspondants. Nous nous référons à l'algorithme avec cette stratégie de sélection comme *Q-apprentissage utilisant LB/Proba*. Cette comparaison de probabilité est

très efficace lorsque les actions partagent la même valeur ($Q(s, a_1) = \ldots = Q(s, a_n)$). Dans ce cas, nous choisissons au départ une action au hasard parce que nous considérons que toutes les actions sont identiques. En fait, elles peuvent ne pas avoir la même valeur si elles conduisent à un état avec une très forte probabilité d'existence des passagers. La comparaison des probabilités réduit ce genre d'erreur.

Après avoir effectué une action, nous mettons à jour la valeur de l'action dans l'état actuel s avec récompense r et l'état suivant s'. Lorsque nous visitons un nouvel état s', les compteurs des nombres de visites et passagers trouvés sont incrémentés et la probabilité d'existence des passagers est également recalculée. Nous répétons cette procédure jusqu'à ce que nous trouvions un passager.

A.3.2 Apprentissage de modèles

Les méthodes basées sur des modèles apprennent les modèles de transition et de récompense et utilisent ces modèles pour mettre à jour les fonctions de valeur.

La plupart des méthodes basées sur des modèles sont basées sur des modèles de transition sous la forme de réseaux bayésiens dynamiques (DBN) où la transition de chaque entité est supposée indépendante de celle des autres [7, 8, 23, 54]. Dans cette sous-partie, nous étudions une méthode basée sur un modèle. En particulier, nous abordons le problème d'un MDP factorisé [7, 8, 23] dont l'état est représenté par un vecteur de n variables.

Nous présentons un algorithme pour apprendre la structure des fonctions de transition DBN avec des arcs synchroniques, représentés dans l'Algorithme 19. Nous utilisons des arbres de décision pour représenter les fonctions de transition au lieu des représentations tabulaires.

Semblable à R-max [9], toutes les valeurs d'action d'état inconnues sont initialisées par une constante *Rmax* afin d'encourager l'agent à explorer. A chaque fois, l'agent prend une action gloutonne.

Pour construire des arbres de décision, les actions doivent être visitées suffisamment souvent. Nous définissons un paramètre prédéfini m, le nombre minimum de visites nécessaires aux actions inconnues, pour décider si des actions sont connues ou non. Chaque fois qu'une action est prise, le nombre de visites de l'action est incrémenté. Si le nombre de visites pour une action est égal à m, des arbres de décision pour l'action sont créés.

Généralement, étant donné une action, chaque facteur $s'(i)$ a son propre arbre de décision servant à estimer $\Pr(s'(i) \mid \cdot, a)$, c'est-à-dire qu'un arbre de décision représente $\Pr(s'(i) \mid \cdot, a)$. Nous réduisons le nombre d'arbres de décision en choisissant des facteurs pertinents dont les valeurs sont constamment changées chaque fois que l'action est prise. Les fonctions de transition des facteurs non modifiés sont des fonctions d'identité. Puisque les arbres de décision de ces facteurs non modifiés n'affectent pas l'estimation de la transition de l'état s to s', nous ne créons pas leurs arbres de décision. Nous collectons tous les facteurs à valeur modifiée dans F_a chaque fois que l'action a est choisie. Ensuite, pour chaque facteur $s'(i)$ dans F_a, nous créons un arbre de décision des DBN.

LearnTransitionFunction estime $\Pr(s'(i) \mid s, a)$ depuis l'arbre de décision correspondant et met à jour l'arbre avec s et $s'(i)$. La valeur d'action est calculée avec les

fonctions de transition obtenues. S'il y a un facteur qui n'est pas encore prévisible, nous mettons à jour la valeur de l'action avec *Rmax* pour en apprendre plus sur la valeur de l'action d'état.

Algorithm 19 Apprendre la structure du DBN avec des arcs synchroniques

1: Entrée : valeur d'action initiale *Rmax*, nombre de visites minimum sur l'action m
2: // Initialisation
3: $\forall a \in A, \forall s \in S, \quad Q(s,a) \leftarrow Rmax$
4: **repeat**
5:     **repeat**
6:         $a \leftarrow argmax_{a' \in A} Q(s,a')$
7:         Exécuter a, obtenir la récompense r et observer l'état suivant s'
8:         **if** $C(a) < m$ **then**
9:             $C(a) \leftarrow C(a) + 1$
10:             $F_a \leftarrow RecordChangedFactors(a)$
11:         **else**
12:             **if** $C(a) == m$ **then**
13:                 $BuildDecisionTrees(a)$
14:             **end if**
15:             // Estimer la fonction de transition
16:             **for** chaque facteur $s'(i)$ **do**
17:                 $\Pr(s'(i) \mid s,a) \leftarrow LearnTransitionFunction(s,a,s'(i))$
18:             **end for**
19:             // Mettre à jour les valeurs d'action
20:             **if** $\exists i, \Pr_i(s'(i) \mid s,a) = \perp$ **then**
21:                 $Q(s,a) \leftarrow Rmax$
22:             **else**
23:                 $Q(s,a) \leftarrow R(s,a) + \gamma \sum_{s'} \Pr(s' \mid s,a) max_{a' \in A} Q(s',a')$
24:             **end if**
25:         **end if**
26:         $s \leftarrow s'$
27:     **until** l'état terminal est atteint
28: **until** l'algorithme converge

BuildDecisionTrees est représenté dans l'Algorithme 20. Pour construire des arbres de décision, nous sélectionnons d'abord les facteurs parents qui seront utilisés comme nœuds de l'arbre de décision. Pour chaque facteur $s'(i)$ de F_a, $FindParents$ applique le test du χ^2 à tous les autres facteurs de temps t et à tous les autres facteurs de F_a à l'instant $t+1$ pour trouver ses parents, $Parent(s'(i))$, comme dans l'équation (3.4). par est l'ensemble de ses parents à l'instant t et par_{sync} est ses parents à l'instant $t+1$. Cependant, il est difficile de conclure que par_{sync} est un ensemble de parents de facteur $s'(i)$ parce que le test du χ^2 détermine s'il existe une relation significative entre deux variables mais il ne détermine pas lequel cause l'autre. Pour décider lequel est un parent, on prédéfinit l'ordre des facteurs. Supposons qu'il y ait deux facteurs x_i et x_j qui sont reliés l'un à l'autre au temps $t+1$ par le test du χ^2. Si x_i précède x_j dans l'ordre, alors nous considérons x_j comme un parent synchronique de x_i. Pour la fonction x_i, nous plaçons ses parents potentiels

après x_i. Tous les facteurs suivants sont candidates comme parents synchroniques. À l'aide de cette vérification, $FindRealSyncParents(i)$ détermine quels sont les vrais parents synchroniques et renvoie par_{sync}. $CreateDecisionTree$ construit un arbre de décision dont les nœuds sont des éléments de par et par_{sync}. Dans notre algorithme, nous utilisons HoeffdingTree [30, 55] qui est un algorithme d'induction d'arbre de décision incrémentielle capable d'apprendre à partir de flux de données massifs[1].

