


This is to certify that the

thesis entitled

HIERARCHICAL MULTI AGENT REINFORCEMENT LEARNING

presented by

Rajbala Makar

has been accepted towards fulfillment
of the requirements for

Master's degree in Computer Science & Engineering

M. R. Mahadevan

Major professor

Date JUNE 8 2000

SRIDHAR MAHADEVAN

# Hierarchical Multi Agent Reinforcement Learning

By

*Rajbala Makar*

A Thesis

Submitted to
Michigan State University
in partial fulfillment of the requirements
for the degree of

MASTER OF SCIENCE

Computer Science & Engineering

2000

# Abstract

## Hierarchical Multi Agent Reinforcement Learning

By

*Rajbala Makar*

Reinforcement Learning has been extensively studied as a generalized approach for sequential decision making problems involving actions of a single agent. Hence, single agent learning and the challenges and issues involved are very well understood. Recent focus has been on co-operative multi agent learning as that opens up a whole new dimension to machine learning and the type of problems it can be used for. The complexity introduced as a result of having a number of agents is a major deterrent for formulation of machine learning problems with multiple agents. Hierarchical learning makes it easier to solve large problems by taking advantage of abstraction. This work focuses on applying a hierarchical reinforcement learning approach to multi agent learning. The multi agent learning algorithm has been built upon the MAXQ framework for hierarchical reinforcement learning, as it naturally extends to the multi agent case. Each agent uses the same MAXQ hierarchy to decompose a task into sub-tasks. Coordination skills among agents are learned by configuring the Q nodes at the highest level of the hierarchy to represent the joint action space among multiple agents.

The algorithm is tested on a complex AGV (Automated Guided Vehicle) scheduling task, where four AGVs are available to carry material from a warehouse to one of four assembly stations, and to carry the finished assemblies back to the warehouse. Experiments show that the co-operative multi agent reinforcement learning algorithm performs considerably better than the case where the agents act selfishly and try to maximize their own rewards. The multi agent algorithm is also shown to perform better than heuristics usually used for AGV scheduling.

To My Mother (Smt. Narbada Devi)

# ACKNOWLEDGMENTS

I would like to take this opportunity to thank all the people around me who have made this thesis possible. First of all, I express my heartfelt gratitude for my advisor, Dr. Sridhar Mahadevan, who guided me throughout my time here with his constant support and encouragement, and spent a lot of his valuable time with me. I would also like to thank Dr. John Weng and Dr. R. L. Tummala for being in my thesis committee. I would like to thank Dr. James Bean, Dr. Yavuz Bozer and Dr. Abhijeet Gosavi for their help in getting literature on AGV scheduling. I would like to acknowledge the help of everyone in the Autonomous Agents lab at MSU, their helpful comments and the use of their machines. I would like to dedicate this thesis to my family who have constantly encouraged me and taught me how to dream. I do not know where I would be without their love, patience, understanding, and support. Finally, I am grateful to all my friends for providing a nice environment around me, and encouraging me in bad times.

# Table of Contents

# List of Figures

# Chapter 1

# Introduction

The recent focus on machine learning [1] as an approach for solving a wide variety of extremely diverse tasks shows great promise in this approach. The reward/punishment framework of reinforcement learning is general enough to be applicable in virtually every domain from robotics [10], to backgammon [15] to elevator scheduling [14] to channel allocation in cellular phones [17]. An agent learns incrementally by trial and error in the reinforcement learning framework, as opposed to the supervised learning framework, which assumes that there is an external teacher that provides good and bad examples to learn from. Supervised learning is an important form of learning, but is not adequate if an agent needs to learn from interaction with the environment. In interactive problems it might be impractical to obtain examples of desired behavior that are correct and represent all possible situations the agent might find itself in. In reinforcement learning, an agent is not told what actions to perform, but must discover what actions yield the maximum amount of reward, which is measured in terms of a scalar numerical signal which is generated according to the

current state and provides feedback to the agent.

## 1.1 Reinforcement Learning

Reinforcement learning considers the problem of a goal directed agent interacting with a possibly unknown and uncertain environment [2]. In many challenging cases, actions might effect future situations in which the agent might find itself in, in addition to providing immediate reward. This characteristic of reinforcement learning is called delayed reward. Thus, an agent is not explicitly told what are good actions or bad actions in reinforcement learning, but has to learn on the basis of positive or negative feedback from the environment at some point in time after taking a series of actions. This approach has solid foundations in human and animal psychology. It is a well known fact that pets can be taught to obey their master if they are scolded after doing something undesirable and given reward after behaving themselves. Another natural process which can be compared to the reinforcement learning approach is how babies learn to walk. They fall a number of times and get hurt before learning the correct way of walking. The most important distinguishing feature of reinforcement learning is that it uses training information to evaluate an action executed rather than instruct by giving correct actions. Thus, the goal of the agent is to maximize its long term reward, or *expected return.* The expected return might be a simple sum of rewards, but that would create problems in a continuous process task, where the final time step might be infinity. Another way of defining the long term reward would be the sum of discounted rewards the agent receives over the future. Thus the *expected*

*discounted return*,

$$R_t = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \ldots = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \tag{1.1}$$

where $0 \leq \gamma \leq 1$ is a parameter called the discount rate, is what the agent tries to maximize in the reinforcement learning framework.

## 1.2 Hierarchical Reinforcement Learning

The key to solving large problems which are intractable using the traditional reinforcement learning methods (also called flat reinforcement learning methods), is multi level decomposition of the problem and abstraction. Reinforcement learning in terms of abstract states or actions is termed as hierarchical reinforcement learning [43]. Abstracting the state space provides advantages in terms of modeling only those components of the state space which are relevant to a particular task. Action space can also be decomposed into the overall task, then the subtasks, up to the level of primitive actions. This provides the advantage that knowledge about certain subtasks which are common to different tasks can be reused once they are learned. Also, the structure of the overall task is captured in the decomposition. The structure can thus be used to limit the possible number of attainable states by performing a particular action. Abstraction leads to constraining the policy space, which speeds up learning considerably.

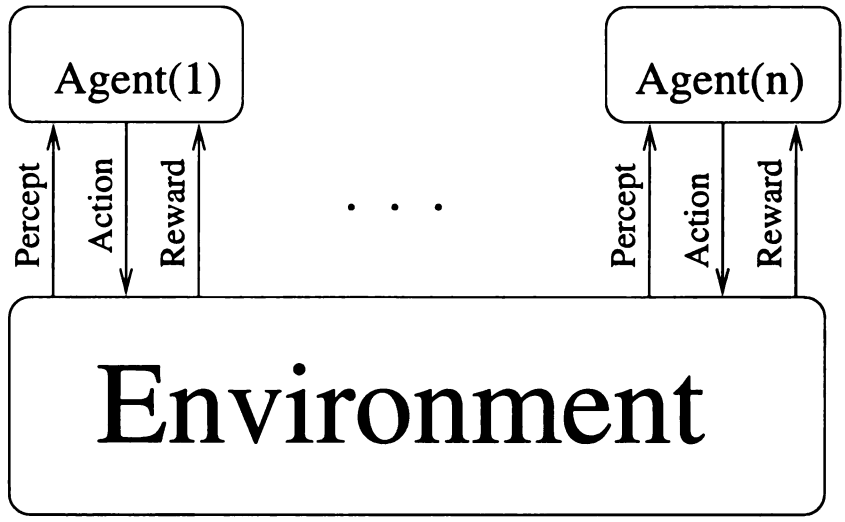



Figure 1.1: Dynamics of interaction in a multi agent system. The state of the environment might be changed by the actions of other agents while one agent is performing a task.

## 1.3 Multi Agent Reinforcement Learning

Many problems are complex and require the power of many independent agents working together towards a common goal. For example, any search task, such as foraging, is parallel in nature and would benefit by using multiple agents. One can think of other tasks where it might be essential to have multiple agents, like a predator prey task in which the prey might be stronger than one predator but could be captured by two predators [13], or robosoccer [20].

The complexity introduced as a result of having a number of agents is a major deterrent for formulation of machine learning problems with multiple agents, as the interactions among the agents have to be modeled in addition to the interaction between the agent and the environment. Hence, in a multi agent scenario, the interaction dynamics takes on the form shown in the figure 1.1. Changes in the environment can be brought about by actions of any of the agents. The state of the

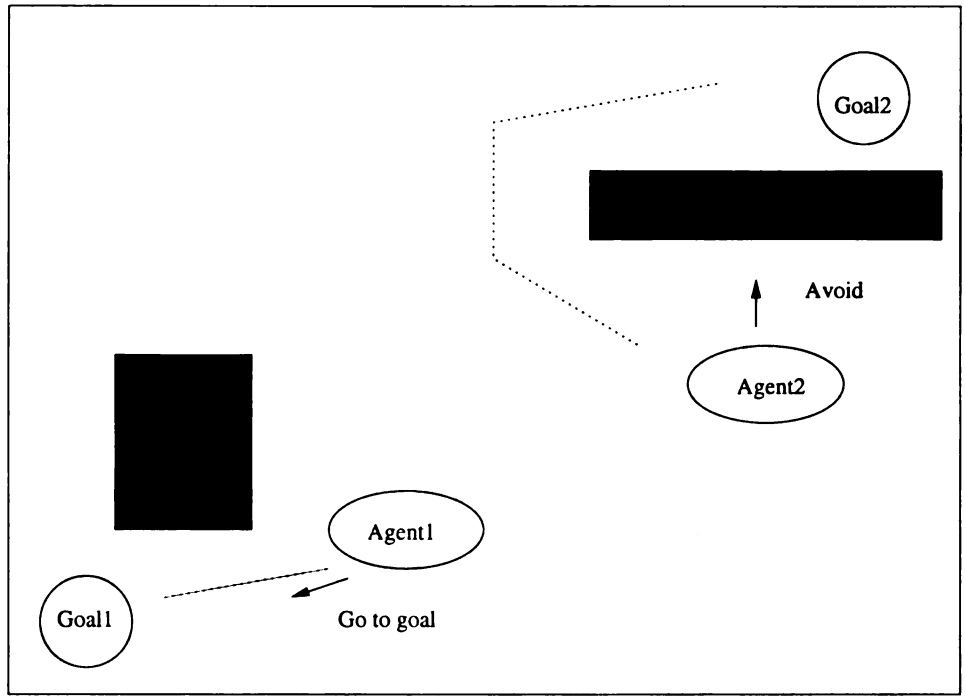



Figure 1.2: Example of a multi agent task.

environment might be changed by actions of other agents even while one agent is performing an action. For example, in figure 1.2, where the two agents are navigating to two different goal locations, agent 1 might reach its goal location while agent 2 is learning to avoid obstacles. In a flat framework, the agents would have to consider the state of both agents and actions being performed by both agents to define the state of the system completely. Modeling interactions between agents would make state and action space exponential in terms of the number of agents if the simple flat reinforcement learning approach is used. This is a huge burden and slows down learning. Even for problems where this kind of model is feasible, many agents performing random actions and trying to learn the best overall policy poses challenges in terms of learning. For example, again consider that two agents are jointly trying to get to opposite corners of a room. Note that the agents need to learn to reach

the goal locations while avoiding obstacles. This kind of learning might succeed in a static environment with fixed obstacles, which is is rarely the case. If the obstacles are moving, the low level learning of obstacle avoidance might give the agent rewards which might hinder with learning the overall task of reaching the goal (refer figure 1.2). Thus, low level interactions among agents might override the learning process and produce a kind of chaos.

These difficulties arise due to the fact that the agent has no knowledge of the fact that the goal of the agent is to reach a particular location, and obstacle avoidance is a behavior it has to turn on and off in order to achieve the goal without harming itself or anyone else. It is possible to give this knowledge to the system in terms of the hierarchy of the task. Thus, in hierarchical learning, the agent is always aware of the high level task even though it might be be currently performing some low level behavior. Thus, in this new hierarchical multi agent learning method, the agents learn co-ordinating their actions at the highest level of abstraction, thus hiding away the low level details, which leads to a very organized emergent behavior for the system as a whole. Joint action values need to be considered only at the abstract level in this framework, which ensures a tremendous big savings in terms of memory requirements.

We also face a decision about whether to model the joint state space of the agents for multi agent hierarchical learning. Modeling the joint state space might lead to the agents learning a very good combined policy, but in real problems, the memory and computational requirements are very high. If we consider a fairly complex task with, say, $10^5$ states, having four agents would mean $10^{20}$ states, which would be intractable. Thus, in our approach, the agent only models its own state space. The

only information it has about the other agents is the high level action being executed by them. Hence, each agent bases its decision on the next action it should execute upon its own state in the environment and the actions being executed by other agents. There is less communication overhead here, as agent only need to communicate their high level actions to other agents.

## 1.4 Thesis Outline

The outline of the thesis is as follows. Chapter 2 briefly reviews relevant literature on hierarchical reinforcement learning methods and explains the theory behind it. Chapter 3 describes the new hierarchical multi agent learning algorithm. A simple office environment task is described and results for multi agent hierarchical learning approach for the task are presented in Chapter 4. Chapter 5 deals with the the AGV scheduling task. It describes the task and issues involved in solving it, and presents experimental results. Chapter 6 presents the conclusions and a number of interesting ways this approach can be modified and its future research scope.

# Chapter 2

# Theoretical Foundations and Related Work

The theory behind reinforcement learning needs to be understood before we present our approach for multi agent reinforcement learning. Thus, we briefly describe the theory behind approaches to reinforcement learning, hierarchical reinforcement learning, and multi agent reinforcement learning.

## 2.1 Markov Decision Processes

Sequential decision making problems occur when an agents performance is dependent not on a single decision, but on a series of complex decisions taken one after another. A stochastic sequential decision making problem can be modeled as a Markov decision problem. Figure 2.1 is a simple example of a two state MDP, where one action is possible in each state. Execution of the single action is highly likely to bring the

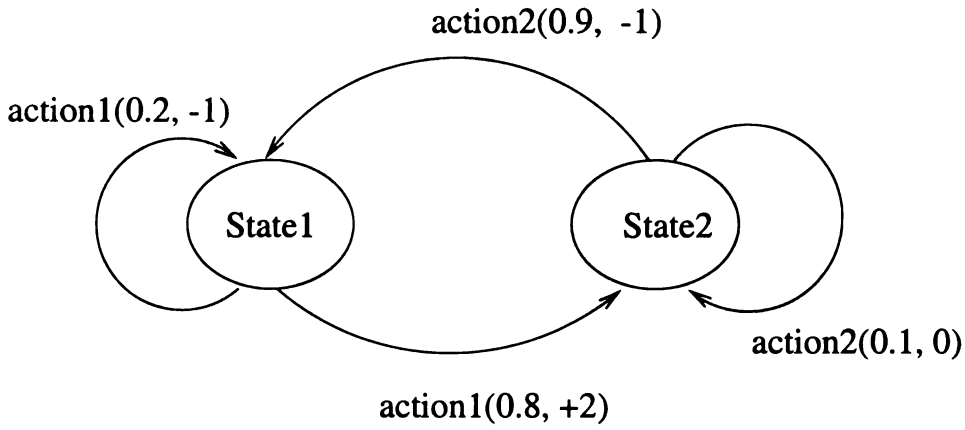



Figure 2.1: A simple two state Markov decision problem. The transition probability associated with the actions, and reward for performing the action in a particular state is shown in the bracket.

agent to the other state. But in many real life situations, there is a possibility that the action might fail. This is modeled with the help of the transition function, which says that the action taken in a state will take the agent to the other state, i.e. will succeed with probability $p$, and the agent would remain in the same state, i.e. the action would fail, with probability $(1 - p)$. A transition model gives the resulting distribution of states if action $a$ was executed in state $s$, for all state-action pairs. The Markov property is said to hold if the transition probabilities from any given state depend only on the state and not on previous history. Thus, MDPs are a model for sequential decision making under uncertainty, taking into account the current and future decision outcomes.

Formally, a MDP $M$ can be defined as a four tuple $(S, A, P, R)$, where

- $S$ is a set of environment states.
- $A$ is a finite set of actions.
- $P$ is a set of action dependent transition probabilities.

- $R$ is a reward function.

