Reinforcement Learning: Principles And Applications

B.Tech Project Report

submitted in partial fulfillment of the requirements
for the award of degree of
Bachelor of Technology

By

Ankit Jain
Roll No : 99010049

under the guidance of
Dr. P. G. Awate

Department of Mechanical Engineering
Indian Institute of Technology
Bombay
May, 2003

Acceptance Certificate

The B. Tech. project report entitled **"Reinforcement Learning: Principles and Aplications"**, submitted by **Ankit Jain (Roll No. 99010049)** may be accepted.

12 May, 2003.

(Guide: Dr. P.G. Awate)

Acknowledgement

I would like to express my sincere gratitude to my guide **Dr. P. G. Awate** for the invaluable guidance and support given to me during the course of the project.

IIT Bombay — Ankit Jain

May 2003 — (99010049)

Abstract

Reinforcement learning is one of the most active research areas in machine learning, artificial intelligence, and neural network research. This field has developed strong mathematical foundations and impressive applications. It can be used to solve very large and complex stochastic decision and control problems. Reinforcement learning combines central ideas of dynamic programming, machine learning, neural network architectures and simulation. This report contains a discussion on some of the major issues in reinforcement learning including dynamic programming and neural networks. For application part two problems, Elevator Scheduling and Traveling Salesman Problem, that can be solved using reinforcement learning are discussed in detail. For each problem the algorithms and approximation architectures to be used are explained. Many algorithms are developed for solving TSP using ant system and there relative merits and demerits are compared by actually implementing these algorithms on various Traveling Salesman Problem.

Contents

Acceptance Certificate i

Acknowledgement ii

Abstract iii

Contents iv

List of Figures ix

List of Tables x

Nomenclature xi

1 Introduction 1

1.1 Organisation of the Report 2

2 Principles of Reinforcement Learning 3

2.1 Elements of Reinforcement Learning 4

2.1.1 Agent 4

2.1.2 Environment 4

2.1.3 Policy 4

2.1.4 Reward Function 5

2.1.5 Return 5

2.2 Markov Decision Process 6

2.3 Value Function and Cost Function 6
2.4 Optimal Functions 7
2.5 Balancing Exploration and Exploitation 8
2.5.1 ϵ-Greedy Method 8
2.5.2 Softmax Action Selection 9
2.5.3 Constant Step Size Method 9

3 Dynamic Programming 11
3.1 Introduction 11
3.2 Finite Horizon Problem 12
3.3 Infinite Horizon Problem 12
3.4 Stochastic Shortest Path Problems 13
3.4.1 Value Iteration 14
3.4.2 Asynchronous Value Iteration 15
3.4.3 Policy Iteration 16
3.4.4 Asynchronous Policy Iteration 16
3.5 Discounted Problems 17

4 Simulation Methods for a Lookup Table Representation 18
4.1 Introduction 18
4.2 Monte Carlo Method 19
4.2.1 Policy Evaluation by Monte Carlo Simulation 19
4.2.2 On Policy Monte Carlo Control 20
4.3 Temporal Difference learning 21
4.3.1 TD(0) Algorithm 21
4.3.2 N-step TD Prediction 22
4.3.3 TD(λ) Algorithm 22
4.3.4 Advantages of TD Prediction Methods 23
4.3.5 Every-Visit and First Visit Variants 23
4.3.6 Off-Line and On-Line Variants 24

4.3.7 TD(λ) for Discounted Problems 24
4.3.8 Using Eligibility Traces 24
4.4 Q-Learning 25
4.5 Optimistic Policy Iteration (Actor-Critic Methods) 27

5 Neural Networks 29

5.1 What is a Neural Network? 29
5.1.1 Structure of the Nervous System 30
5.1.2 Models of a Neuron 30
5.2 Types of Activation Function 31
5.3 Feedback 33
5.4 Architectures for Approximation 34
5.5 Linear Architecture 35
5.6 Non linear Architecture 36
5.7 Back-propagation Algorithm 37
5.7.1 The Two Phases of Computation 38
5.7.2 Differentiation of Sigmoidal Activation Function 38
5.8 Steepest Gradient Method 39

6 Function Approximation using Neural networks 41

6.1 Approximate TD(1) 41
6.2 Approximate TD(λ) for general λ 42
6.3 Approximate Q-learning 43

7 Applications of Reinforcement Learning Algorithms 45

7.1 Introduction 45
7.2 Job-shop Scheduling 45
7.3 Robot Control 46
7.4 TD-Gammon 46

8 Elevator Scheduling using Reinforcement Learning 48

8.1 Problem Formulation 48

8.1.1 Passenger Arrival Patterns 48

8.1.2 The Elevator Simulator 50

8.1.3 System Dynamics 51

8.1.4 State Space 51

8.1.5 Control Action and Constraints 52

8.1.6 Performance Criteria 52

8.2 The Algorithm 53

8.2.1 Elevator System in Continuos State Space 53

8.2.2 Method of Omniscient Reinforcements 55

8.2.3 Method of Online Reinforcements 55

8.2.4 Making Decisions and Updating Q-Values 56

8.3 The Network Used in Elevator Scheduling 57

9 Solving Traveling Salesman Problem (TSP) 60

9.1 The Problem 60

9.1.1 Application of TSP 61

9.2 A Real Ants Colony 61

9.2.1 Foraging Behavior of Ants 62

9.3 Artificial Ants 64

9.3.1 Ant Algorithm for TSP 64

9.3.2 Ant Decision 65

9.3.3 Pheromone Update 66

9.4 Using Candidate Lists 67

9.5 Parameter setting and basic properties 67

9.6 Results 68

9.7 Discussion on ACS algorithm 72

9.8 Local Search Algorithms for TSP 74

9.9 Result of ACS System with Local search 75
9.10 MAX-MIN Ant System 76
9.10.1 Fastening the Trial Update 77
9.11 Result of MAXMIN ANT SYSTEM 77
9.12 Improved Max-Min Ant System 78

10 Conclusion 79

11 Future Work 81

Appendix 84

A-1 Probability Distribution 84
A-1.1 Poisson Distribution 84
A-1.2 Negative Exponential Distribution 84
A-1.3 Gamma Probability Distribution 85
A-2 Generating Random numbers 85
A-2.1 Inversion Method 85
A-2.2 Inversion by Chop Down Search from 0 86

References 87

List of Figures

2.1 The reinforcement-learning framework 4

4.1 Accumulating and replacing traces [Singh & Sutton 1996] 25

4.2 The Actor-Critic Architecture [Sutton & Barto, 1998] 27

5.1 Nonlinear model of a neuron 30

5.2 Threshold function 32

5.3 Piecewise-linear function 32

5.4 Sigmoidal Function for varying slope parameter$a$ 33

5.5 sigle-loop feedback system 33

5.6 A feedforward neural network with a single hidden layer and a single output 36

8.1 Three main passenger traffic components 49

8.2 Neural Network Architecture 59

9.1 Foraging behavior of ants 63

9.2 A 2-Opt move: original tour on the left and resulting tour on the right . . . 74

9.3 Two possible 3-Opt moves: original tour on the left and resulting tours on the right 75

11.1 The game of Tetris 82

11.2 Finding optimal tour through the network 83

List of Tables

4.1 On-Policy Monte Carlo Control Algorithm 20

8.1 Mean Passenger arrivals 51

9.1 Variation of convergence rate and value of optimum reached in ACS algorithm with initial value of Pheromone τ_0. 68

9.2 Variation of convergence rate and value of optimum reached in ACS algorithm with the parameter q_0 69

9.3 Variation of convergence rate and value of optimum reached in ACS algorithm with the number of ants. 69

9.4 Variation of convergence rate and value of optimum reached in ACS algorithm with the size of candidate list. 70

9.5 Variation of convergence rate and value of optimum reached in ACS algorithm with the Pheromone evaporation factor ρ. 70

9.6 Variation of convergence rate and value of optimum reached in ACS algorithm with the value of α/β 71

9.7 Performance of ACS algorithm when applied to various TSPs. 71

9.8 Performance of ACS algorithm with local search when applied to various TSPs. 75

9.9 Performance of MMAS algorithm with local search when applied to various TSPs. 77

Nomenclature

t	Discrete time step
T	Final time step of an episode
n	Discrete step
i	Any state
u	Control/Action
s_t	State at t
a_t	Action at t
r_t	Reward at t
R_t	Return following t
s'	State following s
j	State following i
π	Policy
μ	Stationary Policy
$g(i, u, j)$	Cost incurred while making a transition from state i to state j under control u
$R_t^{(n)}$	n-step return
R_t^{λ}	λ-return
$\pi(s, a)$	Probability of taking action a in state s under policy π
I	Set of all non terminal states
$R_{ss'}^{a}$	Expected immediate reward on transition s to s' under action a
$P_{ss'}^{a}$	Probability of transition from state s to state s' under action a
$A(s)$	Set of all possible actions in state s
$U(i)$	Set of all possible controls

$J^{\pi}(i)$	Cost-to-go of the state when in state i and following policy π thereafter
$Q(i,u)$	Cost-to-go for state action pair (i,u)
$J^{*}(i)$	Optimal Cost-to-go of state i
$\tilde{J}(i,r)$	Approximation for $J(i)$
r	Weight of neural network
$V^{\pi}(s)$	The long term value of state under policy π
$V^{*}(s)$	Value of a state s under the optimal policy
ϵ	Percentage of non greedy action (exploration)
d_m	Temporal difference error at m
α	Learning rate in TD prediction, step size
γ	Discount-rate parameter
$e_n(i)$	Eligibility trace for state i at n
λ	Decay-rate parameter for eligibility traces

Chapter 1

Introduction

There are several parallels between animal and machine learning. Many of the machine learning algorithm developed because psychologists were making efforts towards computational models of their theories of animal and human learning [Dorigo, Maniezzo & Colorni, 1996]. Reinforcement learning algorithm is one such algorithm. The term "Reinforcement learning" is a part of Psychologist jargon since mid thirties wheareas in artificial intelligence community its a relatively new term.

Now, the reinforcement learning is one of the most active research areas in machine learning and artificial intelligence research. The reinforcement learning algorithms claim to deal effectively with the dual curses of dynamic programming and stochastic optimal control: *Bellman's curse of dimensionality* (the exponential computational explosion with the problem dimension) and the *curse of modeling* (the requirement of an explicit system model). Reinforcement learning is particularly attractive for dynamic situations where it is costly or difficult (if not impossible) to gather a satisfactory set of input-output examples. Over the years reinforcement learning has developed many interesting applications. The problem that is being solved using reinforcement learning algorithms in this project is - *Traveling salesman Problem.* Given a set of cities, find an optimal routethrough which the tour length is minimum. The algorithm applied in this case is inspired by the behavior of an ant colony.

Other problem that is being discussed in detail and algorithms are decveloped but not solved due to time constraint is - Given a 10-story building with 4 elevators, the problem is

to distribute the elevator in such a way that the sum of squared waiting time of all passengers is minimized.

1.1 Organisation of the Report

The report is divided into two parts.

1. *First part*, consisting of chapter 2 to chapter 6 mainly describes the various reinforcement learning algorithm. It also contains introductory material on dynamic programming and neural networks. Chapter 2 provides the basics of reinforcement learning with the definition of various terms used in the algorithms. Chapter 3 introduces the concept of dynamic programming and the various iteration algorithms used. Chapter 4 gives a description of the algorithms like TD learning and Q-learning which are center of reinforcement learning. Chapter 5 provides a brief overview of Neural networks that are being used as function approximators. The chapter ends with the description of *back-propagation algorithm* that is being used in the elevator scheduling problem. Chapter 6 describes approximation of the cost-to-go functions using neural networks.

2. *Second Part*, consisting of chapter 7 to chapter 9 focuses on the application part of reinforcement learning. Chapter 7 describes some of the interesting application of reinforcement learning in brief. Chapter 8 provides a detail description of the elevator scheduling problem and the algorithm that is being used to solve the problem. chapter 9 starts with a study of behavior of real ants and then describes some artificial ant algorithms to solve Traveling Salesman Problem. The relative virtuosity of these algorithms are compared as they are applied to various TSPs.

Chapter 2

Principles of Reinforcement Learning

Reinforcement learning as a learning paradigm falls somewhere in between the traditional paradigms of supervised learning and unsupervised learning. In reinforcement learning there is no teacher but there is a 'critic' who give a reinforcement signal (reward) positively correlated with the merits of the action taken [Mitchell, 1997]. In most of the cases the actions taken by the learning agent may affect not only the immediate reward, but also the next situation and through that all subsequent rewards. These two characteristics–*trial and error search* and *delayed reward*–are the two most important feature of reinforcement learning [Sutton & Barto, 1998].

An interesting problem that arises when using Reinforcement Learning is the tradeoff between *exploration* and *exploitation*. If an agent has tried a certain action in the past and got a decent reward, then repeating this action is going to reproduce the reward. In doing so, the agent is exploiting what it knows to receive a reward. On the other hand, trying other possibilities may produce a better reward, so exploring is definitely a good tactic sometimes. Without a balance of both exploration and exploitation the RL agent will not learn successfully. The most common way to achieve a nice balance is to try a variety of actions while progressively favoring those that stand out as producing the most reward.

A variety of different problems can be solved using Reinforcement Learning. These methods aim to provide effective optimal/suboptimal solutions of planning and decision making under uncertainty, that for a long time were thought to be intractable. Because RL agents can learn without expert supervision, the type of problems that are best suited to RL

are complex problems where there appears to be no obvious or easily programmable solution. The two fields where RL algorithms are particularly useful are game playing (checkers, backgammon etc) and control problems such as job shop scheduling, elevator scheduling. Recently RL has drawn attention of researchers to use it as a way of programming robots by reward and punishment without the need to specify how the task is to achieve.

2.1 Elements of Reinforcement Learning

2.1.1 Agent

In a reinforcement learning problem the learner and decision maker is called the learning *agent*

2.1.2 Environment

It comprises of everything outside the agent. The learning agent and environment interact continually, the agent selecting actions and the environment responding to those actions and presenting new situations to the agent. An agent manipulates its environment through a

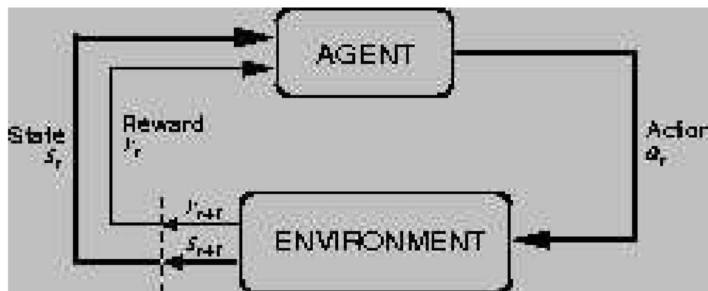

Figure 2.1: The reinforcement-learning framework

series of actions, and in response to each action, receives a reward value.

2.1.3 Policy

A policy defines the learning agent's way of behavior at a given time [Sutton & Barto, 1998]. At each time step, the agent implements a mapping from state representation to probabilities of selecting each possible action. The mapping is called the agent policy and denoted π_t.

$\pi_t(s,a)$ is the probability that (action taken at time step t) $a_t = a$ if (state at time step t) $s_t = s$. A policy π can be thought of as a sequence $\pi = \{\mu_0, \mu_1.....\}$ where each μ_k is a function mapping states into controls with $\mu_k(i) \in U((i)$ for all states i. A policy π is said to be stationary if it is of the form $\pi = \{\mu, \mu...\}$. For brevity the policy $\{\mu, \mu....\}$ is referred as a stationary policy μ.

- On-Policy RL: The on-policy reinforcement learning methods attempt to evaluate and improve the same policy that they use to make decisions.
- Off-Policy RL: In off-policy methods there is a policy used to generate behavior or actions, called the behavior policy, which may in fact be unrelated to the policy that is evaluated and called the estimation policy.

2.1.4 Reward Function

A reward function defines the goal in a reinforcement learning problem. It maps the state action pairs of the environment to a single number, a *reward*, indicating the intrinsic desirability of the state. The value of the reward varies from step to step. In RL most of the problems are of **delayed rewards**. Through its choice of action the agent affects not only its next reward but also all subsequent rewards, which are said to be delayed with respect to the previous actions. A reward can be both positive or negative. A negative reward is sometime referred as costs. $g(i, u, j)$ denotes the cost incurred while moving from state i to state j under control u. [Bertsekas & Tsitsiklis, 1996]

2.1.5 Return

Return R_t is defined as some specific function of reward sequence. In simplest case it is sum of rewards:

$$R_t = r_{t+1} + r_{t+2} + r_{t+3} + ... + r_T \tag{2.1}$$

where $r_{t+1}, r_{t+2}, r_{t+3}....r_T$ are rewards obtained at time step $t+1, t+2, t+3....T$ (the final time step). This approach make sense in application when the agent environment interaction

breaks naturally into subsequences called *episodes*, such as plays of a game, trips through maze etc [Sutton & Barto, 1998].

There is another case of continual tasks, where there is need of a concept called discounting. The agent tries to select actions so that the sum of delayed rewards or *discounted return* is maximized.

$$R_t = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + ... = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \tag{2.2}$$

2.2 Markov Decision Process

An environment satisfies the Markov property if its state completely summarizes the past without degrading the ability to predict the future. For example, a checkers position–the current position of all the pieces on the board would serve as Markov state because it summarizes everything important about the complete sequence of positions that led to it. The information that really matters is retained while the irrelevant information might be lost. A reinforcement learning task that satisfies the Markov property is called a *Markov decision process* or *MDP*. If the state and action are finite then it is called a finite MDP. A particular finite MDP is defined by its state and action sets and by one step-dynamics of the environment. Given any state and action s and a, the transition probability of each possible next state, s', is

$$P^a_{ss'} = Pr\{s_{t+1} = s' | s_t = s, a_t = a\} \tag{2.3}$$

Similarly given any current state and action, s and a, together with any next state, s', the expected value of the next reward is

$$R^a_{ss'} = E\{r_{t+1} | s_t = s, a_t = a, s_{t+1} = s'\}. \tag{2.4}$$

2.3 Value Function and Cost Function

The value of a state is the total amount of reward an agent can expect to accumulate over the future starting from that state. Values indicate the long-term desirability of states after taking into account the states that are likely to follow, and the rewards available in those

states. The value of a state s under a policy π, denoted $V^{\pi}(s)$, is the expected return when starting in s and following π thereafter. For MDPs with infinite state space (*infinite horizon problem* $V^{\pi}(s)$ can be defined as:

$$V^{\pi}(s) = E_{\pi}\{R_t|s_t = s\} = \lim_{N\to\infty} E_{\pi}\{\sum_{k=0}^{N-1} \gamma^k r_{t+k+1}|s_t = s\} \tag{2.5}$$

where $E\{\}$ denotes the expected value given that the agent follows policy π. The function V^{π} is called the state-value function for policy π.

Similarly we can define the value of taking action a in state s under a policy π, denoted $Q^{\pi}(s,a)$, as the expected return starting from s, taking the action a and thereafter following policy π.

$$Q^{\pi}(s,a) = E_{\pi}\{R_t|s_t = s, a_t = a\} \tag{2.6}$$

Q^{π} is called the action value function for policy π.

