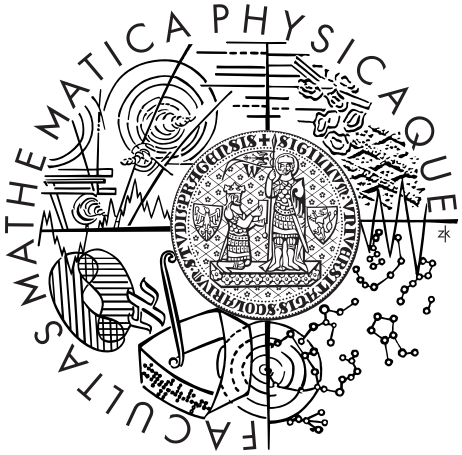
Charles University in Prague
Faculty of Mathematics and Physics

DOCTORAL THESIS



Mgr. Stanislav Slušný

Control algorithms for autonomous embodied agents

Department of Software Engineering

Supervisor of the doctoral thesis: Mgr. Roman Neruda, CSc.
Study programme: Computer Science
Specialization: Software Engineering

Prague 2014

I declare that I carried out this doctoral thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular the fact that the Charles University in Prague has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 paragraph 1 of the Copyright Act.

In ..... date ..... signature of the author

Název práce: Řídicí mechanismy pro autonomní vtělené agenty

Autor: Mgr. Stanislav Slušný

Katedra: Katedra softwarového inženýrství, Matematicko-fyzikální fakulta, Univerzita Karlova v Praze

Vedoucí disertační práce: Mgr. Roman Neruda, CSc., Ústav informatiky Akademie věd České republiky, v.v.i., Praha

Abstrakt: Tato práce se zabývá studiem řídících algoritmů pro adaptivní vtělené agenty. Zkoumáme přístupy založené na neuronových sítích, genetických algoritmech a posilovaném učení, a navrhujeme jejich vylepšení. Hlavním výsledkem práce je návrh architektury vtěleného autonomního agenta, která kombinuje reaktivní a deliberativní paradigmata. Tato architektura je testována na realistických simulacich pro řešení složitých úkolů v reálném světě. Efektivita nového vysokoúrovňového plánovače založeného na programování s omezenými podmínkami a konečných automatech je demonstrována v praktické aplikaci.

Klíčová slova: Robotika, Neuronové sítě, Učení posilovaním, Programování s omezujícími podmínkami.

Title: Control algorithms for autonomous embodied agents

Author: Mgr. Stanislav Slušný

Department: Department of Software Engineering, Faculty of Mathematics and Physics, Charles University, Prague

Supervisor: Mgr. Roman Neruda, CSc., Institute of Computer Science, Academy of Sciences of the Czech Republic, Prague

Abstract: This work studies control algorithms for adaptive embodied agents. The available approaches, based on neural networks, genetic algorithms and reinforcement learning are investigated and potential improvements suggested. Architecture of adaptive embodied autonomous agents, that combines the existing reactive and deliberative paradigms, is proposed and demonstrated in a realistic simulator solving a complex real world task. The performance of a novel high-level planner based on constraint programming and finite automata is demonstrated on a practical application.

Keywords: Robotics, Neural networks, Reinforcement learning, Constraint programming.

Contents

Introduction

Motivation

Ever since the Czech novelist Karel Čapek formulated the term "robot", intelligent machines, which make life pleasant by doing the type work we don't like to do, has been an active dream for humankind. While much of robotics is still in its infancy, robots are moving from factories and assembly lines into households. An assembly line is orders of magnitude more predictable than a private home. Robotics is becoming a software science, where the goal is to develop robust software that enables machines to withstand the numerous challenges arising in unstructured and dynamic environments.

The idea of "intelligent" devices has an enormous potential to change society and the design of intelligent embodied agents represents one of the key research topics of modern artificial intelligence. Learning is often viewed as an essential part of an intelligent system. Adaptive agents modify their behavior according to acquired knowledge and environmental changes. Learning enables agents to perform more effectively over time. The ultimate goal of the process is to develop an embodied and autonomous agent with a high degree of adaptive possibilities. Since birth of the research discipline[1], two dominant approaches have emerged.

In 1955, Marvine Minsky indicated that an intelligent machine *"would tend to build up within itself an abstract model of the environment in which it is placed. If it were given a problem it could first explore solutions within the internal abstract model of the environment and then attempt external experiments."* This approach, characterized by a strong dependence upon the use of representational knowledge and deliberative reasoning for robotic planning, then dominated robotic research for the next thirty years.

In 1987, the results of Rodney Brooks captured the attention of researchers around the world. According to Brooks, *"planning is just a way of avoiding figuring out what to do next"*. In contrast to the deliberative approach, he emphasized reactive agents working in dynamic, noisy and uncertain environments. He focused on the intelligent behaviors that arise as a result of an agent's interaction with its environment.

In general, both approaches have limitations when they are considered in isolation. Modern robotic systems, which include mobile platforms for planetary exploration or cars that travel autonomously on highways, incorporate a combination of deliberative reasoning and a lower-level reactive system. These robots are able to cope with fundamental problems, like localization in maps, map building, path planning or collision avoidance. However, their cognitive complexity is still very low.

The study of learning paradigms can shed more light into the problem of designing intelligent autonomous agents. Evolutionary robotics is an ideal framework for synthesizing agents whose behavior emerges from a large number of interactions among their constituent parts. It is the approach that connects robotics with two widely studied disciplines: evolutionary algorithms and artificial neural

[1]The birth of artificial intelligence as a distinct field is generally associated with the Dartmouth Summer Research Conference held in August 1955.

networks.

The evolutionary algorithms represent a stochastic search technique used to find approximate solutions to optimization and search problems. They use techniques inspired by evolutionary biology such as mutation, selection, and crossover.

Artificial neural networks represent one of the approaches that are able to handle the supervised learning problem. The neural network research dates back to the early 1950s, when McCulloch and Pitts defined a formal model of neuron. Although their original motivation was to model neural systems of living organisms, neural networks are applicable in such diverse fields as modeling, time series analysis, pattern recognition, signal processing, and control.

Another tool widely used for developing control mechanisms for embodied agents is reinforcement learning. Unlike supervised learning, it deals with situations when an instant reward of agent actions is not available. Positive or negative behavior patterns of an agent are evaluated on the coarser time scale and this information is used to strengthen successful partial behavior patterns over time.

Automated design methodologies have been exploited mainly in reactive controllers. Reactive control is a technique for tightly coupling perception and actions, typically in the context of motor behaviors, to produce timely robotic responses in dynamic worlds. This completely different approach to robot control is represented by deliberative controllers. Deliberative reasoning systems rely heavily on symbolic representation world models. A robot employing deliberative reasoning requires relatively complete knowledge about the world and uses this knowledge to predict the outcome of its actions. That enables it to optimize its performance relative to its model of the world. In dynamic worlds, the world model may quickly become inaccurate and the outcome of reasoning may be invalid. Therefore hybrid deliberative/reactive robotic architectures have recently emerged combining aspects of traditional symbolic methods, but maintaining the goal of providing the responsiveness, robustness and flexibility of purely reactive systems.

As neither approach is entirely satisfactory considered separately, hybrid approach must be taken into account to produce intelligent, robust and flexible robotic controllers. However, the nature of the boundary between deliberation and reactive execution is not well understood at this time, and it is determined by characteristics of the particular environment. As environmental diversity is enormous and may even vary over time, controllers have to continuously adapt their behaviors and react to changes. In other words, in order to become ubiquitous, robots have to learn.

Goals and Objectives

The main goal of this work is to study and develop control algorithms for adaptive embodied agents. In particular, incorporation of adaptive elements into existing state-of-art controllers should be examined. The available approaches, based on neural networks, genetic algorithms and reinforcement learning should be investigated and potential improvements suggested. Based on the obtained theoretical and experimental results, architecture of adaptive embodied autonomous agents should be proposed, possibly combining the existing reactive and deliberative

paradigms.

The goal of the work will be achieved by means of the following objectives:

- **Study of controllers for embodied autonomous agents.**

  Traditionally, robot controllers took advantage of deliberative reasoning. They built representations of the outer world, reasoned about it and planned actions accordingly. Behavior based robotic systems demonstrated how reactive systems with relatively poor models of their environment can effectively produce robust performance in complex and dynamic domains. Hybrid deliberative/reactive robotic architectures, which combine both approaches, are prevalent these days. These approaches should be studied and put into comparison. Conditions determining their applicability and superiority in dynamic environments should be defined.

- **Study of adaptive paradigms for autonomous embodied agents.**

  Several adaptive methods have been studied so far. Supervised learning techniques are not suitable for these kind of problems, as there is not typically any presentation of input/output pairs. The supervision of the agent is through the notion of rewards. The agent is told the reward that represents the outcome of its previous actions, but is not told which action would have been in its best long-term interest. Evolutionary robotics tackles the problem through a self-organization process based on artificial evolution. The broad class of reinforcement learning algorithms approaches the problem by extending algorithms originally developed in operations research and statistics communities. The existing methods should be studied in detail, put into perspective and their applicability in robotic controllers should be discussed.

- **Design of a controller for autonomous adaptive agents.**

  Recent advances in robotics have allowed robots to operate in cluttered and complex spaces. However, to efficiently handle the full complexity of the real world tasks, new reasoning strategies for real world tasks are required. To handle changes in the environment we focus on anytime planning algorithms that can update the plan when the goal is modified. The main concern of further research is the creation of adaptive hybrid controllers.

- **Performance evaluation of the proposed methods.**

  The proposed architecture should be demonstrated on a physical robot or in a realistic simulator solving a complex real world task.

Structure of the Work

Let us describe the structure of this work in order to help its reader navigate throughout the book. In general, the organization of this thesis reflects the course of development of our work. It starts with chapters studying adaptive reactive agents, continues with descriptions of hybrid agents exploiting both reactive and deliberative planning, and ends with the chapter describing a novel high-level planner based on constraint programming.

Chapter 1 introduces adaptive reactive agents and presents the two most popular learning paradigms in robotics: evolutionary and reinforcement learning algorithms. These automated design methodologies are intended to work with no or minimal human intervention and to synthetize sensible behavior out of a "tabula rasa". The chapter starts with the description of genetic algorithms (section 1.1) and neural networks, the cornerstones of evolutionary robotics. The mathematical foundation of learning algorithms based on Markov decision processes is presented in section 1.2. While dynamic programming algorithms from subsection 1.2.1 require a model of the environment, reinforcement learning algorithms examined in subsection 1.2.2 work even in cases, when the model of the environment is not known in advance.

The experimental part of the chapter starts with section 1.4. It compares both introduced learning paradigms on a classic benchmark task. A robot, without any prior knowledge, has to learn to navigate and synthetize an obstacle avoidance behavior. In subsection 1.4.4 we hypothesize, that more powerful architectures of neural networks lead to development of better controllers. Section 1.5 demonstrates how minimal changes to fitness function can synthetize more complex behaviors. It deals with a group of autonomous robots, that accomplish a common task in a distributed fashion. Section 1.6 presents an experiment, when the evolutionary algorithm develops surprisingly well performing active learning behavior. In the last section 1.6 we introduce a social-SLAM experiment, in which a robot is taught to move its head and maintain as much certainty about the state of the social environment as possible. Although this time the complexity of the problem forces a human to manually implement the active learning behavior, adaptive algorithms still accelerate the implementation. The chapter ends with discussion of power and limits of reactive agents.

To overcome limitations of purely reactive agents, the hybrid architecture that takes advantage of both reactive and deliberative approaches is introduced in chapter 2. Section 2.1 puts both approaches into perspective and presents the hybrid architecture that we utilized in our experiments. The remaining sections in chapter 2 describe motion planning components, while the discussion of the high-level planner is left the chapter 3. The localization module, responsible for estimating the position of the robot in the known map, is described in section 2.3. The path-planning component based on the value iteration algorithm is studied in the section 2.4. Although path planning algorithms are well established, implementation of these algorithms in low-cost robots is challenging due to their very limited sensory system. The section 2.5 summarizes the performance of the motion planning module within a low-cost E-puck robot.

Chapter 3 describes the deliberative planner and studies its performance on a real life problem. The goal of the robot is to clean out a collection of wastes spread in a building; but under the condition of not exceeding its internal storage capacity and minimizing the covered trajectory. The task is an important variant of the popular vehicle routing problem and we exploited constraint programming to tackle it. The exact mathematical formulation is given in section 3.1. The crucial aspect of the high-level planner in our design is the ability to produce a solution (possibly sub-optimal) in a very short time frame. The constraint programming planner is introduced in section 3.2. We present two approaches. The approach presented in subsection 3.2.1 is inspired by the operations research model, name-

ly by the network flows. A novel constraint model based on finite automata is presented in the subsection 3.2.2. The experimental evaluation and comparison of both models with emphasis on the further adaptation to the robotics domain are discussed in the section 3.3.

The main results of the work are summarized in the last chapter, where several possible directions for future work are outlined.

Related Works by the Author

The results presented in this work have already been published in papers presented at both international and local conferences, and published in journals, or technical reports.

Publications dealing with neural networks and their learning include [68, 65, 67]. The results concerning evolutionary robotics were published in [95, 70, 98, 111, 112, 72]. The reinforcement learning based adaptive algorithms were studied in [73, 96, 99, 66, 69, 100]. And finally, the results of a hybrid agent incorporating a deliberative planner were presented in [103, 102, 97, 14].

1. Adaptive Reactive Agents

One of the key questions of artificial intelligence is how to design adaptive agents. In order to gain an understanding of a phenomenon as complex as natural intelligence, we need to study complex behavior in elaborate environments. Traditionally, AI has concerned itself with complex agents in relatively simple environments, simple in the sense that they could be precisely modeled and involved little or no noise and uncertainty. One of the many characteristics of intelligence is that it arises as a result of an agent's interaction with intricate environments. In contrast to traditional systems, reactive and behavior based systems have placed agents with low levels of cognitive complexity into noisy and uncertain environments.

Several techniques have been studied so far. Note, that supervised learning techniques are not suitable for these kind of problems, as there is not typically any presentation of input/output pairs. One approach to develop autonomous intelligent agents, called *evolutionary robotics* (ER), is through a self-organization process based on artificial evolution. It is an ideal framework for synthesizing agents whose behavior emerge from a large number of interactions among their constituent parts [76]. In our work, the control systems of evolutionary robots are artificial neural networks, but they might be parameters of predefined control programs, computer programs themselves or learning rules.

Evolutionary algorithms can be viewed as an alternative to classical optimization techniques, based on a biological metaphor: over many generations, natural populations evolve according to the principles of natural selection and "survival of the fittest", first clearly stated by Charles Darwin in The Origin of Species [30].

Artificial neural networks (NNs) are a computational paradigm modeled on the human brain that have been applied to a variety of classification and learning tasks for a few reasons. Despite their simple structure, they provide very general computational capabilities and they can adapt themselves to different tasks, i.e. they are able to learn. Neural networks are popular in robotics for various reasons. They provide straightforward mapping from input signals to output signals, several levels of adaptation and they are not sensitive to noise. The network usually provides direct mapping between robot sensors and effectors, i.e. from the robot sensor values to the actual speeds of robot wheels.

Most current ER applications use traditional multi-layer perceptron networks. We will utilize classical multi-layer perceptron networks in our experiments, too, but we will also compare their performance to recurrent Elman networks and local unit network architecture called Radial Basis Function (RBF) networks. RBF networks have competitive performance, more learning options, and (due to their local nature) better interpretation possibilities.

Another possibility is to apply one of the well studied algorithms of *reinforcement learning*. The supervision of the agent is through the notion of rewards. The agent is told the reward that represents the outcome of its previous actions, but is not told which action would have been in its best long-term interest. The agent interacts with the environment with its sensors and effectors (motors) in a discrete time perception-action cycle and we assume that it is able to sense rewards from the environment. The goal of learning algorithm is to find a prescription that defines the agent's behavior. This behavior is usually defined as a

function that maps the agent's internal state to some action.

1.1 Adaptation Based on Genetic Algorithms

Evolutionary robotics is a research discipline that uses evolutionary computation to develop controllers for autonomous robots. The pioneering work dates back to the late 80s, when the first simulated artificial organisms with a sensory motor system began evolving on computer screens. In 1992, a group of researchers surrounding Floreano and Mondada at the Swiss Federal Institute of Technology in Lausanne, reported promising results from experiments on artificial evolution of autonomous robots and the success of this research triggered a wave of popularity in labs and universities around the world. The book [76] summarizes their early results and provides a comprehensive introduction into the field of evolutionary robotics. Teams at the University of Sussex at Brighton and at the University of Southern California formed the field of ER, too. Since then, ER has been reviewed in various publications [54, 42, 56, 58].

Not surprisingly, evolutionary robotics is based on *Genetic Algorithms* (GA). Genetic algorithm has been investigated by Holland [44], and Fogel [37], among others, with a marked increase in interest within the last decade. Genetic search refers to a class of stochastic optimization techniques — loosely based on processes believed to operate in biological evolution — in which a population of candidate solutions evolve under selection and random "genetic" diversification operators.

The majority of experiments in evolutionary robotics have resorted to neural networks for the control system of the evolving robot. This choice is often justified by one or more of the following issues:

1. Neural networks offer a relatively smooth search space. In other words, gradual changes to the parameters defining a neural network (weights, time constants, architecture) will often correspond to gradual changes of its behavior.

2. Neural networks provide various levels of evolutionary granularity. One may decide to apply artificial evolution to the lowest level specification of a neural network, such as the connection strengths, or to higher levels, such as the coordination of predefined modules composed of predefined sub-networks.

3. Neural networks provide a straightforward mapping between sensors and motors. They can accommodate continuous (analog) input signals and provide either continuous or discrete motor outputs, depending on the transfer function chosen.

4. Neural networks are not sensitive to noise. Since their units are based upon a sum of several weighted signals, oscillations in the individual values of these signals do not drastically affect the behavior of the network. This is a very useful property for physical robots with noisy sensors that interact with noisy environments.

Every member of population is called an *individual* and represents a potential solution to a problem. Each individual is assigned a *fitness* that is a measure of

how good a solution it represents. The better the solution is, the higher the fitness value. The problem is to find the maximum of *fitness function*. The evolution starts from a population of (usually completely random) individuals and iterates in generations. The population evolves toward better solutions (in terms of a fitness function). In each generation, the fitness of each individual is evaluated, and genetic operators (usually selective reproduction, crossover, and mutation) are applied to generate a new population. The new population is then used in the next iteration of the algorithm. This generational process is repeated until a desired individual is found, or until the best fitness value in the population stops increasing.

Algorithm 1 reveals the framework of genetic algorithm used in our experiments. There is a wide variety of components that can be plugged into it and form the final scheme of the algorithm. To fully describe the genetic algorithm, we have to define:

- How potential solutions are represented.
- How the initial population is created.
- The form of the fitness function.
- The genetic operators.

In our experiments, the initial population is formed randomly. The exact form of the fitness function depends on the specific experiments and will be discussed in experimental part of this chapter. We utilize traditional *crossover* and *mutation* operators. The crossover operator composes a pair of new individuals combining parts of two old individuals. First, a crossover point is randomly chosen in both individuals, and then the corresponding parts of individuals are swapped. The positive effect of the crossover is the creation of new solutions recombining the current individuals. Finally, the mutation operator represents small local random changes of an individual. Both the crossover and mutation are applied with certain probabilities only. The exact form of these operators depends on the type of neural network and will be specified in further sections.

An artificial chromosome (genotype) is a representation of an individual (phenotype) that encodes its characteristics. In our experiments, each individual represents a neural network. The chromosome might encode several variables, such as connection weights of a neural network or distribution of neurons. Several types of encoding techniques (binary, grey coding, real valued, etc.) and alphabets (binary, ternary, etc.) have been used in the literature [116]. The standard genetic algorithm uses binary strings. In such a representation scheme, each parameter is represented by a number of bits of a certain length. Although there are several encoding methods that encode real numbers with different range and precision, a trade off often has to be made between precision and range. Real-world experiments demand big precision, what causes long chromosome for which the evolution process becomes non-efficient. Therefore, a different encoding method is utilized in our experiments. The weights are encoded using a *floating-point encoding*, when the synaptic weights of neural networks are directly encoded on the genotype as a string of real values, so an individual is a vector of floating-point values of network weights.

Input : N: number of individuals in the population
E: number of elits
G_{max}: maximum number of populations
Output: The best found solution of a problem

Start: Create the initial population of N individuals $P(0) = \{I_1, ..., I_N\}$, $i = 0$.

repeat

Evaluation of individuals: To compute fitness function for every individual I, build ANN corresponding to I and initialize the simulated environment. Let the robot (controlled by ANN) freely carry actions for fixed time period. Evaluate the robot by a real value, depending how well it was doing.

Creation of new population $P(i+1)$ **from population** $P(i)$: Create empty population $P(i+1)$.

Selection: Choose E best individuals from population $P(i)$ and move them to the population $P(i+1)$. Apply the selection operator, choose $N - E$ individuals and insert them into the population $P'(i)$.

Reproduction: Apply the crossover and the mutation operators on population $P'(i)$, resulting population is $P(i+1)$.

Crossover step: If population $P'(i)$ contains odd number of individuals, insert chromosome of the first individual from population $P'(i)$ into population $P''(i)$. Choose pairs of chromosomes C and D from population $P'(i)$ and apply on them crossover operator, insert new chromosomes C' and D' into population $P''(i)$.

Mutation step: Apply mutation operator on every chromosome from population $P''(i)$, insert new chromosome into population $P(i+1)$.

New generation: $i = i + 1$.

until $i = G_{max}$
Finish and return the solution represented by the individual with the maximal value of fitness function.

Algorithm 1: Skeleton of the evolutionary algorithm.

Neural networks allow different levels of evolutionary adaptation. The final form of our evolutionary algorithm is based on our previous experiences with the evolution of neural networks [65]. Connection weights are determined by a genetic algorithm, while network architecture is predefined in this work. Adaptation of connection weights is usually realized by some supervised learning algorithm (back-propagation). However, there are several reasons for evolving the values of synaptic weights:

1. A genetic algorithm explores a population of networks, not a single network as in other learning algorithms.

2. There are no constraints on the type of architecture, activation function, etc.

3. It is not necessary to have a detailed specification of the network response for each patterns, as in supervised learning methods. This point is the most important in evolutionary robotics.

The alternative approaches to the floating-point encoding of connection weights include a developmental encoding based on a set of rewriting rules encoded on the genotype [50]. This method can encode quite complex networks on a very compact genotype and is well suited for evolving modular networks. Work [40] has developed this method further by encoding network structures as grammatical trees and employed genetic programming. The blueprint of a network architecture may evolve, its actual structure may develop during the initial stages of the robot life, and its connection strengths may adapt in real time while the robot interacts with the environment [41]. When evolving architectures, it is common practice to encode only some characteristics of the network, such as the number of nodes, the probability of connecting them, the type of activation function, etc. This strategy is also known as *indirect encoding* to differentiate it from *direct coding* of all network parameters [116]. Fine tuning of the weights is usually obtained by applying a learning algorithm to the decoded network.

Each neural network represents a potential robot controller. Evaluation of an individual by a fitness value is carried out in a simulation. To evaluate the individual, a neural network is constructed from a chromosome, an environment is initialized and the robot controlled by the constructed neural network is put into its niche (starting location is usually chosen randomly). The inputs of neural networks are interconnected with the robot's sensors and outputs with the robot's motors. The robot is then left to "live" in the simulated environment for some (fixed) time period, fully controlled by it's neural network. Depending on how well the robot is performing, the individual is evaluated by fitness value. The higher the evaluation, the more successful the robot has been executing a particular task. *Roulette wheel selection*, which performs the equivalent role to natural selection — it chooses individuals for the next population proportionally to their fitness values, was used in all experiments.

In our work, we utilize three neural network architectures:

- Feedforward perceptron networks

- Recurrent neural networks

- RBF networks

We will briefly cover each of them in the subsequent sections.

1.1.1 Feedforward Perceptron Networks

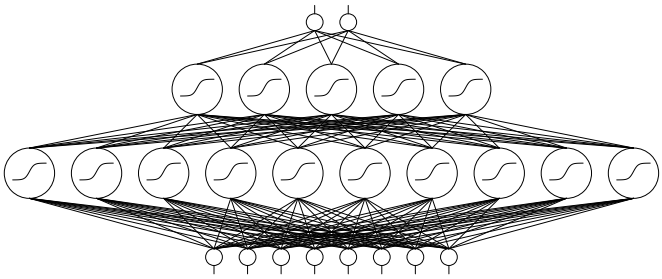


Figure 1.1: Feedforward perceptron network.

A multilayer feedforward neural network (Figure 1.1) is an interconnected network of simple computing units called neurons, which are ordered in layers, starting with an input layer and ending with an output layer [43]. Between these two layers there can be a number of hidden layers. Connections in these kind of networks only go forward from one layer to the next. This way, neural network computes a function that is composed of constituent "semi-linear" functions constrained by network architecture, wherein the constituent functions correspond to neurons.

A multilayer perceptron network (MLP) [43] is one of the most widely used neural networks. The perceptron is a computational unit with n real inputs $\vec{x}$ and one real output y. It realizes the function

$$y(x) = g\left(\sum_{i=1}^{n} w_i x_i\right), \tag{1.1}$$

where x is the neuron with n input dendrites $(x_0 \dots x_n)$, one output axon $y(x)$, $(w_0 \dots w_n)$ are weights and $g : \mathbb{R} \to \mathbb{R}$ is the activation function. We have used one of the most common activation functions, the logistic sigmoid function

$$\sigma(\xi) = 1/(1 + e^{-\xi t}), \tag{1.2}$$

where t determines its steepness.

Weight training in ANN is usually formulated as a minimization of error function, and is carried out by some gradient descent algorithm such as back-propagation, or one of its many variants. Work [59] compares the performance of a genetic algorithm with that of back-propagation on a task of sonar signal classification. The results showed that genetic algorithms find much better networks and in many less computational cycles than back-propagation of error. These results have been confirmed also on a different classification task by other authors [114]. Another strategy consists of combining evolution and supervised learning for the same task. Since back-propagation is very sensitive to initial weight values, genetic algorithms can be used to find the initial weight values of networks trained with back-propagation [15]. The fitness function is computed using the residual

error of the network after having being trained with back-propagation on a given task (notice that the final weights after supervised training are not coded back into the genotype, i.e. evolution is Darwinian, not Lamarkian).

Input : $I_1 = \{w_1^1, \ldots, w_N^1\}$,
$I_2 = \{w_1^2, \ldots, w_N^2\}$
Output: $\{I_1^*, I_2^*\}$

$k_{cross} \leftarrow random(N)$
$I_1^* \leftarrow \{w_1^1, \ldots w_{k_{cross}}^1, w_{k_{cross}+1}^2, \ldots, w_N^2\}$
$I_1^* \leftarrow \{w_1^2, \ldots, w_{k_{cross}}^2, w_{k_{cross}+1}^1, \ldots, w_N^1\}$

Algorithm 2: One-point crossover operator.

