Reinforcement Learning for a Dynamic Java™ Virtual Machine

Reinforcement Learning för en Dynamisk Java™ Virtuell Maskin

By
Eva Andreasson
01 October 2001 - 17 April 2002

NA-E-XXXX

A Master’s Thesis in Computer Science (20 credits) at the School of Computer Science and Engineering, Royal Institute of Technology
Supervisor at NADA was Frank Hoffmann
Examiner was Anders Lansner
Performed at Appeal Virtual Machines
Supervisor at the company was Olof Lindholm
Reinforcement Learning for a Dynamic Java™ Virtual Machine

May reinforcement learning enhance current garbage collection techniques? Theoretically machine learning could contribute to more adaptive solutions, but is such an approach feasible in practice? One key feature of human intelligence is the ability to learn from experience, but is it possible for technical system to learn through interaction with a real environment?

This project has researched an adaptive, automated decision process that takes decisions regarding which garbage collector technique should be used and how it should be used, based on information extracted from the currently active applications. The system learns through trial and error, which are the best actions to take in an initially unknown environment. In particular reinforcement learning has been used to decide under which circumstances to invoke the garbage collecting processing. Suggestions of when an adaptive decision process would be useful for a garbage collector is provided in this report as well as test result of a prototype designed for deciding when to garbage collect. The results show that the proposed approach achieves comparable performance with current heuristics used in garbage collection and even better performance in some cases. The conclusion of this project is that further development and research within this area is of highest interest.

Reinforcement Learning för en Dynamisk Java™ Virtuell Maskin

Kan reinforcement learning förbättra nuvarande skräpsamlingstekniker? Teoretiskt kan maskininlärning bidra till mer adaptiva lösningar, men är det realiserbart i praktiken? En viktig egenskap hos den mänskliga intelligensen är förmågan att lära genom att interagera med omgivningen, men är det möjligt för ett datasystem att lära sig genom interaktion med en verklig miljö?

Detta projekt har undersökt möjligheten att låta en adaptiv, lärande beslutsprocess fatta beslut kring vilka skräpsamlings-tekniker som skall användas och hur dessa skall användas, baserat på information extraherad från körande applikationer. Systemet lär sig genom ”trial and error” vilka handlingar som är de bästa att utföra i en från början okänd miljö. Reinforcement learning har använts för att bestämma under vilka omständigheter en skräpsamling bör ske. Förslag på när en adaptiv beslutsprocess skulle kunna vara användbar för en skräpsamlare är presenterade i denna rapport tillsammans med testresultat av en prototyp som designats att besluta när skräpsamling bör aktiveras. Resultaten visar att den föreslagna approachen uppnår jämförbar prestanda med nuvarande använda skräpsamlingsheuristiker, i vissa fall bättre prestanda. Slutsatsen av detta projekt är att vidare utveckling och forskning inom detta område är av högsta intresse.
# Table of Contents

1 **Survey** .................................................................................................................. 9  
   1.1 Introduction ........................................................................................................... 9  
      1.1.1 Experimental platform .................................................................................. 9  
      1.1.2 Terminology ................................................................................................... 9  
      1.1.3 Report Overview ............................................................................................ 10  
   1.2 Problem .................................................................................................................. 10  
      1.2.1 General Problem Definition .......................................................................... 10  
      1.2.2 Purpose .......................................................................................................... 10  
      1.2.3 Background .................................................................................................... 10  
      1.2.4 Concrete Problem Definition ......................................................................... 11  
      1.2.5 Related work .................................................................................................. 11  
      1.2.6 Assignment .................................................................................................... 11  
      1.2.7 Restrictions and Requirements ..................................................................... 11  
   1.3 Method .................................................................................................................... 12  
   1.4 Results .................................................................................................................... 12  

2 **Theoretical Part** ..................................................................................................... 13  
   2.1 About Part Two ..................................................................................................... 13  
   2.2 Machine Learning .................................................................................................. 13  
      2.2.1 Supervised Learning Methods ....................................................................... 13  
      2.2.2 Learning with a critic ..................................................................................... 14  
      2.2.3 Unsupervised Learning Methods .................................................................. 16  
   2.3 Garbage Collection ................................................................................................ 17  
      2.3.1 Different Techniques ..................................................................................... 18  
      2.3.2 Features ......................................................................................................... 19  
      2.3.3 The “Mostly-concurrent” Garbage Collector ................................................. 23  
   2.4 Conclusions ............................................................................................................ 24  

3 **Technical Part** ....................................................................................................... 25  
   3.1 About Part Three ................................................................................................... 25  
   3.2 Problem Specification ............................................................................................ 25  
      3.2.1 Concrete Problem Definition ......................................................................... 25  
      3.2.2 Technical Problem Definition ....................................................................... 25  
      3.2.3 Conclusions .................................................................................................... 26  
   3.3 Method .................................................................................................................... 26  
      3.3.1 Reinforcement Learning Methods .................................................................. 26  
      3.3.2 Exploration vs. Exploitation ........................................................................ 27  
      3.3.3 Continuous States and Actions ...................................................................... 28  
      3.3.4 Conclusions .................................................................................................... 30  
   3.4 Suggested System Design ....................................................................................... 30  
      3.4.1 General Model of the System ........................................................................ 31  
      3.4.2 Possible State Features .................................................................................. 31  
      3.4.3 Possible State Representation ........................................................................ 32  
      3.4.4 Possible Rewards ............................................................................................ 32  
      3.4.5 Possible Action Features ............................................................................... 35  
      3.4.6 Possible Action Representation ...................................................................... 36  
      3.4.7 Interesting Comparative Measurements ....................................................... 36  
   3.5 Adapting the System Design for the Concrete Problem ......................................... 36
3.5.1 Restrictions ................................................................. 37
3.5.2 State and Action Features of the Concrete Problem ............ 37
3.5.3 Adapting the SARSA Algorithm ..................................... 37
3.5.4 Problems During Development ..................................... 38
3.6 The Prototype .................................................................... 39
3.7 Results ............................................................................. 40
3.8 Discussion and Developments .......................................... 52
3.9 Conclusions ..................................................................... 53

4 Acknowledgments .................................................................. 55

5 References ............................................................................ 57
5.1 Literature .......................................................................... 57
5.2 Papers .............................................................................. 57
5.3 WWW ............................................................................... 57

Appendix A: Vocabulary .......................................................... 61
Table of Figures

**Figure 1** At the top an allocated list is shown. In the middle a memory leak is illustrated. At the bottom a memory leak and a dangling reference are illustrated...

**Figure 2** Mark-and-sweep collectors mark all reachable objects to be live and the remaining objects are assumed to be garbage. Stop-and-copy collectors divide the heap into two semi-spaces and copy all live objects from one semi-space to the other before sweeping the recently scanned semi-space...

**Figure 3** The generational garbage collector divides the heap into an older and a younger generation. During garbage collection of the younger generation all live objects are promoted to the older generation. When the older generation is full a complete garbage collection is invoked. In this case the old generation uses a compacting technique...

**Figure 4** Illustration of the “pig in the python”-problem. The pig symbolizes large, long-living objects. A huge bulge in an otherwise skinny environment, gradually moving down the path as the pig's snout approaches the python's nether regions. The picture is taken from [15] and some of the expressions from [24]...

**Figure 5** Coarse coding is illustrated to the upper left (1) and tile coding to the upper right (2). To the lower left (3) a radial basis function is presented and to the lower right (4) Kanerva coding is illustrated...

**Figure 6** A model of a reinforcement learning system. First the decision process observes the current state and reward. Then the decision process performs an action that affects the environment. Finally the environment returns a reward and the new state...

**Figure 7** A good situation with a high freeing rate and much memory left in the unallocated part of the heap is illustrated to the left (1). A worse situation is illustrated to the right (2), where there is little memory left in the unallocated heap and the garbage collection has a low freeing rate. This last situation may cause problems...

**Figure 8** To the upper right (2) half of the new allocated memory was successfully allocated in the fragmented heap. To the lower left (3) the same percent was successfully allocated in the fragmented heap although space for all new allocate objects exists in the fragmented area. To the lower right (4) all new allocated objects could be successfully allocated in the fragmented heap...

**Figure 9** Pseudo code modified to suit the concrete problem...

**Figure 10** To the left the interval performance of the RLS is compared to the interval performance of JRockit when running the application with short intervals. To the right the accumulated time performance is illustrated. The upper charts show the performances during the first 20 intervals and the lower chart shows the performances during 20 intervals after ca 50000 time steps...

**Figure 11** The upper chart shows the accumulated penalty for the RLS compared to the accumulated reward for JRockit when running the application with short intervals. The lower chart shows the average penalty as a function of time...

**Figure 12** To the left the interval performance of the RLS is compared to the interval performance of JRockit when running the application with long intervals. To the right the accumulated time performance is illustrated. The upper charts show the performances during the first 20 intervals and the lower charts show the performances during 20 intervals after ca 50000 time steps...
Figure 13  The upper chart shows the accumulated penalty for the RLS compared to the accumulated reward for JRockit when running the application with long intervals. The lower chart shows the average penalty as a function of time.  

Figure 14  To the left the interval performance of the RLS is compared to the interval performance of JRockit when running the application with randomly appearing intervals. To the right the accumulated time performance is illustrated. The upper charts show the performances during the first 20 intervals and the lower charts show the performances during 20 intervals after ca 50000 time steps.  

Figure 15  The upper chart illustrates the accumulated penalty for the RLS compared to JRockit during a test session with the application with randomly appearing intervals. The lower chart illustrates the average penalty as a function of time.  

Figure 16  The figure shows the development of the state-action value function, the Q-function, over time. The upper chart shows the Q-function after ca 2500 time steps. The middle chart shows the Q-function after ca 10000 time steps and the lower chart shows the Q-function after ca 50000 time steps and is then constant.  

Figure 17  The upper chart shows the accumulated penalty for JRockit compared to the accumulated penalty for the RLS using two state features when running the test application with randomly appearing intervals. The lower chart shows that the system still runs out of memory after ca 50000 time steps and hence has not learned all states that lead to running out of memory due to the increased amount of states and to the additional state feature not giving enough information.  

Figure 18  The figure shows a contour-plot of the Q-function at time step 2500, when the system has not yet run out of memory. To the upper left the plot for choosing action $a_0$ (not to garbage collect) is illustrated, while to the upper right the plot for choosing $a_1$ (to garbage collect) is illustrated. To the lower left the difference in Q-values is illustrated and to the lower right the states never visited are illustrated.  

Figure 19  The figure shows a contour-plot of the Q-function at time step 10000, when the system has started to occasionally run out of memory. To the upper left the plot for choosing action $a_0$ (not to garbage collect) is illustrated, while to the upper right the plot for choosing $a_1$ (to garbage collect) is illustrated. To the lower left the difference in Q-values is illustrated and to the lower right the states never visited are illustrated.  

Figure 20  The figure shows a contour-plot of the Q-function at time step 50000, when the system has stopped learning. To the upper left the plot for choosing action $a_0$ (not to garbage collect) is illustrated, while to the upper right the plot for choosing $a_1$ (to garbage collect) is illustrated. To the lower left the difference in Q-values is illustrated and to the lower right the states never visited are illustrated.  

Figure 21  The figure shows an enlarged contour-plot of the Q-function at time step 50000, to be able to see the detailed decision boundary when $s_1$ and $s_2 < 15\%$. To the upper left the plot for choosing action $a_0$ (not to garbage collect) is illustrated, while to the upper right the plot for choosing $a_1$ (to garbage collect) is illustrated. To the lower left the difference in Q-values is illustrated and to the lower right the states never visited are illustrated.
1 Survey

1.1 Introduction
This Master thesis investigates how machine learning methods might enhance current garbage collection in a Java™ Virtual Machine (JVM) and thereby contribute to more adaptive solutions.

The Master's project has been administrated and supervised by the Department of Numerical Analysis and Computer Science (NADA) at the Royal Institute of Technology (KTH), Stockholm. The work has been performed at Appeal Virtual Machines, Stockholm, starting the 1st of October 2001 and ending the 17th of April 2002. Supervisor at the company has been Olof Lindholm and supervisor at KTH has been Frank Hoffmann. The examiner at KTH was professor Anders Lansner.

This report presents the background to the assignment, a specification of the problem and achieved results. The results are based on comparisons between the performance of the proposed prototype and the performance of a conventional garbage collector the company uses in their JVM JRockit™. The results are restricted to one particular approach of reinforcement learning pursued in this thesis and do not include possible variations or improvements of reinforcement learning techniques.