Algorithm 20 BuildDecisionTrees

1: Entrée : action a
2: **for** chaque facteur $s'(i)$ de F_a **do**
3: $(par, par_{sync}) \leftarrow FindParents(i)$
4: $par_{sync} \leftarrow FindRealSyncParents(i)$
5: $CreateDecisionTree(par, par_{sync})$
6: **end for**

A.4 Parcours du Web ciblé

Le but de parcours du Web ciblé est de collecter autant de pages Web que possible sur le sujet ciblé tout en évitant les pages non pertinentes car le robot d'exploration est supposé avoir des ressources limitées, telles que le trafic réseau ou le temps d'analyse. Ainsi, dans une séquence d'exploration, la sélection de lien ne doit pas être un choix aléatoire.

Pour atteindre l'objectif d'exploration, à partir d'une page, l'agent sélectionne le lien le plus prometteur susceptible de conduire à une page pertinente. Même si une page liée semble moins pertinent pour le sujet ciblé, si elle peut potentiellement mener à une page pertinente à long terme, il pourrait être utile de la sélectionner. À chaque pas de temps, l'agent doit estimer quel hyperlien peut mener à une page pertinente. Ce sera un facteur clé de succès dans le parcours du Web ciblé si l'agent a la capacité d'estimer quel hyperlien est le plus rentable à long terme.

L'apprentissage par renforcement trouve une action optimale dans un état donné qui produit la récompense totale la plus élevée à long terme par interaction répétée avec l'environnement. Avec l'apprentissage par renforcement, la valeur estimée optimale des hyperliens (actions) est apprise au fur et à mesure que les pages (états) sont visitées. L'agent peut évaluer si une sélection de lien peut donner une récompense optimale à long terme et sélectionne le lien le plus prometteur basé sur l'estimation. Comme la plupart des robots ciblés, nous supposons que les pages avec des sujets similaires sont proches l'un à l'autre. Notre stratégie de parcours est basée sur la localité des sujets et la technique de tunnellisation. Nous supposons également que toute la structure du graphe Web n'est pas connu de l'agent de parcours à l'avance.

A.4.1 MDP pour le parcours ciblé du Web

Pour modéliser l'environnement d'exploration dans un MDP $M = \langle S, A, R, T \rangle$, nous définissons les pages Web comme des états S et les hyperliens directs d'une page

[1] `http://weka.sourceforge.net/doc.dev/weka/classifiers/trees/HoeffdingTree.html`

comme des actions A. Lorsque l'agent d'exploration suit un lien hypertexte de la page en cours, une transition de la page actuelle à la page liée se produit et la pertinence pour le sujet cible est calculée pour la page liée pour évaluer si le lien hypertexte sélectionné conduit à une page pertinente pour le sujet cible ou non. La fonction de transition T est la probabilité de transition de la page en cours à la page liée lors de la prise du lien hypertexte. La récompense $r \in R$ est une valeur de pertinence de la page liée au sujet donné. Pour la prochaine étape d'exploration, l'agent sélectionne un lien hypertexte avec la valeur d'estimation la plus élevée de la page nouvellement visitée, et ainsi de suite.

Comme nous l'avons mentionné ci-dessus, nous définissons les pages Web comme des états et des hyperliens directs d'une page en tant qu'actions. Cependant, les pages Web sont toutes différentes, il y a énormément de pages sur le Web, et elles sont liées entre elles comme les fils d'une toile d'araignée. Si chaque page Web est définie comme un état et chaque lien hypertexte direct comme une action, cela rend l'apprentissage d'une politique intraitable en raison de l'immense nombre de paires d'états–actions. De plus, dans l'apprentissage par renforcement, les valeurs d'action optimales sont dérivées après avoir visité chaque paire d'états–actions infiniment souvent. Il n'est pas nécessaire pour un robot de visiter plusieurs fois la même page. Ainsi, nos MDP ne peuvent pas être modélisés directement à partir d'un graphe Web. Au lieu de cela, nous devons généraliser les pages et les liens en fonction de certains facteurs qui représentent les pages Web et la sélection du lien suivant.

États. Une page Web est abstraite par certains facteurs afin de définir un état. Les facteurs d'un état consistent en deux types d'informations. Le premier type sont des informations sur la page elle-même. Le second sont des informations concernant les pages environnantes. La pertinence des pages par rapport au sujet cible et à certaines catégories sont les informations des pages de premier type. Le changement de pertinence, la pertinence moyenne des pages parents, la distance par rapport à la dernière page représentent la relation avec les pages entourant la page en cours. Afin d'obtenir correctement les informations de relation, chaque lien non visité doit conserver les liens parents. L'agent d'exploration est supposé ne pas connaître à l'avance le graphe Web complet, de sorte que chaque lien ne sait pas initialement combien de parents ils possèdent, mais les informations parentales sont progressivement mises à jour au fur et à mesure que les pages sont explorées. Lorsqu'une page est visitée, l'URL de la page en cours est ajoutée à tous les liens externes de la page en tant que parent. Chaque lien a au moins un lien parent. Si un lien a beaucoup de parents, cela signifie que le lien est référencé par plusieurs pages.

- **Pertinence du sujet cible**
- **Changement de pertinence du sujet cible**
- **Pertinence des catégories**
- **Pertinence moyenne de tous les parents**
- **Pertinence moyenne des parents pertinents**
- **Distance de la dernière page pertinente**

Actions. Pour définir des actions, tous les liens hypertexte d'une page Web sont également extraits avec certaines facteurs de la même manière que les pages. La pertinence à propos du sujet ciblé et de certaines catégories est utilisée pour prédire la pertinence de la page vers laquelle pointe un lien hypertexte. À la différence des pages, les hyperliens n'ont pas suffisamment d'informations pour calculer les valeurs. Ainsi, le texte de l'URL, le texte d'ancrage et le texte environnant d'un lien hypertexte sont utilisés pour la calculer. Ici, la pertinence n'est pas une vraie pertinence mais une prédiction car il n'est pas possible de savoir quelle page sera pointée par un lien hypertexte avant de suivre le lien. Afin de soutenir la prédiction de la pertinence, la pertinence moyenne des pages parent est également utilisée comme un élément qui représentent la relation avec les pages entourant le lien. Chaque lien hypertexte a au moins un parent. Si le lien est référencé par plusieurs pages, il peut avoir plusieurs parents. Comme mentionné ci-dessus, les informations parentes sont progressivement mises à jour au fur et à mesure que les pages sont explorées et chaque lien non visité conserve les liens parents. Ensuite, l'information parente est utilisée pour calculer la pertinence moyenne des pages parent. Les facteurs pour l'action sont un sous-ensemble de ceux des états, à savoir :

- **Pertinence du sujet cible**
- **Pertinence des catégories**
- **Pertinence moyenne de tous les parents**
- **Pertinence moyenne des parents pertinents**

A.4.2 MDP avec priorités de mises à jour

Dans un parcours du Web ciblé, l'agent visite une page Web et extrait tous les liens hypertexte de la page. Les hyperliens sont ajoutés à la file d'attente prioritaire, appelée frontière. Un lien avec la plus haute priorité est sélectionné de la frontière pour la prochaine visite. La frontière joue un rôle crucial dans le processus d'exploration. L'agent peut prendre la vue large de la limite du graphe parcouru, ne se concentrant pas sur une zone spécifique de l'ensemble du graphique crawlé. Les URL non visitées sont conservées à la frontière avec score de priorité et donc, pour chaque itération, le lien le plus prometteur être sélectionné de la limite du graphe exploré. Ainsi, le robot d'indexation Web peut sélectionner systématiquement le meilleur lien, quelle que soit sa position actuelle.