The cost of a state is the total expected cost that can accumulate over the future starting from that state. The cost of a state i, under a policy π, denoted $J^{\pi}(i)$, is the expected cost when starting in i and following π thereafter. For MDPs with finite state space *finite horizon problem* with N-state, $J_N^{\pi}(i)$ can be defined as:

$$J_N^{\pi}(i) = E_{\pi}\{\gamma^N G(i_N) \sum_{k=0}^{N-1} \gamma^k g(i_k, \mu_k(i_k), i_{k+1})|i_0 = i\} \tag{2.7}$$

where $\gamma^N G(i_N)$ is a terminal cost for ending up with final state i_N. For an infinite horizon problem $J^{\pi}(i)$ is

$$J^{\pi}(i) = \lim_{N\to\infty} E_{\pi}\{\sum_{k=0}^{N-1} \gamma^k g(i_k, \mu_k(i_k), i_{k+1})|i_0 = i\} \tag{2.8}$$

2.4 Optimal Functions

A function denoted by $V^{\star}$ is called the optimal value function if:

$$V^{*}(s) = max_{\pi} V^{\pi}(s) \tag{2.9}$$

for all $s \in S$. The optimal N stage cost-to-go function starting from state i, denoted by $J_N^{*}(i)$ is

$$J_N^{*}(i) = min_{\pi} J_N^{\pi}(i) \tag{2.10}$$

The optimal infinite horizon cost-to-go starting from state i, denoted by $J^*(i)$, is defined as

$$J^*(i) = min_\pi J^\pi(i) \tag{2.11}$$

Similarly an optimal action value function is defined as:

$$Q^*(s,a) = max_\pi Q^\pi(s,a) \tag{2.12}$$

for all $s \in S$ and for all $a \in A(s)$.

2.5 Balancing Exploration and Exploitation

A distinguishing feature of reinforcement learning method is that it uses training information that evaluates the actions taken rather than instructs by giving correct actions. Purely evaluative feedback indicates how good the action taken was, but not whether it was the best or the worst action possible. This is what creates the need for active exploration, for an explicit trial-and-error search for good behavior. In this section the evaluative aspect of RL is studied in a non-associative setting, one that does not involve learning to act in more than one situation. An example of non-associative, evaluative feedback problem is an **n-armed bandit problem** in which one is faced with a choice among **n** different actions, but each time one end up in same state. As mentioned earlier there is a constant tradeoff between exploration and exploitation. Exploitation is the right thing to do to maximize expected reward on the one play but exploration may produce greater total reward in the long run.

2.5.1 ϵ-Greedy Method

The true value of an action $Q^*(a)$ is the mean reward given that the action is selected. One natural way to estimate this is by averaging the rewards actually received when the action was selected [Sutton & Barto, 1998]. If after t decisions, action a has been chosen k_a times, yielding rewards r_1, $r_2 .... r_{k_a}$, then its value is estimated to be

$$Q_t(a) = \frac{r_1 + r_2 + ...r_{k_a}}{k_a} \tag{2.13}$$

The simplest action selection rule is to select on the tth play one of the greedy actions, a_t^*, for which $Q_{t-1}(a_t^*) = max_a Q_{t-1}(a)$. But this method don't explore at all. A good alternative is to use ϵ-greedy method which is defined as follows:

- Choose the action a with maximum $Q(s, a)$ value with probability 1-ϵ.
- Choose a random action otherwise

2.5.2 Softmax Action Selection

Although ϵ-greedy action-selection is an effective and popular means of balancing exploration and exploitation in reinforcement learning, one drawback is that when it explores it chooses equally among all actions. This means that it is just as likely to choose the worst appearing action as it is to choose the next-to-best [Sutton & Barto, 1998]. An alternative is to use softmax action selection method in which actions are ranked and weighted according to their value estimates. The most common softmax method uses a Gibbs, or Boltzmann, distribution. It chooses action a on the tth play with probability

$$\frac{e^{Q_{t-1}(a)/\tau}}{\sum_b e^{Q_{t-1}(b)/\tau}} \tag{2.14}$$

Where τ is a positive parameter called the temperature. High temperatures cause the actions to be all (nearly) equi-probable. Low temperatures cause a greater difference in selection probability for actions that differ in their value estimates.

2.5.3 Constant Step Size Method

Let Q_k denote the average of its first k rewards. Given this average and a $(k+1)^{st}$ reward, r_{k+1}, the average of all $k+1$ rewards can be represented as:

$$Q_{k+1} = Q_k + \frac{1}{k+1}[r_{k+1} - Q_k] \tag{2.15}$$

The above equation is of the form:

New-estimate $\leftarrow$ Old-estimate + Step size[Target - Old-estimate]

The step size is generally represented by $\alpha_k(a)$. In the above case step size $\alpha_k(a)$ is equal to $\frac{1}{k_a}$ which is not constant. If the step size is taken as constant $0 < \alpha \leq 1$ then the equation will be of the form

$$Q_k = (1-\alpha)^k Q_0 + \sum_{i=1}^{k} \alpha(1-\alpha)^{k-i} r_i \tag{2.16}$$

Each reward r_i is given a weight $\alpha(1-\alpha)^{k-i}$ depending on how many step ago it was observed. The weight decays exponentially according to the exponent on $1-\alpha$

Chapter 3

Dynamic Programming

In this chapter some of the basics of dynamic programming such as "Value Iteration" and "Policy Iteration" are discussed. The finite horizon Dynamic Programming (DP) is modeled as a stochastic shortest path problem. There is a section on Temporal Difference-Based Policy Iteration towards the end.

3.1 Introduction

The principal elements of a problem in dynamic programming are

1. A discrete-time dynamic system whose state transition depends on a control/action. Throughout this chapter, there is an assumption that there are n states, denoted by 1,2,...n, plus possibly an additional termination state denoted by 0. When at state i, the control must be chosen from a given finite set $U(i)$. At state i, the choice of a control/action u specifies the transition probability $p_{ij}(u)$ to the next state j

2. A cost that accumulates additively over time and depends upon the state visited and the controls chosen. At the kth transition the cost incurred is $\gamma^k g(i, u, j)$, where g is a cost function, and γ is a scaler with $0 < \gamma \leq 1$, called the discount factor. The meaning of $\gamma < 1$ is that the future cost is mattered less as compared to the cost incurred at the present time.

3.2 Finite Horizon Problem

Consider the case when there is only one stage, that is N=1. Now by equations (2.7) and (2.10), the optimal cost-to-go is

$$J_1^*(i) = \min_{\mu_0} \sum_{j=1}^{n} p_{ij}(\mu_0(i))(g(i, \mu_0, j) + \gamma G(j)) \tag{3.1}$$

For any fixed state i, the minimization over μ_0 is equivalent to minimization over $u \epsilon U(i)$, therefore

$$J_1^*(i) = \min_{u \epsilon U(i)} \sum_{j=1}^{n} p_{ij}(u)(g(i, u, j) + \gamma G(j)) \tag{3.2}$$

The above formula is interpreted as the the optimal control choice with one period to go must minimize the sum of the expected present stage cost and the expected future cost $G(j)$ appropriately discounted by γ. The above formula is generalized as the optimal control choice with k stages to go must minimize the sum of the expected present stage cost and the expected optimal cost $J_{k-1}^*(j)$ with $k-1$ stages to go, appropriately discounted by γ. That is

$$J_k^*(i) = \min_{u \epsilon U(i)} \sum_{j=1}^{n} p_{ij}(u)(g(i, u, j) + \gamma J_{k-1}^*(j)) \tag{3.3}$$

for $i = 1, ...n$

Thus, the optimal N-stage cost-to-go vector $J_N^*(i)$ can be calculated recursively with the above formula, starting with

$$J_0^*(i) = G(i) \tag{3.4}$$

for $i = 1, ...n$

3.3 Infinite Horizon Problem

The problem can be divided into three parts:

1. *Stochastic shortest path problems.* Here $\gamma = 1$ but there exists an additional state 0, which is a cost free termination state. The structure of the problem is such that termination is inevitable, at least under an optimal policy. The problem is in effect

a finite horizon problem, but the length of the horizon may be random and may be affected by the policy being used.

2. *Discounted Problems.* Here $\gamma < 1$. The absolute one-stage cost $|g(i,u,j)|$ is bounded from above by some constant M for all i, u, j. Therefore the cost-to-go $J^\pi(i)$ is well defined because it is the infinite sum of a sequence of numbers that are bounded in absolute value by the decreasing geometric progression $\gamma^k M$. For these problems the infinite horizon cost is the limit of the corresponding N-stage problem.

$$J^*(i) = \lim_{N\to\infty} J_N^*(i) \tag{3.5}$$

3. cost per stage problems. Sometimes $J^\pi(i) = \infty$ for every policy π and initial state i (for example when $\gamma = 1$ and the cost of every state and action is positive). In such cases the average cost per stage is minimized which is given by

$$\lim_{N\to\infty} \frac{1}{N} J_N^\pi(i) \tag{3.6}$$

3.4 Stochastic Shortest Path Problems

Here, there is no discounting ($\gamma = 1$) and there exists a cost free termination state, denoted by 0. Once the system reaches that state, it remains there at no further cost, that is

$$p_{00}(u) = 1,\ g(0,u,0) = 0, \text{ for all } u \in U(0)$$

A stationary policy μ is said to be **proper** if, using this policy, there is positive probability that the termination state will be reached after at most n stages, regardless of the initial state, that is, if

$$\rho_\mu = \max_{i=1,\ldots,n} P(i_n \neq 0 | i_0 = i, \mu) < 1 \tag{3.7}$$

It is assumed that there exist at least one proper policy and for every improper policy μ, the corresponding cost-to-go $J^\mu(i)$ is infinite for at least one state i.

3.4.1 Value Iteration

For any vector $J = (J(1), J(2)...., J(n))$, the vector TJ is obtained by applying one iteration of the DP algorithm to J; the components of TJ are

$$TJ(i) = \min_{u \in U(i)} \sum_{j=0}^{n} p_{ij}(u)(g(i,u,j) + J(j)) \tag{3.8}$$

for $i = 1, ...n$ and $J(0) = 0$. TJ is the optimal cost-to-go vector for the one stage problem that has one stage cost g and terminal cost J. Similarly for any vector J and any stationary policy μ, the components of vector $T_\mu J$ are

$$T_\mu J(i) = \sum_{j=0}^{n} p_{ij}(\mu(i))(g(i,\mu(i),j) + J(j)) \tag{3.9}$$

for $i = 1, ...n$ and $J(0) = 0$.

T^k is viewed as the composition of the mapping T with itself k times; that is, for all k

$$(T^k J)(i) = (T(T^{k-1}J))(i) \tag{3.10}$$

for $i = 1, ...n$ and for $k = 1, 2...$

For $k = 0$

$$(T^0 J)(i) = J(i) \tag{3.11}$$

Similarly the components of $T_\mu^k J$ are defined by

$$(T_\mu^k J)(i) = (T_\mu(T_\mu^{k-1}J))(i) \tag{3.12}$$

for $i = 1, ...n$ and for $k = 1, 2...$

For $k = 0$

$$(T_\mu^0 J)(i) = J(i) \tag{3.13}$$

It can be seen that the $(T_\mu^k J)$ is the cost-to-go of a stationary policy μ for the k-stage stochastic shortest path problem with initial cost i and terminal cost J.

The following proposition gives the main results regarding the above stochastic shortest path problems.

1. The optimal cost-to-go vector J^* has finite components and satisfies

$$J^* = TJ^* \tag{3.14}$$

   Furthermore, J^* is the only solution of the equation $J = TJ$.

2.

$$\lim_{k \to \infty} T^k J = J^* \tag{3.15}$$

   for every vector J

3. A stationary policy μ is optimal if and only if

$$T_\mu J^* = TJ^* \tag{3.16}$$

4. For every policy μ and every vector J, the associated cost-to-go vector J^μ satisfies

$$\lim_{k \to \infty} T_\mu^k J = J^\mu \tag{3.17}$$

   and

$$J^\mu = T_\mu J^\mu \tag{3.18}$$

   J^μ is the only solution of the above equation.

3.4.2 Asynchronous Value Iteration

It is not necessary to maintain a fixed order for iterating on the cost-to-go estimates $J(i)$ of the different states, an arbitrary order can be used, as long as $J(i)$ is iterated infinitely often for each state i. This type of method is called the *asynchronous value iteration method* [Bertsekas & Tsitsiklis, 1996]. In this method an arbitrary initial vector J_0 is chosen and at the k_{th} iteration an index $i_k \in \{1, ..., n,$ is chosen. Replace the i_k^{th} component of the current vector J_k with $(TJ_k)(i_k)$, and leave all other components of J_k unchanged

$$J_{k+1}(i) = \begin{cases} (TJ_k)(i) & if\ i = i \\ J_k(i) & otherwise \end{cases} \tag{3.19}$$

Assuming that all components are chosen for iteration infinitely often, the generated sequence of vectors J_k converges to J^*.

3.4.3 Policy Iteration

In general, value iteration requires an infinite number of iterations to obtain the optimal cost-to-go vector. There is an alternative to value iteration, called *Policy Iteration*, which always terminates finitely. In this algorithm a proper policy μ_0 is chosen and a sequence of policies $\mu_1, \mu_2.....$ is generated. The algorithm consists of two steps.

1. *Policy Evaluation.* Given a policy μ_k, $J^{\mu_k}(i)$, $i = 1, ..., n$ is computed as the solution of the linear system of equations

$$J(i) = \sum_{j=0}^{n} p_{ij}(\mu_k(i))(g(i, \mu_k(i), j) + J(j)), \qquad i = 1, ..., n \tag{3.20}$$

   in the n unknowns $J(1), ...J(n)$.

2. *Policy Improvement* Compute a new policy μ_{k+1} as

$$\mu_{k+1} = arg \min_{u \in U(i)} \sum_{j=0}^{n} p_{ij}(u)(g(i, u, j) + J^{\mu_k}(j)), \qquad i = 1, ..., n \tag{3.21}$$

   The process is repeated with μ_{k+1} used in place of μ_k, unless $J^{\mu_{k+1}}(i) = J^{\mu_k}(i)$ for all i, in which case the algorithm terminates with the policy μ_k.

3.4.4 Asynchronous Policy Iteration

This is a more general policy iteration algorithm, whereby value iterations and policy iterations are executed selectively, for some of the states [Bertsekas & Tsitsiklis, 1996]. This type of algorithm generates a sequence J_k of cost-to-go estimates and a corresponding sequence μ_k of stationary policies. Given a pair $(J_k, \mu_k$, a subset S_k is selected, and the new pair $(J_{k+1}, \mu_{k+1}$ can be generated in two possible ways.

1. Update J_k according to

$$J_{k+1}(i) = \begin{cases} (T_{\mu k} J_k)(i), & if\ i \in S_k \\ J_k(i) & otherwise \end{cases} \tag{3.22}$$

   while the policy is left unchanged by setting $\mu_k = \mu_{k+1}$

2. Update μ_k according to

$$\mu_{k+1}(i) = \begin{cases} (arg\min_{u \in U(i)} \sum_{j=0}^{n} p_{ij}(u)(g(i,u,j) + J^{\mu_k}(j)) & if\ i \in S_k \\ \mu_k(i) & otherwise \end{cases} \tag{3.23}$$

while leave the cost-to-go estimate unchanged by setting $J_k = J_{k+1}$

3.5 Discounted Problems

In this case the definition of the mappings T and T_μ is modified in the absence of termination state. For any vector $J = (J(1), ..., J(n))$, the vector TJ is obtained by applying one iteration of the DP algorithm to J; the components of TJ are

$$TJ(i) = \min_{u \epsilon U(i)} \sum_{j=0}^{n} p_{ij}(u)(g(i,u,j) + \gamma J(j)) \qquad i = 1, ..., n \tag{3.24}$$

Similarly for any vector J and any stationary policy μ, the components of vector $T_\mu J$ are

$$T_\mu J(i) = \sum_{j=0}^{n} p_{ij}(\mu(i))(g(i,\mu(i),j) + \gamma J(j)) \qquad i = 1, ..., n \tag{3.25}$$

The results (3.14) to (3.18) which hold in case of stochastic shortest path problem, also hold in discounted case.

Chapter 4

Simulation Methods for a Lookup Table Representation

4.1 Introduction

The computational methods for dynamic programming described in the previous chapter require an explicit model of the cost structure and the transition probabilities of the system. In some cases such model is unavailable but the model can be simulated using a digital computer. The state space and the action space are known and there is a computer program that simulates for a given action u, the transitional probabilities $p_{ij}(u)$, and calculates the corresponding transitional cost $g(i, u, j)$. In these cases the methods discussed in previous chapter can be applied.

There are some methods which do not require the transition probabilities of the system to be explicitly estimated, but instead in these methods the cost-to-go function of a given policy is progressively calculated by generating several sample trajectories and associated costs. There are mainly two type of methods

1. *Using Look-up Table.* In this case the separate variable $J(i)$ is kept in memory for each state i. These methods cannot be used when the number of states is large, but these methods are applicable when the state space is of moderate size and an exact model is unavailable.

2. *Using Neural Networks.* In this method the J is represented as a function of a smaller set of parameters. These methods are best when the number of states is very large.

These methods are discussed in chapter 6.

Throughout this chapter it is assumed that a stationary and proper policy is fixed. In order to simplify notation the dependence of various quantities on the policy is not shown (for ex. $g(i,j)$ in place of $g(i,a,j)$

4.2 Monte Carlo Method

Monte Carlo (MC) methods are stochastic techniques, meaning they are based on the use of random numbers and probability statistics to investigate problems. MC methods are used in everything from economics to nuclear physics to regulating the flow of traffic. The basic idea in Monte-Carlo simulation is to generate a number of samples $v1, ...v_N$ of the random variable v and then to estimate the mean of v by forming the sample mean

$$M_N = \frac{1}{N}\sum_{k=1}^{N} v_k \tag{4.1}$$

The sample mean can be calculated recursively according to

$$M_{N+1} = M_N + \frac{1}{N+1}(v_{N+1} - M_N) \tag{4.2}$$

The *every visit MC method* estimates $J^{\mu}(i)$ as the average of the costs following all the visits to i in a set of episodes. Within a given episode, the first time i is visited is called the first visit to i. The *first-visit MC* method averages just the costs following first visits to i.

4.2.1 Policy Evaluation by Monte Carlo Simulation

Suppose a number of simulation run, each ending at the termination state 0 are performed. Consider the mth time a given state i_0 is encountered, and let $(i_0, i_1, ...i_k, ..i_N)$ be the remainder of the corresponding trajectory., where $i_N = 0$. Let $c(i_0, m)$ be the corresponding cumulative cost up to reaching state 0

$$c(i_0, m) = g(i_0, i_1) + ... + g(i_{N-1}, i_N) \tag{4.3}$$

By Monte Carlo method $J(i_k)$ is estimated as

$$J(i_k) = \frac{1}{K}\sum_{m=1}^{K} c(i_k, m) \tag{4.4}$$

At the end of each run $J(i_k)$ can be iteratively calculated as

$$J(i_k) = J(i_k) + \alpha_m(c(i_k, m) - J(i_k)), \qquad \alpha_m = \frac{1}{m}, \qquad m = 1, 2, ... \tag{4.5}$$

Above formula can be rewritten as

$$J(i_k) = J(i_k) + \alpha_k(g(i_k, i_{k+1}) + g(i_{k+1}, i_{k+2}) + ... + g(i_{N-1}, i_N) - J(i_k)) \tag{4.6}$$

-for each state $k = 0, ...N - 1$.