MDPs can be defined over continuous states and actions, but we are only considering the discrete case here. For each sequence of observed states $(x_0, x_1, \ldots x_{t-1}, x_t)$, in an MDP, the probability of being in the current state $y$ is given by

$$P(X_{t+1} = y | X_0 = x_0, a_0, X_1 = x_1, a_1, \ldots X_t = x_t, a_t = a) = P(X_{t+1} = y | X_t = x, a_t = a) \tag{2.1}$$

where $t = 0, 1, \ldots$ is the decision epoch.

This essentially says that the current state and action provide all the information for predicting the next state. Thus, if the Markov property is satisfied, knowledge of the current state is all that is required for making a decision.

### 2.1.1 Value of a Policy

An agent observes the state of the environment at decision epochs, which are discrete times in process evolution, and takes an action based on the state. The policy, $\pi$ is defined as the complete mapping from states to actions, which tells an agent what action to perform at each state. In order to identify a good policy, we should be able to distinguish between policies. This can be done with the help of the value function of a policy. The value function for a particular policy is defined as a mapping $V^\pi : S \rightarrow R$ which determines the expected utility of each state $s$, if actions are chosen according to the policy $\pi$. The reward function assigns a numeric value to a state or an action, based on the short term view, and the value function assigns a real valued number

based on a long term view. The expected total discounted reward of a policy $\pi(x)$ is

$$V_\gamma^\pi(x) = E_x^\pi(\sum_{i=0}^{\infty}(\gamma^t r_t)) \tag{2.2}$$

where $E_x^\pi$ is the expected value (under policy $\pi$) of starting from state $x$ and performing action $a$. for $0 \leq \gamma < 1$. The discount factor $\gamma$ measures the present value of one unit of reward received one epoch in the future. Discounting the reward accounts for the value of the reward as time passes. The aim is to maximize the value function over all states by identifying a policy $\pi$ where $V^\pi(x) \geq V(x)$ for all states $x$. This policy $\pi^*$ whose value function is greater than that of all other policies is called the optimal policy.

$$V^*(x) = max_\pi V_\gamma^\pi(x) \; \forall \; x \; \in \; S \tag{2.3}$$

$V^*$ refers to the optimal value function and $V_\gamma^\pi$ is the value function under policy $\pi$. Therefore,

$$V^*(x) = V_\gamma^{\pi^*}(x) \; \forall \; x \; \in \; S \tag{2.4}$$

where $\pi^*$ is the discounted optimal policy. More than one optimal policy might exist, but all these policies would define the same value function. The main algorithms which are used for solving a Markov decision problem are discussed next.

### 2.1.2 Dynamic Programming

Dynamic Programming (DP) is the traditional analytical approach for solving MDPs [5]. The complete model of the system, i.e. the transition probabilities and the reward

function must be known precisely to apply DP techniques. DP is based on the fact that the value function of a policy can be written recursively using value function prediction for the next state. Thus the value function can be rewritten as:

$$V^{\pi}(x) = r(x,a) + \sum_{y \in S}(P_{xy}(a)V^{\pi}(y)) \tag{2.5}$$

Thus the value of a state under a particular policy is the sum of the immediate reward of choosing an action dictated by the policy and the expected value of the resulting state. Value iteration and Policy iteration algorithms can be used for computing the optimal value function and policy. The optimal value function is computed and the optimal policy directly derived from this in the value iteration algorithm [6]. This is based on the Bellman optimality equation [45]:

$$V^{*}(x) = max_{a}(r(s,a) + \sum_{y} P_{x,y}(a)V^{*}(y)) \tag{2.6}$$

Thus, if the optimal value function is known, the optimal action in state $x$ is the action that maximizes the value function of a policy. The value iteration algorithm begins with an arbitrary value function and iteratively improves it until the value function between steps is arbitrarily close. The policy iteration algorithm evaluates a policy and chooses a greedy policy with the respect to the corresponding value function, thus obtaining an improvement over the original policy. Policy evaluation and improvement are carried out till the policy improvement step can no longer produce a better policy.

### 2.1.3 Reinforcement Learning

Reinforcement learning methods are online methods which require direct interaction with the environment in order to approximate the value function. The optimal value function is arrived at incrementally. This is in contrast to the DP methods, where updates across all states are performed simultaneously.

#### Temporal Difference Learning

The method of temporal differences (TD) is a general reinforcement learning framework for solving MDPs [16]. TD provides a method for updating the evaluation of a state based on the difference between the current evaluation of the state, and the evaluation of the successor state. Thus, updates occur only when an agent encounters a state.

#### Q Learning

Watkins Q-Learning method [7] utilizes the TD method to ensure that no knowledge of a system model is required to compute the optimal policy. The transition model need not be initially specified in this case, as the agent is embedded in the system, and can observe the actual state transitions. Thus, the action value of a state is updated each time the state is encountered according to the equation:

$$Q_{k+1}(x,a) = Q_k(x,a) + \alpha(r_{immi}(x,a) + max_b Q_{k+1}(y,b) - Q_k(x,a)) \qquad (2.7)$$

where $0 \leq \alpha \leq 1$ is the learning rate, $r_{immi}$ is the immidiate reward, $max_b Q_k(y, b) = V(y)$, and $Q_k(x, a)$ is the action-value of state x and action a before the update. Actions are chosen according to a greedy policy (actions with higher values are chosen), with a certain probability. Random actions are also chosen with some probability to allow the agent to explore the state space. Watkins showed that the Q values will asymptotically converge to optimal $Q^*$ values, if each action is executed infinite number of times in each state, and $\alpha$ is decayed as the learning progresses.

### 2.1.4 Discrete Event Model

Discrete event models have been used as the basis for simulation studies in a multitude of domains, ranging from manufacturing, queuing, networking, and transportation [41]. For the purpose of simulation optimization, the discrete time reinforcement learning framework has to be generalized to a discrete event framework. Time is explicitly modeled as a continuous variable, but the agent observes the environment and makes decisions only at certain discrete points (or decision epochs). In between these epochs, the state of the system can be changing in some complex way, but these changes may not provide the agent with any additional information. Furthermore, actions take non-constant time periods, modeled by some arbitrary time distribution. It is not possible to model the optimization of factory processes as a discrete-time MDP, without drastic loss of information (for example, Poisson demand processes or failure distributions require using real-valued stochastic time distributions). Formally, the evolution of the environment at decision epochs can be modeled as a semi-Markov

decision process (SMDP) [45].

### 2.1.5 Semi Markov Decision Processes

MDP models assume discrete time. Decisions occur at equally spaced points in time, after every state transition. For problems such as navigation and AGV scheduling, where the time taken to complete any action varies, a more natural model would be one which taken time into account. This leads us to the notion of Semi Markov Decision Processes (SMDP) [47]. Thus, in SMDPs, actions are allowed only at discrete points in time, but the state can change between actions. This is in contrast to an MDP, where state changes are solely due to actions being performed by the agent.

Formally, a semi Markov model of continuous time decision problems can be defined as a five tuple *(S, A, P, R, F)*, where

- $S$ is a set of environment states.
- $A$ is a finite set of actions.
- $P$ is a set of action dependent transition probabilities.
- $R$ is a reward function.
- $F$ is a function giving the probability of transition times for each state action pair.

P is a function mapping the elements of $S \times A$ into discrete probability distributions over $S$. $P(s'|s,a)$ denotes the probability that the environment will make

a transition from state s $\in S$ to state $s' \in S$ under action $a \in A$. $P$ describes the transitions at decision epochs only.

$F$ is a function where $F(t|s,a)$ is the probability that the next decision epoch occurs within $t$ time units after the agent chooses action $a$ in state $s$ at a decision epoch. Let $Q$ denote the joint probability that the system will be in state $s'$ for the next decision epoch, at or before $t$ time units after choosing action $a$ in state $s$. The expected transition time between decision epochs can be calculated from $Q$, which can be computed from $F$ and $P$ by

$$Q(t, s'|s, a) = P(s'|s, a).F(t|s, a) \tag{2.8}$$

The reward function for an SMDP is usually more complex than in the MDP model. This is because in addition to the fixed reward *k(s,a)* accrued due to an action performed at a decision epoch, an additional reward may be accumulated at the rate $\gamma(s', s, a)$ for the time the natural process remains in state $s'$ between decision epochs. The natural process may change state several times between decision epochs. Hence, the rate at which rewards are accumulated between decision epochs may vary.

## 2.2 Hierarchical Reinforcement Learning

The basic principle behind hierarchical learning is to constrain the policy space by using abstraction to increase learning speed and to reduce the memory and computational requirements [43]. There are two fundamental ways in which abstraction can be built into reinforcement learning. Temporal abstraction or abstraction in terms

of actions would mean that a set of primitive actions be considered a single abstract action. In other words, an abstract action would be a policy over primitive actions. Similarly, a number of states can be grouped together to form a single abstract state (for example, a room) in state abstraction. State or spatial abstraction, or function approximation, techniques develop mappings which are more compact than the general lookup table. Nearest-neighbor methods, neural networks, fuzzy logic, and many other approaches have been used to achieve such mappings.

### 2.2.1 Approaches to Hierarchical Reinforcement Learning

A variety of different approaches have been developed to achieve hierarchical reinforcement learning objectives. A brief overview of some of them is presented here.

#### Feudal Reinforcement Learning

The feudal reinforcement learning method by Dayan and Hinton [32] creates a managerial hierarchy in which high level managers learn how to assign tasks to their sub managers who learn how to satisfy them. The manager receives reinforcement from the external environment. Its actions consist of commands that it can give to the low-level learner. When the manager generates a particular command to the sub-manager, it must reward the sub-manager for taking actions that satisfy the command, even if they do not result in external reinforcement. The manager, learns a mapping from states to commands. The sub-manager learns a mapping from commands and states to external actions.

### Options

Another approach to hierarchical reinforcement learning is the options model described by Precup, Sutton, and Singh [9]. Their principle abstraction, essentially a closed loop sequence of actions, is termed a behavior. Behaviors can operate at different time scales, and can be composed of other behaviors. They incorporate a general closed-loop policy, instead of being simply a fixed sequence of actions, and can have a wide variety of completion criteria. Formally, an option is defined by the set of states in which it applies, the decision rule specifying actions to be performed, and a completion function. The decision rule is similar to a reinforcement learning policy, except that it can choose from among different behaviors as well as different primitive actions. Decisions can be based not only on the current state but on all states and actions from the start of the behavior. A completion function can specify completion after a fixed number of steps or after attainment of some subgoal.

Feudal reinforcement learning and the options method approaches that take advantage of temporal abstraction. Hierarchical Distance to Goal, HAM and MAXQ method take advantage of both spatial and temporal abstraction. These are described next.

### Hierarchical Distance to Goal

The Hierarchical Distance to Goal (HDG) algorithm by Kaelbling [3] imposes a set of landmark states and the agent reaches the goal state via this series of landmarks. Navigation through the environment consists of high-level navigation between land-

marks until the appropriate region is reached, at which point lower level actions take over to locate the goal.

### Hierarchical Abstract Machines

Hierarchical abstract machines (HAM) provide another mechanism for utilizing abstraction in reinforcement learning. The HAM model [33] uses nondeterministic finite state machines to constrain policies. For example, a simple machine could specify the actions "repeatedly go up or right" and when applied would constrain the overall policy in this manner. There are four types of HAM states: Action states execute an action in the environment. Call states execute another HAM as a subroutine. Choice states nondeterministically select the next machine state. Stop states halt execution and return control to the previous level. Application of a HAM to a MDP to yields an induced MDP. It is shown that the optimal policy for this induced MDP is equivalent to an optimal policy for the original MDP that is consistent with the HAM. An algorithm HAMQ-learning is developed which is shown to converge to the optimal action at every choice point in the induced MDP. The HAMQ-learning algorithm operates by keeping track of the accumulated reward and accumulated discount since the previous choice state. An extended Q-table is maintained, where Q-values are given for each environment state/machine state pair and choice point action. Upon transition to a choice state, the Q-value is updated using the cumulative reward and discount values.

### 2.2.2 The MAXQ Method for Hierarchical Reinforcement Learning

The MAXQ method for Hierarchical Reinforcement Learning by Dietterich is the basis for the multi agent hierarchical reinforcement learning algorithm developed here, hence we provide a detailed overview of the theory behind it [4].

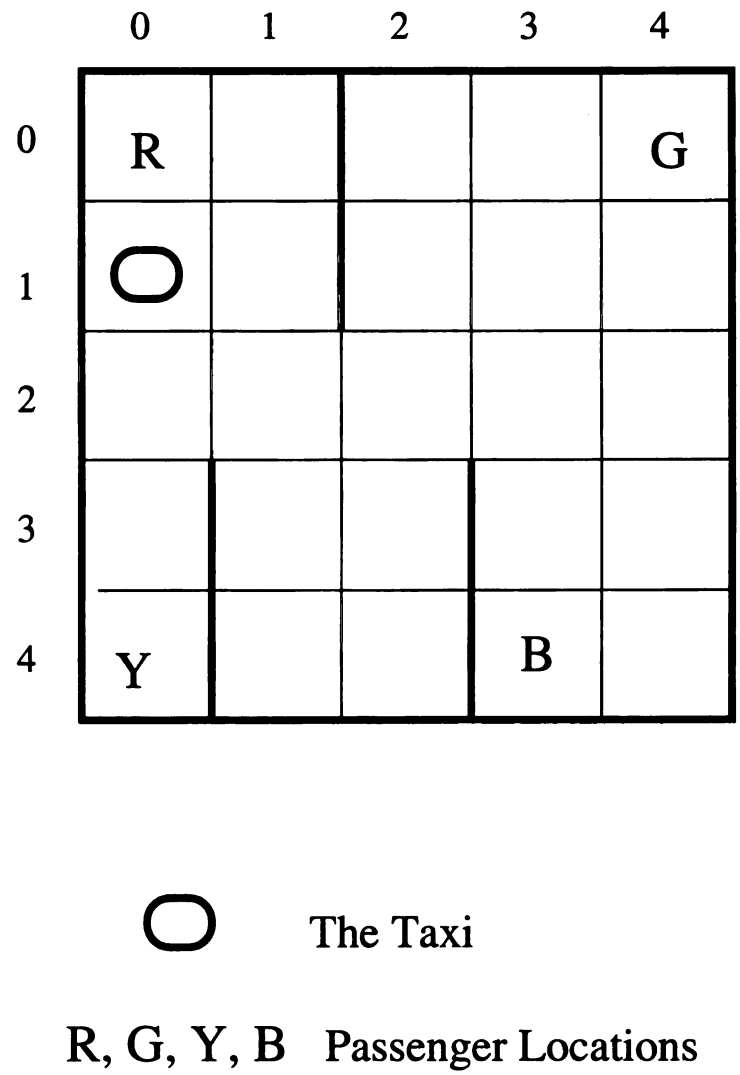



Figure 2.2: The taxi-cab domain.

The MAXQ method for Hierarchical Reinforcement Learning involves the use of a graph to store a distributed value function. The overall task is first decomposed into

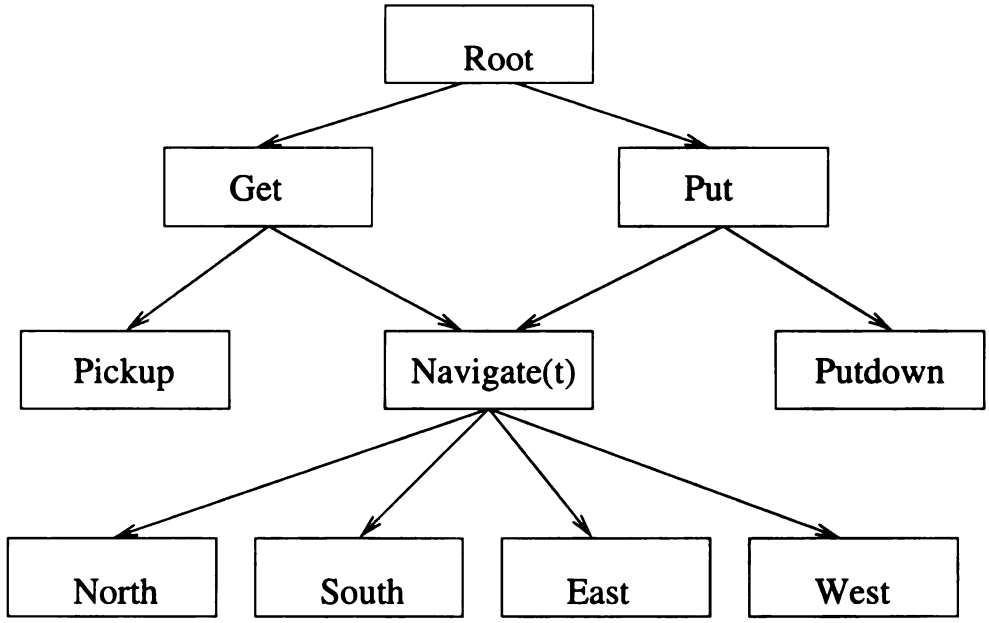



Figure 2.3: The task graph for the taxi-cab domain.

subtasks up to the desired level of detail, and the task graph constructed. A simple example would be the taxi-cab domain (figure 2.2) analyzed by Dietterich in his work. There are four special locations (denoted by R, G, Y, B) in a $5 \times 5$ grid world. A passenger needs to be transported from one of these four locations to another. Thus, the goal of the taxi agent is to go to the passenger location, pick him up, go to the destination location, and put him down. The task graph and MAX graph for this problem are shown in figures 2.3 and 2.4. This problem is considered to be episodic. The taxi is placed at a random location in each episode. Each episode starts with the passenger being at the start locations and ends with the passenger being deposited at the destination location.