Input : $I = \{w_1, \ldots, w_N\}$
Output: I^*

$k \leftarrow random(N)$
$I^* \leftarrow \{w_1, \ldots, w_{k-1}, w_k + \text{random}(-1.0, 1.0), w_{k+1}, \ldots, w_N\}$

Algorithm 3: Additive mutation operator.

To apply the GAs to ANN network learning, we incorporated one-point crossover (Algorithm 2) and additive mutation (Algorithm 3) operators into Algorithm 1.

1.1.2 Recurrent Neural Networks

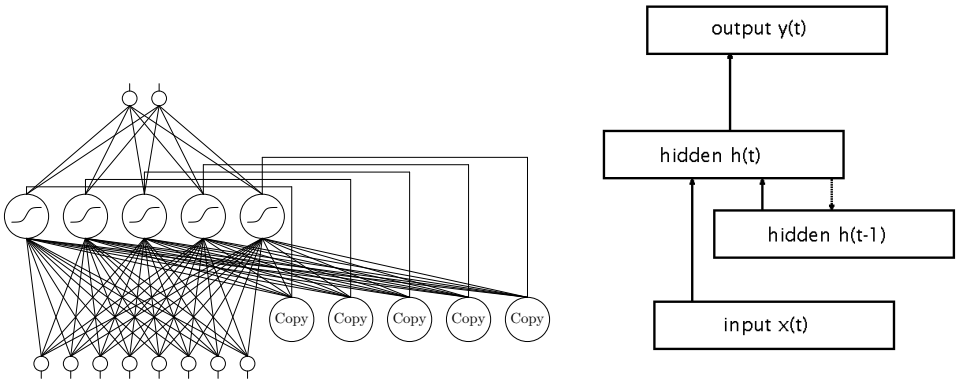


Figure 1.2: **Left:** Elman's network. **Right:** Scheme of layers in the Elman's network.

A feedfoward network architecture has a connectivity structure that is an acyclic graph. However, in recurrent neural networks, along with the feedforward connections, recurrent connections can occur.

In 1990, Elman [34] introduced the *simple recurrent neural network (Elman network)*, which is a widely-used recurrent neural network (Figure 1.2). Elman

networks are used in a number of fields, including cognitive science, psychology, economics and physics. The recurrent connections hold a copy (memory) of the activations at the previous time step. Since hidden units encode their own previous states, this network can detect and reproduce long sequences in time. The sequence of operations is as follows:

- Compute hidden unit activations using net input from input units and from the copy layer
- Compute output unit activations as usual based on the hidden layer.
- Copy new hidden unit activations to the copy layer.

The operators sketched in Algorithm 2 and Algorithm 3 were utilized to produce new offsprings in the recombination step of Algorithm 1.

1.1.3 RBF Networks

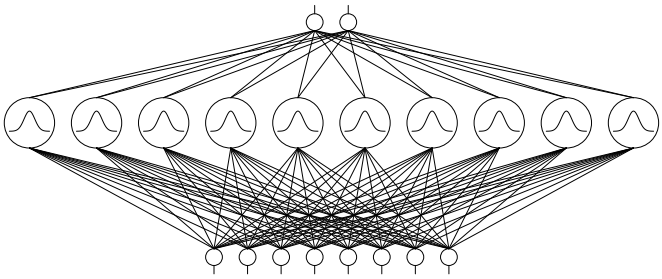


Figure 1.3: RBF network.

An *RBF neural network* [23, 61, 83] represents a relatively new neural network architecture (Figure 1.3). In contrast with the multilayer perceptrons the RBF network contains local units, which were motivated by the presence of many local response units in human brain.

Other motivation came from numerical mathematics, particularly from the study of interpolation problems, where *radial basis function* were first introduced. A radial function is a function that is determined by its *center* and its output depends only on the distance of the argument from this center. In a 2-d dimensional space with the Euclidean metric, the points with the same output values lay in circles. A radial basis function networks are a method to approximate functions and data by applying "kernel methods" to "neural networks".

Both the biological and numerical motivation meet with the regularization theory that created the theoretical background for the RBF network architecture. The regularization approach with radial stabilizers leads to regularization networks with radial basis functions in their hidden layer. The hidden layer of an RBF network consists of RBF units realizing a particular radial basis function.

Definition. (Radial Function) *Let* $f : \mathbb{R}^d \to \mathbb{R}$ *be a function*

$$f(x) = \varphi(\| x - c \|^2), \tag{1.3}$$

where $\varphi : \mathbb{R} \to \mathbb{R}$ *and* $\| \cdot \|$ *is a suitable norm (typically the Euclidean norm). Then* f *is called a radial function and* c *is called a center.*

By an RBF unit we denote a neuron with n real inputs $\vec{x}$ and one real output y, realizing a radial basis function φ, such as Gaussian.

Definition. (RBF Unit) *An RBF unit is a neuron with multiple real inputs* $x = (x_1, \ldots, x_d)$ *and one real output* y*, realizing a function*

$$y(\vec{x}) = \varphi\left(\frac{\| \vec{x} - \vec{c} \|}{b}\right), \tag{1.4}$$

where φ *is a radial basis function,* $c \in \mathbb{R}^d$ *is a center and* $b \in \mathbb{R}$ *is a width.*

The RBF network [83, 61], used in this work, is a feed-forward neural network with one hidden layer of *RBF units* and linear output layer (Figure 1.3).

Definition. (RBF Network) *An RBF network is a 3-layer feed-forward network with the first layer consisting of* d *input units, a hidden layer consisting of* h *RBF units, and an output layer of* m *linear units. Thus, the network computes the following function:* $f = (f_1, ..., f_s, ..., f_m) : \mathbb{R}^d \to \mathbb{R}^m$*:*

$$f_s(\vec{x}) = \sum_{j=1}^{h} w_{js} \varphi\left(\frac{\| \vec{x} - \vec{c}_j \|}{b_j}\right), \tag{1.5}$$

where $w_{ji} \in \mathbb{R}$ *and* φ *is a radial basis function.*

There are a variety of algorithms for RBF network learning. Their behavior and possibilities of their combinations were studied in [64, 65]. The three most significant algorithms are the *three step learning*, *gradient learning* and *genetic learning*. All the considered learning algorithms assume that the number of hidden units is given in advance.

The learning algorithm that we use for RBF networks was motivated by the commonly used three-step learning. Parameters of RBF networks are divided into three groups: centers, widths of the hidden units, and output weights. Each group is then trained separately. The centers of hidden units are found by clustering (*k-means algorithm*) and the widths are fixed so the areas of importance belonging to individual units cover the whole input space. Finally, the output weights are found by GA. The advantage of such an approach is the lower number of parameters to be optimized by GA, i.e. smaller length of an individual.

To apply the GAs to RBF network learning, one has to devise a suitable way of encoding the parameters and set the genetic operators to work on corresponding individuals. The individual consists of h blocks:

$$I_{RBF} = \{B_1, \ldots, B_h\}, \tag{1.6}$$

where h is a number of hidden units. Each of the blocks contains parameter values of one RBF units:

$$B_k = \{c_{k1}, \ldots, c_{kn}, b_k, w_{k1}, \ldots, w_{km}\}, \tag{1.7}$$

where n is the number of inputs, m is the number of outputs, $\vec{c_k} = \{c_{k1}, \ldots, c_{kn}\}$ is the k-th unit's centre, b_k the width and $\vec{w}_k = \{w_{k1}, \ldots, w_{km}\}$ the weights connecting k-th hidden unit with the output layer. The parameter values are encoded using direct floating-point encoding.

Input : $I_1 = \{B_1^1, \ldots, B_h^1\}$
Output: I_1^*

$k \leftarrow random(h)$
$B_k^* \leftarrow B_k$
for p *in* $B_k^* = \{c_{k1}, ..., c_{kd}, b_k, w_{k1}, ..., w_{km}\}$ **do**
  $\delta \leftarrow random(-1.0, 1.0)$
  $p \leftarrow p + \delta$
end
$I^* \leftarrow \{B_1, \ldots, B_k^*, \ldots, B_h\}$

Algorithm 4: Additive mutation operator for RBF networks.

Input : $I_1 = \{B_1^1, \ldots, B_h^1\}$,
$I_2 = \{B_1^2, \ldots, B_h^2\}$
Output: $\{I_1^*, I_2^*\}$

$k_{cross} \leftarrow random(h)$
$I_1^* \leftarrow \{B_1^1, \ldots B_{k_{cross}}^1, B_{k_{cross}+1}^2, \ldots, B_h^2\}$
$I_1^* \leftarrow \{B_1^2, \ldots, B_{k_{cross}}^2, B_{k_{cross}+1}^1, \ldots, B_h^1\}$

Algorithm 5: One-point crossover operator for RBF networks.

1.2 Adaptation Based on MDPs

The algorithms based on dynamic programming [17] have been studied for more than 50 years already and they have solid theoretical backgrounds built around Markov chains and several proved fundamental results. On the other side, as we will see shortly, it is often very difficult to fulfill theoretical assumptions of these algorithms in the experiments. Unlike supervised learning problems, there are no labeled examples of correct and incorrect behavior. However, a reward signal can be perceived.

If a complete model of the environment is available, dynamic programming can be used to teach an agent the optimal strategy. If a model is not available, an optimal value function can be learned from experience via model-free techniques, such as temporal difference learning, which combine elements of dynamic programming with Monte Carlo estimation.

The general model of agent-environment interaction is modeled through the notion of *rewards*. The essential assumption states that an agent is able to sense rewards coming from the environment. Rewards evaluate taken actions and the agent's task is to maximize them. The next assumption is that the agent is working in discrete time steps. We will denote a set of *states* by symbol S and a set of *actions* by symbol A. In each time step t, the agent determines its actual state and chooses one action. Therefore, the life of the agent can be written formally as a sequence

$$s_0 a_0 r_0 s_1 a_1 r_1, \cdots \tag{1.8}$$

where $s_t \in S$ denotes state, which is determined by processing sensors input, $a_t \in A$ action and finally symbol $r_t \in R$ represents reward, that is received at time t.

The fundamental assumption of these algorithms is the *Markov property*, which states, that the agent does not need the history of previous states to make decisions, but the decision of the agent is based on the last state s_t only. When this property holds, we can use a theory from the field of *Markov decision processes* (MDPs) [84]. Originally developed in the operations research and statistics communities, MDPs are now commonly used in artificial intelligence and robotics communities. The term MDP comes from operations research literature, sometimes *Markov decision task* (MDT) is used, too.

Most physical environments have infinite state sets and are continuous time systems. However, tasks faced by agents embedded in such environments can frequently be modeled as MDPs by discretizing the state space and choosing actions at some fixed frequency. It is important to keep in mind that it may be possible to represent the same underlying physical task by several different MDPs, simply by varying the resolution of the state space or by varying the frequency of choosing actions.

Another strong assumption is that the state of the environment is determined by agent's perception, an assumption that may not be satisfied in some real-world tasks with embedded agents.

Although general MDPs may have infinite (even uncountable) state and action spaces, we will only work with finite discounted infinite-horizon MDPs in this thesis, defined as a tuple (S, A, P, R, γ). The role of symbol γ will be explained shortly.

Definition. (A Discounted Infinite Horizon Markov Decision Process) *A discounted infinite horizon Markov decision process is a tuple* (S, A, R, P, γ)*, where*

- S *is a finite set of states,*
- A *is a finite set of actions,*
- R *denotes reward probabilities, where*

  $$P(r_{t+1}|s_t, a_t, s_{t-1}, a_{t-1}, \ldots, s_0, a_0)$$

  denotes expected reward given history

  $$\{s_t, a_t, s_{t-1}, a_{t-1}, \ldots, s_0, a_0\}$$

- P *denotes state transition probabilities, where*

  $$P(s_{t+1}|s_t, a_t, s_{t-1}, a_{t-1}, \ldots, s_0, a_0)$$

  denotes probability of transition to state s_{t+1}*, given history*

  $$\{s_t, a_t, s_{t-1}, a_{t-1}, \ldots, s_0, a_0\}$$

- γ *is discount rate,* $0 \leq \gamma < 1$.

The Markov property of the environment means that we can assume that state transition probabilities and expected rewards depend only on the last state and the last executed action. Formally:

$$P(s_{t+1}|s_t, a_t, s_{t-1}, a_{t-1}, \ldots, s_0, a_0) = P(s_{t+1}|s_t, a_t), \tag{1.9}$$

$$P(r_{t+1}|s_t, a_t, s_{t-1}, a_{t-1}, \ldots, s_0, a_0) = P(r_{t+1}|s_t, a_t). \tag{1.10}$$

This property allows us to to define symbol $P^a_{ss'}$, the transition probability from state s to state s', given the agent chooses action a:

$$P^a_{ss'} = P(s_{t+1} = s'|s_t = s, a_t = a). \tag{1.11}$$

Similarly, we can define symbol $R^a_{ss'}$, the expected reward in state s if agent chooses action a and enters state s':

$$R^a_{ss'} = E\{r_{t+1}|s_{t+1} = s', s_t = s, a_t = a\}. \tag{1.12}$$

The agent's task is to find the *optimal policy* (strategy) π^*, which controls the agent's choice of action in a particular state. In its most general form, the chosen action might depend upon the entire history of the agent; $\pi : (S \times A)^* \times S \to A$. However, in this thesis, we only work with Markov policies, that depend on the last state only. For every MDP (with the optimality criteria defined below), there exists a Markov policy that performs as well as the best full policy.

The optimal policy maximizes some cumulative measure of the payoffs received over time. Ideally, we would like to choose a policy that maximizes the long term sum of immediate rewards. Unfortunately, the naive sum $\sum_{t=0}^{\infty} r_t$ could diverge. Therefore, different optimality criteria are utilized. The number of time steps over which the cumulative payoff are determined is called the *horizon* of the MDP. We will address only agents that have theoretically infinite life-times as these problems are easier to model mathematically. In the case of *finite horizon problems*, the optimal policies might be *non-stationary* (depends on time). As for infinite-horizon discounted MDPs, there exists an optimal policy, which is deterministic and stationary [110], we can define agent's policies as mapping $\pi : S \to A$, where $\pi(s_t) = a_t$. We can ignore the class of *stochastic (soft) policies*, which execute actions with only some probabilities in each state.

The quality of any policy π can be quantified formally through a *value function*. We will define the *value of a state* as the quantity $V^\pi(s_t)$ [76], the expected reward, if the agent starts in state s_t and follows policy π. The symbol γ becomes apparent here. Rather than limiting the distance we look into the future, we discount rewards we will receive in the future by a multiplicative factor, γ, for each time step. Variable γ can be seen as an inflation rate.

Definition. (Value Function) *Let (S, A, R, P, γ) be a discounted infinite horizon Markov decision process. Value function is the function $V^\pi(s_t)$ defined as the discounted cumulative reward of expected future payoffs, computed as:*

$$V^\pi(s_t) = \sum_{i=0} \gamma^i r_{t+i}, \tag{1.13}$$

where discount rate $0 \leq \gamma < 1$ *is a constant that determines the relative value of delayed versus immediate rewards.*

Optimal policy π^* is then the one, that maximizes the expected reward for all states. Even if there are several distinct optimal policies, they all share the same unique optimal value functions.

Definition. (Optimal policy) *Let (S, A, R, P, γ) be a discounted infinite horizon Markov decision process. The optimal policy π^* is the policy, that maximizes the value of all states*

$$\pi^* = argmax_\pi V^\pi(s), \qquad \forall s \in S. \tag{1.14}$$

To simplify the notation, we will write $V^*(s)$ instead of symbol $V^{\pi^*}(s)$, value function corresponding to optimal strategy π^*

$$V^*(s) = \max_\pi V^\pi(s), \qquad \forall s \in S. \tag{1.15}$$

With the framework that we have just introduced, the problem of finding an optimal strategy for embedded agents can be reduced to the problem of finding the optimal policy π^* in MDP. Even within MDP frameworks, the task itself can still be modeled at various levels of complexity:

- Discrete versus continuous MDPs.

  Discrete MDPs are typically easier to solve, as it is difficult to represent continuous actions and state spaces on computers. There is wide variety of methods available to approximate continuous value functions (neural networks, splines, polynomials, RBF networks, support vector machines, decision trees, wavelets, etc.), see [46] for a comprehensive overview.

- Full versus partial observability

  Partially observable Markov decision processes (POMDP) model environments, when the agent does not receive complete state information (as in MDP), but after each step of execution, the agent receives a noisy observation. The observation may be used to determine the agent's state. POMDPs provide more accurate models of environments, at the price of a bigger complexity.

- Model-based versus model-free algorithms

  Model-based algorithms use a model of the environment to update the value function, either a model that is given a priori, or one that is estimated on-line. The model of the environment is formed by transition probabilities and expected rewards. Sometimes the problem of deriving an optimal policy using the full MDP model is known as planning. *Model-free algorithms*, on the other hand, do not use a model of the environment and therefore have no access to the state transitions or the payoff function for the different policies.

- Finite versus infinite horizon problems

  The finite horizon problems may require non-stationary policies that depend on time.

1.2.1 Dynamic Programming

Dynamic Programming [17] (DP) is a collection of algorithms that are based on Bellman's[1] powerful principle of optimality, which states that "an optimal policy has the property that whatever the initial state and action are, the remaining actions must constitute an optimal policy with regard to the state resulting from the first action.". There are two fundamental tasks associated with MDPs [94]:

1. *Policy evaluation* algorithm computes V^π for all states given a MDP with the fixed policy π.

2. In *optimal control*, the goal is to find an optimal control policy π^* for a given MDP.

If we knew to evaluate a policy, the optimal control problem might be solved in a simplistic way by iterating all possible policies for very small MDPs (number of all policies is $|A|^{|S|}$ for finite MDPs). Fortunately, there are better ways. The standard approach to finding the value function for a policy over an MDP is using a recursive formulation (known as the Bellman equation) of the optimality criteria. We define two backup operators based on Bellman's equations.

Definition. (Bellman Policy Evaluation Operator) *Let (S, A, R, P, γ) be a discounted infinite horizon Markov decision process. Bellman policy evaluation operator is a function $B^\pi(s) : S \to \mathbb{R}$ defined as:*

$$B^\pi(s) = \sum_{s'} P_{ss'}^{\pi(s)}[R_{ss'}^{\pi(s)} + \gamma V^\pi(s')], \qquad \forall s \in S. \tag{1.16}$$

According to the Bellman's equations, evaluating policy π requires solving the $(|S| \times |S|)$ linear system of fixed-point equations:

$$V^\pi(s) = B^\pi(s), \qquad \forall s \in S. \tag{1.17}$$

The linear system has a unique solution [110] and could be solved analytically, but it is not feasible for bigger state spaces. We will define similar operator for optimal control and then introduce iterative algorithms based on these operators.

Definition. (Bellman Backup Operator) *Let (S, A, R, P, γ) be a discounted infinite horizon Markov decision process. Bellman backup operator is function $B^*(s) : S \to \mathbb{R}$ defined as:*

$$B^*(s) = \max_a (\sum_{s'} P_{ss'}^a [R_{ss'}^a + \gamma V^\pi(s')]), \qquad \forall s \in S. \tag{1.18}$$

One way to find an optimal policy is to find the optimal value function. According to the Bellman's principle of optimality [17], to determine the value function, we have to again solve the following $(|S| \times |S|)$ **non-linear** system of fixed-point equations:

$$V^*(s) = B^*(s), \qquad \forall s \in S. \tag{1.19}$$

[1] Richard Bellman is the major name associated with dynamic programming. He established the optimality equations that form the basis of dynamic programming.

Note, that both operators B^π and B^* require a model of the environment as they assume knowledge of the state transition probabilities. The former operator B^π is linear while the latter operator B^* is a nonlinear backup operator.

DP provides approximate iterative algorithms for solving both problems. These algorithms use repeated application of update operators to generate an approximation of value function. The updates need not be done in strict order, but instead can occur asynchronously in parallel provided that value of every state gets updated infinitely often on an infinite run. An algorithm is termed *synchronous* [94] if in every $k|S|$ applications of the state-update equation the value of every state in set S is updated exactly k times. Different researchers have used different models of an asynchrony in iterative algorithms. Usually, *asynchronous* algorithms place no constraints on the order in which the state-update equation is applied, except that in the limit the value of each state will be updated infinitely often.

The repeated usage of update operator $B^\pi(s)$ could be used to evaluate a policy π. The approximate iterative Algorithm 6 that uses operator $B^*(s)$ can be shown to converge to the correct V^* values (and therefore solve the optimal control problem). It is called *value iteration* [17].

One problem with value iteration is that it is not obvious when to stop it. In practice, value iteration can be terminated when the expected loss relative to an optimal policy is small enough. The work [115] shows how to bound the regret of value iteration for discounted infinite-horizon problems with the following results. If we define *Bellman residual* of the current value function ϵ as

$$\epsilon = \| V_t - V_{t+1} \|_\infty \tag{1.20}$$

then the value of the greedy policy differs from the value function of the optimal policy by no more than $(2\epsilon\gamma)/(1-\gamma)$. This provides an effective stopping criterion.

Input : (S, A, R, P, γ): Discounted infinite horizon MDP
Output: Value function V such that $regret(PV) \leq \delta$

Initialization: $\forall s \in S$: $V_0(s) \leftarrow 0$
repeat
  [Update step]
  $\forall s \in S$: $V_{t+1}(s) \leftarrow \max_a(\sum_{s'} P^a_{ss'}[R^a_{ss'} + \gamma V^\pi_t(s')])$
  [Compute residual]
  $\epsilon = \max_s |V_t(s) - V_{t+1}(s)|$
  [Determine policy]
  $\pi(s) = argmax_a(\sum_{s'} P^a_{ss'}[R^a_{ss'} + \gamma V^\pi(s')])$
until $2\epsilon\gamma/(1-\gamma) \leq \delta$

Algorithm 6: Value iteration for optimal control problem [94].

The path planner module (see Section 2.4) is based on the presented value iteration algorithm.

1.2.2 Reinforcement Learning

In the previous section we discussed methods for obtaining an optimal policy for an MDP assuming that we already had a model of the environment. Reinforcement learning is primarily concerned with how to obtain the optimal policy when a model of an environment is not known in the advance, and therefore agent cannot use the transition probabilities in its calculations, but it is able to interact with an environment.

A number of model-free (Q-learning [113], Adaptive Heuristic Critic [46], TD(λ) [106]) and model-based (Prioritized-sweeping [62], Dyna [107], Dyna-Q [80]) algorithms have been proposed. The work [46] surveys the field of reinforcement learning and summarizes a broad selection of recent work.

The first breakthrough of RL was the Q-learning algorithm [113] invented by Watkins in his Phd thesis. Watkins proposed the notation $Q(s, a)$ as the *state-action value function* for state-action pair (s, a).

Definition. (State-action Value Function) *Let (S, A, R, P, γ) be a discounted infinite horizon Markov decision process. State-action value function is the function $Q^\pi(s, a)$ defined as the discounted cumulative reward of expected future payoffs, if agent takes action a in state s and follows policy π afterwards:*

$$Q^\pi(s, a) = E(\sum_{i=0}^{\infty} \gamma^i r_i | s_o = s, a_0 = a, \pi), \tag{1.21}$$

where $0 \leq \gamma < 1$ is a discount rate.

Again, to simplify the notation, we will write $Q^*(s, a)$ instead of $Q^{\pi^*}(s, a)$. The relationship between the optimal policy π^* and $Q^*(s, a)$ is then

$$\pi^*(s) = argmax_a Q^*(s, a), \qquad \forall s \in S, \tag{1.22}$$

and the optimal value function $V^*(s)$ can be obtained from $Q^*(s, a)$ by the equality

$$V^*(s) = \max_{a'} Q^*(s, a'), \qquad \forall s \in S. \tag{1.23}$$

In the same way as we wrote Bellman equations for state values, it is possible to rephrase them in Q-notations and relate a state-action value of the optimal value function to optimal state-values which can be reached from that state using a single local transition:

$$Q^*(s, a) = \sum_{s'} P^a_{ss'}(R^a_{ss'} + \gamma V^*(s')), \qquad \forall s \in S, \forall a \in A. \tag{1.24}$$

Again, an iterative DP algorithm can be derived in terms of the Q-values. The value iteration on Q-values would utilize the following rule in the update step of Algorithm 6:

$$Q^{k+1}(s, a) \leftarrow \sum_{s'} P^a_{ss'}(R^a_{ss'} + \gamma \max_{a'} Q^k(s, a')). \tag{1.25}$$

This model of the environment is still required, as the previous equation iterates through all possible next states for state-action pair (s, a). In each step

the Q-function looks ahead one step using recursive update rule. It can be shown that $\lim_{k\to\infty} Q^k = Q^*$, when starting from an arbitrary Q^0 containing only finite values.

We defined Q-values because Q-learning algorithm, one of most commonly used and well-known reinforcement learning algorithms, operates on them. The clever introduction of Q-values let Watkins overcome problem with non-linearity of Bellman Backup operator and prove convergence of the algorithm to the optimal value function. It does not require a model of the environment, but uses only an experience tuple $< s_t, a_t, r_t, s_{t+1} >$ to update Q-values. Q-learning is a form of *temporal difference learning* [106].

On each time step, agent updates its internal representation of Q-values according to the rule

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha_t(r + \gamma \max_{a'} Q(s_{t+1}, a') - Q(s_t, a_t)), \tag{1.26}$$

where α_t is a learning rate and it controls to what extent the newly acquired information will override the old information. In practice, a small constant value is used.

Q-learning is an *on-line algorithm* [94]. It not only learns a value function but also simultaneously controls a real environment, therefore it cases the tradeoff between exploration and exploitation. In other words, it has to choose between executing actions that allow it to improve its estimate of the value function and executing actions that return high payoffs.

Input : S: set of states,
A: set of actions,
R: set of rewards,
γ: discount factor
Output: $Q(s, a)$: table of Q-values

Arbitrary initialize $Q(s, a)$, $\forall(s, a)$.
for *t=0 ...* **do**
  Observe current state s_t by processing sensors
  Choose and execute action a_t (by using ϵ-greedy policy)
  Observe reward r_t
  Observe new state s_{t+1}
  $Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha_t(r + \gamma \max_{a'} Q(s_{t+1}, a') - Q(s_t, a_t))$
end

Algorithm 7: Q-learning algorithm [113].

Q-learning algorithm (Algorithm 7) guarantees convergence to optimal values of $Q^*(s, a)$, if Q-values are represented without any function approximations (in a table), rewards are bounded and every state-action pair is visited infinitely often in infinite run [45]:

Theorem. *For finite MDPs, the Q-learning algorithm converges with probability one to Q^* if the following conditions hold true:*

1. *every state-action pair is updated infinitely often,*

2. Q_0 *is finite (rewards are bounded), and*

3. $\forall(s,a) \in (S \times A); \sum_{t=0}^{\infty} \alpha_t = \infty$ *and* $\sum_{t=0}^{\infty} \alpha_t^2 = \infty$.

To fulfill the first condition, every action has to be chosen with non-zero probability. When the Q-values are nearly converged to their optimal values, it is appropriate for the agent to act greedily, taking the action with highest Q-value. However, during learning, there is a difficult exploration vs. exploitation trade-off. In practice, one can use an ϵ-*greedy* policy, which chooses an action at random with probability ϵ and the action with the maximum expected reward with probability $1 - \epsilon$.

