NADA
Numerisk analys och datalogi
Kungl Tekniska Högskolan
100 44 STOCKHOLM

Department of Numerical Analysis
and Computer Science
Royal Institute of Technology
SE-100 44 Stockholm, SWEDEN

An Evolutionary Approach to Automatic Construction of the Structure in Hierarchical Reinforcement Learning

Stefan Elfwing

TRITA-NA-Eyynn

Master's Thesis in Computer Science (20 credits)
at the School of Computer Science and Engineering,
Royal Institute of Technology, Februari 2003
Supervisor at Nada was Anders Lansner
Examiner was Anders Lansner

Abstract

Because the learning time is exponential in the size of the state space, a hierarchical learning structure is often introduced into reinforcement learning (RL) to handle large scale problems. However, a limitation to the use of hierarchical RL algorithms is that the learning structure, representing the strategy for solving a task, has to be given in advance by the designer. This thesis presents an evolutionary approach to automatic construction of the learning structure in hierarchical reinforcement learning. The proposed method combines the MAXQ hierarchical reinforcement learning method and genetic programming. The MAXQ method learns the policy based on the task hierarchies obtained by genetic programming, while genetic programming explores the appropriate hierarchies using the result of the MAXQ method. To show the validity of the proposed method, computer simulations for a foraging task are performed. In the task, the agent collects food, represented by battery packs, into its nest. The simulation results show a strong connection between the complexity of the evolved hierarchies and the complexity of the environment.

Ett evolutionärt tillvägagångssätt för automatisk konstruktion av inlärningsstrukturen i hierarkisk reinforcement learning

Sammanfattning

En hierarkisk inlärningsstruktur introduceras ofta i reinforcement learning (RL) för att hantera storskaliga problem. Orsaken är att inlärningstiden växer exponentiellt med avseende på storleken på tillståndsrymden. En begränsning av användningen av hierarkiska RL-algoritmer är att inlärningsstrukturen, som representerar strategin för att lösa uppgiften, måste ges av konstruktören i förväg. Den här rapporten presenterar ett evolutionärt tillvägagångssätt för automatisk konstruktion av inlärningsstrukturen i hierarkisk RL. Den föreslagna metoden kombinerar hierarkisk RL MAXQ och genetisk programmering. MAXQ-metoden lär en policy baserad på uppgiftshierarkierna som erhålls från genetisk programmering. Den genetiska programmeringen söker i sin tur efter lämpliga hierarkier, genom att använda resultaten från MAXQ-metoden. För att visa metodens giltighet utförs simuleringar för en uppgift där agenten ska söka efter mat. I uppgiften samlar agenten in mat, som representeras av batterier, till sitt bo. Simuleringsresultaten visar ett starkt samband mellan de evolverade hierarkiernas komplexitet och miljöns komplexitet.

Acknowledgments

I want to thank Dr. Kenji Doya, supervisor at ATR (Advanced Telecommunications Research Institute International), and Dr. Eiji Uchibe for valuable suggestions and ideas during the work. I also want to thank Anders Lansner, supervisor at KTH, for helpful advice and for support in the initial contacts with Japan. Finally, I want to thank Anders Eriksson for co-development of the MATLAB simulator used for my experiments and also for his friendship and support during the time in Japan.

I am very grateful for the economical support from the Sweden-Japan foundation and ATR, which made this thesis possible.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In everyday life humans and animals are experts in adapting to the current task and environment. For most tasks only a few basic behaviors are used, but these are combined and executed in an order which gives a strategy that is suited to the environment.

In this thesis a method for an artificial agent to adapt its strategy to the environment is presented. To represent the strategies of the artificial agent the MAXQ hierarchical reinforcement learning (RL) framework is used. The strategies are represented as trees of subtasks, denoted task hierarchies. When the strategies are used, the subtasks are executed sequentially, according to a learned policy, to complete the overall task.

The largest focus of research in the field of hierarchical RL has been in the development of learning algorithms for given hierarchical structures. Very few studies have been done in adaptation or learning of the task hierarchy. A good task hierarchy is a basic condition for hierarchical RL algorithms to work efficiently. Therefore, it is important to development methods that can construct a good hierarchy for a given problem. Most hierarchical RL applications are applied to simple problems in a discrete grid world with fixed environmental settings. In such cases it is relatively easy to provide a good task hierarchy. For large scale problems, especially if the agent has to rely on local and continuous state information, this is not the case.

In this thesis an evolutionary approach is used for adapting the strategies, task hierarchies, to the environmental settings during the lifetime of the agent. An agent is seen as having a number of competing strategies inside its " brain". A good strategy is evolved over time by the mixing of the competing strategies. Genetic programming (GP) is chosen as evolutionary method, since the genetic operators of GP, responsible for the mixing of the competing strategies, are designed to be applied on tree structures.

1.1 Cyber Rodent Project

The work with this thesis has been performed within the Cyber Rodent project in Dr. Mitsuo Kawato's department at ATR (Advanced Telecommunications Research Institute International) in Japan, under the supervision of Dr. Kenji Doya. The goal of the Cyber Rodent Project is to understand the adaptive mechanisms necessary for artificial agents that have the same fundamental constraints as biological agents, namely, self-preservation and self-reproduction. Previous studies of RL assumed arbitrarily defined "rewards", but in the models of biological systems, rewards should be grounded as the mechanisms for self-preservation and self-reproduction. To be able to achieve these goals the Cyber Rodent robots have the capability of finding and re-charging from battery packs and copying programs via IR-ports.

The main research topics of Cyber Rodent project are

- **Development of programs for basic behaviors.** Examples of basic behaviors of the Cyber Rodent are foraging, detection of friends and enemies, and communication via IR-ports.
- **Implementation of RL algorithms.** The learning algorithms shall improve the basic behaviors, create new behaviors or combine already learned behaviors for more complex behaviors, for example collaborative behaviors.
- **Development of meta-learning algorithms.** To achieve robust and efficient learning the parameters of the learning algorithms, the meta-parameters, also have to be learned or tuned.
- **Exploration of communication mechanisms.** The Cyber Rodent robots can communicate in various types of ways. The goal of this research is to investigate how the communication can be integrated with the learning to improve the performance.
- **Design of an evolutionary framework.** To be able to achieve the ultimate goal, to create evolution in the laboratory, an artificial gene code has to be designed. The code has to have the ability to describe behaviors and learning algorithms, and also to maintain functionality under genetic manipulation.

1.2 Motivation and Goal

The goal of this study is to find a method for automatic construction of the task hierarchy in hierarchical RL. The main motivation is that the existing hierarchical RL algorithms have no capability to learn or adapt the task hierarchy [3, 14]. The designer has to provide the task hierarchy in advance and as already mentioned this is non-trivial for many problems.

A second motivation for this thesis is that an automatic construction method could provide meta-learning capability to the learning structure. Meta-learning is the study of learning about learning. In this case meta-learning capability means that the automatic construction method could facilitate learning of related tasks and environments. If the agent has learned a good hierarchy for one task and environment and then is moved to a related task and/or environment, an automatic construction method could make small changes to the learning structure, to adapt the agent's strategy to the new situation.

The task hierarchies are seen as strategies for a single agent to solve a problem during its lifetime. The task hierarchies are built up from a few basic behaviors that the agent already knows how to perform. As the adaptation of the task hierarchy is accomplished during the agent's lifetime, the population in the evolutionary computation represents different types of competing hierarchies within one agent and the learning in the hierarchies continue through the evolutionary process. Also, as the agent uses a number of already learned basic behaviors these basic behaviors are shared by all hierarchies. An aim in this thesis is that the artificial agent acts as a realistic autonomous agent, using local and continuous information about the environment.

Another important topic for this thesis is to study the evolution process. Artificial evolution methods have the advantage compared with real biological evolution that the details of the process can be studied in detail. In real biological evolution only the final result can be studied.

1.3 Related Works

Very few studies have been done in the field of adapting or learning of the task hierarchy in hierarchical RL. The author has not find any previous works that are directly related to the work presented in this thesis. The author found the three studies presented below the most interesting works in the scientific field of this thesis.

Doya *e*t al. [5] proposed multiple model-based reinforcement learning for RL in continuous time and space. Each module consists of a state prediction model and a reinforcement learning controller, and their system could decompose a complicated task into simpler subtasks based on the idea of a softmax selection of the prediction errors. However, it is difficult to extend their method to a multi-layer architecture since only one softmax function is allowed for the upper layer.

Downing [4] applied GP to construct the state space for successful learning to converge. Uchibe *e*t al. [18] also applied GP to obtain hierarchical structures for cooperative and competitive mobile robots. In their case, a terminal set consisted of multiple behaviors was obtained by RL. Since their methods just obtained switching conditions according to the situation, the value function obtained by RL was not utilized effectively.

In the related field of on-line learning and evolutionary robotics there are several interesting studies. Nordin *e*t al. [13] applied GP directly to the binary code to learn an obstacle avoiding behavior, using the robot's infrared distance sensors. The experiments were performed on the Khepera robot and the learning time, 1.5 s, is exceptionally fast. Martin [10] used GP to learn a program for visual obstacle avoiding behavior.

Hornby *et* al. [7] proposed an autonomous evolutionary algorithm for developing dynamic locomotion gaits for the Sony quadruped robot, using the robot's digital camera and infrared sensors.

Chapter 2

Background

2.1 Reinforcement Learning

RL is a computational approach to learning from experience, by interaction with the environment. Compared with other machine learning approaches, RL is much more goal-directed. RL is not a *supervised learning* technique, like for example supervised artificial neural networks and decision trees. In supervised learning the designer provides the agent with a number of training examples. The success of the learning is then based on how well the agent generalizes from these training examples.

2.1.1 Concept

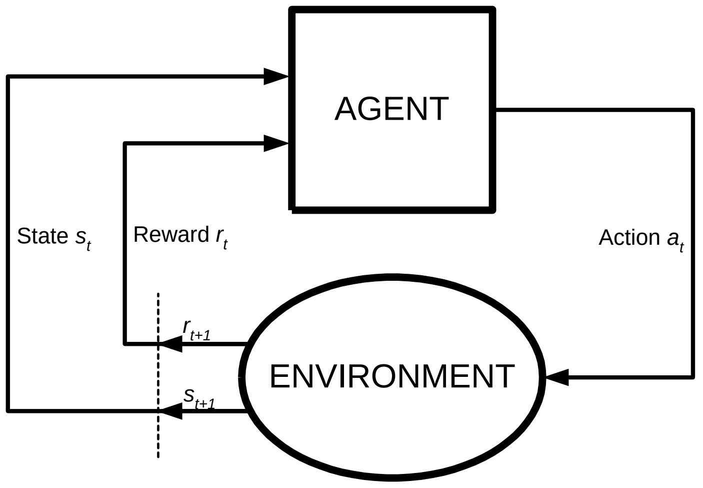


Figure 2.1. The RL model

Figure 2.1 shows the RL framework. An *agent* interacts with the *environment* by, in each *state*, selecting an *action*, which puts the agent in a new state and the agent also receives a numerical *reward* signal from the environment.

More formally, in each discrete time step, $t = 0, 1, 2, \ldots$, the agent has a representation of the state of the environment, $s_t \in \mathbb{S}$, where $\mathbb{S}$ is the set of possible states. On the basis of the current state information the agent selects an action, $a_t \in \mathbb{A}(s_t)$, where $\mathbb{A}(s_t)$ is the set of possible actions in state s_t. Thereby, the agent receives a numeric reward, $r_{t+1} \in \mathbb{R}$, and is in the new state s_{t+1}. RL methods learn by experience a *policy* π_t, a mapping from state representations to action selection probabilities. $\pi_t(s, a)$ is the probability for selecting action $a_t = a$ if $s_t = s$. The general goal of reinforcement learning is to find a policy that maximizes the expected future cumulative reward.

The expected future cumulative reward, which is the objective of learning, is called the *expected return*, R_t. In simple cases where the learning of the task can be broken down into subsequences, episodes, and there is a final time step for each episode, T, corresponding to the agent being in a *terminal state*, the expected return is defined as

$$R_t = r_{t+1} + r_{t+2} + \cdots + r_T = \sum_{k=0}^{T} r_{t+k+1}, \tag{2.1}$$

where $r_{t+1} + r_{t+2} + \cdots + r_T$ is the sequence of rewards received after time step t. This type of tasks is called *episodic tasks* and a typical example of such a task is game playing, where the task terminates when one play of the game is finished.

Many tasks do not fulfill the requirements of being episodic. They can not naturally be broken down into episodes and do not have terminal states that terminate the task. This type of tasks is called *continual tasks*. For continual tasks $T = \infty$ and therefore, if R_t would be defined as in **Equation 2.1** the expected return would be infinite. To avoid this, the concept of *discounting* is introduced. The goal of the agent is then to maximize the expected *discounted return*, where the future cumulative reward is discounted exponentially as

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

where γ is the *discount rate* parameter, $0 \leq \gamma \leq 1$. If γ is zero the agent only tries to maximize the immediate reward and as γ approaches one the agent becomes more far-sighted, meaning that future rewards are more strongly weighted in the calculation of the discounted return. **Equations 2.1** and **2.2** can be unified to one equation as

$$R_t = \sum_{k=0}^{T} \gamma^k r_{t+k+1}, \tag{2.3}$$

where $T = \infty$ for continual tasks and $\gamma = 1$ for episodic tasks.

The reward function is the designer's tool for telling the agent what to achieve. Therefore, it is very important to design the reward function carefully, so that the reward signals in the different states really represent the goal the designer wants the agent to fulfill. The problem of assigning the appropriate reward to the different states is known as the *credit assignment problem.* A typical example of a reward function, for the episodic task chess playing, is that the agent receives -1 for loosing, $+1$ for winning and 0 for drawing and for all non-terminal states.

2.1.2 Markov Decision Process

The mathematical foundation for RL is developed under the assumption that the environment has the *Markov property* and thereby fulfills the requirement for being a *Markov decision process (MDP).*

RL problems have the Markov property if the state signal, received by the agent from the environment, contains all relevant information about the environment in all situations. This means that the agent has a perfect observation of the environment at all times. An example of a problem fulfilling the Markov property is chess playing, where the position of the board gives the agent perfect state information. If a problem has the Markov property the new state is only depending on the current state and the action selected by the agent.

Formally, in the general case the response of the environment at time t is depending on all earlier events: $s_t, a_t, r_t, \ldots, r_1, s_0, a_0$, expressed as the probability distribution in **Equation 2.4**. If a problem has the Markov property the response is only depending on the current state, s_t, and the action a_t, as seen in **Equation 2.5**. The Markov property is satisfied if and only if **Equation 2.4** is equal to **Equation 2.5.**

$$\mathbf{P}\left\{s_{t+1} = s', r_{t+1} = r | s_t, a_t, r_t, \ldots, r_1, s_0, a_0\right\} \tag{2.4}$$

$$\mathbf{P}\left\{s_{t+1} = s', r_{t+1} = r | s_t, a_t\right\} \tag{2.5}$$

for all s', r, and all possible earlier events: $s_t, a_t, r_t, \ldots, r_1, s_0, a_0$.

If the Markov property is satisfied, the RL task is a MDP. More specifically if the action and state spaces are finite sets the RL task is a *finite MDP.* A finite MDP is completely defined by the action space, state space and the 1-step dynamics of the environment defined as

$$P(s'|s,a) = \mathbf{P}\left\{s_{t+1} = s' | s_t = s, a_t = a\right\} \tag{2.6}$$

$$R(s'|s,a) = \mathbf{E}\left\{r_{t+1} | s_t = s, a_t = a, s_{t+1} = s'\right\} \tag{2.7}$$

$P(s'|s,a)$ is the probability distribution for the next state being s', given state s and action a. $R(s'|s,a)$ is the expected value of the next reward, given state s, action a and the next state s'.

All RL tasks do not satisfy the Markov property and many RL algorithms do not assume a perfect model of the environment. Although, it is appropriate to think of the state signal from the environment as an approximate of the Markov property, where the policy and the value functions are only functions of the current state and the selected action.

2.1.3 Value Functions

To realize the maximization of discounted return, almost all RL algorithms estimate the *value functions.* The value functions are the expected return for the agent to be in a state or to perform a given action in a state, according to the current policy.

The *state-value function for a policy* π, $V^{\pi}(s)$, is the value of starting in state s and thereafter following policy π, defined for a MDP as

$$\begin{aligned} V^{\pi}(s) &= \mathbf{E}_{\pi}\left\{R_t | s_t = s\right\} \\ &= \mathbf{E}_{\pi}\left\{\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} | s_t = s\right\} \\ &= \mathbf{E}_{\pi}\left\{r_{t+1} + \gamma \sum_{k=0}^{\infty} \gamma^k r_{t+k+2} | s_t = s\right\} \\ &= \sum_{a} \pi(s,a) \sum_{s'} P(s'|s,a)\left[R(s'|s,a) + \gamma V^{\pi}(s')\right], \end{aligned} \tag{2.8}$$

where $\mathbf{E}_{\pi}$ is the expected value given that the agent follows the policy π and s' is the next state. **Equation 2.8** is called the *bellman equation* for V^{π} and expresses that the value function can be defined recursively as a relationship between the value of current state and the value of the next state.