Thus, formally, the MAXQ method decomposes an MDP M into a set of subtasks $M_0, M_1 \ldots M_n$ . Each subtask is a three tuple $(T_i, A_i, \overline{R_i})$ defined as:

$T_i(s_i)$ is a termination predicate which partitions the state space S into a set of active

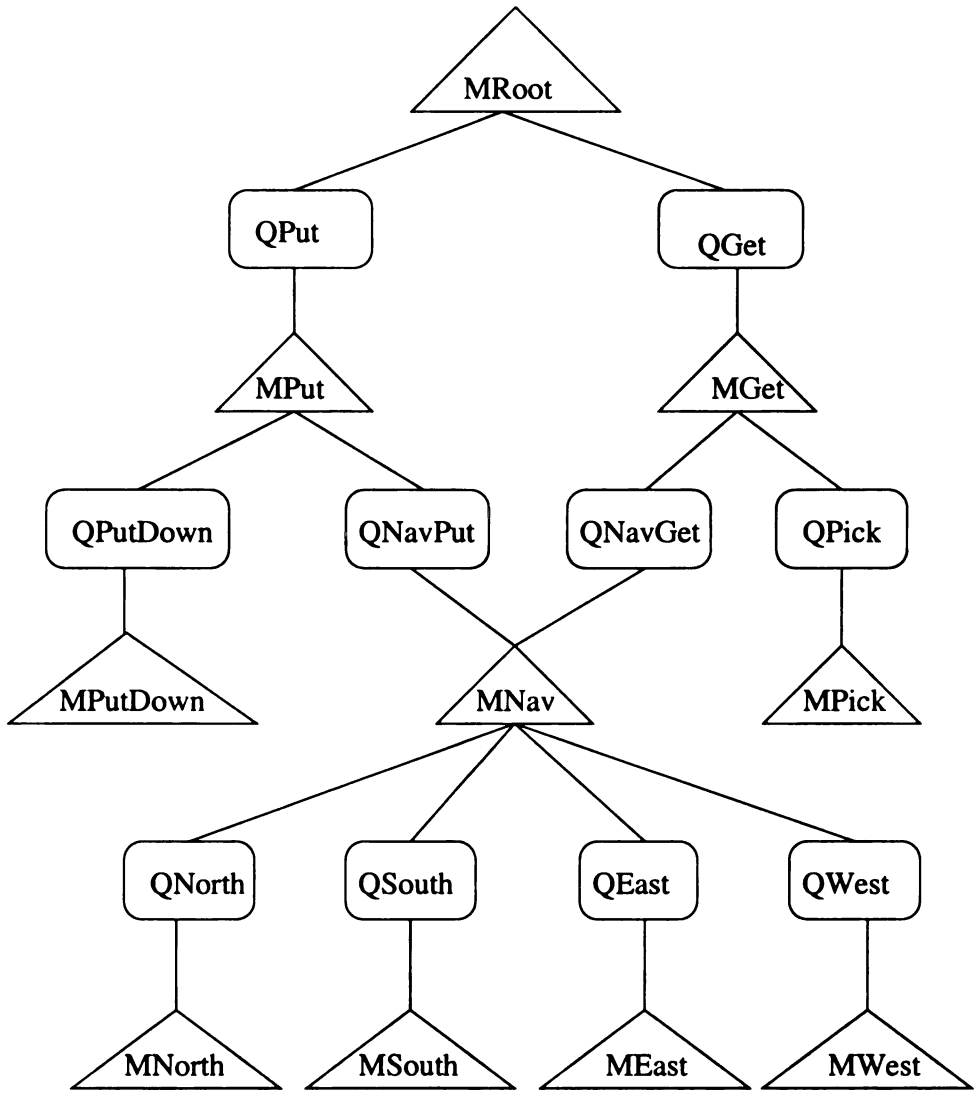



Figure 2.4: The MAXQ graph for the taxi-cab domain.

states $S_i$, and a set of terminal states $T_i$. The policy for subtask $M_i$ can only be executed if the current state $s$ is in $S_i$.

$A_i$ is a set of actions that can be performed to achieve subtask $M_i$ . These actions can either be primitive actions from $A_i$ , the set of primitive actions for the MDP, or they can be other subtasks, which can be denoted by their indexes $i$.

$\overline{R_i}$ $(s'|s,a)$ is the pseudo reward function, which specifies a pseudo reward for each transition from a state $s \in S_i$ to a terminal state $s' \in T_i$. This pseudo reward tells how desirable each of the terminal states is for this particular subtask. It is typically employed to give goal terminal states a pseudo reward of 0 and non goal terminal states a negative reward.

Each primitive action $a$ from M is a primitive subtask in the MAXQ decomposition, such that action $a$ is always executable, it terminates immediately after execution, and it's pseudo reward function is uniformly zero. The projected value function $V^\pi$ is the value of executing hierarchical policy $\pi$ starting in state $s$, and at the root of the hierarchy. The completion function $(C^\pi(i,s,a))$ is the expected cumulative reward, discounted back to the point where action $a$ begins execution, of completing subtask $M_i$ after invoking the subroutine for subtask $M_a$ in state $s$.

Figure 2.4 lists the MAXQ learning algorithm. The value function $V_t(i, s')$ in the

function MAXQ(MaxNode $i$, State $s$)

Let $seq = ()$ be the sequences of states visited while executing $i$
if $i$ is a primitive MaxNode
    execute $i$, receive $r$, and observe result state $s'$
    $V_{t+1}(i, s) = (1 - \alpha_t(i)).V_t(i, s) + \alpha_t(i).r$
    append $s$ into the beginning of $seq$
    else
    let $count = 0$
    while $T_i(s)$ is false do
        choose an action $a$ according to the current exploration policy $\pi(i, s)$
        let $childseq$ = MAXQ(a, s), where $childseq$ is the sequence of states visited
            while executing action $a$.
        observe result state $s'$
        let $a^* = argmax_{a'}[\overline{C}_t(i, s', a') + V_t(a', s')]$
        let $N = length(childseq)$
        for each $s$ in $childseq$ do
            $\overline{C}_{t+1}(i, s, a) = (1 - \alpha_t(i))\overline{C}_t(i, s, a) + \alpha_t(i)\gamma^N[\overline{R}_i(s') + \overline{C}_t(i, s', a^*)$
              $+ V_t(a^*, s')]$
            $C_{t+1}(i, s, a) = (1 - \alpha_t(i))C_t(i, s, a) + \alpha_t(i)\gamma^N[C_t(i, s', a^*) + V_t(a^*, s')]$
        append $childseq$ onto the front of $seq$
        $s = s'$
        end
    end
return $seq$
end MAXQ

Figure 2.5: The MAXQ learning algorithm.

algorithm is calculated with the decomposition equations:

$$V_t(i,s) = \begin{cases} max_a Q_t(i,s,a) & if\ i\ is\ composite \\ V_t(i,s) & if\ i\ is\ primitive \end{cases}$$

$$Q_t(i,s,a) = V_t(a,s) + C_t(i,s,a) \tag{2.9}$$

The definition of policy and value function with respect to the hierarchical case is now presented. A hierarchical policy $\pi$ is a set containing a policy for each of the subtasks in the problem: $\pi = \{\pi_0 \ldots \pi_n\}$. Similarly, the projected value function in the hierarchical case, denoted by $V^{\pi}(s)$, is the value of executing hierarchical policy $\pi$ starting in state $s$ and starting at the root of the task hierarchy. A *recursively* optimal policy for MDP $M$ with MAXQ decomposition $\{M_0 \ldots M_k\}$ is a hierarchical policy $\pi = \{\pi_0 \ldots \pi_k\}$ such that for each subtask $M_i$ the corresponding policy $\pi_i$ is optimal for the SMDP defined by the set of state $S_i$, the set of actions $A_i$, the state transition probability function $P^{\pi}(s', N|s, a)$, and the reward function given by the sum of the original reward function $R(s'|s, a)$ and the pseudo-reward function $\overline{R}_i(s')$. The MAXQ learning algorithm has been proven to converge to $\pi_r^*$, the unique recursively optimal policy for MDP $M$ and MAXQ graph $H$, where M = $(S, A, P, R, P_0)$ is a discounted infinite horizon $MDP$ with discount factor $\gamma$, and $H$ is a MAXQ graph defined over subtasks $\{M_0 \ldots M_k\}$.

## 2.3 Multi Agent Learning

Multi-agent reinforcement learning is an extremely challenging problem. Complexity is introduced as a result of having a number of agents, as the interactions among the agents have to be modeled in addition to the interaction between the agent and the environment. Various approaches have been tried to tackle this problem. Most of these approaches extend "flat" algorithms, such as Q-learning, to the multi-agent case. A review of such algorithms is presented next. Littman [12], and Hu and Wellman [18] have studied the the framework of Markov games for multi agent learning. Markov games is an extension of game theory to MDP like environments. Littman studies a minmax-Q learning algorithm using a simple two player game where both players have opposing goals. Thus, there is a single reward function which one agent tries to minimize and another tries to maximize. In this framework, the *max* operator in the update step of a standard Q learning algorithm is replaced by the minmax operator that is evaluated by linear programming. Hu and Wellman [18] extend this work to a broader framework, and prove the convergence of their method to a Nash equilibrium under specified conditions. Tan [13] studied the extension of flat reinforcement learning to the multi agent case. He studies the impact of sharing observation, policies or episodes, and having joint tasks for a multi agent predator prey problem. The agents are shown to perform better when knowledge and information is communicated between agents. However, these approaches are extensions of "flat" Q learning and are not likely to scale as the state-action value space quickly becomes intractable.

Crites and Barto [14] apply reinforcement learning for group elevator control by using a global reinforcement signal. This is based on the idea that if each member of a team of agents employs a reinforcement learning algorithm, a new collective learning algorithm emerges for the team as a whole. Each agent is responsible for controlling one elevator car in this case. Accumulated cost is updated incrementally after every passenger arrival, passenger transfer and car arrival event. The amount of cost accumulated between events is the same for all cars since they share the same objective function, but the amount accumulated between decisions is different as the cars make decisions asynchronously. Each car has a storage location where the total discounted cost it incurred since it's last decision is accumulated. The joint state space is modeled here too. Stone and Veloso [20] propose a method for scaling multi-agent learning by learning value functions over a small set of action based features instead of states. Thus, the complexity of the learning task is reduced as a small feature space is constructed, which partitions the state into regions. Each agent learns to act only within it's own partition. The feature set here is problem dependent. Behavior based multi agent robotics has been studied by Balch [22] and Mataric [11]. In Balch's behavior based formation control, each robot maintains its position in the formation depending on the locations of the other robots. Behaviors such as avoid-obstacle, avoid-robot, move-to-goal and maintain formation are implemented to enable the agents to perform the desired task. Mataric has proposed a method for minimizing the multi agent learning space through the use of behaviors and conditions, and shaping reinforcement with heterogeneous reinforcement functions and progress estimators, which take advantage of the implicit domain knowledge in order to accelerate learning.

States are clustered into conditions, which are the necessary and sufficient subsets of state required for triggering the behavior sets. These approaches are dependent on the behaviors selected initially.

## 2.4 Discussion

A wide variety of methods have been tried to solve the problem of multi agent learning with some significant successes. However, the area of hierarchical reinforcement learning has not yet been explored as a vehicle for utilizing the hierarchy to help with structuring the problem or to improve scalability, which are the key benefits of hierarchical learning. The MAXQ method for Hierarchical Reinforcement Learning is extensible to the multi agent case. Each agent can use the MAXQ hierarchy to decompose a task into sub-tasks. Coordination skills among agents can learned by configuring the Q nodes at the highest level of the hierarchy to represent the joint action space among multiple agents. The value function is propagated upwards from the lower level nodes whenever a high level node needs to be evaluated. This enables the agent to simultaneously learn subtasks and high level tasks. Thus, by using this method, the agent learns the co-ordination skills and the individual low level tasks and subtasks all at once. These ideas have been used in the multi agent hierarchical learning algorithm which is the topic of this thesis. The next chapter explains the new hierarchical multi agent learning approach proposed here.

# Chapter 3

# Co-operative MAXQ: A Hierarchical Multi Agent Reinforcement Learning Method

The MAXQ method stores the value function in a distributed way in all nodes in the subtask graph. The value function is propagated upwards from the lower level nodes whenever a high level node needs to be evaluated. This enables the agent to simultaneously learn subtasks and high level tasks. Thus, by using this method, the agent learns co-ordination skills and individual low level tasks and subtasks all at once. However, it is necessary to generalize the MAXQ framework to make it more applicable to multi agent learning. A broad class of multi agent optimization tasks, such as AGV scheduling, can be viewed as discrete-event dynamic systems. For such tasks, the termination predicate used in MAXQ has to be redefined to take care of

the fact that the completion of certain subtasks might depend on the occurrence of an event rather than just a state of the environment.

## 3.1 Revisiting the Termination Predicate

In the original MAXQ formulation, the termination predicate is defined to be true or false depending on whether the particular state $s$ is in the set of active states $S_a$ or the set of terminal states $S_t$. For some types of problems, when a task terminates might depend on some factors in the environment other than just the state. In many cases, the exact same state of the environment may result when a subtask is completed by the agent, as when it was started. For example, consider Dietterichs taxi-cab domain which was explained in chapter 2. Now make an assumption that the task is not episodic and that passengers can appear at one the the four R, G, B, Y locations randomly. Also, the passenger disappears after being dropped at the destination location. Now consider that the MAXQ hierarchy contains the subtask *deliver passenger from source location to destination location* (there might be other subtasks like refuel the taxi etc.), and the agent chooses to perform it. Suppose the passenger source location is R and destination location is G, and that the taxi is at the passenger destination location G currently. Thus, the state of the environment at the onset of subtask *deliver passenger from R to G* is: taxi at G, passenger at R and no passenger in taxi. While the taxi is performing the chosen subtask, another passenger might come at R, waiting to be taken to some destination. Thus, when the taxi completes the subtask, the state is still: taxi at G, passenger at R and no passenger

in taxi. Thus, without special care, these kinds of conditions might result in the agent not exiting from a subtask to be able to choose another high level task at the correct time. One way to avoid this situation might be to construct the task graph in such a way that these conditions do not occur. If we separate the subtask *deliver passenger from source location to destination location* into two subtasks, namely, *go to source location and pick up passenger* and *go to destination location and put down passenger* , it would again become possible to unambiguously define termination predicates with respect to states for these subtasks. But this approach would obviously require a lot of care on the part of the person constructing the hierarchy. It might also mean more subtasks than necessary, which would increase the memory requirements and the complexity of the algorithm.