Q-learning is probably the most commonly used algorithm of RL. However, several improvements have been suggested to speed up the algorithm. In classical Q-learning algorithm are Q-values stored in the table. In real life applications, state space is usually too big and convergence toward optimal strategy is slow. In recent years, there have been a lot of efforts devoted to rethinking idea of states by using function approximators [18], defining notion of options and hierarchical abstractions [39]. Relational reinforcement learning [33] is approach that combines RL with *inductive logical programming*. Our work [66] tackles the robot learning problem via relational reinforcement learning.

1.3 Experimental Framework

We start by introducing two popular miniature robot architectures: the older one called Khepera [3] and its successor named E-puck [2]. Robot miniaturization was an important milestone in robotics research as it brought robots from dedicated laboratories to scientists and students and it has brought several advantages [76]. Robot miniaturization makes possible to build complex environments on a limited surface, fundamental laws of physics give higher mechanical robustness to a miniature robot and while the miniature robot usually resists the collision, the other robot would probably report serious damage.

Khepera [3] (Figure 1.4, left) is a miniature mobile robot with a diameter of 70 mm and a weight of 80 g. The robot is supported by two lateral wheels that can rotate in both directions and two rigid pivots in the front and in the back. The sensory system employs eight "active infrared light" sensors distributed around the body, six on one side and two on other side.

E-puck [2] (Figure 1.4, right) is a follower of the Khepera robot, as it follows similar concept. E-puck is a widely used robot for scientific and educational purposes — it is open-source and low-cost. Despite its cheapness and limited sensor system, limited localization can be successfully implemented, as we will show in Chapter 2. E-puck sensors can detect a white paper at a maximum distance of approximately 8 cm. Sensors return values from interval $[0, 4095]$. Effectors accept values from interval $[-1000, 1000]$. The higher the absolute value, the faster is the motor moving in either direction. See Appendix 1 for more information about these robots.

Although experiments on real robots are feasible, they might require a long time, especially when dealing with self-improving systems. As stated in [78], the experiment on evolution of a homing navigation performed by authors took 10

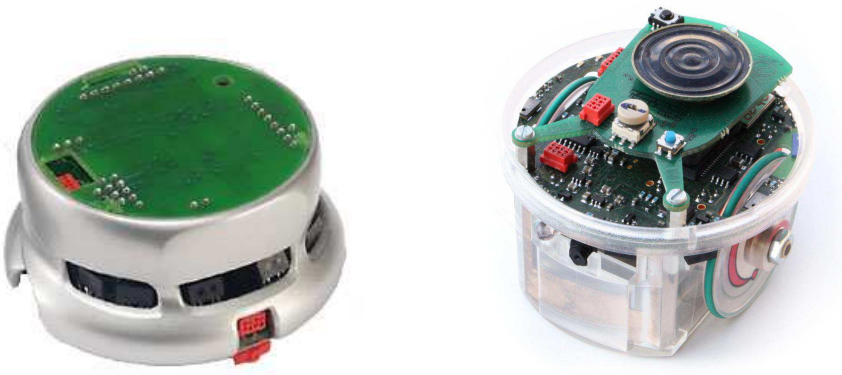


Figure 1.4: **Left:** The first widely used miniature robot Khepera built by K-Team SA (www.k-team.com). **Right:** Follower of Khepera robot named E-puck.

days when carried out entirely on physical robots. One way to avoid the problem of time is to experiment with robots in simulation and then run the most successful individuals on a physical robot. Simulated robots might run faster and the power of parallel computers can be easily exploited for running more individuals at the same time. However, simulation present other problems. There is a real danger that programs which run well on simulated robots will completely fail on real robots because of the differences in real and simulated world sensing. Fortunately several evolutionary experiments conducted in simulations and successfully validated on real robots demonstrated that (at least for a restricted class of robot-environment interactions) it is possible to develop robot controllers in simulation.

One of the widely used simulation software for Khepera robots is *Yaks* [27] simulator, which is freely available. Yaks can also control real robot through serial cable connection, so algorithms successful in simulation can be directly tested on real robot. Simulation consists of predefined number of discrete steps. Yaks provides only some basic functionality (like customizing and designing world with walls or obstacles), and it has been obsoleted by more modern tools.

In our later experiments, we have been using professional commercial development environment *Webots* [4]. Webots provides much more possibilities. It enables to test robot in a physically realistic world and contains models of most popular robotic platforms (including Khepera and E-puck).

1.4 Obstacle Avoidance

We will compare both introduced learning paradigms. There are a wide variety of both GAs and RL methods in use today. However, in order to compare these different approaches empirically we must focus on specific instantiations. We have chosen Q-learning and the standard version of GA, as they have been very popular in other works. They also offer us the obvious practical advantage - their familiarity enables us to use them with confidence and others to easily interpret our results.

No comparison between two learning methods that are parameterized differently can be completely objective. Learning methods usually have some number

of parameters to set, and the amount of time devoted to adjusting them can have a huge impact on the success of learning. Throughout our experiments, we tried to give equal effort to optimizing the parameters of each algorithm. We are mainly interested in the qualitative analysis of evolved behavior and we will not focus on other aspects of learning methods (speed of convergence), that are especially sensitive to parameter settings.

Obstacle avoidance is a classic task that most people working in mobile robotics have some experience with and therefore it is a suitable benchmark task. For its simplest formulation there is already a well known optimal and simple solution: the Braitenberg's vehicle [20]. The work [35] compares a neural controller trained by an evolutionary algorithm with Braitenberg's vehicle. We follow the methodology introduced in [35], but in our experiments, we focus on the analysis of multiple neural architectures and an RL based controller. Both GAs and RL methods have proven effective at developing controllers for small mobile robots. However, since few rigorous empirical comparisons have been conducted, there are no general guidelines describing the methods' relative strengths and weaknesses [108]. We will start with a discussion of the pros and cons of both approaches in this section, and will point out differences in both methods with problems of more complexity.

In our experiments, E-puck was trained to explore the environment and avoid walls. The robot was trained in a simulated environment (we used Webots simulator) of size 100x60 cm and tested in a more complex (previously unseen) environment of size 110x100 cm. The simulation process consisted of a predefined number of steps. In each simulation step, the agent processed sensor values and set speed to the left and right motor. One simulation step took 32 ms.

To simplify the interpretation of the evolved behavior, we reduced the state space. The sensor and effector values were processed before they were delivered to the robot controller. Instead of raw sensor values, learning algorithms worked with "perceptions". Instead of 4095 raw sensor values, we used only 5 perceptions (see Table 1.1). The effector's values were processed in a similar way: instead of 2000 values, the learning algorithm was allowed to choose from values $[-500, -100, 200, 300, 500]$. We also grouped pairs of sensors together. The back sensors were not used at all. This processing reduced the complexity of the task, although learning algorithms do not require it. While neural networks directly support continuous input values, RL methods rely on function approximators in continuous domains, which map state-action pairs to values via parameterized functions and use supervised learning methods to set these parameters. However, human analysis of learned behaviors turns out to be difficult in such cases. The reader is referred to our work [71] for analysis of evolved RBF networks on obstacle avoidance task without described preprocessing of input values.

Pre-processing of sensor values also has another effect. It mostly removes sensor noise and therefore effectively makes task Markovian. Without it, agents could only partially observe the true state of the world and the task would be non-Markovian. In other words, the probability distribution over the next states would not be independent of the agents' state and action histories. This fact can cause problems for RL methods. In practice, though, they still usually perform well even within non-Markovian tasks [105].

Sensor value	Perception	Sensor value	Perception
0-50	NOWHERE	1001-2000	NEAR
51-300	FEEL	2001-3000	VERYNEAR
301-500	VERYFAR	3001-4095	CRASHED
501-1000	FAR		

Table 1.1: Raw sensors values were processed, "perceptions" were then delievered to the robot controller.

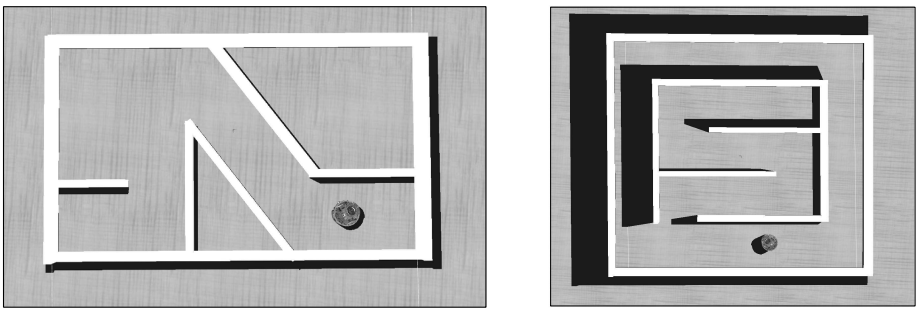


Figure 1.5: Simulated environments for agent training and testing. **Left:** Agent was trained in the environment of size 100×60 cm. **Right:** Testing was performed in a more complex environment of size 110 × 100 cm.

1.4.1 Rules Extracted from RBF Networks

In the first experiment the evolutionary RBF network with 3 input units, 5 hidden Gaussian units, and 2 output units was trained to control the mobile robot. The three inputs corresponded to the coupled sensor values (two left sensors, two front sensors, two right sensors), which were preprocessed as described in Table 1.1. The two outputs corresponded to the left and right wheel speeds and before being applied to the robot wheels they were rounded to one of 5 valid values.

Fitness evaluations consisted of $N_T = 2$ trials, which differed by the agent's starting location (the two starting positions are on opposite ends of the maze). The agent was left to live in the environment for $N_S = 800$ simulation steps. In each simulation step, the agent processed sensor values and set the speed to the left and right motor.

In each step, a three-component score was calculated that motivated the agent to learn to move and to avoid obstacles [35]:

$$T_{k,j} = V_{k,j}(1 - \sqrt{\Delta V_{k,j}})(1 - i_{k,j}).$$

The first component $V_{k,j}$ was computed by summing the absolute values of motor speed (scaled to $\langle -1, 1\rangle$) in the k-th simulation step and the j-th trial, generating a value between 0 and 1. The second component $(1 - \sqrt{\Delta V_{k,j}})$ motivated the two wheels to rotate in the same direction. $\Delta V_{k,j}$ was computed by adding 0.5 to each read value, substracting them together and taking the absolute value of the difference. The value $i_{k,j}$ of the most active sensor (scaled to $\langle 0, 1\rangle$) in the k-th simulation step and the j-th trial provided a measure of the robot distance to the nearest object. The closer it was, the higher the measured value in range from 0.0 to 1.0 was. Thus, $T_{k,j}$ was in range from 0.0 to 1.0.

In the j-th trial, the score S_j, was computed by summing normalized trial gains $T_{k,j}$ in each simulation step:

$$S_j = \sum_{k=1}^{N_S} \frac{T_{k,j}}{N_S}.$$

To stimulate maze exploration, the agent was rewarded when it passed through one of the predefined zones. There were three zones located in the maze. They could not be sensed by an agent. The reward $\Delta_j \in \{0, 1, 2, 3\}$ was given by the number of zones visited in the j-th trial.

The fitness value was then computed as follows:

$$Fitness = \sum_{j=1}^{N_T} (S_j + \Delta_j),$$

thus generating values between 0 and 8.

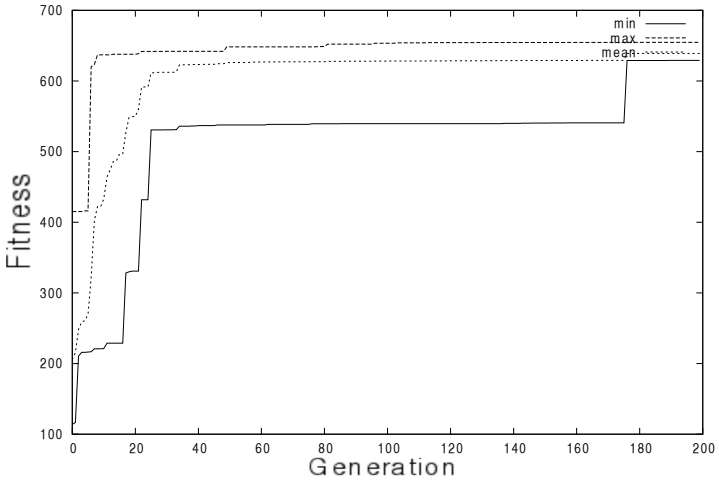


Figure 1.6: The mean, minimal, and maximal fitness function over 10 runs of evolution in the obstacle avoidance experiment. Fitness is scaled in a way that a successful walk through the whole maze corresponds to the fitness 600 and higher.

The experiment with the evolutionary RBF network was repeated 10 times, each run lasted 200 generations. In all cases, the successful behavior was found, i.e. the evolved robot was able to explore the whole maze without crashing into the walls. The average performance (as measured by the fitness function) is depicted in Figure 1.6 (the fitness function is rescaled to an interval 0-800). In a typical run, the robot controller quickly gained the ability to explore the environment and pass through the zones, as indicated by the steepness of the fitness function in 30 generations. In the following generations, the controllers mostly improved their speed and obstacle avoidance ability near walls or corners. This is suggested by a shallow learning curve in the remaining generations.

In work [35] authors analyzed individual terms of the fitness function in isolation to gain a better understanding of the evolved behavior. We exploited superior interpretation possibilities of RBF networks compared to MLP and examined the evolved rules. Table 1.2 shows the parameters of an evolved network

Sensor			Width	Motor	
left	**front**	**right**		**left**	**right**
VERYNEAR	NEAR	VERYFAR	1.56	500	-100
FEEL	NOWHERE	NOWHERE	1.93	-500	500
NEAR	NEAR	NOWHERE	0.75	500	-500
FEEL	NOWHERE	NEAR	0.29	500	-500
VERYFAR	NOWHERE	NOWHERE	0.16	500	500

Table 1.2: Rules, that were learned in the obstacle avoidance experiment, as represented by RBF units (listed values are original RBF network parameters after discretization).

with five RBF units. We can understand them as rules providing mapping from input sensor space to motor control. However, these "rules" act in accordance, since the whole network computes a linear sum of the corresponding five gaussians. To gain a deeper understanding of the evolved behavior, we performed additional analysis of learned controllers. Appendix 2 shows rules from actual run of the robot in the training arena. They were obtained by presenting specific inputs to the robot controller and recording the generated command to robot effectors. The nine most frequently used rules are shown in the latter table. It can be seen that this agent represents a typical evolved left-hand wall follower — a surprisingly effective controller. Straight movement is a result of situations when there is a wall far on the left, or both far on the left and right. If the robot sees nothing, it rotates to the left (rule 2), The front collision is avoided by turning right, as well as a near proximity to the left wall (rules 6–8). As the wall follower represents a very effective strategy for the given task, we started evaluating the performance in previously unseen environment. The evolved robot was tested in a bigger testing maze (Figure 1.5, right). It behaved in a consistent manner, using the same rules, demonstrating generalization of the behavior trained in the former maze.

1.4.2 Rules Induced by Reinforcement Learning

In the second experiment we used Q-learning algorithm. Each state was represented by three perceptions. For example, the state [NEAR, NOWHERE, NOWHERE] means that the robot sees a wall on its left side only. The action was represented by a pair [left speed, right speed]. The learning process was divided into 10000 episodes. Each episode took at most 800 simulation steps. At the start of each episode, the agent was moved to one from 5 randomly chosen positions. The episode could finish earlier, if the agent hit the wall.

The training process is depicted in Figure 1.7. Even though the agent always chose the best action in a testing phase (Figure 1.8), it crashed occasionally. It's limited sensing abilities do not allow it to recover from some difficult situations (corners, sharp edges...). This fact also affected the relatively low speed of induced strategies, as can be seen from the fitness value. However, the representation of inducted knowledge in a table simplified the rule extraction procedure. Table 1.3 contains states with the biggest and smallest Q-values and

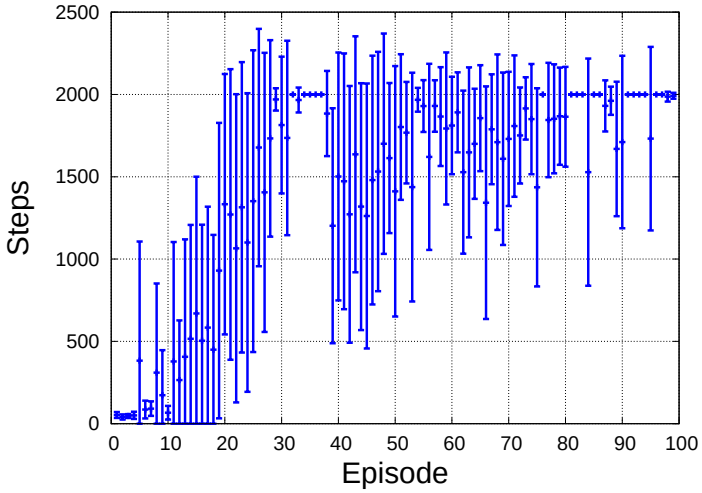


Figure 1.7: Average number of steps (of thirty runs) executed by the agent in the training phase of RL learning.

their best action. The states with the biggest Q-values contain mostly perception NOWHERE. On the other side, the states with the smallest Q-values contain perception CRASHED.

The verification of the synthetized behavior is very straightforward. Learned behavior corresponds to obstacle avoidance behavior. The most interesting are the states, which contain the perception "NEAR". Expected rules "when obstacle left, then turn right" can be found. States without perception "NEAR" were evaluated as safe — even if a bad action was chosen in this state, it could be fixed by choosing a good action in the next state. Therefore, these actions do not tell us a lot about the agent's behavior. On the other hand, an action with the perception VERYNEAR led to a crash, usually. In these instances, the agent was not able to avoid the collision.

State			**Action**	**Q-value**
left	**front**	**right**		
NOWHERE	NOWHERE	VERYFAR	[500, 300]	5775.71729
NOWHERE	NOWHERE	NOWHERE	[300, 300]	5768.35059
VERYFAR	NOWHERE	NOWHERE	[300, 500]	5759.31055
NOWHERE	NOWHERE	FEEL	[300, 300]	5753.71240
NOWHERE	VERYFAR	NOWHERE	[500, 100]	5718.16797
CRASHED	CRASHED	CRASHED	[300, 500]	-40055.38281
CRASHED	NOWHERE	CRASHED	[300, 300]	-40107.77734
NOWHERE	CRASHED	VERYNEAR	[300, 500]	-40128.28906
FAR	VERYNEAR	CRASHED	[300, 500]	-40210.53125
NOWHERE	CRASHED	NEAR	[200, 500]	-40376.87891

Table 1.3: 5 states with the biggest and smallest Q-values and their best actions

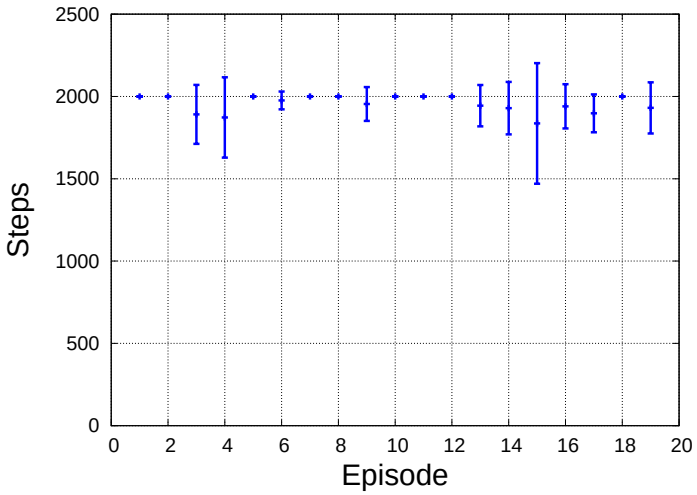


Figure 1.8: The average number of steps (of thirty runs) executed by the agent in the testing phase of RL learning.

1.4.3 Discussion

We have presented two experiments, that show how RL and ER algorithms can train a robot controller to explore an unknown maze. It is known from the literature, and from our previous work [99], that this problem is manageable by both paradigms. We illustrated some important features of these methods on the obstacle avoidance task:

- It is not trivial to extract learned rules from developed controllers. We simplified this process by reducing the state space, but this process also significantly limited the number of the possible controllers. In more realistic scenarios, the researcher has to verify the quality of evolved strategy in various environments. This may be very difficult, especially in the case of big environmental diversity (or if environment is unknown).

- The reactive controller may perform reasonably well on this simple task. The effective policy is to follow the left or the right wall. By analyzing learned rules, we verified, that neuro-evolution synthetized such a controller. However, to our surprise, the RL algorithm did not. We do not conclude that RL algorithm is worse thought. Often, changing some basic environment constraints (dimensions of the environment, for example) can have a suprising effect on the learning process. Slight modifications of the environment are likely to cause a drop in performance. Therefore, deep analysis of evolved behavior may be inevitable in some cases.

- Design of an appropriate fitness function may be a non-trivial task, too. Reactive navigation is considered to be one of the simplest tasks in robotics. It is justified to expect that our automated design methodology would evolve reasonable policies without human intervention or prior knowledge about sensors, motors and environments. Unfortunately, human involvement is still required. The choice of fitness function allows learnability of the system. There must be a path in the search space that can be discovered by

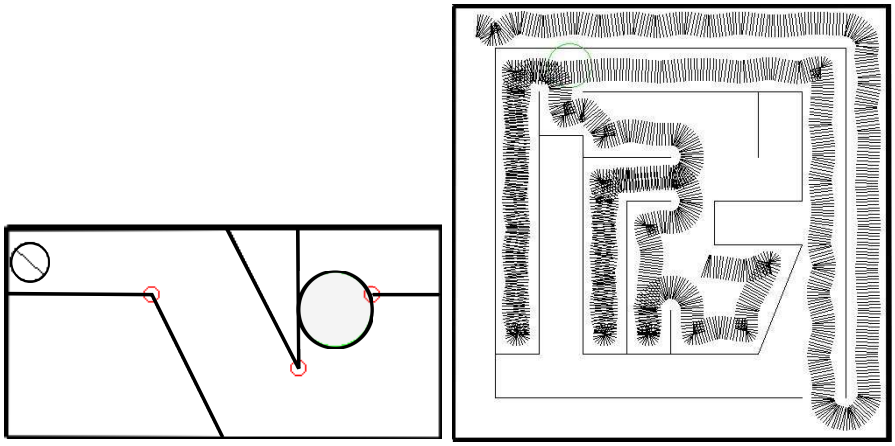


Figure 1.9: **Left:** In the exploration task, the agent is rewarded for passing through the zone, which can not be sensed. The zone is drawn as the bigger circle, the smaller circle represents the Khepera robot. The training environment is 60×30 cm. **Right:** The testing arena 100×100 cm and the trajectory of any evolved individual. The agent's strategy is to follow the wall on it's left side.

the limited sample of potential solutions considered. Although it has been suggested how fitness function for reactive navigation should look [76], we had to introduce zones to motivate the agent to move.

- It may be difficult to fulfill theoretical assumptions of RL algorithms. Even noise, which is ubiquitous in robotics, may break Markov property. Although it has been shown that RL methods work reasonably well even with non-Markovian tasks [105].

1.4.4 Comparison of NN Architectures

The third experiment in this section compares performances of various neural network architectures. We hypothesized that more complex neural network architectures would develop more powerful strategies, perhaps at a price of a slower learning progress. Authors in [79] showed how problems that can not be mastered by simple reactive individuals can be solved by providing evolving robots with more complex neural controllers able to keep trace of previously experienced sensory states.

The Khepera robot was put in a maze of 60×30 cm (Figure 1.9) with a goal to cover the longest possible distance without hitting the objects. The experiment methodology was similar to previous experiments, however, different environments and different parameters of evolutionary algorithm were used and the special processing of sensors and effectors was not involved. The reader is referred to our work [71] for further details.

The most successful individuals could have achieved a fitness value in a range from 4 to 5. The fractional part of the fitness value reflected the speed of the agent and its ability to avoid obstacles, while the integer part expressed the number of randomly distributed "zones" reached in the arena.

All the networks included in the tests were able to learn the task of finding a random zone from four starting positions. The resulting best fitness values

(Table 1.4) were all in the range of 4.3–4.4 and they differed only in the order of a few percent. It can be seen that the MLP networks performed slightly better, RBF networks were in the middle, while recurrent networks were a bit worse in terms of the best fitness achieved. According to their general performance, which took into account ten different EA runs, the situation changed slightly. In general, the networks can be divided into two categories. The first one represents networks that performed well in each experiment in a consistent manner, i.e. every run of the evolutionary algorithm out of the ten random populations ended in finding a successful network that was able to find the zone from each trial. MLP networks and recurrent networks with 5 units fall into this group. The second group has in fact a smaller trial rate because, typically, one out of ten runs of EA did not produce the optimal solution. The observance of average and standard deviation values in Table 1.4 shows this clearly. This might still be caused by the less-efficient EA performance for RBF and Elman networks.

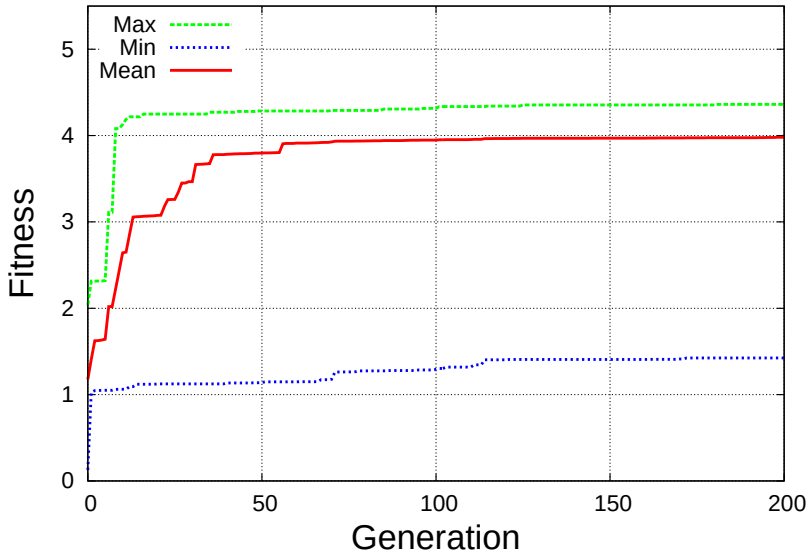


Figure 1.10: The plot of fitness curves in consecutive populations (maximal, minimal, and average individuals) for a typical EA run (one of ten) training the RBF network with 5 units in the exploration task.

The performance of the evolved controller in a bigger maze is depicted in Figure 1.9. Each of the architectures was capable of efficient space exploration behavior that had emerged during the learning to find random zone positions. The above mentioned figure shows, that the robot trained in a quite simple arena and endowed by a relatively small network of 5–10 units, was capable of navigating the testing environment.

The results in terms of the best individuals were comparable for different architectures with reasonable network sizes. The MLP is the overall winner mainly when considering the overall performance averaged over ten runs of EA. The behavior of EA for Elman and RBF networks was less consistent, there were again several runs that obviously got stuck in local extrema (Table 1.4).