4.2.2 On Policy Monte Carlo Control

One way to use on-policy method is to use ϵ-greedy policies. All non-greedy actions are given the minimal probability of selection, $\frac{\epsilon}{|U(i)|}$, and the remaining bulk of the probability, $1 - \epsilon + \frac{\epsilon}{|U(i)|}$, is given to the greedy action. The algorithm for a ϵ-soft on-policy Monte Carlo control algorithm is given below:

Table 4.1: On-Policy Monte Carlo Control Algorithm

Initialize, for all $i \in I$ $u \in U(i)$
$Q(i, u) \leftarrow$ arbitrary
Costs C$(i, u) \leftarrow$ empty list
$\pi \leftarrow$ an arbitrary ϵ-soft policy
Repeat forever:
(a) Generate an episode using π
(b) For each pair i, u appearing in the episode:
g(i,u) $\leftarrow$ cost following first occurrence of i, u
Append g(i,u) to C(i, u)
$Q(i, j) \leftarrow$ average(Costs(i, u))
(c) for each i in the episode:
$u^* \leftarrow arg\min_u Q(i, u)$
for all $u \in U(i)$
$\pi(i, j) \leftarrow \begin{cases} 1 - \epsilon + \frac{\epsilon}{\lvert U(i)\rvert} & if \text{ u=u}^* \\ \frac{\epsilon}{\lvert U(i)\rvert} & if \text{ u} \neq u^* \end{cases}$

The value of ϵ is chosen such that there is a right mix of exploration and exploitation. The costs $C(i, u)$ is the cumulative cost up to reaching state i_N where i_N is the termination state.

$$C(i, u) = g(i, u, i_1) + ... + g(i_{N-1}, i_N) \tag{4.7}$$

4.3 Temporal Difference learning

The idea that is central and novel to RL is *temporal difference* (TD) learning [Sutton & Barto, 1998]. Like Monte Carlo methods, TD methods can learn directly from raw experience without a model of the environment's dynamics. Like DP, TD methods update estimates based in part on other learned estimates, without waiting for a final outcome. Whereas conventional prediction-learning methods are driven by the error between predicted and actual outcomes, TD methods are driven by the error or difference between temporally successive predictions [Tesauro, 1992]. With them, learning occurs whenever there is change in prediction over time.

4.3.1 TD(0) Algorithm

The equation (4.6) can be rewritten as

$$\begin{aligned} J(i_k) = J(i_k) + \alpha((g(i_k, i_{k+1}) + J(i_{k+1}) - J(i_k)) \\ +(g(i_{k+1}, i_{k+2}) + J(i_{k+2}) - J(i_{k+1})).... \\ +(g(i_{N-1}, i_N) + J(i_N) - J(i_{N-1}))) \end{aligned} \tag{4.8}$$

Equivalently,

$$J(i_k) = J(i_k) + \alpha(d_k + d_{k+1} + ...d_{N-1}) \tag{4.9}$$

where the quantities d_k, which are called temporal differences are defined by

$$d_k = g(i_k, i_{k+1}) + J(i_{k+1}) - J(i_k) \tag{4.10}$$

The temporal difference d_k represents the difference between an estimate $g(i_k, i_{k+1}) + J(i_{k+1})$ of the cost-to-go based on the simulated outcome of the current stage, and the current estimate $J(i_k)$.

The lth temporal difference d_l is known as soon as the transition from i_l to i_{l+1} is made. This raises the possibility of carrying out the update incrementally, that is, by setting

$$J(i_k) = J(i_k) + \alpha(d_l), \qquad l = k, ...N - 1 \tag{4.11}$$

If the update is made only after one step, i.e.

$$J(i_k) = J(i_k) + \alpha(g(i_k, i_{k+1}) + J(i_{k+1}) - J(i_k)) \tag{4.12}$$

the resulting algorithm is called TD(0) algorithm. In this the update can be carried out by picking an arbitrary state i_k and only simulating the transition to a next state i_{k+1} rather than an entire trajectory.

4.3.2 N-step TD Prediction

Monte Carlo methods perform a backup for each state based on the entire sequence of observed rewards from that state until the end of episode. On the other hand TD(0) is based on just the next reward. One kind of intermediate method would be to perform a backup based on an intermediate number of rewards: more than one but less than all of them until termination. The methods in which temporal difference extends over n steps are called **n-step TD methods**. An iteration for N-step TD prediction is

$$J(i_k) = J(i_k) + \alpha(g(i_k, i_{k+1}) + ...g(i_{k+n-1}, i_{k+n}) + J(i_{k+n}) - J(i_k)) \tag{4.13}$$

4.3.3 TD(λ) Algorithm

In the absence of any special knowledge about n that could make one value of n preferred over other, it is considered forming a weighted average of all possible n step backups. The TD(λ) algorithm can be understood as one particular way of averaging n-step backups. Each n-step backup is weighted proportional to λ^{n-1}, where $0 \leq \lambda \leq 1$. A normalization factor of $1 - \lambda$ ensures that the weights sum to 1. The resulting equation is

$$J(i_k) = J(i_k) + \alpha(1-\lambda) \sum_{n=1}^{\infty} \lambda^{n-1} (\sum_{m=0}^{n} g(i_{k+m-1}, i_{k+m}) + J(i_{k+n}) \tag{4.14}$$

This equation get simplified to

$$J(i_k) = J(i_k) + \alpha \sum_{m=k}^{\infty} \lambda^{m-k} d_m \tag{4.15}$$

where α is the step size parameter and the temporal difference d_m is given by

$$d_m = g(i_m, i_{m+1}) + J(i_{m+1}) - J(i_m) \tag{4.16}$$

Here each d_m is zero for m greater than N, the termination state. The above approach is called the *forward view* TD(λ) because for each state visited one look forward in time to all the future rewards and decide how best to combine them. After looking forward from and updating one state, one move onto the next and never has to work with the preceding state again. If $\lambda = 1$ then the resulting algorithm is called **TD(1)** algorithm which is same as the Monte Carlo policy evaluation method described earlier.

4.3.4 Advantages of TD Prediction Methods

- TD methods are more incremental and therefore easier to compute. For example, the TD method for predicting Saturday's weather can update each day's prediction on the following day, whereas the conventional method must wait until Saturday, and then make the changes for all the days of the week [Mahadevan, Sridhar & Connell, 1992].

- TD methods tend to make more efficient use of their experience: they are usually converge faster and produce better predictions.

- TD methods have an advantage over classical DP methods in that they do not require a model of the environment, of its reward and next-state probability distributions.

4.3.5 Every-Visit and First Visit Variants

If a series is visited more than once by the same trajectory than under the same trajectory, then under the every-visit variant, the update is to be carried out more than once, with each update involving the summation of $\lambda^{m-k} d_m$ over the portion of the trajectory that follows the visit under consideration. Let in a single trajectory, state i is visited a total of M times i.e. at $m_1, m_2, ...m_M$. Then the total update is given by

$$J(i_k) = J(i_k) + \alpha \sum_{j=1}^{M} \sum_{m=m_j}^{\infty} \lambda^{m-m_j} d_m \tag{4.17}$$

whereas the update in case of first visit variant is given by

$$J(i_k) = J(i_k) + \alpha \sum_{m=m_1}^{\infty} \lambda^{m-m_1} d_m \tag{4.18}$$

4.3.6 Off-Line and On-Line Variants

In the most straightforward implementation of $TD(\lambda)$, all of the updates are carried out simultaneously, after the entire trajectory has been simulated. This is called the *Off-line* version of the algorithm. In an alternative procedure, called the *On-line* version, the updates are made one term at a time, following each transition.

$$\begin{array}{ll} J(i_0) = J(i_0) + \alpha d_0 & following\,the\,transition\,(i_0, i_1) \\ \left\{ \begin{array}{c} J(i_0) = J(i_0) + \alpha\lambda d_1 \\ J(i_1) = J(i_1) + \alpha d_1 \end{array} \right\} & following\,the\,transition\,(i_1, i_2) \end{array} \tag{4.19}$$

4.3.7 TD(λ) for Discounted Problems

Discounted problem can be handled in two different ways. The first approach is to convert a discounted problem into an equivalent stochastic shortest path problem. This conversion amounts to viewing the discount factor γ as a continuation probability. At each stage a transition is simulated and process is terminated with probability $1-\gamma$. With this approach, every simulated trajectory is of finite duration with probability 1. Its duration is random and can be arbitrarily large. With this approach the Eq. (4.15) can be rewritten as

$$J(i_k) = J(i_k) + \alpha \sum_{m=k}^{\infty} (\lambda\gamma)^{m-k} d_m \tag{4.20}$$

And the temporal difference d_m is given by

$$d_m = g(i_m, i_{m+1}) + \gamma J(i_{m+1}) - J(i_m) \tag{4.21}$$

The other method is to use Backward view of TD(λ) which is discussed in next section.

4.3.8 Using Eligibility Traces

The idea behind the eligibility traces is very simple. Each time a state is visited it intimates a short term memory process, a trace, which then decays gradually over time. This trace marks the state as eligible for learning [Singh & Sutton 1996]. When a TD error occurs, only the eligible states or actions are assigned credit or blame for the error.

For a single infinitely long trajectory, one cannot afford to wait until the end of the trajectory in order to carry out update, the on-line version of the algorithm is used. It is

also essential that the step size α be gradually reduced to zero during the course of infinitely long simulation. The update equation for this case is

$$J_{m+1}(i) = J_m(i) + \alpha_m(i) e_m(i) d_m(i) \tag{4.22}$$

where $e_m(i)$ is eligibility trace and $d_m(i)$ is given by Eq. (4.21). There are two methods for determining eligibility traces in this case

1. In the every-visit TD(λ) method eligibility traces for all states decay by $\gamma\lambda$, and the eligibility trace for the one state visited on the step is incremented by 1:
$$e_m(i) = \begin{cases} \gamma\lambda e_{m-1}(i) & if\ \mathrm{i}_m \neq i \\ \gamma\lambda e_{m-1}(i) + 1 & if\ \mathrm{i}_m = i \end{cases} \tag{4.23}$$

2. In a replacing eligibility traces method each time the state is visited the trace is reset to 1 regardless of the presence of a prior trace.

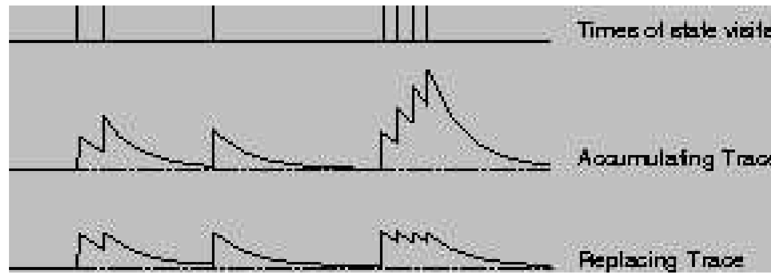

Figure 4.1: Accumulating and replacing traces [Singh & Sutton 1996]

$$e_m(i) = \begin{cases} \gamma\lambda e_{m-1}(i) & if\ \mathrm{i}_m \neq i \\ 1 \ \text{ if } \ i_m = i \end{cases} \tag{4.24}$$

Although replacing traces are only slightly different from accumulating traces, they can produce a significant improvement in learning rate.

4.4 Q-Learning

Q-learning is the on-line stochastic method for learning the optimal policy through experience gained solely on the basis of the samples of the form

$$s_n = (i_n, u_n, j_n, g_n) \tag{4.25}$$

where n denotes discrete time, and each sample s_n is described by the four-tuple consists of a trial action/control u_n on state i_n that results in a transition to state $j_n = i_{n+1}$ at a cost $g_n = g(i_n, u_n, j_n)$. Q-learning is highly suitable for solving Markovian decision problems without explicit knowledge of the transition probabilities. The Bellman optimality equation for Optimal action value function is given by (refer Eq. 3.3)

$$Q^*(i,u) = \sum_{j=1}^{N} p_{ij}(u)(g(i,u,j) + \gamma \min_{b \in U(j)} Q^*(j,b)) \qquad for\ all\ (i,u) \tag{4.26}$$

Using the value iteration algorithm described in chapter 3, the Q(i,u) can be computed by successive iterations. The one iteration of algorithm would look like

$$Q(i,u) = \sum_{j=1}^{N} p_{ij}(u)(g(i,u,j) + \gamma \min_{b \in U(j)} Q(j,b)) \qquad for\ all\ (i,u) \tag{4.27}$$

The small step size version of this iteration is described by

$$Q(i,u) = (1-\alpha)Q(i,u) + \alpha \sum_{j=1}^{N} p_{ij}(u)(g(i,u,j) + \gamma \min_{b \in U(j)} Q(j,b)) \qquad for\ all\ (i,u) \tag{4.28}$$

Here,α as usual is a small learning rate parameter that lies in the range $0 < \alpha < 1$.

An iteration of the algorithm described above requires knowledge of the transition probabilities. The above need can be eliminated by formulating a stochastic version of the equation. Specifically the averaging performed in an iteration over all possible state is replaced by a single sample, thereby resulting in the following update for the Q-factor:

$$Q_{n+1}(i,u) = (1-\alpha_n)Q_n(i,u) + \alpha_n(g(i,u,j) + \gamma \min_{b \in U(j)} Q_n(j,b)) \quad for\ (i,u) = (i_n, u_n) \tag{4.29}$$

and for all other state action pairs

$$Q_{n+1}(i,u) = Q_n(i,u) \qquad for\ (i,u) \neq (i_n, u_n \tag{4.30}$$

Here α_n is shown as a function of n. Generally value of α_n diminishes as n increases. If we define $\Delta Q_n(i,u) = Q_{n+1}(i,u) - Q_n(i,u)$, then

$$\Delta Q_n(i,u) = \begin{cases} \alpha_n[(g(i,u,j) + \gamma \min_{b \in U(j)} Q_n(j,b) - Q_n(i,u)] & for\ (i,u) = (i_n, u_n) \\ 0 & for\ (i,u) \neq (i_n, u_n) \end{cases} \tag{4.31}$$

4.5 Optimistic Policy Iteration (Actor-Critic Methods)

In this type of algorithm policy updates are made on the basis of only an incomplete evaluation of the current policy. These methods in many aspects similar to the asynchronous policy iteration algorithm described in chapter 3. Policy updates are performed more frequently, without waiting for the policy evaluation algorithm to converge. These algorithms have two learning units: an actor and a critic. An actor is a decision maker with a tunable parameter. A critic is a function approximator. The critic tries to approximate the value function of the policy used by the actor, and the actor in turn tries to improve its policy based on the current approximation provided by the critic. With each communication an update in policy is made. Under the assumption that the critic communicates to the actor an infinite number

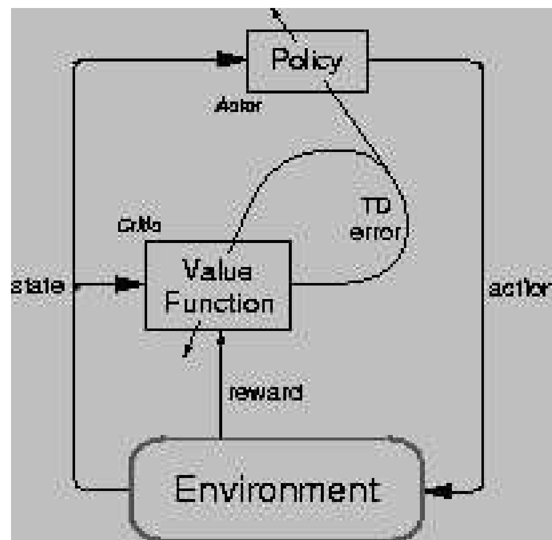

Figure 4.2: The Actor-Critic Architecture [Sutton & Barto, 1998]

of times, if the sequence of policies μ converges, then the limit must be an optimal policy. If μ converges, then the critic vector J must converge to J^{μ}. The vectors J received by the actor also converge to J^{μ}. Since the policy converges to μ, thus $T_{\mu}J^{\mu} = J^{\mu}$ and therefore J^{μ} is a solution of Bellman's equation, proving that $J^{\mu} = J^{*}$.

After each action selection, the critic evaluates the new state to determine whether things have gone better or worse than expected. That evaluation is the TD error:

$$d_k = g(i_k, i_{k+1}) + \gamma J(i_{k+1}) - J(i_k) \tag{4.32}$$

If the TD error is positive, it suggests that the tendency to select the state j should be strengthened for the future, whereas if the TD error is negative it suggests the tendency should be weakened. If $p_{i,j}$ is the transition probability then the strengthening or weakening can be implemented simply by incrementing or decrementing p_{ij} by

$$p_{ij} \leftarrow p_{ij} + \beta d_k \tag{4.33}$$

where β is any positive step size parameter.

Chapter 5

Neural Networks

For many important problems computational requirements of reinforcement learning algorithm is very large. In such a problem there is a need to construct approximate representations of the optimal cost-to-go function or other functions of interest, because this is the only possible method for breaking the curse of dimensionality in the face of a very large state space. Neural networks are one such function approximators that are best suited in Reinforcement learning context.

5.1 What is a Neural Network?

An artificial neural network (ANN) is an information-processing paradigm inspired by the way the densely interconnected, parallel structure of the mammalian brain processes information. The network is usually implemented by using electronic components or is simulated in software or a digital computer. An artificial neural network resembles the brain in two aspects.

- Knowledge is acquired by the network from its environment through a learning process.
- Interneuron connection strengths, known as synaptic weights, are used to store the acquired knowledge. The procedure used to perform the learning process is called a learning algorithm.

5.1.1 Structure of the Nervous System

In the simplest sense the nervous system consists of neurons that are connected to each other in a rather complex way. Each neuron can be thought of as a node and the interconnections between them are edges. Synapses are elementary structural and functional unit that mediate the interconnections between neurons.

5.1.2 Models of a Neuron

There are three basic elements of the neuronal model [Haykin, 2001]

1. A set of synapse or connecting links, each of which is characterized by a weight or strength of its own. A signal x_j at the input of synapse j connected to neuron k is multiplied by the synaptic weight w_{kj}. The synaptic weight of an artificial neural network can take both positive and negative values.

2. An adder for summing the input signals, weighted by the respective synapses of the neuron.

Figure 5.1: Nonlinear model of a neuron

3. An activation function for limiting the amplitude of the output of a neuron. The activation function is also referred to as a squashing function in that it squashes the permissible output range of the output signal to some finite value. The neural network may include an externally applied bias, denoted by b_k. The bias b_k has the effect of increasing or decreasing the net input of the activation function, depending upon whether it is positive or negative, respectively.

   In mathematical terms following equations can be written

$$u_k = \sum_{j=1}^{m} w_{kj} x_j \tag{5.1}$$

   and

$$y_k = \phi(u_k + b_k) \tag{5.2}$$

   where $x_1, x_2, ...x_m$ are the input signals; $w_{k1}, w_{k2}, ...w_{km}$ are the synaptic weights of neuron k; u_k is the linear combiner output due to input signals; b_k is the bias; $\phi(\cdot)$ is the activation function; and y_k is the output signal of the neuron.