1.1.1 Experimental platform
Equipment needed for carrying out this project is:

- A computer
- A JVM test environment
- Reinforcement learning software

The company provided the computer and the test environment. The software was implemented during the project and constitutes a part of the project deliverables.

1.1.2 Terminology
This Master project will be referred to as the project. The problem to solve in the project will be referred to as the problem. The problem is defined in later sections. The concrete problem is a specification of the problem and will be defined in the technical study of this report.

The words approach, method, algorithm and function are used for different meanings. Approach is a way of addressing a problem. A method is a general notion for a way of solving a special kind of problems, while an algorithm is a concrete, specified recipe for solving a specific problem. A function is specific code performing a well-defined task or computation.

Reinforcement learning uses a function to describe the expected future discounted rewards in a particular state or for a particular state action pair. This function is called the $Q$-function or the $Q$-value function in literature [4]. It will also be referred to as the state-action value function or the value function. The usage of the two latter terms depends on the specific meaning that is intended.

A vocabulary can be found at the end of this report (see Appendix A).
1.1.3 Report Overview

This report consists of three chapters. The first chapter is an introduction to the assignment and a summary of the project and its results. The results and the information provided in this chapter originate from the two subsequent chapters. Hence, justifications to the results and design decisions presented in this chapter are given in the other two. This choice has been made in order to provide a comprehensive introduction to this report directed to the casual reader. This chapter also constitutes a review of the project and resulting conclusions for the company.

Chapter 2 is concerned with the theoretical foundations required for formulating the problem and justifies the chosen approach. This chapter aims at readers with little or out-of-date knowledge of reinforcement learning or machine learning methods in general. It is also contains an introduction to automatic memory de-allocation, i.e. garbage collection. The chapter provides general information of different machine learning methods and discusses their suitability and applicability for the problem of dynamic garbage collection decision-making. It also provides information about different garbage collection techniques.

The third and final part of this report gives a complete survey of the technical solution process. It ends with results and a discussion about future developments. This part is directed to readers more familiar with machine learning methods and garbage collection or those that read the theoretical part. The third part also contains the system specification, namely the particular reinforcement learning algorithm and definitions of state and actions representations.

The second and third chapter may be read independently of each other. There are however references between the parts for the interested reader to follow. It is highly recommended though to read the theoretical part before the technical.

1.2 Problem

1.2.1 General Problem Definition

The problem to investigate is: how to design and implement an automatic and learning decision process for more dynamic garbage collection in a modern JVM.

1.2.2 Purpose

The purpose of addressing this problem is to contribute to the development of the Java™ software platform. The goal is to enhance the design of modern JVMs by means of learning techniques in order to make them more efficient, dynamic and flexible. A more adaptive JVM is desirable since it will result in improved performance and faster execution of applications based on Java™.

1.2.3 Background

A JVM render possible for Java byte code (the compiled code for Java applications) to be translated and executed on any platform. Another important function of the JVM is to handle the automatic memory management, i.e. the garbage collector. Depending on the application environment the garbage collector affects the performance of the JVM significantly.
The JVM JRockit™, by Appeal Virtual Machines, was designed recognizing that all applications are different and have different needs. Thus, a garbage collection technique and a garbage collection strategy that works well for one particular application may work poorly for another. To provide good performance across many applications, various garbage collection techniques with different characteristics have been implemented. However, a particular garbage collection technique can never achieve its optimal performance if one lacks a strategy of how and when to apply it. This observation motivates the investigation of better and more adaptive strategies.

1.2.4 Concrete Problem Definition

Due to the interest in finding more dynamic garbage collection strategies, the problem is restricted to focus on how to design and implement a learning decision process within the garbage collector of a modern JVM.

1.2.5 Related work

During the literature research for this project no other approach to utilize machine learning in a JVM has been found. Therefore, it is impossible to provide references to similar approaches for this particular problem. Many papers on garbage collection techniques include some sort of heuristics of when the technique should be applied, but these heuristics are usually quite simple as they are based on general rules that do not take the specific characteristics of the application into account.

Brecht et al [6] provide an analysis on when garbage collection should be invoked and when the heap should be expanded in the context of a Boehm-Demers-Weiser (BDW) collector. However, they do not introduce any adaptive learning but instead investigate the pros and cons of different heuristics.

1.2.6 Assignment

The objective of this project is to enhance the current garbage collection process in JRockit™. Instead of letting static variables decide which garbage collector technique to use and how to apply it, this project investigates an automatic, learning decision process that takes the decision while the application is running.

The assignment consists of an investigation whether and how an automatic, learning decision process can be designed to improve the current garbage collecting system in JRockit™ such that currently existing garbage collectors operate more dynamically and effectively.

The motivation for this assignment is the interest to find out if machine learning could contribute in some way to the improved performance of a commercial product. One of the desired features of JRockit™ is adaptive and dynamic behavior. Theoretically machine learning might contribute to more adaptive solutions, but the question remains whether such an approach is feasible in practice?

1.2.7 Restrictions and Requirements

The results of the investigation consist of a system specification and a module specification for implementation of an automated learning decision process for more dynamic garbage collection. The main focus regarding the results of this project is the identification of possible state features, possible actions and possible rewards.

A prototype for an automatic, learning decision process for more dynamic garbage collection will be implemented and analyzed. The prototype is only supposed to handle one decision in the context of one particular benchmark application.

The analysis will consist of a performance comparison between the prototype and the current system. The learning time and convergence behavior of the prototype will also be analyzed.
The investigation will mainly concern reinforcement learning for solving the problem. The suitability of other prominent machine learning methods, such as neural networks, decision trees and genetic algorithms will be briefly analyzed.

The investigation will concentrate on the concurrent single-generational garbage collector currently used in JRockit™.

1.3 Method
A reinforcement learning method called on-policy SARSA has been applied to solve the problem. In order to approximate the value function for continuous states, a gradient-descent function approximation has been explored. First a linear approximation was tried, but later during the development of the prototype a non-linear approximation function was explored as well.

This thesis only presents one possible approach to solve the problem. There are probably several other ways to utilize learning for the decision process, for example by using supervised machine learning methods. Also various characteristic of the approach taken in this thesis are based on ad-hoc decision rather than careful analysis and evaluation of possible design alternatives as discussed in the technical part.

Justifications for the choice of method and their applicability are presented in both the theoretical part and the technical part, but a short summary of the design choices is presented below.

SARSA was chosen because it is an on-policy temporal-difference method. On-policy evaluation, namely following and improving the behavior policy simultaneously, is desirable insofar as the system for solving the concrete problem needs to improve its performance during runtime. Tile coding has been chosen for extracting state representations of continuous state feature values. There are other possible approaches for achieving a proper function approximation but there has not been enough time during this project for investigating other approaches thoroughly.

1.4 Results
This report investigated how to design and implement an automatic and learning decision process for more dynamic garbage collection in a modern JVM. The preliminary results of this thesis can be summarized: It has been shown that a reinforcement learning system is able to learn to make correct decisions on when to perform garbage collection in a dynamic environment.

Since this is the first attempt to investigate the possibility of using machine learning for a more dynamic JVM, the research within this area is far from complete. There are still too many unexplored application areas and garbage collecting methods in order to be able to give a conclusive answer.

The conclusion of this project is that machine learning may be used to improve the performance of a JVM in certain application environments.
2 Theoretical Part

2.1 About Part Two

This part will discuss different methods that potentially can be used to solve the problem. It also contains a review of current garbage collection techniques.

First there will be an introduction to machine learning in general followed by more detailed descriptions of various machine learning methods. Each description of a machine learning method is followed by a discussion of benefits and disadvantages with respect to the suitability of that method for the problem at hand. Some of the most common machine learning methods are compared with respect to the garbage collection problem. A reservation has to be made, that there might be other methods better suited for solving the problem, which are not known to the author at this point of time.

This part also introduces garbage collection techniques and their characteristics and how these can be combined in different ways when designing a garbage collector. A section about problems to solve or consider when designing a garbage collector follows.

2.2 Machine Learning

One key feature of human intelligence is the ability to learn from experience. Humans and animals interact with their environment and adapt their behavior. Therefore a basic requirement of any artificial intelligent systems is the ability to learn – especially through interaction with the surrounding environment [3, 4, 21].

A well-known definition of machine learning is: “Any change in a system that allows it to perform better the second time on repetition of the same task or on another task drawn from the same population” (Herbert Simon 1983) [21].

Another proposal of a definition of machine learning is: “A method that learns within its domain, by searching domain specific concepts to reach more general concepts”. The generalization contributes to the ability to handle new concepts within the domain [21, 30].

Machine learning methods can be classified according to three different categories: supervised learning, learning with a critic and unsupervised learning.

2.2.1 Supervised Learning Methods

Supervised learning is learning from examples provided by a knowledgeable external expert. Therefore, a supervised learning method needs a set of training examples. It also needs a model that represents its knowledge about the domain that is updated during training. In the machine learning literature this model is also referred to as a hypothesis [30, 28, 23, 26].

Training patterns for supervised learning methods are composed of two parts, an input vector and an associated output. Training consists of presenting a set of inputs together with a set of desired responses as output. The method processes the input information and updates the model according to the error that is defined as the difference between the desired response and the actual output. These errors used to determine changes in the hypothesis of the method according to a learning rule [28, 23, 30].

The next two sections describe two representative examples of supervised learning methods: decision tree and neural networks. For details and further information on supervised learning methods see references [4, 30, 28, 23, 26].
**Decision Trees**

The hypothesis (explained in Section 2.2.1) in systems using decision trees consists of nodes forming a tree structure. The input set contains features that describe an object or a situation. The output consists of yes and no answers (or any other binary decision). Due to the binary nature of inputs and outputs decision trees form Boolean functions. The task of a decision tree is to decide to which class the object or situation belongs to according to its observable features [4, 18, 19].

To train a tree, known examples with known outcomes are needed to learn which features are associated with which class [4, 18].

**Neural Networks**

Neural networks consist of a set of computational units, connected via weighted links. The hypothesis is represented by the weights, which strengths are adapted during training. The network-units operate in a distributed and parallel fashion [4, 25, 30].

The hypothesis is represented by the current values of the weights in the network. An input is presented to the network and the difference between the desired output and the actual network output is observed. By making small adjustments to the weights, the network output becomes more similar to the training data. The goal of these adjustments is to minimize the summed squared error over the training set [3, 25, 27, 30].

**Benefits and Disadvantages**

Supervised learning methods are very efficient when the desired behavior is known in form of input-output training examples. If the set of training examples is large enough and representative for the domain the networks can be trained efficiently and are able to successfully generalize correctly to previously unseen examples [3, 4].

If training examples are difficult or costly to obtain or not available at all supervised learning methods cannot be applied [10]. Still it would be possible for a supervised learner to imitate the behavior of an existing garbage collector, but this will not result in any improvement of its performance. Therefore, the primary goal of this project to optimize the decision process cannot be achieved with a supervised learning approach.

Often it is necessary for the system to learn online, in case training examples become available as the system is running, rather than in batch mode in which case the entire data set is available prior to training. Backpropagation is an example of a learning method that in principle is capable of online learning, whereas other supervised methods such as decision trees can only be trained in batch mode [3, 4].

**2.2.2 Learning with a critic**

Learning with a critic means that no explicit examples of correct input output pairs are needed for training, but merely that a “critic” tells the system whether it performs well or poorly.

A “learn with a critic”-system uses “trial and error”-search to learn the best action to take in a given situation. This is realized through a reward system constituting the critic. The objective is to choose those actions that maximize the future rewards. The rewards for actions are not necessarily immediate but might be delayed. Therefore, the system has to address the temporal credit assignment problem, namely to identify those states and actions that in the long run will result in optimal rewards [5, 9, 29].

In contrast to the earlier described supervised learning methods which learn based on the error, learning with a critic involves interacting with an initially unknown environment and observing the consequences of the actions [34].
The following sections describe two examples of methods that learn with a critic: genetic algorithms and reinforcement learning methods. For a more detailed survey of “learning with a critic” approaches see references [4, 5, 29, 30, 34].

**Genetic Algorithms**

Genetic algorithms are search and optimization methods that mimic the processes that occur in natural evolution. They operate with a population of candidate solutions to which random modifications are applied. Individuals are represented as bit strings, which encode parameters of a possible solution. By selecting better individuals for reproduction to the next generation the quality of the individuals in the population improves over time [3, 26]. Although based on the same principle as genetic algorithms other evolutionary algorithms employ different representations and genetic operator. In the case of genetic algorithms the fitness function plays the role of the critic. Individuals of the same generation are evaluated according to the fitness function. The best-suited individuals of a generation are selected to generate offspring to the next generation.