The *action-value function for a policy* π, $Q^{\pi}(s,a)$, is the expected return starting in state s, taking action a, and thereafter following policy π, defined for a MDP as

$$\begin{aligned} Q^{\pi}(s,a) &= \mathbf{E}_{\pi}\left\{R_t | s_t = s, a_t = a\right\} \\ &= \mathbf{E}_{\pi}\left\{\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} | s_t = s, a_t = a\right\} \\ &= \sum_{s'} P(s'|s,a)\left[R(s'|s,a) + \gamma \sum_{a'} \pi(s',a') Q^{\pi}(s',a')\right], \end{aligned} \tag{2.9}$$

where **Equation 2.9** is the bellman equation for Q^{π}.

As the value functions are defined for a policy, reinforcement learning algorithms try to find the best policy for solving the task, the *optimal policy* π^*. An optimal policy for a definite MDP is defined as $\pi^* \geq \pi$ if and only if $V^{\pi^*}(s) \geq V^{\pi}(s)$ for all $s \in \mathbb{S}$ and all policies π. All optimal policies, there is at least one policy that is better than all other policies, share the same state-value function, the *optimal state-value function*, $V^*(s)$, and it is defined as

$$\begin{aligned} V^*(s) &= \max_{\pi} V^{\pi}(s) \\ &= \max_{a \in \mathbf{A}(s)} \sum_{s'} P(s'|s,a)\left[R(s'|s,a) + \gamma V^*(s')\right], \end{aligned} \tag{2.10}$$

for all $s \in \mathbb{S}$. The optimal policies also share the same *optimal action-value function*, defined as

$$\begin{aligned} Q^*(s,a) &= \max_\pi Q^\pi(s,a) \\ &= \sum_{s'} P(s'|s,a)\left[R(s'|s,a) + \gamma \max_{a'} Q^*(s',a')\right], \end{aligned} \tag{2.11}$$

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

To illustrate the ideas so far consider the following example. **Figure 2.2** shows a

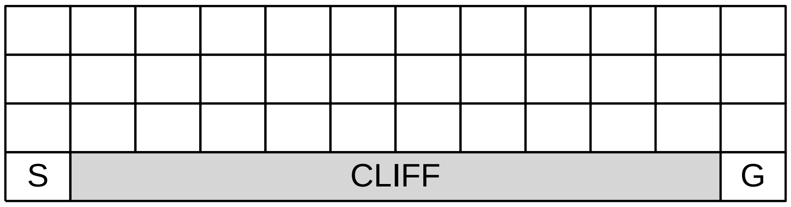


Figure 2.2. Cliff walking task

cliff walking task. The environment is a grid world and the goal for the agent is to move from start state, S, to the terminal goal state, G. The agent has four actions that move the agent 1-step up, down, right or left. The agent receives a -1 reward for every state transition, except for moving into the cliff region. In this case the agent receives a -100 reward and the agent is placed in start state, S, again. This is a standard undiscounted episodic task.

-14	-13	-12	-11	-10	-9	-8	-7	-6	-5	-4	-3
-13	-12	-11	-10	-9	-8	-7	-6	-5	-4	-3	-2
-12	-11	-10	-9	-8	-7	-6	-5	-4	-3	-2	-1
-13	CLIFF										0

Figure 2.3. Optimal state-value function for the cliff walking task

Figure 2.3 shows the optimal state-value function for the cliff walking task. The optimal state-value function for the start state, S, is -13, corresponding to that the shortest path contains 13 steps, moving along the edge of the cliff.

2.1.4 Action Selection

An important issue in RL is the so called *exploration-exploitation* problem during learning. In every state it is always at least one action whose estimated action value function is the largest, i.e. the *greedy action*, $a_t^* = \operatorname{argmax}_{a \in \mathbb{A}(s_t)} Q(s_t, a)$. If the agent chooses the greedy action, the agent exploits the current knowledge. This is often the right thing to do, especially if the learning has converged or if the agent wants to maximize the immediate reward. During learning, especially in the early stages, the agent has to explore the environment, by taking non-greedy actions. By exploring the environment the agent gets new knowledge and improves the estimates of the value functions.

To allow for exploration, the ϵ-*greedy* action selection is a simple alternative to the greedy action selection. The agent selects the greedy action most of the time, but with a small probability, ϵ, the agent selects a random action. The problem with ϵ-greedy action selection is that the agent selects all actions with the same probability when exploring. The natural alternative is to rank the actions according to their estimated action-value functions in the current state. This is called *softmax* action selection and the most common method uses a Boltzmann distribution. The selection probability for an action, a_t, at time step t is defined as

$$\mathbf{P}(a_t) = \frac{e^{Q_{t-1}(s_t,a_t)/\tau}}{\sum_{b\in\mathbb{A}(s_t)} e^{Q_{t-1}(s_t,b)/\tau}}, \tag{2.12}$$

where the positive parameter τ is called the *temperature*. When τ approaches infinity the action selection becomes random and when τ approaches zero the action selection becomes greedy. The right mix between exploration and exploitation depends strongly on the task and therefore it is very important to set the right τ or ϵ when designing RL algorithms. It is common to start with a large part of exploration and then gradually decrease the exploration rate, by decreasing τ or ϵ. In such case the agent acts randomly in the beginning of the learning and acts greedier as the learning converges.

2.1.5 Temporal Difference Learning

In RL there are three standard types of RL algorithms: *dynamic programming* (DP) methods, *Monte Carlo* methods and *temporal difference* (TD) learning methods. DP methods are algorithms that can calculate the optimal policy, given a perfect model of the environment as a MDP. The big drawback of DP methods are that they are computationally expensive. Monte Carlo algorithms do not need a perfect model of the environment, but are all based on averaging sample returns. They are therefore only applicable for episodic tasks, because good return estimates are only available after completing an episode. In this thesis only TD learning has been used and is the only type of RL that is covered in detail.

TD methods have the following features:

- They do not need a complete model of the environment.
- They estimate the state-value function or action-value function based on other earlier estimates of the value functions, i.e. they do not have to wait an episode until updating the estimates of the value functions, only 1 time step. This is called bootstrapping.

- They are naturally implemented online, due to the bootstrapping.

The central part of TD learning is the *TD error*, δ_t, defined as

$$\delta_t = \underbrace{r_{t+1} + \gamma V(s_{t+1})}_{value\ prediction} - V(s_t). \tag{2.13}$$

The TD error, the difference between temporally successive value predictions, is used to learn the value function. A positive TD error indicates that the action-value for selecting action a_t in state s_t, $Q(s_t, a_t)$, shall be larger and a negative TD error indicates that the action-value shall be smaller. The learning rule for updating the state-value function, $V(s_t)$, and the action-value function, $Q(s_t, a_t)$ in the most simple TD methods, TD(0) and SARSA(0), have the following definitions

$$V(s_t) \leftarrow V(s_t) + \alpha \underbrace{[r_{t+1} + \gamma V(s_{t+1}) - V(s_t)]}_{\delta_t} \tag{2.14}$$

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \underbrace{[r_{t+1} + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)]}_{\delta_t}, \tag{2.15}$$

where α is a step size parameter, called *learning rate*, $0 \leq \alpha \leq 1$. In each time step the value function estimate is updated by the TD error multiplied with the learning rate. The learning rate decides how much of the TD error that is going to be used in the update in each time step. In general, a low α gives slower but stable learning and a high α gives quicker but sometimes unstable learning. The pseudo code for the TD(0) algorithm is shown in **Algorithm 2.1**.

Algorithm 2.1 TD(0)

```
1: Initialize $V(s)$ arbitrarily and $\pi$ to the policy to be evaluated
2: for each episode do
3:   Initialize $s$
4:   repeat
5:     $a \leftarrow$ action given by $\pi$ for $s$
6:     Take action $a$, observe reward $r$, and next state s'
7:     $V(s) \leftarrow V(s) + \alpha [r + \gamma V(s') - V(s)]$
8:     $s \leftarrow s'$
9:   until $s$ is terminal
10: end for
```

The most famous reinforcement algorithm is maybe Watkins's *Q-learning* [19]. In Q-learning the agent estimates the optimal action-value function, Q^*, directly, instead of estimating the state-value function for a policy, V^π, or the action-value function for a policy, Q^π, as in previous algorithms. In the simplest form, *1-step Q-learning*, the Q-values are updated as

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \left[r_{t+1} + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t) \right] \tag{2.16}$$

By taking the maximum over all actions, a, of the Q-values for the next state, s_{t+1}, Q-learning becomes independent of the policy. It has been proved that Q-learning converges relatively fast with the probability one to Q^*. This holds under some basic constraints, such as that the learning rate is sufficiently small and that all state-action pairs are continually visited and updated. The pseudo code for 1-step Q-learning is shown in **Algorithm 2.2**. Note that the policy is still important. The policy is responsible for that all state-action pairs are visited, i.e. to handle the exploration-exploitation problem.

Algorithm 2.2 1-step Q-learning

```
1: Initialize $Q(s,a)$ arbitrarily
2: for each episode do
3:   Initialize $s$
4:   repeat
5:     Choose $a$ from $s$ using policy derived from $Q$ (e.g. $\epsilon$-greedy or softmax)
6:     Take action $a$, observe reward $r$, and next state, $s'$
7:     $Q(s,a) \leftarrow Q(a,a) + \alpha[r + \gamma \max_{a'} Q(s',a') - Q(s,a)]$
8:     $s \leftarrow s'$
9:   until $s$ is terminal
10: end for
```

2.1.6 Function Approximation

To calculate the optimal value functions it is required that the values for all states are stored, for example in a lookup-table. For large scale problems or for problems where the state signal is continuous this is not possible. In general, the learning time grows exponentially with the size of the state space.

To handle large state spaces or continuous state information the value functions have to be approximated. The idea is that the approximator, the *function approximator*, has the ability to generalize from experience gathered in a limited subset of the state space to a larger subset of the state space. Function approximation is a form of supervised learning and almost all supervised learning techniques can be used as function approximators, such as artificial neural networks, pattern recognition techniques and statistical methods. For example, the maybe most successful RL application yet, Gerry Tesauro's TD-gammon [17], uses a neural network to approximate the state-value function for a state space of size larger than 10^{20}. The latest version of TD-gammon plays backgammon at the same level as the best human players.

When using function approximation the approximated state-value function at time t, V_t, is represented as a parameterized functional form with parameter vector $\vec{\theta}_t$. As the purpose of the function approximation is to generalize, the number of parameters is normally much fewer than the number of possible states. If one parameter, $\theta_t(i)$, is changed the estimated values for several states are affected. It is therefore almost impossible to reduce the difference between the approximated value, $V_t(s)$, and the true value, $V^\pi(s)$, to zero for all states. The most widely used performance measure in supervised learning is the mean squared error (MSE) over some distribution, P, of the inputs. For RL tasks the inputs are states, which give the following MSE for the approximation of the true value $V^{\pi}(s)$ by $V_t(s)$, using the parameter vector $\vec{\theta}_t$

$$\mathrm{MSE}(\vec{\theta}_t) = \sum_{s \in \mathbb{S}} P(s) \left(V^{\pi} - V_t(s)\right)^2 , \tag{2.17}$$

where P is a distribution weighting the errors of different states.

One widely used type of function approximator is *gradient-descent* function approximation. The function approximator is represented by the column parameter vector $\vec{\theta}_t = (\theta_t(1), \theta_t(2), \ldots, \theta_t(n))^T$, where the approximated state-value function at time t, $V_t(s)$, is a smooth differentiable function of $\vec{\theta}_t$ for all $s \in \mathbb{A}(s)$. Here T denotes the vector transponate. The idea of gradient-descent methods is to adjust the parameter vector with a small amount in the direction that reduces the MSE the most, in every time step. This direction is equal to the negative gradient direction. If $f(\vec{\theta}_t)$ denotes the squared error, $(V^{\pi} - V_t(s))^2$, the gradient, $\nabla_{\vec{\theta}_t} f(\vec{\theta}_t)$, is equal to the vector of partial derivatives, $(\partial f(\vec{\theta}_t)/\partial\theta_t(1), \partial f(\vec{\theta}_t)/\partial\theta_t(2), \ldots, \partial f(\vec{\theta}_t)/\partial\theta_t(n))^T$. The MSE is reduced by successively adjusting the parameter vector as

$$\begin{aligned} \vec{\theta}_{t+1} &= \vec{\theta}_t - \frac{1}{2}\alpha \nabla_{\vec{\theta}_t} \left(V^{\pi}(s_t) - V_t(s_t)\right)^2 \\ &= \vec{\theta}_t + \alpha \left(V^{\pi}(s_t) - V_t(s_t)\right) \nabla_{\vec{\theta}_t} V_t(s_t). \end{aligned} \tag{2.18}$$

The problem is that it is not possible to use the true value of $V^{\pi}(s_t)$, because it is unknown. Instead an approximate has to be used and in TD learning the approximate is equal to the value prediction, $r_{t+1} + \gamma V_t(s_{t+1})$. This gives the following general update rule for TD learning

$$\vec{\theta}_{t+1} = \vec{\theta}_t + \alpha \left(r_{t+1} + \gamma V_t(s_{t+1}) - V_t(s_t)\right) \nabla_{\vec{\theta}_t} V_t(s_t). \tag{2.19}$$

A special case of gradient-descent methods is *linear gradient-descent* methods. In the linear case the approximated state-value function, $V_t(s)$, is a linear function of the parameter vector $\vec{\theta}_t$, which gives

$$V_t(s) = \vec{\theta}_t^T \vec{\phi}_s = \sum_{i=1}^{n} \theta_t(i)\phi_s(i), \tag{2.20}$$

where $\vec{\phi}_s$ is a column vector of features, $\vec{\phi}_s = (\phi_s(1), \phi_s(2), \ldots, \phi_s(n))^T$, for every state, s, and with same length as $\vec{\theta}_t$. Linear function approximators have the nice property that the gradient is only function of $\vec{\phi}_s$,

$$\nabla_{\vec{\theta}_t} V_t(s_t) = \vec{\phi}_s. \tag{2.21}$$

A natural function approximator for continuous state input is *normalized Gaussian radial basis functions (normalized Gaussian RBFs).* A normalized Gaussian feature, $\phi_s(i)$ is defined as

$$\phi_s(i) = \frac{e^{-\frac{\|s-c_i\|^2}{2\sigma_i^2}}}{\sum_{j=1}^{n} e^{-\frac{\|s-c_j\|^2}{2\sigma_j^2}}}. \tag{2.22}$$

The response of a normalized Gaussian is only depending on the distance between the center position of the feature and the current state, $\|s - c_i\|$, and the variance of the feature, σ_i. **Figure 2.4** shows an example, with 5 normalized Gaussian features placed equidistantly in the interval $[-60°, 60°]$, with $\sigma_i = 10$. The continuous input state signal is 1-dimensional and represents some sort of angle input to the agent.

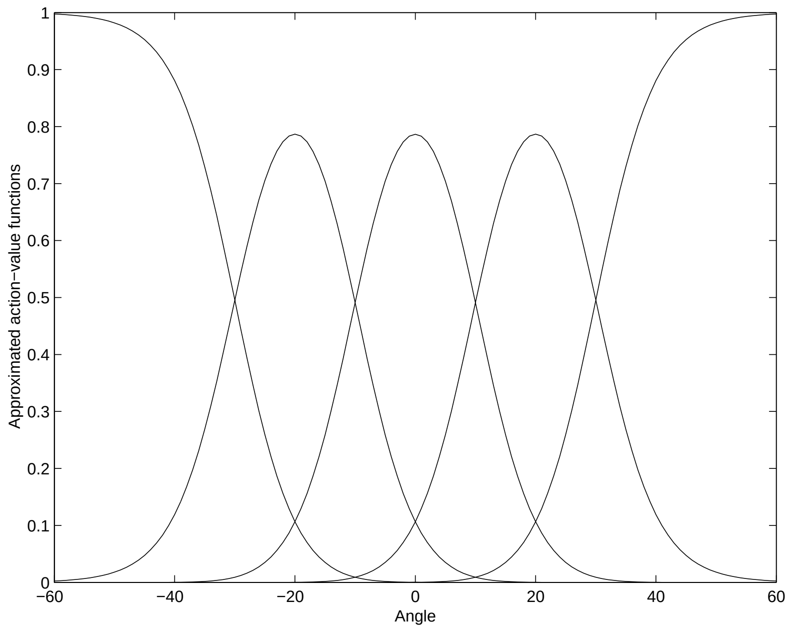


Figure 2.4. Five normalized Gaussian features

Similarly, the action-value function, $Q^{\pi}(s,a)$, is approximated by $Q_t(s,a)$ as

$$\vec{\theta}_{t+1,a_t} = \vec{\theta}_{t,a_t} + \alpha\left(r_{t+1} + \gamma Q_t(s_{t+1}, a_{t+1}) - Q_t(s_t, a_t)\right)\vec{\phi}_{s_t}, \tag{2.23}$$

using a linear gradient-descent function approximator, where $Q_t(s,a) = \vec{\theta}_a^T \vec{\phi}_s$. For each action, a, there is a column parameter vector, $\vec{\theta}_a$, representing the action-value function. The computation of the approximated value functions become a *linear Gaussian RBF network*. **Figure 2.5** shows a linear Gaussian RBF network approximating the action-value function, where the state space is of size m, the action space is of size k and there is n Gaussian features. A parameter $\theta_a(i)$ can be seen as a weight, deciding how much the feature $\phi_s(i)$ affects the computation of the action-value function for an action a. The outputs from the RBF network in a state, s, are equal to the approximated action-values, $Q_t(s,a)$, for all actions in that state, $a \in \mathbb{A}(s)$.