Another way to deal with this problem would be to separate the termination predicate into two parts, one for starting a particular subtask, and another for ending it. Thus the start termination predicate $T_{start(i)}(s)$ is the regular termination predicate as defined in the MAXQ formulation, which divides the states into active and non-active states, where active states mean that the subtask can be attempted by an agent, and non-active means that the subtask cannot be attempted in the current state. Once an agent starts a subtask after seeing that the state was present in the active state set, the end of this task depends on the end termination predicate, $T_{end(i)}$ which is governed by the occurence of an event. Thus, examining the earlier example, once an agent $j$ has started the task *deliver passenger from source to destination*, it will end only when the event *putdown passenger at destination by agent* $j$ occurs. This kind of termination predicate is not significantly more difficult to incorporate in

the MAXQ algorithm, and helps in making sure that inconsistencies are eliminated. Thus the start termination predicate $T_{start(i)}(s)$ is a predicate that partitions the set of states $S$ into a set of active states, $S_a$ and a set of non-active states $S_t$. The policy for subtask $M_i$ can be executed only if the current state $s$ is in $S_a$. The end termination predicate $T_{end(i)}$ is a predicate that signals when a subtask $i$ is completed. It is initialized to 0 but changes to 1 when the subtask completion event occurs. $T_{end(i)}$ is reinitialized to 0 once the agent observes its value.

## 3.2 State Abstraction

The MAXQ state abstraction is used, which gives us a compact way of representing the Completion functions, and speeds up the algorithm. This abstraction is based on five premises, as defined in [8].

- **Max Node Irrelevance** The set of state variables which are irrelevant to a max node are ignored. Let $M_i$ be a Max node in a MAXQ graph H for MDP M. A set of state variables Y is irrelevant to node $i$ if the state variables of M can be partitioned into two sets X and Y such that for any abstract hierarchical policy $\pi$ executed by the descendents of $i$, the state transition probability distribution at node $i$ can be factored into the product of two distributions, one for the states in X, and another for the states in Y, and for any pair of states differing only in the variables in set Y, and any child action a, $V^{\pi}(a, s_1) = V^{\pi}(a, s_2)$ and $\overline{R}_i(s_1) = \overline{R}_i(s_2)$.

- **Leaf Irrelevance** The set of state variables Y which are irrelevant to a primitive action (or leaf node) are ignored. This occurs when the expected value of the reward function does not depend on the values of the state variables in Y.

- **Result Distribution Irrelevance** A set of state variables $Y_j$ is irrelevant for the result distribution of action $j$ if, for all abstract policies $\pi$ executed by node $j$ and its descendents in the MAXQ hierarchy, the following holds: for all pairs of states $s_1$ and $s_2$ that differ only in their values for the state variables in $Y_j$, $P^{\pi}(s', N|s_1, j) = P^{\pi}(s', N|s_2, j)$ for all $s'$ and N.

- **Termination** This occurs when a subtask is guaranteed to cause its parent task to terminate in the goal state. Thus, if $M_i$ is a task in the MAXQ graph such that for all states $s$ where the goal termination predicate is true, the pseudo reward function $\overline{R}_i = 0$, then for any policy executed at node $i$, the completion cost is zero and does not need to be explicitly represented.

- **Shielding** Let $M_i$ be a task in a MAXQ graph, and $s$ be a state such that for all paths from the root of the graph down to node $M_i$, there exists a subtask $j$ (possibly equal to $i$) whose termination predicate $T_j(s)$ is true, then the Q nodes of $M_i$ do not need to represent completion function values for state $s$.

If these conditions are examined with reference to the start termination predicate with the new definition of the termination predicate, they hold true without any changes. This is because the definition of the start termination predicate $T_{start(i)}(s)$ is the same as before. Thus, all these five kinds of state abstraction can be taken

advantage of in the multi agent MAXQ algorithm, even with the new definition of the termination predicate.

## 3.3 The Multi Agent MAXQ Algorithm

The most salient feature of this algorithm is that the top level (the level immediately below the root) of the hierarchy stores the completion function (C) values for the joint (abstract) actions of all agents. This completion function $C_j(i, s, a^1, a^2 \ldots a^j \ldots a^n)$, where there are $n$ agents, is defined as the expected discounted reward of completion of subtask $a^j$ by agent $j$, after invoking the subroutine for subtask $i$ in the context of the other agents performing subtasks $a^1 \ldots a^n$.

The recursive MAXQ algorithm is used for learning the C values. Thus, an agent starts at the root task and successively chooses subtasks till it gets to a primitive action. The primitive action is executed, the reward observed, and the leaf V values updated. When a subtask terminates, C($i$,$s$,$a$) values are updated for all states visited during the execution of that subtask. Similarly, when one of the tasks at the level just below the root max node terminates, the $C(i, s, a^1, a^2 \ldots a^j \ldots a^n)$ values are updated according to the MAXQ learning algorithm. The new multi agent MAXQ learning algorithm for each agent (say agent number $j$) is shown in figure 3.1. where $a^j$ is the action being performed by the current agent. We assume here that the high level actions $a^1, a^2 \ldots a^n$ being performed by all the agents and the agent number of the current agent is available globally. The decomposition equations used for calculating the projected value function V now take on the form:

$$V_t(i,s) = \begin{cases} max_{a^1 \ldots a^n} Q_t(i,s,a^1 \ldots a^j \ldots a^n) & if\ i\ is\ composite \\ V_t(i,s) & if\ i\ is\ primitive \end{cases}$$

$$Q_t(i,s,a^1 \ldots a^j \ldots a^n) = V_t(a^j,s) + C_t(i,s,a^1 \ldots a^j \ldots a^n) \tag{3.1}$$

for the highest level of abstraction, where the joint action values are considered. The decomposition equations for the lower levels in the hierarchy do not change.

```
function MAXQ(MaxNode i, State s)

Let seq = () be the sequences of states visited while executing i
 if i is a primitive MaxNode
      execute i, receive r, and observe result state s'
     V_{t+1}(i,s) = (1 − α_t(i)).V_t(i,s) + α_t(i).r
     append s into the beginning of seq
else
     let count = 0
     if T_{start(i)}(s) is false do
          while T_{end(i)}(s) is false do
               choose an action a for the current agent according to the current
                    exploration policy.
               update the global action vector by replacing the action being performed
                    by agent j by action a, if action a is a highest level abstract action.
               let childseq = MAXQ(a,s), where childseq is the sequence of states
                    visited while executing action a.
               observe result state s'
               if i = root task do
                  UpdateJointC(i,s,s',a,childseq)
               else do
                  UpdateC(i,s,s',a, childseq)
               append childseq onto the front of seq
               s = s'
               end \\ while
          end \\ if
     end \\ else
return seq
end MAXQ
```

Figure 3.1: The multi agent MAXQ algorithm

function UpdateJointC($i, s, s^{'}$, $a$, *childseq*)
let $a^{*} = argmax_{a^{'}}(\overline{C}_t(i, s, a^1 \ldots a^{'} \ldots a^n) + V_t(a^{'}, s^{'})$ where $a^{'}$ is the subtask being executed by the current agent $j$.
let $N = length(childseq)$
for each $s$ in *childseq* do

$$\overline{C}_{t+1}(i, s, a^1 \ldots a^j \ldots a^n) = (1 - \alpha_t(i))(\overline{C}_t(i, s, a^1 \ldots a^j \ldots a^n) + \alpha_t(i).\gamma^N[\overline{R}_i(s^{'}) + (\overline{C}_t(i, s^{'}, a^1 \ldots a^* \ldots a^n) + V_t(a^*, s^{'})]$$

$$C_{t+1}(i, s, a^1 \ldots a^j \ldots a^n) = (1 - \alpha_t(i)).C_t(i, s, a^1 \ldots a^j \ldots a^n) + \alpha_t(i).\gamma^N[C_t(i, s^{'}, a^1 \ldots a^* \ldots a^n) + V_t(a^*, s^{'})]$$

$N = N - 1$
end \\ for
end UpdateJointC

function UpdateC($i, s, s^{'}$, $a$, *childseq*)
let $a^{*} = argmax_a^{'}(\overline{C}_t(i, s^{'}, a^{'}) + V_t(a^{'}, s^{'})$
let $N = length(childseq)$
for each $s$ in *childseq* do

$$\overline{C}_{t+1}(i, s, a) = (1 - \alpha_t(i)).\overline{C}_t(i, s, a) + \alpha_t(i).\gamma^N[\overline{R}_i(s^{'}) + \overline{C}_t(i, s^{'}, a^*) + V_t(a^*, s^{'})]$$

$$C_{t+1}(i, s, a) = (1 - \alpha_t(i)).C_t(i, s, a) + \alpha_t(i).\gamma^N[C_t(i, s^{'}, a^*) + V_t(a^*, s^{'})]$$

$N = N - 1$
end \\ for
end UpdateC

Figure 3.2: The algorithm for updating the joint and individual completion function values at abstract levels for the multi agent MAXQ algorithm

# Chapter 4

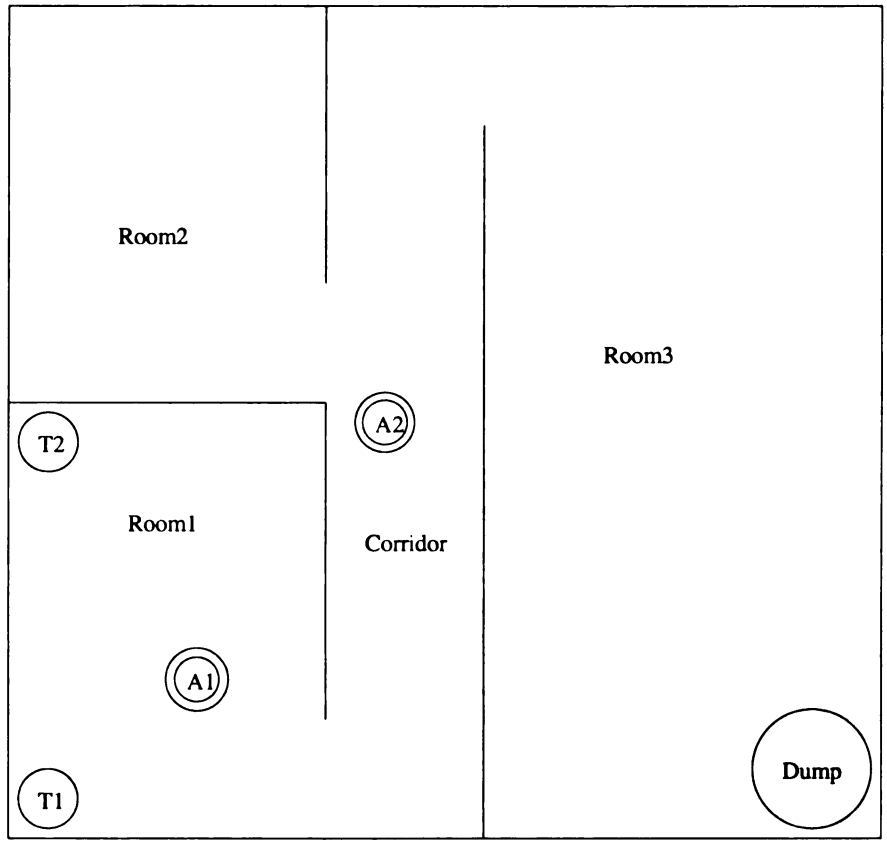
# A Multi Agent Robot Task

To test of the new multi agent MAXQ method, experiments were conducted on a simple robot navigation domain to verify its effectiveness in learning co-operation skills. We present the results for a trash collecting robot task in this chapter.

## 4.1 The Task

Consider the case where a robot is assigned the task of picking up trash from trash cans over an extended area and accumulating it into one centralized trash bin, from where it might be sent for recycling or disposed. This is a task which can be parallelized, if we have more than one agent working on it. An office (rooms and connecting corridors) type environment as shown in figure 4.1 was used for conducting experiments with this task. A1 and A2 represent the two agents in the figure.

This trash depositing task was decomposed into subtasks and the task graph which emerged is shown in figure 4.2. The task graph is then converted to the MAXQ graph,




T1: Location of one trash can.

T2: Location of another trash can.

Dump: Final destination location for depositing all trash.

Figure 4.1: The Environment used for the multi agent trash depositing task.

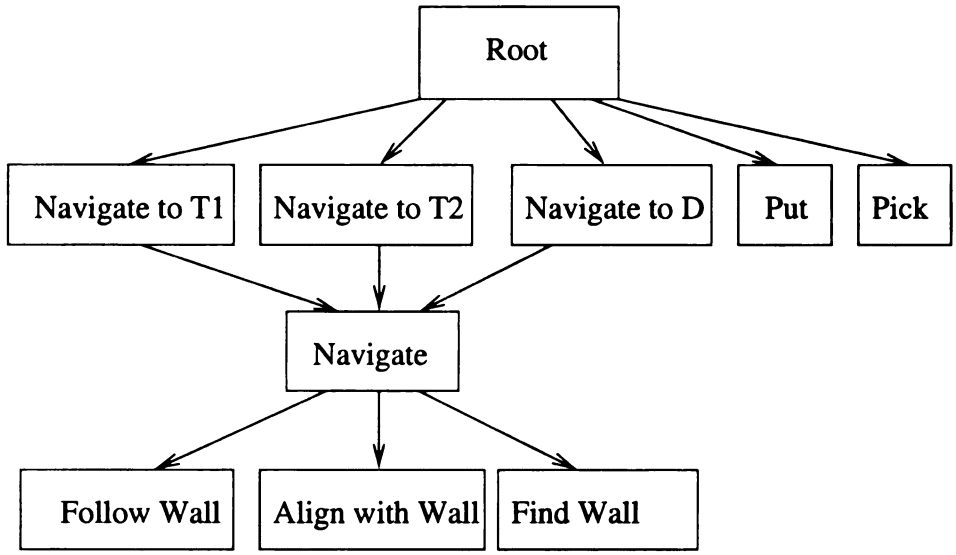



Figure 4.2: The task graph used for the multi agent trash depositing task.

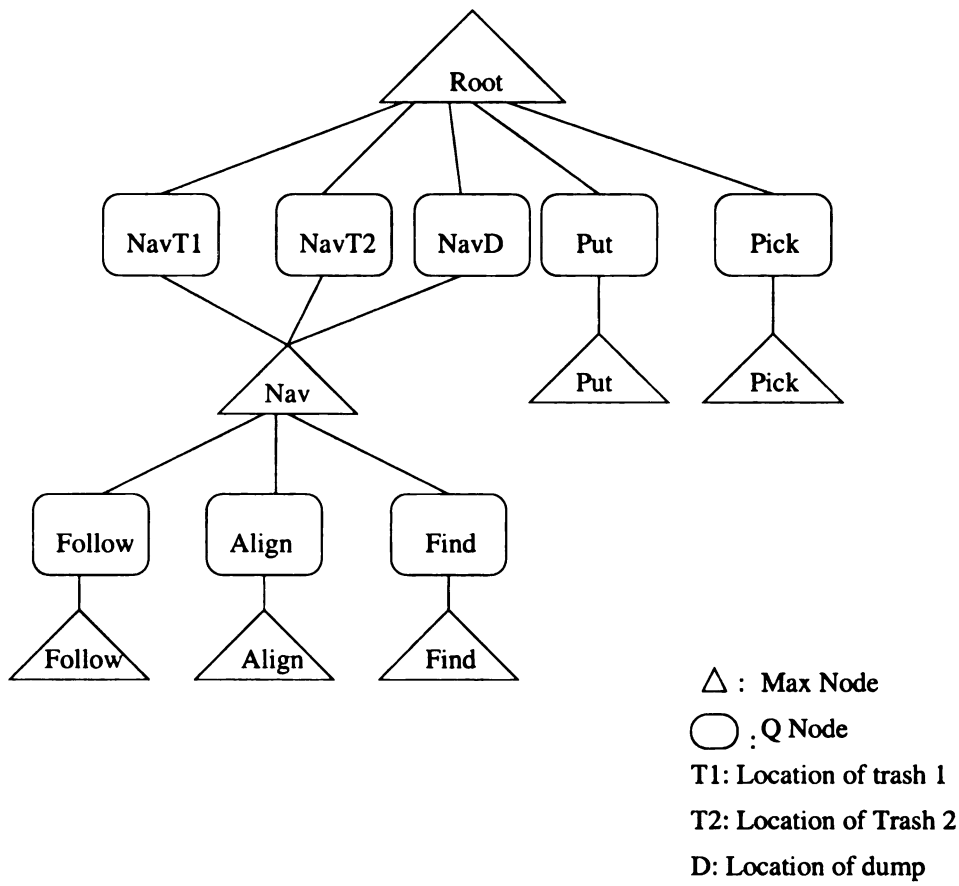



Figure 4.3: The MAXQ graph used for the multi agent trash depositing task.

which is shown in figure 4.3. The notion of termination predicates as defined in the MAXQ framework is used "as is" in this task, as the state of the environment can indeed be used as a correct indicator of the completion of tasks here.