In this experiment, neural networks of any of the three types were able to develop the exploration behavior and typical behavioral patterns, such as following the right wall, have been developed, which in turn resulted in the very efficient

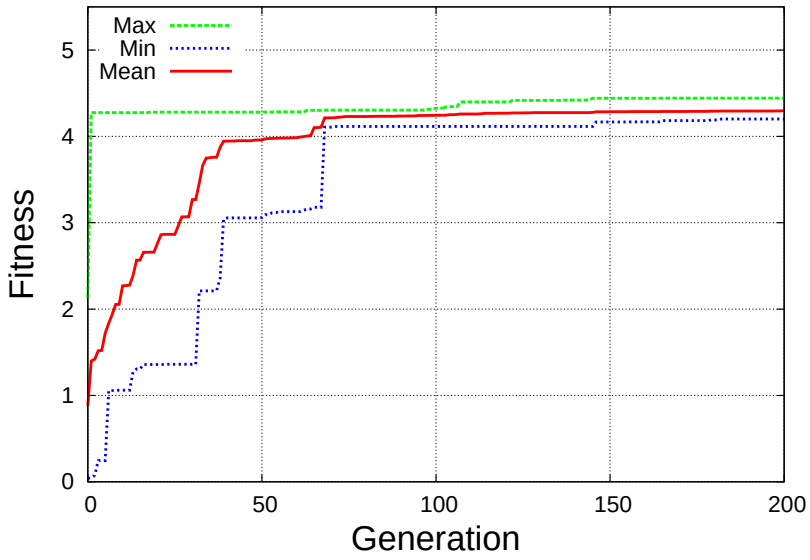


Figure 1.11: The plot of fitness curves in consecutive populations (maximal, minimal, and average individuals) for a typical EA run (one of ten) training the MLP network with 5 units in the exploration task.

Network type	**Exploration task**			
	mean	std	min	max
MLP 5 units	4.29	0.08	4.20	4.44
MLP 10 units	4.32	0.07	4.24	4.46
ELM 5 units	4.24	0.06	4.14	4.33
ELM 10 units	3.97	0.70	2.24	4.34
RBF 5 units	3.98	0.90	1.42	4.36
RBF 10 units	4.00	0.97	1.23	4.38

Table 1.4: Comparison of the fitness values achieved by different types of network in the exploration task.

exploration of an unknown maze. However, the best results achieved by any of the network architectures, were quite comparable with simpler perceptron networks (such as the 5-hidden unit perceptron) marginally outperforming Elman and RBF networks. We did not confirm our hypothesis that more advanced neural network architecture would lead towards the development of better controllers. We made extra sure to run each case for a satisfactory number of generations, so that the learning process plateaued and slower learners were not penalized.

1.5 Collective Behavior

In this experiment, we will show how fitness function from the previous experiment can be extended in an iterative fashion, so that it is applicable to a different problem domain — this time we will work with a group of robots. By changing the fitness function, we redefine controllers at the individual level, but we will

evaluate the performance of the system (consisting of multiple controllers) as a whole. Despite its extensibility, the presented fitness function describes behavior quite precisely. We tried to avoid such detailed specification of fitness function. Ideally, fitness measure should be a survival criterion, that would be translated into sensible behavior by automatic design methodology. However, it is difficult to produce such a high-level fitness function, that is not biased by human view, as in the so-called *bootstrap problem* [49], which usually shows up in non-trivial problems.

Collective robotics deals with groups of fully autonomous robots, which usually accomplish common tasks, such as motion coordination, in a distributed fashion. Several researchers showed [25, 32], that the design of such control systems may be performed by evolutionary algorithms. The pioneering work was done by Martinoli [55]. He solved a task, in which a group of simulated Khepera robots were asked to find "food items" randomly distributed in an arena. The control system was developed by the artificial evolution.

Another example of artificial evolution applied to design coordinated, cooperative behavior for real robots was presented in the work [85]. Artificial evolution was employed to design neural network controllers for small, homogeneous teams of mobile autonomous robots. The robots were evolved to perform a formation movement task from random starting positions, equipped only with infrared sensors.

The work [11] presents a set of experiments in which a group of simulated robots were evolved for the ability to aggregate and to move together toward a light target. Evolved individuals displayed interesting behavioral patterns in which groups of robots acted as a single unit. Moreover, groups of evolved individuals displayed primitive forms of "situated" specializations in which different individuals showed different behavioral functions according to the circumstances. The neural networks used in the experiments were simple feed forward neural networks without hidden neurons.

A related emerging field called *swarm robotics*, studies systems composed of swarms of robots that interact and cooperate to achieve common goals [12]. While swarm robotics emphasizes decentralization of control, in our experiment, there was a central authority controlling the other robots (leader). A group of simulated robots was evolved to show the ability of collective homing behavior. The team leader was equipped with a light source, while the remaining robots had sensors measuring the reflected light intensity.

The environment (Figure 1.12) used in the experiment was a rectangular arena of 80×50 cm surrounded by walls. The group consisted of three Khepera robots each equipped with 8 infrared sensors. The goal of the robots was to find the circular arena randomly located in the environment. The robots could sense the arena with the simulated ground sensor: the sensor returned value 1, if the robot was located in the arena and 0 if it was not. The robot sensed the world with 17 sensors (8 active sensors, 8 passive sensors and 1 ground sensor). Sensor measurements (17 values from the interval $[0, 1]$) served as input to the neural network. The network's output (two values from interval $[0, 1]$) was then used as a command for robot effectors (two motors). All robots were guided by the same neural network, constructed from the chromosome.

The team leader was carrying a simulated light bulb, which was attached to

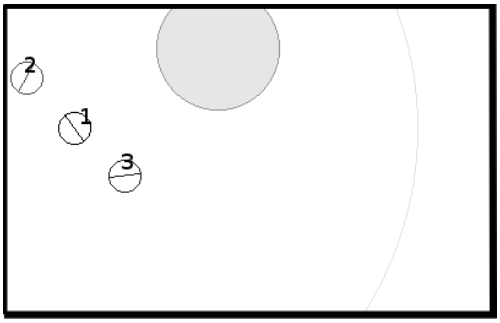


Figure 1.12: The environment for training grouping behavior. The thick lines represent the walls surrounding the arena of 80×50 cm. The full circle represents the target "home" arena. The arc indicates the light intensity, but meaningful values could be measured up to 20 cm with only the robot's sensors. The robot marked with "1" is the team leader, carrying the light source.

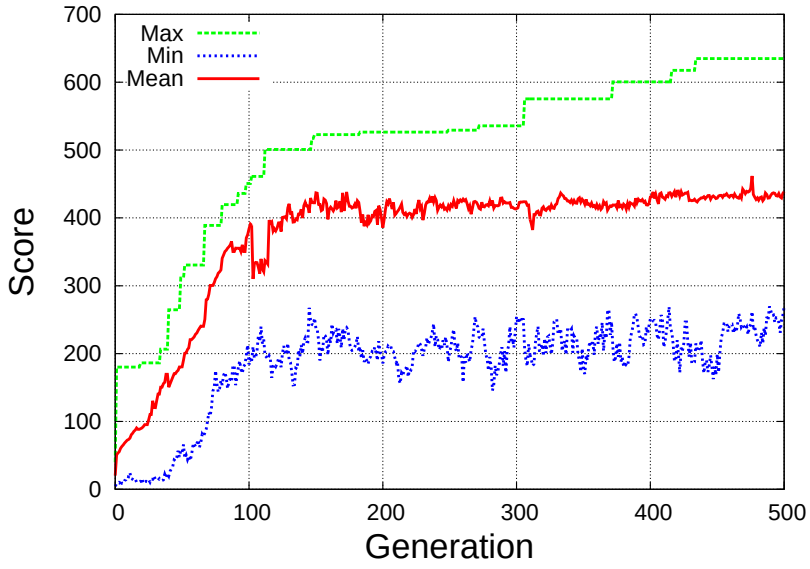


Figure 1.13: The plot of trial scores in consecutive populations (maximal, minimal, and average individuals) for a typical EA run training in the collective experiment. Values bigger than 400 represent well trained individuals, in which the grouping behavior is clearly present.

the top of the leader's body at a the height of 3 cm. The remaining team members could use information from the passive sensors to produce "following behavior" - the emitted light could be detected up to a distance of 20 cm.

To evaluate the individuals, simulation was launched 10 times (we call "trials" the individual runs). In each trial, the environment was initialized and the starting position of robots was randomly chosen.

The robots were "put" in the simulated environment for 500 simulation steps, fully controlled by the neural network. As soon as any robot hits a wall or another robot, simulation was stopped. Depending on how well the robots were performing, the individual were evaluated by "trial score". The higher the trial score, the more successful the robots were in executing the task in a particular

trial. The fitness value was then obtained by summing up all the trial scores.

Let's denote by $T_{k,j}$ the trial score in k-th simulation step and j-th trial, generating value between 0 and 2, where

$$T_{k,j} = L_{k,j} M_{1,k,j} M_{2,k,j}.$$

$T_{k,j}$ is computed as a product of the evaluation of the leader ($L_{k,j}$) and team members ($M_{1,k,j}$ and $M_{2,k,j}$):

$$L_{k,j} = V_{k,j}(1 - \sqrt{\Delta V_{k,j}})(1 - i_{k,j}) + Z_{k,j}.$$

$L_{k,j}$ is the component responsible for the evolution of obstacle avoidance behavior. The first component $V_{k,j}$ was computed by summing the absolute values of motor speed of the team leader in k-th simulation step and j-th trial, generating a value between 0 and 1. $\Delta V_{k,j}$ was computed by adding 0.5 to each read value, substracting them together and taking the absolute value of the difference. The value $i_{k,j}$ was the value of the most active sensor in k-th simulation step and j-th trial. Finally, $Z_{k,j}$ held value a 1, if the robot was in k-th simulation step and j-th trial in the target area, and 0 otherwise. Thus, $L_{k,j}$ was in range from 0 to 2.

$M_{i,k,j}$ is the term controlling behavior of the remaining team members:

$$M_{i,k,j} = (1 - D_{k,j}(i, 0)),$$

where value $D_{k,j}(i, 0)$ is a normalized distance of robot i to the team leader, computed as follows:

$$D_{k,j}(i,j) = \begin{cases} 1 & : \quad dist(i,j) > 200\text{mm} \\ \frac{dist(i,j)}{200} & : \quad dist(i,j) \leq 200\text{mm} \end{cases}$$

Therefore, values of $M_{i,k,j}$ are in the interval $< 0, 1 >$. In the j-th trial, the score S_j was computed by adding trial gains $T_{k,j}$ in each simulation step, generating a value between 0 and 1000:

$$S_j = \sum_{k=1}^{500} T_{k,j}.$$

The evolutionary algorithm was stopped after the 500th generation. Starting from approximately the 200th generation, the best individuals were able to find the target area and they kept increasing the driving speed. This is demonstated in Figure 1.13, which shows a typical trial curve of the population. Starting from approximately the 300th generation, the best individuals were able to find the target area in all ten trials, In the remaining generations, the robot increased mostly in speed and the group became more compact. Since this point, the learning process has plateaued and the desired behavioral patterns have started to dominate in the population. However, the group achieved only 40% of the speed of the best controllers trained in the previous experiment. This should come as no surprise, given the more difficult working conditions.

The robot team, corresponding to the best individual, clearly showed grouping behavior — the distance between team members did not get over 20 cm and the team leader was clearly guiding the group. The remaining robots followed the

team leader, keeping a safe distance (Figure 1.14, left). If the distance between robots was too small, the robots crashed or team members lost trace of the team leader (obviously, a rapid change in direction by team leader caused the problems for the remaining team members).

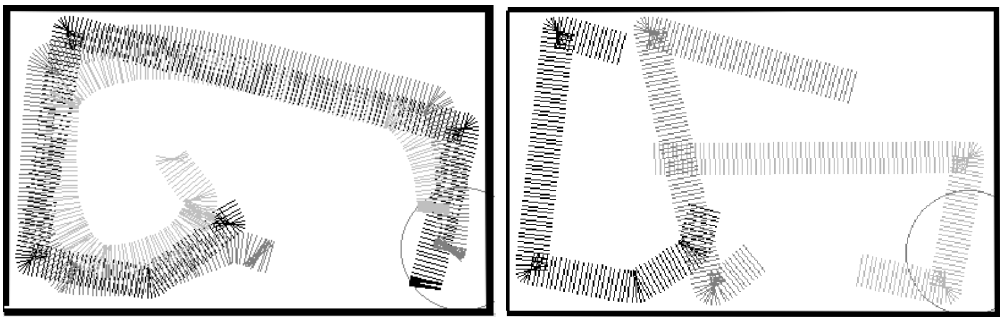


Figure 1.14: **Left:** Typical run of the best individual from the 500th generation. Three robots aggregated together and were guided by the team leader (the robot in the middle), which carried the light source. After reaching the target area, they did not leave it. **Right:** Testing trained behavior when no light source was in the environment. No group behavior was present, the robots were exploring the environment searching for the light.

To prove their ability to follow the team leader, the robots were put into the environment without a light source (Figure 1.14, right). In these conditions, the grouping behavior was not present: each robot explored the environment on it's own. Whenever the light bulb was switched off during the experiment, the robots scattered. The robots even ignored the target arena. After switching the bulb on again, the group formation was slowly rebuilt. Although the robots were guided by identical neural networks, task specialization was clearly present: the team leader was responsible for exploring the arena, while the remaining team members follow him. The whole group acted as a single unit.

As shown, grouping behavior emerged through the evolutionary process. The robots did not know their relative positions. Although the team leader could have been recognized because of the light bulb, it was impossible for the robots to discriminate between the wall and another robot, as they were only using infrared sensors. Despite these limitations, the evolutionary algorithm successfully solved the task. A single neural network was able to guide the group of three robots.

As demonstrated, ER can cope with more complex problems. Although, the design of the fitness function is not trivial, it can be found by iterative process, when a researcher starts with a simple well-known task. The fitness measure is then extended by terms that describe additional behavioral patterns (while eliminating unwanted behavior). This approach usually leads to so-called *behavioral fitness function* [63]. Behavioral fitness functions generally include several sub-functions that are combined into a weighted sum or product. These sub-functions (behavioral terms) are intended to measure simple action-response behaviors or low-level sensor-effector mappings. Ideally, we would like to use a more high-level fitness function that specifies what the robot should accomplish, without specifying how. The experiment in the next section uses a type of fitness function, called *aggregate fitness function* [63]. Aggregate fitness functions select only for

high-level success or failure to complete a task without regard as to how the task was completed. This type of selection reduces the injection of human bias into the evolving system. Controllers based on aggregate fitness functions often face the so-called bootstrap problem [49]. Completely aggregate selection produces no selective pressure in sub-minimal competent populations at the beginning of evolution and hence the process cannot get started.

In the next section, we will demonstrate example of an aggregate fitness function in an experiment, in which the evolutionary process managed to find surprisingly effective behavior.

1.6 Active Learning

The term *active learning* [90] applies to a wide range of situations in which a learner is able to exert some control over its source of data. In robotics, the robot has to actively select maximally informative actions. In this section, we will introduce an experiment, that illustrates (the reader is referred to our work [101] for more details) how an agent can utilize active learning to distinguish between strongly overlapping sensory patterns that require different motor answers. This process of selecting sensory patterns, which are easy to discriminate through motor actions, is usually referred to as active perception [75]. The first experiment demonstrates active learning, which is naturally present in an evolutionary setting, while the second experiment shows active learning cleverly incorporated into a reinforcement learning algorithm by a researcher. We will compare both approaches and outline their differences.

The first experiment follows a study originally carried out by Nolfi [74]. A Khepera robot controlled by a neural network has to discriminate between the sensory patterns produced by walls and small cylinders. As the agents' sensory system is very limited, this problem turns out to be a difficult one. Passive networks (i.e. networks which are passively exposed to a set of sensory patterns without being able to interact with the external environment through motor action) are mostly unable to discriminate between these objects [76]. However, agents that are left free to move in their environment obtain much better results. This experiment demonstrates the advantage of embodied agents, which can use active learning to avoid highly overlapped input spaces.

In our setting, the Khepera robot was allowed to sense the environment with only six frontal infrared sensors, which provided it with limited information. The fitness evaluation consisted of five trials and the individual trials differed by the robot's starting location. The robot was left to live in the environment for 500 simulation steps. In each simulation step, the trial score was increased by 1, if the robot approached the cylinder, or 0.5, if the robot approached the wall. The fitness value was then obtained by summing up all trial scores. The environment was an arena of 40×40 cm surrounded by walls.

The best individuals incorporated an ability to avoid walls very quickly, then to explore the environment and stay in the vicinity of the cylinder. It should be noted that complex obstacle avoidance behavior was not evolved. The evolution process exploited arena constraints and the relative simplicity of the training environment.

This task was solved quite easily by simple neural network architectures. It

Network	Wall and cyllinder			
	mean	std	min	max
MLP-5	2326.1	57.8	2185.5	2390.0
MLP-10	2331.4	86.6	2089.0	2391.5
ELM-5	2250.8	147.7	1954.5	2382.5
ELM-10	2027.8	204.3	1609.5	2301.5
RBF-5	1986.6	230.2	1604.0	2343.0
RBF-10	2079.4	94.5	2077.5	2359.5

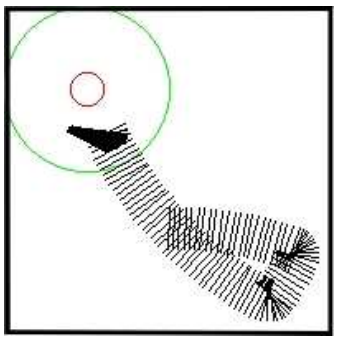


Figure 1.15: **Left:** Comparison of the fitness values achieved by different types of networks in the wall and cylinder task. **Right:** Trajectory of an agent doing the wall and cylinder task. The small circle represents the target cylinder. The agent is rewarded in the zone represented by a bigger circle. It is able to discriminate between the wall and cylinder, and after discovering the cylinder it stays in its vicinity.

may seem surprising, given the similarity of walls and cylinders. The advantage of embodied agents is clearly demonstrated here. Due to the sensor limitations of the agent, this task requires a synchronized use of a suitable position change and simple pattern recognition. This problem seems to be more difficult than the obstacle avoidance behavior (partially due to spare reward function), nevertheless, most of the neural architectures were able to locate and identify the round target regardless of its position.

The results in terms of the best individuals are again quite comparable for different architectures with reasonable network sizes. The differences are more pronounced than in the case of the previous tasks though. The MLP is the overall winner, mainly when considering the performance averaged over ten runs of EA. We hypothesized that recurrent networks might evolve new behavioral patterns, as this task seems to be suitable for that, but we were unable to confirm this hypothesis. The behavior of EA for Elman and RBF networks was less consistent. There were, again, several runs that obviously got stuck in local extrema (Table 1.15). This is in line with the results of work [74]. The authors reported that networks without internal units were able to solve the task perfectly well. As a consequence, the addition of internal units (recurrent networks were not considered) did not produce any improvement in performance, but even resulted in less efficient individuals. This might be explained by longer genotypes that enlarged the space to be searched by GA.

Even more interesting than the performance of specific neural network architectures, is the fact that the active learning behavior was naturally synthetized without human intervention in this task. The researcher did not have to integrate it into fitness function (or perhaps even know about it). The agent alone "realized" that it is beneficial to reduce uncertainty in its environment, and adjusted its strategy accordingly. The presented fitness function is an example of aggregate fitness function [63]. The robot was only rewarded if it reached some subgoal, without specifying the steps that would lead to it. The evolutionary process did not suffer from the bootstrap problem in this experiment. The bootstrap problem is often recognized as one of the main challenges of evolutionary robotics: if all individuals from the first randomly generated population perform equally poorly,

the evolutionary process does not generate any interesting solution. Although several improvements to EA have been suggested [49, 36], a universally applicable solution has not been found yet. Usually, the researcher has to guide the controllers and instruct them towards better strategies. This is demonstrated in the second experiment which works with a more complex task, and also includes active learning. This time, EA is replaced by RL and the controller is guided to reduce uncertainty in its environment by including an RL algorithm tailored for this purpose (*infomax control*). This way, problems of a much larger complexity can be tackled.

The goal of the "social SLAM" experiment[2] was to develop a computational framework for a social robot that would track information about locations and facial expression of humans in a room. The agent learned to act in a way that optimally gathered information about some phenomena in the environment.

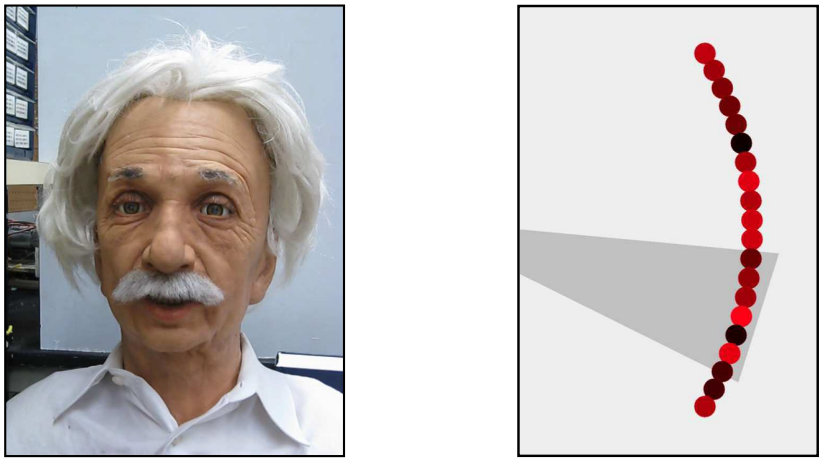


Figure 1.16: **Left:** Einstein robot. **Right:** Probability map in social SLAM experiment indicating the locations of people. The robot's field of view is split into discrete locations. The darker color means higher probability of a human being in the location.

Unlike our previous experiments, in which controllers received rewards from an environment, in this experiment the learner did not need any external reward. The controller maintained its intrinsic reward function, which expressed uncertainty coming from its surroundings. Recently, *information gain* has been proposed as a suitable candidate of such intrinsic motivation for lifelong learning agents. The resulting learning process is self-supervised and it can be paraphrased as "learning to learn".

In this experiment, the robotic head, which was designed to look like Albert Einstein (Figure 1.16, left) had directional cameras in its eyes. Image processing algorithms could locate human faces, detect their head pose, and recognize their facial expressions. The robot was also equipped with a microphone array which could localize sources of sound and a wide-field sillicon retina which could rapidly detect a variety of subtle motion cues. The robot was placed in a large room in which people could enter and leave randomly. The goal of the robot was to move its head and maintain as much certainty about the state of the social environment

[2]The author participated on the experiment during Telluride Neuromorphic Cognition Engineering Workshop 2009, which was held in Telluride, Colorado.

as possible. Occasionally, people approached the robot and tried to interact with it. The robot tried to track information about the location and facial expression of people by rotating its head and selecting the maximally informative action.

The information seeking problem can be modeled as a *Partially Observable Markov Decision Process* (POMDP) [84]. The term partially observable here refers to the fact that the states of interest (e.g., location and expression of people) are not directly observable, and we only have access to sensors that provide partial information about the states of interest.

At first, a classic dynamic programming algorithm was tried to find exact solutions to the POMDP problem. The approach worked well as long as the number of locations and sensory observations was small. Unfortunately dynamic programming approaches were utterly impractical as the number of states and observations were increased to realistic values.

Therefore, reinforcement learning approaches were applied. The idea was to use *information gain* as a reward signal and use reinforcement learning to find a control policy for a POMDP in which movement and sensing actions are selected to reduce *Shannon entropy*. The input to the controller was the probability map (Figure 1.16, right) indicating the locations of people and their states. The output was a camera orientation. This was a challenging problem that we approached using *infomax control* algorithm. The basic goal in infomax control is to behave in a manner that gathers as much information as possible about variables of interest. An infomax control policy was learned in a simulated environment by utilizing Natural Actor Critic Policy Gradient algorithm [82]).

We found that depending on the reliability of the sensors and the dynamics governing the people in the social world, a number of different behaviors emerged automatically from the learning algorithm, typically within a hundred timesteps. These included tracking individual faces, occasionally its attention between people, and turning to look towards sudden outbursts of sound.

Although problems tackled in these two experiments were different, the final strategies involved active learning in some form. In the first experiment, behavior reducing uncertainty was synthetized without the researcher's intervention. That is a very convenient and desirable feature of learning algorithms, as body morphology or sensorimotor interaction did not have to be specified. Unfortunately, this advandage quickly vanishes with an increasing complexity of problems. The second experiment is example of such a problem. In order to develop an optimal policy, a mathematical algorithm tailored to this application was used. As can be seen, RL has a wide variety of algorithms based on solid mathematical backround, that can be applied in specific circumstances. However, incorporation of such algorithm is not cheap usually. In our case, it was necessary to develop quantitative models of the properties of the sensory motor system (probability distribution of the observations given actions and states, for example) for this purpose.

In our previous experiments, ER bypassed the difficulties of hand-coding the control architectures of mobile robots through an artificial selection process that eliminated ill-behaving individuals in a population while favoring the reproduction of better-adapted competitors. According to this approach, a robot's controller, was progressively adapted to the specific environment. Unfortunately, it is not clear yet how to generalize this approach for more complex problems. The

ER still lacks mathematical foundations present in RL. As of there is yet no clear consensus on which type of evolutionary algorithm, and which parameter settings, are appropriate for particular problems, and much experimentation is still on a trial and error basis. Moreover, problems, that are suitable for ER, share some common properties. For example, in more than half of the literature surveyed in [63], wheeled robots that employed differential drive for steering were used. For more complex problems (social SLAM problem), human knowledge must be injected into the controller. Although this approach is much more work-intensive, it is still more powerful.

1.7 Conclusions

In this chapter, we used the two most common automated design methodologies, that can synthetize sensible behavior out of a "tabula rasa". Instead of directly implementing control algorithms, these algorithms are intended to work with little or no human intervention. As has been shown, the robots are able to learn simple behaviors by being rewarded for the good ones, and without explicitly specifying bad actions. Experiments in Section 1.4 illustrated, that both EA and RL designed simple reactive navigation controllers. Unfortunately, as the complexity of an environment and task increases, more prior knowledge and human interference is required. Also, the exact form of fitness function becomes less obvious. We hypothesized that the increasing robustness of trained controllers would help to overcome these problems, but moving this direction did not bring the expected outcome for us.

If the rewards provided to an agent are sparse, there is no path in the search space that can be discovered by the limited sample of potential solutions considered and more human bias must be inserted into the learning process. Although several methods have been suggested to beat the bootstrap problem (incremental evolution [49], co-evolution [36], introduction of intermediate fitness scores [77]), it is unclear, which class of problems may benefit from these improvements. As the methodology has been missing, defining a good fitness function is a trial-and-error process. For some problems, an experimenter may work in iterative fashion and incrementally work towards more complex behaviors, as demonstrated in Section 1.5. The move into multi-robot domain was surprisingly easy, but on the other side, it violated basic principles of automated design, as it required the inclusion of additional behavioral terms into fitness function. Ideally, it should be feasible to perform behavior engineering of intelligent controllers without resorting to explicit design of their cognitive abilities and restricting the range of their actions.