**Benefits and Disadvantages**

Genetic algorithms are usually slow and require a large number of fitness evaluations. They only indirectly use the information provided by the critic to update their behavior. If the learning takes place in the real environment, poorly adapted individuals might significantly deteriorate the overall performance of the system for unacceptable long periods of time. The fitness function only considers the accumulated reward over time, but does not relate the reward to particular states and actions. The genetic algorithm maintains no explicit model of states and therefore information available for direct learning of good actions and states cannot be utilized. In the type of decision problems relevant for this project, genetic algorithms learn much slower than for example reinforcement learning algorithms presented in the next section [1, 5, 37].

**Reinforcement Learning Methods**

Reinforcement learning methods solve a class of problems known as Markov Decision Processes (MDP) or reinforcement problems. If it is possible to formulate the problem at hand as an MDP, reinforcement learning provides a suitable approach to its solution [1, 4, 5].

A reinforcement learner observes a state (situation) and decides what action to take in that particular situation. The choice of action depends on a state-action value function, Q(s, a) that calculates the value of taking an action a in state s. The q-value reflects the expected future discounted rewards of taking action a in state s and following an optimal policy afterwards. The action chosen is the one with the highest Q-value within the current state. As a result of the action taken by the reinforcement learner the environment transitions to a new state provides a reward value as feedback. Based on the observed reward and the state-action value of the new state the reinforcement learning method updates its beliefs about the state-action value of the previous situation. The reward function constitutes the critic [3, 4, 22, 17].
More formally, a policy is a mapping from states to actions $\pi : S \times \Lambda \rightarrow [0,1]$, in which $\pi(s, a)$ denotes the probability with which the reinforcement system chooses action $a$ in state $s$. As a result of the action taken by the agent in the previous state, the environment transitions to a new state $s_{t+1}$. Depending on the new state and the previous action the environment might pay a reward to the agent. The scalar reward signal indicates how well the agent is doing with respect to the task at hand. However, reward for desirable actions might be delayed, leaving the agent with the temporal credit assignment problem [9], of figuring out which actions lead to desirable states of high rewards. The objective for the agent is to choose those actions that maximize the sum of future discounted rewards:

$$R = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} \ldots$$

The discount factor $\gamma \in [0,1]$ favors immediate rewards over equally large payoffs to be obtained in the future, similar to the notion of an interest rate in economics [1, 3, 5].

Notice, that usually neither the state transition nor the reward function are known to the reinforcement system, neither do these functions need to be deterministic. In the general case the system behavior is determined by the transition probabilities $P(s_{t+1} | s_t, a_t)$ for ending up in state $s_{t+1}$ if the agent takes action $a$ in state $s$ and the reward probabilities $P(r | s_t, a_t)$ for obtaining reward $r$ for the state action pair $s_t, a_t$ [1, 3, 5].

Whereas for instance dynamic programming requires a model of the environment for computing the optimal actions, reinforcement learning methods are model free and the reinforcement system obtain knowledge about the environment through interaction. The agent explores the environment in a trial and error fashion, observing the rewards obtained of taking various actions in different states. Based on this information the agent updates its beliefs about the environment and refines its policy that decides what action to take next [3, 5].

**Benefits and Disadvantages**

To maximize the reward over time, a learning system must choose the most valuable action. The problem is that the best action may be an action not yet tried and evaluated. Finding a balance between making decisions on experience by choosing the best evaluated action so far and finding new alternatives that might be better than the known ones, is a difficult problem when designing and using reinforcement learning systems. The “exploration vs. exploitation”-issue is discussed in the technical study of this report [3, 5].

Another important issue to consider is the choice of reward function, since it affects the behavior of the system. The proper definition of the reward function therefore plays an important role in the design of reinforcement learning systems [4, 5, 22].

### 2.2.3 Unsupervised Learning Methods

In contrast to supervised learning and learning with a critic, which is applicable only when the outcome is known or if information is available about what constitutes good or bad behavior, unsupervised learning needs no hint at all of what the correct outcome should be [8, 30, 34]. Instead they cluster the input data according to the similarity of features and thereby identify the underlying structure of the input domain. Often unsupervised learning methods are used to preprocess the data before a supervised learning algorithm is applied.

These kinds of methods are not of interest when solving the problem since there is a need of control in a JVM system. For instance the system should never run out of memory, or at least learn quickly not to run out of memory, hence a system handling the problem must be controlled in some way.
2.3 Garbage Collection

Some programming languages use explicit memory allocation and deallocation, for instance C and C++. This demands that programmers using such languages have a lot of knowledge of how a computer is built and how it works. If the programmer would lack this knowledge when constructing a computer program it could result in a computer program with memory leaks and dangling references [2, 36].

Memory leaks are memory that is referenced by deallocated memory, illustrated in Figure 1.2. A dangling reference is a reference to memory that has been deallocated, illustrated in Figure 1.3. These problems cause the computer program to eventually crash, or even worse, to keep running but calculating wrong values [2, 36].

![Figure 1](image-url)  
At the top an allocated list is shown. In the middle a memory leak is illustrated. At the bottom a memory leak and a dangling reference are illustrated.

To simplify for programmers, program languages were developed that did not use explicit memory allocation. The first high-level, compiler-using language was Fortran (1957). Other programming languages based on the same idea developed later, for instance Lisp, Small Talk and Java [2].

Implicit memory allocating languages need a system that handles the freeing of objects that are no longer used by the running program. A system that handles this is called a garbage collector, since it takes care of garbage caused by the running program [2].

One purpose of garbage collection is: “To relieve the programmer from the burden of discovering memory management errors by ensuring that they cannot arise” [2]. In other words: “To free the programmer from having to keep track of when to free allocated memory, thereby preventing many potential bugs and headaches” [36].
2.3.1 Different Techniques

One difficulty with garbage collecting is to decide which objects are no longer alive (dead). An object is dead if no references to that object exist. If there still are references to an object it is said to be live. For instance, an object-oriented program uses the stack and registers for storing class variables (among other things). The objects that the running program stores are certain to be live. Objects known to be live are called roots. By following the references from the roots all other live objects can be found.

Another difficulty is to prevent heap fragmentation. That is, preventing the free memory spaces of the heap of becoming to small and to scattered so that new objects cannot be allocated, although the total amount of free memory may be sufficient [36].

There are mainly two basic different techniques that a garbage collector can rely on: reference counting collectors and tracing collectors. The following sections discuss these techniques. For further reading about garbage collection see references [2, 6, 7, 8, 13, 20, 36].

Reference Counting Collectors

Reference counting collectors perform the garbage collection by counting and storing the amount of references to an object. When an object is initiated the reference number is set to one. For each new change in the amount of references to an object, the reference count is increased or decreased. If the amount of references to an object becomes zero, the object is freed and all the objects that the garbage collected object refers to have their reference counts decreased. These decreases may, in turn, lead to garbage collection of other objects [2, 20, 36].

The advantage with this method is that there is no need for scanning the heap for live objects. On the other hand this approach has difficulties handling cyclic structures and the updating of references must be synchronous.

Tracing Collectors

A typical tracing collector is the mark-and-sweep collector presented in Figure 2. It uses a technique that consists of finding all from the running program reachable objects, i.e. all live objects. When a live object is found it is marked. The marking can be performed in several ways, for example by setting a certain bit in the header of the object. When the collector has found all live objects in the heap and marked them it is time for the next step, which is sweeping all unmarked objects away – freeing their memory [2, 6, 20, 36].

A problem using mark-and-sweep is that it causes fragmentation of the heap. That is, scattered memory pieces that cannot satisfy a certain memory need, although the free memory in the heap in total would. To deal with the problem of a fragmented heap, a compacting mark-and-sweep has been developed. This type of collector is called mark-and-compact. During sweeping, the mark-and-compact collector tries to move all remaining, live objects towards one end of the heap to get rid of small free memory spaces that causes fragmentation [2, 12, 20, 36].
Another kind of tracing collector is the copying garbage collector, stop-and-copy, illustrated in Figure 2. Copying collectors divide the heap into two semi-spaces, X and Y. The scanning for garbage begins e.g. in X. If an object is alive, that is, can be reached from the program, the garbage collector will copy it into Y and then start scanning for the next live object in X. When the garbage collector is through scanning X, it may start scanning Y for live objects, copying them into X. It follows that objects may be allocated in only one semi-space at a time. To change from scanning X to scanning Y is called to flip [2, 20, 36].

The mark-and-compact technique is in many situations more time consuming than stop-and-copy. However, when it comes to large objects or few non-living objects in the heap the stop-and-copy collector is a lot worse than the mark-and-compact collector, since copying takes a lot of effort in those situations. Another disadvantage using stop-and-copy is that the heap uses only half its capacity, since it has to be divided into two equal semi-spaces and use only one of them at a time for allocation [2, 20, 36].

2.3.2 Features
The above described, different techniques can be varied in many ways. Unfortunately there is no “best solution”. One solution works better for certain application areas and others work better under different circumstances. The problem is that existing applications, using JVMs and accordingly garbage collectors, are so different that it is hard to design and implement a garbage collector that works perfectly in all situations [2, 20, 36].

There are some features that have been developed for fulfilling different needs in garbage collecting that can be varied to some extent. Below a briefing of some important features is given as well as the effects a change in these features will have on the performance of the garbage collector.
Handle Based or Direct Pointers

From the name it is understood that direct pointers are pointers that point directly at the objects. A reference to the object contains the explicit address of the object.

Handle based pointers, on the other hand, are pointers that point at a table handle. The contents of the table space of that handle are a reference to an object. Handle based pointers point in other words indirectly at the object.

The use of this kind of look-up table remarkably simplifies the updating of object pointers. It is easier since only the table needs updating, not the pointers to the objects. One problem is that the table uses much more memory than direct pointers. Another is that it takes more time to run the program since using the look-up table increases the time for locating an object.

Identification of Pointers

Pointers need exact identification if objects are relocated, as in the case of copying. This approach is thus called exact [20].

When objects are not moved, pointers do not need exact identification. A non-exact approach can be used, the so-called conservative approach. All live objects are found anyway, along with a few non-living objects. This approach is important, since it allows programs that were written without garbage collection in mind to use garbage collection anyway [20].

Moving Objects

Small objects are easy to move and when moving them closer together in the heap fragmentation is prevented. By moving objects, the cache locality is also improved, which means referring objects are situated closer to each other [2].

Disadvantages appear when the moving concerns large objects. To move large objects is very ineffective since the process of moving them affects the total garbage collecting process time considerably. Another disadvantage with moving large objects is that all objects pointing at an object to be moved need to be found. In the worst case this means a scan of the entire heap [2].

In systems where copying collectors or compacting collectors are used objects may be moved [2].

Generational Garbage Collection

Most objects are considered to die young. The solution of not having to continue scanning long-living objects is to divide the heap into generations. Old objects are stored in a certain part of the heap and young in another. A generational collector is illustrated in Figure 3.
The generational garbage collector divides the heap into an older and a younger generation. During garbage collection of the younger generation all live objects are promoted to the older generation. When the older generation is full a complete garbage collection is invoked. In this case the old generation uses a compacting technique.

The part, where the young objects are stored, is small and hence garbage collected more frequently, while the region of the heap, where older objects are stored, is garbage collected more seldom. Objects that survive a certain number of garbage collections in a younger generation are promoted to an older generation. This approach enhances the interruption time of the running program remarkably and the garbage collection in total [2, 13, 20].

To be able to garbage collect a younger generation without collecting older generations as well, all objects in older generations are considered to be alive. Another important issue is to keep track of which old objects that are pointing at younger objects, so that the referenced younger, live objects will not be garbage collected.

One issue to consider, when it comes to generational garbage collection, is how fast an object ages, i.e. is promoted to the next generation. The promotion rate has to be decided. A low rate makes the garbage collection sessions faster, but may also cause promotion of comparatively young objects and accordingly a lot of garbage in older generations, which is undesirable. A high promotion rate gives more stable old generations, but also longer breaks for collecting the youngest generation. The trade-off problem with the promotion rate is often called the “pig in the python”-problem illustrated in Figure 4. Collection effort will be wasted as a large and long-living object survives and is promoted from generation to generation. The similarity with the “pig in the python” is the immobilization of the snake as it digests a much too large prey – the pig [2, 13, 20, 24].
There is no obligation for the different generations to use the same garbage collection technique. By using different techniques to garbage collect different parts of the heap, process time may be shortened and other desired goals may be achieved. These goals are discussed further on in this chapter [2, 13, 20].