5.2 Types of Activation Function

There are three basic type of Activation function.

1. *Threshold Function* This function is defined as:

$$\phi(v) = \begin{cases} 1 \; if \; v \geq 0 \\ 0 \; if \; v < 0 \end{cases} \tag{5.3}$$

   Correspondingly, the output of neuron k employing such a threshold function is expressed as:

$$y_k = \begin{cases} 1 \; if \; v \geq 0 \\ 0 \; if \; v < 0 \end{cases} \tag{5.4}$$

2. *Piecewise-linear Function* For the piecewise-linear function we have:

$$\phi(v) = \begin{cases} 1 & v \geq \frac{1}{2} \\ v & \frac{1}{2} > v > -\frac{1}{2} \\ 0 & v \leq -\frac{1}{2} \end{cases} \tag{5.5}$$

Figure 5.2: Threshold function

Figure 5.3: Piecewise-linear function

3. *Sigmoidal Function* The sigmoidal function is one the most popular activation function used in the construction of artificial neural networks. It is defined as a strictly increasing function that exhibits a graceful balance between linear and nonlinear behavior. The function is differentiable and has the following property

$$-\infty < \lim_{\xi \to -\infty} \sigma(\xi) < \lim_{\xi \to \infty} \sigma(\xi) < \infty$$

Some common choices of sigmoidal function are the hyperbolic tangent function

$$\sigma(\xi) = tanh(\xi) = \frac{e^{\xi} - e^{-\xi}}{e^{\xi} + e^{-\xi}} \tag{5.7}$$

Figure 5.4: Sigmoidal Function for varying slope parameter$a$

and the logistic function

$$\sigma(\xi) = \frac{1}{1 + e^{-\xi}} \tag{5.8}$$

5.3 Feedback

Feedback is said to br exist in a dynamic system whenever the output of an element in the system influences in part the input applied to that particular element, thereby giving rise to one or more closed paths for the transmission of signals around the system. Feedback plays a major role in the study of a special class of neural networks known as recurrent networks.

Figure 5.5: sigle-loop feedback system

The above figure shows the signal flow graph of single-loop feedback system where the input signal $x_j(n)$, internal signals $x_j^{'}(n)$ and output signal $y_k(n)$ are functions of the discrete-time variable n. The system is assumed to be linear, consisting of a forward path and a feedback path that are characterized by the operators A and B, respectively. The system has the

following input-output relationships:

$$y_k(n) = A[x_j'(n)] \tag{5.9}$$

$$x_j'(n) = x_j(n) + B[y_k(n) \tag{5.10}$$

Eliminating $x_j'(n)$ from the above equations, one can get

$$y_k(n) = \frac{A}{1 - AB}[x_j(n)] \tag{5.11}$$

$A/(1 - AB)$ is referred as the closed loop operator and AB as the open loop operator of the system.

5.4 Architectures for Approximation

The first step in the development of an approximate representation of the optimal cost-to-go functions is to choose an approximation architectures, that is a certain functional form involving a number of free parameters (synaptic weights). These parameters are to be tuned so as to provide a best fit of the function to be approximated. The process of tuning these parameters are often referred to as *learning* or *training*.

There are two important preconditions for the development of an effective approximation. First, an approximation architecture, rich enough to provide an acceptable close approximation of the function. Second an effective algorithm for tuning the parameters of the approximation algorithm. Having a rich set of approximating functions usually means that there is large number of parameters to be tuned or that the dependence on the parameter is nonlinear.

In Supervised learning or learning by example, the objective is to construct a function $y = f(x)$, given some training data pairs (x_i, y_i) that best explains these data pairs. In the context of reinforcement learning, one is interested in approximating the optimal cost-to-go function J^*, an ideal set of training data would consist of pairs $(i, J^*(i))$, where i ranges over some subset of state space. However, in case of reinforcement learning the function J^* is neither known nor it can be measured experimentally, and therefore such training data are

unavailable. As a result RL is faced with an additional difficulty as compared to classical uses of neural networks.

The situation become easier if there is a single policy π because the expected cost-to-go function $J^{\pi}(i)$ under that policy can be estimated by setting the initial state to i and simulating the system under that policy π. For this reason most of the popular RL algorithm focus on a single policy at a time and try to estimate the cost-to-go function of this policy by using simulation. Then, on occasion, they switch to another policy that appears more promising on the basis of the simulation data obtained.

5.5 Linear Architecture

Let $J : S \to \Re$, where S is the state space, is the function to be approximated and r be a vector of parameters (weights). Now the cost to go function $J(i)$ is approximated by $\tilde{J}(i, r)$. Here $\tilde{J}$ is known function which is easy to compute once the vector r is fixed. The objective is to choose the parameter vector r such that the distance between the function J and the approximant $\tilde{J}$ is minimized.

A linear architecture is of the general form

$$\tilde{J}(i, r) = \sum_{k=0}^{K} r(k)\phi_k(i) \tag{5.12}$$

where $r(k)$, $k = 0, 1....K$, are the components of the parameter vector r and ϕ_k are fixed, easily computable functions. It is common to use the constant function $\phi_0 \equiv 1$ as a basis function in the linear architectures. So the above equation take the following form:

$$\tilde{J}(i, r) = r(0) + \sum_{k=1}^{K} r(k)\phi_k(i) \tag{5.13}$$

This allows a tunable offset from zero for the approximation $\tilde{J}(i, r)$, and enlarge the range of mappings that the linear architecture can effectively approximate.

Let there is some training data pairs $(i, J(i))$ which is to be fit using a linear architecture. Now the problem is similar to minimize the square of error between training data and approximation.

$$\sum_{i}(J(i) - \sum_{k} r(k)\phi_k(i))^2 \tag{5.14}$$

over all vectors r. Several algorithm are available for solving this problem.

5.6 Non linear Architecture

In case of *nonlinear architecture*, the dependence of $\tilde{J}(i,r)$ on r is non linear. In this case problem of minimizing the square error

$$\sum_i (J(i) - \tilde{J}(i,r))^2 \tag{5.15}$$

cannon be reduced to a linear algebraic problem.

A common nonlinear architecture is the multilayer perceptrons or *feedforward neural network* with a single hidden layer. Under this architecture state i is encoded as a vector x with components $x_l(i), l = 1, ..., L$, which is then transformed linearly through a linear layer involving the coefficients $r(k,l)$, to give K scalers

$$\sum_{l=1}^{L} r(k,l)x_l(i) \tag{5.16}$$

where $k = 1, ..., K$

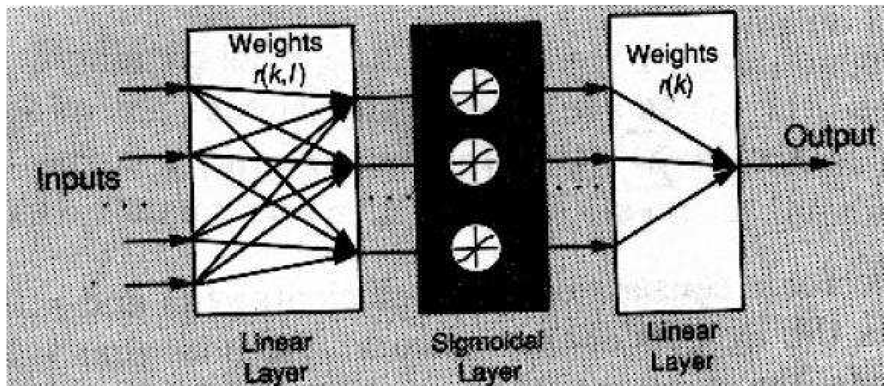

Figure 5.6: A feedforward neural network with a single hidden layer and a single output

Each of these scalers become the input to a sigmoidal function $\sigma(\cdot)$. At the outputs of the sigmoidal functions, the scalers

$$\sigma(\sum_{l=1}^{L} r(k,l)x_l(i)) \tag{5.17}$$

where $k = 1, ..., K$

are obtained. These scalers are linearly combined using coefficients $r(k)$ to produce the final output

$$J(i,r) = \sum_{k=1}^{K} r(k)\sigma(\sum_{l=1}^{L} r(k,l)x_l(i)) \tag{5.18}$$

The parameter vectors r which consists of the coefficients $r(k)$ and $r(k,l)$, are also known as the *weights* of the network. It is common practice to provide the constant 1 as an additional input to the linear layer, or equivalently to fix one of the components of x at the value 1. Without such a bias, in the case where $\sigma(0) = 0$, the approximation $J(i,r)$ provided by the above equation would be forced to the value 0 when $x = 0$.

5.7 Back-propagation Algorithm

A multilayer perceptrons with many hidden layer can be represented by introducing certain mappings to describe the linear and the sigmoidal layers. In particular, let $L_1, \ldots. L_{m+1}$ denote the matrices representing the linear layers; that is the output of the 1st linear layer is the vector $L_1 x$ and the output of the kth linear layer $(k > 1)$ is $L_k \xi$, where ξ is the output of the preceding sigmoidal layer. Similarly, let $\sum_1, \ldots.., \sum_m$ denote the mapping representing the sigmoidal layers; that is when the input of the kth sigmoidal layer $(k > 1)$ is the vector y with components $y(j)$, the output is the vector $\sum_k y$ with components $\sigma(y(j))$. Thus the output of the multilayer perceptrons with m hidden sigmoidal layers is

$$F(L_1, \ldots., L_{m+1}, x) = L_{m+1} \sum_m L_m \ldots. \sum_1 L_1 x \tag{5.19}$$

The squared error between the output and a desired output y is

$$E(L_1, \ldots., L_{m+1}) = \frac{1}{2}(y - F(L_1, \ldots., L_{m+1}, x))^2 \tag{5.20}$$

Now the gradient of the error with respect to the weights can be efficiently calculated by using a special procedure known as $back - propagation$, which is just an intelligent way of using the chain rule. In particular, the partial derivative of the error function $E(L_1, \ldots, L_{m+1})$ with respect to $L_k(i,j)$ the ij^{th} component of the matrix L_k is given by

$$\frac{\partial E(L_1, \ldots, L_{m+1}}{\partial L_k(i,j)} = -eL_{m+1} \overline{\sum}_m L_m \ldots.. L_{k+1} \overline{\sum}_k I_{ij} \sum_{k-1} L_{k-1} \ldots. \sum_1 L_1 x \tag{5.21}$$

where e is the error vector

$$e = y - F(L_1, ..., L_{m+1}, x) \tag{5.22}$$

$\overline{\sum}_n, n = 1, ...m$ is the diagonal matrix with the diagonal terms equal to the derivatives of the sigmoidal function σ of the nth hidden layer evaluated at the appropriate points, and I_{ij} is the matrix obtained from L_k by setting all of its components to 0 except for the ij^{th} component which is set to 1.

The correction $\Delta L_k(i,j)$ applied to $L_k(i,j)$ is defined as

$$\Delta L_k(i,j) = -\eta \frac{\partial E(L_1, ..., L_{m+1}}{\partial L_k(i,j)} \tag{5.23}$$

where η is the *learning rate parameter* of the back-propagation algorithm.

5.7.1 The Two Phases of Computation

In the application of the back propagation algorithm two distinct passes of computation are distinguished. The first pass is referred to as the forward pass, and the second is referred to as the backward pass.

1. *Forward Pass* In the forward pass the synaptic weights remain unaltered through out the network. The output of the linear layers $L_1 x, L_2 \sum_1 L_1 x, ..., L_{m+1} \sum_m L_m ..... \sum_1 L_1 x$ is calculated sequentially in order to obtain the error e.

2. *Backward Pass* In the backward pass the synaptic weights are change. It starts at the output layer by passing the error signals leftward through the network, layer by layer. The terms $eL_{m+1}\overline{\sum}_m L_m .....L_{k+1}\overline{\sum}_k$ in the derivative formula are calculated sequentially starting with $eL_{m+1}\overline{\sum}_m$, proceeding to $eL_{m+1}\overline{\sum}_m L_m \overline{\sum}_{m-1}$, and continuing to $eL_{m+1}\overline{\sum}_m .....L_2 \sum_1$.

5.7.2 Differentiation of Sigmoidal Activation Function

1. *Logistic Function* This form of sigmoidal nonlinearity in its general form is defined by

$$\sigma(\xi) = \frac{1}{1 + e^{-a\xi}} \tag{5.24}$$

where $a > 0$ and $-\infty < \sigma(\xi) < \infty$. According to this linearity, the amplitude of the output lies in the range $0 \leq y_j \leq 1$. Diffrentiating the above equation with respect to ξ

$$\sigma^{'}(\xi) = \frac{ae^{-a\xi}}{1 + e^{-a\xi}{}_2} \tag{5.25}$$

As $y = \sigma(\xi)$ the exponential term may be eliminated from the equation

$$\sigma^{'}(\xi) = ay[1 - y] \tag{5.26}$$

Now the derivative $\sigma^{'}(\xi)$ attains its maximum value at $y = 0.5$, and its minimum at value (zero) at y=0 and y=1. Since the amount of change in the synaptic weight of the network is proportional to the derivative $\sigma^{'}(\xi)$ by Eq. (1.21) and (1.21), it follows that for a sigmoid activation function the synaptic weight are changed the most for those neurons in the network where the functions are in their midrange. It is this feature of back-propagation learning that contributes to its stability as a learning algorithm.

2. *Hyperbolic Tangent Function* Another common sigmoidal function is the hyperbolic tangent function which is defined as

$$\sigma(\xi) = atanh(b\xi) = a\frac{e^{b\xi} - e^{-b\xi}}{e^{b\xi} + e^{-b\xi}} \tag{5.27}$$

Diffrentiating the equation with respect to ξ

$$\begin{aligned} \sigma^{'}(\xi) &= absech^2(b\xi) \\ &= ab(1 - tanh^2(b\xi)) \\ &= \frac{b}{a}[a - y][a + y] \end{aligned} \tag{5.28}$$

5.8 Steepest Gradient Method

In most of the problems of neural network training the objective is to find the set of parameters or weights of an approximate architecture that provide the best fit between the given

set of input/output data pairs. Typically these problems are of the least square form

$$minimize f(r) = \sum_i (J(i) - \tilde{J}(i,r))^2 \tag{5.29}$$

Here $J(i)$ is the desired cost-to-go function and $\tilde{J}(i,r)$ is its approximation. The iterative steepest gradient method is used when the function f is continuously differentiable function. The algorithm start at some point r_0 (an initial guess) and then successively generate vectors $r_1, r_2, ...$ according to the formula

$$r_{t+1} = r_t + \alpha_t s_t, \qquad t = 0, 1, ... \tag{5.30}$$

where α_t is a positive stepsize, and the direction s_t is the steepest descent direction given by

$$s_t = -\nabla f(r_t) \tag{5.31}$$

Chapter 6

Function Approximation using Neural networks

In this chapter the cost to go function J^μ of a given policy μ, the optimal cost-to-go function J^*, or the Q-factors $Q^*(i,u)$ are approximated. This is done using a function which, given a state i, produces an approximation $\tilde{J}(i,r)$ of $J^\mu(i)$ or given also a control u, an approximation $\tilde{Q}(i,u,r)$ of $Q^*(i,u)$. The approximating function involves a tunable parameter vector r, and is implemented using a neural network.

The lookup table representation considered in the chapter 4 can be viewed as a limiting form of an approximate representation. If the dimension of the parameter vector r is the same as the number of non-terminal states and if $\tilde{J}(i,r) = r(i)$ for all i, then maintaining the values of the parameter vector r is same as keeping the values $J(i)$ in the lookup table.

6.1 Approximate TD(1)

As discussed in chapter 4 the TD(1) algorithm is similar to the Monte Carlo method. Here an approximate cost-to-go function $\tilde{J}(i,r)$ is used to approximate $J^\mu(i)$, where r is a vector of tunable parameters. The problem considered is a stochastic shortest path problem, with 0 being a cost-free absorbing state. Consistent with the convention of fixing $J(0)$ to zero, $\tilde{J}(0,r)$ is assumed to be 0.

As discussed in the chapter 4, in Monte-Carlo method an initial state i_0 is fixed and a sample trajectory $(i_0, i_1, ...i_N)$ is generated , where i_N is the first time the state 0 is reached.

The objective is to minimize the square error by choosing the proper parameter vector r. The square error is given by

$$Sq.error = \sum_{k=0}^{N-1} (\tilde{J}(i_k, r) - \sum_{m=k}^{N-1} g(i_m, i_{m+1}) \tag{6.1}$$

The proper value of r can be attained by using an incremental gradient update method

$$r = r - \alpha \sum_{k=0}^{N-1} \nabla \tilde{J}(i_k, r)\Big(\tilde{J}(i_k, r) - \sum_{m=k}^{N-1} g(i_m, i_{m+1})\Big) \tag{6.2}$$

There is also a temporal difference implementation of the incremental gradient iteration. define temporal differences d_k by

$$d_k = g(i_k, i_{k+1}) + \tilde{J}(i_{k+1}, r) - \tilde{J}(i_k, r), \qquad k = 0, ..., N-1 \tag{6.3}$$

The Eq. (6.2) now becomes

$$r = r + \alpha \sum_{k=0}^{N-1} \nabla \tilde{J}(i_k, r)\Big(d_k + d_{k+1} + ... + d_{N-1}\Big) \tag{6.4}$$

Instead of waiting for the end of trajectory to update r the part of update involving d_k may be performed as soon as d_k become available. This lead to the update equation

$$r = r + \alpha d_k \sum_{m=0}^{k} \nabla \tilde{J}(i_m, r) \qquad k = 0, 1, ...N-1 \tag{6.5}$$

For example following the state transition (i_0, i_1)

$$r = r + \alpha d_0 \nabla \tilde{J}(i_0, r) \tag{6.6}$$

and following the state transition (i_1, i_2)

$$r = r + \alpha d_1 (\nabla \tilde{J}(i_0, r) + \nabla \tilde{J}(i_0, r)) \tag{6.7}$$

6.2 Approximate TD(λ) for general λ

In the off-line version of TD(λ), a trajectory $i_0, ...i_N$ is simulated and afterwards the parameter vector r is updated by letting

$$r = r + \alpha \sum_{m=0}^{N-1} \nabla \tilde{J}(i_m, r) \sum_{k=m}^{N-1} d_k \lambda^{k-m} \tag{6.8}$$

In the on-line version, the terms involving d_k are used for a partial update of r as soon as d_k becomes available. r values are updated following the state transition (i_k, i_{k+1}) as

$$r = r + \alpha d_k \sum_{m=0}^{k} \lambda^{k-m} \nabla \tilde{J}(i_m, r) \qquad for\ k = 0, ...N-1 \tag{6.9}$$

For discounted problems, the TD(λ) update rule is almost the same as for the undiscounted case. The temporal difference d_k is modified as

$$d_k = g(i_k, i_{k+1}) + \gamma \tilde{J}(i_{k+1}, r) - \tilde{J}(i_k, r) \tag{6.10}$$

The online updating rule following the transition from i_k to i_{k+1} becomes

$$r = r + \alpha d_k \sum_{m=0}^{k} (\gamma\lambda)^{k-m} \nabla \tilde{J}(i_m, r) \qquad for\ k = 0, ...N-1 \tag{6.11}$$

In the absence of an absorbing termination state, the trajectory never terminates and the entire algorithm involves a single infinitely long trajectory. In this case it is necessary to gradually reduce α towards 0 as the algorithm progresses.

