Introduction to Adaptive Systems

Marco A. Wiering

Preface

This syllabus contains six chapters for the course "Introduction to Adaptive Systems". Other topics which will be studied in this course are quite well covered by Tom Mitchell's book: Machine Learning. For the topics which are not described as complete chapters, we will include the slides used for presentations as technical material. There are still many citations missing and there may be occasional errors in the text. I would appreciate it if you could write down corrections of this text and deliver it to me. I hope you will enjoy this course and pass it with a high grade!

Marco A. Wiering

Contents

Chapter 1

Introduction

Everywhere around us we can observe change, in fact without change life would be extremely boring. Life implies change, since if there would not be any change anymore in the universe, everything would be dead. Physicists think it is likely that after very many years (think about 10^{500} years), the universe would stop changing and enter a state of thermal equilibrium in which it is extremely cold (near the absolute minimum temperature) and in which all particles (electrons, neutrinos and protons) are isolated and stable (in this stable state even dark holes will have evaporated). This means that the particles will not interact anymore and change will stop. This view relies on the theory that the universe is expanding — and this expansion is accelerating which is implied by a positive cosmological constant (the energy density of vacuum). The theory that the universe would contract again after some while (which may imply a harmonic universe) is not taken very serious anymore nowadays. So, after a long time, the universe will reach a stable state without change. Fortunately since this takes so long, we should not worry at the moment. Furthermore, there are some thoughts that intelligent life may change all of this.

A realistic model of any changing system (e.g. the weather or the stock market) consists of a description of the state at the current time step and some function or model which determines in a deterministic way (only 1 successor state is possible) or in a stochastic way (there are multiple possible successor states which may occur with some probability) the next state given the current state. We will call the state of the system at time-step t: $S(t)$. It is clear that if we examine the state of the system over time, that there is a sequence of states: $S(t), S(t+1), \ldots, S(t+n), \ldots$. Such a sequence of states is often referred to as the state-trajectory of the system. Note that we often consider time to be discrete, that is that all time-steps are positive natural numbers: $t \in \{0, 1, 2, \ldots, \infty\}$. We only use discrete time due to computational reasons and simplicity, since representing continuous numbers on a computer using bit-representations is not really feasible (although very precise approximations are of course possible). Mathematically, we could also consider time as being continuous, although the mathematics would involve some different notation.

1.1 Adaptive Systems

Although the "objective" state of the universe would consist of a single representation of all elements, and therefore a single state, in reality we can observe different objects which can be modelled as separate elements. Therefore instead of a single state at time t: $S(t)$, we

may consider the world to consist of l objects and write $S_i(t)$ where $1 \leq i \leq l$, to denote the state of object i at time t. In this way there are trajectories for all different objects. If all these objects would evolve completely separately, the universe would basically consist of many sub-universes, and we can look at the trajectory of every single object alone. However, in most real systems, the objects will **interact**. Interaction means that the state of some object influences the trajectory of another object, e.g. think about Newton's laws in which gravity causes attraction from one object to another one.

At this point we are ready to understand what an adaptive system is.

> **An adaptive system** is a system in which there is interaction between the system and its environment so that both make transitions to changing states.

Of course it may happen that after a long period of time, the adaptive system enters a stable state and does not change anymore. In that case we still speak of an adaptive system, but if the adaptive system never made transitions to different states, it would not be an adaptive system. So the first requirement is that an adaptive system is dynamic (changing), at least for a while. Sometimes an adaptive system is part of another system. Think for example about some robot which walks in a room, but does not displace any objects in that room. We have to think about this situation as a room which has a robot inside of it. Since the robot is changing its position, the room is also changing. So in this case the robot is the adaptive system and the room is the changing environment.

Another requirement for an adaptive system is that the adaptive system will change itself or its environment using its trajectory of states in order to attain a goal that may be to **simulate** some process — to understand what will happen under some conditions, (e.g. we can simulate what happens if we put ten sharks in a pool and do not feed them), or the goal to **optimize** something (e.g. a robot which keeps the floors clean).