**Incremental Collection**

An incremental collector divides the heap into sections and collects one section at a time. One consequence of this is that only a small amount of the garbage – the garbage of one section of the heap – is collected at a time and that it may not be enough to satisfy the allocation needs of the program. A resulting positive feature is that an incremental garbage collection does not cause such a large break in the running program as a complete garbage collection of the heap might do. Why this technique is seldom used is because it is very hard to implement [2, 13, 20].

**Concurrent Collection**

Another effective, but also hard implemented garbage collector technique is the concurrent approach. A concurrent garbage collector works in a certain thread by itself, at the same time as the program. To work “at the same time as the program” means that the program and the collector take turns executing instructions.

Both the incremental and the concurrent collectors collect little garbage at a time. The difference between the two approaches is that incremental “little at a time”-approach means little garbage is collected at a time, where little refers to a small area of the heap. Concurrent “little at a time”-approach, on the other hand, means little garbage collection at a time, that is, the garbage collection is divided into steps and only one step at a time is performed. In other words little, but not necessarily complete, garbage collection is performed at a time. Garbage collection steps of a mostly-concurrent garbage collector are described in Section 2.3.3. Consequently concurrent collectors need to consider allocations made by the program in between the step executions of the collector. Another important issue is to keep track of the changes made by the running program in order to be able to update all pointers correctly [2, 13, 20].

This technique is hard to implement, but is very effective according to total interruption time of the running program. The alternative is to stop the program and complete the garbage collection and then return to the program, which would cause a much more noticeable interruption [2, 13, 20].

**Parallel Collection**

The parallel collection technique may be used when the system where the collector is being used has more than one processor. Only in this case would it be possible for several threads to really work at the same time, i.e. in parallel [7, 8].

Advantages with this technique are that the garbage collector may work concurrently and incrementally on each processor and thereby shorten the total time of the garbage collection, i.e. shorten the interruption time in the running program [7, 8].
Important to consider when it comes to parallel garbage collection is the need of synchronization of the garbage collecting threads. It is also important to distribute the work to the separate processors in an efficient and fair way [7, 8].

2.3.3 The “Mostly-concurrent” Garbage Collector

JRockit™ has a “mostly-concurrent” garbage collector that is based on five steps. The first step includes stopping the running program and finding all objects directly reachable from the roots [8, 13].

After the first step, the running program is allowed to run again, while the garbage collector marks all reachable objects from the found roots. At the same time the garbage collector keeps track of all changes made by the running program during this concurrent phase. The changed objects are marked dirty, which means that those objects must be checked again before sweeping [8, 13].

The third step contains pre-cleaning. Pre-cleaning is concurrently checking dirty objects and also keeping track of new changes. Hopefully the checking of dirty objects will take less time than it will take for the running program to allocate many new objects (change the heap). The purpose of pre-cleaning is to remove some work pressure from step four, which causes a second stop of the running program [8, 13].

Step four is the final marking pause and includes checking all remaining, dirty objects as well as the roots once again. If any live object is found, it is marked as the earlier found living objects [8, 13].

The fifth and last step is the sweeping phase. In the sweeping phase all non-marked objects are freed and returned to the free-list. The free-list is a linked list of free memory sections in the heap [8, 13].

2.3.4 Optimization Through Minimization

The following sections describe desired goals regarding the performance of a garbage collector.

Each of the earlier described techniques and features can be combined and varied in many ways to accomplish these goals in various environments. A major challenge for programmers in this area today is to design and implement a garbage collector that is able to achieve the goals in a very dynamic and maybe unknown environment [6, 13].

Memory Blocking

The garbage collector has to make sure that the running program never runs out of memory. The goal is to free enough memory and to compact the blocked memory in order to satisfy the allocation needs of the running program. The desire is to keep the memory blocking as low and as compact as possible [2, 6, 13, 20].

Breaks

A major issue is to have as few and as short interruptions (breaks) as possible in the running program. A break is when the program running is stopped completely [1, 13, 6, 20].

Total Process Time

In a broader perspective the total occupied process time is a factor for minimization, just as the other factors described above. Total process time does not need to be an issue in the case with a parallel garbage collector if the throughput is satisfying enough [1, 13, 6, 20].
2.4 Conclusions

The reinforcement learning methods are able to learn from interaction with the environment and time-delayed feedback. As it would be difficult to if not impossible to obtain direct examples of the “best possible garbage collection decisions” supervised learning methods are not suitable for the optimization problem at hand. Since the objective of the envisioned solution is to optimize the garbage collecting process based on the observed memory states and performance during runtime, reinforcement learning methods potentially constitute an approach to solve the problem.

The reinforcement learning approach seems to be the most suitable of the four specific machine learning methods, described earlier in this chapter, for solving the problem. Hence, the technical part of this report reviews reinforcement learning methods and investigates how to apply them to the garbage collection problem.
3 Technical Part

3.1 About Part Three
This chapter describes the technical problem solving process. The previous part provided the theoretical background on garbage collection and machine learning algorithms relevant to the problem. In the following a thorough specification of the problem and a method to use for solving it is presented.

First, a technical definition of the problem is given. The technical definition motivates the decision of how and why a certain reinforcement learning method is used. Secondly, a more detailed survey of reinforcement learning methods is presented. The survey provides further justification for the particular approach taken in this project. Some features of different reinforcement learning methods are discussed in the context of the problem as well. Finally, a system specification for a prototype is developed. The performance of the implemented prototype is evaluated for a number of benchmark applications. The chapter concludes with a presentation and analysis of the results obtained in these experiments and a discussion of future problems and investigations.

3.2 Problem Specification
The problem addressed in this project is: how to design and implement a learning decision process for more dynamic garbage collection in a modern JVM. In the following sections the problem is specified more concretely in terms of more specific objectives of dynamic garbage collection, the type of reinforcement learning algorithm that is used to achieve those objectives and the information that it processes.

3.2.1 Concrete Problem Definition
Concretizing the problem leads to a more understandable justification of why reinforcement learning is a suitable solution method. This also contributes to a less abstract explanation of how to solve the problem in practice. The performance of the adaptive decision process with respect to the concrete problem can be measured and compared to current state of the art garbage collecting heuristics. In order to concretize the problem we look at one particular decision in a garbage collector, namely the decision when to garbage collect.

This is an important decision in a JVM as it affects the run-time performance of the application. If garbage collection is invoked too late the running program runs out of memory. Neither must it start too early, as this causes unnecessary garbage collections, which consumes computational resources otherwise available to the running program.

The specification is within the scope of the original problem formulation. The solution to the concretized problem provides valuable insights to the general problem of more dynamic garbage collection. From now on the concrete problem refers to the above specified, concrete task, namely to design a learning decision process for when to garbage collect.

3.2.2 Technical Problem Definition
As discussed in the theoretical part, reinforcement learning methods are the standard way to solve Markov Decision Processes (MDP). Therefore, if the concrete problem can be formulated as an MDP, reinforcement learning can be applied to solve it.
A system has the Markov property if its future evolution only depends on the current state but not its history. A reinforcement learning task that satisfies the Markov property is called an MDP. More formally: if \( t \) indicates the time step, \( s \) is a state signal, \( a \) is an action and \( r \) is a reward, then the system has the Markov property if and only if for all states:

\[
\Pr\{s_{t+1} = s', r_{t+1} = r \mid s_t, a_t, r_t, s_{t-1}, a_{t-1}, \ldots, r_1, s_0, a_0\}
\]

is equal to

\[
\Pr\{s_{t+1} = s', r_{t+1} = r \mid s_t, a_t\}
\]

Which means that the probabilities of the next state \( s_{t+1} \) and reward \( r_{t+1} \) only depend on the current state \( s_t \) and action \( a_t \) [4].

If it is possible to define a way of representing states such that relevant information for making a decision is retained in the current state the garbage collection problem can be formulated as an MDP. Therefore, a prerequisite for being able to use reinforcement learning methods successfully is to select a good state representation. Later in this report an attempt to find such a representation is presented.

In theory it is required that the agent has complete knowledge about the state of the environment in order to guarantee that the learning algorithm asymptotically converges to the optimal solution. However, in practical applications fast learning is often more important than the guarantee of asymptotic optimal performance. In practice, many reinforcement learning schemes are still able to learn proper decision making in a reasonable amount of time even if the Markov property is violated [9].

### 3.2.3 Conclusions

The earlier described properties of reinforcement methods (given in the theoretical part) make them seem like a suitable candidate for solving the concrete problem. The environment and various features of the concrete problem (e.g. the need for online-learning; lack of initial knowledge about the dynamics of the environment; delayed consequences of actions) show that it falls into the realm of reinforcement learning methods. This observation justifies the earlier assertion, stated in the beginning of this chapter, that using reinforcement learning methods may be one solution to the problem.

### 3.3 Method

The following sections discuss reinforcement learning methods in detail. The sections are concerned with features and implementation details of the methods. Above all, the following sections try to answer the question: “Which reinforcement method is most suitable for solving the problem?”

#### 3.3.1 Reinforcement Learning Methods

There are several kinds of reinforcement learning methods. The most common methods are Monte Carlo, temporal-difference, actor-critic and R-learning. Short descriptions of the above mentioned methods are presented below. Pros and cons for why a certain method is more or less suitable for solving the concrete problem are also presented:
• **Monte Carlo** methods, like all reinforcement learning methods, require no model of the environment, but have the disadvantage that the policy is not updated before the end of an episode. In the case of garbage collection an episode either corresponds to a complete execution of the running program or at least the period until the program runs out of memory. Waiting until the end of an episode before updating the policy makes Monte Carlo methods impractical. Therefore, Monte Carlo methods are not used for solving the problem.

• **Temporal-difference** methods update their policy immediately after a new state and reward are observed. This approach seems to be the most suitable approach and is therefore investigated further.

• **Actor-critic** methods use separate memory structures for action selection and state evaluation. The memory usage is almost as crucial as the time performance for a JVM. Hence actor-critic methods are not of interest.

• **R-learning** is primarily a method for undiscounted, continuing tasks. An undiscounted task makes no difference between rewards accomplished earlier or later. This is not used since the concrete problem is a discounted task, i.e. rewards achieved later are less worth than earlier achieved rewards.

Derived from above presented information about different reinforcement learning methods, a temporal-difference method seems to be best suited for solving the concrete problem. There are mainly two different approaches when it comes to temporal-difference methods: Q-learning and SARSA.

To be able to decide which of these two methods is the more suitable one, key issues regarding reinforcement learning in general need to be explained.

### 3.3.2 Exploration vs. Exploitation

Systems solving reinforcement learning problems are confronted with a trade-off between exploration and exploitation. On the one hand they should maximize their reward by always choosing the action \( a = \max_a Q(s, a) \) that has the highest Q-value in the current state \( s \). However, there is also a need to explore alternative actions in order to learn more about the environment. Each time the agent (i.e. the reinforcement learning system) takes an action it faces two possible alternatives. One is to execute the action that according to the current beliefs has the highest Q-value. The other possibility is to explore a non-optimal action with a lower expected Q-value of higher uncertainty. Due to the probabilistic nature of the environment, an uncertain action of lower expected Q-value might ultimately turn out to be superior to the current best-known action. Obviously there is a risk that taking the sub-optimal action diminishes the overall reward. However, it still contributes to the knowledge about the environment, and therefore allows the learning program to take better actions with more certainty in the future [4, 5, 10, 11].

It is said that a learning program needs to explore in the beginning and needs to rely on knowledge later on [4]. Based on that assumption, a way of solving the “exploration versus exploitation”-problem is to use on-policy methods or off-policy methods. As explained before, a policy is representing the behavior of the system: the action selection and the update of Q-values.

The off-policy method follows one policy while updating another. The policy followed in the beginning takes a large number of explorative actions. The off-policy approach satisfies the exploration need as long as the exploring policy is followed. At the same time the experience of the exploration is used to update the non-exploring, non-followed policy. As time progresses, the need for exploration decreases while the need for exploiting increases and therefore the exploring policy is applied less and less frequently in favor of the non-exploring policy.
The on-policy methods, on the other hand, use the same policy for action selection and update. In other words, the on-policy approach evaluates and improves the very same policy that takes the decisions. This approach is used in systems that need to improve while running.

Regardless of what policy approach is being used (off-policy or on-policy), there are three different algorithms for choosing action:

- **The greedy algorithm** chooses the action that is optimal according to the current state-action value function. Whatever action has the calculated, best state-action value in the present state is chosen. This algorithm emphasizes the need for exploitation.

- **The e-greedy algorithm** chooses the calculated, best action most of the times, but with small probability a random action is selected instead. This algorithm satisfies both needs for exploration and exploitation.