6.3 Approximate Q-learning

The approximate Q-learning algorithm using neural network is described step-by-step as [Haykin, 2001]

1. Start with an initial vector r_0, resulting in the Q-factor $Q(i_0, u_0, r_0)$; the weight vector refers to the neural network used to perform the approximation.

2. For iteration $n = 1, 2, ...$ do the following

    (a) For the setting r_n of the neural network, determine the optimal action

    $$u_n = \min_{u \in U(i_n)} Q_n(i_n, u, r_n) \tag{6.12}$$

    (b) Determine the target Q-factor

    $$Q_n^{target}(i_n, u_n, r_n) = g(i_n, u_n, j_n) + \gamma \min_{b \in U(j_n)} Q_n(j_n, b, r_n) \tag{6.13}$$

(c) Update the Q factor

$$Q_{n+1}(i_n, u_n, r_n) = Q_n(i_n, u_n, r_n) + \Delta Q_n(i_n, u_n, r_n) \tag{6.14}$$

where

$$\Delta Q_n(i_n, u_n, r_n) = \begin{cases} \alpha_n(i_n, u_n)(Q_n^{target}(i_n, u_n, r_n) - Q_n(i_n, u_n, r_n)) & for\ (i,u) = (i_n, u_n) \\ 0 & for\ (i,u) \neq (i_n, u_n) \end{cases} \tag{6.15}$$

(d) Apply (i_n, u_n) as the input to the neural network producing the output $\tilde{Q}(i_n, u_n, r_n)$ as an approximation to the $Q_n^{target}(i_n, u_n, r_n)$. Change the weight vector r_n slightly in a way that brings $\tilde{Q}(i_n, u_n, r_n)$ closer to the $Q_n^{target}(i_n, u_n, r_n)$ according to the following formula

$$r_{n+1} = r_n + \alpha_n \nabla \tilde{Q}(i_n, u_n, r_n)\Big(Q_n^{target}(i_n, u_n, r_n) - \tilde{Q}(i_n, u_n, r_n)\Big) \tag{6.16}$$

(e) $i_{n+1} = j_n$. Go back to step and (a) and repeat the computation

Chapter 7

Applications of Reinforcement Learning Algorithms

7.1 Introduction

The range of application of Reinforcement learning is very large. It is used as a powerful tool for solving different types of problems which can be formulated into goal directed problems such as Scheduling (Job-Shop Scheduling, elevator Scheduling) problems, Game Playing (TD gammon, Samuel's Checkers player), Dynamic Channel allocation, Control (Robot navigation) and Optimization problems. Several of these are substantial applications of potential economic significance.

The two applications of Reinforcement learning, Elevator Scheduling and Solving Traveling Salesman problem are discussed in detail in the next two chapters. Some other applications of Reinforcement learning developed by researchers are presented in the next action very briefly.

7.2 Job-shop Scheduling

Many jobs in industry and elsewhere require completing a collection of tasks while satisfying constraints like some tasks have to be finished before others can be started or two tasks requiring the same resource cannot be done simultaneously. The objective is to create a schedule specifying when each task is to begin and what resources it will use that satisfies all the constraints while taking as little overall time as possible. This is the job-shop scheduling

problem. There is probably no efficient procedure for exactly finding shortest schedules for arbitrary instances of the problem.

Zhang and Dietterich [Zhang and Dietterich, 1995] present a reinforcement learning approach to the problem. They applied the TD(λ) with value iteration to train a neural network to learn a heuristic evaluation function over state. This evaluation function used by Function approximation was by a multi-layer neural network trained by backpropagating TD errors. Actions were selected by an ϵ -greedy policy, with ϵ decreasing during learning. One-step lookahead search was used to find the greedy action.

7.3 Robot Control

Reinforcement learning in comparison to other methods have several advantages when applied to robot control tasks:

- Reinforcement learning is incremental. Thus the robot is continually improving its performance as it learns.
- Supplying the robot with a suitable reward function turns out to be relatively straightforward.
- It does not require supplying the robot with a theory of its domain, as is required by explanation-based learning, which would be a substantial undertaking in any real world robotics task.

On the other hand reinforcement learning is slow to converge, requiring several thousand instances.

Various RL algorithms such as Q- learning, Optimistic Policy Iteration, TD(λ), and Sarsa(λ) are being applied to robot control problem.

7.4 TD-Gammon

One of the most impressive applications of reinforcement learning to date is that by Gerry Tesauro's to the game of backgammon. The game is played with 15 white and 15

black pieces on a board of 24 locations, called points. The number of possible states for the game is approximately estimated to be around 10^{22}.

TD-Gammon is a neural network that trains itself to be an evaluation function for the game of backgammon by playing against itself and learning from the outcome [Tesauro, 1992]. While it may seem that forgoing the tutelage of human masters places TD-Gammon at a disadvantage, it is also liberating in the sense that the program is not hindered by human biases or prejudices that may be erroneous or unreliable. TD-Gammon used a nonlinear form of TD(λ) To implement the value function, TD-Gammon used a standard multi-layer neural network. TD-Gammon 2.1 plays at a strong master level that is extremely close (within a few hundredths of a point) to equaling the world's best human players.

Chapter 8

Elevator Scheduling using Reinforcement Learning

Scheduling Elevators in a multi story building is very difficult. This is because the elevator systems operate in high dimensional continuos state spaces and in continuos time as discrete event dynamic systems. Their states are not fully observable and they are non-stationary due to changing passenger arrival rates. The state space of a typical elevator scheduling problem is very large.

A team of Reinforcement learning agents, each of which is responsible for controlling one elevator car is used to solve the problem. Each agent uses artificial neural networks to store its action value estimates. The algorithm used is Approximate Q-Learning, the neural network is trained by error back-propagation algorithm.

8.1 Problem Formulation

8.1.1 Passenger Arrival Patterns

Since its not possible to work on a real elevator system, the passenger arrival patterns should be known. Arrival patterns vary during the course of the day. In a typical office building, the morning rush hour brings a peak level of up traffic, while a peak in down traffic occurs during the afternoon. Other parts of the day have their own characteristic patterns. Up-peak and down-peak elevator traffic are not simply equivalent patterns in opposite directions. Down-peak traffic has many arrival floors and a single destination, while up-peak traffic has

a single arrival floor and many destinations. This distinction has significant implications. For example, in light up traffic, the average passenger waiting times can be kept very low by keeping idle cars at the lobby where they will be immediately available for arriving passengers. In light down traffic, waiting times will be longer since it is not possible to keep an idle car at every upper floor in the building, and therefore additional waiting time will be incurred while cars move to service hall calls. The situation is reversed in heavy traffic. In heavy up traffic, each car may fill up at the lobby with passengers desiring to stop at many different upper floors. The large number of stops will cause significantly longer round-trip times than in heavy down traffic, where each car may fill up after only a few stops at upper floors. For this reason, down-peak handling capacity is much greater than up-peak capacity.

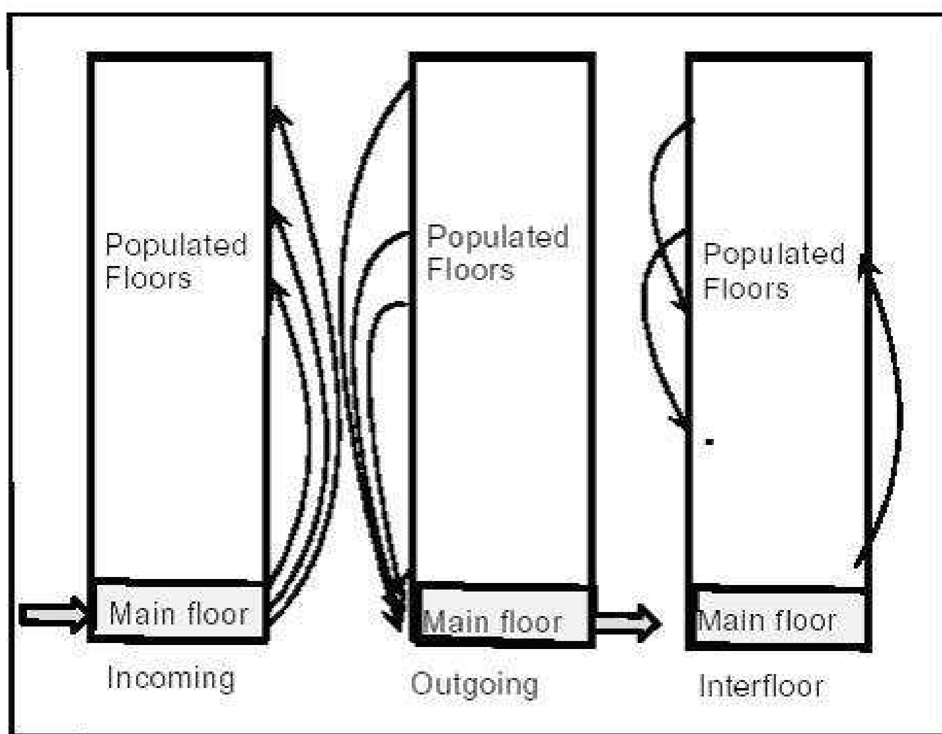

Figure 8.1: Three main passenger traffic components

Since up-peak handling capacity is a limiting factor, elevator systems are designed by predicting the heaviest likely up-peak demand in a building, and then determining a configuration that can accommodate that demand. If up-peak capacity is sufficient, then down-

peak generally will be also. Up-peak traffic is the easiest type to analyze, since all passengers enter cars at the lobby, their destination floors are serviced in ascending order, and empty cars then return to the lobby. The only control decisions in pure up traffic are to determine when to open and close the elevator doors at the lobby. These decisions affect how many passengers will board an elevator at the lobby. Once the doors have closed, there is really no choice about the next actions: the car calls registered by the passengers must be serviced in ascending order and the empty car must then return to the lobby. More general two way traffic comes in two varieties. In two way lobby traffic, up-moving passengers arrive at the lobby and down-moving passengers depart at the lobby. Compared with pure up traffic, the round trip times will be longer, but more passengers will be served. In two way interfloor traffic, most passengers travel between floors other than the lobby. Interfloor traffic is more complex than lobby traffic in that it requires almost twice as many stops per passenger, further lengthening the round trip times.

Two way and down-peak traffic patterns require many more decisions than does pure up traffic. After leaving the lobby, a car must decide how high to travel in the building before turning, and at what floors to make additional pickups. Because more decisions are required in a wider variety of contexts, more control strategies are also possible in two ways and down-peak traffic situations. For this reason, a downpeak traffic pattern was chosen as a testbed for the research.

8.1.2 The Elevator Simulator

The particular elevator system is a simulated 10-story building with 4 elevator cars. Passenger arrivals at each floor are assumed to be Poisson, with arrival rates that vary during the course of the day[Crites & Barto, 1998]. Simulations use a traffic profile which dictates arrival rates for every 5-minute interval during a typical afternoon down-peak rush hour. More about the Poisson distribution is given in the appendix. Now the mean number of passengers varies during a typical down-peak hour. Following Table shows the mean number of passengers arriving at each of floors 2 through 10 during each 5-minute interval who are headed for the lobby.

Table 8.1: Mean Passenger arrivals

Time	00	05	10	15	20	25	30	35	40	45	50	55
Rate	1	2	4	4	18	12	8	7	18	5	3	2

8.1.3 System Dynamics

The system dynamics are approximated by the following parameters[Crites & Barto 1995]:

- Floor time (the time to move one floor at maximum speed): 1.45 secs.
- Stop time (the time needed to decelerate, open and close the doors, and accelerate again): 7.19 secs.
- Turn time (the time needed for a stopped car to change direction): 1 sec
- Load time (the time for one passenger to enter or exit a car): random variable with a range from 0.6 to 6.0 secs and a mean of 1 sec.
- Car capacity: 20 passengers.

8.1.4 State Space

The state space is continuous because it includes the elapsed times since any hall calls were registered, which are real-valued. Even if these real values are approximated as binary values, the size of the state space is still immense. Its components include 2^{18} possible combinations of the 18 hall call buttons (up and down buttons at each landing except the top and bottom), 2^{40} possible combinations of the 40 car buttons, and 18^4 possible combinations of the positions and directions of the cars (rounding off to the nearest floor). Ignoring everything except the configuration of the hall and car call buttons and the approximate position and direction of the cars, an extremely conservative estimate of the size of a discrete approximation to the continuous state space is:

$$2^{18} * 2^{40} * 18^4 = 10^{22} \, states \tag{8.1}$$

8.1.5 Control Action and Constraints

Each elevator has a small set of primitive actions. If it is stopped at a floor, it must either "move up" or "move down". If it is in motion between floors, it must either "stop at the next floor" or "continue past the next floor". Due to passenger expectations, there are two constraints on these actions: a car cannot pass a floor if a passenger wants to get off there and cannot turn until it has serviced all the car buttons in its present direction. Three additional constraints are also added to build some primitive prior knowledge:

1. A car cannot stop at a floor unless someone wants to get on or off there
2. It cannot stop to pick up passengers at a floor if another car is already stopped there
3. Given a choice between moving up and down, it should prefer to move up (since the down-peak traffic tends to push the cars toward the bottom of the building).

Because of this last constraint, the only real choices left to each car are the stop and continue actions. The actions of the four elevator cars are executed asynchronously since they may take different amounts of time to complete.

8.1.6 Performance Criteria

The performance objectives of an elevator system can be defined in many ways. One possible objective is to minimize the average wait time, which is the time between the arrival of a passenger and his entry into a car. Another possible objective is to minimize the average system time, which is the sum of the wait time and the travel time. A third possible objective is to minimize the percentage of passengers that wait longer than some dissatisfaction threshold (usually 60 seconds). Another common objective is to minimize the average squared wait time. The latter performance objective is chosen since it tends to keep the wait times low while also encouraging fair service. For example, wait times of 2 and 8 seconds have the same average (5 seconds) as wait times of 4 and 6 seconds. But the average squared wait times are different (34 for 2 and 8 versus 26 for 4 and 6).

8.2 The Algorithm

8.2.1 Elevator System in Continuos State Space

Discrete-Event Reinforcement Learning Elevator systems can be modeled as discrete event systems, where significant events (such as passenger arrivals) occur at discrete times, but the amount of time between events is a real-valued variable. In such systems, the constant discount factor used in most discrete-time reinforcement learning algorithms is inadequate. This problem can be approached using a variable discount factor that depends on the amount of time between events. In this case, the cost-to-go function (Reward function) is defined as an integral rather than as an infinite sum, as follows:

$$\sum_{t=0}^{\infty} \gamma^t c_t \quad becomes \quad \int_0^{\infty} e^{-\beta\tau} c_\tau d\tau \tag{8.2}$$

where c_t is the immediate cost at discrete time t, c_τ is the instantaneous cost at continuous time t (the sum of the squared wait times of all currently waiting passengers), and β controls the rate of exponential decay. Since the wait times are measured in seconds, instantaneous cost is scaled down by a factor of 10^6 to keep the cost-to-go values from becoming exceedingly large. Because in elevator system time is real valued, the Eq. (4.31) take the following form

$$\Delta Q(i,u) = \alpha\Big(\int_{t_x}^{t_y} e^{-\beta(\tau - t_x)} c_\tau d\tau + e^{-\beta(t_y - t_x)} \min_{b \in U(j)} Q(j,b) - Q(i,u)\Big) \tag{8.3}$$

where action u is taken from state i at time t_x, the next decision is required at time t_y, α is the learning rate parameter, and c_τ and β are defined above. $e^{-\beta(t_y - t_x)}$ acts as a variable discount factor that depends on the amount of time between events which is random as the time t_y is random. Indeally the quantity $\Delta Q(i,u)$ is also a function of random time t_y and hence $\Delta Q(i,u)$ is random too. Since the aim of the problem is to minimize the average squared wait times, it can be assumed that c_τ is quadratic. The integral in the Q-learning update rule then takes the form

$$\int_{t_x}^{t_y} \sum_p e^{-\beta(\tau - t_x)} (\tau - t_x + w_p)^2 d\tau \tag{8.4}$$

where w_p is the amount of time each passenger p waiting at time t_y has already waited

at time t_x. The integral when solved by parts yields

$$\sum_p e^{-\beta w_p}\left[\frac{2}{\beta^3}+\frac{2w_p}{\beta^2}+\frac{w_p^2}{\beta}\right]-e^{-\beta(w_p+t_y-t_x)}\left[\frac{2}{\beta^3}+\frac{2(w_p+t_y-t_x)}{\beta^2}+\frac{(w_p+t_y-t_x)^2}{\beta}\right] \quad (8.5)$$

A difficulty arises in using this formula, it requires knowledge of the waiting times of all waiting passengers. However, only the waiting times of passengers who press hall call buttons will be known in a real elevator system. The number of passengers arrived after the button is pressed and their exact waiting times will not be available. There are two ways of dealing with this problem:

1. omniscient reinforcement schemes

2. online reinforcement schemes

The simulator has access to the waiting times of all passengers. It could use this information to produce the necessary reinforcement signals. This is called the omniscient reinforcements scheme.

The other method is to train the elevator system using only information that would be available to a real system online. Such online reinforcements assume only that the waiting time of the first passenger in each queue is known (which is the elapsed button time). If the Poisson arrival rate λ for each queue is known or can be estimated, the Gamma distribution can be used to estimate the arrival times of subsequent passengers. The time until the nth subsequent arrival follows the Gamma distribution $(n, 1/\lambda)$. More about the gamma distribution is given in appendix. For each queue, subsequent arrivals will generate the following expected costs during the first b seconds after the hall button has been pressed:

$$\sum_{n=1}^{\infty}\int_0^b (probability\ n^{th}\ arrival\ occurs\ at\ time\ \tau) * (cost\ given\ arrival\ at\ time\ \tau)d\tau \quad (8.6)$$

The above integral is equal to

$$\sum_{n=1}^{\infty}\int_0^b \frac{\lambda^n\tau^{n-1}e^{-\lambda\tau}}{(n-1)!}\int_0^{b-\tau} w^2 e^{-\beta(w+\tau)}dwd\tau = \int_0^b\int_0^{b-\tau}\lambda w^2 e^{-\beta(w+\tau)}dwd\tau \quad (8.7)$$

This integral can be solved by parts to yield expected costs.

8.2.2 Method of Omniscient Reinforcements

Omniscient reinforcements are updated incrementally when a passenger arrives at a queue or when a passenger gets on or off of a car, and when a control decision is made. These incremental updates are a natural way of dealing with the discontinuities in reinforcement that arise when passengers begin or end waiting between a car's decisions, e.g., when another car picks up waiting passengers. The amount of reinforcement between events is the same for all the cars since they share the same objective function, but the amount of reinforcement each car receives between its decisions is different since the cars make their decisions at different time. Therefore, each car i has an associated storage location, $R[i]$, where the total discounted reinforcement it has received since its last decision (at time $d[i]$) is accumulated.