Nous utilisons une méthode d'apprentissage par renforcement de différence temporelle (DT) pour que les agents de parcours apprennent de bonnes politiques de manière incrémentielle, en ligne, comme le font les agents d'analyse. Dans la plupart des méthodes DT, chaque itération des mises à jour de valeur est basée sur un épisode, une séquence de transitions d'état d'un état de départ à l'état terminal. Par exemple, à l'instant t, dans l'état s, l'agent prend une action a en fonction de sa politique, ce qui entraîne une transition vers l'état s'. A l'instant $t+1$ dans l'état successeur de s, l'état s', l'agent prend sa meilleure action a' suivie d'une transition vers l'état s'' et ainsi de suite jusqu'à l'état terminal. En parcourant le Web, si l'agent continue d'avancer en suivant les transitions d'états successives, il

peut tomber dans des pièges de parcours ou des optima locaux. C'est la raison pour laquelle une frontière est utilisée de manière importante en parcourant le Web. Il est nécessaire d'apprendre les fonctions de valeur de la même manière que les tâches d'exploration.

Pour garder l'idée principale des tâches d'exploration, nous modélisons l'apprentissage de notre agent d'exploration avec la priorité de l'ordre des mises à jour qui est l'une des méthodes d'itération de valeur pour propager les valeurs de manière efficace. Avec une méthode de mise à jour par ordre de priorité, l'agent d'exploration ne suit plus l'ordre successif des transitions d'état. Chaque paire d'états–actions est ajoutée à frontière avec sa valeur d'action estimée. À chaque fois, il sélectionne la paire d'états–actions la plus prometteuse parmi toutes les paires, comme le fait l'agent d'exploration traditionnel.

A.4.3 Approximation par fonctions linéaires avec priorité des mises à jour

La frontière du parcours est une file d'attente avec priorités. Chaque URL de la frontière est associée à une valeur de priorité. Les liens sont ensuite extraits de la file d'attente dans l'ordre des priorités attribuées. Dans notre modèle d'exploration, nous estimons une valeur d'action pour chaque lien non visité et l'ajoutons à la frontière avec sa valeur d'action.

Dans l'apprentissage par renforcement, si un espace d'état est petit et discret, les fonctions de valeur d'action sont représentées et stockées sous forme de tableau. Mais cette méthode ne convient pas à notre problème d'exploration avec un grand espace d'actions et d'états. Ainsi, nous utilisons une méthode d'approximation de fonction, à savoir une approximation par fonction linéaire, pour représenter les valeurs d'action. La fonction de valeur d'action est approximée en combinant linéairement le vecteur de facteurs $\mathbf{x}(s, a)$ et le vecteur de poids $\mathbf{w}$ comme suit :

$$\hat{q}(s, a, \mathbf{w}) \doteq \mathbf{w}^\intercal \mathbf{x}(s, a) \doteq \sum_{i=1}^{d} w_i x_i(s, a). \tag{A.2}$$

Les fonctions d'état et d'action sont utilisées comme composants d'un vecteur de facteurs $\mathbf{x}(s, a)$. À chaque pas de temps, le vecteur de poids $\mathbf{w}$ est mis à jour en utilisant une méthode de descente de gradient :

$$\mathbf{w}_{t+1} \doteq \mathbf{w}_t + \alpha \left[r_{t+1} + \gamma \hat{q}(s_{t+1}, a_{t+1}, \mathbf{w}_t) - \hat{q}(s_t, a_t, \mathbf{w}_t)\right] \nabla \hat{q}(s_t, a_t, \mathbf{w}_t). \tag{A.3}$$

La valeur d'action approximative obtenue à partir de l'équation (A.2) est utilisé comme score de priorité.

Lorsque nous calculons des valeurs d'action uniquement pour les liens externes de la page en cours avec les poids nouvellement mis à jour et les ajoutons à la frontière, un problème peut survenir dans le portée des paires d'états–actions concernant le calcul de la valeur de l'action. Ce problème est causé par l'ordre de priorité de la sélection d'un lien de la frontière. Si l'agent continue d'avancer en suivant les transitions d'états successives, il est correct de calculer les valeurs d'action uniquement pour les liens externes directs, car la sélection suivante est décidée parmi l'un des

liens sortants. Cependant, dans l'ordre de priorité sélectionné à partir de la frontière, lorsque le poids du vecteur $\mathbf{w}$ est modifié, les valeurs d'action de tous les liens de la frontière doivent également être recalculés avec le nouveau $\mathbf{w}$. Nous appelons cela une méthode *synchrone*. Recalculer pour tous les liens est la bonne méthode mais cela implique un surcoût de calcul excessif. Sinon, nous pouvons calculer la valeur d'action uniquement pour les liens externes de la page en cours et/ou recalculer tous les liens (actions) dans la frontière qui proviennent du état. Les valeurs d'action de tous les autres liens de la frontière restent inchangées. Nous appelons ceci une méthode *asynchrone*. Cette méthode n'entraîne pas de surcharge de calcul, mais les valeurs d'action de tous les liens dans la frontière sont calculés à différentes étapes de temps et le rendent difficile de choisir le meilleure action de la frontière. Dans les expériences, nous comparons les performances des deux méthodes.