RL algorithms have a strong mathematical background that comes with a great deal of theoretical and empirical results developed in control theory and therefore it may be simpler to utilize them in specific circumstances. They are based on MDT framework, which offers many attractions for formulating learning tasks faced by embedded agents. They deal naturally with the perception-action cycle of embedded agents. They require very little prior knowledge about an optimal solution and they can be used for stochastic environments. On the other side, it is often difficult to fulfill their theoretical requirements, and some of these assumptions are usually ignored in the real-world. In some cases, automated design

may utilize surprisingly good strategies, like in active learning demonstrated in the experiment in Section 1.6. However, for more complex tasks, like social SLAM in Section 1.6, utilization of algorithm designed for this task may be required. Exploitation of RL algorithms may be preferred to EA in these domains. In social SLAM experiment, successful exploitation of infomax control was demonstrated, but it was no free lunch. It was necessary to develop quantitative models of the properties of the sensory motor system.

The agents that we have discussed in this chapter belong to a class of agents based on behavior-based paradigm. This paradigm in general does not use representational knowledge, and emphasizes tight coupling between sensing and motor actions. It might be difficult to extract learned knowledge from these architectures. Automated design especially often comes up with solutions that match an environment in ways that are often unpredictable from the perspective of an external observer. Verification and testing of evolved controllers is far from a trivial process. For complex, real-world robotic systems, hybrid architectures that combine adaptive methods with traditional planning are widely used. We will demonstrate such systems in the following chapters. The field of automatic intelligent control learning for autonomous robots is in its infancy. Much of the research focuses on learning how to perform relatively simple tasks. In the next chapter, we will show how methods based on DP algorithms revisited in this chapter form the core of modern path planning algorithms. Machine learning paradigms are used mostly to fine-tune the parameters of already successful designs. Developing machine learning methods for use in robotic systems has, in fact, become a major focus of contemporary autonomous robotics research.

2. Hybrid Agents

In this chapter, we will describe a hybrid design of a robot controller, that utilizes modern path planning algorithms. Hybrid architecture, which combines both reactive and deliberative planning will be demonstrated. Deliberative layer will be described in detail in the next chapter. In this chapter, we will focus on motion planning and our experiences with its implementation for small low-cost mobile robots. The path-planning module is based on the value iteration algorithm from the previous section (Algorithm 6). Although path planning algorithms are established and well working in general, implementation of these algorithms in low-cost robots is challenging due to their very limited sensory system. Further details can be found in articles [104] and [99].

2.1 Agents Taxonomy

Traditionally, robot controllers were mostly serial processing units in AI. They built representations of the outer world, reasoned about it and planned actions accordingly. The internal world often comprised geometric maps and they were often based on symbolic reasoning. Historically, Nilsson's SHAKEY robot was the first successful demonstration of symbolic, AI-type problem solving in robotics. Unfortunately, it was too brittle to operate in a dynamic environment [6].

In 1986, Rodney Brooks and his colleagues, pioneered and popularized a different approach called *behavior-based robotics* [22]. They built a series of wheeled and legged robots utilizing the *subsumption architecture* [21]. The behavior-based approach states that intelligence is the result of the interaction among an asynchronous set of behaviors and the environment. Behavior-based systems are also reactive and they use relatively little internal variable states to model the environment. It has been shown that others designs based on reactive paradigm can effectively produce robust performance in complex and dynamic domains. The successful examples include motor schema-based systems, action-selection or colony-architecture. These architectures, in general, do not use representational knowledge and emphasize tight coupling between sensing and action and decomposition into behavioral units.

As we have discussed in the previous chapter, reactive design can serve as a disadvantage at times. In some cases, exploitation of deliberative planners may be beneficial. If the world can be accurately modeled (with restricted uncertainty) or if there are only a few changes in the world during execution, deliberative planning may be preferred.

In general, both approaches have limitations when they are considered in isolation. Therefore, hybrid deliberative/reactive robotic architectures have emerged. They combine aspects of traditional AI symbolic methods but maintain the goal of providing responsiveness and flexibility of purely reactive systems. The majority of more advanced mobile agents have to address some fundamental problems, like localization in map, map building, path planning or collision avoidance. Usually, these algorithms are implemented as interacting layers. Since the millenium, a performance of a hybrid architecture has been succesfully demonstrated several times. Modern robots contain several modules, that communicate asyn-

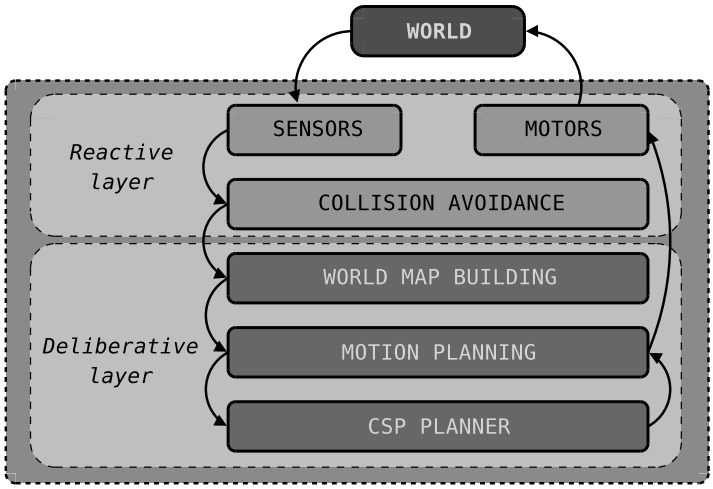


Figure 2.1: The proposed hybrid hierarchical architecture, that combines deliberative and reactive components.

chronously. The software architecture of tour-guide robot RHINO consisted of 20 processes, which were executed in parallel on 3-board PCs and 2 off-board SUN workstations, connected via Ethernet [24].

A multilayered hybrid architecture, comprising top-layer planning system, and a lower-level reactive system has been emerging as the architectural design of choice since 1995 [5]. In general, two layers are needed at minimum: one to represent deliberation and the other reactivity. The reactive layer is used for short-term planning, so that the robot can operate in dynamic environment, where changes occur quickly. The deliberative layer is used for long-term planning. Another common approach involves introducing an explicit third layer, concerned with coordinating the two components.

Planning can be incorporated into the overall system architecture in various ways. Work [6] contains a broader overview of hybrid architectures that combine both reactive and deliberative planning. Planning can be viewed as configuration, and planner component can reconfigure the system to adapt to new challenges [7]. Planning can be also viewed as advice giving (Atlantis [38]). In this case, the planner suggests changes to reactive control system, that may be refused. Planning may also be viewed as least commitment process (Procedural Reasoning System [57]). In this design, the planner defers making decisions on actions until as late as possible. In our design (see Figure 2.1), planning is viewed as adaptation. The planner continuously alters lower layers of the control system in light of changing conditions within the world and task requirements. The system consists of the following components:

- Obstacle avoidance module: The layer with a highest priority, collision avoidance module, is activated only when any obstacle is found to be too close. In that case, the robot executes an avoiding manoeuvre.

- Motion-planning module: This module consists of three components. The map building component is able to fully construct a map. The localization component determines the robot's position and orientation in a map. Path planning module is responsible for scheduling robot movements that bring

the robot from its current position to final destination. We will discuss each component separately.

- Task control module (high-level planner): The layer with lowest priority provides long-term plans based on inputs from lower layers. It is a layer that usually utilizes traditional AI technique (constraint programming in our case). This layer will be discussed in detail in the next chapter.

We will now discuss each component involved in the motion planner separately.

2.2 Mapping

We assume, that motion planning algorithms know a map of an environment in advance (distribution of obstacles and walls in an environment and the position of landmarks). We do not consider the more difficult *simultaneous localization and mapping* (SLAM) problem in this work (the case, when a robot does not know its own position in advance and does not have the map of the environment available). To obtain the map, the robot is manually guided by a human to explore the whole environment. Sensor measurements are then processed by the mapping module (see Figure 2.1) and used to construct the map.

The output of the mapping process is the *occupancy grid* [109]. Occupancy of all $< x, y >$ locations in the environment is estimated from sensor data. Let c_{xy} denote a random variable with event $\{0, 1\}$ that corresponds to the occupancy of a location $< x, y >$. 1 stands for occupied, and 0 stands for free. The problem of mapping is to estimate the probability of occupancy for each grid cell given the history of measurements, denoted as

$$P(c_{xy}|o_1, ..., o_t),$$

where o_t is sensor measurement at a time step t.

Occupancy grids are illustrated in Figure 2.6 and Figure 2.9.

2.3 Localization

Localization is the process of estimating a robot's current position in the known map. In our case, we estimate the robot's position and orientation in the 3-dimensional space. Although even low-cost robots have devices to estimate covered trajectory, dedicated algorithms are needed to estimate robot position to obtain satisfactory precision. Position could be estimated using robots shaft encoders and precise stepper motors. This process is called *dead reckoning* [6]. For robots equipped with a differential drive like E-puck (Figure 2.2), the dead reckoning process is very simple. The position of the robot can be estimated by looking at the difference in the encoder values Δs_R and Δs_L. By estimating the position of the robot, we mean the computation of tuple $< x, y, \Theta >$ as a function of the previous position $< x_{OLD}, y_{OLD}, \Theta_{OLD} >$ and encoder values (Δs_R and Δs_L):

$$\begin{pmatrix} x \\ y \\ \theta \end{pmatrix} = \begin{pmatrix} x_{OLD} \\ y_{OLD} \\ \theta_{OLD} \end{pmatrix} + \begin{pmatrix} \Delta x \\ \Delta y \\ \Delta\theta \end{pmatrix} \tag{2.1}$$

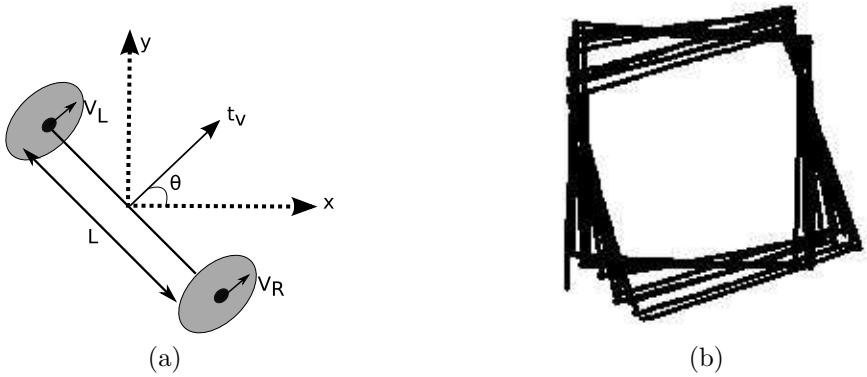


Figure 2.2: **Left:** Differential drive robot schema. **Right:** Illustration of error accumulation. E-puck was ordered to make 10 squares of size 30 cm, real trajectory was captured by a camera. Odometry errors are caused mostly by rotation movement.

where

$$\begin{aligned}\Delta\theta &= \tfrac{\Delta s_R - \Delta s_L}{L},\\ \Delta s &= \tfrac{\Delta s_R + \Delta s_L}{2},\\ \Delta x &= \Delta s.cos(\theta + \tfrac{\Delta\theta}{2}),\\ \Delta y &= \Delta s.sin(\theta + \tfrac{\Delta\theta}{2}).\end{aligned}$$

The fundamental flaw is error accumulation. Each time an encoder measurement is taken, the imprecision of sensors will cause a difference between real and calculated position. This error accumulates over time and therefore accurate tracking over large distances is virtually impossible without additional feedback. Tiny differences in wheel diameter will result in important errors after a few meters, if they are not properly taken into account, as demonstrated in Figure 2.2.

For longer trajectories, more clever methods are needed. Several algorithms are used to estimate a pose in the known map and cope with errors, that arise due to inaccuracy of robot sensors and effectors. Methods based on Kalman filter [47] (EKF localization, or some of its variants) or particle filter [88] (Monte-Carlo localization) have shown good performance in a number of application areas. We are using Monte-Carlo localization (MCL), one of the most popular localization algorithms in robotics [109].

The Monte-Carlo localization works with quantity $p(x_t)$, which stands for the probability that the robot is located at the position x_t in time t. This quantity is represented by a set of particles $X_t = \{x_t^{[1]}, \ldots, x_t^{[M]}\}$. Each particle $x_t^{[m]}$ represents a hypothesis, that corresponds to a robot's pose at time t. As opposed to dead reckoning, MCL incorporates sensors measurements z_t.

The advantage of MCL compared to some other approaches is that it can handle the *global localization problem*, as demonstrated in Figure 2.3, top. Global localization is the problem of determining the position of a robot under global uncertainty. The other advantage of MCL is the ability to represent multi-modal probability distributions (track many different possible positions at the same time), as illustrated in Figure 2.3, bottom.

The more sensors an embodied agent can utilize, the better. *Sensor fusion* is the combining of sensory data from disparate sources so that the resulting

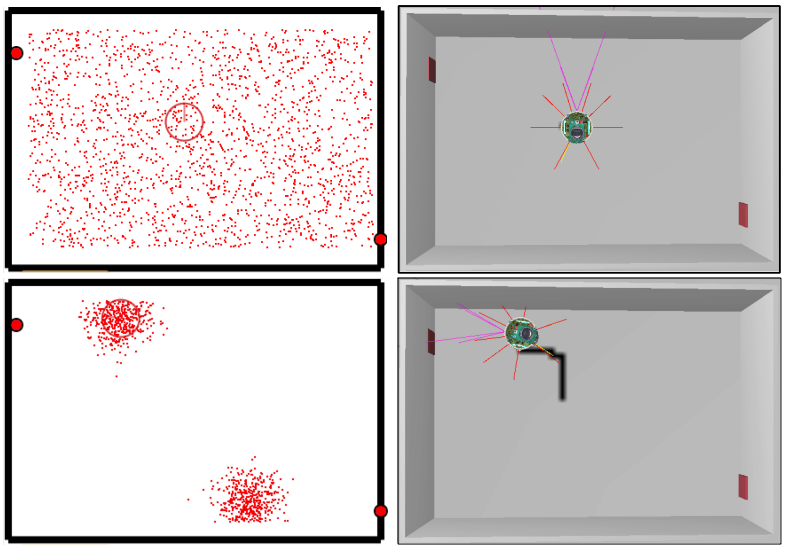


Figure 2.3: Representation of probability distribution by a set of particles. Although only the x- and y-coordinates of each particle is illustrated, each particle is defined over the 3-dimensional space. **Top:** Initially unknown position is represented by particles equally distributed in the arena. **Bottom:** The state of particle filter 50 steps later. Particle filter represents a multimodal distribution. It is a consequence of symmetry of the environment.

information is more accurate than would be possible when these sources were used individually. The algorithm requires functions that model both robot sensors and actuators. Function *motion_model()* implements a probablistic odometry model, which is defined as a conditional probability distribution $p(x_t|u_t, x_{t-1})$, where x_t and x_{t-1} are both robot poses and u_t is a motion command. Similarly, the function *measurement_model()* implements a measurement model, which is defined as a conditional probability distribution $p(z_t|x_t)$, where x_t is the robot pose and z_t is the measurement at time t. These probabilistic measures model a noise and inherent uncertainty in a robot's sensors and effectors. Such models can be obtained in several ways, as discussed in detail in [109].

The MCL algorithm (Algorithm 8) takes the set of particles X_{t-1}, the most recent control command u_t and most recent sensor measurements z_t as an input. The output is a new set of particles X_t with corresponding weights. The algorithm possesses three basic steps: state prediction, observation integration and re-sampling.

1. State prediction based on odometry model: The first step is the computation of temporary particle set $\bar{X}_t$ from X_{t-1}. It is created by applying the odometry model $p(x_t|u_t, x_{t-1})$ to each particle $x_{t-1}^{[m]}$ from X_{t-1}.

2. Correction step (observation integration): The next step is the computation of *importance factor* $w_t^{[m]}$, which is the probability of the measurement z_t under particle $x_t^{[m]}$, given by $w_t^{[m]} = p(z_t|x_t^{[m]})$. This step typically involves integrating multiple sensor measurements.

3. Re-sampling: The last step incorporates so-called *importance sampling*. The algorithm draws with replacement M particles from temporary set

Input : X_{t-1}: Set of particles at time step $t-1$,
u_t: Control command at time step t,
z_t: Sensors measurement at time t,
M: Number of particles in set.
Output: X_t: New set of particles at time step t.

$\bar{X}_t = X_t = \emptyset$
for $m = \{1 \cdots M\}$ **do**
  State prediction based on odometry model:
  $x_t^{[m]}$ = sample_motion_model (u_t, $x_{t-1}^{[m]}$)
  Correction step:
  $w_t^{[m]}$ = measurement_model (z_t, $x_t^{[m]}$)
  $\bar{X}_t = \bar{X}_t + < x_t^{[m]}, w_t^{[m]} >$
end
Re-sampling:
for $m = \{1...M\}$ **do**
  draw m with probability $\propto w_t^{[m]}$
  add $x_t^{[m]}$ to X_t
end
return X_t

Algorithm 8: MCL, or Monte Carlo Localization [109], a localization algorithm based on particle filters.

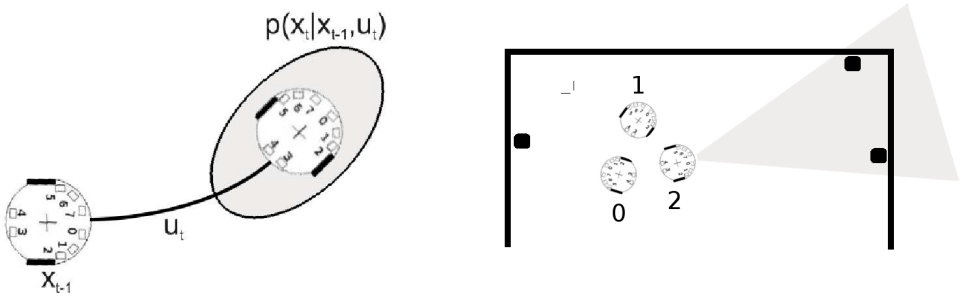


Figure 2.4: **Left:** In the prediction step of MCL, an odometry model based on movement u_{t-1} is applied to each particle x_{t-1} and new hypothesis x_t is sampled from distribution $p(x_t|x_{t-1}, u_t)$. **Right:** In the correction step of MCL, each particle is assigned an importance factor, corresponding to the probability of observation z_t. If image processing detects two landmarks on the actual camera image, particles 0 and 1 will be assigned a smaller weight than particle 2.

$\bar{X}_t$ and creates new particle set X_{t+1}. The probability of drawing each particles is given by its importance weight computed in the previous step. This is our second encounter with *the survival of the fittest* principle ([48]).

2.4 Motion Planning

The collision avoidance module considers only local constraints. The motion planning module takes a more global view. It determines the motion strategy that will take the robot from its current position to the desired position. There have been plenty of algorithms designed, with different levels of complexity [52]:

1. Methods exploring a search graph
   (a) Visibility graph algorithm
   (b) Retraction-like algorithms
   (c) Algorithms based on exact cellular decomposition
   (d) Algorithms based on approximate cellular decomposition
   (e) Probabilistic roadmap and its variants
2. Methods building a search tree
   (a) Grid-based methods (Dynamic programming, A* algorithm...)
   (b) Rapidly-Exploring Random Tree
3. Heuristic approaches
   (a) Navigation function
   (b) Path deformation

The majority of work addresses more complicated problems than the one considered in this work, such as motion planning in higher-dimensional and continuous space. Motion planning for circular robots that can turn on the spot is often performed in 2D, ignoring costs of rotation and the dynamics of the robot, and it is our case, too. Such simplification yields only sub-optimal results, but greatly reduces complexity of motion planning, which is known to be exponential in the number of degrees of freedom [24].

In the previous chapter, we introduced the value iteration (Algorithm 6) in context of a robot controller. Value iteration is a good path planning algorithm, as well. Fundamentally, both planning and robot control address the same problem: to select actions. The important criterion is the amount of uncertainty that these algorithms support:

1. Deterministic versus stochastic action effects: There is no uncertainty in the classical robot planning paradigm. Value iteration supports the stochastic nature of the robots and their environments.
2. Fully observable versus partially observable systems: Our version of value iteration works with fully observable worlds, and assumes that sensors can measure the full state of the environment. As the sensors are not perfect, this is an unrealistic assumption.

As the value function is computed only in x-y-space, the cost of rotation and the robot's velocity is ignored. Such planners, that can not cope with robot dynamics are usually combined with fast collision avoidance modules. Consideration of the full robot state would require planning in at least five dimensions and would be much more CPU intensive.

The version of the algorithm that we use assumes finite state and action spaces. We therefore process similar pre-processing as in the previous chapter. A decomposed 2D grid serves as an approximation of the continuous optimal value function, where each grid cell corresponds to one state. The finer representation usually yields better results, at the price of increased computational requirements. In our experiments, each grid cell is of the same size.

In the Initialization step of Algorithm 6, value function for each state is set to ∞, besides the target location, which is set to 0:

$$V_{x,y} = \begin{cases} 0, & : \text{ if } < x, y > \text{ is target cell,} \\ \infty, & : \text{ otherwise.} \end{cases}$$

In the Update step of Algorithm 6, all values of non-target grid cells are updated by the value of their best neighbors, plus the costs of moving to this neighbor:

$$V_{x,y} \leftarrow \min_{\substack{\Delta x=-1,0,1 \\ \Delta y=-1,0,1}} \{V_{x+\Delta x,y+\Delta y} + P(c_{x+\Delta x,y+\Delta y})\}.$$

The cost here is equivalent to the probability $P(c_{x,y})$ that a grid cell $< x, y >$ is occupied. When the update converges, the value of each state measures the cumulative cost for moving to the nearest goal.

Output of the algorithm is illustrated in Figure 2.6 and Figure 2.9. The brighter a location, the higher its value. To determine where to move, the robot generates a minimum-cost path to the goal. In other words, it uses greedy action selection with respect to the computed value function. This is done by the steepest descent in V, starting at actual robot position.

Disadvantages of value iteration usage in the path planning domain are:

- It is inefficient to allow the robot to navigate while simultaneously learning a map.
- It is suitable for smaller low-dimensional spaces.

Attractive features a motion planner based on value iteration are:

- It is an any-time algorithm, since it allows the generation of (suboptimal) robot action at almost any time, before the iteration has converged.
- It computes value function for each cell, not just the current location of the robot. As a consequence, the robot can quickly react if it finds itself in an unexpected location. This is especially important in the presence of a collision avoidance module that can dislocate the robot from the planned path.

2.5 Motion Planning with Low-cost Platform

The affordability of miniature hardware platforms such as E-puck with mature software simulation environments makes small robots a commonly used experimental platform. It is important to consider that such a platform possesses certain special properties differentiating it from other professional robots. Small mobile robots are usually not equipped with a suite of sophisticated sensors. This experiment studies the performance of localization algorithm based on a particle filter with a small miniature low-cost E-puck robot. Information from a cheap VGA camera and eight infrared sensors are used to correct the estimation of the robot's pose.

Unfortunately, due to their characteristics (Appendix 1), infra-red sensors of E-puck robot can be used as bumpers only. We distributed special objects of different colors, called landmarks, into the arena (Figure 2.5) and implemented a simple algorithm, which processes images obtained from the camera and performs landmark recognition. Landmarks were objects of rectangular shape of size 5×5 cm and three different colors — red, green and blue. We developed an image processing module that detected a relative position of the landmark from the robot.

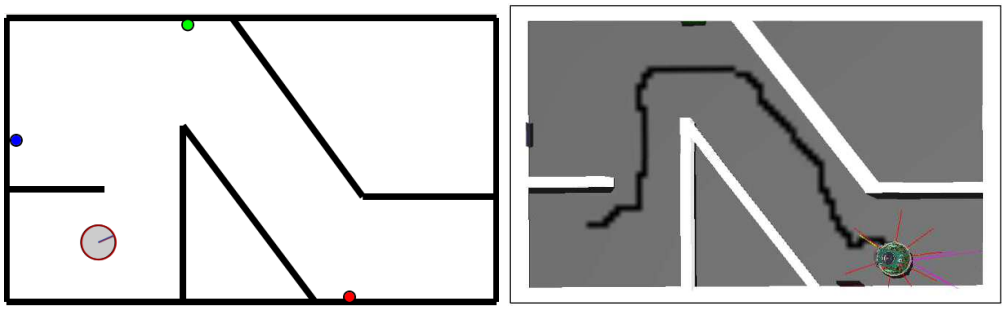


Figure 2.5: **Left:** Performance of localization algorithm was measured in an arena with red, green and blue landmarks. **Right:** Simulated arena with E-puck and trajectory covered when executing the experiment.

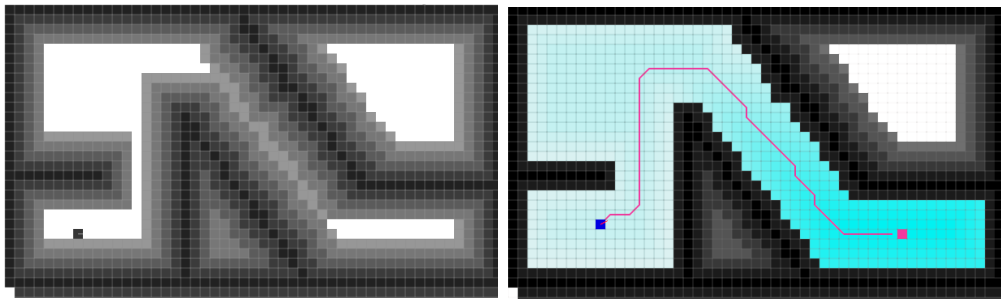


Figure 2.6: **Left:** Occupancy grid map corresponding to the environment depicted in Figure 2.5. **Right:** Example of value iteration over state spaces in robot motion. The blue rectangle depicts the robot position, the red rectangle represents the goal location. Shading corresponds to the value function. The robot trajectory covered when executing this plan is depicted in Figure 2.5, right.

Algorithm	Error (cm)
Dead reckoning	4.9
MCL (no landmarks)	3.75
MCL (3 landmarks)	1.43

Table 2.1: Average of a localization error from 30 runs.

The localization module was based on MCL(Algorithm 8). In the correction step of the algorithm, two types of sensors measurements were fused:

- Measurements coming from distance sensors. Because of their characteristics, eight infra-red distance sensors were used as bumpers only.
- Measurements obtained by image processing. The expected position of the landmarks and detected position of the landmarks from camera image were compared. E-puck's camera can be used to detect objects or landmarks. However, the information about distance to the landmark extracted from the camera turned out not to be reliable (due to the noise), and we did not use it.

Output provided by the image processing algorithm was the relative position and color of the detected landmarks ("I see red landmark by angle 15 degrees"). Image processing incorporated the following steps (see [91]):

- Gaussian filter was used to reduce camera noise.
- Color segmentation into the red, blue and green color was performed.
- Blob detection was used to detect the position and size of the objects on the image.
- Object detection was used to remove objects that have non-rectangular shape from the image.