At the time of each event, the following computations are performed: Let t_0 be the time of the last event and t_1 be the time of the current event. For each passenger p that has been waiting between t_0 and t_1 , let $w_0(p)$ and $w_1(p)$ be the total time that passenger p has waited at t_0 and t_1 respectively. Then for each car i, $\Delta R[i]$ is given by

$$\Delta R[i] = \sum_p e^{-\beta(t_0 - d[i])}\left[\frac{2}{\beta^3} + \frac{2w_0(p)}{\beta^2} + \frac{w_0^2(p)}{\beta}\right] - e^{-\beta(t_1 - d[i])}\left[\frac{2}{\beta^3} + \frac{2w_1(p)}{\beta^2} + \frac{w_1^2(p)}{\beta}\right] \quad (8.8)$$

8.2.3 Method of Online Reinforcements

Online reinforcements are updated incrementally after every hall button is pressed, signaling the arrival of the first waiting passenger at a queue or the arrival of a car to pick up any waiting passengers at a queue and car arrival event i.e when a control decision is made. It is assumed that online reinforcements caused by passengers waiting at a queue end immediately when a car arrives to service the queue, since it is not possible to know exactly when each passenger boards a car. The Poisson arrival rate $\tilde{\lambda}$ for each queue is estimated as the reciprocal of the last inter-button time for that queue, i.e., the amount of time from the last service until the button was pushed again. However, a ceiling of maximum $\tilde{\lambda} \leq 0.04$ passengers per second is placed on the estimated arrival rates to prevent any very small inter-button times from creating huge penalties.

At the time of each event, the following calculations are made Let t_0 be the time of the

last event and t_1 be the time of the current event. For each hall call button b that was active between t_0 and t_1, let $w_0(b)$ and $w_1(b)$ be the elapsed time of button b at t_0 and t_1 respectively. Then for each car i,

$$R[i] = e^{-\beta(t_0-d[i])} \sum_b \Bigg[\frac{2\tilde{\lambda}_b(1-e^{-\beta(t_1-t_0)})}{\beta^4} + \left(\frac{2}{\beta^3} + \frac{2w_0(b)}{\beta^2} + \frac{w_0^2(b)}{\beta}\right) - e^{\beta(t_1-t_0)}\left(\frac{2}{\beta^3} + \frac{2w_1(b)}{\beta^2} + \frac{w_1^2(b)}{\beta}\right) + \tilde{\lambda}_b\left\{\left(\frac{2w_0(b)}{\beta^3} + \frac{w_0^2(b)}{\beta^2} + \frac{w_0^3(b)}{3\beta}\right) - e^{-\beta(t_1-t_0)}\left(\frac{2w_1(b)}{\beta^3} + \frac{w_1^2(b)}{\beta^2} + \frac{w_1^3(b)}{3\beta}\right)\right\}\Bigg] \tag{8.9}$$

8.2.4 Making Decisions and Updating Q-Values

A car that is in motion between floors generates a car arrival event when it reaches the point where it must decide whether to stop at the next floor or continue past the next floor. In some cases, cars are constrained to take a particular action, for example, stopping at the next floor if a passenger wants to get off there. The algorithm used by each agent for making decisions and updating its Q-value estimates is as follows:

1. At time t_x, observing state x, car i arrives at a decision point. It selects an action u using the Boltzmann distribution over its Q-value estimates. (refer Softmax action selection method chapter 1)

$$Probability(stop) = \frac{e^{Q(x,cont)/T}}{e^{Q(x,stop)/T} + e^{Q(x,cont)/T}} \tag{8.10}$$

   where T is a positive "temperature" parameter that is "annealed" or decreased during training. The value of T controls the amount of randomness in the selection of actions. At the beginning of training, when the Q-value estimates are very inaccurate, high values of T are used, (that means more of exploration) which give nearly equal probabilities to each action. Later in training, when the Q-value estimates are more accurate, lower values of T are used, which give higher probabilities to actions that are thought to be superior (greedy actions. more exploitation), while still allowing some exploration to gather more information about the other actions. The eqaution is similar to the softmax action selection method given in chapter 2 except the difference that here Q values are minimized.

2. Let the next decision point for car i be at time t_y in state y. After all cars (including car i) have updated their $R[\cdot]$ values, car i adjusts its estimate of $Q(x, u)$ toward the following target value

$$Q(x,u) = R[i] + e^{-\beta(t_y - t_x)} \min_{stop,cont} Q(x', \cdot) \tag{8.11}$$

Car i then resets its reinforcement accumulator $R[i]$ to zero.

3. Let $x \leftarrow y$ and $t_x \leftarrow t_y$. Go to step 1.

8.3 The Network Used in Elevator Scheduling

The Q-value estimates can be written as $Q(i, u, r)$ where r is a vector of the parameters or weights of the networks. The exact update equation is [Crites & Barto, 1998]:

$$\Delta r = \alpha[R[i] + e^{-\beta(t_y - t_x)} \min_{stop,cont} Q(j, \cdot, r) - Q(i, u, r)] \nabla Q(i, u, r) \tag{8.12}$$

where α is a positive learning parameter, and the gradient $\nabla Q(i, u, r)$ is the vector of partial derivatives of $Q(i, u, r)$ with respect to each component of r.

At the start of training, the weights of each network are initialized to be uniform random numbers between 1 and +1. There are around 47 input units which are as follows

1. *18 units:* Two units encode information about each of the nine down hall buttons. A real-valued unit encodes the elapsed time if the button has been pushed and a binary unit is on if the button has not been pushed.

2. *16 units:* Each of these units represents a possible location and direction for the car whose decision is required. Exactly one of these units will be on at any given time. Note that each car has a different egocentric view of the state of the system.

3. *10 units* These units each represent one of the 10 floors where the other cars may be located. Each car has a "footprint" that depends on its direction and speed. For example, a stopped car causes activation only on the unit corresponding to its current floor, but a moving car causes activation on several units corresponding to the floors

it is approaching, with the highest activations on the closest floors. No information is provided about which one of the other cars is at a particular location.

4. *1 unit:* This unit is on if the car whose decision is required is at the highest floor with a waiting passenger.

5. *1 unit:* This unit is on if the car whose decision is required is at the floor with the passenger that has been waiting for the longest amount of time.

6. *1 unit:* The bias unit is always on.

Under the above architecture state i is encoded as a vector x with components $x_l(i), l = 1, ..., 47$ which is then transformed linearly through a linear layer involving the coefficients $r(k, l)$, to give 20 scalers.

$$\sum_{l=1}^{47} r(k,l)x_l(i) \tag{8.13}$$

where $k = 1, ..., 20$ Each of these scalers become the input to a sigmoidal function $\sigma(\cdot)$. At the outputs of the sigmoidal functions, the scalers

$$\sigma(\sum_{l=1}^{47} r(k,l)x_l(i)) \tag{8.14}$$

where $k = 1, ..., 20$ are obtained. These scalers are linearly combined using coefficients $r(k, u)$ where $u = 1or2$ to produce the final output

$$Q(i,u,r) = \sum_{k=1}^{20} r(k,u)\sigma(\sum_{l=1}^{47} r(k,l)x_l(i)) \tag{8.15}$$

where $u = 1$ corresponds to action "stop" and $m = 2$ corresponds to action "continue".

The target vaue is given by the equation 8.11. The objective is to minimize the mean square error between the "target" given by Eq 8.11 and the "estimate" given by the above equation. Therefore the error is given by

$$E = \frac{1}{2}[R[i] + e^{-\beta(t_y - t_x)} \min_{stop,cont} Q(x', \cdot) - Q(i,u,r)]^2 \tag{8.16}$$

According to sttepest-descent approach the leqarning rule for network weights is given by

$$\Delta r = -\alpha \frac{\partial E}{\partial r} \tag{8.17}$$

Expanding E we get Equation 8.12 which is weight update equation.

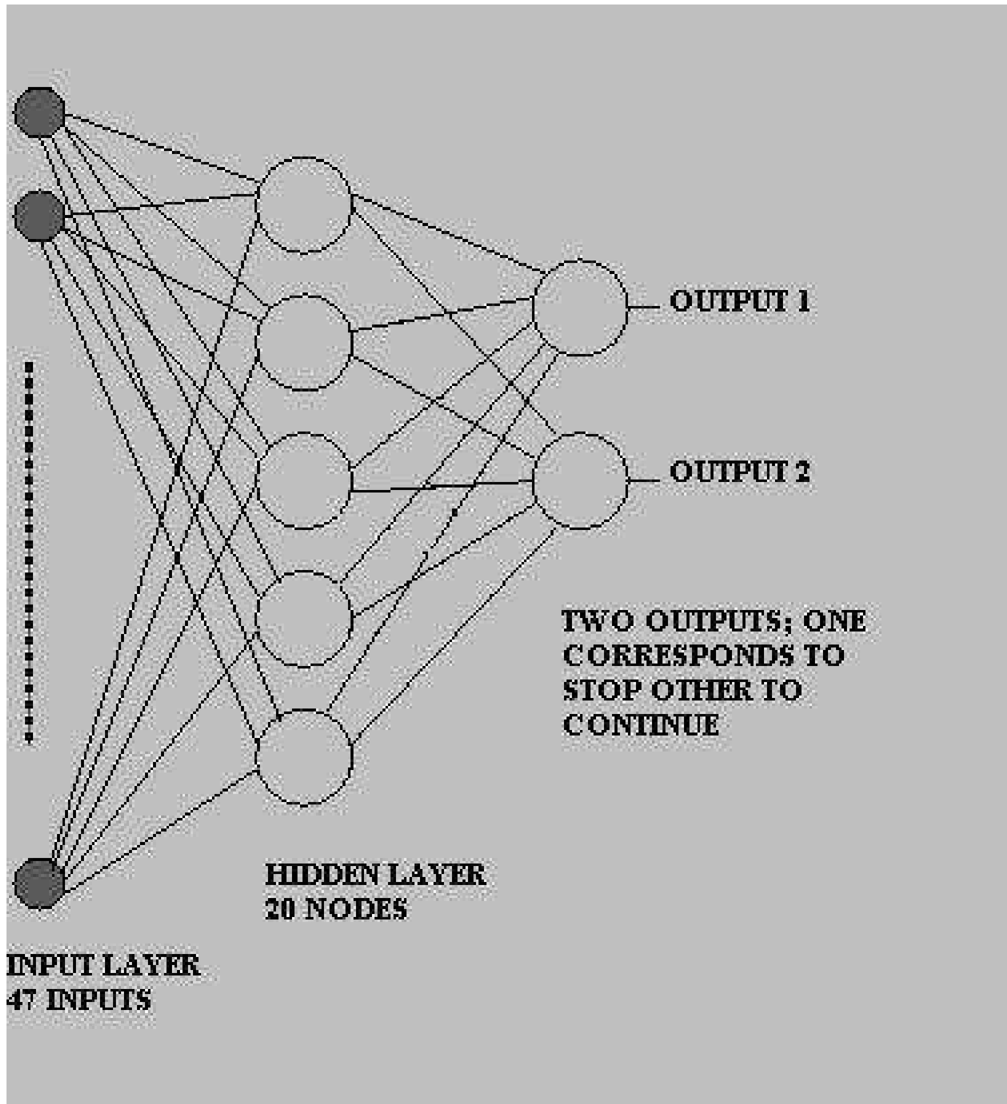

Figure 8.2: Neural Network Architecture

Chapter 9

Solving Traveling Salesman Problem (TSP)

Traveling Salesman Problem is one of the oldest problem in the graph theory. In this chapter a system based on the behavior of real ant is applied to solve the problem. The approach used is similar to a Reinforcement Learning algorithm where each "good step" is rewarded with a positive reinforcement.

9.1 The Problem

The Traveling Salesman Problem is easiest defined as follows:
"You are a traveling salesman, and you have to visit a number of cities and finally return to your starting point. The challenge is to finish the tour so that the total distance is the least." Given below are some of the version of the TSP.

A *Symmetric TSP* is the type of TSP problem where any city has a road to all other cities, and the distance from city a to city b is the same as from b to a. The formalized statement would be:
"Given a set of n nodes and distances for each pair of nodes, find a round trip of minimal total length visiting each node exactly once". The distance from node i to node j is the same as from node j to node i.

In an *Asymmetric Traveling Salesman Problem (ATSP)* the distance from node i to node j and the distance from node j to node i may be different.

The *Hamiltonian Cycle Problem* asks whether a given graph has a Hamiltonian cycle. Given a finite graph G with $V(G)$ the set of vertices and $E(G)$ the set of edges. A Hamiltonian cycle c of G is a cycle that goes through every vertex exactly once.

9.1.1 Application of TSP

Much of the work on the TSP is not motivated by direct applications, but rather by the fact that the TSP provides an ideal platform for the study of general methods that can be applied to a wide range of discrete optimization problem. The numerous direct applications of the TSP bring life to the research area and help to direct future work.

The TSP naturally arises as a sub-problem in many transportation and logistics applications, for example the problem of arranging school bus routes to pick up the children in a school district. A second TSP application from the 1940's involved the transportation of farming equipment from one location to another to test soil. More recent applications involve the scheduling of service calls at cable firms, the delivery of meals to homebound persons, the scheduling of stacker cranes in warehouses, the routing of trucks for parcel post pickup etc.

Although transportation applications are the most natural setting for the TSP, the simplicity of the model has led to many interesting applications in other areas. A classic example is the scheduling of a machine to drill holes in a circuit board or other object. In this case the holes to be drilled are the cities, and the cost of travel is the time it takes to move the drill head from one hole to the next.

9.2 A Real Ants Colony

Individual ants are simple insects with limited memory and capable of performing simple actions. However, an ant colony expresses a complex collective behavior providing intelligent solutions to problems such as carrying large items, forming bridges and finding the shortest routes from the nest to a food source.

A single ant has no global knowledge about the task it is performing. The ant's actions are based on local decisions and are usually unpredictable. The intelligent behavior naturally

emerges as a consequence of the self-organization and indirect communication between the ants. The fascinating behavior of ants has been inspiring researches to create new approaches based on some of the abilities of the ants' colonies. Some of the existing applications include the Traveling Salesman Problem, graph coloring, logistics etc.

In order to solve any problem, two characteristics of ants' colonies will be particularly useful:

1. Their ability to find the shortest route between the nest and a food source, which will be used to find and optimize a path in the graph;

2. The simplicity of each individual ant, which will make it easy for us to model the ant colony as a Multi-Agent System;

9.2.1 Foraging Behavior of Ants

First, understand the foraging behavior of ants and how they can manage to find the shortest path between the nest and a food source using simple local decisions.

Ants use a signaling communication system based on the deposition of pheromone over the path it follows, marking a trail. Pheromone is a hormone produced by ants that establishes a sort of indirect communication among them. Basically, an isolated ant moves at random, but when it finds a pheromone trail there is a high probability that this ant will decide to follow the trail [Dorigo, Maniezzo Colorni, 1996].

An ant foraging for food lay down pheromone over its route. When this ant finds a food source, it returns to the nest reinforcing its trail. Other ants in the proximities are attracted by this substance and have greater probability to start following this trail and thereby laying more pheromone on it. This process works as a positive feedback loop system because the higher the intensity of the pheromone over a trail, the higher the probability of an ant start traveling through it. This elementary behavior of real ants can be used to explain how they can find the shortest path that reconnects a broken line after the sudden appearance of an unexpected obstacle has interrupted the initial path. In fact, once the obstacle has appeared, those ants, which are just in front of the obstacle, cannot continue to follow the pheromone

trail and therefore they have to choose between turning right or left. In this situation we can expect half the ants to choose to turn right and the other half to turn left. A very similar situation can be found on the other side of the obstacle. It is interesting to note that those ants which choose, by chance, the shorter path around the obstacle will more rapidly reconstitute the interrupted pheromone trail compared to those which choose the longer path. Thus, the shorter path will receive a greater amount of pheromone per time unit and in turn a larger number of ants will choose the shorter path. Due to this positive feedback (auto-catalytic) process, all the ants will rapidly choose the shorter path. The most

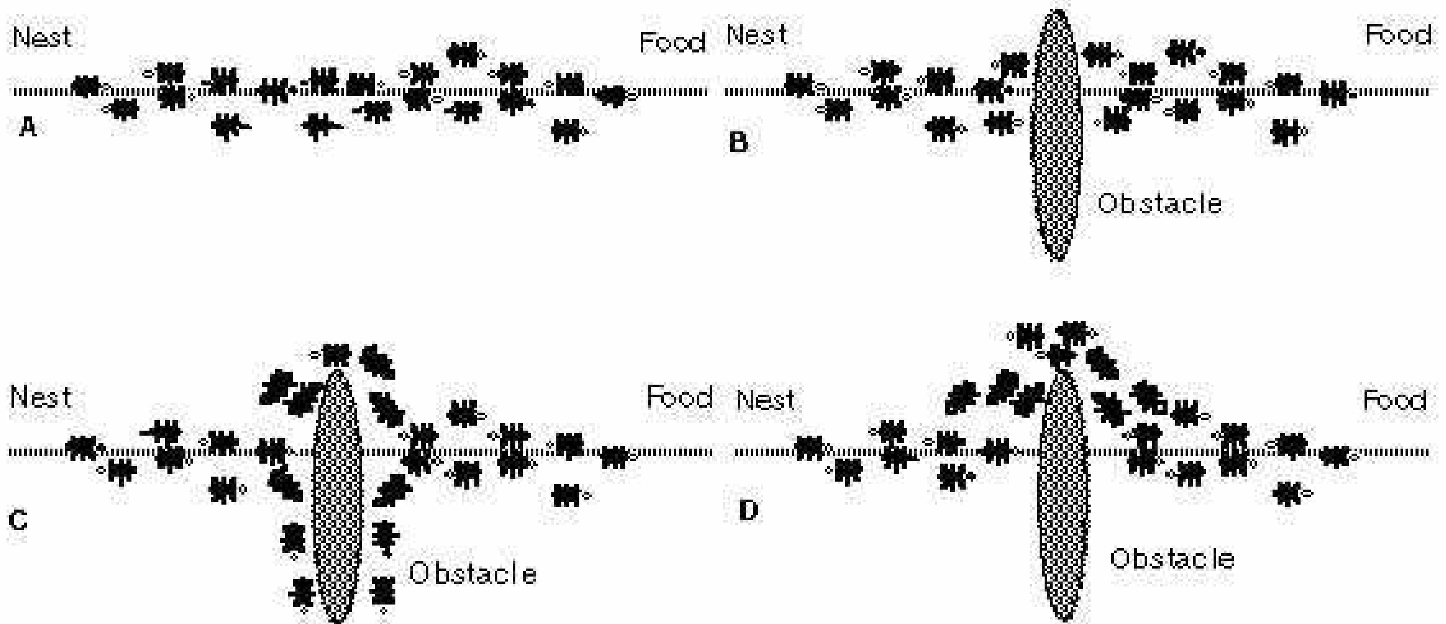

Figure 9.1: Foraging behavior of ants

interesting aspect of this auto-catalytic process is that finding the shortest path around the obstacle seems to be an emergent property of the interaction between the obstacle shape and ants distributed behavior: Although all ants move at approximately the same speed and deposit a pheromone trail at approximately the same rate, it is a fact that it takes longer to contour obstacles on their longer side than on their shorter side which makes the pheromone trail accumulate quicker on the shorter side. It is the ants' preference for higher pheromone trail levels, which makes this accumulation still quicker on the shorter path. The similar

process can be put to work in a simulated world inhabited by artificial ants that try to solve the traveling salesman problem.