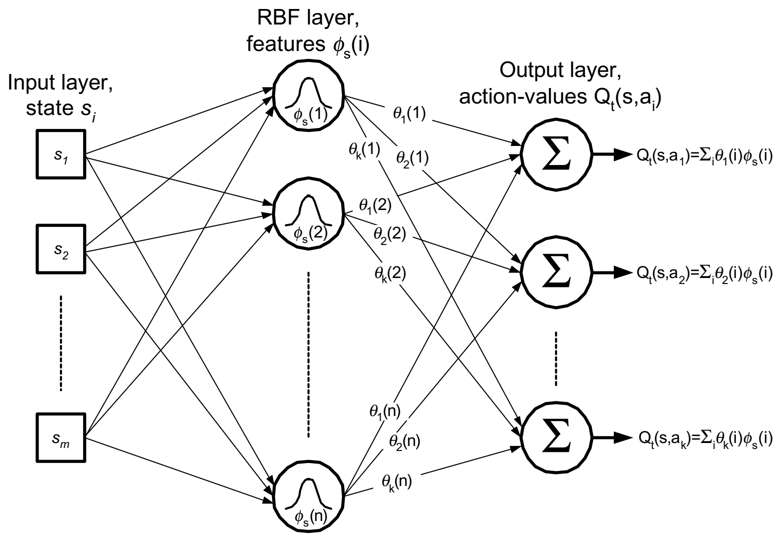

RBF layer, features $\phi_s(i)$
Input layer, state s_i
Output layer, action-values $Q_t(s,a_i)$
s_1
s_2
s_m
$\phi_s(1)$
$\phi_s(2)$
$\phi_s(n)$
$\theta_1(1)$
$\theta_k(1)$
$\theta_2(1)$
$\theta_1(2)$
$\theta_2(2)$
$\theta_k(2)$
$\theta_1(n)$
$\theta_2(n)$
$\theta_k(n)$
$Q_t(s,a_1)=\Sigma_i\theta_1(i)\phi_s(i)$
$Q_t(s,a_2)=\Sigma_i\theta_2(i)\phi_s(i)$
$Q_t(s,a_k)=\Sigma_i\theta_k(i)\phi_s(i)$

Figure 2.5. A linear RBF network approximating $Q_t(s,a)$

Algorithm 2.3 shows the pseudo code for 1-step Q-learning using a normalized Gaussian RBF-network as function approximator and ϵ-greedy action selection. Note that function approximators only approximate the input, i.e. the state signal. To handle continuous output, i.e. the actions, other methods not covered here are required.

2.2 Hierarchical Reinforcement Learning

As already mentioned RL suffers from the curse of dimensionality: the learning time grows exponentially with the size of the state space. Hierarchical RL algorithms are methods for introducing abstraction to the RL framework, to be able to apply RL to large scale problems. The goal of hierarchical RL is to find hierarchical structures in complex MDPs. This is realized by breaking down an overall task into smaller suitable subtasks. This gives a task hierarchy for the problem, where the actions

Algorithm 2.3 1-step Q-learning, using normalized Gaussian RBF network as function approximator, $\phi_s(i) = \frac{exp\left(-\frac{\|s-c_i\|^2}{2\sigma_i^2}\right)}{\sum_{j=1}^{n} exp\left(-\frac{\|s-c_j\|^2}{2\sigma_j^2}\right)}$, and ϵ-greedy action selection.

```
1: Initialize the parameter vectors $\vec{\theta}_a$ arbitrarily
2: **for each** episode **do**
3:    Initialize $s$
4:    **for** all $a \in \mathbb{A}(s)$ **do**
5:       $Q_a \leftarrow \sum_{i=1}^{n} \theta_a(i)\phi_s(i)$
6:    **end for**
7:    **repeat**
8:       **with** probability 1-$\epsilon$ **do**
9:          $a^* \leftarrow \text{argmax}_a Q_a$
10:      **else**
11:         $a^* \leftarrow$ random action $\in \mathbb{A}(s)$
12:      Take action $a^*$, observe reward $r$, and next state $s'$
13:      $\delta \leftarrow r - Q_{a^*}$
14:      **for** all $a \in \mathbb{A}(s')$ **do**
15:         $Q_a \leftarrow \sum_{i=1}^{n} \theta_a(i)\phi_{s'}(i)$
16:      **end for**
17:      $a' \leftarrow \text{argmax}_a Q_a$
18:      $\delta \leftarrow \delta + \gamma Q_{a'}$
19:      $\vec{\theta}_{a^*} \leftarrow \vec{\theta}_{a^*} + \alpha\delta\vec{\phi}_s$
20:      $s \leftarrow s'$
21:   **until** $s$ is terminal
22: **end for**
```

of the subtasks are other subtasks or primitive commands. In general the subtasks learn their own policies for a subset of the state space.

There are several different hierarchical RL algorithms. The probably most well known are the *options* formalism by Sutton, Precup and Singh [16], the *hierarchies of machines* (HAMs) by Parr and Russel [14], and the *MAXQ* framework by Dietterich [3], which is used in this thesis and is the only method that is covered in detail.

2.2.1 Semi-Markov Decision Process

Many of the subtasks in the task hierarchy represent abstract subgoals that often can not be accomplished in one time step. A subtask is therefore extended in time and the subtask is active until a well defined termination condition is fulfilled. A termination condition is either fulfilled because the goal of the subtask is completed or because the subtask is not applicable in the current state. For example, a subtask for an agent is to capture a battery pack using the angle to the battery pack, given by a vision system. The goal is naturally that the agent captures the battery pack, but if the agent looses sight of the battery pack the subtask also terminates, because the agent has no valid state information.

The *semi-Markov decision process (semi-MDP)* [1] model is a generalization of the MDP model. In a semi-MDP the actions take a variable amount of time to complete and semi-MDP is therefore a natural mathematical foundation for hierarchical reinforcement learning. For discrete semi-MDPs the state transition probability function, expressed as **Equation 2.6** for MDPs, is extended to a joint distribution of the next state, s', and number of discrete time steps, N, given the action a executed in state s:

$$P(s', N|s, a) = \mathbf{P}\left\{s_{t+1} = s', N|s_t = s, a_t = a\right\}. \tag{2.24}$$

The expected reward, **equation 2.7** for MDPs, is also depending on the number of time steps, which gives

$$R(s', N|s, a) = \mathbf{E}\left\{r_{t+1}|s_t = s, a_t = a, s_{t+1} = s', N\right\}. \tag{2.25}$$

Using this two definitions the semi-MDP versions of the Bellman equations, see **Equation 2.8** and **2.9** for the MDP versions, become

$$V^\pi(s) = \sum_a \pi(s, a) \sum_{s', N} P(s', N|s, a)\left[R(s', N|s, a) + \gamma^N V^\pi(s')\right] \tag{2.26}$$

$$Q^\pi(s, a) = \sum_{s', N} P(s', N|s, a)\left[R(s', N|s, a) + \gamma^N \sum_{a'} \pi(s', a') Q^\pi(s', a')\right]. \tag{2.27}$$

Note that the discount factor, γ, is decreased exponentially by the number of time steps, N, it takes to complete action a.

2.2.2 MAXQ

MAXQ [3] is a hierarchical RL framework. To be able to use the MAXQ framework the designer has to be able to decompose an overall task into suitable subtasks. This gives a MAXQ graph of the task. The tree graph consists of two different types of subtasks: primitive subtasks, leaf nodes, that execute commands to the agent and composite subtasks, inner nodes, that select other subtasks to solve their tasks. The MAXQ method uses the MAXQ graph to describe how to decompose the overall value function for a policy into a collection of value functions for the subtasks, recursively.

To illustrate the MAXQ method Dietterich's taxi problem [3] is used as an example. Figure 2.6 shows a 5x5 grid world, where the task is for a taxi agent to pickup and putdown passengers at specified locations, taxi stands. There are four

[1]This formulation of semi-MDPs follows Dietterich's [3], which is different from the standard formulation. The difference is due to that in the standard formulation do an action, a, terminate after the continuous time t, which gives more complex expressions.

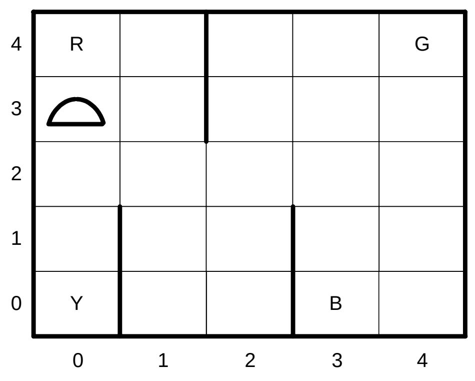


Figure 2.6. The taxi problem

taxi stands marked with R, B, G, and Y in the figure. A passenger is placed randomly at one of the taxi stands (the "source") and the destination of the passenger is one of the taxi stands (the "destination"), which can be the same as the "source". The task for the taxi agent is to go to the "source", pickup the passenger, go to the "destination" and putdown the passenger there.

The taxi problem consists of six primitive actions:

- four moving actions that move the taxi 1-step in the directions North, South, East or West.
- a Pickup action.
- a Putdown action.

The taxi agent receives a -1 reward for every primitive action executed. In addition the taxi agent receives a $+20$ reward for successfully delivering the passenger at the "destination" and a -10 reward for executing an illegal Pickup or Putdown action.

Figure 2.7 shows a task decomposition graph for the taxi problem. The Root node represents the overall task. The overall task is decomposed into two main subtasks: Get, for getting the passenger and Put, for delivering the passenger. The Get and Put subtasks use the Navigate(t) subtask to move the taxi to the right location, t, and the Pickup and Putdown subtask, respectively, to complete their tasks.

Formally, the MAXQ decomposition takes a given MDP M and decomposes it into a set of subtasks $\{M_0, M_1, \ldots, M_n\}$, where M_0 is the root subtask.

Definition 2.1. A subtask is a three-tuple, $(T_i, A_i, \tilde{R}_i)$, defined as follows

- $T_i(s_i)$ is a termination predicate that partitions $\mathbb{S}$ into a set of *active states*, S_i, and a set of *terminal states*, T_i. The policy for a subtask M_i can only be executed if the current state $s_i \in S_i$.

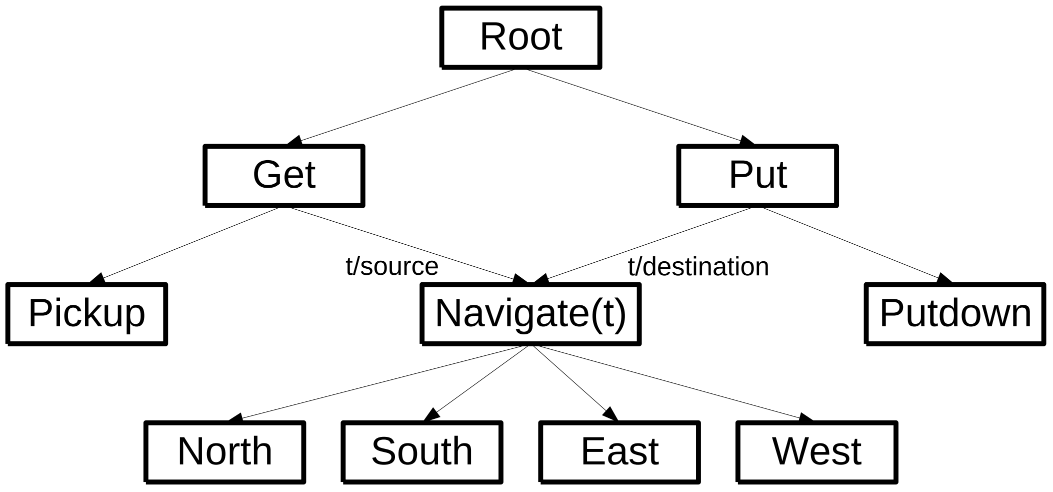


Figure 2.7. Task decomposition of the taxi problem

- A_i is a set that can be performed to achieve subtask M_i. These actions can either be primitive actions $\in \mathbb{A}$, the set of primitive actions for M, or they can be other subtasks, denoted i.

- $\tilde{R}_i(s'|s,a)$ is the *pseudo-reward* function, which specifies a pseudo-reward for each transition from a state $s \in S_i$ to a terminal state $s' \in T_i$. The pseudo-reward tells how desirable each of the terminal states is for the subtask. Typically, goal terminal states have a pseudo-reward of 0 and all non-goal terminal states have negative pseudo-rewards.

Each primitive action a from M is a primitive subtask in the MAXQ decomposition, such that a is always executable, it always terminates immediately after execution, and its pseudo-reward function is uniformly zero.

In the MAXQ method the subtasks have their own policies. The collection of policies is called a *hierarchical policy.* A hierarchical policy, π, is a set containing a policy for each of the subtasks in the problem: $\pi = \{\pi_0, \ldots, \pi_n\}$. In a hierarchical policy each subroutine executes until it enters a terminal state $\in T_i$ for its subtask i.

The objective of learning in the MAXQ method is to find a *recursively optimal policy*, defined as

Definition 2.2. 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 π_i is optimal for the semi-MDP defined by states S_i, the set of action A_i, the state transition probability function $P^\pi(s', N|s, a)$, and the reward function of the original reward function $R(s'|s, a)$ and the pseudo-reward function $\tilde{R}(s')$.

Recursive optimality is a form of local optimality, in which the policy at each node is optimal given the policy of its children. A subtask tries to find the optimal policy without reference to the parent node's policy.

The pseudo-reward function specifies how good each terminal state is for a subtask. As for the normal reward function it is very important that the designer carefully defines the pseudo-reward function, to achieve the desired behavior of the agent. It is of course possible to design a very specific pseudo-reward function for a problem, but the following very simply definition of $\tilde{R}$ has proven to work well [3]. For each subtasks two predicates is defined: the termination predicate, T_i, and the goal predicate, G_i. The goal predicate defines a subset of the terminated states that are goal states for the subtask, and these have the pseudo-reward of 0. All other terminated states have a fixed constant negative pseudo-reward. This tells the agent that it is always better to terminate in a goal state than in a non-goal-state.

The value function, $V^{\pi}(s)$, for executing a hierarchical policy, π, is called the *projected value function*. The projected value function is the value for executing π, starting in state s and at the root of the task hierarchy. The projected value function, $V^{\pi}(i, s)$, for subtask i in state s is decomposed into two parts, defined as

$$V^{\pi}(i,s) = \begin{cases} Q^{\pi}(i,s,\pi_i(s)) & \text{if } i \text{ is composite} \\ \sum_{s'} P(s'|s,i)R(s'|s,i) & \text{if } i \text{ is primitive,} \end{cases} \tag{2.28}$$

where R is the reward function for the primitive subtasks and $Q^{\pi}(i, s, a)$ is recursively defined as

$$Q^{\pi}(i,s,a) = V^{\pi}(a,s) + C^{\pi}(i,s,a). \tag{2.29}$$

$C^{\pi}(i, s, a)$ is called the *completion function* and is defined as the discounted cumulative reward for completing subtask M_i after invoking the subroutine for subtask M_a in state s:

$$C^{\pi}(i,s,a) = \sum_{s',N} P_i^{\pi}(s',N|s,a)\gamma^N Q^{\pi}(i,s',\pi(s')). \tag{2.30}$$

Equations 2.28, **2.29** and **2.30**, the *decomposition equations*, tell how to decompose the projected value for the root, $V^{\pi}(0, s)$, into projected value functions for the individual subtasks, $\{M_0, M_1, \ldots, M_n\}$, and the individual completion functions, $C^{\pi}(j, s, a)$, for $j = 1, \ldots, n$. The projected value function is stored explicitly as V values for all primitive actions and implicitly as C values for all composite subtasks.