Experiments were carried out in an arena of size 1×0.75 meters. Three landmarks were placed into the arena, as shown in Figure 2.5. As a first step, E-puck robot was guided to map the environment, producing the occupancy grid The robot was then put into the arena 30 times, ordered to cover the path from Figure 2.6, right each time. The localization algorithm was reset on each run, so that the robot did not know its starting position.

MCL algorithm worked with 2000 particles, and after several steps, it relocated the particles into real location of the robot. The robot was able to localize itself. The Table 2.1 shows an average error between the real and the estimated position during experiment. The robot was able to autonomously navigate in this simple environment. We were surprised by the flawlessness of the algorithm, as we used only bearing information from camera (no distance information). Position error could be easily reduced, if we put more landmarks into environment. The utilization of landmarks improved performance of localization algorithm by a great extent, as shown in Table 2.1. Localization without landmarks would not be feasible. The impact of infrared sensors was obvious, too. The proximity to the wall suddenly significantly reduced the position uncertainty.

The localization algorithm performed well in other arenas (Figure 2.3) and retained such performace (in terms of average localization error) in rectangular areas up to a size of three meters.

2.6 Waste Collection Task

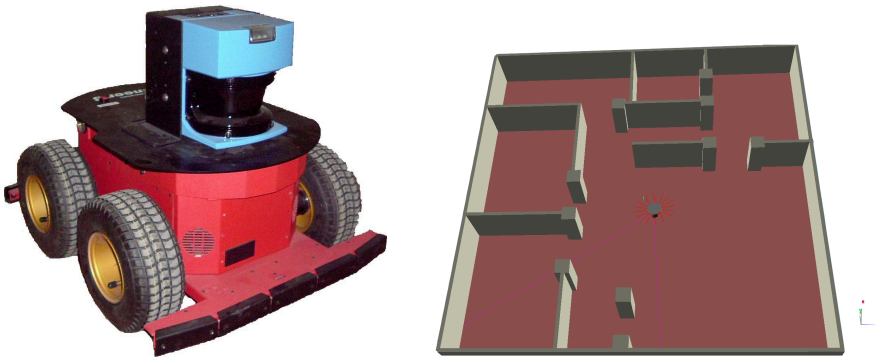


Figure 2.7: **Left:** The more advanced mobile robot Pioneer-2 is equipped with a SICK LMS-2000 laser sensor, which provides distance measurements over a 180 degree area up to 80 meters away. **Right:** Training arena for Pioneer-2 robot.

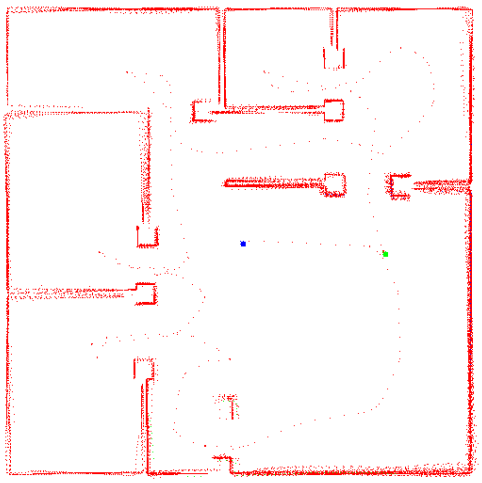


Figure 2.8: The output of the map building module. The map was built by a random walk in the arena depicted in Figure 2.7.

Although localization and motion planning are feasible with E-puck in simple environments, more elaborate environments require sensors with a very good precision. Therefore, in this experiment, we will utilize a model of the professional robot Pioneer-2. This robot is equipped with a *SICK LMS-2000* laser sensor, which provides distance measurements over a 180 degree area up to 80 meters away. The part responsible for handling localization and motion planning is based on the well established open-source *CARMEN* software [60], which contains the complete model of Pioneer-2 (Figure 2.7, left), including the odometry model and the motion model.

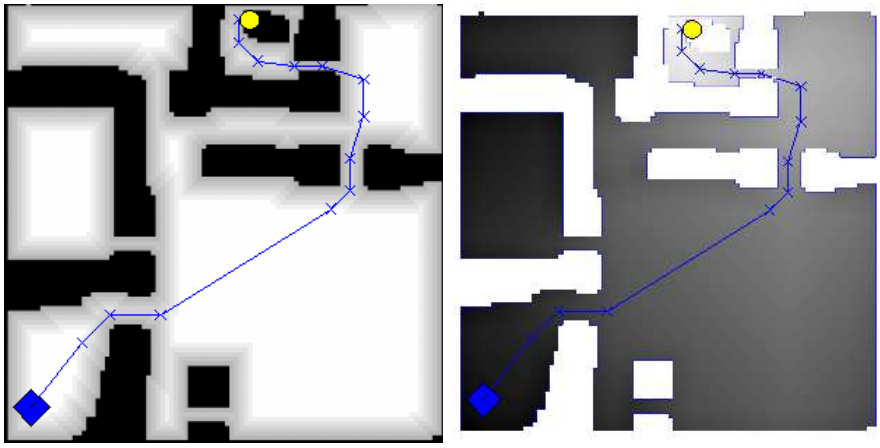


Figure 2.9: **Left:** An additional post-processing of map from Figure 2.8 increased the side-clearance to obstacles. **Right:** The motion planner uses DP to compute the shortest path to the nearest goal for every location in the unoccupied space, as indicated by the gray shading.

CARMEN is an open-source collection of software for mobile robot control. It is designed in a modular way to provide basic navigation primitives including obstacle avoidance, localization, path planning, and mapping. Communications between CARMEN modules is handled using a separate package called IPC that shadows developer from details of inter-process communication. We connected CARMEN with the professional simulator Webots, which contains a realistic model of the Pioneer-2 robot.

The architecture of the robot is the same as in the previous experiment, only motion planning modules are based on components from CARMEN toolkit. Path planner module still implements the value iteration and the localization is performed by MCL algorithm. The experiment workflow remains the same, too: As a very first step, the whole environment (Figure 2.7, right) is mapped and obtained measurements (Figure 2.8) are integrated into the occupancy grid (Figure 2.9, left). The performance of value iteration algorithm is depicted in Figure 2.9, right. A all software components were integrated into the simulator, as can be seen in Figure 2.10. The measurements of a laser sensor are illustrated in the left part of the figure (window titled "Robot Graph"). The map of the environment is in the upper left part. The blue diamond in the map represents the current location of the robot and the yellow circle the current target location that was obtained from the high-level planner.

The high-level planner evaluates data from lower layers and accordingly modifies the current plan. In this experiment, the robot has to clean out a collection of wastes (represented by red balls) spread in a building; but in any time, its internal storage capacity cannot be exceeded. Providing the storage tank is filled up, the robot has to empty it into one of the available collectors (represented by the yellow circle). The goal of the planner is to come up with a routing plan, that minimizes the covered trajectory.

Recent research in robotics has produced a variety of frameworks for task-level planners [24]. The tour-guide robot RHINO [24] uses GOLOG [53], which is a first-order language, that represents knowledge in the situation action calculus. Recent increasing interest in this area brought another powerful frameworks

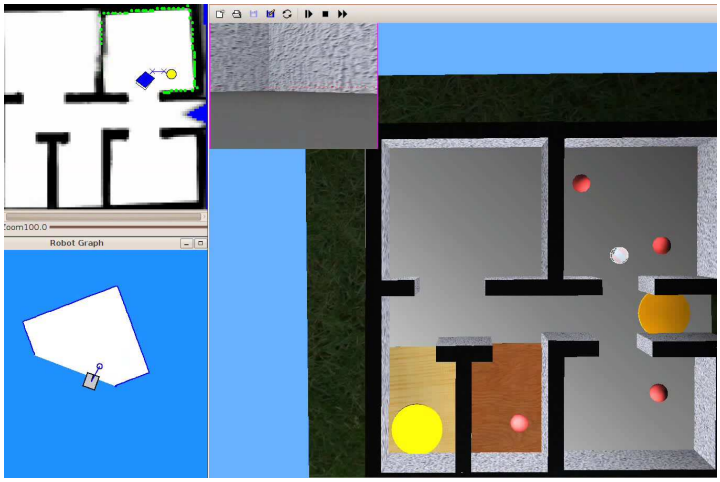


Figure 2.10: Commercial Webots simulator with model of Pioneer-2 and localization and motion planning modules. Map built by the robot corresponding to the testing environment with output of the motion planner module is depicted in the upper left corner of the figure.

(PRODIGY [26], COLBERT [51]...). In the next chapter we will show that constraint programming, which has been often overlooked in the robotics community, fullfills all requirements given by high-level planners in robotic systems.

2.7 Conclusions

In this chapter we introduced a hybrid architecture of embodied agents, that combines reactive and deliberative paradigms. Such architecture is typically organized into layers. We focused on layer that performs localization and motion planning. As we have shown, the localization process can be carried out even with low-cast robot in simple environments. More realistic experiments require more advanced sensory systems, such as the one offered by the Pioneer-2 robot. The algorithms that we discussed in this chapter are able to seamlessly take care of motion planning and recompute the current plan in a few milliseconds, when executing on moderate PCs, for environments of modest sizes. In the next chapter we will discuss the high-level planner based on constraint programming that controls the discussed motion planner. As we will show, such a planner fits well into introduced design. It shows very good performance in our experiments and has features that makes it suitable for deployment in robots.

3. Deliberative Planning

In the previous chapter, we introduced a hybrid robot architecture that combines both a reactive execution and deliberative planning. In this chapter, we will describe the deliberative planner, which controls high-level tasks, in detail. The goal of the robot is to clean out a collection of wastes spread in a building; but under the condition of not exceeding its internal storage capacity. The storage tank can be emptied in one of the available collectors. The high-level planner has to prepare a routing plan, which minimizes the covered trajectory.

As the high-level task can be converted into a graph exploration problem, we modeled it using CP paradigm. The problem is an important variant of popular vehicle routing problem (VRP) that has not been yet extensively studied. We will model it in CP language and verify its performance in experiments. We will discuss requirements given by the robotics domain on deliberative planners and we will demonstrate that high-level CP planner fulfills them. The crucial aspect of the high-level planner in our design is the ability to produce a solution (possibly sub-optimal) in a very short time frame. We will show, that CP planner is able to find the first solution in hundreds of microseconds on problems of reasonable sizes.

3.1 Problem Formulation

The environment consists of navigation points defined by locations of waste and collectors. We use a mixed weighted graph (V, E) with both directed and undirected edges to represent this environment. Figure 3.1 gives an example of the initial environment (left) and the planned path for the robot (right). The undirected edges allow us to minimize the size of the representation. The set of vertices $V = \{I\} \cup W \cup C \cup \{D\}$ consists of the initial position I, the set W of waste vertices, the set C of collectors and the destination vertex D.

There are directed arcs from I to all vertices in W, because the robot has to visit some waste vertices from the initial position. As the robot can travel between waste vertices, we assume a complete undirected graph between vertices

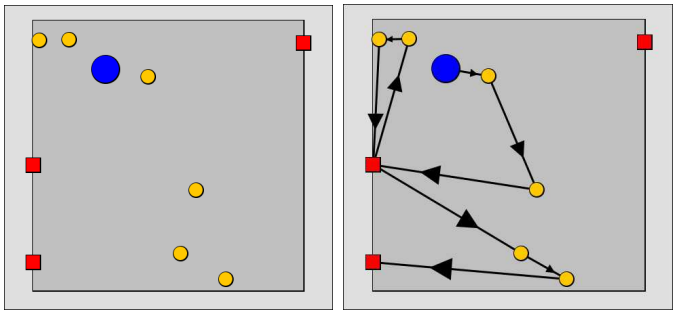


Figure 3.1: Example of waste collection task. The biggest circle represents the robot, six smaller circles show distribution of wastes. The three squares are collectors. The goal of the robot is to collect wastes and minimize covered distance. In this example, the robot can hold at most two wastes at a time.

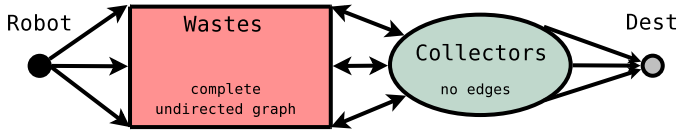


Figure 3.2: A schema of the graph describing the robot's environment with the navigation points.

in W. The robot can go to a collector from any waste vertex, therefore, we use a directed edge there. The fact that the robot can go to any waste from any collector is again modelled by a directed edge. Directed edges are required as we need to count the number of incoming and ongoing edges for collectors. There are no edges between the collector vertices. We use a dummy destination vertex that is connected to all collector vertices by a directed edge. The weight of each edge describes the distance between the navigation points. The edges going to the dummy destination vertex D has zero weight so the robot can actually finish at any collector. The task to find a minimal-cost path starting at I, finishing at D and visiting each vertex in W exactly once such that the number of any consecutive vertices from W does not exceed the given capacity of the robot. Figure 3.2 shows the schema of the graph with the navigation points.

The task is to develop a robot solving a specific routing problem - an often overlooked variant of the standard vehicle routing problem. The amount of literature addressing VRP subject and its derived variants is enormous. However, to our best knowledge, the VRPSF variant has not been solved extensively. The latest published results can be found in [13], and are mentioned also in [28]. The authors presented an exact procedure for solving the VRPSF that combines heuristics with methods from polyhedral theory in a branch and cut framework. They focused on obtaining an optimal solution that took almost an hour on 15-customer case instances. Nowadays, there exist truly numerous optimization attitudes ranging from the "classic" operation research field to those inspired by nature [16, 10]. In the published results, a majority of authors show their testing method is applicable to use or even outperforms some other methods for the studied problem variants. Therefore, to properly decide which way to choose is a hard row to hoe.

There have been several ways of tackling VRP problem using CP presented so far. Shaw [92] is among the first pioneers in this field, who used limited discrepancy search (LDS) methods with large neighborhood search for capacitated VRP and VRP with time windows (VRPTW). Another work [9] describes a method for combining constraint programming with tabu search and guided local search techniques to solve the standard VRP problem. A combined method was able to outperform the other methods on most problems. To mention one more approach, the authors of [87] studied benefits of several constraint-based operators on 56 Solomon instances of VRPTW; the method showed good results and allowed easy further modification of the model.

Our primary goal is to develop an algorithm that returns good solutions in a short time (almost anytime algorithm) and that can be easily extended by additional constraints. Hence ad-hoc exact techniques are not appropriate due

to their long runtime and limited extendibility and we decided to use CP to solve the problem. Neither of the existing CP-oriented works solves the above problem, but we can use them as the initial motivation for the design of our constraint model. Most of the routing models are based on the formulation of the problem using network flows [93] so we also proposed a constraint model based on this standard technique. Nevertheless, the performance of this model was not satisfactory in our experiments so we proposed a radically new approach to model the problem using a finite state automaton. In our preliminary experiments, this model outperformed the traditional model and solved larger instances of the problem.

3.2 Constraint Programming Planner

Constraint satisfaction programming (CP) [86, 31] is the process of finding a solution to a set of *constraints* that impose conditions that the variables must satisfy. The central notion is that of a constraint - a relation over the domains of sequence of variables. One can view it as a requirement that states which combinations of values from the variable domains are admitted. In turn, a *c*onstraint satisfaction problem (CSP) consists of a finite set of constraints, each on a subsequence of variables.

To solve a given problem by means of constraint programming we first formulate it as a constraint satisfaction problem. This part of the problem solving is called *m*odeling. In general, more than one representation of a problem as a CSP exists. Then to solve the chosen representation, we use mostly the general methods, which are concerned with the ways of reducing the search space and with specific *search methods*. The algorithms that deal with the search space reduction are usually called *constraint propagation algorithms*. They maintain equivalence while simplifying the considered problem and achieve various forms of *local consistency* that attempt to approximate the notion of (global) consistency.

We present two approaches based on constraint programming techniques for the introduced problem. The former one is inspired by the operations research model, namely by the network flows, while the second one is driven by the concept of finite state automaton. The experimental comparison and enhancements of both models are discussed with emphasis on the further adaptation to the robotics domain.

3.2.1 Model Based on Network Flows

The first model that we propose resembles the traditional operations research models of vehicle routing problems based on network flows and Kirchhoff's laws. Basically, we are describing whether or not the robot traverses a given edge. For every edge e we introduce a binary decision variable X_e stating whether the edge is used in the path (value 1) or not (value 0). Let $IN(v)$ and $OUT(v)$ denote the set of incoming and outgoing directed edges for the vertex v. For example, for $v \in W$ the set $IN(v)$ contains the arc from the vertex I and the arcs from the vertices in C. Let $ICD(v)$ be a set of undirected edges incident to vertex v. This set is empty for the collector vertices; for waste vertices it

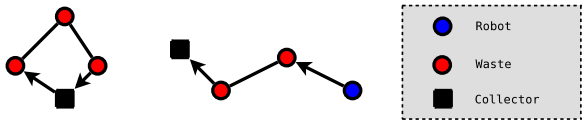


Figure 3.3: An ineligible loop (left) satisfying the routing (*Kirchhoff's*) constraints.

contains undirected edges connecting the vertex with other waste vertices. The following constraints describe that the robot leaves the initial position I, reaches the destination position D, and enters each collector c the same number of times as it leaves it:

$$\sum_{e \in \mathbf{OUT}(I)} X_e = 1, \quad \sum_{e \in \mathbf{IN}(D)} X_e = 1, \tag{3.1}$$

$$\forall c \in \mathbf{C} : \sum_{e \in \mathbf{OUT}(c)} X_e = \sum_{e \in \mathbf{IN}(c)} X_e \tag{3.2}$$

Let us now describe the constraint that each waste vertex w is visited exactly once. It means that exactly two edges incident to a waste vertex w are active (used in the solution path) and there can be at most one active incoming and outgoing directed edge connecting the waste with the collectors or with the initial node.

$$\forall w \in W : \sum_{e \in \mathbf{OUT}(w) \cup \mathbf{IN}(w) \cup \mathbf{ICD}(w)} X_e = 2, \tag{3.3}$$

$$\forall w \in W : \sum_{e \in \mathbf{OUT}(w)} X_e \leq 1, \tag{3.4}$$

$$\forall w \in W : \sum_{e \in \mathbf{IN}(w)} X_e \leq 1 \tag{3.5}$$

The above constraints describe any path leading from I to D, but they also allow isolated loops as Figure 3.3 shows. This is a known issue of this type of model that is usually resolved by additional sub-tour elimination constraints forcing any two subsets of vertices to be connected. In our particular setting, we need to carefully select these pairs of subsets of vertices because there could be collector vertices that are not visited. Hence, we consider any pair of disjoint subsets $S_1, S_2 \subseteq (W \cup C)$, such that neither S_1 nor S_2 consists of collector vertices only. More precisely, we assume the pairs of subsets S_1, S_2 such that:

$$S_2 = (W \cup C) \setminus S_1, \quad S_1 \cap W \neq \emptyset, \quad S_2 \cap W \neq \emptyset \tag{3.6}$$

The sub-tour elimination constraint can then be expressed using the following formula ensuring that there is at least one active edge between S_1 and S_2.

$$\sum_{e \in E : e \cap S_1 \neq \emptyset \, \wedge \, e \cap S_2 \neq \emptyset} X_e \geq 1 \tag{3.7}$$

Clearly, there is an exponential number of such pairs S_1 and S_2, which makes it impractical to introduce all such sub-tour elimination constraints. Some authors [29] propose using single or multi-commodity flow principles to reduce the

number of constraints by introducing auxiliary variables. However, our combination of directed and undirected edges makes it complicated to use this approach so we applied another approach based on the lazy (on-demand) insertion of sub-tour elimination constraints. Briefly speaking, we start with the model without the sub-tour elimination constraints and we find a solution. If the solution forms a valid path then we are done. Otherwise we identify the isolated loops, add the sub-tour elimination constraints for them, and start the solver with the updated model. This process is repeated until a valid path is found. Obviously, it is a complete procedure because in the worst case, all sub-tour elimination constraints are added.

What remains is to define the constraints describing the limited capacity of the robot. For this purpose we introduce auxiliary non-decision capacity variables C_v for every waste vertex $v \in W$. These variables indicate the amount of waste in the robot after visiting the particular vertex. The non-decision character of the variables means that they are not instantiated by the search procedure, but they are instantiated by the inference procedure only. In particular, if their domain becomes empty during inference then it indicates inconsistency. The following constraints are used during the inference ($w \in W$). First, if the waste vertex w is visited directly after the collector then there is exactly one waste in the robot:

$$\sum_{e \in \mathbf{IN}(w)} X_e = 1 \implies C_w = 1 \tag{3.8}$$

Second, if the waste vertices u and v are visited directly before respectively after w (or vice versa) then the following constraints must hold between the capacity variables:

$$\forall e, f \in \mathbf{ICD}(w), e = \{u, w\}, f = \{w, v\} : X_e + X_f = 2 \implies |C_u - C_v| = 2 \tag{3.9}$$

$$\forall e = \{u, w\} \in \mathbf{ICD}(w) : |C_u - C_w| = 1 \tag{3.10}$$

Finally, to restrict the capacity of the robot by constant *cap* we use the following constraints for the capacity variables:

$$\forall w \in W : 1 \leq C_w \leq cap \tag{3.11}$$

The objective function to be minimised is the total cost of edges used in the solution path:

$$Obj = \sum_{e \in E} X_e \cdot weight(e) \tag{3.12}$$

where w$eight(e)$ is the weight of edge e.

Search Procedure

The constraint model describes how the inference is performed so the model needs to be accompanied by the search procedure that explores the possible instantiations of variables X_e. Our search strategy resembles the greedy approach for solving Travelling Salesman Problems (TSP) (Ausiello et al. [8]). The variable X_e for instantiation is selected in the following way. If the path is empty, we start at the initial position I and instantiate the variable $X_{\{I,w\}}$ such that w$eight(\{I, w\})$

is the smallest among the weights of arcs going from I. By instantiating the variable we mean setting it to 1; the alternative branch is setting the variable to 0. If the path is non-empty then we try to extend it to the nearest waste. Formally, if u is the last node in the path then we select the variable $X_{\{u,w\}}$ with the smallest $weight(\{u, w\})$, where w is a waste vertex. If this is not possible (due to the capacity constraint), we go to the closest collector. The optimisation is realised by the branch-and-bound approach: after finding a solution with the total cost *Bound*, the constraint $Obj < Bound$ is posted and the search continues until any solution is found. The last found solution is the optimum.

3.2.2 Model Based on Finite State Automata

The second model that we propose brings a radically new approach not seen so far when modelling VRPs or TSPs. Recall that we are looking for a path in the graph that satisfies some additional constraints. We can see this path as the word in a certain regular language. Hence, we can base the model on the existing regular constraint (Pesant [81]). This constraint allows a more global view of the problem so the hope is that it can infer more information than the previous model and hence decreases the search space to be explored.

First, it is important to realise that the exact path length is unknown in advance. Each waste vertex is visited exactly once, but the collector vertices can be visited more times and it is not clear in advance how many times. Nevertheless, it is possible to compute the upper bound on the path's length. Let us assume that the path length is measured as the number of visited vertices, the robot starts at the initial position and finishes at some collector vertex (we will use the dummy destination in a slightly different meaning here), and the weight/cost of arcs is non-negative. Let $K = |W|$ be the number of waste vertices and $cap \geq 1$ be the robot's capacity. Then the maximal path length is $2K + 1$. This corresponds to visiting a collector vertex immediately after visiting a waste vertex. Recall that each waste vertex must be visited exactly once and there is no arc between the collector vertices.

Our model is based on four types of constraints. First, there is a restriction on the existence of a connection between two vertices - a *routing constraint*. This constraint describes the routing network (see Figure 3.2). It roughly corresponds to the constraints 3.1-3.5 from the previous model. Note that the sub-tour elimination constraints 3.6-3.7 are not necessary here. Second, there is a restriction on the robot's capacity stating that there is no continuous subsequence of waste vertices whose length exceeds the given capacity - a *capacity constraint*. This constraint corresponds to the constraints 3.8-3.11 from the previous model. Third, each waste must be visited exactly once, while the collectors can be visited more times (even zero times) - an *occurrence constraint*. This restriction was included in the constraints 3.1-3.5 of the previous model, while we model it as a separate constraint. Finally, each arc is annotated by a weight and there is a constraint that the sum of the weights of used arcs does not exceed some limit - a *cost constraint*. This constraint is used to define the total cost of the solution as in 3.12.

In the constraint model we use three types of variables. Let $N = 2K + 1$ be the maximal path length. Then we have N variables $Node_i$, N variables Cap_i,

and N variables $\mathrm{C}ost_i (i = 1, \ldots, N)$ so we assume the path of maximal length. Clearly, the real path may be shorter so we introduce a dummy destination vertex that fills the rest of the path till the length N. In other words, when we reach the dummy vertex, it is not possible to leave it. This way, we can always look for the path of length N and the model gives flexibility to explore the shorter paths too.

The semantic of the variables is as follows. The variables $\mathrm{N}ode_i$ describe the path hence their domain is the set of numerical identifications of the vertices. We use positive integers $1, \ldots, K (K = |W|)$ to identify the waste vertices, $K+1, \ldots, K+L$ for the collector vertices ($L = |C|$), and 0 for the dummy destination vertex. In summary, the initial domain of each variable $\mathrm{N}ode_i$ consists of values $0, \ldots, K + L$. $\mathrm{C}ap_i$ is the used capacity of the robot after leaving vertex $\mathrm{N}ode_i (Cap_1 = 0$ as the robot starts empty), the initial domain is $\{0, \ldots, cap\}$. $\mathrm{C}ost_i$ is the cost of the arc used to leave the vertex $\mathrm{N}ode_i (\mathrm{C}ost_N = 0)$, the initial domain consists of non-negative numbers. Formally:

$$\begin{array}{rrcl} & 0 & \leq & \mathrm{N}ode_i \leq K + L \\ \forall i = 1, \ldots, N (N = 2K + 1): & 0 & \leq & \mathrm{C}ap_i \leq cap, \mathrm{C}ap_1 = 0 \\ & 0 & \leq & \mathrm{C}ost_i, \mathrm{C}ost_N = 0 \end{array} \tag{3.13}$$

We will start the description of the constraints with the *occurrence constraint* saying that each waste vertex is visited exactly once. This can be modelled using the global cardinality constraint [89] over the set $\{\mathrm{N}ode_1, \ldots, \mathrm{N}ode_N\}$. The constraint is set such that the each value from the set $\{1, \ldots, K\}$ is assigned to exactly one variable from $\{\mathrm{N}ode_1, \ldots, \mathrm{N}ode_N\}$ - each waste node is visited exactly once. The values $\{0, K+1, \ldots, K+L\}$ can be used any number of times. Formally:

$$\begin{array}{rc} gcc(\{\mathrm{N}ode_1, \ldots, \mathrm{N}ode_N\}, & \\ \{v: & [1,1] \forall v = 1, \ldots, K, \\ 0: & [0, \infty], \\ v: & [0, \infty] \forall v = K+1, \ldots, K+L\}) \end{array} \tag{3.14}$$

where $v : [\min, \max]$ means that value v is assigned to at least min and at most max variables from $\{\mathrm{N}ode_1, \ldots, \mathrm{N}ode_N\}$. The *gcc* constraint allows specifying the number of appearances of the value using another variable rather than using a fixed interval as in 3.14. Let D be the variable describing the number of appearances of value 0 (identification of the dummy vertex) in the set $\{\mathrm{N}ode_1, \ldots, \mathrm{N}ode_N\}$, then we can use the following constraints instead of 3.14:

$$\begin{array}{rc} gcc(\{\mathrm{N}ode_1, \ldots, \mathrm{N}ode_N\}, & \\ \{v: & [1,1] \forall v = 1, \ldots, K, \\ 0: & D, \\ v: & [0, \infty] \forall v = K+1, \ldots, K+L\}) \end{array} \tag{3.15}$$

$$\mathrm{N}ode_{N-D} > 0 \tag{3.16}$$

The constraint 3.16 says that $\mathrm{N}ode_{N-D}$ is not a dummy vertex; actually it is the last real vertex in the path. We can also set the upper bound for D by using

the information about the minimal path length (*MinPathLength* is a constant computed in advance):

$$D \leq N - \mathrm{M}inPathLength \tag{3.17}$$

These additional constraints 3.16 and 3.17 are not necessary for the problem specification but they improve inference (we use them in experiments).