Finally there can be **learning** adaptive systems that have the ability to measure their own performance and are able to change their own internal knowledge parameters in order to improve their performance. In this case we say that the adaptive system is optimizing its behavior for solving a particular task. If we call the state of the internal knowledge parameters: $S_I(t)$ then learning means to change the state of the internal knowledge parameters after each iteration (time-step) so learning will cause a trajectory: $S_I(t), S_I(t+1), \ldots, S_I(T)$ where the final state $S_I(T)$ may be a stable state and has (near)-optimal performance on the task. When you are not acquainted with machine learning, a learning computer system may seem strange. However, machine learning receives a lot of interest in the artificial intelligence community nowadays, and learning computer programs certainly exist. A very simple example of learning is a computer program which can decide between option 1 and option 2. Each time it selects option 1 the environment (possibly a human teacher) tells the system that it was a success, and each time the program selects option 2 it is told that it is a failure. It will not be surprising that with a simple learning program the system will quickly always select option 1 and optimizes its performance. More advanced learning systems such as for speech recognition, face recognition, or handwritten text recognition are also widely spread.

Other terms which are very related to adaptive systems are: cybernetics, self-organising systems, and complex adaptive systems. The term cybernetics as it is used nowadays stems from Norbert Wiener and is motivated in his book: Cybernetics: or, Control and Communication in the Animal and the Machine (1948). Before Norbert Wiener worked on gunfire control. Freudenthal wrote about this:

> *While studying anti-aircraft fire control, Wiener may have conceived the idea of considering the operator as part of the steering mechanism and of applying to him such notions as feedback and stability, which had been devised for mechanical systems and electrical circuits. ... As time passed, such flashes of insight were more consciously put to use in a sort of biological research ... [Cybernetics] has contributed to popularising a way of thinking in communication theory terms, such as feedback, information, control, input, output, stability, homeostasis, prediction, and filtering. On the other hand, it also has contributed to spreading mistaken ideas of what mathematics really means.*

There are many adaptive systems to be found, some examples are:

- Robots which navigate through an environment with some particular goal (e.g. showing visitors of a museum a sequence of different objects or helping people in elderly homes to walk around in the corridors)
- Learning systems which receive data and output knowledge, e.g. classifying the gender of humans using photos of their faces, or recognising speech from recorded and annotated speech fragments
- Automatic driving cars or unmanned aerial vehicles (UAVs)
- Evolutionary systems in which the distribution of the gene-pool adapts itself to the environment
- Economical systems in which well performing companies expand and bad performing ones go out of business
- Biological systems such as earthquakes or forest fires

1.2 Intelligent Agents

A fairly new concept in artificial intelligence is an **Agent**. The definition of an agent is a computer system that is situated in some environment, and that is capable of autonomous action in this environment in order to meet its design objectives. An agent possesses particular characteristics such as:

- Autonomy: The agent makes its own choices based on its (virtual) inputs of the environment; even if a user tells the agent to drive of a cliff, the agent can refuse
- Reactivity: Agents are able to perceive their environment, and respond in a timely fashion to changes that occur in it in order to satisfy their design objectives
- Pro-activeness: Agents are able to exhibit goal-directed behavior by taking the initiative in order to satisfy their design objectives
- Social Ability: Intelligent agents are capable of interacting with other agents (and possibly humans)

Examples of agents are robots, mail-clients, and thermostats. The advantages of using the agent metaphor becomes clear when we have to control a system (e.g. a robot). First of all it becomes easier to speak about the sensory inputs which an agent receives from its environment though its (virtual) sensors. Using the inputs and possibly its current internal state, the agent selects an action. The action leads to a change in the environment. The agent usually has goals which it should accomplish. There can be goals of achievement (reaching a particular goal state) or maintenance goals (keeping a desired state of the system). The goals can often be easily modelled as a reward function which sends the agent utility values for reaching particular states. The reward function could also give a reward (or penalty which is a negative reward) for individual actions. E.g. if the task for a robot-agent is to go to office R12 as soon as possible, the reward function could emit -1 for every step (a penalty) and a big reward of +100 if the agent reaches the desired office.

An intelligent agent can perceive its environment, reason, predict, and act (using its actuators). A **rational agent** acts to maximize its performance measure so that it will reach its goal with the least amount of effort. An **autonomous agent** acts according to its own experiences. So it does not execute a fixed algorithm which always performs the same operations (such as a sorting algorithm), but uses its perceptions to direct its behavior. The agent is modelled in a **program** which is executed on an **architecture** (computer, hardware). The program, architecture, and environment determine the behavior of the agent.