The MAXQ task decomposition is graphically represented by a *MAXQ graph*. The MAXQ graph for the taxi problem is shown in **Figure 2.8**. The MAXQ graph contains two kind of nodes, Max nodes and Q nodes. The Max nodes represent the subtasks in the task decomposition, see **Figure 2.7**. The Q nodes represent the actions that are available for each subtask. The Max nodes can be seen as computing the projected value function $V^{\pi}(i, s)$, for subtask i, by asking the Q nodes for the value of $Q^{\pi}(i, s, a)$. The Q nodes obtain these values by asking the children, a, for the projected value function, $V^{\pi}(a, s)$, and then adding the completion function

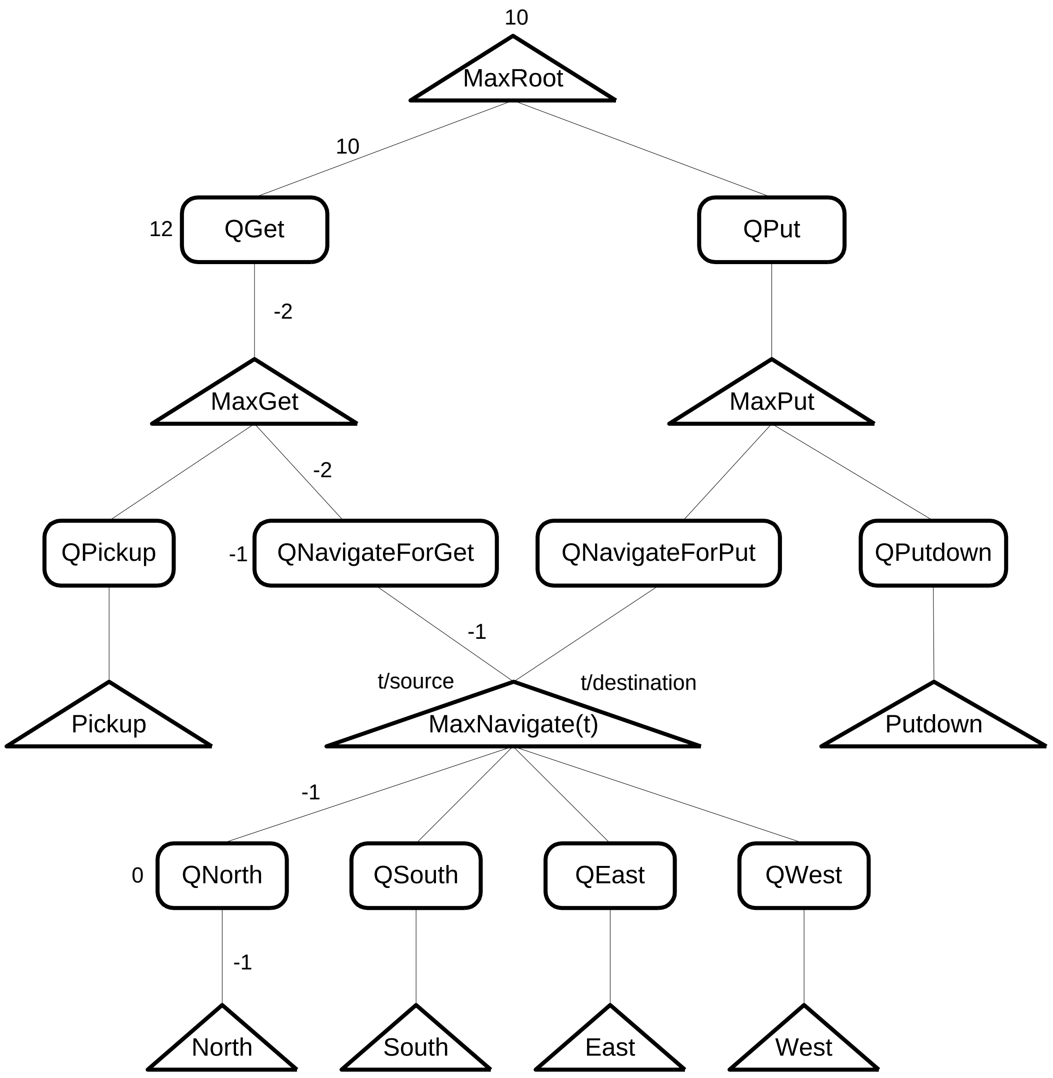


Figure 2.8. MAXQ graph of the taxi problem. Triangles denote Max nodes and rounded boxes show Q nodes.

$C^{\pi}(i, s, a)$. For example, consider the situation for the taxi problem in **Figure 2.6**, denoted as state s_1. Suppose that the passenger is at taxi stand R and wants to go to taxi stand B, and that the taxi agent executes the optimal hierarchical policy, π^*. The value of the state s_1 is 10: it will cost 1 unit to move the taxi to R, 1 unit to pickup the passenger, 7 units to move the taxi to B, 1 unit to putdown the passenger and finally the agent receives 20 units for completing the task. The MAXQ graph, **Figure 2.8**, also shows the Q and C values for the relevant nodes in state s_1. The recursive computation of the projected value function for the root node $V^{\pi^*}(0, s_1)$ is performed as follows

- $Q^{\pi^*}(\mathsf{Navigate}(R), s_1, \mathsf{North}) = -1 + 0$
- $V^{\pi^*}(\mathsf{Navigate}(R), s_1) = -1$
- $Q^{\pi^*}(\mathsf{Get}, s_1, \mathsf{Navigate}(R)) = -1 + -1$
  (-1 to perform Navigate plus -1 to complete Get)
- $V^{\pi^*}(\mathsf{Get}, s_1) = -2$
- $Q^{\pi^*}(\mathsf{Root}, s_1, \mathsf{Get}) = -2 + 12$
  (-2 to perform Get plus $+12$ to complete the Root task).

In general the decomposition of the projected value function can be expressed as

$$V^{\pi}(0, s) = V^{\pi}(a_m, s) + C^{\pi}(a_{m-1}, s, a_m) + \ldots + C^{\pi}(a_1, s, a_2) + C^{\pi}(0, s, a_1), \quad (2.31)$$

where $a_0, a_1, \ldots, a_m$ is the path of Max nodes from the root node to a primitive leaf node, given by the hierarchical policy, π.

The learning algorithm in the MAXQ method is called *MAXQ-Q*. To be able to learn recursively optimal policies MAXQ-Q uses two completion functions, C and $\tilde{C}$. C is the normal completion discussed so far in this thesis. C is used by the parent task to compute $V(i, s)$, the expected reward for performing action i starting in state s. The second completion function, $\tilde{C}$, is only used inside node i in order to discover the local optimal policy for M_i. $\tilde{C}$ uses both the "real" reward function, $R(s'|s, a)$ and the pseudo-reward function, $\tilde{R}_i(s')$.

The pseudo code for MAXQ-Q is shown in **Algorithm 2.4**. The learning rule for updating $\tilde{C}$, line 16, is a kind of Q-learning for semi-MDP and the learning rule for C, line 17, is similar to SARSA for semi-MDP. The learning of the projected value function for the primitive actions, line 5, is accomplished by SARSA or Q-learning where the discount factor, γ, is set to zero. This means that the primitive actions only try to maximize the immediate reward and do not perform any value prediction. Note that MAXQ-Q uses batch updating of the completion functions. The recursive call at line 11 returns a list of all states visited during the execution of the call. At line 16 and 17 the completion functions are updated for all visited states, starting with the most recent visited state.

The projected value function, $V(a, s)$, for the composite subtasks, line 13, 16 and 17, is recursively computed as shown in **Algorithm 2.5**. At line 8, $\tilde{C}$ is used to find the local optimum "inside" the node, but when returning, at line 9, C is used to compute the projected value function "outside" the node.

2.3 Genetic Algorithms

Genetic algorithms (GA) are adaptation or optimization algorithms inspired by natural selection and evolution. As for all optimization problems, the task is to maximize or minimize an objective function $f(x)$ over a given space $x \in \mathbb{X}$ of arbitrary

Algorithm 2.4 MAXQ-Q

```
1: function MAXQ-Q(MaxNode $i$, State $s$)
2: $seq \leftarrow \{\}$ is the sequence of states visited while executing $i$
3: if $i$ is a primitive MaxNode then
4:    execute $i$, receive $r$, and observe next state $s'$
5:    $V(i,s) \leftarrow V(i,s) + \alpha[r - V(i,s)]$
6:    push $s$ into the beginning of $seq$
7: else
8:    $count \leftarrow 0$
9:    while $T_i(s)$ is false do
10:      choose an action $a$ according to the current policy $\pi_i(s)$
11:      $childSeq \leftarrow$ MAXQ-Q$(a,s)$, where $childSeq$ is the sequence of states visited
              while executing action $a$.
12:      observe next state $s'$
13:      $a* \leftarrow \mathrm{argmax}_{a'}[\tilde{C}(i,s',a') + V(a',s')]$
14:      $N \leftarrow \mathrm{length}(childSeq)$
15:      for each $s \in childSeq$ do
16:         $\tilde{C}(i,s,a) \leftarrow \tilde{C}(i,s,a)+\alpha(i)\{\gamma^N[\tilde{R}_i(s')+\tilde{C}(i,s',a^*)+V(a^*,s)]-\tilde{C}(i,s,a)\}$
17:         $C(i,s,a) \leftarrow C(i,s,a) + \alpha(i)\{\gamma^N[C(i,s',a^*) + V(a^*,s')] - C(i,s,a)\}$
18:         $N \leftarrow N - 1$
19:      end for
20:      append $childSeq$ onto the front of $seq$
21:      $s \leftarrow s'$
22:   end while
23: end if
24: return $seq$
```

Algorithm 2.5 Recursive computation of the projected value function in MAXQ-Q

```
1: function EvaluateMaxNode(MaxNode $i$, State $s$)
2: if $i$ is a primitive MaxNode then
3:    return $V(i,s)$
4: else
5:    for each $j \in A_i$ do
6:       $V(j,s) \leftarrow$ EvaluateMaxNode$(j,\ s)$
7:    end for
8:    $j^* \leftarrow \mathrm{argmax}_j[V(j,s) + \tilde{C}(i,s,j)]$
9:    return $V(j^*,s) + C(i,s,j^*)$
10: end if
```

dimension. In GA the potential solutions of the optimization problem are represented by a *population* of competing individuals. After each *generation* the individuals are evaluated according to how well they are able to maximize or minimize $f(x)$ and are thereby assigned a *fitness* value. After a lifetime of the current population a new generation is *evolved* by applying *genetic operations* to the current population, where *selection* of individuals are based on their fitness values. The genetic operations are inspired by reproduction and mutation in biological evolution, where selection of individuals are based on Darwin's "survival of the fittest" principal.

2.3.1 Concept

The terminology in GA comes from biology and the basic concepts are described below.

- *gene*, a functional entity that encodes a certain feature. In animals there is for example a certain gene that codes eye color.
- *genome*, the set of genes that completely defines the individuals in the population, i.e. the encoding of the individuals.
- *genotype*, the genome of a specific individual.
- *phenotype*, the interpretation of the genotype for the given optimization problem. In biology the genome is represented by the DNA of a species and the phenotype is represented by the physical make-up of an individual of that species.
- *population*, the set competing genotypes/individuals.

The encoding of the individuals is of course a very important issue. In the simplest encoding scheme, binary encoding, the genome consists of a bit string of fixed length. Each gene is a single bit, as shown in the example below.

$$\overbrace{0\quad 1\quad 1\quad \underbrace{0}_{4th\ gene}\quad 1\quad 1\quad 0\quad 1}^{genome}$$

To evaluate the performance of the individuals in the population, each individual is assigned a *fitness value*, corresponding to how well the individual solves the optimization problem. The fitness values are calculated either directly from the genotypes in simple cases or from the phenotypes in more complex cases, e.g. when learning is involved.

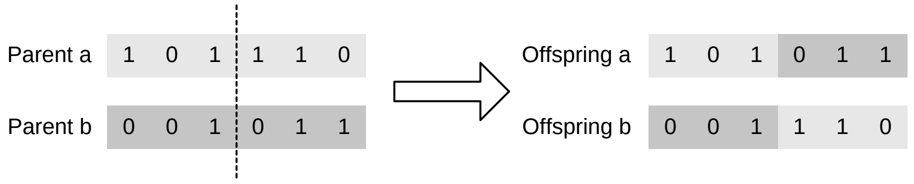
The selection of individuals for genetic operations is based on the fitness values, according to Darwin's survival principal. A basic selection method is to select individuals proportionate to their fitness value:

$$\mathbf{P}(x_i) = \frac{f(x_i)}{\sum_{i=1}^{n} f(x_j)}, \tag{2.32}$$

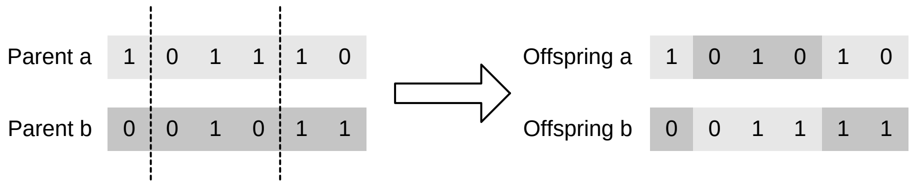
where $f(x_i)$ is the fitness value for individual x_i, $i = 1, \ldots, n$. Another widely used selection method is *tournament selection*. In tournament selection with *tournament* *size* of k, k individuals are chosen randomly with uniform probability and the individual with the largest fitness value, among the randomly chosen, is selected.

There are three types of genetic operators: *reproduction*, *crossover* and *mutation*. Reproduction is simply to copy the genome of one or more individuals to the next generation. The reproduced individuals can either be chosen by some selection method or be the individuals with the largest fitness values, to ensure that the best individuals survive.

Crossover corresponds to reproduction in biology and therefore requires two parent individuals. To create the offsprings a crossover position is chosen randomly with uniform probability and substrings are then switched between the parents, according to the crossover point. **Figure 2.9** shows two types of crossover, 1-point crossover and 2-point crossover, for bit strings.



(a) 1-point crossover



(b) 2-point crossover

Figure 2.9. Crossover in GA

Mutation corresponds to genetic mutation in biology. The purpose of the mutation operator is to maintain genetic diversity in the population. The mutation operator works like the reproduction operator, but some part of the genome is randomly altered. For bit strings this equals that one or more bits are flipped. **Figure 2.10** shows an example of mutation for a bit string, where a randomly chosen bit is flipped, the 5th bit in this example. In general, mutation shall occur with very low probability, $p_m \in [0.001 \ldots 0.1]$.

The evolution process stops when some predefined termination condition is fulfilled. Examples of termination conditions are that a predefined percentage of population has the same genome, the difference between the best solution and the optimal solution is sufficiently small (requires that the optimal solution is known), or that a fixed predefined number of generations have been evolved. **Algorithm 2.6** shows

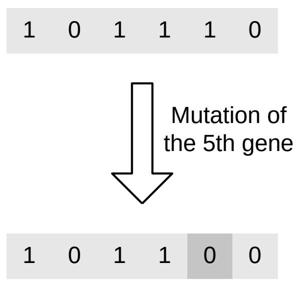


Figure 2.10. Mutation in GA

the general scheme for the evolutionary process in GA.

Algorithm 2.6 Evolutionary scheme for GA

```
1: Create the initial population of individuals
2: loop
3:   Evaluate the fitness of each individual
4:   if Termination condition is fulfilled then
5:     break {Terminate the evolution process}
6:   else
7:     Create a new population by reproduction, crossover and mutation of indi-
       viduals
8:   end if
9: end loop
```

An example of an evolutionary approach to solve the Brachystrochrone problem by a GA is shown in **Figure 2.11**. The objective is to optimize the track between the start and the end point, so that a frictionless point mass travels the path in minimal time. The fitness of individuals is computed as the time for point mass to travel from the start point to the end point. The genome represents heights of the track at n points along the track and the parameters are binary encoded by bit strings: $y : \{y_0, y_1, \ldots, y_n\}$. A parameter, y_i, is the height of the track at point i: $y_i = \sum_{j=1}^{k} s_j 2^{k-1}$, where k is the length of the parameters and s_j is the bit at position j. The parameters has the range $[0, 2^k - 1]$ and to scale the parameters to a range suitable for the problem, $[a, b]$, the following equation is used

$$y_i = a + \frac{(b-a)}{2^k - 1} \sum_{j=1}^{k} s_j 2^{k-1} \tag{2.33}$$

The type of evolution discussed so far is called *Darwinian evolution*, where only

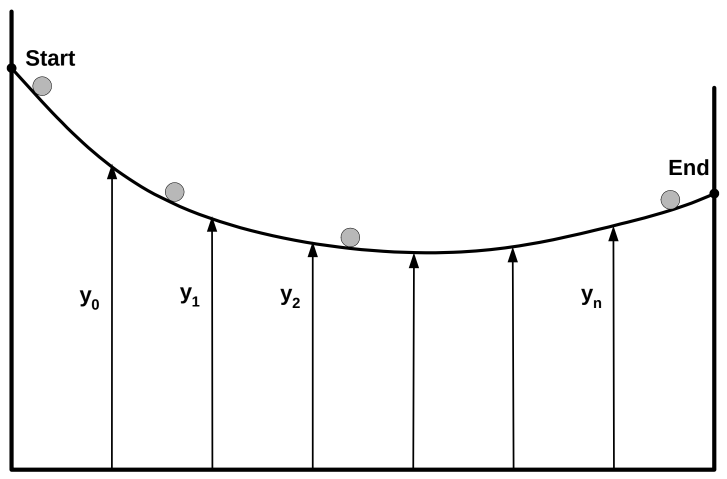


Figure 2.11. GA applied to the Brachystrochrone problem

genome of the individuals is inherited to the next population. Another type of evolution is *Lamarckian evolution*, where both the genome and the skills or behaviors learned during the lifetime of the individuals are inherited to the next population. Lamarckian evolution has no correspondence to biological evolution; in such a case can skills like riding a bike be inherited to newborn human babies. In evolutionary computation Lamarckian evolution is both possible and can speed up the evolutionary process, since the individuals in newly created populations do not need to learn behaviors from scratch.