In the single agent scenario, one robot starts in the middle of Room 1 and learns the task of picking up trash from T1 and T2 and depositing it into Dump. The goal state is reached when trash from both T1 and T2 has been deposited in Dump. The robot starts all over again with the initial configuration at this point, in order to improve it's policy. Two experiments are conducted in the multi agent case. The two robots model the joint action space at the highest level in both cases, but in the first scenario, the robots model the joint state space of both agents, and in the second scenario, each agent ignores the state of the other agent. The state space here is the orientation of the robot (N,S,W,E), and another component based on its percept. We assume that a ring of 16 sonars would enable the robot to find out whether it is in a corner, (with two walls perpendicular to each other on two sides of the robot), near a wall (with wall only on one side), near a door (wall on either side of an opening), in a corridor (parallel walls on either side) or in an open area (the middle of the room). Thus, each room is divided into 9 states, and the corridor into 4 states. Thus, we have $(9 \times 3) + 4) \times 4$, or 124 locations for a robot. Also, trash 1 can be at T1, with robot, or at D, and trash 2 can be at T2, with robot, or at Dump. Thus the total number of environment states is $124 \times 3 \times 3$, or 1116 for the single agent case. Going to the two agent case would mean that the trash can be at either T1/T2, Dump, or with one of the two robots. Thus the state space would now be $124 \times 124 \times 4 \times 4$, or $\approx 24 \times 10^4$. The environment is such that it is fully observable with this state

decomposition, as the direction which the robot is facing, in combination with the percept, with the assumption that the agent knows what room it is in, gives us a unique value for each state. The primitive actions considered here are behaviors to find a wall in one of four directions, align with the wall on left or right side, follow wall, enter or exit door, align south or north in the corridor, or move in the corridor.

## 4.2 Results

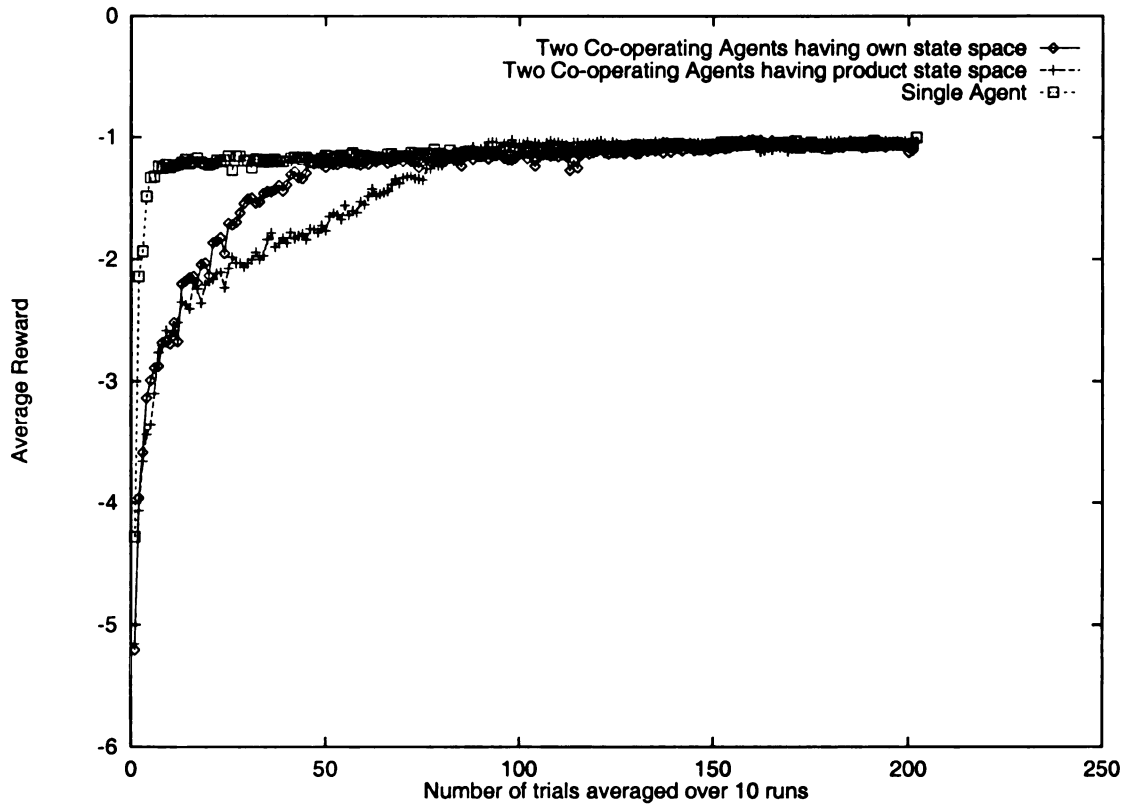



Figure 4.4: The average reward accumulated from start state to the goal state.

Experiments were conducted and the learning curve and performance of the agents compared for the single agent, multi agent case with joint state space, and multi agent case with own state space. The average reward accumulated while traveling from start

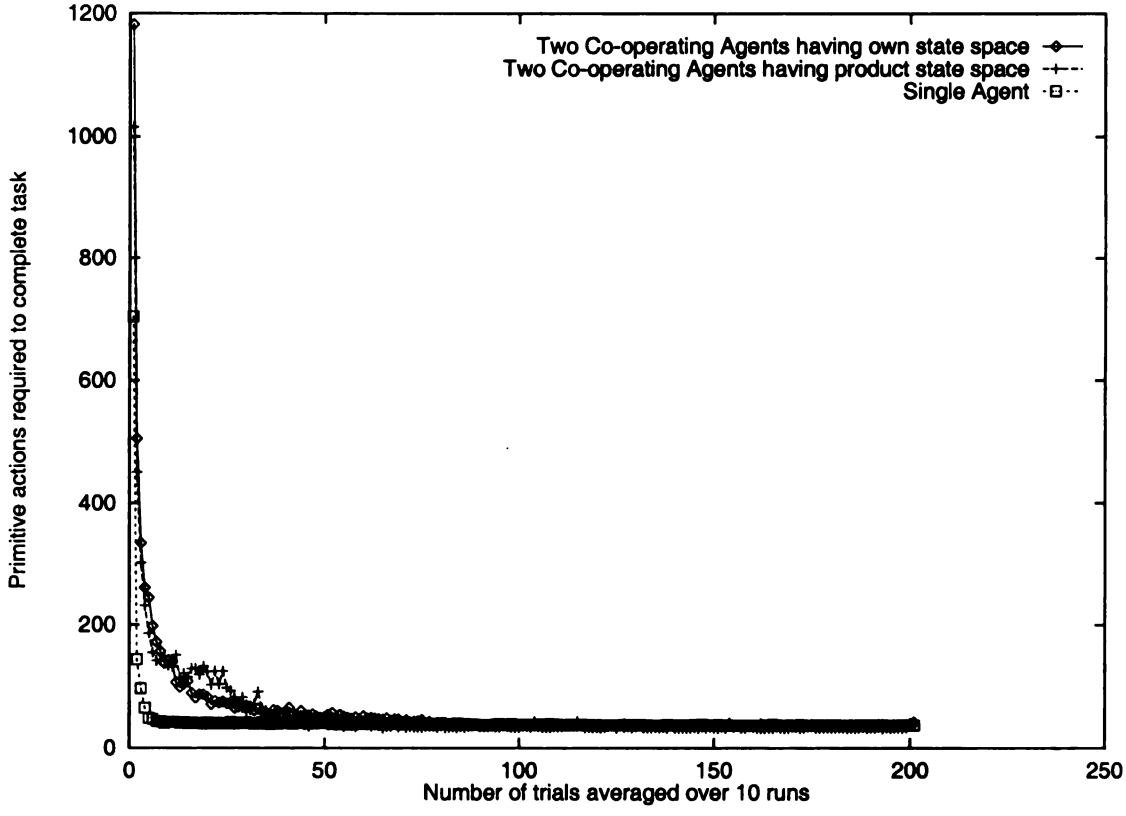



Figure 4.5: The number of primitive actions required to get to the goal state.

state to goal state is shown in figure 4.4, and the number of primitive actions required to reach the goal state in figure 4.5.

As seen from the figures 4.4 and figure 4.5, all three methods converge in a similar manner, and are able to finish the task of depositing both trash 1 and trash 2 from the individual cans to the trash dump. The difference is seen as the learned policies are examined. At each step, the actions and subtasks performed by the agents are stored, in order to look at the final policy learned by the agents. In the single agent case, the task is performed in a sequential manner and the final abstract policy is shown in figure 4.6.

**Learned Policy for Single Agent**

*root*
    *navigate to trash 1*
        *go to location of trash 1 in room 1*
    *pick trash 1*
    *navigate to bin*
        *exit room 1*
        *enter room 3*
        *go to location of dump in room 3*
    *put trash 1 in dump*
    *navigate to trash 2*
        *exit room 3*
        *enter room 1*
        *go to location of trash 2 in room 1*
    *pick trash 2*
    *navigate to bin*
        *exit room 1*
        *enter room 3*
        *go to location of dump in room 3*
    *put trash 2 in dump*
*end*

Figure 4.6: Learned policy for single agent performing trash collecting task using the MAXQ hierarchical decomposition.

On examining the policy in the case where two agents are acting in parallel (refer figure 4.7), it is seen that the total number of actions required to finish the task by each robot are half of those in the single agent case, thus the time taken to finish the task is reduced by half.

**Learned Policy for Agent 1**

*root*
    *navigate to trash 1*
        *go to location of trash 1 in room 1*
    *pick trash 1*
    *navigate to bin*
        *exit room 1*
        *enter room 3*
        *go to location of dump in room 3*
    *put trash 1 in dump*
*end*

**Learned Policy for Agent 2**

*root*
    *navigate to trash 2*
        *go to location of trash 2 in room 1*
    *pick trash 2*
    *navigate to bin*
        *exit room 1*
        *enter room 3*
        *go to location of dump in room 3*
    *put trash 2 in dump*
*end*

Figure 4.7: The top and bottom boxes show the learned policy for agent number 1 and 2 respectively, when the two agents are collectively performing trash collecting task, using the co-operative multi agent MAXQ algorithm

Figure 4.8 clearly shows the difference in the performance for the co-operative multi agent and single agent case. This plot shows the number of steps required to complete the task of depositing trash, measured in terms of the number of primitive actions

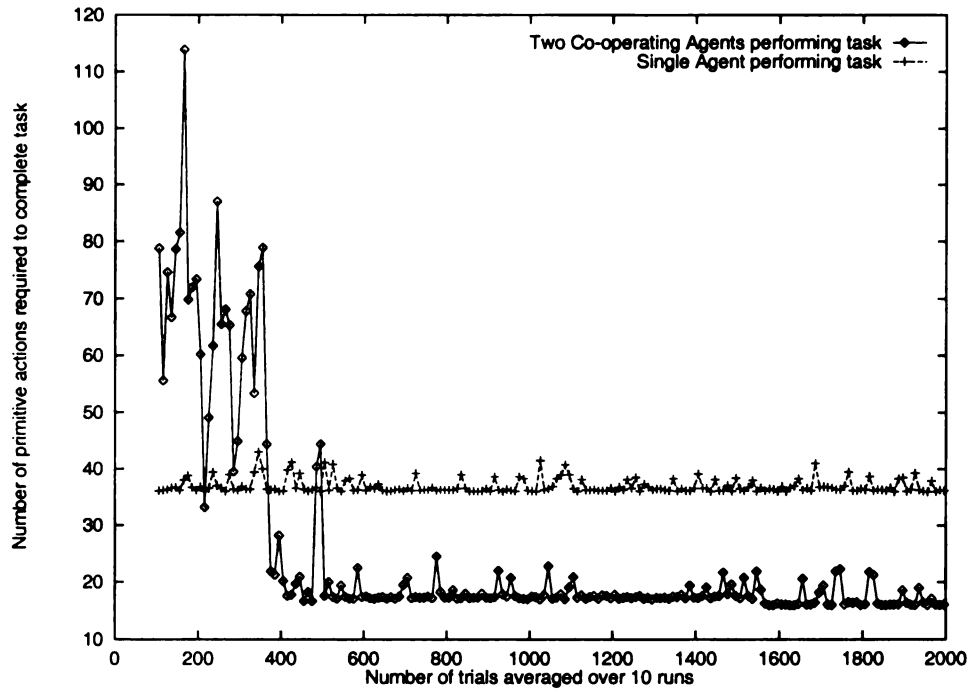



Figure 4.8: The number of primitive actions required to complete task.

required in both cases. Parallel execution of two primitive actions in the multi agent case almost halves the number of steps required.

## 4.3 Conclusion

Looking at the encouraging results presented above for the simple navigation problem, we decided to test the hierarchical multi agent learning algorithm on the complex AGV scheduling task. The following chapters discuss the AGV scheduling task details and the experimental results obtained with the co-operative multi agent algorithm.

# Chapter 5

# Experiments with AGV Scheduling

An Automated Guided Vehicle (AGV) system is a set of co-operating robotic vehicles used for fetching and delivering material and finished assemblies from a warehouse to the machines processing them and back. Automated Guided Vehicles (AGVs) are routinely used in industry flexible manufacturing systems (FMS) for material handing. Any FMS system using AGVs has to deal with the problem of optimally scheduling the paths of the AGVs in the system. AGV scheduling can be defined as the optimization process which selects and sequences activities for AGVs, given a set of constraints that reflect the temporal relationships among activities, and the capacity limitations of a set of shared resources [24]. Assemblies may be queued at any station, awaiting delivery to their destinations. In addition, more parts might arrive into the system even as deliveries are being made. The problem of sequencing an AGVs pickups and deliveries so that parts reach their destinations as quickly as possible is known as the AGV Scheduling Problem. The complex nature of this problem makes it very interesting, as the states keep changing not only due to the

agent's actions, but even otherwise. This is what prompted us to use this problem as a testbed for the multi agent reinforcement learning algorithm developed. The uncertain and ever changing nature of the job shop makes it virtually impossible to plan moves ahead of time. Hence, AGV scheduling requires dynamic dispatching rules.

## 5.1 Scheduling in Flexible Manufacturing Systems

Planning is the process of selecting and sequencing activities such that they achieve one or more goals and satisfy a set of domain constraints. Scheduling is the process of selecting among alternative plans and assigning resources and times to the set of activities in the plan. These assignments must obey a set of rules or constraints that reflect the temporal relationships between activities and the capacity limitations of a set of shared resources. The assignments also affect the optimality of a schedule with respect to criteria such as cost, tardiness, or throughput. In summary, scheduling is an optimization process where limited resources are allocated over time among both parallel and sequential activities [35].

A flexible manufacturing system consists of a set of machine tools and a material handling system linked by a network of computers controlling and interfacing them [27]. Unlike the traditional material handling system, where a human element is involved in the transportation of materials between various locations, most of the control is done with the help of computers in flexible manufacturing systems. This has been made possible by developments in guided vehicle technology and computer

controlled systems and machines [26]. With elimination of human intervention, the AGV control system and the software governing it are complex and critical, especially in the case of multi vehicle systems. Issues in material handling systems include the number of vehicles required, the track layout, the traffic pattern and traffic control, but the focus of the thesis is on the traffic control problem. AGV scheduling requires dynamic dispatching rules, which are dependent on the state of the system like the number of parts in each buffer, the state of the AGV and the processing going on at the workstations. Since this problem is analytically intractable, various heuristics and their combinations are generally used to schedule AGVs [23] [25] [28]. Some heuristics usually used are:

FCFS First Come First Serve. The assignment which enters the queue first is the one which is serviced first.

SS Stay in Same Station. After dropping off a part at a dropoff Station of one machine, the robot goes to the pickup station of the same machine to pick up the finished assembly.

NS Nearest Station. After finishing an assignment, the AGV scans through a list and acquires the nearest assignment, and proceeds towards the same.

HQ Highest Queue First. The AGV scans through a list of assignments and goes to the pick up station of the machine with the highest number of in process jobs.

HOQ Highest Output Queue First. The AGV first services the station which has the highest number of jobs waiting to be serviced in its output queue.

LWKR Least Work Remaining First. The part which has least amount of work remaining to be done on it gets priority in the AGV selection process.

RAN Random Selection.