- **The soft-max algorithm** works similar to the e-greedy algorithm but does not choose alternative actions completely at random but according to a weighted probability. The probability of an action is weighted with respect to the estimated Q-value of the current state and that action. The main difference between e-greedy and the soft-max algorithm is that in the latter case, when a non-optimal action is chosen, it is more likely that the system chooses the next-best action rather than an arbitrary action. The highest probability is always given to the estimated current best action.

The greedy algorithm suits best in deterministic environments, while the e-greedy algorithm works best in stochastic environments. The soft-max algorithm is the most secure algorithm since it has a low probability of choosing inferior actions. The uncertainty about the application environment, the run-time context and the incomplete state information introduces a stochastic component into garbage collection problem. Hence, the e-greedy algorithm is chosen for the solution of the concrete problem.

Since the system for solving the concrete problem needs to improve while running and explore a lot in the beginning and less over time, the on-policy method SARSA is preferred over the off-policy scheme of Q-learning.

### 3.3.3 Continuous States and Actions

Another common but not always occurring problem are environments that have continuous, and consequently infinitely many states. In these environments it is not possible to store state-action values in a simple look-up table. Such a representation is only feasible for a small number of discrete states and actions. Generalization of states, or rather function approximation of the Q-value function, provides a solution to this kind of problem.

The two main variants of function approximation are: gradient-descent methods and linear methods. Actually the linear methods are a special case of gradient-descent methods, where the approximated Q-value is a weighted linear sum of present state features values. A way to represent continuous states is the use of conjunctions of feature values. In this case the Q-function becomes linear in the binary feature vector and is parameterized by the weights associated to the individual feature. There are mainly four approaches for extracting generalized representation of states:
• **Coarse coding** is a generalization method using a binary vector, where each index of the vector represents a feature of the state, either present (1) or absent (0). In Figure 5 coarse coding is illustrated. The circles are state features and state X has the features A and C present. Since state X has only one feature in common with Z, only partly generalization among them occurs. State X, on the other hand, is completely generalized from Z, since both features are present in both states.

• **Tile coding** is a form of coarse coding where the state feature areas are grouped together in partitions of the state space. These partitions are called *tilings*, and each element of a partition is called a *tile*. This approach approximates the state more accurately. The more tilings there are the more accurate approximation is achieved, but at the cost of higher complexity. In Figure 5 a tiling is shown, divided into four tiles (the stripes). The state X generalizes from state Z, but not from state Y.

• **Radial basis functions** generalize continuous state features in a more accurate way than coarse coding. A feature is represented by a continuous value in the interval [0, 1] rather than a binary value. This value denotes the similarity between the state and the cluster represented by the radial basis function. A radial basis function of a continuous feature A is represented in Figure 5. State X resembles more to state Z than state Y as X and Z more belong to the radial basis function B than A.

• **Kanerva coding** is an alternative representation form of states if the state space has very high dimensionality. Kanerva coding uses an example based representation typical for nearest neighbor methods. A state is then generalized to one of these example states based on how close the state is to the example state. The distance may for instance be measured by counting the number of bits the two states have in common. In Figure 5 the state X is generalized to the example state Z, since this is the closest example state.

![Figure 5](image-url)

*Figure 5* Coarse coding is illustrated to the upper left (1) and tile coding to the upper right (2). To the lower left (3) a radial basis function is presented and to the lower right (4) Kanerva coding is illustrated.
The approaches that are easiest to implement are coarse coding and tile coding. The radial basis functions are probably also of interest when solving the concrete problem, but there is not enough time for investigating all possible approaches within this project.

The generalized state representation, the action value and an approximation parameter vector constitute the input-parameters of the function that calculates an approximated state-action value. A generalized state is represented by a vector \( s \) and an action \( a \). These values are combined linearly by weighting them with the parameter vector \( \theta \):

\[
Q(s, a, \theta) = \theta_1 s_1 + \ldots + \theta_m s_m + \theta_{m+1} a
\]

\( Q(s, a, \theta) \) is the approximated Q-value for being in state \( s \), taking action \( a \). Since the elements of \( s \) are equal to one or zero in tile coding, the Q-value approximations are just a summation of those weight parameters \( \theta \), that correspond to present features \( s_i = 1 \).

### 3.3.4 Conclusions

The answer to the question: “Which method is the most suitable for solving the problem?” is according to the considerations in the above sections: on-policy SARSA with tile coding for generalization of continuous state feature values. A short summary of the motivation for using SARSA is given in the following sections.

One motivation for using SARSA is the fact that it is an on-policy temporal-difference method. On-policy evaluation is desirable since the system for solving the concrete problem needs to improve while running. Tile coding is a commonly used approach for generalization of continuous values and will therefore be used. A restriction to only investigate this approach has been made, since there is not enough time during this project for investigating other approaches.

### 3.4 Suggested System Design

The previous section considered the design and implementation alternatives that are necessary to tackle the concrete problem. The remainder of this thesis is concerned with the design of the system for solving the concrete problem, followed by results of the prototype and a discussion of future work. The reinforcement learning scheme is implemented into a concrete algorithm and the most important results of this project – identification of possible state features, possible actions and possible rewards – are presented.

There are some questions that are relevant to answer when creating a model of the concrete problem for a system specification. The most important question to answer for being able to proceed with the reinforcement approach is: “Is it possible to formulate the concrete problem as a reinforcement problem?”. The answer is: “Yes”, according to previous sections, but only if the states are represented in a way that they contain relevant information about the environment. Other questions to answer are:

- How are the states represented such that they provide all relevant information about the environment?
- Which actions are available to the agent to interact with the environment?
- What rewards are given and how can they be quantified in order to achieve the desired behavior of the garbage collector?

The answers to these questions are provided in following sections.
3.4.1 General Model of the System
The general model of a system for a learning decision process, using reinforcement learning, is shown in Figure 6 below. The reinforcement learning algorithm obtains the information about the current state and the reward from the environment. The reinforcement learning algorithm decides what action to take next and updates its prior belief about the world based on the observed reward and the new state. The process either terminates when a final goal state is reached or in case of an infinite horizon problem continues forever.

1. Environment $\rightarrow$ State + Reward $\rightarrow$ Decision process
2. Decision process $\rightarrow$ Action $\rightarrow$ Environment
3. Environment $\rightarrow$ new State + new Reward $\rightarrow$ Decision process

Figure 6 A model of a reinforcement learning system. First the decision process observes the current state and reward. Then the decision process performs an action that affects the environment. Finally the environment returns a reward and the new state.

3.4.2 Possible State Features
Below some suggestions of possible state features are presented. It cannot be emphasized enough that the choice of state features and penalty/reward function play a crucial role for the ultimate behavior of a reinforcement system. The system can only optimize its behavior according to the objectives specified through the reward function.

A fragmentation factor keeping track of how much of the heap is fragmented could be useful. If the heap is very fragmented garbage collection should be performed more frequently. This is desired in order to collect dead nearby objects of "fragmentations" as fast as possible. By doing that larger blocks of free memory may appear that can be reused. In other words garbage collection should be performed when a lot of non-useful, small blocks of free memory (fragments) occur.

It is important to keep track of how much memory is available in the heap. Based on this information the reinforcement learning system is able to learn at which "allocated memory"-percentage it is most rewarding to perform a certain action, for instance to garbage collect.
If it would be possible to measure the speed at which the running program allocates memory, it would be possible to keep track of when, at the latest, the garbage collector must start garbage collecting for a certain application running. During closer consideration this measurement corresponds to keeping track of the amount of available memory the last time a decision was made.

If it is possible to measure how much time that is really spent on executing instructions of the running program, some evaluation of what extra features that may be added to the default garbage collector could be made. For instance, the longer an application runs the more fragmentation will occur. If fragmentation becomes a problem, compacting becomes useful. If the application runs for a long time, the choices regarding compaction or not would hence be useful additions.

The average size of new allocated objects might provide valuable information about the application running that might affect the performance of the garbage collector. Another feature of the same category is average age of new allocated objects. The number of new allocated objects is another possible feature.

The three last suggestions of state features are vague and may not be relevant, but they cannot be proven to be of no importance either. Hence they need to be explored.

In the case of the concrete problem all the previously described state features may not be required, which is discussed in later sections.

### 3.4.3 Possible State Representation

Each possible measurable value described in the previous section constitutes a possible feature of a state. Since the values are continuous they need to be translated into discrete values. Tilings (see Section 3.3.3) may be used for achieving the translation. One tiling could for example represent a feature combination or feature-action combination. Each tiling is divided into tiles, where each tile corresponds to an interval of one continuous feature or combinations of feature intervals.

A suggestion for representing a state in the general case is to let an array of all tiles constitute the state of the system. Each tile may have the value 1 (the continuous value of the state feature lies within this interval (tile) of the feature tiling) or 0 (it lies not within this interval):

- Current state feature value lies within the corresponding tile \( \rightarrow 1 \)
- Current state feature value lies not within the corresponding tile \( \rightarrow 0 \)

So for example a state could be represented as \( s = [1, 1, 0, \ldots, 1, 0, 1] \), where each index of the vector corresponds to one single tile.

### 3.4.4 Possible Rewards

To evaluate the present situation of the system, measurable values of the goals of the garbage collector are desired. The goals of the garbage collector (described in the theoretical part) concern maximization of the end-to-end performance and minimization of the long pause times, caused by garbage collection. The goal values constitute a basis for rewards and penalties. The reward is always represented as a real-value. The reward function should accordingly consist of a function assigning real-valued rewards to different situations.
A severe problem when deciding the reward function is to decide what is good and what is bad. In the environment of the concrete problem there are a lot of states that are neither bad nor good themselves, but might lead to bad situations. This is only one aspect of the complexity of the environment. Another is that good states hardly exist, while garbage collection always intrudes on the process time of the running program and always constitutes extra costs. This indicates that the reward should only consist of penalties when things go wrong. When this has been tried and evaluated, some new analysis results might contribute to further developments of what is good and what is bad in different application situations. Suggestions of when penalty should possibly be imposed are presented below.

A severe penalty must be imposed if the program running runs out of memory, since this is the worst situation that might occur.

To impose a higher penalty in proportion to the higher quantity of occupied memory would maybe at first sight seem like a good idea, but it is not. Even if the memory is occupied up to 99% it is not a problem, since the running program might complete within the given memory. This is the most desirable case, i.e. to have the program finishing with no garbage collection required. The conclusion is that imposing high penalties for high occupation of memory would not be a good idea.

The freed memory after completed garbage collection could be compared to the occupied memory of the heap before that garbage collection. This measurement would give an estimate of how large percentage of the memory of the allocated heap that has been freed. This freeing rate together with the size of the still unallocated heap would be of interest. If the percentage is high there is nothing to worry about, as is shown in Figure 7. If the percentage is low and the size of the free memory in the heap is low as well, then problems may occur and penalty may be imposed. The latter situation, illustrated in Figure 7, might occur if a running program has a lot of long-living objects and runs for a long time, so that most of the heap will be occupied.

<table>
<thead>
<tr>
<th>1</th>
<th>Example of a good situation</th>
<th>2</th>
<th>Example of a bad situation</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Allocated heap</strong></td>
<td><strong>Unallocated heap</strong></td>
<td><strong>Allocated heap</strong></td>
<td><strong>Unallocated heap</strong></td>
</tr>
<tr>
<td>A</td>
<td></td>
<td></td>
<td>A</td>
</tr>
<tr>
<td>B</td>
<td></td>
<td>C</td>
<td>B</td>
</tr>
<tr>
<td>C</td>
<td></td>
<td>D</td>
<td>C</td>
</tr>
<tr>
<td>D</td>
<td></td>
<td>E</td>
<td>D</td>
</tr>
<tr>
<td>E</td>
<td></td>
<td>F</td>
<td>E</td>
</tr>
<tr>
<td>F</td>
<td></td>
<td>G</td>
<td>F</td>
</tr>
<tr>
<td>G</td>
<td></td>
<td>H</td>
<td>G</td>
</tr>
<tr>
<td>H</td>
<td></td>
<td>I</td>
<td>H</td>
</tr>
<tr>
<td></td>
<td></td>
<td>J</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

6/8 = 75 % freed memory  
4/12 + 6/12 = 83 % free memory

2/10 = 20 % freed memory  
2/12 + 2/12 = 25 % free memory

**Figure 7**  
A good situation with a high freeing rate and much memory left in the unallocated part of the heap is illustrated to the left (1). A worse situation is illustrated to the right (2), where there is little memory left in the unallocated heap and the garbage collection has a low freeing rate. This last situation may cause problems.
Usually the heap is not garbage collected until the heap is full, but with the reinforcement learning system connected it might be more valuable and adapted to collect earlier. If so, the poorer situation of the two in Figure 7 might occur. For example, when applications allocate memory very fast and garbage collection must start earlier in order to be able to finish before the running programs run out of memory.