2.3.2 Genetic Programming

GP [9] is a specialization of GA where the genome is represented by a tree structure. The most common application of GP is automation of computer programs written in the LISP language. The programs are represented by a parse tree of LISP expressions, as shown in **Figure 2.12**. The tree structure consists of a *function set* and a *terminal set*. The function set corresponds to the inner nodes and represents functions like

- arithmetic functions $\{+, -, \times, /\}$
- logarithmic functions $\{sin, exp, \ldots\}$
- if-else statements

The terminal set corresponds to the leaf nodes and represents

- input variables $\{x_1, x_2, \ldots\}$
- constants

The trees are parsed from left to right, giving the following expression for the tree in **Figure 2.12**

$$(+x_1(\times x_2x_3)) \Longrightarrow x_1 + (x_2 \times x_3) \tag{2.34}$$

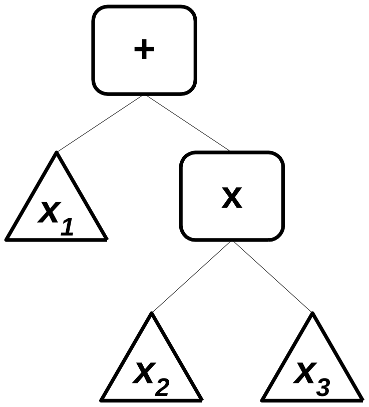


Figure 2.12. Parse tree of LISP expressions

The crossover operation in GP is accomplished by choosing a crossover node in each parent tree structure randomly with uniform probability. The two offsprings are then created by switching subtrees at the crossover nodes between the two parents. An example of crossover in GP is shown in **Figure 2.13**.

The general strength of evolutionary computation, GP as well as GA, is the ability to explore large solution spaces without getting trapped in local minimums. Evolutionary methods are therefore referred to as global optimization methods. The special strength of standard GP is that the obtained solutions are another computer program, not a quantities as in GA. This makes GP very flexible and GP is especially suitable for problems without an analytical solution or when the solution is constantly changing.

The main weakness of evolutionary computation is that it is very time consuming. Another limitation of the use of evolutionary computation is that the algorithms have to be adjusted to the current problem. The designer has to select and fine-tune population size, selection method, termination condition and other problem depend properties. For GP the designer also has to select good terminal and function sets, which is non-trivial for many problems.

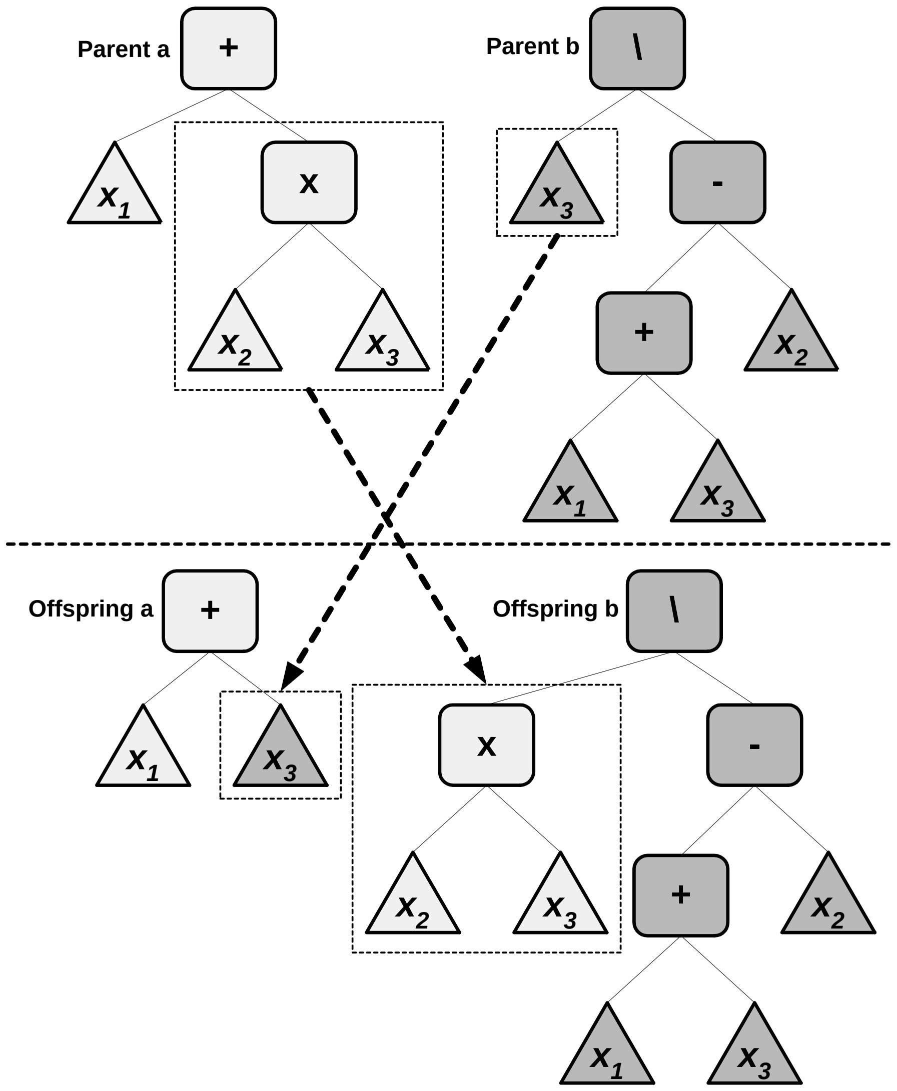


Figure 2.13. Crossover in GP

Chapter 3

Exploration of the MAXQ Hierarchies by GP

The MAXQ method learns a good policy much faster than the standard flat RL methods, but MAXQ provides no way of learning or adapting the structure of the task hierarchy. In this thesis genetic programming is applied to obtain the appropriate task hierarchy, according to the environmental settings.

The basic assumption of the proposed method is that the agent has a number of competing strategies for solving different tasks. A few simple basic behaviors are the building blocks for these strategies. By combing the basic behaviors with more abstract behaviors the agent builds different task hierarchies to solve the tasks. The purpose of the abstract behaviors is to select the correct basic behaviors according to current state information. During the lifetime of the agent, the agent mixes the competing strategies, to come up with a strategy that is adapted to the learning task and environment.

3.1 MAXQ Implementation

MAXQ graphs represent different strategies for the agent to solve a given task. The leaf nodes in the MAXQ graphs represent the basic behaviors that the agent already know and that function as building blocks for the hierarchies. Therefore, there exists only one copy of the trained basic behaviors represented by the leaf nodes in the MAXQ graphs and they are shared by all hierarchies in the population.

In the standard MAXQ method the task is decomposed down to the level where each primitive action corresponds to exactly one command to the agent. In this thesis the considered task is not decomposed down to that level of detail. The primitive subtasks, for example approach the nest or a battery pack, are simple tasks that use multiple states and actions. Therefore, in the proposed method each leaf node in the MAXQ decomposition (hereafter denoted *primitive subtask*) represents a simple subtask with several actions, corresponding to primitive commands to the agent, to select from in each state. Instead of storing the state-value function directly the primitive subtasks store the action-values function, $Q^{\pi}_{prim}(i,s,a)$, where $a \in A_i$. This modifies the first decomposition equation, **Equation 2.28**, as

$$V^{\pi}(i,s) = \begin{cases} Q^{\pi}(i,s,\pi_i(s)) & \text{if } i \text{ is composite} \\ Q^{\pi}_{prim}(i,s,\pi_i(s)) = \sum_{s'} P(s'|s,\pi_i(s)) R(s'|s,\pi_i(s)) & \text{if } i \text{ is primitive.} \end{cases} \tag{3.1}$$

The computation of the decomposed value function for the primitive subtasks at line 3 in **Algorithm 2.5** is therefore modified as

$$V^{\pi}(i,s) = \max_a Q^{\pi}_{prim}(i,s,a) \tag{3.2}$$

and the learning rule for the primitive subtasks, line 5 in **Algorithm 2.4**, is replaced by

$$Q_{prim}(i,s,\pi_i(s)) \leftarrow Q_{prim}(i,s,\pi_i(s)) + \alpha(i)\left[r - Q_{prim}(i,s,\pi_i(s))\right]. \tag{3.3}$$

All types of composite subtasks representing abstract behaviors have also to be given by the designer. For each type of composite subtask all properties that are required by the MAXQ framework have to be provided, such as state space, termination set and goal set. The possible actions of a composite subtask, i.e other composite subtasks and primitive subtasks, are determined by the state spaces of the subtask and the actions. This gives a restriction on the number of possible hierarchies that is allowed to be constructed.

One aim of this study is that the agent shall behave as an autonomous agent, using local and continuous state information. To handle the continuous state input the value functions and completion functions are approximated by normalized Gaussian RBF networks. The approximate of the completion functions, $\tilde{C}_t$ and C_t, and the approximate of the action-value function for the primitive subtasks, $Q_{t,prim}$, are represented with the parameter vectors $\vec{\theta}^{\tilde{C}}_{i,a}$, $\vec{\theta}^{C}_{i,a}$ and $\vec{\theta}^{Q_{prim}}_{i,a}$, respectively. The learning rules at line 5, 16 and 17 in **Algorithm 2.4** are modified according to **Equation 2.23** as

$$\vec{\theta}^{\tilde{C}}_{i,a} \leftarrow \vec{\theta}^{\tilde{C}}_{i,a} + \alpha(i)\left[\gamma^N \left[\tilde{R}_i(s') + \tilde{C}_t(i,s',a^*) + V_t(a^*,s)\right] - \tilde{C}_t(i,s,a)\right] \vec{\phi}^{\tilde{C}}_{i,s} \tag{3.4}$$

$$\vec{\theta}^{C}_{i,a} \leftarrow \vec{\theta}^{C}_{i,a} + \alpha(i)\left[\gamma^N \left[C_t(i,s',a^*) + V_t(a^*,s')\right] - C_t(i,s,a)\right] \vec{\phi}^{C}_{i,s} \tag{3.5}$$

$$\vec{\theta}^{Q_{prim}}_{i,a} \leftarrow \vec{\theta}^{Q_{prim}}_{i,a} + \alpha(i)\left[r - Q_{t,prim}(i,s,a)\right] \vec{\phi}^{Q_{prim}}_{i,s} \tag{3.6}$$

where

$$\tilde{C}_t(i,s,a) = \vec{\theta}^{\tilde{C}^T}_{i,a} \vec{\phi}^{\tilde{C}}_{i,s} \tag{3.7}$$

$$C_t(i,s,a) = \vec{\theta}^{C^T}_{i,a} \vec{\phi}^{C}_{i,s} \tag{3.8}$$

$$Q_{t,prim}(i,s,a) = \vec{\theta}^{Q_{prim}^T}_{i,a} \vec{\phi}^{Q_{prim}}_{i,s} \tag{3.9}$$

Softmax action selection is used in the MAXQ framework, according to **Equation 2.12**. Each subtask is assigned an initial temperature, τ_{init}. The temperature is then exponentially decreased by multiplying the current temperature, τ, with a temperature decrease factor, τ_{df}. The decreasing of temperature is made after a subtask has terminated, i.e. after line 22 in **Algorithm 2.4**. Experience has shown that is important to keep an amount of stochasticity in the action selection, to maintain the exploration of the environment. Therefore, the temperatures are only decreased down to a lower limit, τ_{limit}. Also, if the probability for selecting an action exceeds an upper limit, ρ_{limit} the softmax action selection is replaced by ϵ-greedy action selection, with ϵ set to $1 - \rho_{limit}$. This ensures that the probability for selecting a non-greedy action never becomes less than ϵ.

3.2 GP Implementation

The MAXQ task decomposition graphs represent the genome of the individuals in the population. The function and terminal sets are the composite subtasks and the primitive subtasks, respectively. To be able to apply GP to MAXQ graphs, the graphs are not allowed to contain cycles. For example the MAXQ graph for the taxi problem in **Figure 2.8** contains a cycle where both the subtasks Get and Put are connected to the subtask Navigate. The solution is that Get and Put are connected to identical subtrees representing the Navigate subtask, as shown in **Figure 3.1** (only Max nodes are shown). The learning of identical composite subtasks in a hierarchy is performed separately.

To limit the number of bad hierarchies that is allowed to be constructed a form of *strongly typed GP* [12] is used. In general, strongly typed GP allows the designer to assign a type to the arguments and the return value of each function. In the proposed method the type of a subtask is equal to its state space and the state space of a child node has to be a subset of its parent's state space.

In each generation each individual performs a fixed number of trials, in random order, to solve the task. The fitness value of the current population is computed over the last half of the trials. The aim of the evolutionary process, i.e. the termination condition for GP, is that the best performing half of the population, according to the fitness values, has identical hierarchical structures.

A small fixed number of the best performing hierarchies in the current generation survive and are reproduced to the next generation. An equal small number of the worst performing hierarchies die. The remaining hierarchies in the next population are created by crossover. The parent hierarchies involved in one crossover are chosen by tournament selection. When the tournament selection finds two parents, a crossover between the parents is performed until the following requirements are fulfilled.

- The new child node's state space has to be a subset of the parent node's state space.
- A hierarchy is only allowed to have a maximum of one copy of each composite subtask.

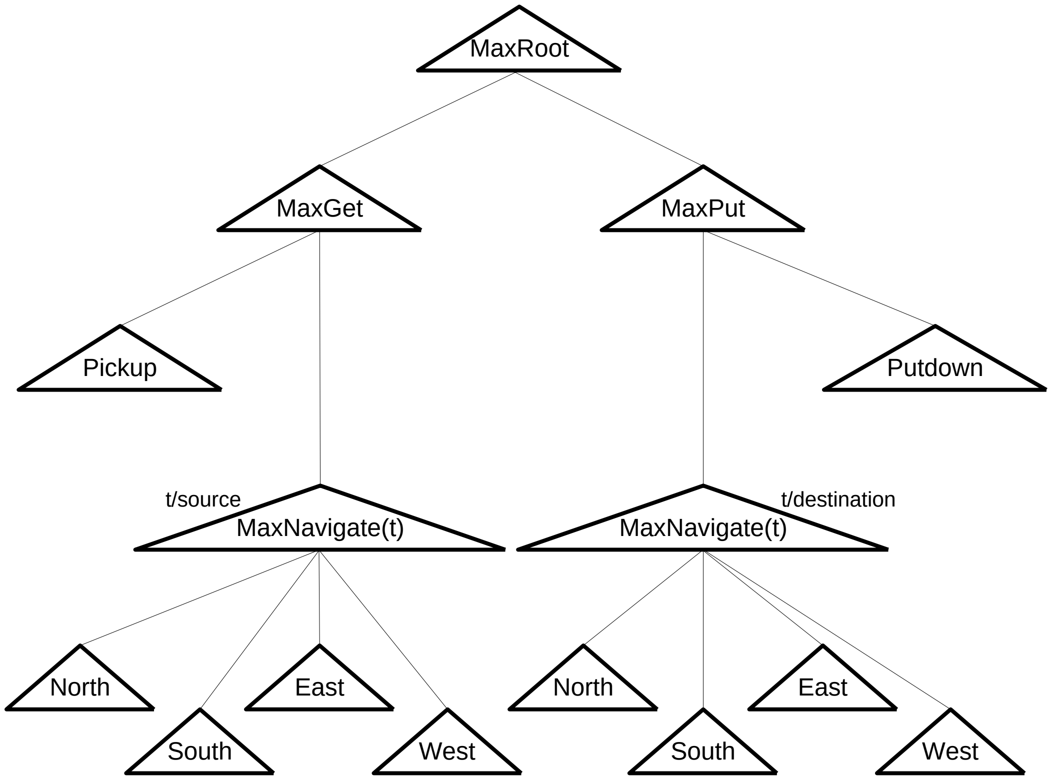


Figure 3.1. Acyclic MAXQ graph of the taxi problem.

- The root node, representing the overall task, has to be able to select an action in all types of states.

Crossover sites in the parents are chosen randomly among the Max nodes, excluding the root nodes. The crossover is performed by switching subtrees at the crossover nodes as shown in **Figure 2.13**, with the difference that the tree structures represent MAXQ graphs instead of LISP parse trees.

As the evolution is occurring within one agent during the agent's lifetime, Lamarckian evolution is used. Both the genome, the learning structure, and the learned behaviors in the hierarchies are inherited to the next population. Because the learning structure is changed in the hierarchies created by crossover it is not desirable to continue the learning process in the composite subtasks that are affected by the crossover. Therefore, the learning parameters are set to their initial values for the subtasks in the path from the parent of the new subtree to the root of the hierarchy. To prevent that the reproduced hierarchies, whose learning structures are unchanged, get an advantage over the hierarchies created by crossover the reproduced hierarchies perform only the last half of the trials, when the fitness values are computed. **Algorithm 3.1** shows the evolutionary scheme of the proposed method.