Puisque la méthode asynchrone a un avantage (ne pas avoir besoin de recalculer les valeurs d'action de tous les liens non visités dans la frontière), nous essayons d'améliorer la méthode asynchrone. Le problème de la méthode asynchrone est que les valeurs d'action calculés en différentes étapes de temps existent ensemble dans la frontière et peuvent causer un bruit dans la sélection. Ainsi, nous réduisons les différences de valeur d'action dans le frontière en manipulant les mises à jour de poids. L'erreur DT est la différence entre les estimations à deux pas de temps, $r + \gamma\hat{q}(s', a', \mathbf{w})$ et $\hat{q}(s, a, \mathbf{w})$. La mise à jour de l'erreur aux poids signifie que l'estimation actuelle $\hat{q}(s, a, \mathbf{w})$ est ajusté vers la cible de mise à jour $r + \gamma\hat{q}(s', a', \mathbf{w})$. Afin de modérer l'erreur DT, nous ajustons l'estimation $\hat{q}(s', a', \mathbf{w})$ par le montant de l'erreur DT lors de la mise à jour des poids comme suit :

$$\mathbf{w} \leftarrow \mathbf{w} + \alpha \left[r + \gamma(\hat{q}(s', a', \mathbf{w}) - \delta) - \hat{q}(s, a, \mathbf{w})\right] \nabla\hat{q}(s, a, \mathbf{w}) \tag{A.4}$$

où $\delta = r + \gamma\hat{q}(s', a', \mathbf{w}) - \hat{q}(s, a, \mathbf{w})$. Nous appelons cette technique une mise à jour *modérée*. En fait, cette mise à jour modérée peut avoir le même effet que la réduction de la taille de l'échelon α de la mise à jour d'origine par $1 - \gamma$.

$$\begin{aligned}\mathbf{w} &= \mathbf{w} + \alpha \left[r + \gamma(\hat{q}(s', a', \mathbf{w}) - \delta) - \hat{q}(s, a, \mathbf{w})\right] \nabla\hat{q}(s, a, \mathbf{w}) \\ &= \mathbf{w} + \alpha\delta\nabla\hat{q}(s, a, \mathbf{w}) - \alpha\gamma\delta\nabla\hat{q}(s, a, \mathbf{w}) \\ &= \mathbf{w} + \alpha(1 - \gamma)\delta\nabla\hat{q}(s, a, \mathbf{w})\end{aligned}$$

L'idée derrière la mise à jour modérée est de diminuer une surestimation de la valeur d'action ou d'augmenter une valeur d'action sous-estimée de la mise à jour cible afin de réaliser un équilibre entre les valeurs d'action mises à jour à différentes étapes de temps.

A.5 Maximisation de l'influence

A.5.1 Problème et notre méthode

Étant donné un réseau social, le problème de maximisation de l'influence est de choisir un ensemble initial optimal de graines d'une taille donnée pour maximiser l'influence sous un certain modèle de diffusion de l'information tel que le modèle de cascade indépendante (IC), modèle à seuil linéaire (LT), etc.

Algorithm 21 parcours du Web ciblé basé sur l'apprentissage par renforcement

1: Entrée : URL des graines *Seeds*, nombre maximum de pages à visiter *LIMIT_PAGES*
2: Initialise les poids de fonction de valeur $w \in \mathbb{R}^d$
3: $B \leftarrow \emptyset$ // contient (s, a) paires
4:
5: **while** *Seeds* n'est pas vide **do**
6: Sélectionner un lien l de *Seeds*
7: $s \leftarrow$ Récupérer et analyser la page l
8: $L' \leftarrow$ Extraire tous les liens externes de l
9: **for** chaque $l' \in L'$ **do**
10: $(l', s', a') \leftarrow$ Récupérer les fonctions d'action a' de l'
11: Ajouter (l', s', a') à (s', a') paire de B avec la Q-valeur initiale
12: **end for**
13: **end while**
14:
15: **while** visited_pages $<$ *LIMIT_PAGES* **do**
16: **if** avec probabilité ϵ **then**
17: Sélectionner une paire (s, a) uniformément à aléatoire à partir de B et sélectionnez un lien (l, s, a) de la paire
18: **else**
19: Sélectionner une paire (s, a) à partir de B avec la Q-valeur la plus élevée et sélectionner un lien (l, s, a) de la paire
20: **end if**
21: **if** l est visité **then**
22: continue
23: **end if**
24: $r, s' \leftarrow$ Récupérer et analyse la page (l, s, a)
25: $L' \leftarrow$ Extraire tous les liens externes de l
26: **for** chaque $l' \in L'$ **do**
27: **if** l' est visité **then**
28: continue
29: **end if**
30: $(l', s', a') \leftarrow$ Récupérer les fonctions d'action a' de l'
31: **end for**
32: **if** la page visitée est pertinente **then**
33: $w \leftarrow w + \alpha \left[r - \hat{q}(s, a, w)\right] \nabla \hat{q}(s, a, w)$
34: **else**
35: Choisir a' en fonction de $\hat{q}(s', \cdot, w)$ avec politique ϵ-gloutonne
36: $\delta \leftarrow r + \gamma \hat{q}(s', a', w) - \hat{q}(s, a, w)$
37: $w \leftarrow w + \alpha \left[r + \gamma \hat{q}(s', a', w) - \hat{q}(s, a, w)\right] \nabla \hat{q}(s, a, w)$ // mise à jour *originale*
38: $w \leftarrow w + \alpha \left[r + \gamma(\hat{q}(s', a', w) - \delta) - \hat{q}(s, a, w)\right] \nabla \hat{q}(s, a, w)$ // mise à jour *modérée*
39: **end if**
40: **for** chaque $(\cdot, \cdot)$ paire $\in B$ **do** // méthode *synchrone*
41: Calcule la Q-valeur de $(\cdot, \cdot)$
42: Mise à jour $(\cdot, \cdot)$ à B avec Q-value
43: **end for**
44: **for** chaque $(s', \cdot)$ paire $\in L'$ **do** // méthode *asynchrone*
45: Calcule la Q-valeur de $(s', \cdot)$
46: Ajouter $(l', s', \cdot)$ à $(s', \cdot)$ paire de B avec Q-valeur
47: **end for**
48: visited_pages $\leftarrow$ visited_pages + 1
49: **end while**

Dans de nombreux algorithmes existants, toute la structure topologique d'un réseau social est supposée être fournie et la connaissance complète est utilisée pour trouver les ensembles de graines optimaux. Cependant, il est connu que la connaissance complète de la topologie de la structure d'un réseau social est généralement difficile à obtenir [107, 70, 39, 77]. Même si le graphe complet est donné, le graphe peut changer dynamiquement [107]. Ainsi, dans cette étude, nous supposons que la structure du graphe est incomplète ou peut changer dynamiquement. Nous trouvons les graines les plus influentes pour un graphe inconnu en sondant des nœuds afin de connaître une partie de la structure du graphe et de découvrir des nœuds potentiellement prometteurs. Le travail le plus apparenté est la maximisation de l'influence pour les graphes inconnus proposé par Mihara et al. [77]. Leur travail montre qu'une diffusion raisonnable de l'influence peut être atteinte même lorsque la connaissance du la topologie du réseau est limitée et incomplète.

Une autre chose irréaliste dans de nombreuses méthodes existantes est que ces méthodes ne tient pas compte des intérêts actuels des utilisateurs. En fait, les utilisateurs ont leur propre intérêts et sont plus susceptibles d'être influencés par des informations liées à leurs intérêts. Autrement dit, la diffusion de l'information varie en fonction du sujet d'un message. Par exemple, un article sur les voitures sera diffusé par les utilisateurs qui sont intéressés par les voitures. Ce sera différent de la propagation de l'information d'un sujet sur les chiens. Il y a quelques travaux qui étudient l'influence basée sur le sujet dans les problèmes de maximisation [49, 15, 13]. Leurs méthodes considèrent plusieurs distributions de sujet sur les nœuds et une requête, mais la notre se concentrera sur un sujet cible et étudiera la maximisation de l'influence pour un sujet donné.