The *cost constraint* can be easily described as

$$\mathrm{O}bj = \sum_{1,\ldots,N} \mathrm{C}ost_i \tag{3.18}$$

so we can use the constraints $\mathrm{O}bj < \mathrm{B}ound$ in the branch-and-bound procedure exactly the same way as in the previous model.

For the cost constraint to work properly we need to set the value of $\mathrm{C}ost_i$ variables. Recall that $\mathrm{C}ost_i$ is the cost/weight of the arc going from vertex $\mathrm{N}ode_i$ to vertex $\mathrm{N}ode_{i+1}$. Hence, we can connect the *Cost* variables with the *Node* variables when specifying the *routing constraint*. In particular, we use the ternary constraints over the variables $\mathrm{N}ode_i, \mathrm{C}ost_i, \mathrm{N}ode_{i+1}$ $i = 1, \ldots, N-1$. This set of constraints corresponds to the idea of slide constraint (Bessiere et al. [19]). We implement the constraint between the variables $\mathrm{N}ode_i, \mathrm{C}ost_i, \mathrm{N}ode_{i+1}$ as a ternary tabular (extensionally defined) constraint; let us call it link, where the triple (p, q, r) satisfies the constraint if there is an arc from the vertex p to the vertex r with the cost q. In other words, this table describes the original routing network with the costs extended by the dummy vertex. Formally:

$$\begin{aligned} \mathrm{l}ink(p,q,r) \equiv \quad & \exists e \in E : e = (p,r), q = \mathrm{w}eight(e) \\ & \vee (q = r = 0 \wedge (p = 0 \vee p > K) \end{aligned} \tag{3.19}$$

$$\forall i = 1, \ldots, 2K : \mathrm{l}ink(\mathrm{N}ode_i, \mathrm{C}ost_i, \mathrm{N}ode_{i+1}) \tag{3.20}$$

It remains to show how the *capacity constraint* is realised. Briefly speaking, we use a similar approach as for the routing constraint. The capacity constraint is realised using a set of ternary constraints over the variables $\mathrm{C}ap_i, \mathrm{N}ode_{i+1}, \mathrm{C}ap_{i+1}$ $i = 1, \ldots, N-1$, again exploiting the idea of slide constraint. The constraint is implemented using a tabular constraint, let us call it *capa*, with the following semantics. Triple (p, q, r) satisfies this constraint if and only if:

- q is an identification of a collector vertex ($q > K$) or a dummy vertex ($q = 0$) and $r = 0$
- q is an identification of a waste node ($0 < q \leq K$) and $r = p + 1$.

Recall that the domain of capacity variables is $\{0, \ldots, cap\}$ so we never exceed the capacity of the robot. Formally:

$$\begin{aligned} capa(p,q,r) \equiv \quad & (q = r = 0) \\ & \vee (q > K \wedge r = 0) \\ & \vee (0 < q \leq K \wedge r = p + 1) \end{aligned} \tag{3.21}$$

$$\forall i = 1, \ldots, 2K : capa(\mathrm{C}ap_i, \mathrm{N}ode_{i+1}, \mathrm{C}ap_{i+1}) \tag{3.22}$$

Any solution to the above described constraint satisfaction problem defines a valid solution of our single robot path planning problem with the capacity constraint. Vice versa, any solution to the path planning problem is also a feasible solution of the specified constraint satisfaction problem. We omit the formal proof due to limited space.

Search Procedure

Similarly to the previous model, it is important to specify the search strategy. In this second model, only the variables Node_i are the decision variables - they define the search space. It is easy to realise that the inference through the routing constraints 3.20 decides the values of the Cost_i variables and the inference through the capacity constraints 3.22 decides the values of the Cap_i variables provided that the values of all variables Node_i are known.

When searching for the solution we first use a greedy approach to find the initial solution (the initial cost). This greedy algorithm instantiates the variables Node_i in the order of increasing i in such a way that the arc with the smallest cost is preferred. We select the node in which the least expensive arc from the previously decided node leads. Naturally, the capacity constraint is taken into account so only the nodes in which the capacity is not exceeded are assumed. This search procedure corresponds to the search strategy of the previous model. The difference in models allows us to use a fixed variable ordering in the model based on finite automata which simplifies implementation of the search procedure. This second model also has fewer decision variables but a larger branching factor.

To find the optimal solution we use a standard branch-and-bound approach with restarts. To instantiate the Node variables we use the *min-dom* heuristic for the variable selection, that is, the variable with the smallest current domain is instantiated first. We select the values in the order defined in the problem (the waste nodes are tried before the collector nodes). Exactly as in the first model after finding a solution with the total cost *Bound*, the constraint O$bj <$ B$ound$ is posted and the search continues until any solution is found. The last found solution is the optimum. Note that using the well known and widely applied min-dom heuristic for the variable selection is meaningful in this model because we have larger domains, while the same heuristic is useless for the previous model which uses binary domains.

3.2.3 Embedding CP Models into Local Search

The current state of the art techniques for solving VRPs are frequently based on hybrid approaches. For example, the paper [87] suggests using CP techniques to explore the neighbourhood within large neighbourhood search. We applied a similar approach with our CP models to check if the solution quality can be improved in comparison with the pure branch-and-bound approaches presented above.

The basic elements in the neighbourhood local search are the concept of the neighbourhood of a solution and the mechanism for generating neighbourhoods. It is eminent that the performance and "success" of the local search algorithm strongly depends on the neighbourhood operator and its state space. In our case, the state corresponds to the plan - a valid path for the robot. The local

search algorithm is repeatedly choosing another solution in the neighbourhood of the current solution with the goal to improve the value of the objective function. This move is realised by a so called *neighbourhood operator*. We have implemented an operator that is successfully used for solving the travelling salesman problems (TSP). The operator relaxes the solution by removing an induced path of a given length and then it calls the CP solver to complete the solution. It means that we add to a given constraint model additional constraints that fix some edges (for the model based on network flows) or forbid using some edges (for the model based on finite state automata). These fixed edges correspond to the edges in the original solution that were not removed by the neighbourhood operator. The role of the CP solver is to optimally complete this partial solution by adding the missing edges. The new solution is the state in which the local search procedure moves.

As the local search repeatedly chooses a move that improves the value of the objective function (we are minimizing the value), it can get "trapped" in a local minimum. We utilized the simplified simulated annealing as a method of escaping from this trap.

As the initial solution for local search we used the first solution obtained from the pure CP model (see the description of the search procedures above).

3.3 Experimental Results

In this section we will present the experimental evaluation of the discussed solving techniques. As there is no standard benchmark set for the studied problem, we generated new problem instances. We used a square-sized robot arena where the positions of the waste and the initial location of the robot were uniformly distributed. The collectors were uniformly distributed along the boundaries of the arena and the weights set up as a point-to-point distance using the Euclidean metric. All the following measurements were performed on `Intel Xeon CPU@2.5GHz` with 4GB of RAM, running a Debian GNU Linux operating system.

3.3.1 Performance of the Network Flow Model

As stated earlier, the model based on network flows corresponds to the traditional operations research approach. However, we modified the model to describe specifics of our robot routing problem. The model was implemented in *Java SE 6* using *Choco*, an open-source constraint programming library. The optimisation search strategy uses the built-in branch-and-bound method, while all constraints correspond to the mathematic formulations described earlier.

Figure 3.4 shows the runtime (a logarithmic scale) to obtain the optimal solution as a function of the instance size measured by the number of waste and by the number of collectors. We generated 15 instances for each problem size and the graph shows the average time the solver needs for finding and proving the optimality of the solution. The capacity of the robot was 3.

As already mentioned in [13], the satellite facilities in VRP (or collectors in our formulation) heavily increase the complexity of the problem. The initial experiment shows that the runtime is increased exponentially with the number of waste, but it is not significantly affected by the increased number of collectors.

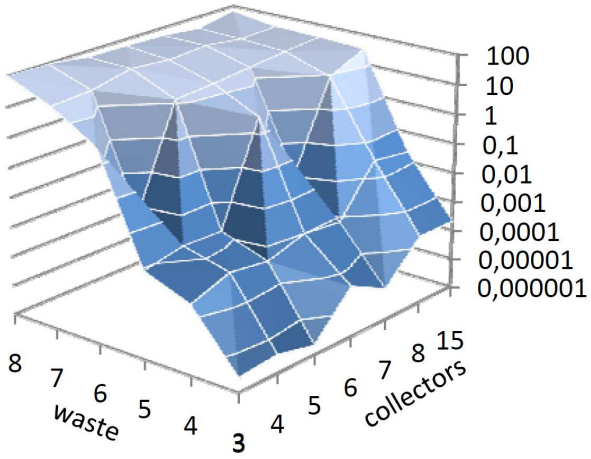


Figure 3.4: Runtime (seconds) for the network flow model.

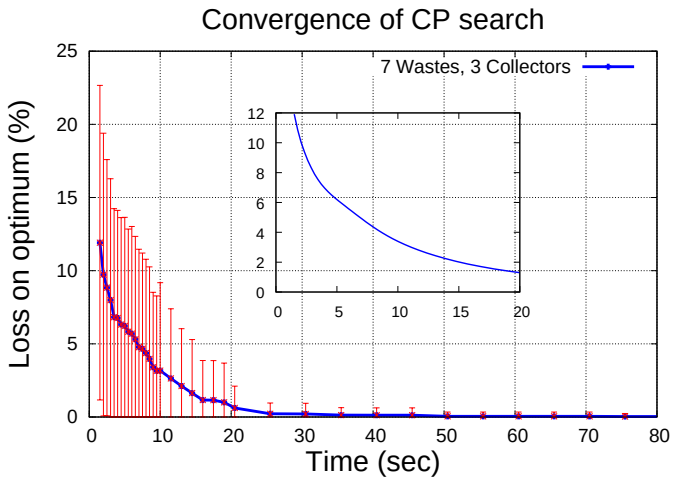


Figure 3.5: Quality convergence for the network flow model.

In fact, it seems that for different quantities of waste there are different numbers of collectors, where the best runtime is achieved. The hypothesis is that for a given number of waste there is some specific number of collectors that gives the best result. Nevertheless, confirmation of this hypothesis requires additional experiments.

While the graph in Figure 3.4 represents the total time the solver needs for finding and proving the optimality of the solution, we are even more interested in how fast a "good enough" solution can be found. This characteristic can be seen in Figure 3.5, where the graph displays the convergence of the solution during the search. We can see that even a simple greedy heuristic performs very well and the difference from the optimal solution was less than 5% within the first 6 seconds for the instance $7 + 3$.

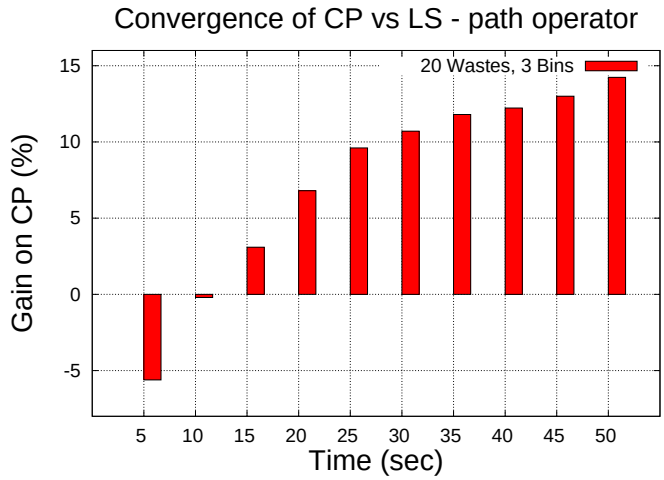


Figure 3.6: Comparison of the quality convergence of the network flow model in the pure CP approach and the CP model embedded into local search.

3.3.2 Performance of the Network Flow Model within Local Search

As mentioned above, the CP model can be used within the large neighbourhood search procedure to solve larger instances, but obviously without any guarantee of optimality. We generated 50 independent problem instances with 20 wastes and 3 collectors (referred to as $20 + 3$). The capacity of the robot was set to 7 units. The neighbourhood operator was allowed to remove 5 randomly selected consecutive edges during the search and the embedded CP solver was allowed to search for 1 second. The graph in Figure 3.6 shows an average one-to-one performance of the pure CP method and the LS method (with the embedded CP model) applied to the produced instances. The graph shows the difference in the quality of a solution found in the corresponding time from the LS viewpoint.

The local search procedure performed better in the long run, when compared to the pure CP method relaying only on its inner heuristic. However, CP beat LS in the first seconds where the convergence drop was steeper. As a consequence, CP seems to be a more appropriate method under very short time constraints, while reasonably good solutions can be found with a combination of LS for larger instances.

3.3.3 Performance of the Finite State Automaton Model

The network flow model represents a standard approach to solving the vehicle routing problems. Therefore, we compared our novel constraint model based on the finite state automaton directly to this approach. The second model was implemented in *S*ICStus Prolog [1]. Figure 3.7 shows the runtime (a logarithmic scale) to obtain the optimal solution using the constraint model based on finite state automata using the same problems as with the model based on network flows (Figure 3.4). As can be seen, there is an exponential growth with the increased number of waste and a weaker dependence on the number of collectors.

[1]SICStus Prolog: http://www.sics.se/sicstus

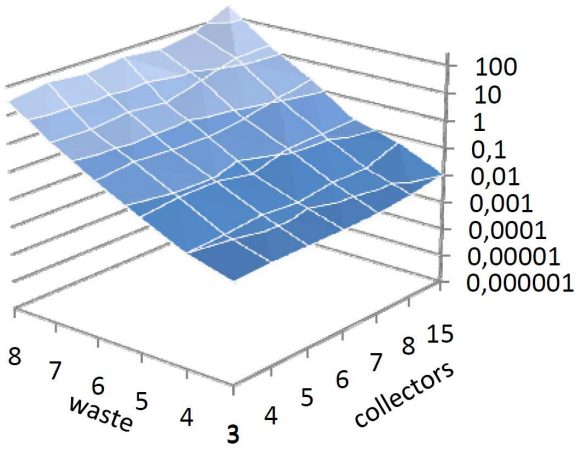


Figure 3.7: Runtime (seconds) for the model based on finite state automata.

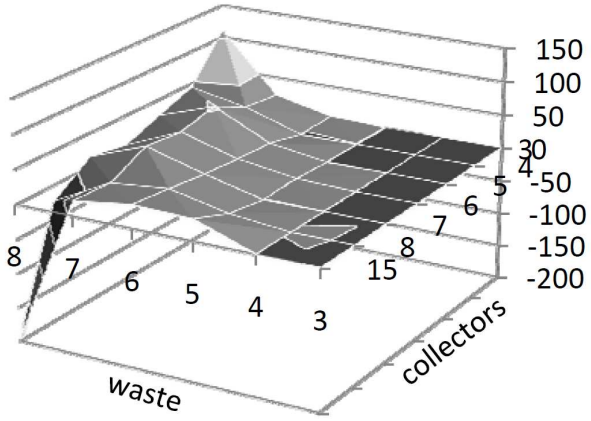


Figure 3.8: Time difference (seconds) between the CP models. Positive values means that the model based on finite state automata is faster.

To directly compare both models, we generated a graph showing the difference of runtimes for the network model and for the automata model - the values above zero mean faster automata model, while the times below zero mean faster network model. Figure 3.8 shows difference times. As the graph shows, the automata-based model is visibly better for a smaller number of collectors where the problem is more constrained and the capacity constraints can prune more of the search space. A bit surprisingly, it seems that the network-based model is better when the number of collectors becomes larger.

Since in robotics, finding a good plan (relatively) fast is more important than having the optimal one late, we investigated again the quality of the plans found by this CP solver in a constrained time. In particular, we embedded the new CP model in the large neighbourhood search procedure as described above and we compared the pure CP model with this LS approach on much bigger instances with 40 wastes and 3 collectors. To our surprise, the LS method was not able to improve the solution found by the CP model in the 2 minutes runtime. As it

is required to produce a good solution in seconds, the pure CP model based on finite state automaton seems more appropriate.

3.4 Conclusions

In this chapter, we presented a high-level planner and demonstrated its performance on a practical application. The waste cleaning problem is an interesting variant of VRP problem. The constraint programming techniques allowed us to naturally define the underlying model for which the solver was able to find the first solution in hundreds of microseconds on problems of reasonable sizes. Loosely speaking, this is a form of anytime planning, where a significantly suboptimal solution may be initially chosen and improved during execution.

The robotics domain has some strict requirements on high-level planners: at any point a plan should be available for execution and its quality should increase over time. The planner based on CP paradigm fullfills these requirements and fits well into the overall architecture discussed in the previous section.

We used a constraint model based on network flows that is traditionally applied to this type of routing problems and we developed a completely new model based on finite automata. We further studied local search techniques that are traditionally used to improve the runtime performance of CP models for vehicle routing problems and we have found that our novel model based on finite automata performs better without them.

In summary, there are two novel contributions. First, we reformulated the traditional network flow model to solve the waste collecting problem with the limited capacity of the robot. Second, we proposed a novel constraint model based on finite automata (state transitions), and we experimentally showed that it outperforms the traditional approach if the number of waste collecting places is small.

Conclusions

This chapter summarizes the work achieved in the thesis and its main results. In addition, several possible directions for further research are discussed.

Main Results

This work deals with the complex problem of designing control algorithms for adaptive agents. In the first chapter, we studied algorithms that can develop sensible behavior out of a "tabula rasa". We utilized evolutionary algorithms and reinforcement learning, the two most common learning paradigms in robotics, which are intended to work with little or no human intervention. Both methodologies developed effective controllers and showed a comparable performance in a simple obstacle avoidance experiment.

In the second experiment, we studied a more complex problem, which involved the coordination of multiple robots. We resorted to the evolutionary learning methodology and naturally extended the fitness function from the previous experiments. A group of simulated robots was evolved to show the ability of collective homing behavior. The evolutionary technique turned out to be a very effective approach for this task and the collective behavior was clearly present in the evolved controllers. However, despite its extensibility, the fitness measure was too specific and biased by human view. Ideally, the fitness function should be a survival criterion, automatically translated into sensible behavior by self-executing design methodology. An example of such aggregate fitness function was given in the active learning experiment. The agent alone "realized" benefits coming from reducing uncertainty in its environment and performed actions appropriate to reach this goal.

In general, as the complexity of an environment and task increases, there is no path in the search space that can be discovered by the limited sample of potential solutions considered and more prior knowledge is required. More human bias must be inserted into the learning process and basic principles of automated design are violated. One way to incorporate adaptive elements into more elaborate problems is to make use of algorithms tailored for particular tasks. In the social SLAM experiment, we exploited the infomax control algorithm, designed for tasks that require the active learning. In order to do so, we had to develop quantitative models of the sensory motor system. Compared to the previous experiment when the active learning was naturally emerged, it required more effort.

The class of synthetized behaviors is often limited to reactive and rather simple behaviors. In the obstacle avoidance experiment, three types of neural networks were able to develop the exploration behavior and typical behavioral patterns, such as following the right wall, which resulted in the very efficient exploration of an unknown maze. We hypothesized that more powerful neural network architecture would lead towards the development of better controllers, however, we were not able to confirm this hypothesis. The automatic design methodology did not exploit more powerful network architectures.

To overcome limitations of a purely reactive agent, an agent that exploits a hybrid architecture was introduced in the second chapter. Hybrid architectures

combine both the reactive and deliberative planning and they are appropriate for advanced mobile agents that have to deal with localization and mapping. These algorithms are usually implemented as interacting layers. We utilized a the three layer architecture, which contained a reactive collision avoidance module, a modern path-planning component based on the the value iteration algorithm and a high-level planner. Although path planning algorithms are extensively studied and well working with robots equipped with an advanced sensory system, we demonstrated that they show reasonable performance even with low-cost E-puck robots in a laboratory environment.

The deliberative planner, based on pure constraint programming paradigm, was fully described in the third chapter. We presented its performance on a practical application — the waste cleaning problem, which is an interesting variant of popular vehicle routing problems. We discussed the strict requirements on high-level planners in the robotics domain. Deliberative planners must implement a form of anytime planning, where a significantly suboptimal solution may be initially chosen and improved during execution. The proposed high-level planner based on constraint programming paradigm fullfills these requirements and fits well into the overall architecture. However, implementation of such a planner for the specific task is not the low-hanging fruit. In our first attempt, we reformulated the traditional network flow model to solve the waste collecting problem with the limited capacity of the robot. Eventually we proposed a novel constraint model based on finite automata (state transitions) and we experimentally showed that it outperforms the traditional approach, if the number of waste collecting places is small.

In general, the main goal of our work was to study and develop control algorithms for autonomous embodied agents. One approach is through a self-organization process based on evolutionary or reinforcement learning. The advantages of such an approach are the undisputed fact, but as discussed in this thesis, the designer still has to face some unresolved problems that are tied with these methods. The alternative approach, which requires the designer to carefully prototype overall architecture and individual components, is widely utilized and more successful in modern robotics. Although adaptive elements may be present in such systems, the burden connected with their implementation falls to the human. Cognitive abilities of such agents are biased by human view and limited to particular implementations.

Future Work

The field of aritificial intelligence is making progress by leaps and bounds. However, the journey to the fully automated and intelligent adaptive agents is certainly long and there are still numerous aspects that need to be addressed. The thesis would be incomplete if we did not mention the possible directions of further research.

Recent research has successfully led to a range of are computationally efficient probabilistic algorithms, for a range of hard robotics problems, like localization or motion planning. However, the task planners often lack the explicit uncertainty handling. In the future, we would like to focus on this area. The high-level planner introduced in the last chapter did not accept feedback from the localiza-

tion module. However, the measure of uncertainty could have an impact on the decision making process.

The results reported above represent just a few steps in the journey toward intelligent agents. Cognitive agent functions are still very limited. We discussed approaches based on neural networks, genetic algorithms or reinforcement learning. Reinforcement learning seems to be a good framework for lifelong learning, that provides transfer of the induced knowledge across domains and agents. We would like to focus on the continuous adaptation to rapidly changing environments in the future. As the boundary between deliberation and reactive execution is fixed in our experiments, it is agent's Achilles' heel, especially if environmental diversity is taken into account. This will be the topic of our future research, too.

Bibliography

[1] Cyberbotics robot curriculum.

[2] E-puck, online documentation. In *http://www.e-puck.org*.

[3] *Khepera II documentation*. http://k-team.com.

[4] *Webots simulator*. http://www.cyberbotics.com/.

[5] Henry H AAAI Spring Symposium, Hexmoor, David Kortenkamp, and American Association for Artificial Intelligence, editors. *Lessons learned from implemented software architectures for physical agents papers from the 1995 AAAI Symposium, March 27-29, Stanford, California*. AAAI Press, Menlo Park, Calif., 1995.

[6] Ronald C. Arkin. *Behavior-Based Robotics*. The MIT Press, May 1998.

[7] Ronald C. Arkin and Douglas C. Mackenzie. Planning to behave: A hybrid Deliberative/Reactive robot control architecture for mobile manipulation. In *International Symposium on Robotics and Manufacturing, Maui, HI*, page 5–12, 1994.

[8] Giorgio Ausiello, M. Protasi, A. Marchetti-Spaccamela, G. Gambosi, P. Crescenzi, and V. Kann. *Complexity and Approximation: Combinatorial Optimization Problems and Their Approximability Properties*. Springer, 1999.

[9] Bruno De Backer, Vincent Furnon, Paul Shaw, Philip Kilby, and Patrick Prosser. Solving vehicle routing problems using constraint programming and metaheuristics. *Journal of Heuristics*, 6(4):501–523, 2000.

[10] Barrie M. Baker and M. A. Ayechew. A genetic algorithm for the vehicle routing problem. *Comput. Oper. Res.*, 30(5):787–800, 2003.

[11] G. Baldassarre, S. Nolfi, and D. Parisi. Evolving mobile robots able to display collective behaviour. *Hemelrijk C. K (ed.), International Workshop on Self-Organisation and Evolution of Social Behaviour*, page 11–22, 2002.

[12] G. Baldassarre, Vito Trianni, M. Bonani, F. Mondada, M. Dorigo, and S. Nolfi. Self-organized coordinated motion in groups of physically connected robots. *IEEE Transactions on Systems, Man, and Cybernetics, Part B: Cybernetics*, 37(1):224–239, February 2007.

[13] Jonathan F. Bard, Liu Huang, Moshe Dror, and Patrick Jaillet. A branch and cut algorithm for the VRP with satellite facilities. *IIE Transactions*, 30(9):821–834, 1998.

[14] Roman Barták, Michal Zerola, and Stanislav Slusny. Towards routing for autonomous robots - using constraint programming in an anytime path planner. In Joaquim Filipe and Ana L. N. Fred, editors, *ICAART 2011 - Proceedings of the 3rd International Conference on Agents and Artificial*

Intelligence, Volume 1 - Artificial Intelligence, Rome, Italy, January 28-30, 2011, pages 313–320. SciTePress, 2011.

[15] Richard K. Belew and Melanie Mitchell. *Adaptive Individuals in Evolving Populations: Models and Algorithms*. Addison-Wesley, 1996.

[16] John E. Bell and Patrick R. McMullen. Ant colony optimization techniques for the vehicle routing problem. *Advanced Engineering Informatics*, 18(1):41 – 48, 2004.

[17] Richard Bellman. *Dynamic Programming*. Princeton University Press, Princeton, NJ, USA, 1 edition, 1957.

[18] D. Bertsekas and J. Tsitsiklis. *Neuro-dynamic programming*. Ahtena Scientific, 1996.

[19] Christian Bessiere, Emmanuel Hebrard, Brahim Hnich, Zeynep Kiziltan, Claude-Guy Quimper, and Toby Walsh. Reformulating global constraints: the slide and regular constraints. In *Proceedings of the 7th International conference on Abstraction, reformulation, and approximation*, SARA'07, page 80–92, Berlin, Heidelberg, 2007. Springer-Verlag.