Algorithm 3.1 Evolutionary scheme for the proposed method

```
1: Hand code the initial population.
2: loop
3:    Let the hierarchies perform a fixed number of trials to solve the learning task,
      except for reproduced hierarchies that only perform the last half of the trials.
4:    Evaluate the fitness of each hierarchy, according to the performance of the
      last half of the trials.
5:    if the best performing half of the population has identical structure then
6:       break {Terminate the evolution process}
7:    else
8:       Reproduce a fixed number of the best performing hierarchies
9:       Create the remaining hierarchies in the new population by crossover opera-
         tions. Reset the learning parameters in composite subtasks in path from the
         parent of the new subtree to the root in the hierarchies created by crossover.
10:   end if
11: end loop
```

The initial population in the evolutionary process has to be carefully hand coded, line 1 in **Algorithm 3.1**. When only crossover is used to create new types of hierarchies it is not possible to add or remove the number of actions of the composite subtasks. The initial population has therefore to include hierarchies of as many types as possible, to ensure that the initial population has the potential of evolving all types of good hierarchies.

To hand code the initial population is, of course, not an ideal solution. For large scale problems it can be very difficult and also time consuming to design a good initial population. It is therefore important in the future, as an extension of the work presented in this thesis, to include a method for automatic construction of the initial population.

Chapter 4

Experiments

4.1 The Cyber Rodent Robot

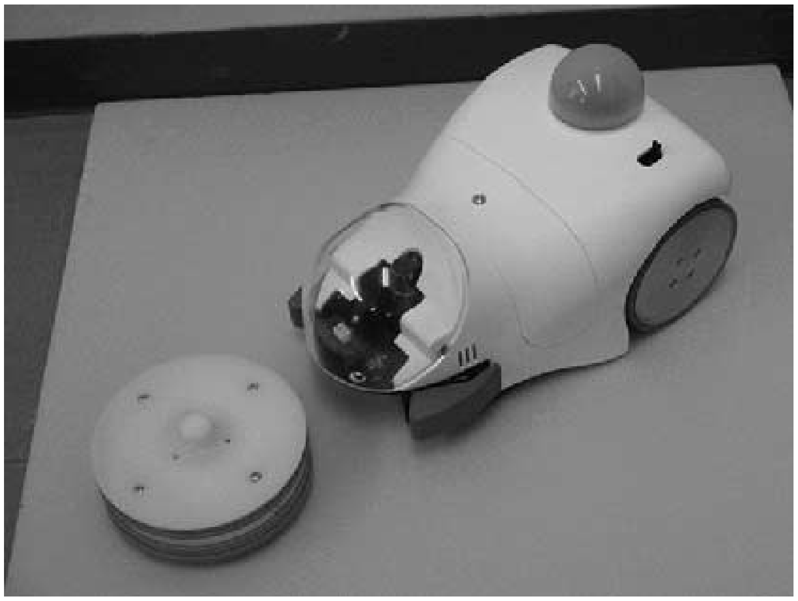


Figure 4.1. The Cyber Rodent robot with a battery pack

To achieve the goals of the Cyber Rodent project, see **Section 1.1**, a rodent like robot, Cyber Rodent, has been developed as experimental platform. The Cyber Rodent, see **Figure 4.1**, is a two-wheel driven mobile robot with a wide-angle C-MOS camera, an infrared range sensor, seven infrared proximity sensors, 3-axis acceleration sensors, 2-axis gyro sensors, a magnetic actuator for catching a battery pack, color LEDs for visual signaling, an audio speaker and two microphones for acoustic communication, an infrared port for communication with a near-by agent, and a wireless LAN card for communication with a host computer.

The experiments conducted in this thesis are performed in a simulated environment. The 2D-simulator is developed in MATLAB by the author and Anders Eriksson for the Cyber Rodent project [6]. The simulator simulates the proximity sensors and the vision system of the Cyber Rodent. In the experiments the five front proximity sensors are used, placed at 0° ±30° and ±90°, according to the real Cyber Rodent robot. The proximity sensors give accurate information about the distance to walls and obstacles in the effective range, set to 70-350 mm in the experiments. The vision system gives information about the distance and angle to colored targets, such as battery packs or the Cyber Rodent's nest. In the experiments the vision system has an angle limit of ±60° and a maximum visible distance of 3500 mm. For both the vision system and the proximity sensors 10% of noise is added to the readings in the experiments.

4.2 Learning Task and Provided Subtasks

The learning task used as experimental test bed in this thesis is a foraging task. The Cyber Rodent shall find, approach and capture a battery pack and then return the battery pack to the its nest. The performance of a hierarchy is measured as the number of time steps to perform the learning task. A small number of time steps is considered a good performance. In the context of GP the fitness value is computed as the number of time steps to perform the last half of the trials in a generation. To solve the task the Cyber Rodent uses continuous vision information about the angle and distance to the closest battery and the nest, and continuous distance sensor information from the five front proximity sensors.

During the performance of the task the agent can be in four different types of state as shown in **Table 4.1**.

State type	**Description**	**Relevant state information**
1	A battery is not visible and the agent has not captured a battery	Proximity sensors
2	A battery is visible and the agent has not captured a battery	Proximity sensors, angle and distance to closest battery
3	The nest is not visible and the agent has captured a battery	Proximity sensors
4	The nest is visible and the agent has captured a battery	Proximity sensors, angle and distance to the nest

Table 4.1. The four different types of state for the foraging task

Figure 4.2 shows an example of a complete task decomposition of the foraging task, using all subtasks, composite and primitive, that is provided to the Cyber Rodent. The different types of states are closely related to the composite subtasks that are provided to the Cyber Rodent for solving the task. **Table 4.2** shows the

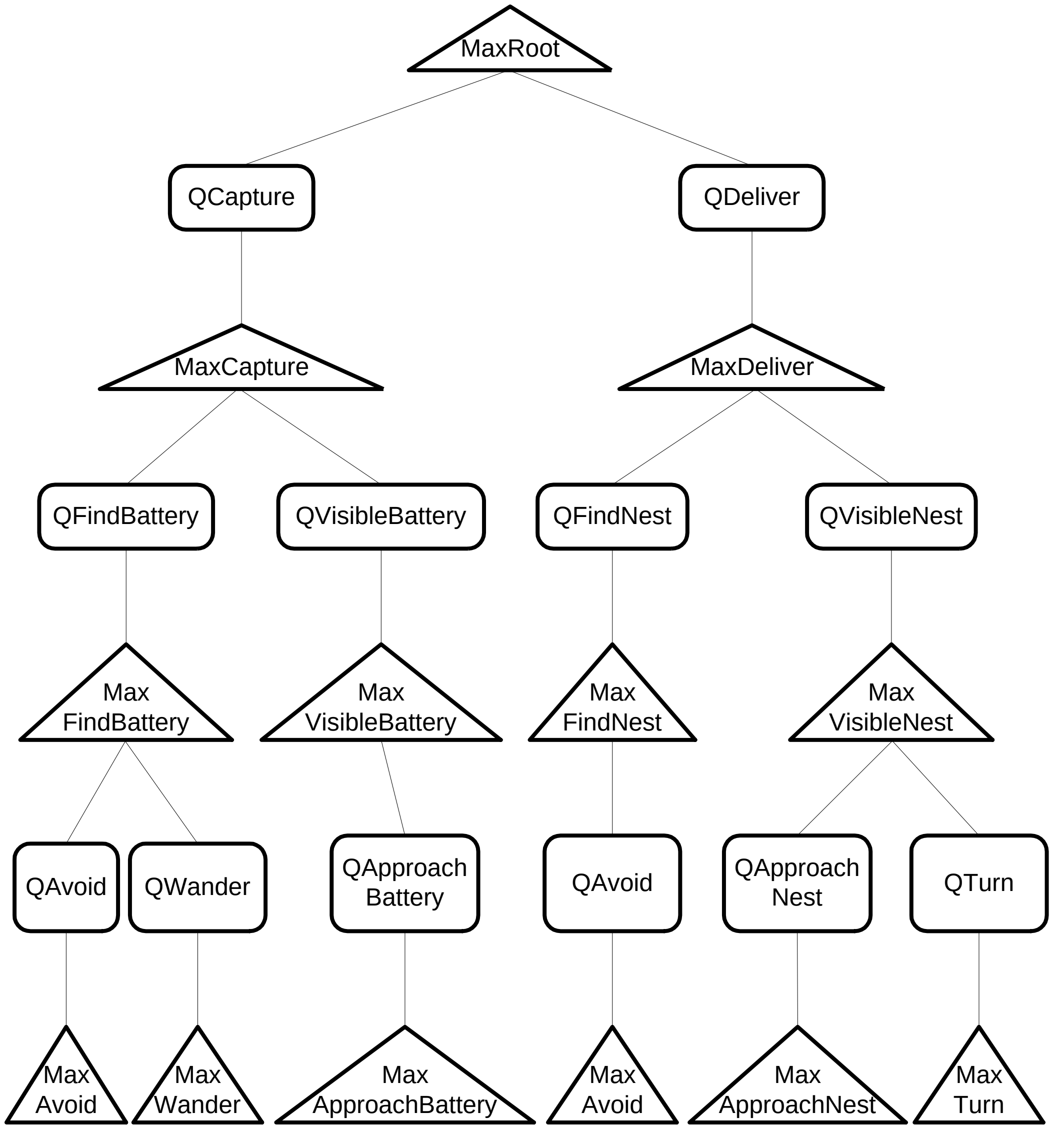


Figure 4.2. MAXQ graph of the foraging task, including all available subtasks

available composite subtasks with a description of the goals of the subtasks and in which types of states the subtasks are active, according to **Table 4.1**. The subtasks in the lowest abstraction layer, Find battery, Visible battery, Find nest and Visible nest, are active in one type of state each. The subtasks in the next abstraction layer, Capture and Deliver, are active in separate halves of the state space. The top abstraction layer, the Root node, is of course active in the entire state space. For all composite subtasks the pseudo-reward, $\tilde{R}_i$, is set to 0 for the goal states and -1 for all other states. The provided primitive subtasks are shown in **Table 4.3**. The

Subtask	**Goal**	**Active states**
Root	Solve the overall task	1,2,3,4
Capture	Find, approach and capture a battery	1,2
Deliver	Find the nest and deliver a battery	3,4
Find battery	Find a battery	1
Visible battery	Approach and capture a visible battery	2
Find nest	Find the nest	3
Visible nest	Deliver a battery to the visible nest	4

Table 4.2. Provided composite subtasks for the foraging task

Subtask	**Behavior**	**Active states**	**Input**
Avoid	Avoid hitting walls	1,2,3,4	Proximity sensors
Wander	Explore the environment	1,2,3,4	Proximity sensors
Approach battery	Approach and capture a battery	2	Angle to closest battery
Approach nest	Deliver the battery to the nest	4	Angle to the nest
Turn	Rotate to a battery or the nest	2,4	Angle to closest battery or the nest

Table 4.3. Provided primitive subtasks for the foraging task. The targets, battery packs and the nest, are identified by their colors: green for battery packs and red for the nest

primitive subtasks are simple reactive behaviors using one type of state input and 4 or 5 discrete actions to accomplish their tasks. The actions are pairs of right and left wheel velocities, in the range from -1300 mm/s to 1300 mm/s, as shown **Table 4.4**. The vision system is able to identify the targets, battery packs and the nest, by their color coding. The battery packs are green and the nest is red. The design of exact values of the wheel velocities of the actions, and also the reward functions, R_i, have been a trial and error process during the work with this thesis. The primitive subtasks shall function as good basic behaviors, using a few well designed actions. Also, there shall be a balance between the reward given to the different primitive subtasks, to achieve an efficient learning in the MAXQ hierarchies.

The reward functions, R_i, for the primitive subtasks are designed to promote the Cyber Rodent to move a long distance forward and keep a small angle to a target, i.e. the closest battery pack or the nest, in each time step. The reward given in each time step is in the interval $[-1, 0]$. The reward functions are linear functions of either movement, for Avoid and Wander, or the angle to a target, for Approach battery, Approach nest and Turn. Naturally the subtasks receive maximum negative reward, -1, if the subtasks fail, i.e. if the smallest proximity sensor reading is below the minimum effective range, 70 mm, for Avoid and Wander, or if the target is not

Subtask	**Actions**				
	1	**2**	**3**	**4**	**5**
Avoid	(250,150)	(150,250)	(300,-300)	(-300,300)	(-200,-200)
Wander	(600,600)	(600,400)	(400,600)	(600,100)	(100,600)
Approach battery	(450,450)	(675,225)	(225,675)	(900,0)	(0,900)
Approach nest	(450,450)	(675,225)	(225,675)	(900,0)	(0,900)
Turn	(-100,100)	(100,-100)	(-200,200)	(200,-200)	—

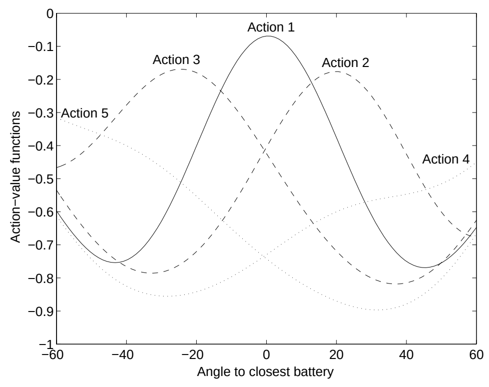
Table 4.4. Actions, pairs of right and left wheel velocities, for the primitive subtasks

visible anymore, for Approach battery, Approach nest and Turn. The reward functions used in the experiments are shown in **Equation 4.1** below.

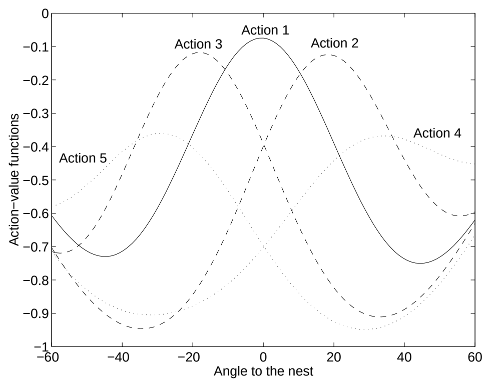
$$\begin{aligned} R_{avoid} &= \begin{cases} -0.5 & \text{action 1 or action 2} \\ -0.75 & \text{action 3 or action 4} \\ -0.875 & \text{action 5} \\ -1 & \text{smallest sensor reading} \leq 70 \text{ mm} \end{cases} \\ R_{wander} &= \begin{cases} -0.25 & \text{action 1} \\ -0.45 & \text{action 2 or action 3} \\ -0.67 & \text{action 4 or action 5} \\ -1 & \text{smallest sensor reading} \leq 70 \text{ mm} \end{cases} \\ R_{approach\ battery} &= \begin{cases} -\frac{|\alpha_b|}{60} & |\alpha_b| \leq 60^\circ \\ -1 & |\alpha_b| > 60^\circ, \text{ i.e. not visible} \end{cases} \\ R_{approach\ nest} &= \begin{cases} -\frac{|\alpha_n|}{60} & |\alpha_n| \leq 60^\circ \\ -1 & |\alpha_n| > 60^\circ, \text{ i.e. not visible} \end{cases} \\ R_{turn} &= \begin{cases} -0.25 - \frac{|\alpha_t|}{80} & |\alpha_t| \leq 60^\circ \\ -1 & |\alpha_t| > 60^\circ, \text{ i.e. not visible} \end{cases} \end{aligned} \tag{4.1}$$

Here the actions corresponds to **Table 4.4**, α_b is the angle to the closest battery, α_n is the angle to the nest and α_t is the angle to the target, either the closest battery pack or the nest, according to the current type of state, see **Table 4.1**.

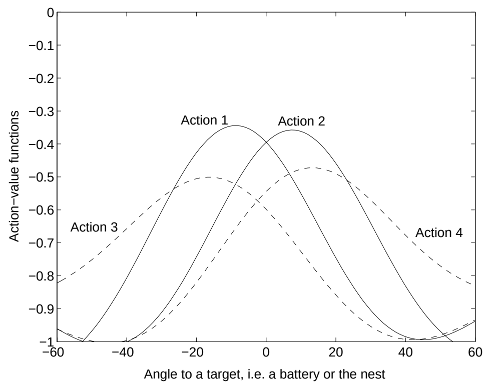
As presented in the previous chapter the primitive subtasks, functioning as basic behaviors, are already trained before the evolution starts. **Figure 4.3** shows the trained behaviors of Approach battery, Approach nest and Turn. The action-value functions approximated by a Gaussian RBF networks are plotted as functions of the input angle. The action-value functions increase as the angle decrease, because the immediate rewards increase as the angle decrease. The learned greedy policy of the trained behaviors is that the Cyber Rodent turns in opposite direction of the input angle and that the Cyber Rodent turns more as the angle increase.