CYC Cyclic Selection.

Various combinations of these heuristics are also used for scheduling purposes. It has been shown that a combination of these heuristics (i.e. composite) heuristics work better than any one heuristic alone [23] [25]. Other studies have looked at the advantages and disadvantages of using multiple vehicle rules or tandem layouts with many single vehicle loops [36], [37]. The tandem layout is based on partitioning all stations into non-overlapping, single vehicle closed loops with additional pickup/dropoff points provided as an interface between adjacent loops. This kind of layout works well to avoid congestion and support distributed control. However, it has less routing flexibility in case of breakdowns, and has a more difficult load routing problem to solve.

## 5.2 Assumptions Used

- Local buffer capacity is limited and the number of AGVs are fixed.
- Input buffer capacity and output buffer capacity is same for all stations.
- AGV travel distance from station to station is known.
- The inter-arrival distribution of parts is normally distributed.

- Once an action (primitive or high level) is started, it has to run to completion.
- Processing times are uniformally distributed.
- The sequence of operations to be performed on the parts is predetermined.

Traditionally, various heuristics such as FCFS ( First Come First Served ), Nearest Station first, Stay in Same Station, or combinations of these heuristics have been used for AGV scheduling in such systems. The heuristics have the advantage that they are simple to implement, but in most systems, the performance has much higher precedence over simplicity.

## 5.3 The Problem Description

Figure 5.1 shows the layout of the system used for experimental purposes. Parts of type $i$ have to be carried to drop off station at workstation $i$ and the assembled parts brought back into the warehouse. The AGV travel is unidirectional (as the arrows show). Figure 5.2 shows the MAXQ graph for this problem.

## 5.4 Implementation Details

Synchronization and concurrency issues arise while implementing a multi agent learning algorithm. These are discussed in this section. For the AGV scheduling task, the parent process first spawns five processes, one of which models the part arrival, and the other four model the machines activity at at each workstation (picking up parts from the dropoff buffer, servicing them, and putting the finished assembly in the

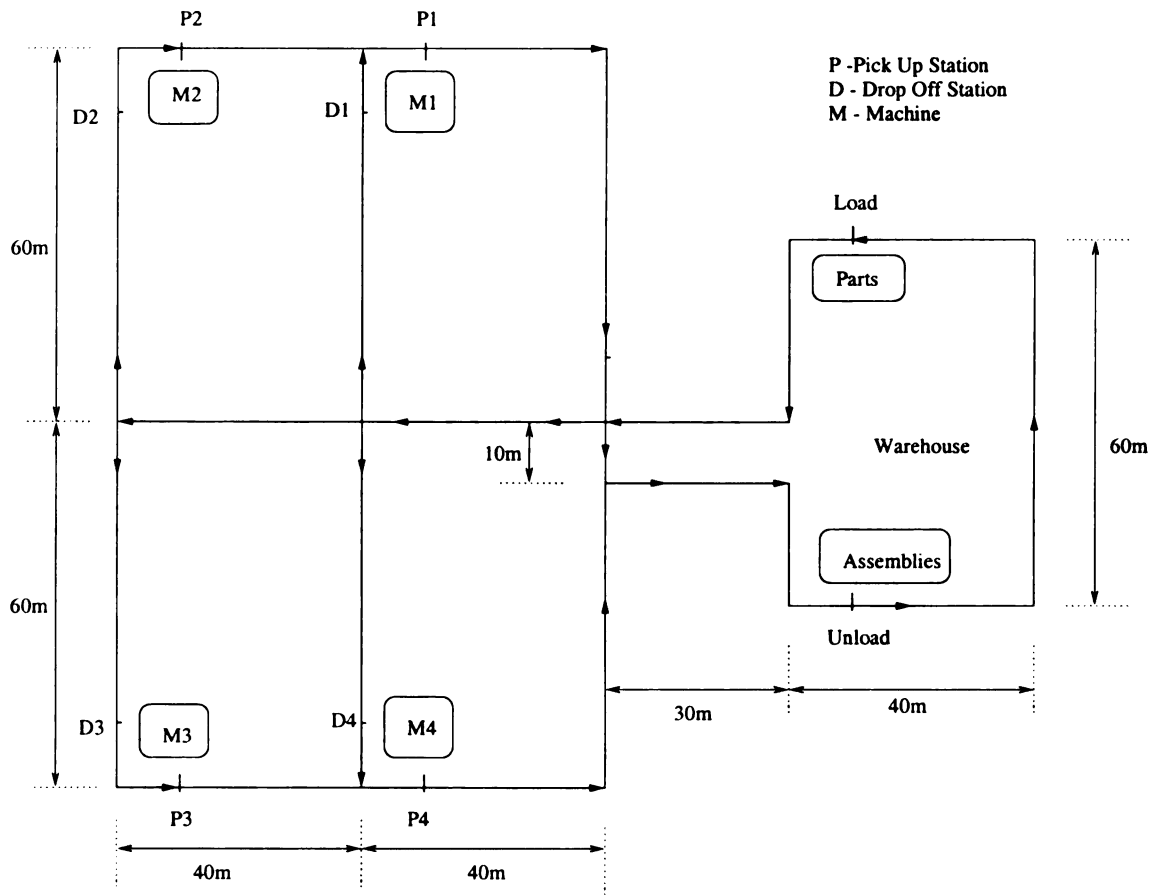



Figure 5.1: A multiple automatic guided vehicle (AGV) optimization task.

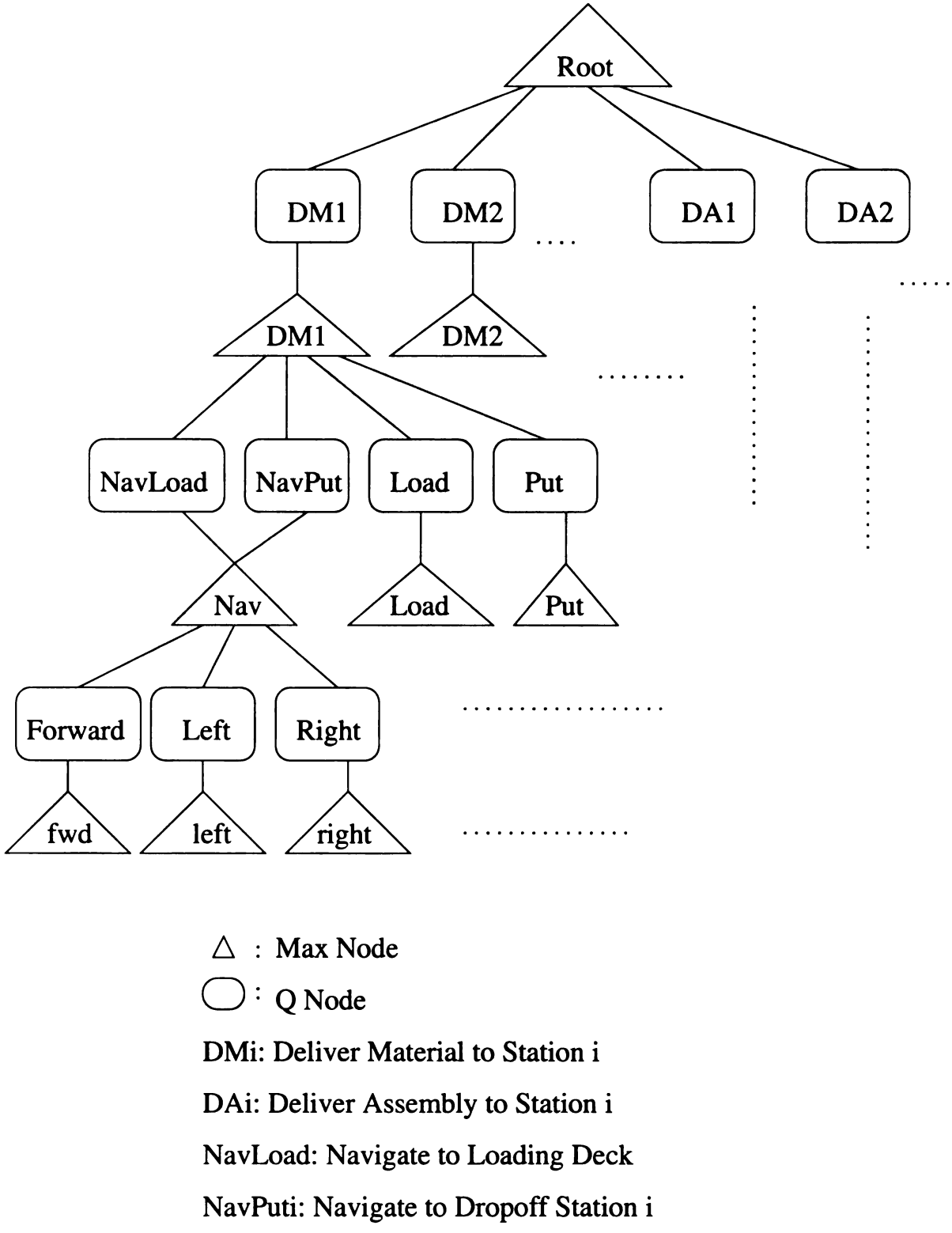



Figure 5.2: The MAXQ graph for an automatic guided vehicle (AGV) optimization task.

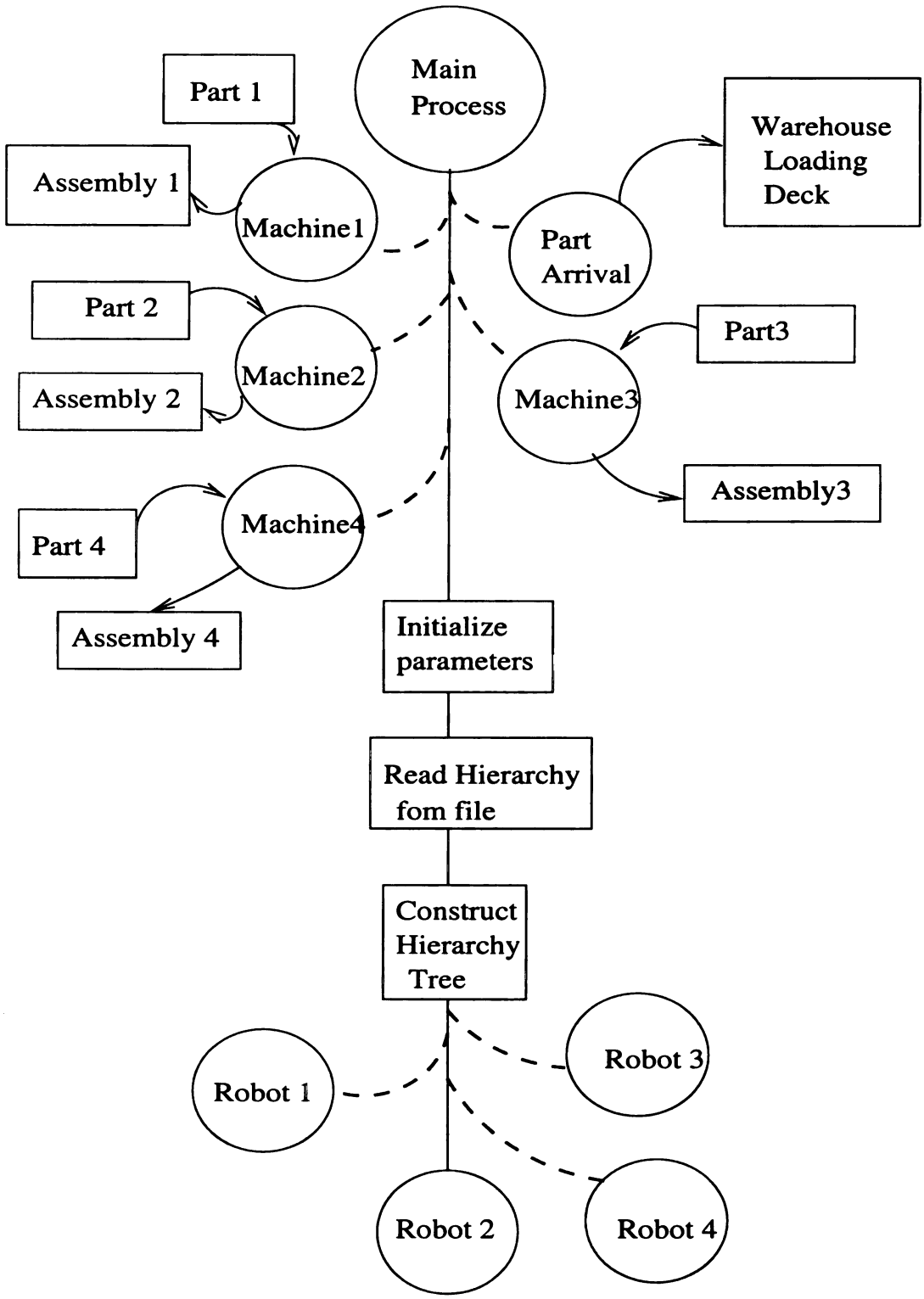



Figure 5.3: The program flow in the implementation of the AGV scheduling task.

pickup buffer). Three processes are then spawned for modeling the three agents, and the parent process models the fourth agent. This is shown diagrammatically in figure 5.3. The circles denote processes spawned by the main process, to perform various activities, as listed. The joint action value function is stored in a file in the hard disk, which is accessible to all four AGV processes. The state of the environment and the actions being currently performed by each agent are stored in shared memory, so that each agent can have access to it. Synchronization issues while accessing shared resources are taken care of with the help of semaphores.

We present the experimental results obtained by using the multi agent MAXQ framework developed here on the AGV scheduling task. Results were obtained for the flat approach, the single agent MAXQ approach, selfish multi agent MAXQ approach (where multiple agents with the same hierarchy try to learn in a common environment without sharing any information), and the new co-operative multi agent approach for this task.

## 5.5 State Abstraction

The state of the environment consists of the number of parts in the pickup station and in the dropoff station of each machine, and whether the warehouse contains parts of each of the four types. In addition, each agent keeps track of its own location and state as a part of the state space. Thus, in the flat case, the size of the state space is $\approx$ 100 locations, 3 parts in each buffer, 9 possible states of the AGV (carrying Part1, Empty ... ), and 2 values for each part in the warehouse, i.e. $100 \times 4^8 \times 9 \times 2^4 \approx 2^{30}$,

which is enormous, and would blow up even more to $(100 \times 9)^4 \times 4^8 \times 2^4$, or $2^{60}$ in the multi agent case with four agents. The MAXQ state abstraction helps in reducing the state space considerably, as for each action, i.e. each node in the Task graph, only the relevant state variables are used while storing the Completion functions. For example, considering the the Navigate subtasks, only the location state variable is relevant, and this subtask can be learned with 100 values. Hence, for the highest level with 8 actions, i.e. DM1 ... DM4, and DM2 ... DM4, the relevant state variables would be $100 \times 9 \times 4 \times 2 \approx 2^{13}$. For the lower level state space, the action with the largest state space is Navigate with 100 values. This state abstraction gives us a compact way of representing the C functions, and speeds up the algorithm.

## 5.6 Experimental Results

We present experimental results on the AGV scheduling task comparing several learning agents, including a single agent using MAXQ, selfish multiple agents using MAXQ (where each agent acts independently and learns its own optimal policy), and the new co-operative multi agent MAXQ approach. The experimental results were generated with the following model parameters. The inter arrival time for part arrival at the warehouse is uniformly distributed with a mean of 4.0 sec. The percentage of Part1, Part2, Part3 and Part4 in the part arrival process are 20, 28, 22 and 30 respectively. The time required for assembling the various parts is normally distributed with means 15, 24, 24 and 30 sec for Part1, Part2, Part3 and Part4 respectively. Each experiment was conducted five times and the results averaged.