When using compacting garbage collection, it would be interesting to observe the success rate of allocated memory in the fragmented area of the heap. The fragmented area of the heap means the area of the heap that is most fragmented. The amount of new memory allocated in the fragmented area of the heap could be compared to the amount of the new memory that theoretically could be allocated in the fragmented area of the heap. An illustration of different possible situations is shown in Figure 8. It is desirable that 100% of the new allocated memory is allocated in the fragmented area of the heap, to decrease fragmentation. A proportional penalty could be imposed for a bad percentage.

![Fragmented Heap Diagram](image)

**Figure 8** To the upper right (2) half of the new allocated memory was successfully allocated in the fragmented heap. To the lower left (3) the same percent was successfully allocated in the fragmented heap although space for all new allocated objects exists in the fragmented area. To the lower right (4) all new allocated objects could be successfully allocated in the fragmented heap.

To be forced to take a heap lock, i.e. to lock the free memory of the heap so that no changes can be made to it, ought to be punished.

The longer a compacting garbage collector iterates over the free-list (for explanation see the theoretical part) the higher penalty should be distributed. The longer the system needs to iterate, the more fragmentation exists in the heap. Much fragmentation is not necessarily bad, but the iteration steals time from the program running, which should be punished.

When it comes to compacting garbage collectors a measurement of the effectiveness of a compaction could be a possible base for assigning a reward or a penalty. If there was no need for compacting, the section in question must have been non-fragmented. Accordingly a situation like this should be assigned a reward.
A fundamental rule for imposing penalty should be to punish all activities that steal time from the running program. For instance a punishment might be imposed every time the system performs a garbage collection. An alternative could be to impose a penalty proportional to how much time of the total run time of the program that is spent on garbage collection.

Another possible penalty situation is when the average time of the breaks approaches the maximum allowed break time. It is also important to secure that the number of breaks does not exceed the maximum allowed number of breaks. If the average break time is high and the number of breaks is low, the situation may be balanced through actions taken. If they both are high, not only a more drastic action has to be taken, but also a penalty might be in order.

Another view of the break issue is to impose a higher penalty the longer a break of the running program is. This coincides with the previous thought of every interruption of the running program being punished.

To impose a penalty for not achieved good behavior might also be an interesting idea to consider. For example, when it is not possible to allocate new objects because of a too fragmented heap, penalty may be given.

There is one possible good situation to which a reward, not a penalty, should be assigned. If a compacting collector frees large, connected chunks with memory, a reward would be appropriate. The opposite, if the garbage collector frees a small amount of memory and the running program is still allocating objects could possibly be punished in a linear way, as some of the other reward situations described above.

3.4.5 Possible Action Features

To garbage collect or not is an important issue, stated earlier in this part. This is in fact the decision that is make by the prototype in this project. A discussion about this issue is presented in later sections.

When the memory is not large enough and the garbage collection did not accomplish to free a satisfactorily large amount of memory, in some cases the heap is extended. The decision of extending the heap or not, or in the future, if a functionality of decreasing the heap size is implemented, to decrease it or not, may be a decision to make – an action to take. A subsequent decision would be to which extent the heap should be increased or decreased.

To save heap space or rather to use it more effectively, a decision of compacting the heap or not, could also be of interest. And if so, how large area of the heap should then be compacted. Another subsequent decision could be what section of the heap to compact.

To handle synchronization between allocating threads of the running program, the heap is divided into Thread Local Areas (TLA). Each allocating thread is allowed to allocate memory within only one TLA at a time and there is only one thread permitted to allocate in a certain TLA. A decision to be made when it comes to TLAs may be the size of each TLA.

When allocating large objects a Large Object Space (LOS) could be used, especially in generational garbage collectors, to prevent large objects to be moved. Deciding the size of the LOS and how large an object has to be, to be treated as a large object, could be additional issues for the reinforcement learning decision process to consider.

Memory Block Size (MBS) is the minimum size of a free memory block for being added to the free list. Different applications may cause different needs when it comes to this size value.

For the moment the MBS and the TLA metrics are dependent of each other. It may be of interest to investigate how to choose different sizes for them in the future.
Should only one generation be used, or, if more than one, how many? This question may be relevant in the future. Today it would only be of interest to decide before start-up of the garbage collector if it should have one or two generations. It may also be possible, even today, to change from two generations into one, but not the other way around. When it comes to future generational garbage collectors it would be of interest to let the system vary the size of the different generations. If there is a promotion rate available, this is a factor that might be interesting for the system to vary.

If the garbage collector should use an incremental approach and, in that case, decide the size of the heap area that should be collected at a time, might be an interesting aspect to consider. The same goes for using the concurrent approach or not, together with the factors of how many garbage collection steps at a time and how long time the system should pre-clean (for explanation see the theoretical part).

When parallel garbage collection is implemented some time in the future, it might not be of interest to be able to choose between parallel garbage collection or not. It would not be of interest since there are only advantages with using parallel collectors where several processors are available.

### 3.4.6 Possible Action Representation

Actions may be represented as positive, discrete values: 1, 2, ..., N; where N is the total number of actions. Each representation value corresponds to a specific action. The representations in the binary choice cases suggested above look like:

- Perform $\rightarrow 1$
- Do not perform $\rightarrow 0$

### 3.4.7 Interesting Comparative Measurements

There are above all two values of interest for comparison with the existing garbage collecting system. On one hand, the measurement of the performance of the current garbage collector compared to the garbage collector integrated with the reinforcement learning system would be of interest. The performance may either be measured based on the accumulated reward over time, since the reward function should reflect achieved good behavior, or by measuring time for completing certain tasks. The time measurements reflect how many times each system has performed a garbage collection, which is the most interesting factor to measure.

On the other hand, it must not take too long for the system to learn. This metric (if measurable) must also be taken under consideration when evaluating the comparison between JRockit and the reinforcement learning system.

It would also be important to observe if learning occurs at all. In other words: does the garbage collector with the integrated reinforcement learning system improve its performance over time? Performance may be measured through observation of the average reward obtained but also according to the goals of the garbage collector (i.e. the features underlying the reward system).

### 3.5 Adapting the System Design for the Concrete Problem

The following sections describe the design of a system solving the concrete garbage collection problem.
3.5.1 Restrictions
The prototype is restricted to handle only one decision, namely the decision of when to garbage collect. The comparison with a “mostly-concurrent” garbage collector will be performed in similar environments. In the beginning only one application running at a time is considered, but with possible extensions further on.

The state features constituting a state representation will exclusively be those concerned with the concrete problem.

The prototype will use tile coding only, although there may be good results with other approaches.

3.5.2 State and Action Features of the Concrete Problem
Necessary measurements, derived from Section 3.4.2, for creating a state for the solution system of the concrete problem would be:

- The amount of allocated memory per time unit
- The amount of allocated memory the last time a decision was made
- How much of the heap is fragmented

To be able to explore the earlier stated application specific state features they can be added to a later version of the prototype. In that case, the application specific state features of interest regarding the concrete problem would be:

- Average size of new allocated objects
- Average age of allocated objects
- Average amount of new allocated objects

It is also important to observe events underlying the rewards and penalties. These are not state features, but are of interest for deciding rewards and penalties. The features underlying the reward system, derived from Section 3.4.4, would be:

- A variable representing if a garbage collection was made during the last time step
- A variable representing if the system ran out of memory during the last time step
- The amount of occupied memory before the garbage collection
- The amount of occupied memory left after completed garbage collection
- The break length of phase one of a “mostly-concurrent” garbage collector
- The break length of phase four of a “mostly-concurrent” garbage collector
- The number of situations where a heap lock needed to be taken

As stated before, the action to take consists of one choice only: the choice of performing a garbage collection or not at a certain time step. The action representation is in this case binary (1 = perform, 0 = do not perform). This means that the action value does not need to be re-calculated in any way.

3.5.3 Adapting the SARSA Algorithm
If \( \mathbf{s} \) and \( \mathbf{a} \) are the vectors representing states and actions, then the estimated state-action value of that state and action is \( Q(\mathbf{s}, \mathbf{a}) \). The linear gradient-descent approximation of the action-value function \( Q(\mathbf{s}, \mathbf{a}) \) will then be \( Q(\mathbf{s}, \mathbf{a}, \mathbf{\theta}) \), where \( \mathbf{\theta} \) is a vector containing the weight coefficients \( \theta_1, \theta_{(m+n)} \) below.
For a fixed $\theta$, the approximated $Q$-function value only depends on $s$ and $a$:

$$Q(s, a) = \theta_1 s_1 + \ldots + \theta_m s_m + \theta_{(m+1)} a_1 + \ldots + \theta_{(m+n)} a_n$$

If $s$ is a vector of size $m$ and $a$ is a vector of size $n$, then $\theta$ must be a vector of size $m + n$. Remember that each index of $s$ corresponds to either a single state feature interval, a combined interval of two or more state features or combinations of actions and state feature intervals, while the indexes of $a$ corresponds to different actions.

The gradient of the function approximation $Q(s, a, \theta)$ is needed for using gradient-descent function approximation. The gradient of $Q(s, a, \theta)$ with regard to $\theta_i$ is:

$$\nabla_\theta Q(s, a, \theta)[d Q(s, a, \theta) / d \theta, \ d Q(s, a, \theta) / d \theta]$$

where in the linear case:

$d Q(s, a, \theta) / d \theta_i = s_i$, for $0 \leq i < n$

$d Q(s, a, \theta) / d \theta_i = a_i$, for $n \leq i < n + m$

The pseudo code in Figure 9 is based on pseudo code from Sutton’s book about reinforcement learning [5]. Modification of it has been made in order to suit the problem better. The pseudo code concerns SARSA with linear, gradient-descent function approximation using a soft-max policy.

```
Initialize $\theta$ arbitrarily.
Repeat for each episode:
  $s \leftarrow$ initial state of episode
  $t \leftarrow 0$
  For all $a \in A(s)$ (all possible actions $a$ to take from state $s$)
    $F_s \leftarrow$ set of features present in $s = [s_1, \ldots, s_m]$ and $a = [a_1, \ldots, a_n]$
    $Q_s \leftarrow \sum_{s' \in F_s} \theta_i$
    $a \leftarrow \argmax_{a'} Q_{s'}$
    with probability $\epsilon = 1/t$: $a \leftarrow$ random action $\in A(s)$
  Repeat for each step of the episode:
  Take action $a$
  Observe $r$ (the reward)
  Observe $s'$ (the next state)
    For all $a' \in A(s')$
      $F_{s'} \leftarrow$ set of features present in $s' = [s'_1, \ldots, s'_m]$ and $a' = [a'_1, \ldots, a'_n]$
      $Q_{s'} \leftarrow \sum_{s' \in F_{s'}} \theta_i$
      $a' \leftarrow \argmax_{a'} Q_{s'}$
    with probability $\epsilon = 1/t$: $a' \leftarrow$ random action $\in A(s')$
    $\theta \leftarrow \theta + \alpha [r + Q(s', a') - Q(s, a)] V(s, a, \theta)$,
    where $Q(s', a') = Q_{s'}$ and $Q(s, a) = Q_s$
    $a \leftarrow a'$
    $t \leftarrow t + 1$
  Until eternity
  * For each action-state pair there will be an estimate of the value of the pair, based on the sum of the values in $\theta$
  * Find the highest state-action value for this state and choose the corresponding action.
```

**Figure 9** Pseudo code modified to suit the concrete problem.

### 3.5.4 Problems During Development

During development of a pre-version of the prototype a problem occurred when applying the linear approximation. The problem showed to be a common problem in neural networks systems: the Exclusive-Or (XOR) problem. The XOR problem concerns how a learning system may arrive at identical output when the input data has nothing in common and is based on XOR reasoning. For further explanation of the XOR problem see references [31, 33, 32, 38].
Another problem that occurred while developing the prototype was that the exploring decreased too fast. This problem was solved through changing the random action choice function to a non-linear function:

Probability to choose a random action \( P = P_0 \times e^{-\frac{\text{TimeStep2}}{C}} \)

Where \( C \) is somewhere between 2000-5000 and \( P_0 = 0.5 \). \( C \) corresponds to the square number of steps at which the original probability \( P_0 \) of choosing a random action decreased by a factor \( e^1 \).