9.3 Artificial Ants

An artificial ant is an agent, which moves from city to city on a TSP graph. It chooses the city to move to using a probabilistic function both of trail accumulated on edges and of a heuristic value, which was chosen here to be a function of the edges length. These are three ideas from natural ant behavior that we have transferred to our artificial ant colony:

1. The preference for paths with a high pheromone level
2. The higher rate of growth of the amount of pheromone on shorter paths
3. The trail mediated communication among ants.

Artificial ants were also given a few capabilities which do not have a natural counterpart, but which have been observed to be well suited to the TSP application: artificial ants can determine how far away cities are, and they are endowed with a working memory M_k used to memorize cities already visited.

9.3.1 Ant Algorithm for TSP

Initially, each of the m artificial ants is placed on a randomly chosen city and then iteratively applies at each city a state transition rule. An ant constructs a tour as follows.

At a city i, the ant chooses a still unvisited city j probabilistically, biased by the pheromone trail strength $\tau_{ij}(n)$ on the arc between city i and city j and with the help of a heuristic information, which is a function of the arc length. Ants probabilistically prefer cities which are close and are connected by arcs with a high pheromone trail strength. To construct a feasible solution each ant has a limited form of memory, called tabu list , in which the current partial tour is stored. The memory is used to determine at each construction step the set of cities which still has to be visited and to guarantee that a feasible solution is built. Additionally, it allows the ant to retrace its tour, once it is completed. After all

ants have constructed a tour, the pheromones are updated. This is typically done by first lowering the pheromone trail strengths by a constant factor and then the ants are allowed to deposit pheromone on the arcs they have visited. The trail update is done in such a form that arcs contained in shorter tours and/or visited by many ants receive a higher amount of pheromone and are therefore chosen with a higher probability in the following iterations of the algorithm. The amount of pheromone $\tau_{ij}(n)$ represents the learned desirability of choosing next city j when an ant is at city i.

9.3.2 Ant Decision

An artificial ant k in city i chooses the city j to move to among those which do not belong to its working memory M_k by applying the following probabilistic formula

$$j = \left\{ \begin{array}{cc} argmax_{l \in N_i^k} \left\{ [\tau_{i,l}(n)]^{\alpha} \cdot [\eta_{i,l}]^{\beta} \right\} & if\ q \leq q_0 \\ J & otherwise \end{array} \right\} \tag{9.1}$$

where $\eta_{ij} = 1/d_{ij}$ is an a priori available heuristic, q is a value chosen randomly with uniform probability in [0,1], q_0, $0 < q_0 < 1$, is a parameter, and J is a random variable selected according to the following probability distribution, which favors edges which are shorter and have a higher level of pheromone trail.

$$p(J = j)(n) = \frac{[\tau_{ij}(n)]^{\alpha} \cdot [\eta_{ij}]^{\beta}}{\sum_{l \in N_i^k} [\tau_i(n)]^{\alpha} \cdot [\eta_{il}]^{\beta}} \qquad if\ j\ \in\ N_i^k \tag{9.2}$$

where $\eta_{ij} = 1/d_{ij}$ is an a priori available heuristic value, α and β are two parameters which determine the relative importance of the pheromone trail and the heuristic information, and N_i^k is the feasible neighborhood of ant k, that is, the set of cities which ant k has not yet visited. If $\alpha = 0$, the closest cities are more likely to be selected: this corresponds to a classical stochastic greedy algorithm If $\beta = 0$, only pheromone amplification is at work: this method will lead to the rapid emergence of a stagnation situation with the corresponding generation of tours which, in general, are strongly suboptimal. Hence, a trade-off between the relative influence of the heuristic information and the pheromone trails exists. This trade-off is similar to the trade-off between *exploration and exploitation* that exists in classical reinforcement learning.

9.3.3 Pheromone Update

There can be many methods to update pheromones. Three such methods are described below

1. After all ants have constructed their tours, the pheromone trails are updated. This is done by first lowering the pheromone strength on all arcs by a constant factor and then allowing each ant to add pheromone on the arcs it has visited.

$$\tau_{ij}(n+1) = (1-\rho)\tau_{ij}(n) + \sum_{k=1}^{m} \Delta\tau_{ij}^{k}(n) \tag{9.3}$$

   where $0 < \alpha < 1$ is the pheromone trial evaporation. If an arc is not chosen by the ants, its associated pheromone strength decreases exponentially. $\tau_{ij}^{k}(n)$ is the amount of pheromone ant k puts on the arcs it has visited; it is defined as

$$\Delta\tau_{ij}^{k}(n) = \begin{cases} \frac{1}{L^k(n)} & if\ arc\ (i,j)\ is\ used\ by\ ant\ k \\ 0 & otherwise \end{cases} \tag{9.4}$$

   where $L_k(n)$ is the length of the kth ant's tour. By above Eq., the better the ant's tour is, the more pheromone is received by arcs belonging to the tour. In general, arcs which are used by many ants and which are contained in shorter tours will receive more pheromone.

2. *Global pheromone trail update* In this method the global best ant (i.e the ant that complete the tour in shortest distance among all ants) is allowed to add pheromone after each iteration. The update equation for this case is

$$\tau_{ij}(n+1) = (1-\rho)\tau_{ij}(n) + \alpha\Delta\tau_{ij}^{gb}(n) \tag{9.5}$$

   where $\Delta\tau_{ij}^{gb}(n) = \frac{1}{L^{gb}}$. Here the trial update only applies to the arcs of the global-best tour, not to all the arc visited as in previous case

3. *Local pheromone trail update.* In this method in addition to global updating the ants use a local update rule that they apply immediately after having crossed an arc during the tour construction

$$\tau_{ij} = (1-\xi)\tau_{ij} + \xi\tau_0 \tag{9.6}$$

where $0 < \xi < 1$ and τ_0 is a small constant. The effect of the updating rule is to make an already chosen arc less desirable for a following ant. In this way the exploration of not yet visited arcs is increased.

The ant colony can be interpreted as a reinforcement learning system, in which reinforcements modify the strength (i.e. pheromone trail) of connections between cities. An ant can either, with probability q_0, exploit the experience accumulated by the ant colony in the form of pheromone trail (pheromone trail will tend to grow on those edges which belong to short tours, making them more desirable), or with probability $(1 - q_0)$, apply a biased exploration (exploration is biased towards short and high trail edges) of new paths by choosing the city to move to randomly, with a probability distribution that is a function of both the accumulated pheromone trail, the heuristic function, and the working memory M_k.

9.4 Using Candidate Lists

For the large TSP the solution can be achieved faster by the use of a data structure called candidate list. A candidate list is a list of preferred cities to be visited; it is a static data structure which contains, for a given city i, the cl closest cities. In practice, an ant in ACS with candidate list first chooses the city to move to among those belonging to the candidate list. Only if none of the cities in the candidate list can be visited then it considers the rest of the cities. Using nearest neighbor lists is a reasonable approach when trying to solve TSPs. For example, in many TSP instances the optimal tour can be found within a surprisingly low number of nearest neighbors. For example, an optimal solution is found for *pcb*442, which has 442 cities, within a subgraph of the 6 nearest neighbors.

9.5 Parameter setting and basic properties

There are a number of parameters which affect the rate of convergence and even the value of the optimum reached. The parameters that can affect directly or indirectly the time of computation and optimum value are

1. α and β: The relative importance given to nearest neighbour heuristic and the pheromone value.

2. ρ : The pheromone evaporation factor

3. Size of candidate list : The number of nearest neighbour in a city's candidate list.

4. q_0 : The parameter that controls rate of exploration and exploitation.

5. m : The number of ants used in a trial.

6. τ_0 : The initial value of the pheromone.

Several value of each parameter is tested while the other parameters are held constant. Each experiment is done 15 times in order to remove the error due to randomness. Most of the parameter testing is done using a 70 cities problem called st70.tsp (70-city problem Smith/Thomson).

9.6 Results

Table 9.1: Variation of convergence rate and value of optimum reached in ACS algorithm with initial value of Pheromone τ_0.
Problem is st70, a 70 cities symmetric traveling Salesman Problem. Number of iteration = 1500. Values of other parameters are fixed at $q_0 = 0.9, \alpha = 1, \beta = 2$, Number of Ants $m = 50$, Pheromone evaporation factor $\rho = 0.9$, Size of candidate list $cl = 10$

Initial Value of pheromones τ_0	ACS best	ACS average	ACS variance
0.000005	680.229	690.66	86.6914
0.00001	680.79	688.901	92.3596
0.00002	678.95	685.822	27.0133
0.00003	677.11	687.31	53.3965
0.00005	680.829	688.674	52.7406
0.0001	680.87	696.294	58.1373
0.001	701.957	713.701	27.4605

Table 9.2: Variation of convergence rate and value of optimum reached in ACS algorithm with the parameter q_0
Problem is st70, a 70 cities symmetric traveling Salesman Problem. Number of iteration = 1500. Values of other parameters are fixed at $\alpha = 1, \beta = 2$, Number of Ants $m = 50$, Pheromone evaporation factor $\rho = 0.9$, Size of candidate list $cl = 10$, initial value of Pheromone $\tau_0 = 0.00002$.

Value of q_0	ACS best	ACS average	ACS variance
0.95	682.323	690.668	78.3872
0.90	678.412	684.807	29.6723
0.80	677.11	684.092	36.4745
0.70	677.11	683.541	30.8436
0.60	677.11	686.715	51.5705
0.50	677.11	690.904	85.0386
0.40	687.212	705.102	137.622
0.30	702.497	722.136	224.512

Table 9.3: Variation of convergence rate and value of optimum reached in ACS algorithm with the number of ants.
Problem is st70, a 70 cities symmetric traveling Salesman Problem. Number of iteration = 1500. Values of other parameters are fixed at $q_0 = 0.9, \alpha = 1, \beta = 2$, Pheromone evaporation factor $\rho = 0.9$, Size of candidate list $cl = 10$, initial value of Pheromone $\tau_0 = 0.00002$.

Number of ants	ACS best	ACS average	ACS variance
10	699.413	721.101	117.472
20	686.16	701.038	111.236
30	684.545	691.233	86.9454
50	677.71	684.171	48.2336
70	677.11	683.927	57.4518
100	677.11	682.768	43.7627

Table 9.4: Variation of convergence rate and value of optimum reached in ACS algorithm with the size of candidate list.
Problem is st70, a 70 cities symmetric traveling Salesman Problem. Number of iteration = 1500. Values of other parameters are fixed at $q_0 = 0.9, \alpha = 1, \beta = 2$, Pheromone evaporation factor $\rho = 0.9$, Number of ants $m = 50$, initial value of Pheromone $\tau_0 = 0.00002$.

Size of candidate list	ACS best	ACS average	ACS variance
7	687.405	696.517	45.4973
10	677.11	683.601	38.5414
20	680.58	685.94	20.9813
30	680.251	689.478	55.9769
40	680.659	688.96	40.2249
50	680.582	687.803	61.5907
70	680.295	687.062	42.9344

Table 9.5: Variation of convergence rate and value of optimum reached in ACS algorithm with the Pheromone evaporation factor ρ.
Problem is st70, a 70 cities symmetric traveling Salesman Problem. Number of iteration = 1500. Values of other parameters are fixed at $q_0 = 0.9, \alpha = 1, \beta = 2$, Size of candidate list $cl = 10$, initial value of Pheromone $\tau_0 = 0.00002$, size of candidate list $cl = 10$, number of ants=50..

Pheromone evaporation Factor ρ	ACS best	ACS average	ACS variance
0.99	677.11	682.033	41.7969
0.98	677.11	680.682	37.3487
0.95	677.11	683.426	43.6835
0.92	677.11	684.802	60.0056
0.90	677.71	684.171	48.2336
0.85	681.715	691.304	59.7964
0.80	682.511	693.126	75.0588
0.70	689.058	689.329	32.1685
0.60	685.448	704.766	214.164

Table 9.6: Variation of convergence rate and value of optimum reached in ACS algorithm with the value of α/β
Problem is st70, a 70 cities symmetric traveling Salesman Problem. Number of iteration = 1500. Values of other parameters are fixed at $q_0 = 0.9$, Pheromone evaporation factor $\rho = 0.9$, Size of candidate list $cl = 10$, initial value of Pheromone $\tau_0 = 0.00002$, size of candidate list $cl = 10$.

Value of β/α	ACS best	ACS average	ACS variance
0.5	677.438	686.373	49.3661
1.0	677.11	684.615	33.8028
2.0	677.11	684.171	48.2336
2.5	677.11	689.095	56.7334
3.0	677.11	687.497	45.2908
4.0	682.965	695.475	89.9429
5.0	686.679	696.234	76.5872

Table 9.7: Performance of ACS algorithm when applied to various TSPs.
Values of parameters are fixed at $q_0 = 0.9, \alpha = 1, \beta = 2$, Pheromone evaporation factor $\rho = 0.9$, Size of candidate list is between $cl = 10$ to $cl = 20$, initial value of Pheromone $\tau_0 = 1/(n \cdot T^{heuristic})$. All the data is taken from TSPLIB (home page: http://www.iwr.uni-heidelberg.de/groups/comopt/software/TSPLIB95/)

Problem name	ACS best	ACS average	ACS variance	Avg No of iteration for convergence	Optimum Integer Solution
st70	677.11	684.615	33.8028	1500	677
berlin52	7542.79	7798.64	29597.3	1000	7542
bays 29	9076	9147.63	1676.4	800	9076
rd100	7917.99	8045.19	15968.8	4000	7910
ch150	6540.34	6585.83	1502.41	5000	6528
d198	15885.84	16122.4	27885.78	6000	15780

9.7 Discussion on ACS algorithm

The ACS algorithm seems to work fine for problems up to 100 cities. For these problems the average value of ACS algorithm is not very far from optimum. For large problems the performance of ACS is not that good. It takes too many step to converge and the converged value is generally far from optimum. The effective exploration Vs exploitation rate which was large at the start of the iteration goes down fast. Therefore for a large problem there is very much probability that the algorithm converge to a value which is far away from optimum. Exploration Vs exploitation rate can be increased by increasing the value of initial pheromone level τ_0 or by decreasing the pheromone evaporation factor ρ. But the problem is number of iteration for convergence increases and hence computation time increases to a large extent.

The dependence of convergence rate of ACS algorithm with various parameters is also studied. The factor that affects most is the initial value of the pheromone level τ_0. That is so because the value of initial pheromone level controls the exploration at the start. A very high level of initial pheromone increases the pheromone level to such an extent that the convergence is not achieved even after 5000 iteration. Algorithm converges to a local optimum when the initial pheromone level is too low. Usually the value of $\tau_0 = 1/(n \cdot T^{heuristic})$, where $T^{heuristic}$ is the approximate tour length by nearest neighbour algorithm, is found to be close to optimum. The other important factor that affects the rate of convergence is the size of the candidate list *cl*. For st70 problem the size of the optimum candidate list is found to be around 10 cities. The size of optimum candidate list increases with the complexity of problem but it was generally observed that a candidate list of about 20-30 cities is sufficient for symmetric TSP containing up to 1000 cities. Another very important factors that influences the virtuosity of ACS algorithm is q_0, the parameter that controls the rate of exploration and exploitation. A value of q_0 which is close to 1 generally gives a quick solution (usually the algorithm converges in less than 400 steps) which is more or less good enough. ACS mean is found to be close to optimum and variance is low when a high q_0 value is used. But the number of times the algorithm actually converges to optimum is small. For st70 problem $q_0 = 0.7$ gives maximum instance of ACS algorithm converging to

actual optimum. However a large value of q_0 can't be used while solving large TSP problem because of requirement of large computational time for convergence.

Generally it was observed that the quality of solution can be augmented by increasing the number of ants but a heavy price in terms of computational time is paid for that. Therefore the number of ants is kept between $0.2n$ to $0.6n$, where n is the number of cities in the problem, for problems with cities less then 100 and between 50 to 100 for problems that are larger than 100 cities. Another factor which affects the convergence rate and quality of solution to some extent is the pheromone evaporation factor. A pheromone evaporation factor ρ close to 1 generally slows down the convergence rate but increases the overall quality of the solution. For st70 problem $\rho = 0.98$ gives the best solution. Generally ρ is taken to be close to 0.9 while solving large TSP occurence.

It was found that the ratio β/α generally does not influence the overall solution. Thats because the goal of the heuristic is to give initial direction to the algorithm. it does not influence the overall quality of the solution much. However an unusually high ratio ($>$ 5) hampers the algorithm in general. As discussed earlier the ACS algorithm is not very appropriate for solving large TSP instances. There are two main reason for not-so-good performance of the ACS algorithm.

1. Search stagnation:- After some thousands of iteration search get stagnated as the probability with which the algorithm chooses a city with pheromone level less than maximum get infinitesimally small.

2. Slow Convergence rate:- The heuristic used in ACS algorithm is nearest neighbour heuristic which for large TSP occurence is not very good.

The two algorithms that are discussed next - ACS algorithm with local search and MAXMIN ant algorithm performs much better then ACS algorithm as they try to counter the above two problems.

9.8 Local Search Algorithms for TSP

Local search starts from some initial assignment (initial tour) and then it repeatedly tries to improve the current assignment by local changes. If in the neigbourhood of the current tour T a better tour T' is found it replaces the current tour and the local search is continued from T'. Given a feasible tour, these algorithms repeatedly performs operations (*exchanges or moves*) from the given class, so long as each reduces the length of the current tour, until a tour is reached for which no operations yields an improvement. The most widely used local search algorithms are 2-opt and 3-opt algorithms.

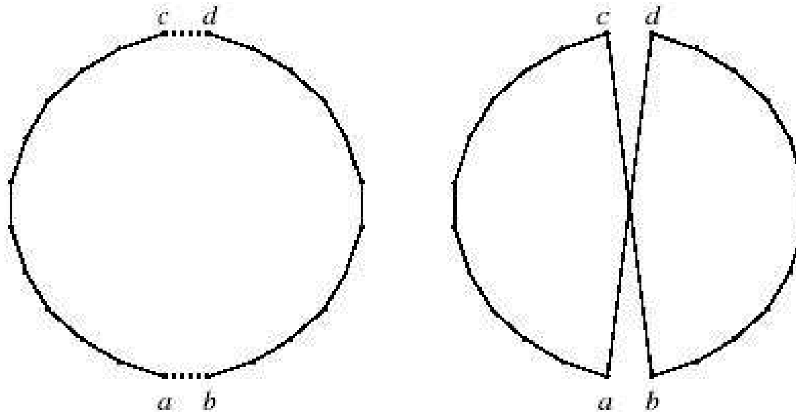

Figure 9.2: A 2-Opt move: original tour on the left and resulting tour on the right

In a 2-opt algorithm a move deletes two edges thus breaking the tour into two paths, and then reconnects those paths in the other possible way. In a 3-opt algorithm a move deletes the tour into three parts and connect them in other possible ways.