Dans cette étude, nous abordons un problème de maximisation de l'influence basée sur un sujet pour un graphe inconnu. En supposant qu'un graphe social $G = (V, E)$ est orienté, V est connu mais E n'est pas connu, nous trouvons les graines les plus influentes pour maximiser l'influence basée sur un sujet lors de la vérification des nœuds qui peuvent avoir un grand groupe d'audience. Pour sélectionner une graine, au lieu de différencier tous les nœuds individuels, nous choisissons d'abord certaines caractéristiques qui représentent l'information propre à un nœud et les informations de relation par rapport aux nœuds environnants. Nous appellerons la forme généralisée avec les caractéristiques sur les informations de relation *l'état*. La forme généralisée avec les caractéristiques sur le propre d'un nœud s'appelle *l'action*. Ensuite, nous évaluons un nœud en fonction de son action et de son état. Une *valeur d'action* signifiera combien il est utile de choisir une action (un noeud) à activer dans un état donné afin de maximiser la propagation de l'influence. L'agent choisit un noeud en fonction de sa valeur d'action à activer. Comme il est similaire au concept de valeur d'action dans l'apprentissage par renforcement [94], nous utilisons la méthodologie de l'apprentissage par renforcement pour apprendre les valeurs d'action. En bref, nous sondons les nœuds pour découvrir la structure du graphe et choisir des nœuds avec le la plus haute valeur d'action que les graines.

Avant de passer à la prochaine sous-partie, nous discutons du problème de la maximisation de l'influence et du problème de parcours du Web ciblé pour aider à comprendre notre modélisation. Rappelez-vous que dans le parcours du Web ciblé, l'agent recueille des pages Web pertinentes pour le sujet ciblé en utilisant une frontière. Le problème lui-même ne considère pas les effets à long terme, mais une

approche d'apprentissage par renforcement permet d'estimer les scores de liens à long terme, comme nous l'avons vu dans la partie précédente. Dans la maximisation de l'influence, l'agent vise à choisir les graines les plus influentes pour maximiser l'influence. Ce problème prend déjà en compte à long terme les valeurs, mais pas nécessairement la dimension de planification que l'apprentissage par renforcement introduit.

Ces deux problèmes reposent sur des objectifs différents et ont été étudiés de différentes façons, mais ils ont quelques similarités causées par des caractéristiques des graphes Web et par la nature des tâches.

Dans l'analyse ciblée, les pages Web sont connectées par des liens hypertexte, mais elles ne sont pas liées de manière aléatoire. Les pages sont susceptibles d'être liées à des pages du même sujet. Dans la maximisation de l'influence, les utilisateurs sont également susceptibles d'être des amis d'autres utilisateurs qui ont des intérêts similaires. La sélection de facteurs dans la sous-partie suivante est inspirée des facteurs utilisés dans le problème de parcours du Web ciblé.

En outre, la sélection des graines avec les valeurs d'action les plus élevées est similaire aux sélections de liens de la frontière dans un ordre de priorité dans le problème de parcours du Web ciblé. Ainsi, il peut y avoir un problème similaire à celui discuté dans la partie précédente et les valeurs d'action peuvent être équilibrées de la même manière que nous l'avons fait dans le parcours du Web ciblé. Cependant, alors que l'agent d'exploration sélectionne un lien d'une frontière pour chaque pas de temps, l'agent dans le problème de maximisation d'influence sélectionne un nœud sondé avec les valeurs d'action les plus élevées, puis se base sur la diffusion de l'information du nœud.

Nous continuons les détails de notre modélisation pour le problème de maximisation de l'influence dans la sous-partie suivante en considérant de telles similarités et différences.

A.5.2 Modélisation et Algorithme

Nous expliquons d'abord comment définir les états et les actions et calculer la valeur de actions afin de sélectionner les graines et ensuite discuter de la façon de sonder les nœuds. L'algorithme entier est représenté par l'Algorithme 22.

A.5.2.1 Sélection de graines

Comme nous l'avons mentionné ci-dessus, un nœud est généralisé avec certains facteurs qui représentent les informations propres du nœud et la relation par rapport aux nœuds avoisinants, appelés action et état, respectivement. Ensuite, nous évaluons un nœud basé sur son état et son action. Les caractéristiques des états et des actions sont présentées ci-dessous.

Etat. Les entités d'état sont basées sur des informations de relation par rapport aux nœuds environnants. Puisque la structure complète du graphe n'est pas connue à l'avance, chaque noeud ne connaît pas tous les parents réels (c'est-à-dire, les nœuds entrants) et donc nous devons mettre à jour progressivement les informations sur les parents lors de la visite des nœuds en sondant ou en traçant les nœuds activés.

En visitant un nœud, nous enregistrons pour ses nœuds enfants l'information de parenté du nœud actuel.

Pour les facteurs basées sur un sujet (ou une catégorie), afin de décider si un message est pertinent pour le sujet (ou la catégorie), nous pouvons utiliser une méthode de classification ou similarité cosinus entre un vecteur de mot du sujet donné (ou catégorie) et celle d'un poste. Lorsque nous utilisons la similarité de cosinus, un seuil θ doit être sélectionné. Ensuite, si la similarité est supérieure au seuil θ, on peut la considérer pertinente pour le sujet donné.

Sur cette base, nous pouvons calculer un taux de publication d'un sujet donné (ou catégorie) parmi tous les messages générés par un utilisateur comme suit : pour un utilisateur, le nombre de messages de l'utilisateur qui sont pertinents pour le sujet (ou la catégorie) est divisé par le nombre de tous les messages générés par l'utilisateur.

- **Taux moyen de publication de tous les parents pour un sujet donné**
- **Taux moyen de publication de tous les parents pour les catégories**
- **Changement du taux d'enregistrement pour un sujet donné**
- **Distance du dernier noeud activé**

Action. Les facteurs d'action sont basées sur les informations propres d'un nœud, composées de deux types d'informations. L'un est les comportements généraux de l'utilisateur sur un réseau social et l'autre est l'intérêt de l'utilisateur. Le facteur « Nombre d'enfants » est un bon indicateur pour voir si un utilisateur a un grand groupe d'audience ou non. Le facteur « Nombre de messages » peut être utilisé pour prédire l'activité de l'utilisateur. Les deux autres facteurs, « Taux de message pour le sujet donné » et « Taux de message pour les Catégories », représentent l'intérêt de l'utilisateur.