[20] Valentino Braitenberg. *Vehicles: Experiments in Synthetic Psychology*. MIT Press, 1986.

[21] R. Brooks. A robust layered control system for a mobile robot. *Robotics and Automation, IEEE Journal of*, (1):14 – 23, 1986.

[22] Rodney A. Brooks. Integrated systems based on behaviors. *SIGART Bull.*, 2(4):46–50, 1991.

[23] D.S. Broomhead and D. Lowe. Multivariable functional interpolation and adaptive networks. *Complex Systems*, 2:321–355, 1988.

[24] Wolfram Burgard, Armin B. Cremers, Dieter Fox, Dirk Hähnel, Gerhard Lakemeyer, Dirk Schulz, Walter Steiner, and Sebastian Thrun. Experiences with an interactive museum tour-guide robot. *Artificial Intelligence*, 114(1–2):3–55, October 1999.

[25] Y.U. Cao, A.S. Fukunaga, A.B. Kahng, and F. Meng. Cooperative mobile robotics: antecedents and directions. In *1995 IEEE/RSJ International Conference on Intelligent Robots and Systems 95. 'Human Robot Interaction and Cooperative Robots', Proceedings*, volume 1, pages 226–234 vol.1, August 1995.

[26] Jaime Carbonell, Oren Etzioni, Yolanda Gil, Robert Joseph, Craig Knoblock, Steve Minton, and Manuela Veloso. PRODIGY: an integrated architecture for planning and learning. *SIGART Bull.*, 2(4):51–55, July 1991.

[27] J. Carlsson and T. Ziemke. YAKS - yet another khepera simulator. *Ruckert, S., Witkowski (Eds.), Autonomous Minirobots for Research and Entertainment Proceedings of the Fifth International Heinz Nixdorf Symposium*, 2001.

[28] Jean-Francois Cordeau, Gilbert Laporte, Martin W.P. Savelsbergh, and Daniele Vigo. Vehicle routing. In C. Barnhart and G. Laporte, editors, *Transportation, Handbooks in Operations Research and Management Science*, volume 14, pages 367–428. Elsevier, 2007.

[29] Petrica C.Pop. New integer programming formulations of the generalized t ravelling salesman problem. *American Journal of Applied Sciences*, 11:932–937, 2007.

[30] Charles Darwin. *The origin of species: by means of natural selection of the preservation of favoured races in the struggle for life*. New American Library, New York, N.Y., 2003.

[31] Rina Dechter. *Constraint Processing*. Morgan Kaufmann Publishers, San Francisco, CA, USA, May 2003.

[32] Gregory Dudek, Michael R. M. Jenkin, Evangelos Milios, and David Wilkes. A taxonomy for multi-agent robotics. *Autonomous Robots*, 3(4):375–397, December 1996.

[33] S. Dzeroski, L. De Raedt, and K. Driessens. Relational reinforcement learning. *Machine Learning 43*, pages 7–52, 2001.

[34] J. Elman. Finding structure in time. *Cognitive Science 14*, page 179–214, 1990.

[35] D. Floreano and F. Mondada. Automatic creation of an autonomous agent: Genetic evolution of a neural network driven robot. *Proceedings of the third international conference on Simulation of adaptive behavior: From Animals to Animats 3*, pages 421–430, 1994.

[36] Dario Floreano and Stefano Nolfi. Adaptive behavior in competing coevolving species. In *PROCEEDINGS OF THE FOURTH EUROPEAN CONFERENCE ON ARTIFICIAL LIFE*, page 378–387. MIT Press, 1997.

[37] D. B. Fogel. *Evolutionary Computation: The Fossil Record*. MIT-IEEE Press, 1998.

[38] Erann Gat. *Reliable Goal-directed Reactive Control of Autonomous Mobile Robots*. PhD thesis, Virginia Polytechnic Institute & State University, Blacksburg, VA, USA, 1991. UMI Order No. GAX91-23728.

[39] Mohammad Ghavamzadeh, Sridhar Mahadevan, and Rajbala Makar. Hierarchical multi-agent reinforcement learning. *Autonomous Agents and Multi-Agent Systems*, 13(2):197–229, 2006.

[40] F. Gruau. Cellular encoding of genetic neural networks. Technical report 92-21, Laboratoire de l'Informatique du Parallilisme. Ecole Normale Supirieure de Lyon, France, 1992.

[41] S. A. Harp, T. Samad, and A. Guha. Towards the genetic synthesis of neural networks. In *Proceedings of the Third International Conference on Genetic Algorithms*, page 360–369, 1989.

[42] I. Harvey, P. Husbands, D. Cliff, A. Thompson, and N. Jakobi. Evolutionary robotics: the sussex approach. *ROBOTICS AND AUTONOMOUS SYSTEMS*, 20:205–224, 1997.

[43] S. Haykin. *Neural Networks: A Comprehensive Foundation*. Prentice Hall, 2nd edition, 1998.

[44] J. Holland. *Adaptation In Natural and Artificial Systems*. MIT Press, reprinted edition, 1992.

[45] Tommi Jaakkola, Michael I. Jordan, and Satinder P. Singh. Convergence of stochastic iterative dynamic programming algorithms. *Neural Computation*, 6:1185–1201, 1994.

[46] Leslie Pack Kaelbling, Michael L. Littman, and Andrew W. Moore. Reinforcement learning: a survey. *Journal of Artificial Intelligence Research*, 4:237–285, 1996.

[47] R. E. Kalman. A new approach to linear filtering and prediction problems. *Transactions of the ASME Journal of Basic Engineering*, (82 (Series D)):45, 35, 1960.

[48] K. Kanazawa, D. Koller, and S. J. Russel. Stochastic simulation algorithms for dynamic probabilistic networks. In *Proceedings of the 11th Annual Conference on Uncertainty in AI*, Montreal, Canada.

[49] K. Kawai, A. Ishiguro, and P. Eggenberger. Incremental evolution of neurocontrollers with a diffusion-reaction mechanism of neuromodulators. In *2001 IEEE/RSJ International Conference on Intelligent Robots and Systems, 2001. Proceedings*, volume 4, pages 2384–2391 vol.4, 2001.

[50] Hiroaki Kitano. Designing neural networks using genetic algorithms with graph generation system. *Complex Systems Journal*, 4:461–476, 1990.

[51] Kurt Konolige. COLBERT: a language for reactive control in sapphira. In Gerhard Brewka, Christopher Habel, and Bernhard Nebel, editors, *KI-97: Advances in Artificial Intelligence*, number 1303 in Lecture Notes in Computer Science, pages 31–52. Springer Berlin Heidelberg, January 1997.

[52] Steven M. LaValle. *Planning Algorithms*. Cambridge University Press, May 2006.

[53] Hector J. Levesque, Raymond Reiter, Yves Lespérance, Fangzhen Lin, and Richard B. Scherl. GOLOG: a logic programming language for dynamic domains. *The Journal of Logic Programming*, 31(1–3):59–83, April 1997.

[54] Deepak Kumar Lisa A. Meeden. Trends in evolutionary robotics. 1999.

[55] A. Martinoli. *Swarm intelligence in autonomous Collective robotics: from tools to the analysis and synthesis of distributed control strategies.* Lausanne: Computer Science Department, EPFL, 1999.

[56] Maja Mataric and Dave Cliff. *Challenges in Evolving Controllers for Physical Robots*. 1996.

[57] Amy L. Lansky Michael P. Georgeff and Marcel J. Schoppers. Reasoning and planning in dynamic domains: An experiment with a mobile robot. Technical Report 380, AI Center, SRI International, 333 Ravenswood Ave., Menlo Park, CA 94025, April 1987.

[58] Tom M. Mitchell. *Machine Learning*. McGraw-Hill Science/Engineering/Math, 1 edition, March 1997.

[59] David J. Montana and Lawrence Davis. Training feedforward neural networks using genetic algorithms. In *Proceedings of the 11th International Joint Conference on Artificial Intelligence - Volume 1*, IJCAI'89, page 762–767, San Francisco, CA, USA, 1989. Morgan Kaufmann Publishers Inc.

[60] Michael Montemerlo, Nicholas Roy, and Sebastian Thrun. Perspectives on standardization in mobile robot programming: The carnegie mellon navigation (CARMEN) toolkit. In *In Proc. of the IEEE/RSJ Int. Conf. on Intelligent Robots and Systems (IROS*, page 2436–2441, 2003.

[61] J. Moody and C. Darken. Fast learning in networks of locally-tuned processing units. *Neural Computation*, 1:289–303, 1989.

[62] Andrew W. Moore and Christopher G. Atkeson. Prioritized sweeping: Reinforcement learning with less data and less time. In *Machine Learning*, page 103–130, 1993.

[63] Andrew L. Nelson, Gregory J. Barlow, and Lefteris Doitsidis. Fitness functions in evolutionary robotics: A survey and analysis. *Robot. Auton. Syst.*, 57(4):345–370, April 2009.

[64] R. Neruda and P. Kudová. Learning methods for RBF neural networks. *Future Generations of Computer Systems*, page 1131–1142, 2005.

[65] R. Neruda and S. Slusny. Variants of memetic and hybrid learning of perceptron networks. In *18th International Workshop on Database and Expert Systems Applications, 2007. DEXA '07*, pages 158–162, September 2007.

[66] R. Neruda, S. Slusny, and P. Vidnerova. Performance comparison of relational reinforcement learning and RBF neural networks for small mobile robots. In *Second International Conference on Future Generation Communication and Networking Symposia, 2008. FGCNS '08*, volume 4, pages 29–32, December 2008.

[67] Roman Neruda. Experiments with evolutionary and hybrid learning of multi-layer perceptron neural networks. 2007.

[68] Roman Neruda and Stanislav Slušný. Parameter genetic learning of perceptron networks. In *Proceedings of the 10th WSEAS International Conference on Systems*, ICS'06, page 92–97, Stevens Point, Wisconsin, USA, 2006. World Scientific and Engineering Academy and Society (WSEAS).

[69] Roman Neruda and Stanislav Slušný. Performance comparison of two reinforcement learning algorithms for small mobile robots. *International Journal of Control and Automation.*, (1):59–68, 2009.

[70] Roman Neruda, Stanislav Slušný, and Petra Vidnerová. Evolution of simple behavior patterns for autonomous robotic agent. In *Proceedings of the 6th WSEAS International Conference on System Science and Simulation in Engineering*, ICOSSSE'07, page 411–417, Stevens Point, Wisconsin, USA, 2007. World Scientific and Engineering Academy and Society (WSEAS).

[71] Roman Neruda, Stanislav Slušný, and Petra Vidnerová. Behavior emergence in autonomous robot control by means of evolutionary neural networks. In Sio-Iong Ao, Burghard Rieger, and Su-Shing Chen, editors, *Advances in Computational Algorithms and Data Analysis*, number 14 in Lecture Notes in Electrical Engineering, pages 235–247. Springer Netherlands, January 2009.

[72] Roman Neruda and Petra Vidnerová. Learning algorithms for small mobile robots: Case study on maze exploration. In *Vojtáš, P. (ed.). Information Technologies - Applications and Theory.*, 2008.

[73] Slusny Stanislav Neruda Roman. Two learning approaches to maze exploration: Case study with e-puck mobile robots. In *Lecture Notes in Engineering and Computer Science*, pages 655–660, San Francisco, October 2008.

[74] S. Nolfi. Adaptation as a more powerful tool than decomposition and integration. In T.Fogarty and G.Venturini, editors, *Proceedings of the Workshop on Evolutionary Computing and Machine Learning, 13th International Conference on Machine Learning*, 1996.

[75] S. Nolfi. The power and limits of reactive agents. Technical report, Rome, 1999.

[76] S. Nolfi and D. Floreano. *Evolutionary Robotics — The Biology, Intelligence and Techology of Self-Organizing Machines.* The MIT Press, 2000.

[77] Stefano Nolfi. *Evolving non-Trivial Behaviors on Real Robots: a garbage collecting robot.* 1996.

[78] Stefano Nolfi and Dario Floreano. *Evolutionary Robotics: The Biology, Intelligence, and Technology of Self-Organizing Machines.* The MIT Press, March 2004.

[79] Stefano Nolfi and Davide Marocco. Evolving robots able to integrate sensory-motor information over time. *Theory in Biosciences*, 120(3-4):287–310, December 2001.

[80] J. Peng and Ronald J. Williams. Efficient learning and planning within the dyna framework. In *, IEEE International Conference on Neural Networks, 1993*, pages 168–174 vol.1, 1993.

[81] Gilles Pesant. A regular language membership constraint for finite sequences of variables. In *Principles and Practice of Constraint Programming*, page 482–495, 2004.

[82] Jan Peters, Sethu Vijayakumar, and Stefan Schaal. Natural actor-critic. In João Gama, Rui Camacho, Pavel B. Brazdil, Alípio Mário Jorge, and Luís Torgo, editors, *Machine Learning: ECML 2005*, number 3720 in Lecture Notes in Computer Science, pages 280–291. Springer Berlin Heidelberg, January 2005.

[83] Tomaso Poggio and Federico Girosi. A theory of networks for approximation and learning. Technical report, Massachusetts Institute of Technology, 1989.

[84] Martin L. Puterman. *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. John Wiley & Sons, Inc., New York, NY, USA, 1st edition, 1994.

[85] Matt Quinn, Lincoln Smith, Giles Mayley, and Phil Husbands. Evolving controllers for a homogeneous system of physical robots: Structured cooperation with minimal sensors. *Philosophical Transactions: Mathematical, Physical and Engineering Sciences*, 361(1811):2321–2343, October 2003. ArticleType: primary_article / Issue Title: Biologically Inspired Robotics / Full publication date: Oct. 15, 2003 / Copyright Â© 2003 The Royal Society.

[86] Krzysztof R.Apt. *Principles of Constraint Programming*. Cambridge University Press, 2003.

[87] Louis-Martin Rousseau, Michel Gendreau, and Gilles Pesant. Using constraint-based operators to solve the vehicle routing problem with time windows. *Journal of Heuristics*, 8(1):43–58, 2002.

[88] Reuven Y. Rubinstein. *Simulation and the Monte Carlo Method*. Wiley-Interscience, May 1981.

[89] Jean-Charles Régin. Generalized arc consistency for global cardinality constraint. In *Proceedings of the thirteenth national conference on Artificial intelligence - Volume 1*, AAAI'96, page 209–215, Portland, Oregon, 1996. AAAI Press.

[90] Burr Settles. Active learning literature survey. Technical report, 2010.

[91] L. G. Shapiro and G. C Stockman. *Computer Vision*. Prentence Hall, 2001.

[92] Paul Shaw. Using constraint programming and local search methods to solve vehicle routing problems. In *CP98*, volume 1520 of *LNCS*, pages 417–431. Springer, 1998.

[93] Helmut Simonis. Constraint applications in networks. In Francesca Rossi, Peter van Beek, and Toby Walsh, editors, *Handbook of Constraint Programming*, pages 875–903. Elsevier, 2006.

[94] Satinder P Singh. Learning to solve markovian decision processes. Technical report, University of Massachusetts, Amherst, MA, USA, 1993.

[95] S. Slusny and R. Neruda. Evolving homing behaviour for team of robots. In *Computational Intelligence, Robotics and Autonomous Systems. Palmerston North : Massey University*, 2007.

[96] Stanislav Slusny, Roman Neruda, and Petra Vidnerová. Rule-based analysis of behaviour learned by evolutionary and reinforcement algorithms. In *ICIC (2)*, page 284–291, 2008.

[97] Stanislav Slušný and Roman Neruda. Local search heuristics for robotic routing planner. In Derong Liu, Huaguang Zhang, Marios Polycarpou, Cesare Alippi, and Haibo He, editors, *Advances in Neural Networks – ISNN 2011*, number 6677 in Lecture Notes in Computer Science, pages 31–40. Springer Berlin Heidelberg, January 2011.

[98] Stanislav Slušný, Roman Neruda, and Petra Vidnerová. Behaviour patterns evolution on individual and group level. In *Proceedings of the 6th WSEAS International Conference on Computational Intelligence, Manmachine Systems and Cybernetics*, CIMMACS'07, page 23–28, Stevens Point, Wisconsin, USA, 2007. World Scientific and Engineering Academy and Society (WSEAS).

[99] Stanislav Slušný, Roman Neruda, and Petra Vidnerová. Comparison of RBF network learning and reinforcement learning on the maze exploration problem. In Véra Kůrková, Roman Neruda, and Jan Koutník, editors, *Artificial Neural Networks - ICANN 2008*, number 5163 in Lecture Notes in Computer Science, pages 720–729. Springer Berlin Heidelberg, January 2008.

[100] Stanislav Slušný, Roman Neruda, and Petra Vidnerová. Comparison of behavior-based and planning techniques on the small robot maze exploration problem. *Neural Networks*, 23(4):560–567, May 2010.

[101] Stanislav Slušný, Petra Vidnerová, and Roman Neruda. Testing different evolutionary neural networks for autonomous robot control. In *[ITAT 2007. Conference on Theory and Practice of Information Theory*, 2007.

[102] Stanislav Slušný and Michal Zerola. Plánovanie cesty založené na programovaní s obmedzujúcimi podmienkami. In *Informačné Technológie - Aplikácie a Teória. S. 87-92. - Seňa : Pont, 2010 / Pardubská D.*, 2010.

[103] Stanislav Slušný, Michal Zerola, and Roman Neruda. Real time robot path planning and cleaning. In *Proceedings of the Advanced Intelligent Computing Theories and Applications, and 6th International Conference on Intelligent Computing*, ICIC'10, page 442–449, Berlin, Heidelberg, 2010. Springer-Verlag.

[104] Roman Neruda Stanislav Slusny. Localization with a low-cost robot. In *Vojtáš, P. (ed.). Information Technologies - Applications and Theor*, pages 77–80, 2009.

[105] Richard Sutton and Andrew Barto. *Reinforcement Learning: An Introduction (Adaptive Computation and Machine Learning)*. A Bradford Book, March 1998.

[106] Richard S. Sutton. Learning to predict by the methods of temporal differences. In *MACHINE LEARNING*, page 9–44. Kluwer Academic Publishers, 1988.

[107] Richard S. Sutton. Planning by incremental dynamic programming. In *In Proceedings of the Eighth International Workshop on Machine Learning*, page 353–357. Morgan Kaufmann, 1991.

[108] Matthew E. Taylor, Shimon Whiteson, and Peter Stone. Comparing evolutionary and temporal difference methods in a reinforcement learning domain. In *Proceedings of the 8th Annual Conference on Genetic and Evolutionary Computation*, GECCO '06, page 1321–1328, New York, NY, USA, 2006. ACM.

[109] Sebastian Thrun, Wolfram Burgard, and Dieter Fox. *Probabilistic Robotics (Intelligent Robotics and Autonomous Agents)*. The MIT Press, 2005.

[110] H. Tijms, Stochastic Models, John Wiley, and Sons (qa T. *Primary Reference S.M. Ross, Introduction to Stochastic Dynamic Programming, Academic Press (T57.83 R67). Other References*.

[111] P. Vidnerova, S. Slusny, and R. Neruda. Evolutionary trained radial basis function networks for robot control. In *10th International Conference on Control, Automation, Robotics and Vision, 2008. ICARCV 2008*, pages 833–838, December 2008.

[112] Petra Vidnerová, Stanislav Slušný, and Roman Neruda. Emergence chování robotických agentů: neuroevoluce. In *Kognice a umělý život VIII.*, pages 295–299, May 2008.

[113] Christopher John Cornish Hellaby Watkins. *Learning from Delayed Rewards*. PhD thesis, King's College, Cambridge, UK, May 1989.

[114] D Whitley, T Starkweather, and C Bogart. Genetic algorithms and neural networks: optimizing connections and connectivity. *Parallel Computing*, 14(3):347–361, August 1990.

[115] Ronald Williams and Leemon C. Baird. Tight performance bounds on greedy policies based on imperfect value functions. Technical report, 1993.

[116] Xin Yao. A review of evolutionary artificial neural networks. *International Journal of Intelligent Systems*, 8(4):539–567, 1993.

List of Tables

List of Figures

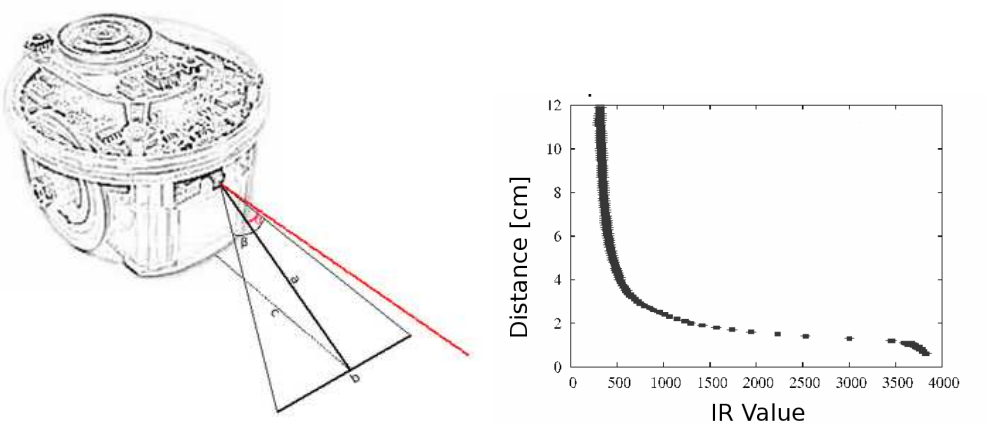
List of Abbreviations

AI	Artificial Intelligence
ANN	Artificial Neural Networks
CP	Constraint Programming
CSP	Constraint Satisfaction Problem
DP	Dynamic Programming
EA	Evolutionary Algorithm
ER	Evolutionary robotics
GA	Genetic Algorithm
LS	Local Search
LDS	Limited Discrepancy Search
MCL	Monte Carlo Localization
MLP	Multilayer Perceptron Network
MDP	Markov Decision Process
MDT	Markov Decision Task
NN	Neural Network
POMDP	Partially Observable Markov Processes
RBF	Radial Basis Function
RL	Reinforcement Learning
SLAM	Simultaneous Localization And Mapping
TSP	Travelling Salesman Problem
VRP	Vehicle Routing Problem
VRPSF	Vehicle Routing Problem with Satellite Facilities
VRPTW	Vehicle Routing Problem with Time Windows

Attachments

1 Miniature Robots

Khepera [3] (Figure 1.4, left) is a miniature mobile robot with a diameter of 70 mm and a weight of 80 g. The robot is supported by two lateral wheels that can rotate in both directions and two rigid pivots in the front and in the back. The sensory system employs eight "active infrared light" sensors distributed around the body, six on one side and two on other side. In "passive mode", they measure the amount of infrared light in the environment, which is roughly proportional to the amount of visible light. In "active mode" these sensors emit a ray of infrared light and measure the amount of reflected light. The closer they are to a surface, the higher is the amount of infrared light measured. The Khepera sensors can detect a white paper at a maximum distance of approximately 5 cm. The robot is equipped with a Motorola MC68331 CPU with 512 Kbytes of EEPROM and 512 Kbytes of static RAM.



Left: The physical parameters of the E-puck real camera (picture taken from [1]). Camera settings used in experiments corresponds to parameters $a = 6$ cm, $b = 4.5$ cm, $c = 5.5$ cm, $\alpha = 0.47$ rad, $\beta = 0.7$ rad. **Right:** Properties of E-puck IR sensor.

Parameters	Value
Maximum translational velocity	12.8 cm / sec
Maximum rotational velocity	4.86 rad / sec
Stepper motor maximum speed	+- 1000 steps / sec
Distance between tires	5.3 cm

Velocity parameters of E-puck mobile robot.

E-puck [2] (Figure 1.4, right) is a follower of the Khepera robot, as it follows similar concept. E-puck is a widely used robot for scientific and educational

purposes - it is open-source and low-cost. E-puck is a mobile robot with a diameter of 70 mm and a weight of 50 g. The sensory system employs eight "active infrared light" sensors distributed around the body, six on one side and two on other side. Similarly to Khepera robot, these sensors can work in active or passive mode. E-puck sensors can detect a white paper at a maximum distance of approximately 8 cm. Sensors return values from interval $[0, 4095]$. Effectors accept values from interval $[-1000, 1000]$. The higher the absolute value, the faster is the motor moving in either direction. Unfortunately, because of their imprecision and characteristics, they should be used as bumpers only. As can be seen, they provide high resolution only within few millimeters. They are very sensitive to the obstacle surface, as well. Besides infrared sensors, the robot is equipped with a low-cost VGA camera with resolution 52×39 pixels. Despite its limitations, the camera can be used to detect objects or landmarks. However, the information about distance to the landmark extracted from the camera is not reliable (due to the noise). The robot is supported by two lateral wheels that can rotate in both directions and two rigid pivots one in the front and one in the back. Two stepper motors support the movement of the robot. A stepper motor is an electromechanical device which converts electrical pulses into discrete mechanical movements. It can divide a full rotation into a 1000 steps, the maximum speed corresponds to about a rotation every second. The robot is equipped with 16-bit processor dsPIC 30F6014A running at 60 Mhz and 8 KB of memory.

2 Rules Induced in the Obstacle Avoidance Task

	Sensor			Motor	
	left	**front**	**right**	**left**	**right**
250	FEEL	NOWHERE	NOWHERE	500	500
172	NOWHERE	NOWHERE	NOWHERE	100	500
105	VERYFAR	NOWHERE	NOWHERE	500	500
98	FEEL	NOWHERE	FEEL	500	500
68	NOWHERE	NOWHERE	FEEL	100	500
67	FAR	NOWHERE	NOWHERE	500	300
60	FEEL	FEEL	NOWHERE	500	300
51	NEAR	NOWHERE	NOWHERE	500	100
20	VERYFAR	FEEL	NOWHERE	500	300
18	NOWHERE	FEEL	NOWHERE	300	300
15	NEAR	FEEL	NOWHERE	500	-100
15	FAR	FEEL	NOWHERE	500	-100
10	FAR	FAR	NOWHERE	500	-500
9	NEAR	FAR	NOWHERE	500	-500
8	NOWHERE	FEEL	FEEL	300	300
6	FEEL	VERYFAR	NOWHERE	500	-100
5	VERYFAR	FAR	NOWHERE	500	-500
5	NEAR	VERYFAR	NOWHERE	500	-500
3	NOWHERE	VERYFAR	FEEL	500	100
3	NEAR	NEAR	NOWHERE	500	-500
2	FEEL	FAR	NOWHERE	500	-100
2	FEEL	FAR	FEEL	500	-100
1	VERYNEAR	NEAR	NOWHERE	500	-500
1	VERYNEAR	FAR	NOWHERE	500	-500
1	VERYFAR	VERYFAR	NOWHERE	500	-500
1	VERYFAR	NEAR	FEEL	500	-500
1	NOWHERE	FAR	FEEL	500	100
1	FEEL	NEAR	FEEL	500	-500
1	FAR	VERYFAR	NOWHERE	500	-500
1	FAR	NEAR	NOWHERE	500	-500

Evolved rules used by the RBF network in the training arena. The first column states how many times was the rule utilized in the experiment.