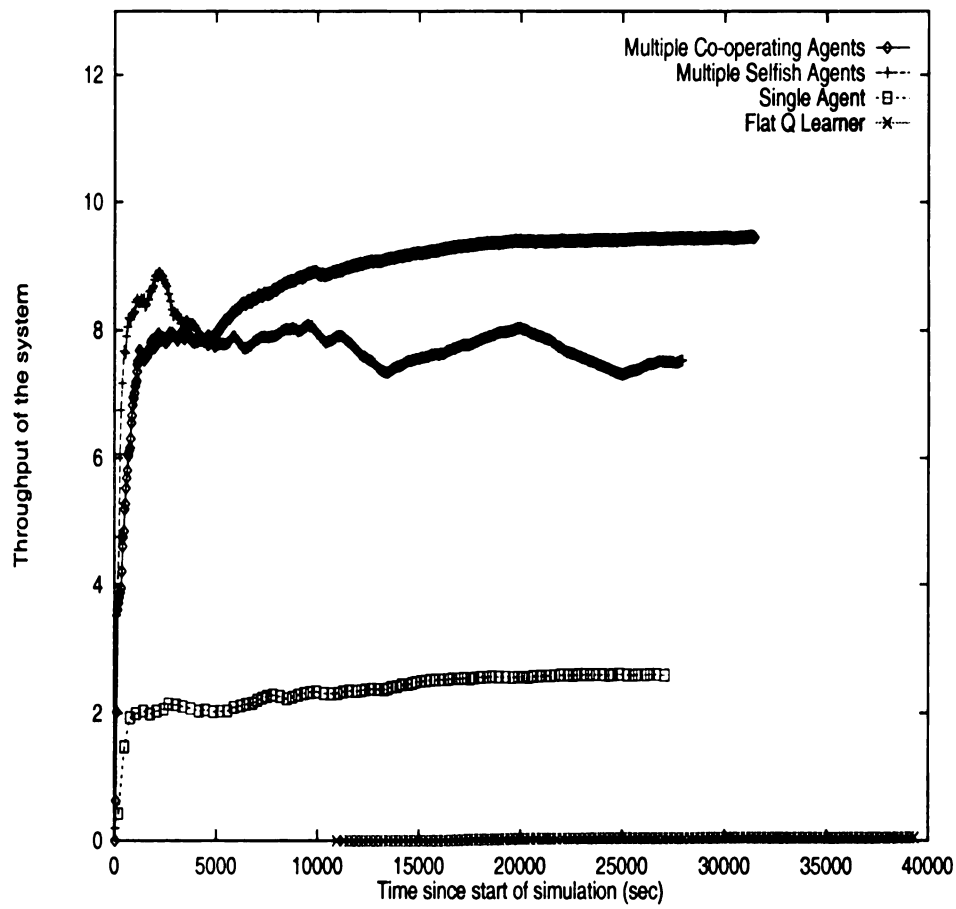



Figure 5.4: This figure compares the performance of single agent, multi-agent selfish MAXQ, multi agent co-operative MAXQ and flat methods for the AGV scheduling task. It shows the throughput of the system when the AGV travel time is very much less compared to the assembly time.

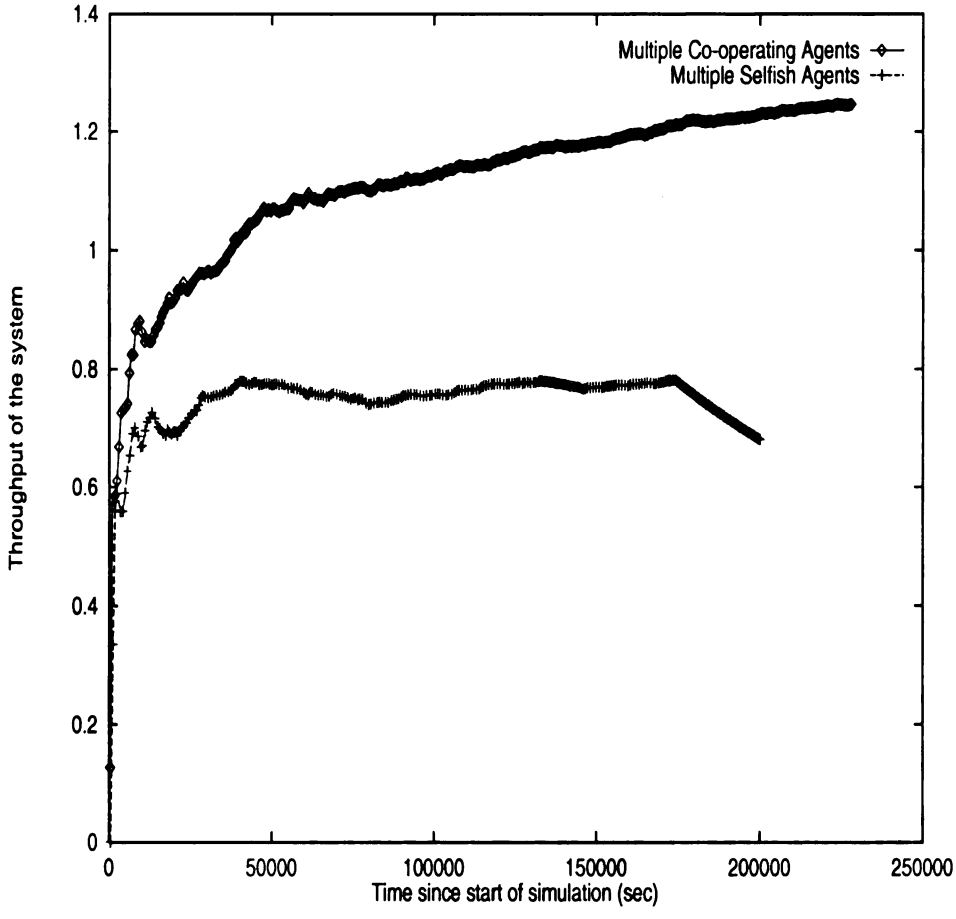



Figure 5.5: This figure compares the performance of single vs. multi-agent MAXQ methods on the AGV scheduling task. It shows the results when the AGV travel time and load/unload time is 1/10th that of the average assembly time.

Figures 5.4 and 5.5 show the throughput of the system for the three learning agents (measured in terms of number of assemblies delivered per min). Four homogeneous agents are considered in the multi agent cases. As seen in figure 5.4, the agents learn a little faster initially in the selfish multi agent method, but after some time, undulations are seen in the graph showing not only that the algorithm does not stabilize, but also that it results in sub-optimal performance. This is due to the fact that two or more agents select the same action, but once the first agent completes the task, the other agents might have to wait for a long time to complete the task, due to the constraints on the number of parts that can be stored at a particular place. The system throughput achieved using the new cooperative multi agent MAXQ method is significantly higher than the single agent or selfish multi agent case. This difference is even more significant in figure 5.5, as when the agents have a longer travel time, the cost of making a mistake is greater.

Figure 5.6 show results from an implementation of a single flat Q-Learning agent with the buffer capacity at each station set at 1. As can be seen from the plot, the flat algorithm converges extremely slowly. The throughput at 70,000 sec has gone up to only 0.07, compared with 2.6 for the hierarchical single agent case.

Figure 5.7 compares the cooperative multi-agent MAXQ algorithm with a well-known AGV dispatching rule, showing clearly the improved performance of the reinforcement learning method.

Figure 5.8 compares the cooperative multi-agent MAXQ algorithm with two AGV dispatching rules, namely, a combination of "nearest station first" and "stay in same station" heuristic, and the "highest queue first" heuristic. The way "highest queue

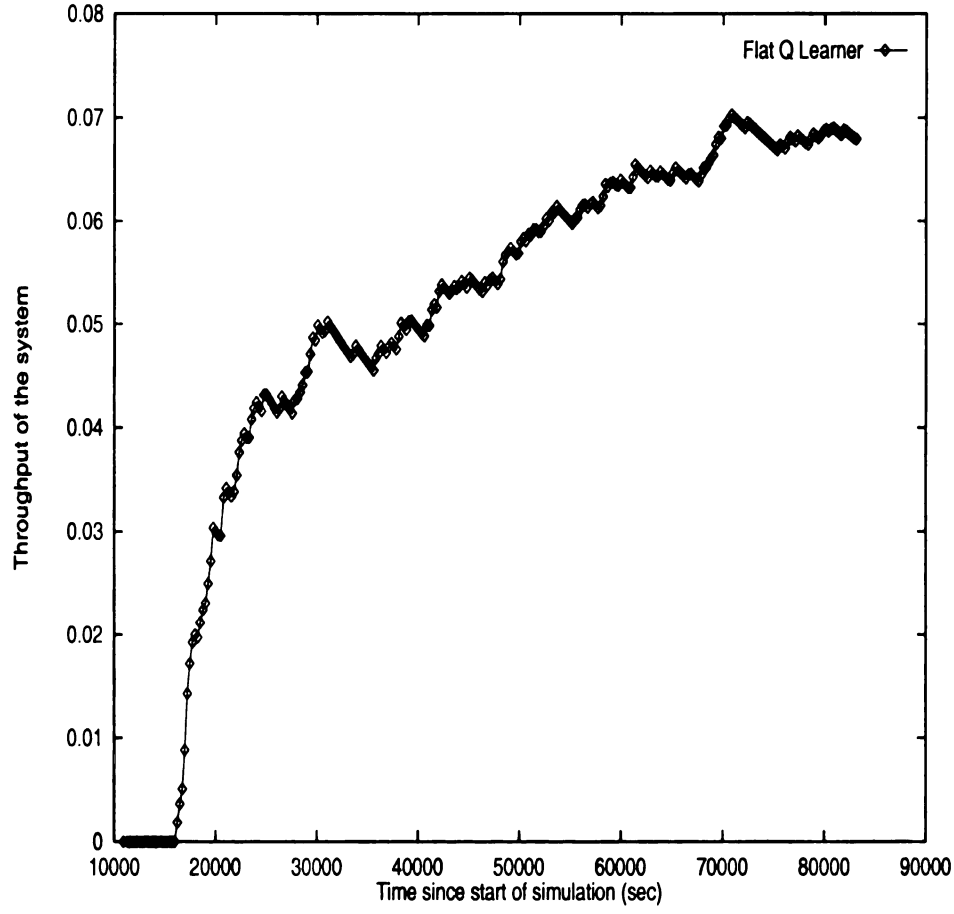



Figure 5.6: A flat implementation of a single agent reinforcement learner is shown here.

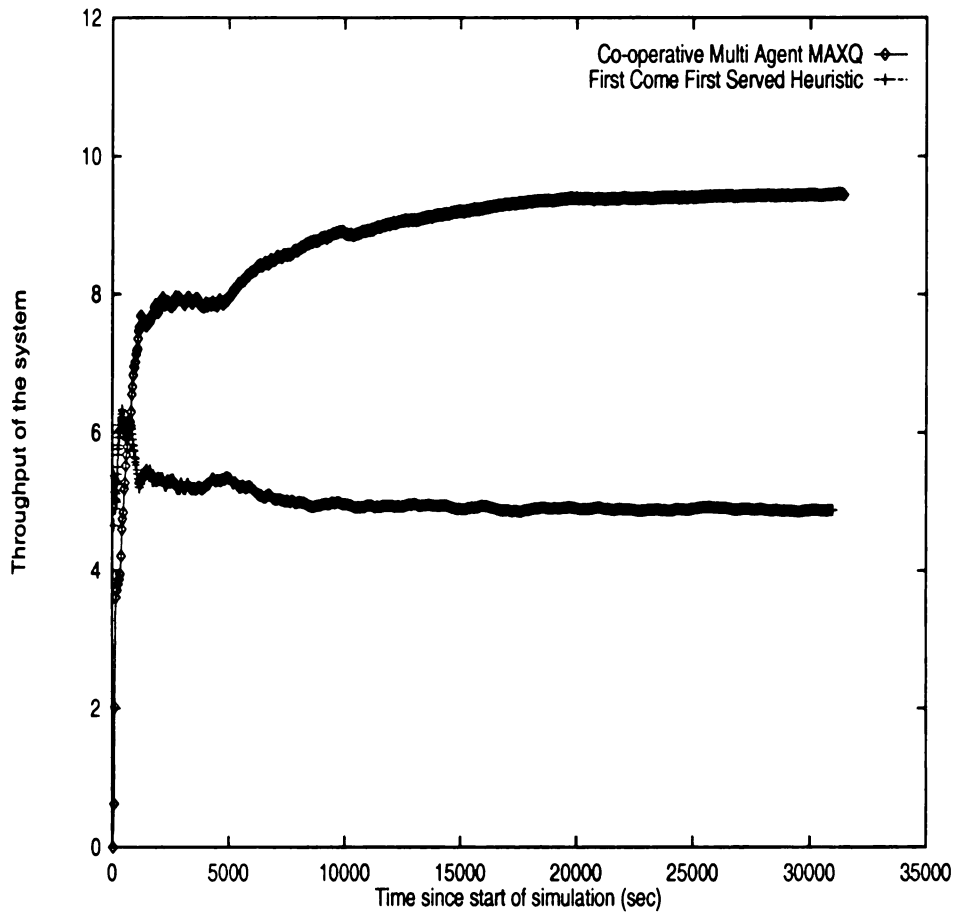



Figure 5.7: The performance of the proposed cooperative multi-agent MAXQ is compared with a well known "first come first serve" heuristic for AGV dispatching.

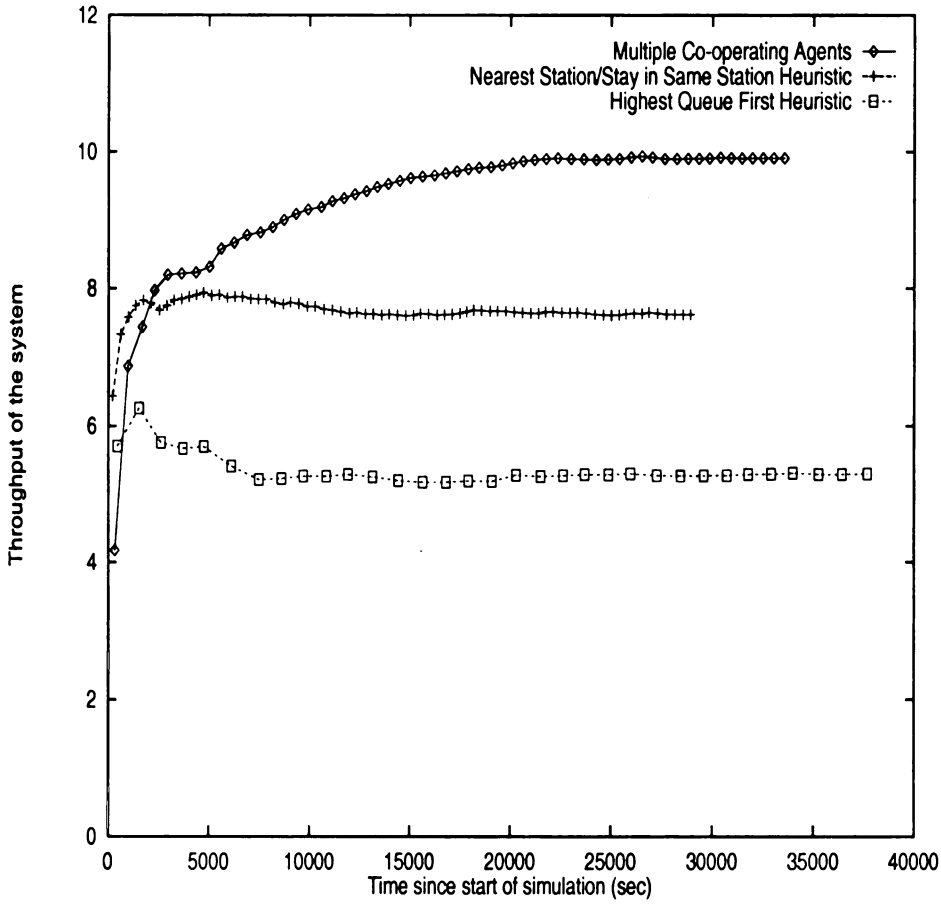



Figure 5.8: The performance of the cooperative multi-agent MAXQ algorithm is compared with a composite of nearest station and stay in same station heuristic, and the highest queue first heuristic for AGV dispatching.