- **Nombre d'enfants**
- **Nombre de messages**
- **Taux de message pour le sujet donné**
- **Taux de message pour les catégories**

Sur la base des valeurs de caractéristiques discutées ci-dessus, nous évaluons la valeur d'action de chaque nœud. Comme dans la partie précédente, nous utilisons une approximation par fonction linéaire à et une descente de gradient pour calculer des valeurs d'action parce que l'espace d'action d'état est très grand. Nous définissons un vecteur de poids $\mathbf{w}$ qui a la même taille de vecteur de facteurs $\mathbf{x}(s, a)$. Rappelons que la valeur de l'action a dans l'état s, $\hat{q}(s, a, \mathbf{w})$, est approximée en combinant linéairement le vecteur de facteurs $\mathbf{x}(s, a)$ et le vecteur de poids $\mathbf{w}$:

$$\hat{q}(s, a, \mathbf{w}) \doteq \mathbf{w}^{\mathsf{T}}\mathbf{x}(s, a) \doteq \sum_{i=1}^{d} w_i x_i(s, a) \tag{A.5}$$

et le vecteur de poids $\mathbf{w}$ est mis à jour comme suit :

$$\mathbf{w}_{t+1} \doteq \mathbf{w}_t + \alpha \left[r_{t+1} + \gamma \hat{q}(s_{t+1}, a_{t+1}, \mathbf{w}_t) - \hat{q}(s_t, a_t, \mathbf{w}_t) \right] \nabla \hat{q}(s_t, a_t, \mathbf{w}_t). \tag{A.6}$$

Ici, α $(0 < \alpha \leq 1)$ est le paramètre de taille de pas qui influence le taux d'apprentissage ; r est une récompense reçue pour l'action a ; le taux d'actualisation γ $(0 \leq \gamma < 1)$ détermine la valeur actuelle des récompenses futures. s_{t+1} est l'état suivant et a_{t+1} est une action dans s_{t+1} sous une politique donnée π. Nous définissons la récompense r comme le taux de nœuds enfants activés, et (s_{t+1}, a_{t+1}) comme la paire action–état d'un noeud enfant ayant la valeur d'action la plus élevée parmi tous les nœuds enfants.

Nous mettons à jour le vecteur de poids $\mathbf{w}$ pour tous les nœuds activés. Après l'activation d'une graine, l'influence de l'information est étendue en fonction des choix des utilisateurs. Ainsi, la mise à jour doit être explicitement sur un noeud activé par un agent d'apprentissage. Nous étendons le champ d'action. L'activation d'un nœud par l'agent d'apprentissage est une *action explicite*. Nous considérons l'activation des nœuds par propagation d'influence à partir du nœud initialement sélectionné en tant qu'*action implicite* parce qu'ils donnent des récompenses et des états suivants dans le même mécanisme des nœuds explicitement activés.

Pour chaque nœud sondé, nous utilisons le $\mathbf{w}$ appris pour évaluer une valeur d'action du noeud et sélectionnons les noeuds avec les valeurs d'action les plus élevées en tant que graines. Cependant, si les valeurs d'action des nœuds sondés sont calculées à un moment différent et s'ils ne sont pas synchronisés avec le même vecteur de poids $\mathbf{w}$, nous pouvons avoir le même problème que nous avons vu dans le problème d'exploration ciblée. Dans ce cas, il serait bon d'utiliser la mise à jour *modérée* afin de réduire les différences de valeur d'action à différentes étapes de temps en manipulant les mises à jour de poids.

A.5.2.2 Sonde de nœuds

Comme une connaissance complète d'un réseau social n'est pas donnée à l'avance, nous devons sonder les nœuds afin de connaître partiellement le graphe. Une méthode efficace pour choisir les nœuds de sondage serait de calculer des valeurs d'action pour tous les nœuds inactifs et de choisir un noeud avec la valeur d'action la plus élevée. Cependant, s'il y a un grand nombre de noeuds, le coût de calcul peut être extrêmement élevé. Alternativement, nous pouvons utiliser les degrés sortants de nœuds. En effet, un nœud avec un degré sortant élevé signifie que le nœud a une grande audience. Un tel nœud est susceptible de diffuser plus largement l'information que les nœuds à faible degré sortant. Dans Sample Edge Count (SEC) [70], une méthode d'échantillonnage biaisée, les nœuds ayant le plus haut degré attendu sont sondés de manière gloutonne. Cette méthode est efficace pour trouver des nœuds concentrateurs de grand degré. Ainsi, dans notre algorithme, les degrés de nœuds attendus sont initialement mis à 0 et nous les mettons à jour progressivement tout en sondant les nœuds et en sélectionnant les nœuds avec les plus hauts degrés attendus.

A.6 Travail futur

Dans cette thèse, nous avons appliqué l'apprentissage par renforcement à plusieurs applications. Pour ces applications, notre travail crée plusieurs directions possibles pour un travail futur.

Dans le problème du routage des taxis, il faut prendre en compte le caractère non-stationnaire du problème pour adapter les politiques à la dynamique de l'environnement. Dans un vrai problème, le comportement des passagers change avec le temps. Cela signifie que les objectifs, la transition et les probabilités de récompense de l'environnement peuvent changer avec le temps.

Comme l'apprentissage par renforcement est essentiellement applicable à un environnement non-stationnaire, notre agent d'apprentissage peut s'adapter continuellement aux changements dynamiques. Cependant, nous ne pouvons pas nous attendre à une adaptation plus réactive aux changements et lorsque l'environnement revient à la dynamique précédemment apprise, les connaissances acquises dans le passé deviennent inutile. Nous aurons besoin d'une méthode qui peut explicitement adresser la non-stationnarité. L'algorithme de Q-apprentissage que nous avons utilisé est rapide, mais il ne considère pas un modèle de l'environnement. Il peut être préférable d'appliquer une méthode basée sur un modèle et de lui faire détecter les changements de l'environnement.

Afin de s'adapter de manière flexible à la dynamique de l'environnement, le modèle d'environnement du problème de taxi peut devoir être divisé en modèles partiels qui sont stockés dans une bibliothèque comme indiqué dans [19, 79, 89, 52]. À chaque fois, l'agent utilise un modèle partiel qui prédit bien l'environnement. Si l'erreur de prédiction du modèle actuel est supérieure à un seuil, l'agent sélectionne un autre modèle de la bibliothèque. Si la dynamique de l'environnement est complètement différentw des modèles existants, il crée un nouveau modèle. Cela sera une adaptation plus flexible à un environnement non stationnaire que de sélectionner des modes prédéfinis par un concepteur de système. En outre, l'application de taxi dont nous avons discuté jusqu'à présent est basée sur un agent, mais il doit être étendu à un paramètre multi-agent. Dans un environnement multi-agent, il sera important que l'agent ait la capacité de détecter les adversaires non stationnaires et d'apprendre les politiques optimales contre les stratégies adverses changées. Lorsque les stratégies adverses ne sont pas connues a priori, l'agent doit s'adapter au nouvel environnement. Au lieu de modèles fixes, les modèles flexibles proposé ci-dessus sera en mesure de traiter de tels problèmes non stationnaires.