A third problem that occurred was that it turned out that phase five of the garbage collection in JRockit is optimized in a way that makes it difficult to measure the fragmentation percentage without redesigning the garbage collector. If fragmentation should be measured in the today existing system it would result in a very high uncertainty of the measured value. Hence, to be able to achieve reliable results of the prototype the decision was made that no consideration would be taken to the fragmentation percentage. The assumption is that the amount of available memory is of more importance to the decision of when to garbage collect than the fragmentation percentage and accordingly will give enough information about a situation for being able to achieve a satisfying behavior.

3.6 The Prototype

The state features \( s_1 \) and \( s_2 \) used in the prototype are the current amount of available memory and the amount of memory available at the previous time step.

There is only one binary decision to make, namely whether to garbage collect or not. Hence, the action set contains only two actions \( \{0, 1\} \), where 1 represents performing a garbage collection and 0 represents not performing a garbage collection.

The reward function of the prototype imposes a penalty (-10) for performing a garbage collection. The penalty for running out of memory is set to -500. It is difficult to specify the quantitative trade-off between using time for garbage collection and running out of memory. In principle the later situation should be avoided at all costs, but a too large penalty in that case might bias the decision process towards too frequent garbage collection. Running out of memory is not desirable since a concurrent garbage collector is used. A concurrent garbage collector must stop all threads if the system runs out of memory and that is to prevent the purpose of using a concurrent garbage collector.

The random probability function that determines whether to pick the action with the highest Q-value or a random action for exploration is implemented according to the formula stated before:

\[
\text{Probability to choose a random action} = 0.5 \times e^{-\frac{\text{TimeStep}}{C}}
\]

where \( C \) is set to 5000 in the prototype, which means that random actions are taken until ca 25000 time steps elapsed. A time step corresponds to a time point where the RLS makes a decision. Between each time step a time interval of about 50ms elapses after which RLS makes a new decision.

The learning rate \( \alpha \) is set to decrease over time. The function that determines the learning rate is implemented according to the formula stated below:

\[
\text{Learning rate} = 0.1 \times e^{-\frac{\text{TimeStep}}{D}}
\]

Where \( D \) is set to 20000 in the prototype. The discount factor gamma is set to 0.9.
The tile coding representation of the state in the prototype is chosen to be one 10x10x2-tiling for the two possible actions combined with each combination of both state features in the case where both state features were used and one 10x2-tiling in the case of when only \( s_1 \) was used.

A non-uniform tiling was chosen, in which the tile resolution is increased for states of low available memory, and a coarser resolution for states in which memory occupancy is still low. The tiles for feature \( s_1 \) correspond to the intervals \([0, 4], [4, 8], [8, 10], [10, 12], [12, 14], [14, 16], [16, 18], [18, 20], [22, 26] \) and \([30, 100] \). The tiles for feature \( s_2 \) are the same as for feature \( s_1 \).

The test applications used for evaluation are designed to behave in three different ways. All test applications alternate between two different memory allocation behaviors, one with a high allocation rate and one with a low allocation rate. Each behavior lasts for a certain *time interval*, which duration is measured in terms of the number of iterations. The first test application has intervals that are 10000 iterations long. The second test application has intervals that consist of 20000 iterations. The third test application alternates randomly between intervals consisting of 10000 iterations and 20000 iterations as well as between the two different allocation behaviors.

### 3.7 Results

One of the main objectives of the project was the identification of suitable state features, underlying reward features and action features for the dynamic garbage collection learning problem. An additional objective was the implementation of a simple prototype and the evaluation of its performance on a restricted set of benchmarks in order to investigate whether the proposed machine learning approach is feasible at all.

This section compares the performance of an ordinary JVM and a JVM using reinforcement learning for making the decision of when to garbage collect. The JVM using reinforcement learning is referred to as the RLS (the Reinforcement Learning System) and the ordinary JVM as JRockit.

Since JRockit is optimized for environments in which the allocation behavior changes slowly, environments where the allocation behavior changes more rapidly might cause a degraded performance of JRockit. In these environments it is of special interest to investigate if an adaptive system, such as an RLS, is able to perform equally well or better than JRockit. So far, both systems are tested and compared only with respect to applications that exhibit different memory allocation rates, described in Section 3.6. In the future it might be of interest to measure and compare other aspects as well.

The left graph in Figure 10 illustrates the interval time performance, namely the number of milliseconds required to complete a fixed number of iterations, for the test application with the short intervals. The upper chart shows the performance during the first 20 intervals of the session and the lower chart shows the performance during 20 intervals after approximately 50000 time steps. In the beginning the RLS performs a lot worse than JRockit due to the random choices of actions and the fact that the RLS is still learning about the environment. After about 50000 time steps the performance of the RLS compared to JRockit is about the same. To the right in Figure 10 the accumulated time performance is illustrated, which also shows the tendency of a decreasing need of time, i.e. decreasing frequency of garbage collections, for the RLS system as it learns.
To the left the interval performance of the RLS is compared to the interval performance of JRockit when running the application with short intervals. To the right the accumulated time performance is illustrated. The upper charts show the performances during the first 20 intervals and the lower chart shows the performances during 20 intervals after ca 50000 time steps.

In Figure 11 the accumulated penalty for the RLS during the test session with the short intervals is illustrated as well as the accumulated penalty for JRockit. The accumulated penalty for running out of memory becomes constant over time, which demonstrates that the RLS actually learned to avoid running out of memory. Notice, that during the entire learning phase the RLS only ran out of memory five times. After 13000 time steps all future penalties imposed on the RLS are due to garbage collection only. After about 20000 time steps the rate at which JRockit and the RLS are penalized for invoking garbage collections becomes similar (lower graph Figure 11).
Figure 11 illustrates the same results as Figure 10, but for the test application with the long intervals. As may be seen, the RLS performs slightly worse in the beginning than in the short interval application case. This application environment seems to be more difficult for the RLS to learn, due to the fact that it runs out of memory more times than in the previous case during the learning phase (nine times instead of five times).
To the left the interval performance of the RLS is compared to the interval performance of JRockit when running the application with long intervals. To the right the accumulated time performance is illustrated. The upper charts show the performances during the first 20 intervals and the lower charts show the performances during 20 intervals after ca 50000 time steps.

In Figure 13 the accumulated penalty for the RLS during the test session with the long intervals is compared to the accumulated penalty for JRockit. The results are almost the same as for the application with the short intervals, as mentioned above. The accumulated penalty for running out of memory becomes constant over time in this case too and the accumulated penalty for invoking garbage collections develops in the same way as in the previous case.
Figure 13. The upper chart shows the accumulated penalty for the RLS compared to the accumulated reward for JRockit when running the application with long intervals. The lower chart shows the average penalty as a function of time.

Figure 14 shows the results of using the RLS and JRockit when running the application with intervals of random duration and memory allocation rate. The left part of Figure 14 shows the interval performances for RLS and JRockit. Due to the random distribution of intervals an interval-to-interval performance comparison of these two different runs is not meaningful. Instead, the accumulated time performances illustrated to the right in Figure 14 are used for comparison. As can be seen in the lower chart to the right the RLS performs slightly better than JRockit in this dynamic environment. This confirms the hypothesis of an RLS being able to outperform an ordinary JVM in a dynamic environment.
To the left the interval performance of the RLS is compared to the interval performance of JRockit when running the application with randomly appearing intervals. To the right the accumulated time performance is illustrated. The upper charts show the performances during the first 20 intervals and the lower charts show the performances during 20 intervals after ca 50000 time steps.

Figure 15 illustrates the accumulated penalty for the RLS during the test session with random appearing intervals compared to the accumulated penalty for JRockit. The results show that the RLS runs out of memory a few times more than in the other cases, but learns to avoid it over time even in this more dynamic case.

The upper chart illustrates the accumulated penalty for the RLS compared to JRockit during a test session with the application with randomly appearing intervals. The lower chart illustrates the average penalty as a function of time.
In Table 1 the accumulated penalty during a time period where the RLS has completed its learning is shown. As may be seen, the results of the RLS are comparable to the results of JRockit. In the case of the test application with random appearing intervals the value in the table verifies the results presented above: that the RLS performs better than JRockit in the environment that was constructed to be more dynamic.

**Table 1** The table illustrates the accumulated penalty from time step 30000 to time step 50000. This corresponds to the performance of the RLS after completed learning.

<table>
<thead>
<tr>
<th>Test application type</th>
<th>Accumulated penalty for the RLS</th>
<th>Accumulated penalty for JRockit</th>
</tr>
</thead>
<tbody>
<tr>
<td>Short intervals</td>
<td>-8640</td>
<td>-7910</td>
</tr>
<tr>
<td>Long intervals</td>
<td>-8970</td>
<td>-8520</td>
</tr>
<tr>
<td>Random intervals</td>
<td>-8400</td>
<td>-8550</td>
</tr>
</tbody>
</table>

The upper chart in Figure 16 shows the Q-function after ca 2500 time steps. The probability of choosing a random action is still very high and the frequency of choosing the action to garbage collect is high enough to prevent the system from running out of memory. On the other hand the high frequency of random actions during the first 5000 time steps does not require the system to pick a garbage collection action, which means that it will always favor not to garbage collect in order to avoid the penalty. Running out of memory never occurs due to the high value of \( p_0 \) (0.5) in the probability function for choosing a random action. This can easily be adjusted by choosing a lower value of \( p_0 \). The only thing the system has learned so far is that it is better to not garbage collect than to garbage collect with a Q-value difference of 10, which is the penalty of invoking a garbage collection.

The middle chart in Figure 16 shows the Q-function after ca 10000 time steps. The probability of choosing a random action has now decreased. The frequency of invoking a garbage collection has led to a situation where the system actually runs out of memory and RLS incurs a large penalty, and thereby improves its knowledge about when it is preferable to garbage collect.

The lower chart in Figure 16 illustrates the Q-function after ca 50000 time steps. At this point of time the Q-values for the different states converged and RLS follows a policy that is optimal with respect to the particular test application and the reward function.
The figure shows the development of the state-action value function, the $Q$-function, over time. The upper chart shows the $Q$-function after ca 2500 time steps. The middle chart shows the $Q$-function after ca 10000 time steps and the lower chart shows the $Q$-function after ca 50000 time steps and is then constant.

The overall behavior of RLS is quite similar for the three test cases presented above. However, there is a slight difference regarding the number of times the system runs out of memory during learning. During the first test application the system runs out of memory five times, while during the second and third test application the system runs out of memory nine and ten times respectively. This indicates that the later two scenarios are a bit more difficult to learn due to the dynamic memory allocation rate.

The performance comparison between the RLS and JRockit in the case of random intervals suggests that further investigation in this kind of environment might be of interest. Regarding the fact that this first version of the prototype only considers a single state feature, it would be interesting to investigate the performance of an RLS that takes additional state features into consideration, in order to achieve even better performance. The results of this investigation are presented below.

Unfortunately, the results from using both the state features $s_1$ and $s_2$ (the current amount of available memory and the previous amount of available memory) showed to be worse than in the case of only one state feature. The main reason for the inferior behavior is probably that the new feature increases the number of states and that therefore converging to the correct $Q$-values requires more time. Another reason is probably that the state feature $s_2$ does not contain the right information as a lot of states that are never visited, e.g. $s_1 = 10\%$ and $s_2 = 70\%$. It would probably be more adequate to use the change in available memory $s_1 - s_2$ as an additional feature at a resolution: $[0 - 2], [3 - 4], [5 - 6], [7 - 8], [9 - 10]$. In any case the probability for choosing a random action and the learning rate should be adjusted such that all states at which the system potentially could run out of memory are visit frequently enough. The lower chart of Figure 17 shows that the system even at time step 45000 runs out of memory twice and therefore has not yet converged to a proper $Q$-function and policy.
Figure 17 The upper chart shows the accumulated penalty for JRockit compared to the accumulated penalty for the RLS using two state features when running the test application with randomly appearing intervals. The lower chart shows that the system still runs out of memory after ca 50000 time steps and hence has not learned all states that lead to running out of memory due to the increased amount of states and to the additional state feature not giving enough information.

Plots of the Q-function at different stages during the test session are illustrated in Figures 18, 19 and 20. In Figure 18 the Q-function at time step 2500 is illustrated.
The figure shows a contour-plot of the $Q$-function at time step 2500, when the system has not yet run out of memory. To the upper left the plot for choosing action $a_0$ (not to garbage collect) is illustrated, while to the upper right the plot for choosing $a_1$ (to garbage collect) is illustrated. To the lower left the difference in $Q$-values is illustrated and to the lower right the states never visited are illustrated.