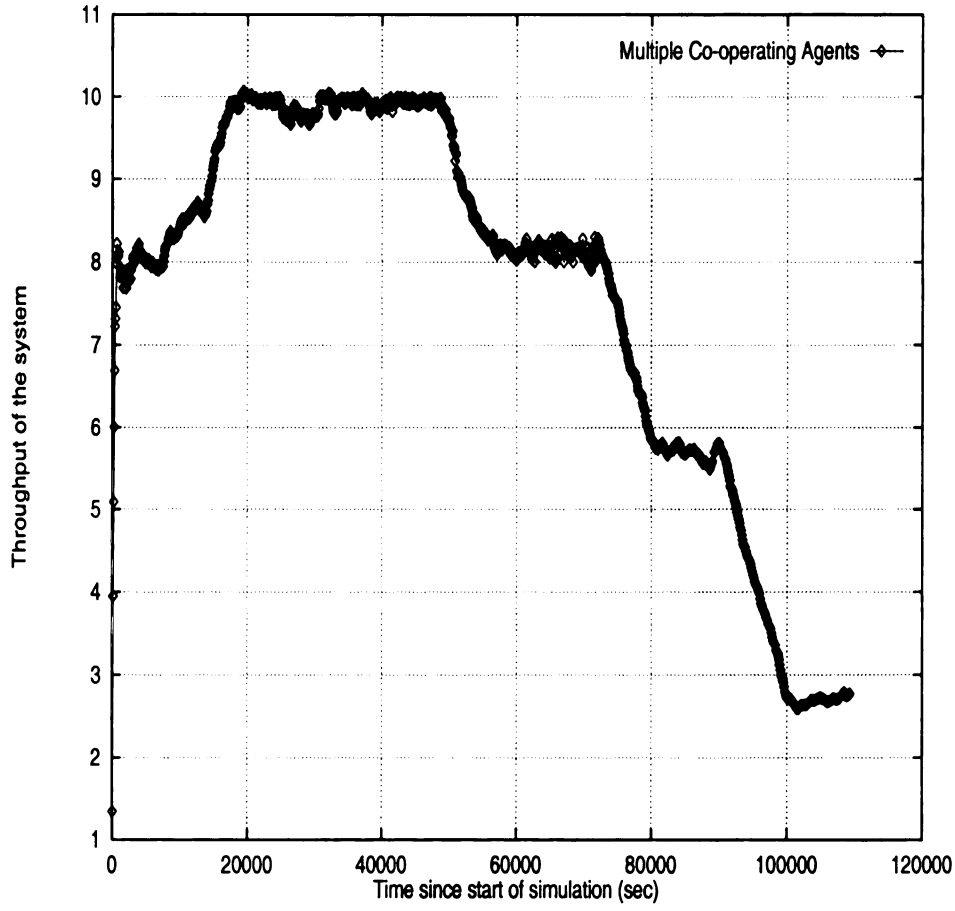



Figure 5.9: The performance of the cooperative multi-agent MAXQ algorithm when one agent breaks down at 50000 sec, another at 70000 sec and the third at 90000 sec.

first" heuristic works is that when a vehicle finishes its task, it scans through the list of pending jobs and chooses to serve the machine with the highest number of in process jobs. The AGV selects the assignments which are at the location nearest to it in the "nearest station first/stay in same station" heuristic. When there are no jobs in its job list the AGV goes to the pickup station of the same machine where it dropped the part, and waits for the job on that part to be finished. Better performance of the hierarchical reinforcement learning algorithm can be clearly seen from this figure. Figure 5.9 shows the throughput achieved by the the ooperative multi-agent MAXQ algorithm when one agent breaks down at 50000 sec, another at 70000 sec, and the third at 90000 sec. The final performance when there is only one agent is comparable to the single agent case. When an agent breaks down here, the other agents assume that is is continuously performing the idle action. These experiments show that the new co-operative multi agent MAXQ method performs better than some heuristics, the flat reinforcement learning method, and single agent and multi agent selfish MAXQ methods, and is thus a promising way to approach multi agent hierarchical reinforcement learning problems. This method can be extended in a number of different ways, which are discussed in the next chapter.

# Chapter 6

# Conclusions and Future Work

This thesis studied the problem of multi agent reinforcement learning, and proposed a novel approach for dealing with it. In particular, we extended the MAXQ approach for hierarchical reinforcement learning to the multi agent domain. This chapter summarizes the new multi-agent hierarchical reinforcement learning approach, and provides several interesting ideas for extending this work in the future.

## 6.1 Conclusion

We proposed a new algorithm which takes advantage of the abstraction provided by hierarchical reinforcement learning in order for multiple agents to co-operatively learn tasks which might require co-ordination among agents. Flat methods do not scale well to the multi agent tasks. Also, interaction among agents at the lower level sometimes make it very difficult for the co-operation skills to be learned. Using hierarchy in combination with multi agent learning speeds up learning enormously.

Also, a good policy is arrived at as co-operation is only learned at the higher level and does not have any effect on the low level policy. The only information an agent has about the other agents is the high level action being performed by them, which is easy as it requires very little communication, and is also an approximate indicator of what states the agents might possibly be in. This approach for scaling multi agent reinforcement learning uses the MAXQ hierarchical learning framework. The basic MAXQ learning algorithm is adapted to the multi agent case by having joint action values at the top level of the hierarchy. Events are used to signal completion of a subtask instead of state information, as that is more feasible in certain kinds of tasks which require discrete event modeling. The proposed method assumes a distributed architecture in which each agent takes it's own decision about the action it should perform, depending on the state of the environment and the actions being performed by other agents, rather than have a centralized controller dictate it's actions. Detailed experimental results from a complex AGV scheduling task were presented which show that the hierarchical multi agent MAXQ approach performed better than either the single agent or simple multi agent MAXQ methods. This novel approach of utilizing hierarchy for learning co-operation skills shows considerable promise as an approach that can be applied to other complex multi-agent domains. We primarily explored the use of the MAXQ hierarchical framework in our study, but we believe that other hierarchical methods could also be used to speed up multi-agent learning. The success of this approach depends on providing the system with a good initial hierarchy.

## 6.2 Future Work

This novel approach of utilizing hierarchy for learning co-operation skills suggests many ideas for interesting possibilities for future research, as a lot still needs to be explored in hierarchical and in multi agent learning. We present a few ideas in that for future work in the next section.

### 6.2.1 Adaptive Hierarchies

Since a large part of the success of this algorithm depends on a good initial hierarchy, changing the hierarchy would dramatically change the results and the learning speed. Thus, an area worth exploring here would be how one can arrive at a good hierarchy for decomposing the overall task into subtasks. Potential for state abstraction is an important factor to be considered here as some hierarchies might permit more savings in terms of state abstraction. Another factor for the multi agent case is that the top level decomposition must decompose the task into essentially parallel subtasks, as these are the subtasks where the co-operation skills are learned. For example, in the AGV scheduling task, subtasks *load material* and *drop material at dropoff station* are sequential as these two have to be performed in a certain order to achieve the goal of *delivering material to a station*, and it would make more sense to group them together under another subtask. This situation also arises due to the fact that our approach only considers co-operation only at the highest level of abstraction, hence the subtask decomposition has to be devised in a way that it is possible to learn co-operation skills only at the highest level of abstraction. It would be helpful to devise

a method by which agents could learn co-operation at the lower levels of abstraction too if desired, but the overhead of doing that might possibly overshadow the benefits. Also, not having co-operation at the lower level provides opportunities for re-using the lower level policy once it has been learned. Coming back to the problem of devising hierarchies, using heterogeneous agents with different initial hierarchies, might be a step in this direction. Agents could even try different variations in parallel by moving around the nodes in the hierarchy and finally choosing a solution which maximizes the overall system performance.

### 6.2.2 Other Hierarchical Learning Approaches

We primarily explored the use of the MAXQ hierarchical framework in our study, but we believe that other hierarchical methods could also be used to speed up multi-agent learning. Thus, it would be worth exploring the possibilities of multi agent learning in combination with the other hierarchical reinforcement learning frameworks and see how well they perform. For example, trying to extend the HAM (Hierarchies of Abstract Machines) or Options framework to the multi agent case might pose new challenges, and might help us in better understanding the hierarchical multi agent reinforcement learning paradigm.

### 6.2.3 Theoretical Foundations

We have not investigated the theory behind the new hierarchical reinforcement learning method. Q-Learning has been proven to converge to an optimal policy, whereas

MAXQ learning has been proven to converge to a recursively optimal policy under certain conditions. These theoretical foundations need to be examined in the multi agent case as well, as that will provide us with a better understanding of the multi agent reinforcement learning problem, which in turn might lead to better solutions. It would also help to have a more formal and unified definition of multi agent and hierarchical multi agent frameworks.

### 6.2.4 Other Extensions

Collective reward sharing might speed up learning even more as all the agents would get reinforcement more often. This would mean that when one agent receives a reward, the other agents would also be notified of it and they would accumulate this reward too between decision epochs. Real life systems are very stochastic in nature. Conditions (eg. requirements for number of assemblies etc.) are constantly changing. Hence, if the algorithm is modified to have rewards which change dynamically, it would be more useful in real world situations. This approach also needs to be tested on more complex problems.

# Bibliography

# Bibliography


[1] Mitchell, Tom (1997) Machine Learning. *McGraw Hill.*

[2] Richard Sutton and Andrew Barto (1998) Reinforcement Learning An Introduction. *The MIT Press.*

[3] Leslie Pack Kaelbling (1993) Hierarchical learning in stochastic domains: Preliminary results. *Proceedings of the Tenth International Conference on Machine Learning*

[4] Dietterich, T. G. (1998) The MAXQ method for hierarchical reinforcement learning. *International Conference on Machine Learning.*

[5] D. Bertsekas (1995) Dynamic Programming and optimal control. *Athena Scientific.*

[6] D. White (1963) Dynamic Programming, Markov Chains, and the method of successive approximation. *Journal of Mathematical Analysis and Applications* vol. 6, pp. 373-376.

[7] C. Watkins (1989) Learning from Delayed Rewards. *PhD Thesis*, Kings College, Cambridge.

[8] Dietterich, T. G. (2000). State abstraction in MAXQ hierarchical reinforcement learning. *Advances in Neural Information Processing Systems*, 12. S. A. Solla, T. K. Leen, and K. R. Muller (eds.), pp. 994-1000, MIT Press.

[9] Dietterich, T. G. (2000). Intra-option learning about temporally abstract actions. *Proceedings of the 15th International Conference on Machine Learning*, pp. 556-564.

[10] Sridhar Mahadevan and Jonathan Connell (1992). Automatic Programming of Behavior-based Robots using Reinforcement Learning. *Artificial Intelligence*, vol. 55, Nos. 2-3, pp. 311-365.

[11] Maja Mataric. (1997) Reinforcement Learning in the Multi-Robot Domain. *Autonomous Robots*, 4(1), 73-83.

[12] Michael Littman. (1994) Markov games as a framework for multi-agent reinforcement learning. *Proceedings of the Eleventh International Conference on Machine Learning*, pp. 157-163.

[13] Ming Tan. Multi-agent reinforcement learning: independent vs. cooperative agents. *Proceedings of the Tenth International Conference on Machine Learning.* pp. 330-337, Amherst, MA.

[14] Crites, R.H. and Barto, A.G. (1998) Elevator group control using multiple reinforcement learning agents. *Machine Learning* 33 pp. 235-262.

[15] Tesauro, G. (1992) Practical issues in temporal difference learning. *Machine Learning* Vol. 8, no. 3.

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

[17] Singh S, Bertsekas D (1997) Reinforcement Learning for Dynamic Channel Allocation in Cellular Telephone Systems. *NIPS 10 proceedings.*

[18] Hu, J. and Wellman, M. (1998) Multiagent reinforcement learning: Theoretical framework and an algorithm. *Fifteenth International Conference on Machine Learning,* pp. 242-250.

[19] Sutton, R. S., Precup, D., Singh, S. (1999) Between MDPs and semi-MDPs: A Framework for Temporal Abstraction in Reinforcement Learning. *Artificial Intelligence* 112 pp. 181-211.

[20] Stone, P. and Veloso, M. (1998) Team-Partitioned, Opaque-Transition Reinforcement Learning. *Agents* pp. 86-91.

[21] Parr, R. E. (1998) Hierarchical Control and Learning for Markov Decision Processes. *PhD Thesis*, University of California, Berkeley.

[22] Tucker Balch and Arkin, Ronald, C. (1998) Behavior-based Formation Control for Multi-robot Teams. *IEEE Transactions on Robotics and Automation*, Vol. 14, Number 6, pp. 1-15.

[23] Lee J. (1996) Composite dispatching rules for multiple-vehicle AGV systems. *SIMULATION* 66(2) pp. 121-130.

[24] Askin, R. G. and Standridge, C. R. (1993) Modeling and Analysis of Manufacturing Systems. *John Wiley and Sons.*

[25] Klein, C. M. and Kim, J. (1996) AGV dispatching. *International Journal of Production Research* 34: (1) pp. 95-110.

[26] Schwind, G. F. (1988) AGVs: creative solutions go looking for problems. *Material Handling Engineering* 43 pp. 44-47.

[27] Mahadevan, B. and Narendran, T. T. (1990) Design of an automated guided vehicle-based material handling system for a flexible manufacturing system. *International Journal of Production Research* 28: (9) pp. 1611-1622.

[28] Sabuncuoglu, I. (1997) A study of scheduling rules of flexible manufacturing systems: a simulation approach. *International Journal of Production Research* 36: (2) pp. 527-546.

[29] Tadepalli, P. and Ok, D. (1996) Scaling up average reward reinforcement learning by approximating the domain models and the value function. *Proceedings of International Machine Learning Conference.*

[30] Wang, Gang and Sridhar Mahadevan (1999) Hierarchical Optimization of Policy-Coupled Semi-Markov Decision Processes. *International Conference on Machine Learning.*

[31] Milos Hauskrecht, Nicolas Meuleau, Craig Boutilier, Leslie Pack Kaelbling, and Thomas Dean. (1998) Hierarchical Solution of Markov Decision Processes Using Macro-Actions. *Proceedings of the Fourteenth International Conference on Uncertainty In Artificial Intelligence.*

[32] Peter Dayan and Geoff Hinton. (1993) Feudal Reinforcement Learning. *Advances in Neural Information Processing Systems., volume 5. Cambridge, MA.*

[33] Ron Parr and Stuart Russel. "Reinforcement Learning with Hierarchies of abstract Machines." Advances in Neural Information Processing Systems. volume 10, Cambridge, MA. 1993.

[34] Doina Precup, Richard Sutton, and Satinder Singh. (1997) Theoretical Results on Reinforcement Learning with Temporally Abstract Behaviors.

[35] M. Zweben and M. S. Fox (1994) *(Eds.): Intelligent Scheduling. Morgan Kaufmann Publishers, Inc.*

[36] Hoo Gon Choi and Hyuk Jin Kwon and Jim Lee (1994) Traditional and Tandem AGV System Layouts: A Simulation Study. *SIMULATION*, pp. 85-93.

[37] Yavuz A. Bozer and Mandyam M. Srinivasan (1991) Tandem Configurations for Automated Guided Vehicles and the Analysis of Single Vehicle Loops. *IIE Transactions*, Vol. 23, Number 1, pp. 72-82.

[38] John W. Sheppard (1998) Co-Learning in Differential Games. *Machine Learning, special issue on Multi-Agent Learning, Vol. 33, No. 2/3*, pp. 201-233.

[39] Sugawara, T. and Lesser, V. (1993) Learning Coordination Plans in Distributed Problem-Solving Environments. *Computer Science Technical Report 93-27* , University of Massachusetts, Amherst.

[40] Kaelbling, L. Littman, Moore, A. 1996. "Reinforcement learning: A survey" in *Journal of Artificial Intelligence Research* 4.

[41] Law, A. Kelton, W. 1991. *Simulation Modeling and Analysis.* New York, USA: McGraw-Hill.

[42] Marchalleck, N. J. 1996. *Improving Simulation Technology Through Reinforcement Learning.* Master's Thesis, University of South Florida.

[43] Thomas G. Dietterich Hierarchical Reinforcement Learning Tutorial. (1999) *given at The Sixteenth International Conference on Machine Learning* Bled, Slovenia.

[44] Mahadevan, S. Kaelbling, L.P. 1996. The NSF workshop on reinforcement learning: Summary and observation. *AI Magazine* 1(12):1-37.

[45] Puterman, M. L. (1994) Markov Decision Processes: Discrete Stochastic Dynamic Programming. *Wiley Series in Probability and Mathematical Statistics* John Wiley and Sons, Inc.

[46] Stuart Russel and Peter Norvig (1995) Artificial Intelligence A Modern Approach. *Prentice Hall.*

[47] M. Puterman (1994) Markov Decision Processes: Discrete Dynamic Stachastic Programming. *John Wiley.*