Dans le problème de parcours du Web ciblé, le plus évident consiste à appliquer notre algorithme dans des ensembles de données plus volumineux et divers, tels que Wikipédia anglais au complet et des ensembles de données du site `http://commoncrawl.org/`, etc. Dans notre travail, nous avons utilisé une base de données de Wikipédia en anglais simple fourni par le site `https://dumps.wikimedia.org/`. L'ensemble de données était suffisamment bon pour vérifier l'efficacité du robot basé sur l'apprentissage par renforcement, mais nous devons considérer des environnements un peu plus grands et plus réalistes.

Une autre possibilité intéressante consiste à mettre en place un mécanisme efficace de sélection de catégorie. Parmi les caractéristiques d'état et d'action, les catégories liées à un sujet ciblé sont présélectionnées manuellement par un concep-

teur de système. Étant donné que la mauvaise sélection des facteurs peut entraîner des performances médiocres, il est très important de sélectionner des catégories pour le sujet cible. Le système actuel repose sur la connaissance humaine et l'intuition sur le domaine spécifique. Ils devraient être sélectionnés d'une manière intelligente et automatique.

Enfin, nous pouvons envisager plusieurs agents d'exploration afin d'accélérer les performances d'exploration. Une méthode simple utilisant plusieurs agents est que tous les agents sont complètement indépendants et qu'ils ne partagent aucune information, y compris la frontière. En fait, lorsque les agents explorent différentes parties de l'environnement, ils auront des politiques de notation distinctes en ce qui concerne leurs propres fonctions de valeur parce qu'ils auront une expérience différente. Une alternative consiste à partager l'information entre des agents tels que la frontière et les politiques de score. Dans ce cas, les politiques de score doivent être fusionnées d'une manière ou d'une autre. Quelques travaux de recherche [45, 100, 33] montrent que la combinaison des différentes politiques surpasse un seul agent.

Dans le problème de maximisation de l'influence, notre méthode n'est pas encore validée par des expériences. L'évaluation expérimentale est laissée pour un travail futur. Elle devrait être basé sur différents modèles de diffusion et différents médias sociaux.

Dans notre étude, nous avons étendu le problème de la maximisation d'influence classique avec une connaissance incomplète de la structure du graphe et avec un intérêt de l'utilisateur basé sur le sujet. En supposant que la structure du graphe est incomplète ou peut changer de façon dynamique, nous avons abordé un problème de maximisation de l'influence basée sur un sujet pour un graphe inconnu.

Nous pouvons l'étendre à un environnement plus réaliste. Dans le monde réel, il n'y a pas qu'une entreprise qui veut promouvoir son produit dans un réseau social en ligne. De nombreuses entreprises peuvent utiliser de manière concurrentielle le marketing viral sur le même réseau social. Nous appelons ce problème le problème de maximisation d'influence compétitive [67, 106, 65]. Ensuite, notre préoccupation sera de savoir comment une entreprise maximise efficacement son influence de l'information dans un média social lorsque de nombreuses entreprises diffusent leurs informations de manière compétitive dans les mêmes médias sociaux. Le problème de maximisation d'influence compétitive vise à trouver une stratégie contre les stratégies de l'adversaire. Lin et al. [67] proposent un apprentissage par renforcement pour le problème de la maximisation d'influence compétitive. Une extension de notre méthode dans un tel environnement sera un défi intéressant pour le travail futur. L'apprentissage par renforcement multi-agents sera une bonne méthode pour le problème de la maximisation d'influence compétitive pour apprendre la stratégie optimale contre les stratégies des adversaires.

A.7 Conclusion

Dans cette thèse, nous avons appliqué des méthodes d'apprentissage par renforcement à des problèmes de décision séquentiels dans des environnements dynamiques et avons exploré plusieurs méthodes différentes d'apprentissage par renforcement telles qu'une méthode sans modèle, une méthode basée sur un modèle, et une méthode

d'approximation linéaire. Il y a beaucoup d'autres différentes méthodes présentées dans la littérature. Nous ne pouvons pas dire quel algorithme est vraiment mieux que d'autres en général. Cependant, nous devons choisir une bonne représentation des états et des actions et une bonne méthode pour le problème donné et son domaine parce que la performance de l'apprentissage est influencé par la représentation et la méthode utilisées.

Nous avons essayé d'utiliser une méthode appropriée pour chaque application.

Par exemple, dans le problème du routage des taxis, une méthode sans modèle est utilisée et elle est suffisamment bonne pour apprendre les fonctions de valeur pour le problème, mais nous pouvons également utiliser une méthode basée sur un modèle lorsque nous considérons la non-stationnarité du problème. Cependant, une méthode d'approximation de fonction ne sera pas nécessaire pour ce problème parce que le problème n'a que des facteurs de position. S'il y a beaucoup de facteurs, une méthode sans modèle basée sur un tableau ne suffit pas à stocker toutes les paires d'états–action, mais doit être étendu avec une méthode approximative.

Dans le problème de parcours du Web ciblé et le problème de maximisation de l'influence, nous avons utilisé une méthode d'approximation de fonction linéaire. Puisque les espaces d'états et d'actions sont grands, nous ne pouvons pas stocker toutes les actions d'état valeurs dans les formes tabulaires. Ainsi, une méthode sans modèle basée sur un tableau n'est pas utilisée. Une méthode basée sur un modèle est également difficile à appliquer aux problèmes parce que les actions sont très bruitées.

Même si nous choisissons une méthode appropriée pour un problème donné, il peut y avoir certaines choses qui ne correspondent pas bien à la nature de la tâche, surtout si le problème est dans des conditions ou des hypothèses légèrement différentes. Dans ce cas, la méthode sélectionnée doit être adaptée au problème. Par exemple, dans le problème d'exploration ciblée et le problème de maximisation de l'influence, les algorithmes d'apprentissage ont dû être réglés pour leurs tâches.

Un autre facteur important qui influence les performances d'apprentissage est de savoir comment représenter les états et les actions. Dans le problème de routage de taxi, les espaces d'état et d'action sont clairs à définir. Cependant, il peut ne pas toujours être clair à l'avance quelles caractéristiques à utiliser pour un problème donné si le problème est complexe et difficile à modéliser dans un MDP ou s'il est difficile de savoir quelles sont les caractéristiques de l'environnement. Par exemple, dans le problème de parcours du Web ciblé et le problème de la maximisation de l'influence, il n'était pas simple de sélectionner des facteurs qui représentent des états et des actions.

Comme nous l'avons vu à travers cette thèse, l'apprentissage par renforcement est une bonne méthode pour résoudre un problème de décision séquentiel dans un environnement dynamique. Il est important de choisir une bonne représentation des états et des actions et une méthode appropriée pour un problème donné.