But the straightforward implementation of these algorithms for large TSP instances are not possible. A single run of a 2-opt or a 3-opt would require $O(n^2)$ or $O(n^3)$ exchanges to be examined. Fortunately, there exist some speed-up techniques achieving, in practice, run-times which grow sub-quadratically. This effect is obtained by examining only a small part of the whole neighborhood. The two techniques are used to reduce the run-time of 2-opt and 3-opt implementations. One, consists in restricting the set of moves which are examined to those contained in a candidate list of the nearest neighbors ordered according to

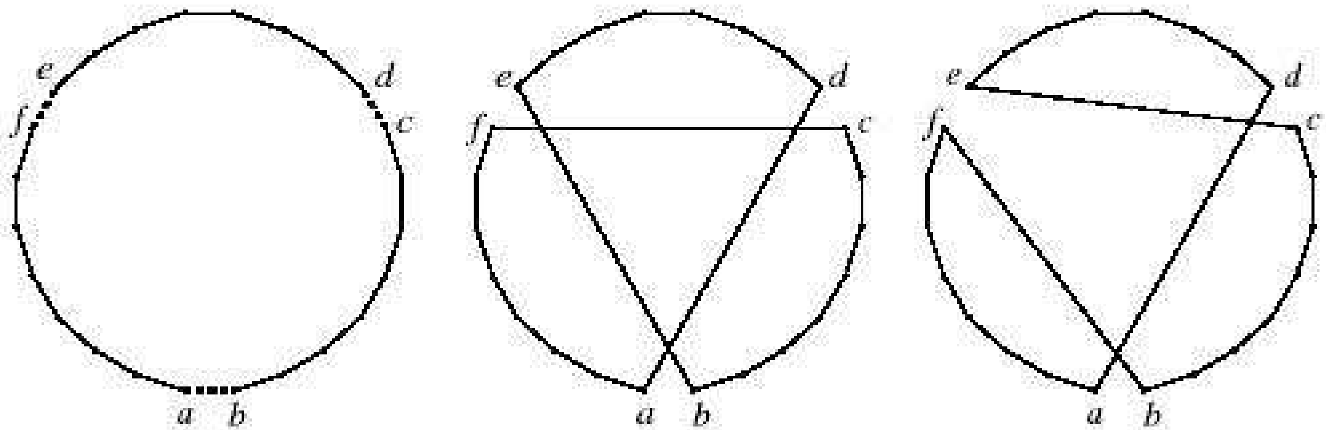

Figure 9.3: Two possible 3-Opt moves: original tour on the left and resulting tours on the right

nondecreasing distances. Using candidate lists, for a given starting node i we only consider moves which add a new arc between i and one of the nodes in its candidate list. Hence, by using a neighborhood list of bounded length, an improving move can be found in constant time. An additional speed-up is achieved by performing a fixed radius nearest neighbor search. For 2-opt at least one newly introduced arc has to be shorter than any of the two removed arcs (i, j) and (k, l). Due to symmetry reasons, it is sufficient to check whether $dij > dik$. A similar argument also holds for 3-opt.

9.9 Result of ACS System with Local search

Table 9.8: Performance of ACS algorithm with local search when applied to various TSPs. Values of parameters are fixed at $q_0 = 0.9, \alpha = 1, \beta = 2$, Pheromone evaporation factor $\rho = 0.9$, Size of candidate list is between $cl = 10$ to $cl = 20$, initial value of Pheromone $\tau_0 = 1/(n \cdot T^{heuristic})$. All the data is taken from TSPLIB (home page: http://www.iwr.uni-heidelberg.de/groups/comopt/software/TSPLIB95/)

Problem name	ACS best	ACS average	ACS variance	Avg No of iteration for convergence	Optimum Integer Solution
st70	677.11	678.04	2.608	50	677
rd100	7910.39	7993,519	3005.14	200	7910
ch150	6528.9	6539.4	85.89	300	6528
d198	15780.79	15812.6	1788.78	400	15780
pr299	48217.25	48346.72	11983.69	600	48191
pcb442	50810.62	50943.78	21653.25	1200	50772

The performance of ACS-3-opt algorithm is very good as compared to the ACS algorithm. Quick and good solution of TSPs can be found using this algorithm. Only disadvantage of using these algorithm is search stagnation. After around 1000 iteration search is stagnated due to the large difference between pheromone value of arcs which are in current solution and of those arcs which are not. The MAX-MIN algorithm is an improvement over ACS-3-opt algorithm because it stop search stagnation by limiting the Pheromone to a maximum and minimum value..

9.10 MAX-MIN Ant System

One of the problem that arises when using ACS algorithm is that after a certain number of steps all the ants follow same path i.e. after some local optimum path is found the probability of finding a better path get reduced. This happens because of high difference between the pheromone value of arcs that are there in the path and that of arcs which are not in the path. As there is no limit over maximum value of pheromone in ACS the arcs that are not in the optimum path is diminished to such an extent that the probability of an ant following that arc is almost zero. MAX-MIN Ant System (MMAS) is a direct improvement over ACS. The solutions in MMAS are constructed in exactly the same way as in ACS, but the pseudo-random proportional action choice rule of ACS is not considered. In fact using that action choice rule, very good solutions of large TSP could be found faster, but the final solution quality achieved was worse.

The main modifications introduced by MMAS with respect to ACS is the following.

To avoid search stagnation (which happened frequently in ACS), the allowed range of the pheromone trail strengths is limited to the interval $[\tau_{min}, \tau_{max}]$, that is, $all \tau_{ij}\ \tau_{min} \leq \tau_{ij} \leq \tau_{max}$. The pheromone trails are initialized to the upper trail limit. This causes a higher exploration at the start of the algorithm as change in value of τ is small as compared to τ initially. After all ants have constructed a solution, the pheromone trails are updated according to

$$\tau_{ij}(t+1) = (1-\rho)\tau_{ij}(t) + \Delta\tau_{ij}^{best} \tag{9.7}$$

where $\Delta\tau_{ij}^{best} = 1/L^{best}$. The ant which is allowed to add pheromone may be the iteration-best solution $Tour^{ib}$, or the global-best solution $Tour^{gb}$.

9.10.1 Fastening the Trial Update

When applying ACO algorithms to the TSP, pheromone trails are stored in a matrix with $O(n^2)$ entries (one for each arc). All the entries of this matrix should be updated at every iteration because of pheromone trail evaporation implemented by the above equation. Obviously, this is a very expensive operation if large TSP instances should be solved. To avoid this, in MMAS pheromone evaporation is applied only to arcs connecting a city i to cities belonging to $i's$ candidate list. Hence, the pheromone trails can be updated in $O(n)$.

9.11 Result of MAXMIN ANT SYSTEM

Table 9.9: Performance of MMAS algorithm with local search when applied to various TSPs. Values of parameters are fixed at $\alpha = 1, \beta = 2$, Pheromone evaporation factor $\rho = 0.9$, Size of candidate list is between $cl = 20$ to $cl = 40$, initial value of Pheromone $= \tau_{max} = 1/(\rho \cdot T^{heuristic})$. $\tau_{min} = \tau max/(2 \cdot n)$ All the data is taken from TSPLIB (home page: http://www.iwr.uni-heidelberg.de/groups/comopt/software/TSPLIB95/)

Problem name	ACS best	ACS average	ACS variance	Optimum Integer Solution
rd100	7910.39	7914.22	35.14	7910
ch150	6528.9	6529.4	8.89	6528
d198	15780.43	15790.11	265.74	15780
pr299	48192.45	48216.72	576.36	48191
pcb442	50772.48	50825.54	3969.43	50772
rat783	8815.25	8878.29	9876.34	8806

The MAXMIN ant system give a very good results when applied to various TSPs. In fact the MAXMIN ant system is quite comparable to Iterative local search algorithm (best solution given at website mentioned above) which is one of the best algorithm to solve TSP.

9.12 Improved Max-Min Ant System

The improved Max-Min ant system is similar to MMAS except three differences. . First a pseudo random proportional rule with a low q_0 value is used to accelerate the convergence to a solution. Second a local updating rule (similar to one used in ACS system) is used which discourage the ants in the same iteration to follow similar path. Third after each iteration both the global best ant and the iteration best ant deposit pheromones but in different proportions (weights). The weight of the global best ant is smaller in beginning and is increased with each iteration whereas the weight of the iteration best ant is larger in beginning and is decreased with every iteration.

There is not much difference between the quality of the solution obtained by improved MAXMIN ant system and MAXMIN ant system. In fact the quality of the solution obtained by the Improved MAXMIN system is slightly on lower side. But there is a much difference in convergence rate. With improved MAXMIN algorithm solution can be achieved faster than MAXMIN ant algorithm..

Chapter 10

Conclusion

Reinforcement learning is a computational approach to understanding and automating goal-directed learning and decision-making. It is distinguished from other computational approaches by its emphasis on learning by the individual from direct interaction with its environment, without relying on exemplary supervision or complete models of the environment. Standard supervised learning techniques are often not applicable in many real world tasks including control, scheduling and routing because the domain information necessary to generate the target outputs are either unavailable or costly to obtain. Reinforcement learning algorithms are advantageous in these cases as it requires least a priori knowledge. Reinforcement learning has two distinct advantages. The first advantage is thet it can handle problems with complex transition mechanisms by making judicious use of the stochastic approximation algorithm thereby eliminating the need to compute or store the transition probabilities. Secondly, RL can integrate within it various function approximation methods (regression, neural networks etc) which can be used to approximate the value functions even when the size of the state space is gargantuan. Reinforcement learning is usually combined with neural networks or supervised learning to solve problems involving very large or even infinite state sets.

In this project reinforcement learning algorithm (ACS system) is used to solve travelling salesman problem. The ACS algorithm illustrates that how Reinforcement Learning can be applied to practically every problem in optimization and scheduling. The performance of the MMAS(Max-Min Ant System) is similar and at times even better than ILS (Iterative

Local Search Algorithm), one of the best algorithm to solve TSPs. Obviously there are some disadvantages too of using Reinforcement learning algorithm. First is value of the parameters used influence the quality of the solution used to a large extent. One can have some intuitive idea of the value of these parameters but there many times have to be found experimentally. Also the value of these parameters are problem specific. A lot of time is wasted on adjusting these parameters so that an optimum solution can be reached. Other disadvantage is that even the best Reinforcement learning algorithm is slow to converge compared to other algorithms. For example even the Improved MMAS algorithm take much more time than ILS algorithm.

The other aspect of RL is its generality. Reinforcement learning can be applied to most of the problems of combbinatorial optimization, allocation, scheduling, finance, game playing. For ex. the Ant Colony system algorithm can be applied to the Quadratic Assignment problem or to the Hitchcock transportation problem with some modifications.

Unfortunately, despite the considerable attention that has been devoted to reinforcement learning over many years, so far there have been only few practical successes in terms of solving large-scale, complex real-world problems. Applications of reinforcement learning are still far from routine and typically require as much art as science. Making applications easier and more straightforward is one of the goals of current research in reinforcement learning.

Chapter 11

Future Work

There are some more interesting applications that can be developed and due to lack of time are not developed in this Project.

Two such applications are

1. *Tetris* A computer can be taught to play tetris using a reinforcement learning algorithm. Tetris is a popular video game played on a two dimension grid. Each square in the grid can be full or empty, making up a wall of bricks with holes. The squares fills up as objects of different shapes fall from the top of the grid. Each falling object can be moved horizontally and can be rotated in all possible ways. There is a finite set of standard shape of the falling objects. The game starts with an empty grid and ends when a square in the top row becomes full and the top of the wall reaches the top of the grid. When a row of full squares is created, this row is removed, the bricks lying above this row move one row downward and the player scores a point. The player objective is to maximize the score attained up to termination of the state.

   The problem can be modeled as of finding an optimal tetris playing strategy as a stochastic shortest path problem. The control denoted by u is the horizontal; positioning and rotation applied to the object. The state consists of two components

   (a) The board position which is a binary description of the full/empty status of each square, denoted by i.

   (b) The shape of the current falling object, denoted by y

Figure 11.1: The game of Tetris

The new component y is generated according to a probability distribution $p(y)$. The number of states in the tetris problem is very large. It is roughly equal to $m2^{hw}$, where m is the different shapes of falling objects, and h and w are the height and width of the grid. For $m = 7, h = 20$ and $w = 10$ the problem has over 10^{61} states. So it is necessary to use some approximate policy iteration method described in chapter 6. Possible candidates are approximate $TD(\lambda)$ and approximate policy iteration algorithm.

2. The problem is to find an optimal path through a set of cities as shown in figure. In this case knowing the Euclidean distance between two cities is not of much help as the straight line path between two cities may be obstacled. The artificial ants algorithm algorithm that is being applied to the TSP in chapter 9 may be applied here with some modifications.

Figure 11.2: Finding optimal tour through the network

Appendix: Generating Random Numbers

In this chapter some of the distributions, that are used in Elevator scheduling problem, and their methods of generation are discussed.

A-1 Probability Distribution

A-1.1 Poisson Distribution

The Poisson distribution is used to model the number of events occurring within a given time interval. The formula for the Poisson probability mass function is

$$P(x, \lambda) = \frac{e^{-\lambda}\lambda^x}{x!} \qquad for\ x = 0, 1, 2, 3... \tag{A.1}$$

λ is the shape parameter which indicates the average number of events in the given time interval. $P(x, \lambda)$ indicates the probability that x number of events occurs given the mean number of events is λ.

A-1.2 Negative Exponential Distribution

If the number of arrivals at a service facility during a specified period occurs according to Poisson distribution, then the distribution of the intervals between successive arrivals must follow the negative exponential distribution [Taha, 2002]. If λ is the rate at which the Poisson events occur, then the distribution of the time x, between successive arrivals is given by

$$f(x) = \lambda e^{-\lambda x}, \qquad \lambda > 0 \tag{A.2}$$

The mean and the variance of the exponential distribution are

$$\begin{array}{r}E\{x\} = \frac{1}{\lambda} \\ var\{x\} = \frac{1}{\lambda^2}\end{array} \tag{A.3}$$

A-1.3 Gamma Probability Distribution

The interarrival times $X_1, X_2, ...$ are continuos, independent random variables, each having the exponential probability density function as given by the above equation. The kth arrival time is simply the sum of the first k interarrival times

$$T_k = X_1 + X_2 + \; + X_k. \tag{A.4}$$

Therefore, the kth arrival time is a continuous random variable and its density function is the k-fold convolution of f given by

$$f_k(t) = \frac{(\lambda t)^{k-1}\lambda e{-}\lambda t}{(k-1)!}, \qquad t > 0 \tag{A.5}$$

This distribution is the special form of gamma distribution with an integer shape parameter k and rate parameter λ.

A-2 Generating Random numbers

One problem that arises while making a software is to generate the random numbers having non-uniform distributions. There are several methods for the random number generation, two of which are described below.

A-2.1 Inversion Method

The objective is to generate a random number with the frequency function $f(x)$. The distribution function is given as

$$F(x) = \int_0^x f(t)dt \tag{A.6}$$

for a continuous distribution, and

$$F(x) = \sum_{i=0}^{x} f(i) \tag{A.7}$$

for a discrete distribution. If u is a random number with the uniform distribution $u \in U(0,1)$ (generated by a computer), then a random number with the desired distribution can be generated by applying the inverse distribution function to u:

$$x = F^{-1}(u) \tag{A.8}$$

This method requires that F^{-1} is easy to calculate. The interarrival time with exponential distribution are usually generated by this method.

A-2.2 Inversion by Chop Down Search from 0

When $f(x)$ is a discrete function (x integer) then the random number x can be calculated by successively adding $f(0) + f(1) + ... + f(x)$ until the sum exceeds u.

Each $f(x)$ is computed from the previous one according to a recursion formula. For a Poisson distribution the recursion formula is

$$f(x) = f(x-1)\frac{L}{x} \tag{A.9}$$

The value of x at which the $f(x)$ exceeds u is the required random number.

References

[1] Bertsekas, P. D. and Tsitsiklis, J. N., Neuro-Dynammic Programming, Athena Scientific, 1996.

[2] Crites, R. H. and Barto, A. G., Improving elevator performance using reinforcement learning, Advances in Neural Information. Processing Systems: Proceedings of the 1995 Conference, pp.1017-1023, Cambridge, MA. MIT Press, 1995.

[3] Crites, R. H. and Barto, A. G., Elevator Group Control Using Multiple Reinforcement Learning Agents, Machine Learning 33, pp. 235-262, 1998.

[4] Dayan, P., The convergence of TD(λ) for general λ, Machine Learning, Vol.8, pp. 341-362, 1995.

[5] Dorigo, M., Maniezzo, V. and Colorni, A., The Ant System: Optimization by a colony of cooperating agents. IEEE Transactions on Systems, Man, and Cybernetics-Part B, 26, pp. 29-44, 1996.

[6] Dorigo, M. and Gambardella, L. M., Ant-Q: A Reinforcement Learning Approach to the Traveling Salesman Problem, Proceedings of the Eleventh International Conference on Machine Learning, pp. 252-260, 1995.

[7] Dorigo, M. and Gambardella, L. M., Ant Colony System: A Cooperative Learning Approach to the Traveling Salesman Problem, IEEE Transactions on Evolutionary Computation, pp. 53-66, 1997.

[8] Haykin, S., Neural Networks: A Comprhensive Foundation, Pearson Education India, 2001.

[9] Mahadevan, Sridhar, Jonathan Connell, Automatic Programming of Behavior-based Robots using Reinforcement Learning, Artificial Intelligence , Vol. 55, Nos. 2-3, pp.311-365, June, 1992.

[10] Mitchell, Tom M., Machine learning, New York, McGraw-Hill, 1997

[11] Nilsson, N. J., Introduction to Machine Learning: An early draft of a proposed textbook, http://robotics.stanford.edu/people/nilsson/mlbook.html, March 26, 2002.

[12] Nilsson, N. J., Principles of artificial intelligence, Springer Verlag, New York, 1982.

[13] Siikonen, M. J., Elevator Group Control with Artificial Intelligence, www.sal.hut.fi/Publications/pdf-files/rsii97a.pdf, Nov 9, 2002.

[14] Singh, S. P. and Sutton, R. S., Reinforcement learning with replacing eligibility traces. Machine Learning, Vol 22, pp.123-158, 1996.

[15] Sutton, R. S., Barto, A. G., Reinforcement Learning: An Introduction, MIT Press, Cambridge, MA, 1998.

[16] Sutton, R. S., Learning to predict by the method of temporal differences, Machine Learning, Vol. 3, pp.9-44, 1988.

[17] Taha, H. A., Operation Research An Introduction, Prentice-Hall India, 2002.

[18] Tesauro, G. J., Practical Issues in Temporal Difference Learning, Machine Learning, Vol. 8, pp.257-277, 1992.

[19] V. S. Borkar, Reinforcement Learning in Markovian Evolutionary Games, http://www.tcs.tifr.res.in/~bor March 24, 2002.

[20] Wagner H. M., Principles of operations research with applications to managerial decisions, Prentice-Hall India, 1993.

[21] Zhang, W., Dietterich, T. G., Value Function Approximations and Job-Shop Scheduling, Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence, pp.1114-1120, 1995.

[W1] http://www-anw.cs.umass.edu/rlr/, Nov 8, 2002. 2002.

[W2] http://www.cse.unsw.edu.au/c̃s9417ml/RL1/algorithms.html, Nov 23, 2002.

[W3] http://web.mit.edu/konda/www/pub.html Nov 8, 2002.