(a) Approach battery



(b) Approach nest



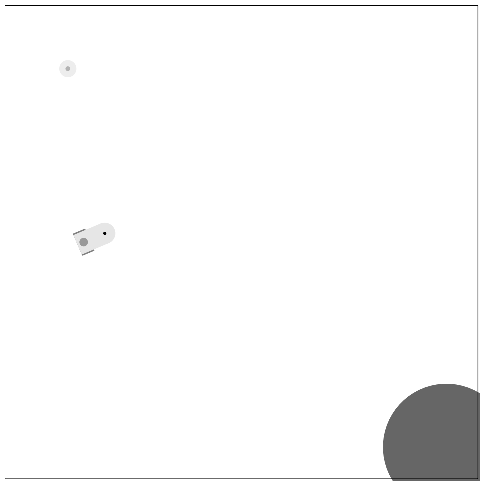
(c) Turn

Figure 4.3. The approximated action-value functions for the three primitive subtasks that use angle as state information. The figures show the already learned behaviors before the evolutionary process begins. The numbering of the actions are according to **Table 4.4**.

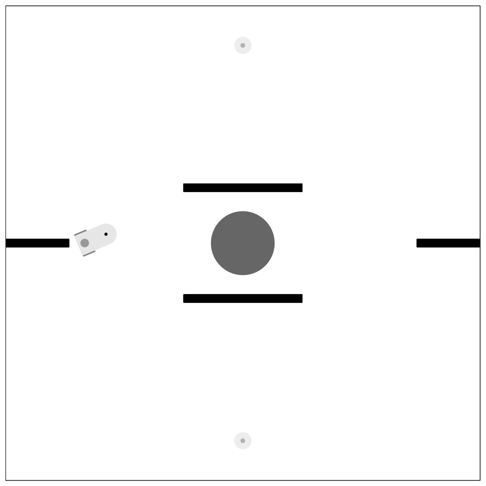
4.3 Environmental Settings

The environment for the Cyber Rodent is a 3-by-3 m play pen, containing wall obstacles, battery packs and a nest. In the experiments three different environmental settings are used, as seen as snapshots from the simulator in **Figure 4.4**. The small filled light gray circles are the battery packs and the larger filled dark gray circles are the nests.

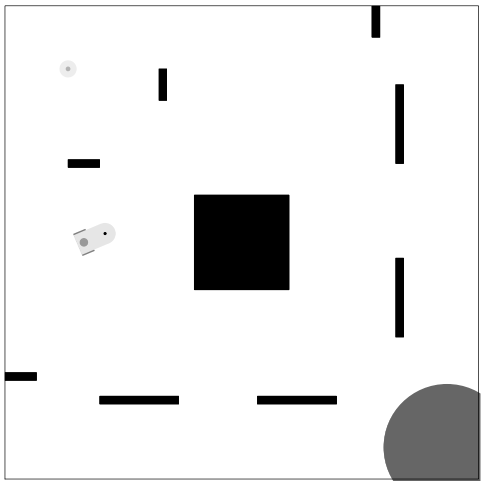
The hypothesis tested in the experiments in this thesis is that the genetic programming is able to obtain a good hierarchy that is adapted to the environmental setting and that the complexity of the environment is thereby reflected in the complexity of the obtained hierarchy. The first environment, **Figure 4.4(a)**, is very



(a) Environment 1



(b) Environment 2



(c) Environment 3

Figure 4.4. Experimental environmental settings

simple, containing no walls blocking the vision system. The expectation is that a good task hierarchy

- has a simple structure with only a few composite subtasks.
- needs only one primitive subtasks to accomplish the task in each type of state, according to **Table 4.1**.

In second environment, **Figure 4.4(b)**, it is more difficult to accomplish the deliver subtask than the capture subtask. This is due to that walls surround the nest and that the two battery packs are located in open spaces. The expectation is that for a good task hierarchy

- the subtree of the task hierarchy solving the deliver subtask is more complex, containing more composite and primitive subtasks, than the subtree of the task hierarchy solving the capture subtask.

The third environment, **Figure 4.4(c)**, contains a lot of obstacles making it relatively difficult to complete both the capture and deliver subtasks. The expectation is that a good task hierarchy

- contains all or almost all of the available subtask.

4.4 Learning and Evolution Settings

In the MAXQ framework a general approach to set the initial values and the parameters of learning is used. All subtasks have the same values, except for the decrease factor of the temperature, τ_{df}. For Avoid and Wander which are the behaviors most difficult to learn and therefore requiring a slower decrease of the temperature, τ_{df} is set to 0.9995. For the rest of the subtasks τ_{df} is set to 0.995. The common values of the learning parameters and initial values are summarized in **Table 4.5**, where θ_{init} is the initial value of the elements in the parameter vectors and the value, 0.123, is used for debugging purpose, according to [3].

Parameter	**Value**
α	0.2
γ	0.9
τ_{init}	2
τ_{limit}	0.05
ρ_{limit}	0.98
ϵ	0.02
θ_{init}	0.123
time-step	0.1 s

Table 4.5. Learning parameters and initial values common for all subtasks

The normalized Gaussian-RBFs, approximating the completion functions or the action-value functions of the subtasks, are placed equidistantly along each dimension of the input state space. The variance of the normalized Gaussian-RBFs are set to $\sqrt{2}$ multiplied by the distance between the centers of two adjacent RBFs. The input state space is normalized, according to the ranges of the input states, to handle differences in magnitude between the input states, i.e. to be able to use both angle and distance input to the RBF networks. The number of RBFs used in the normalized Gaussian RBF networks is shown for each subtask in **Table 4.6**.

Subtask	**Nr. of RBFs**
Root	128
Capture	64
Deliver	64
Find battery	32
Visible battery	320
Find nest	32
Visible nest	320
Avoid	32
Wander	32
Approach battery	11
Approach nest	11
Turn	7

Table 4.6. Number of RBFs in the normalized Gaussian RBF networks

Table 4.7 shows the GP parameters used in the experiments. The small population size, 16, is related to that the maximum depth of the hierarchies is only 4, limiting the number of hierarchies that can be constructed.

Parameter	**Number**
Trials/generation	16
Population size	16
Tournament size	2
Reproduced hierarchies	2
Maximum depth	4

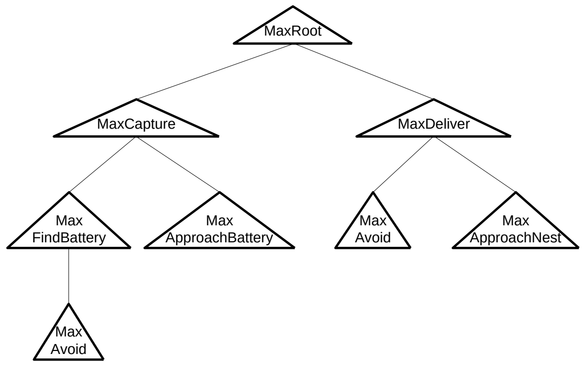
Table 4.7. GP parameters

4.5 Experimental Results

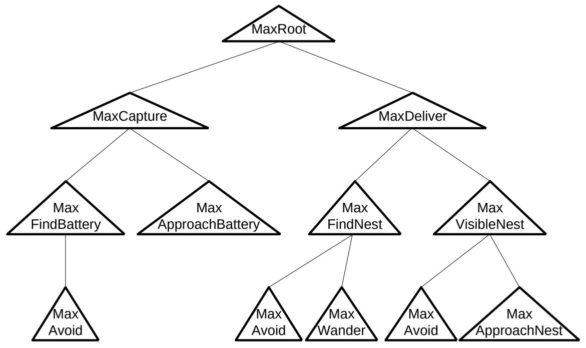
The results presented are for typical examples of evolved hierarchies for the three environments. GP did not find the same hierarchy in each simulation. The important thing though is that the obtained hierarchy, for an environment, represents the same behavior each time.

4.5.1 Evolved Hierarchies

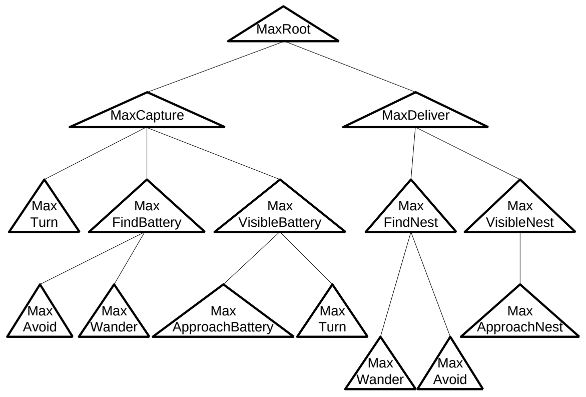
Figure 4.5 shows the hierarchies obtained by GP for the three environments, as MAXQ graphs (only Max nodes are shown). In accordance with the expectation



(a) Environment 1



(b) Environment 2



(c) Environment 3

Figure 4.5. Hierarchies obtained by GP

in **Section 4.3**, there is a strong connection between the complexity of the environments and the complexity of the obtained hierarchical structures. The general conclusion is that GP for a simple environment obtains a simple and specialized hierarchy and that GP for a complex environment obtains a complex and general task hierarchy, as can be seen from **Figure 4.5(a)** and **Figure 4.5(c)**, respectively. In environment 2, **Figure 4.4(b)**, the expectation is that the Cyber Rodent has greater difficulties to solve the deliver subtask than the capture subtask. This is also the conclusion that can be drawn from **Figure 4.5(b)**, based on that the obtained hierarchy has

- a simple structure for solving the capture subtask. The sub-hierarchy for the capture subtask actually has same structure as for environment 1.
- a complex structure for solving the deliver subtask. The sub-hierarchy for the deliver subtask is complete including all available composite subtasks and four primitive subtasks.

The hierarchy for environment 3, **figure 4.5(c)**, shows an artifact of the proposed method. The Turn primitive subtask, which is an action of the capture subtask, is an unnecessary and strange subtask. The reason it occurs in the hierarchy is that the proposed method, as mentioned earlier, has no pruning capability. This results in that the composite subtasks often have a number of actions, representing the same primitive subtasks.

4.5.2 Similarity in Structure

To picture how the evolutionary search process works, it is interesting to see how the difference in hierarchical structure is changing over time. A measurement of the difference between two hierarchies can be achieved by assigning different values to different abstraction layers: 4 for deliver and capture, 2 for the rest of the composite subtasks and 1 for all the primitive subtasks. When performing the structural difference measurement between two hierarchies, see **Figure 4.6**, negative points are received, according to the assigned values, for all composite subtasks that differ between the two hierarchies. Then, an extra negative point, equals the assigned value, is received for each primitive subtask that differs between the hierarchies for each composite subtask. By comparing the average difference for the hierarchies ranked 2-8 in each generation with the best performing hierarchy, the development of the structural changes can be shown.

For all performed simulations the structural differences have been changing according to the same pattern, see **Figure 4.7**. In the beginning of evolutionary search process the average difference is large. At some time approximately in the middle of the process, the difference rapidly becomes smaller. The difference then stays very small until GP terminates, i.e. when the difference becomes zero. This indicates that GP first searches to find what type of structure that is suitable. When a suitable type of structure is found, the GP only fine-tunes the structure, by adjusting the selection of primitive subtasks.

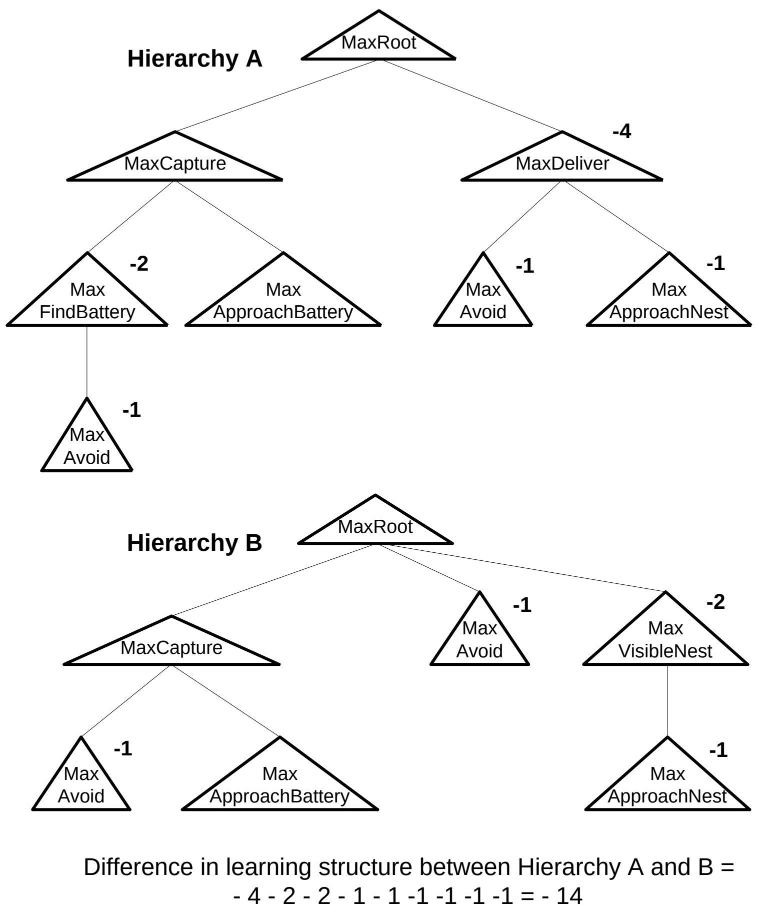


Figure 4.6. Measurement of structural differences between two hierarchies.

4.5.3 Performance Improvement and Intermediate Structures

The development of the performance over the generations has been approximately the same for all environments and simulations, as can be seen in **Figure 4.8** for the 8 best performing hierarchies in each generation in environment 2. In general, the performance improvement is large for the higher ranked hierarchies and small for the lower ranked hierarchies. For the best performing hierarchy the fitness value is approximately the same over the generations, meaning that the performance is not improved. The reason for the lack of improvement for the best performing hierarchy is that there is wide variety of hierarchies that can perform the task with a close to optimal result in the current environment. Many of these hierarchies, however, are not really stable solutions, which mean that the fitness values of these hierarchies vary a lot over the generations. The solution that GP obtains is therefore a solution that is stable over time.

Figure 4.9 shows the best performing hierarchy for 8 selected generations for

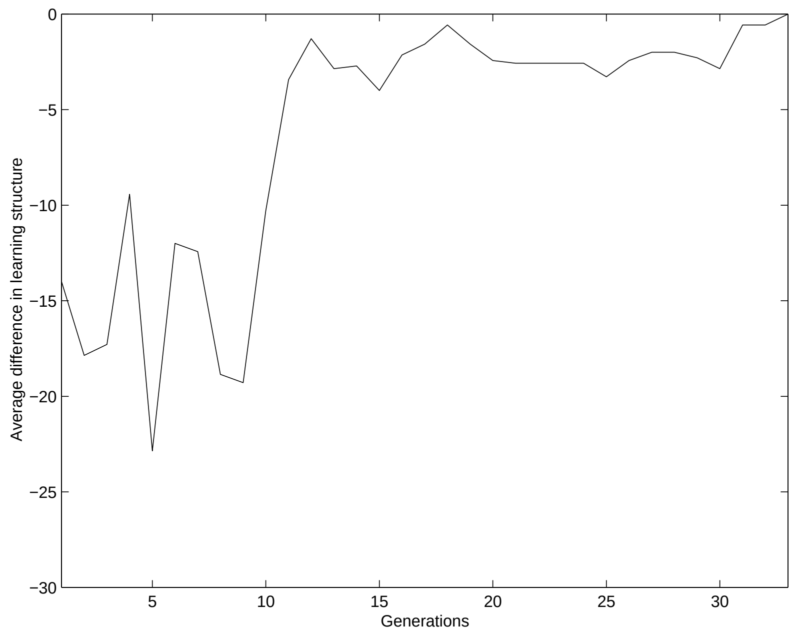


Figure 4.7. Average difference in hierarchical structure between the best performing hierarchy and the hierarchies ranked 2-8, measured for each generation in environment 1.

environment 2, i.e. the same simulation as for the performance development, see **Figure 4.8**. The best performing hierarchies in the intermediate generations confirm the conclusions from the analyzes of the similarity in structure and the performance development. In the early stages of the evolutionary process the structure of the best performing hierarchy is varying a lot, from very simple to very complex structures. Note that, even if the structures are very different the performance of the best performing hierarchy is stable, as seen in **Figure 4.8**. As early as in generation 10, GP has found a type of hierarchy that is stable. During the rest of the evolutionary process only the primitive subtasks are modified, as mentioned in the analyzes of the similarity in structure.

4.5.4 Similarity in Policy

It is important to analyze the learned policy of the obtained hierarchies to verify that they not only have the same hierarchical structure, but also the same behavior. Therefore, the average probability to select primitive subtasks was computed for the 8 best performing hierarchies in each generation, as an approximate of the actual policy. For the four types of states in **Table 4.1** 1000 states were sampled in the three environments. The average probability was then computed in each generation.