Algorithm 22 Maximisation de l'influence basée sur le sujet pour un graph inconnu

1: $S \leftarrow \emptyset$, $A \leftarrow \emptyset$, $C \leftarrow \emptyset$ // ensemble de graines S, ensemble de nœuds actif A, ensemble de nœuds sondé C
2: **for** chaque noeud $z \in V$ **do**
3: $\quad d_{out}(z) \leftarrow 0$
4: **end for**
5: **for** $t = 1 \ldots k$ **do**
6: $\quad$ // Sonde
7: $\quad$ ProbingNodes(C)
8:
9: $\quad$ // Sélectionne un noeud de départ
10: $\quad$ **if** Avec probabilité ϵ **then**
11: $\quad\quad$ Sélectionner une paire (s, a) uniformément au hasard de C et sélectionner un noeud (z, s, a) de la paire
12: $\quad$ **else**
13: $\quad\quad$ Sélectionner une paire (s, a) de C avec la plus grande valeur d'action et sélectionner un noeud (z, s, a) de la paire
14: $\quad$ **end if**
15: $\quad S \leftarrow S \cup \{(z, s, a)\}$
16: $\quad$ Activer le noeud z
17: $\quad$ Créer $A_t = \{(z', d)$: noeud activé z' à l'instant t, distance d de $z\}$
18: $\quad A \leftarrow A \cup A_t$
19: $\quad$ **for** $d' = 0 \ldots \max\ d$ **do**
20: $\quad\quad$ **for** chaque noeud activé $(z', d') \in A_t$ **do**
21: $\quad\quad\quad$ Définir l'état s' et l'action a' de z'
22: $\quad\quad$ **end for**
23: $\quad$ **end for**
24: $\quad$ **for** chaque noeud activé $(z', d) \in A_t$ **do**
25: $\quad\quad$ Récupérer la paire d'actions d'état (s', a') de z' et observer r
26: $\quad\quad (s'', a'') \leftarrow \underset{(s'',a'') \in \text{Etat-Action}(out(z'))}{\arg\max} v(s'', a'', w)$
27: $\quad\quad w \leftarrow w + \alpha\left[r + \gamma\hat{q}(s'', a'', w) - \hat{q}(s', a', w)\right] \nabla\hat{q}(s', a', w)$
28: $\quad\quad$ **if** $z' \in C$ **then**
29: $\quad\quad\quad$ Supprimer (z', d) de C
30: $\quad\quad$ **else**
31: $\quad\quad\quad$ **for** chaque noeud parent $p \notin C$ **do**
32: $\quad\quad\quad\quad d_{out}(p) \leftarrow d_{out}(p) + 1$
33: $\quad\quad\quad$ **end for**
34: $\quad\quad$ **end if**
35: $\quad$ **end for**
36: $\quad t \leftarrow t + 1$
37: **end for**
38: retour S

Algorithm 23 ProbingNodes

1: **for** $j = 1 \ldots m$ **do**
2: **if** Avec probabilité ϵ **then**
3: Sélectionner le noeud inactif $z \notin C$ uniformément au hasard
4: **else**
5: Sélectionnez le noeud inactif $z = \arg\max_{z \in V}\{d_{out}(z) \mid z \notin C\}$
6: **end if**
7: $d_{out}(z) \leftarrow$ réel hors-degré de z
8: **for** chaque noeud parent $p \notin C$ **do**
9: $d_{out}(p) \leftarrow d_{out}(p) + 1$
10: **end for**
11: Extraire l'état s et l'action a de z
12: Calculer la valeur de l'action avec w
13: Ajouter (z, s, a) à s de C avec la valeur de l'action
14: **end for**

Reinforcement Learning Approaches in Dynamic Environments

Miyoung HAN

RÉSUMÉ: L'apprentissage par renforcement consiste en apprendre de l'interaction avec un environnement pour atteindre un but. C'est un cadre efficace pour résoudre les problèmes de décision séquentiels, basée sur l'utilisation des processus de décision de Markov (MDP) comme formulation générale. Dans cette thèse, nous appliquons l'apprentissage par renforcement à des problèmes de décision séquentiels dans des environnements dynamiques.

Nous présentons d'abord un algorithme basé sur le Q-apprentissage avec une stratégie personnalisée d'exploration et d'exploitation pour résoudre un problème réel de routage de taxi. Notre algorithme est capable d'apprendre progressivement les actions optimales pour acheminer un taxi autonome aux points de collecte des passagers. Ensuite, nous abordons le problème des MDP factorisés dans un contexte non-déterministe. Nous proposons un algorithme qui apprend les fonctions de transition en utilisant le formalisme des réseaux bayésiens dynamiques. Nous démontrons que les méthodes de factorisation permettent d'apprendre efficacement des modèles corrects ; à travers les modèles appris, l'agent peut accumuler des récompenses cumulatives plus grandes.

Nous étendons notre travail à de très grands domaines. Dans le problème de parcours du Web ciblé (focused crawling), nous proposons un nouveau mécanisme de score prenant en compte les effets à long terme de la sélection d'un lien, et présentant de nouvelles représentations des caractéristiques des états pour les pages Web et les actions de sélection du lien suivant. Cette approche nous a permis d'améliorer l'efficacité du parcours du Web ciblé. Dans le problème de maximisation de l'influence (MI), nous étendons le problème de la MI classique avec une connaissance incomplète de la structure du graphe et un intérêt utilisateur basé sur le sujet. Notre algorithme trouve les graines les plus influentes pour maximiser l'influence dépendante du sujet, en apprenant des valeurs d'action pour chaque nœud sondé.

MOTS-CLEFS : Apprentissage par renforcement, Processus de décision de Markov, Parcours du Web ciblé

ABSTRACT: Reinforcement learning is learning from interaction with an environment to achieve a goal. It is an efficient framework to solve sequential decision-making problems, using Markov decision processes (MDPs) as a general problem formulation. In this thesis, we apply reinforcement learning to sequential decision-making problems in dynamic environments.

We first present an algorithm based on Q-learning with a customized exploration and exploitation strategy to solve a real taxi routing problem. Our algorithm is able to progressively learn optimal actions for routing an autonomous taxi to passenger pick-up points. Then, we address the factored MDP problem in a non-deterministic setting. We propose an algorithm that learns transition functions using the Dynamic Bayesian Network formalism. We demonstrate that factorization methods allow to efficiently learn correct models; through the learned models, the agent can accrue higher cumulative rewards.

We extend our work to very large domains. In the focused crawling problem, we propose a new scoring mechanism taking into account long-term effects of selecting a link, and present new feature representations of states for Web pages and actions for next link selection. This approach allowed us to improve on the efficiency of focused crawling. In the influence maximization (IM) problem, we extend the classical IM problem with incomplete knowledge of graph structure and topic-based user interest. Our algorithm finds the most influential seeds to maximize topic-based influence by learning action values for each probed node.

KEY-WORDS: Reinforcement Learning, Markov Decision Processes, Focused Crawling