At time step 2500 the system has not yet run out of memory and hence has not yet learned any state that leads to a penalty of -500. The $Q$-value for not performing a garbage collection is always better than the alternative action to perform a garbage collection. After about 10000 decisions (i.e. at time step 10000) the system encounters states in which it runs out of memory. This can be seen in Figure 19 as in states of little memory available the $Q$-values for performing garbage collections are higher than those for not performing garbage collections.
Figure 19  

The figure shows a contour plot of the $Q$ function at time step 10000, when the system has started to occasionally run out of memory. To the upper left the plot for choosing action $a_0$ (not to garbage collect) is illustrated, while to the upper right the plot for choosing $a_1$ (to garbage collect) is illustrated. To the lower left the difference in $Q$-values is illustrated and to the lower right the states never visited are illustrated.

Whereas Figure 19 illustrates the contour plots of the $Q$-function after 10000 time steps, Figure 20 shows the same information after 50000 time steps. At this stage the $Q$-values did converge. It is interesting to observe that the part of the state space for which garbage collection is preferred is much smaller than in the case of only one state feature, where the decision boundary for $s_1$ was about 12-14%.
Figure 20 The figure shows a contour plot of the $Q$-function at time step 50000, when the system has stopped learning. To the upper left the plot for choosing action $a_0$ (not to garbage collect) is illustrated, while to the upper right the plot for choosing $a_1$ (to garbage collect) is illustrated. To the lower left the difference in $Q$-values is illustrated and to the lower right the states never visited are illustrated.

Figure 21 is an enlarged region to show the details from the contour plots in Figure 20 where $s_1$ and $s_2 < 15\%$. As may be observed, $s_2$ plays some role, otherwise the decision boundary would be a line parallel to the $y$-axis. For example, the additional state feature seem to matter in the state $s_1 = 10\%$ and $s_2 = 15\%$. This situation represents a high memory allocation rate (about 5\%) and the $Q$-value for performing a garbage collection is higher than for not performing one. On the other hand, in the state $s_1 = 10\%$ and $s_2 = 12\%$ for which the memory allocation rate is low (about 2\%), the action not to garbage collect has higher $Q$-value than the action garbage collect. Such a behavior is intuitively comprehensible, even though the entire decision boundary for even lower values of $s_1$ and $s_2$ cannot be explained satisfactorily. It might be that these states of very low memory ($s_1, s_2 < 5\%$) are not visited at all once garbage collection is invoked for their successor states. Therefore, the $Q$-values for this part of the state space are not correct.
The figure shows an enlarged contour-plot of the $Q$-function at time step 50000, to be able to see the detailed decision boundary when $s_1$ and $s_2 < 15\%$. To the upper left the plot for choosing action $a_0$ (not to garbage collect) is illustrated, while to the upper right the plot for choosing $a_1$ (to garbage collect) is illustrated. To the lower left the difference in $Q$-values is illustrated and to the lower right the states never visited are illustrated.

In all the plots above it can be observed that for high memory available the difference between the $Q$-values for performing a garbage collection and not performing a garbage collection is about 10, which matches exactly the penalty for performing a garbage collection. This makes sense insofar as the state after performing a garbage collection when the amount of memory available is high is also one of high memory available. It can also be seen that states for which $s_2$ is much smaller than $s_1$ never occur as the memory allocation rate is limited. This observation suggests that the memory allocation rate $s_2 - s_1$ might be a better state feature to use than $s_2$.

The decision boundary in the case where two state features were used is more complex than in the case when only one state feature was used. Basically there are more states in the former case, for which the RLS has to learn that it runs out of memory if it does not perform a garbage collection. A way of handling this problem could be to use more tilings, e.g. one for each state feature separately and one separate for the combination of the two state features.

Another problem to notice is that learning in the case where two state features are considered seems to be more difficult, as the state space is more complex. The complexity depends on the increased number of states, which leads to the increased time it takes for the system to explore the state space. The system also runs out of memory more often due to the increased number of states to visit before learning an optimized behavior. To some extent $Q$-function approximation (i.e. tile coding, function approximation) provides a remedy to this problem. Further investigation regarding this aspect is needed, which is discussed in the next session.

### 3.8 Discussion and Developments

The most important task of future investigation is to systematically investigate the effect of using additional state features for the decision process and to investigate their usefulness for making better decisions.
The second important aspect would be to investigate more complex scenarios of memory allocation, in which the memory allocation behavior switches more rapidly. It would also be of interest to investigate other dimensions of the garbage collecting problem such as object size, levels of references between objects, among others. It is important to underline that the results above are derived from a limited set of test application that cannot adequately represent the range of all possible applications.

The issue of selecting proper test application environments also relates to the problem of generalization. The question is: how much does training on one particular application or a set of multiple applications help to perform well on unseen applications? It would be interesting to investigate how long it takes to learn from scratch or how fast an RLS can adapt when the application changes dynamically.

A suggestion for improving the system is to decrease the learning rate more slowly. The same suggestion applies to the probability for choosing a random action in order to achieve a better balance between exploitation and exploration. The optimal parameters are probably best determined by cross-validation.

An approach for achieving better results when more state features are taken into account might be to represent the state features differently. For instance, as mentioned earlier in this report, radial basis functions might be of interest for generalization of continuous state features. An even better approach would be to represent the state features with continuous values and instead use a gradient-descent method for approximating the Q-function.

A significant factor to consider is the amount of state features. JRockit considers only one parameter for the decision of when to garbage collect. The current performance when using two state features was not improved, most likely due to the wrong state feature information being used. The question is whether the performance of the RLS improves if additional state information is available. The potential strength of the RLS might reveal itself better if the decision is based on more state features than JRockit uses currently. The choice of what parameters to include is crucial to the performance as has been shown above.

Another important aspect is the online vs. offline performance. How much learning can be afforded, or shall one only look at online-performance? That of course is also a design issue for JRockit, which requires a more precise definition of the concrete objectives one expects from a dynamic JAVA virtual machine.

Once a real system has been developed from the prototype, it could also be used to handle some of the other decisions related to garbage collection proposed in this report.

It is recommended to investigate this research area further, since it is far from exhausted. Considering that the results were achieved using a prototype that is poorly adjusted in several aspects, further development might lead to interesting and even better results than obtained within the restricted scope of this project.

3.9 Conclusions

This report has investigated how to design and implement an automatic and learning decision process for more dynamic garbage collection in a modern JVM. The results of this thesis show that it is in principle possible for a reinforcement learning system to learn when to garbage collect. It has also been demonstrated that on simple test cases the performance of the RLS after training in terms of the reward function is comparable with the heuristics of JRockit.

The time it takes for the RLS to learn seems also reasonable since the system only runs out of memory 5-10 times during the learning period. Whether this cost of learning a garbage collecting policy is acceptable in real applications depends on the environment and the requirements on the JVM.
From the results in the case of two state features, it becomes clear that using multiple state features potentially results in more complex decision surfaces than simple standard heuristics. Observations have also been made that there exists an evident trade-off between using more state features, in order to make more optimal decisions, and the increased time required for learning due to an enlarged state space.

From the above conclusions it can be claimed that this project has shown that the use of a reinforcement learning system is particularly useful if an application has a complex dynamic memory allocation behavior, which is why a dynamic garbage collector was proposed in the first place. It is very interesting to observe that machine learning through an adaptive and optimizing decision process can replace a human designed heuristic such as JRockit that operates with a dynamic threshold.
I would like to express my sincere gratitude to my supervisor at KTH, Frank Hoffmann, for sharing his knowledge and experience, giving competent advice and being supportive during the whole project. I would also like to thank my supervisor at Appeal Virtual Machines, Olof Lindholm, for very fruitful discussions and intelligent remarks, not to forget his support and guidance during this project. I also wish to thank my examiner Anders Lansner for his involvement with this project.

Special thanks also go to Anders Ingeborn for his good advice and clever ideas during the development of this project, to Marcus Lagergren, Magnus Bråding and Irja Andreasson for spending precious time proofreading this report and giving valuable advice. For the warm and inspiring atmosphere as well as encouragement I wish to thank everyone working at Appeal Virtual Machines during my work on this project.
5 References

5.1 Literature

5.2 Papers

5.3 WWW
15. Containing a borrowed picture.
http://grial.uc3m.es/~ffernand/publicaciones/roboocup99/node2.html, last visited 2001-10-11.

18. General information about decision trees. 

19. General information about decision trees. 

20. General information about garbage collection, FAQ. 


22. General information about reinforcement learning. 

23. General information about supervised learning. 


http://gepasi.dbs.aber.ac.uk/roy/supervis/anns.htm, last visited 2001-10-09.

http://gepasi.dbs.aber.ac.uk/roy/supervis/suprlrn.htm, last visited 2001-10-09.

http://gepasi.dbs.aber.ac.uk/roy/tutorial/pymstut.htm, last visited 2001-10-09.

http://www.shef.ac.uk/psychology/gurney/notes/l10/subsubsection3_3_6_2.html, last visited 2001-10-02.


http://tpb.tuwien.ac.at/~oemer/doc/neurugen/node34.html, last visited 2001-12-17.

http://cogsci.soton.ac.uk/~harnad/Hypermail/ExplainingMind96/0291.html, last visited 2001-12-17.


37. Wall, M. General information about genetic methods.  
http://lancet.mit.edu/~mbwall/presentations/IntroToGAs/, last visited 2001-10-10.

http://nick-5.neuro.informatik.uni-kassel.de/~werner/NN/NN03/sld012.htm, last visited 2001-12-17.
Appendix A: Vocabulary

**Actions**: actions interact with the environment and are chosen based on a behavior policy from each state according to a state-action value function.

**Beliefs**: see model.

**Conservative**: the word exact is used for the approach where pointers to objects do not need exact identification.

**Concurrent**: garbage collection performed “little at a time”, where “little at a time” means one garbage collection step at a time, is called concurrent garbage collection.

**Dead**: an object is dead if it cannot be reached from a running program.

**Dirty**: an object that has been changed during a concurrent phase of a “mostly-concurrent” garbage collector is said to be dirty and must hence be traced again before sweeping.

**Exact**: the word exact is used for the approach where pointers to objects need exact identification.

**Flip**: to flip is to change the semi-space to be scanned of a copying garbage collector from the one recently scanned to the non-recently scanned semi-space.

**Fragmentation**: scattered memory pieces that cannot satisfy a certain memory need although the free memory in the heap in total would.

**Free-list**: the free-list is a linked list of all free blocks of memory available in the heap.

**Garbage collection**: an automatic memory-deallocating process is called a garbage collection.

**Garbage**: from a running program non-reachable objects

**Gene**: each element of an input string or array to a genetic method is called a gene.

**Goals**: see model.

**Heap**: memory is allocated in the heap.

**Hypothesis**: see model.

**Incremental**: garbage collection performed “little at a time”, where “little at a time” means one area of the heap at a time, is called incremental garbage collection.

**Individual**: the input of a genetic method is called an individual.

**Live**: an object is live if it can be reached from a running program.

**Mark-and-compact**: a garbage collection approach that uses the mark-and-sweep approach, but tries to move objects close together to prevent fragmentation.

**Mark-and-sweep**: a garbage collection approach that marks all live objects and then collects the non-marked objects.

**Markov Decision Process**: a reinforcement learning task that fulfills the Markov property is called a Markov Decision Process (an MDP).

**Markov property**: an input signal succeeding in providing all relevant information for making a correct decision has the Markov property.

**Model**: a model is the beliefs about the environment of a learning system.

**Off-policy**: when following one policy and updating another an off-policy approach is used.
On-policy: when following and updating the same policy an on-policy approach is used.

Parallel: garbage collection performed in parallel, performed in a multi-processor environment, is called parallel garbage collection.

Policy: a policy (or behavior policy) defines the behavior of the system at a given time.

Pre-cleaning: step three of a “mostly-concurrent” garbage collector includes checking objects that are marked dirty, this is called pre-cleaning.

Q-value function: see State-action value function.

Reward: a reward is calculated by a reward function and corresponds to an evaluation of the feedback from the environment after a certain action is performed.

Roots: objects that the running program stores in registers or on the stack are known to be live. Objects that are known to be live are called roots.

State-action value function: the state-action value function is the function that calculates the value of taking a certain action from a certain state.

States: states are representations of the environment, the input of a reinforcement learning system.

Stop-and-copy: a garbage collection approach that divides the heap into two semi-spaces and collect one semi-space at a time by moving all live objects in one semi-space into the other and then flip.

Supervised learning: supervised learning is learning from examples provided by a knowledgeable external supervisor.

Unsupervised learning: unsupervised learning is learning through “trial and error” and improves behavior through a reward function (feedback from the environment).

Update: a learning system can evaluate and improve the policy based on the reward and thereby make better decisions further on.

Value function: see State-action value function