In the obtained hierarchies, there exist only two cases for each type of state:

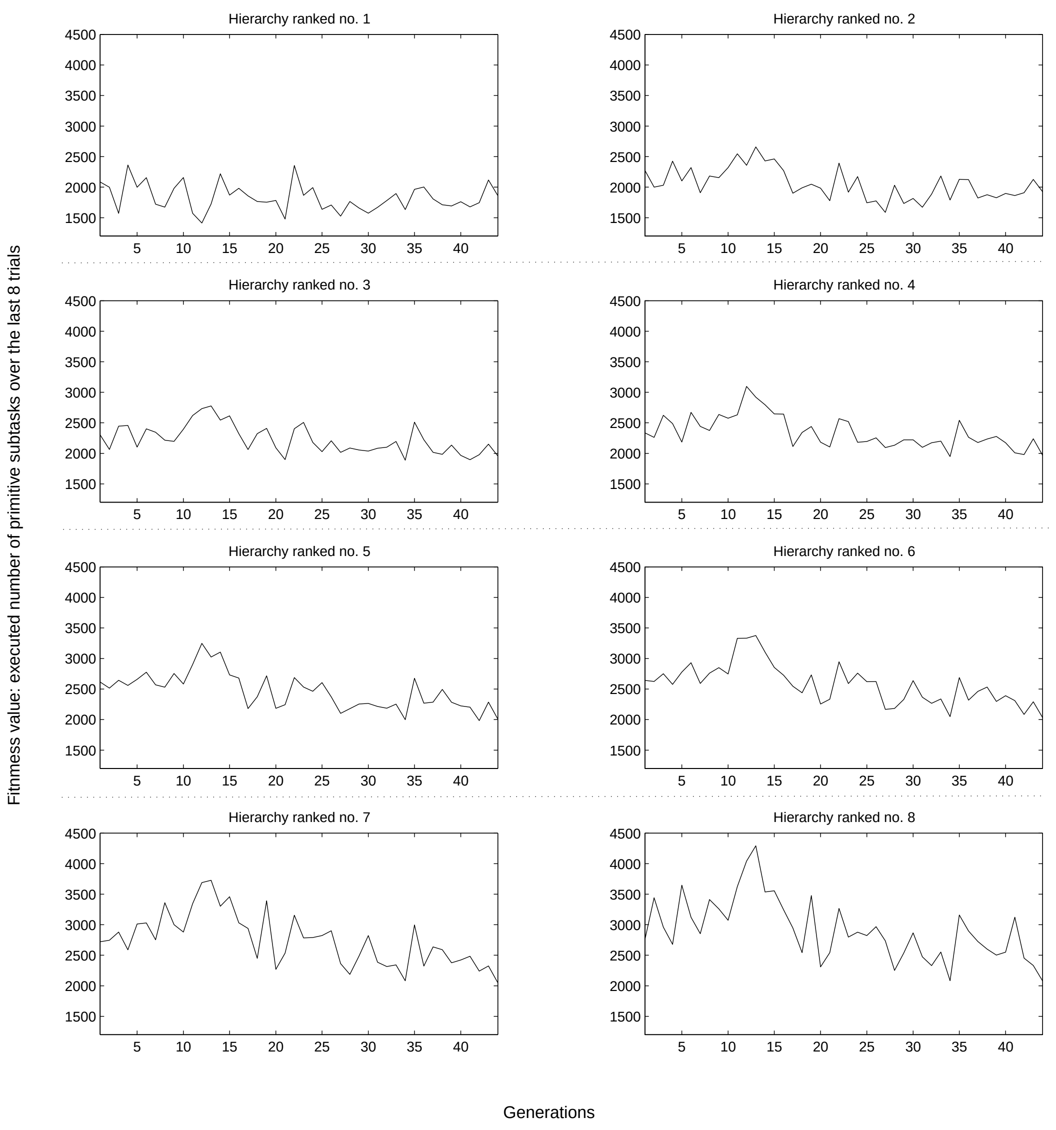
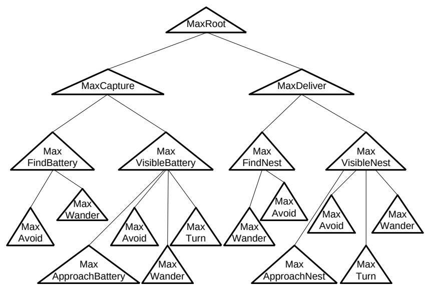


Figure 4.8. Performance development for the 8 best performing hierarchies in each generation. The graphs show the development for the obtained hierarchies in environment 2.

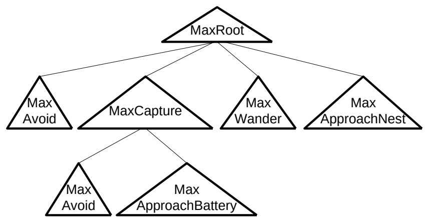
1. the hierarchy can only select one primitive subtask in the current type of state. A typical example of this case is seen in **Figure 4.10(a)** for environment 1 in state type 1. At some point the average probability for one primitive subtask becomes 1 and then stays at 1 until termination. In this case the policies of the 8 best performing hierarchies in the final population are identical, because the policies are completely determined by the hierarchical structure.

2. the hierarchy can select between two primitive subtasks in the current type of

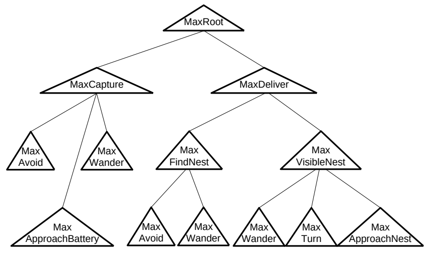
state. This case is more interesting, as seen in **Figure 4.10(b)**, for environment 3 in state type 3. In the beginning of the evolutionary search process when the difference in structure is large, the relation between the average selection probabilities is very unstable. During the later stages, when the difference in structure is smaller, the selection probabilities reaches a more stable relation. This gives indications that the hierarchies use approximately the same policy to solve the task. In this case the evolution process determines which subtasks can be selected and the learning process decides the policy, based on the available subtasks.



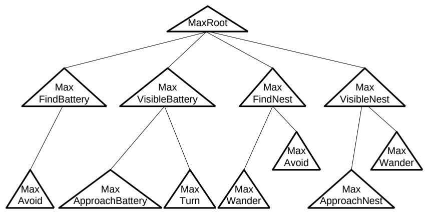
(a) Generation 1



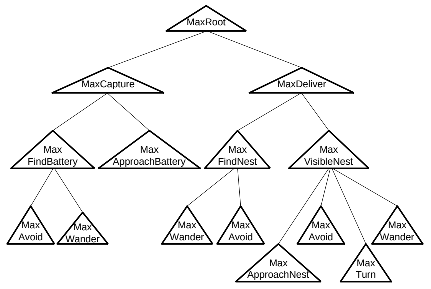
(b) Generation 2



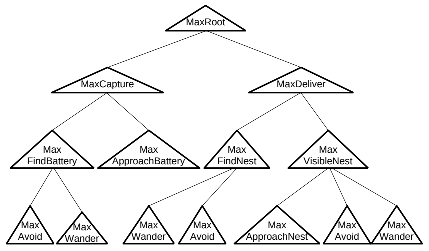
(c) Generation 4



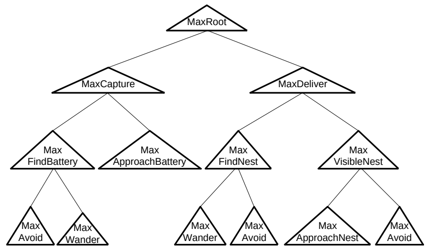
(d) Generation 6



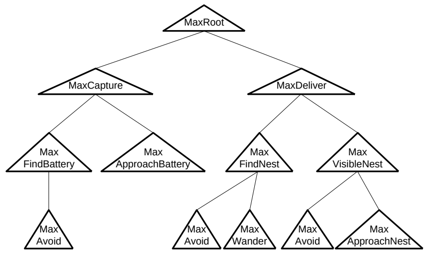
(e) Generation 10



(f) Generation 18

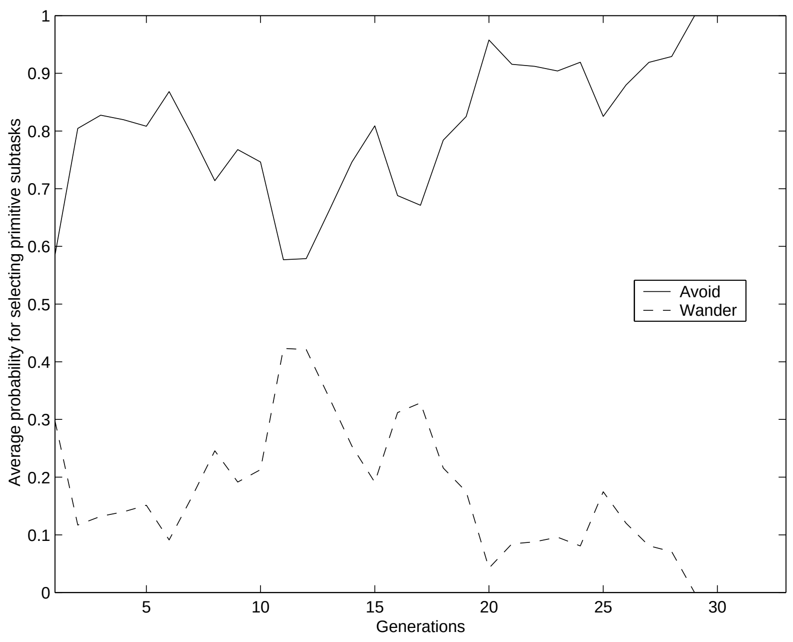


(g) Generation 23

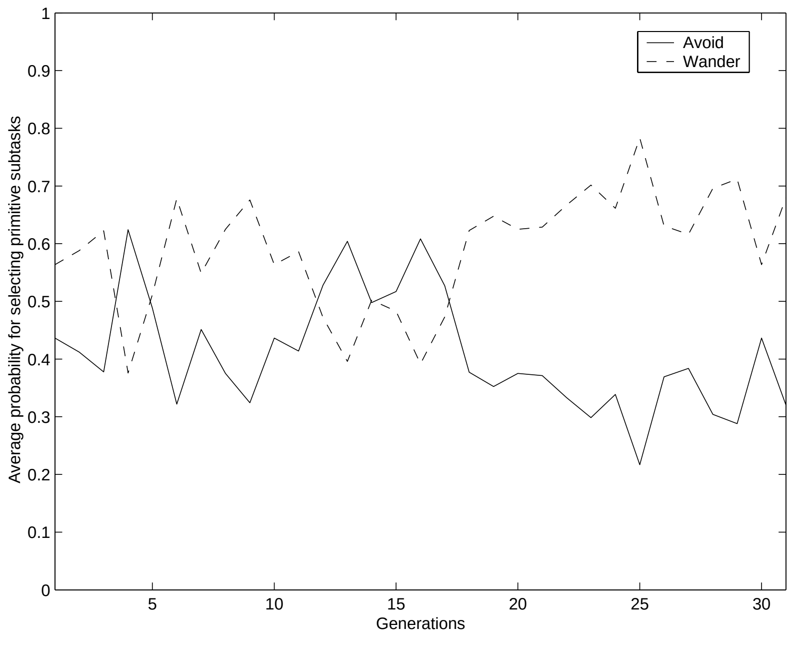


(h) Generation 36

Figure 4.9. The best performing hierarchies in 8 selected intermediate generations in environment 2.



(a) Example of case 1 for state type 1 and environment 1



(b) Example of case 2 for state type 3 and environment 3

Figure 4.10. Average probability for primitive subtask selection.

Chapter 5

Discussion

In this thesis a method for automatic construction of the task hierarchy in hierarchical RL has been presented. The method has proven successful in adapting the task hierarchy to three different environments for a simple foraging task, through experiments in a simulated environment. In an obtained hierarchy the complexity of the environment is reflected in the complexity of the learning structure of the hierarchy.

During the first stage of the evolutionary process there is a great differences in both structure and policy of the best performing hierarchies. At some time approximately at the middle of the evolutionary process these differences become smaller. After termination the learning structures are identical, according to the termination condition, and the policies are approximately the same, indicated by the policy analyzes.

The major disadvantage of the method is that it is very time consuming, taking several hours to perform a simulation for an environment. This is a general problem of RL and evolutionary computation, but it becomes an even more serious problem when the methods are combined. The time consuming calculations of the approximated completion functions and the approximated action value functions are also an important factor for the execution time.

As already mentioned the proposed method has no capabilities to add or remove the number of actions of the composite subtasks. It would be relatively easy to include pruning in the existing framework to be able to remove unnecessary actions. A further improvement would be to include the capability to add actions to the composite subtasks, maybe as a form of mutation operation.

Another improvement of the method would be to include memory capabilities to the learning. This could radically improve the performance of the learning tasks. At this stage the agent relies only on reactive behaviors to solve the tasks. For example, in the foraging task the Cyber Rodent has no knowledge of the position of the targets, battery packs and the nest. Therefore, the search for targets is almost a random search, where the Cyber Rodent moves without hitting walls until a target eventually becomes visible.

To investigate the properties of the proposed method future work would be to

- move the implementation from the simulator to the real hardware, the Cyber Rodent robot. There are plans to do this in the near future at ATR, which would be a good test of the applicability of the method.
- perform more extensive experiments to establish the robustness of the method.
- apply the proposed method to a large scale problem to examine the efficiency of the method.

Lamarckian evolution is used to represent that the strategies the agent uses are evolved during the agent's lifetime. This could be interpreted as there are multiple small agents competing inside the "brain" of the agent. The method for mixing and thereby create new strategies is the GP crossover operator. The crossover operator is a very rough mixing method and not neurophysiologically plausible. A long term extension of this work would be to include methods for mixing and creating hierarchies that are supported by neurophysiological evidence.

References

[1] D. Beasley, D. R. Bull and R. R. Martin. An Overview of Genetic Algorithms, Part 1&2. *University Computing*, 15(2), 58-69, 1993.

[2] T. Bäck and H. Schwefel. *Evolutionary Computation: An Overview.* IEEE Press, Piscataway NJ, 1996.

[3] T. G. Dietterich. Hierarchical Reinforcement Learning with the MAXQ Value unction Decomposition. *Journal of Artificial Intelligence Research,*13:227–303, 2000.

[4] K. L. Downing. Adaptive Genetic Programs via Reinforcement Learning. In *Proc. of the Genetic and Evolutionary Computation Conference*, pages 19–33, 2001.

[5] K. Doya, K. Samejima, K. Katagiri, and M. Kawato. Multiple Model-Based Reinforcement Learning. *Neural Computation*, 14:1347–1369, 2002.

[6] A. Eriksson. Evolution of Meta-parameters in Reinforcement Learning. Master's Thesis, NADA KTH, 2003

[7] G. Hornby, M. Fujita, S. Takamura, T. Yamamoto, and O. Hanagata. Autonomous Evolution of Gaits with the Sony Quadruped Robot. In *Proc. of the Genetic and Evolutionary Computation Conference*, pages 1297–1357, 1999.

[8] L. Kaebling, M. L. Littman and A. W. Moore. Reinforcement learning: A survey. *Journal of Artificial Intelligence Research*, 4:237-285, 1996.

[9] J. R. Koza. *Genetic Programming I : On the Programming of Computers by Means of Natural Selection.* MIT Press, 1992.

[10] M. C. Martin. Visual Obstacle Avoidance Using Genetic Programming: First Results. In *Proc. of the Genetic and Evolutionary Computation Conference*, pages 1107–1113, 2001.

[11] M. Mitchell. *An Introduction to Genetic Algorithms.* MIT Press, 1996.

[12] D. J. Montana. Strongly Typed Genetic Programming. *Evolutionary Computation*, 3(2):199 230, 1995.

[13] P. Nordin and W. Banzhaf. An On-Line Method to Evolve Behavior and to Control a Miniature Robot in Real Time with Genetic Programming. *Adaptive Behavior*, Vol.5, No. 2, pages, 107–140, 1997.

[14] R. Parr, and S. Russel. Reinforcement learning with hierarchies of machines. In *Advances in Neural Information Processing Systems 10*, pages 1043–1049, 1998.

[15] R. S. Sutton, and A. G. Barto. *Reinforcement Learning: An Introduction.* MIT Press/Bradford Books, March 1998.

[16] R. S. Sutton, D. Precup, and S. Singh. Between mdps and semi-mdps: A framework for temporal abstraction in reinforcement learning. *Artificial intelligence*, 112:181-211, 1999.

[17] G. J. Tesauro. Temporal difference learning and TD-Gammon. *Communications of the ACM*, 38:58–68, 1995.

[18] E. Uchibe, M. Nakamura, and M. Asada. Cooperative and Competitive Behavior Acquisition for Mobile Robots through Co-evolution. In *Proc. of the Genetic and Evolutionary Computation Conference*, pages 1406–1413, 1999.

[19] C. J. C. H. Watkins and P. Dayan. Q-learning. *Machine Learning*, 8:279–292, 1992.