1.3 Model for Adaptive Systems

We now want to make a formal model of an adaptive system which interacts with an environment. The objective state of the world is the state of the world at some time-step. Often the adaptive system does not perceive this complete state, but receives (partial) inputs from the environment. Next to current inputs from the environment, the system can have beliefs about the world from its past interaction with the environment. Furthermore, the agent can perform a number of actions, and chooses one of them at every time-step. The control method which uses beliefs and inputs to select an action is often referred to as the policy. There is also a transition function which changes the state of the world according to the previous state and the action that the agent executed. Then there is a reward function which provides rewards to the agent after executing actions in the environment. Finally the system requires a function to update the internal (belief) state. So when we put these together, we get a model $\mathrm{M} = < t, S, I, B, A, \pi, T, R, U >$ with:

- A time-element $t = \{1, 2, 3, \ldots\}$
- A state of the environment at time t: $S(t)$
- An input of the environment received at time t: $I(t)$
- An internal state (belief) of the agent at time t: $B(t)$
- A number of possible actions A with $A(t)$: the action executed by the agent at time t.
- A policy which maps the input and belief to an action of the agent: $\pi(I(t), B(t)) \rightarrow A(t)$
- A transition-rule which maps the state of the environment and the action of the agent to a new state of the environment: $T(S(t), A(t)) \rightarrow S(t+1)$

- A reward-function which gives rewards to the system, for this there are two possibilities, depending on whether the reward function is located in the environment so that we get: $R(S(t), A(t)) \rightarrow R(t)$ or when the reward function is located in the agent and the agent cannot know $S(t)$ we have to use: $R(I(t), B(t), A(t)) \rightarrow R(t)$.
- An update function for the internal (belief) state of the agent $U(I(t), B(t), A(t)) \rightarrow B(t+1)$.

We can note a number of causal relations in the model which are depicted in Figure 1.1.

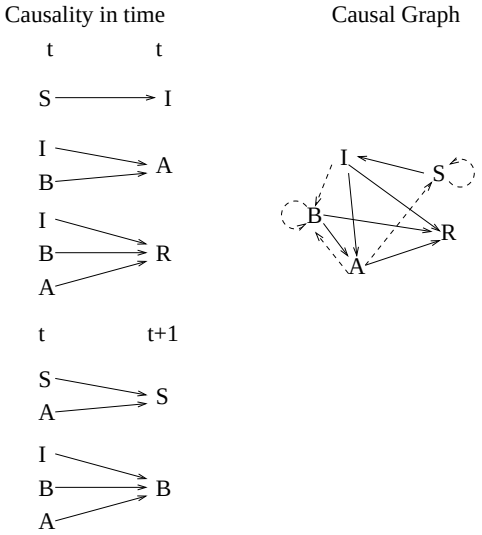

Figure 1.1: The relations between the different elements of an adaptive system.

If we study the figure, we can see that there is one big feedback loop, going from Belief to Action to State to Input to Belief. So Belief influences belief on a later time-step. Note that not all adaptive systems use an internal state (belief), we will go into this in more detail later.

1.3.1 Reward function

An agent usually has one or more goals which it wants to achieve or maintain. To formalise the notion of goal, one could use qualitative goals which can be true or false, such as $Goal(go\_home)$. Such qualitative goals are usually used in logical agents that try to make a plan using operators which bring the current state to a goal state (the plan can be computed forwards from the current state to the goal or alternatively backwards from the goal state to the current state). Another possibility is to use a more quantitative notion of a goal using a reward signal which is emitted after each time-step. The advantage of the latter is that it becomes easier to distinguish between multiple plans which bring about a trajectory which attains a specific goal. E.g. if an agent uses 100 steps or 20 steps to find the kitchen, then clearly using 20 steps should be preferred. However, when qualitative goals are used, they both become true after some time. Even if the planner tries to come up with the shortest plan, efforts to execute the plan are not easily incorporated. Using a reward function we can emit after each step a reward of -1 (so a cost of 1) and for reaching the goal, the agent may

be rewarded with a high bonus. In this way shorter paths are preferred. Furthermore, when different actions require different effort, we can use different costs for different actions (e.g. when climbing a mountain it costs usually a lot of effort to take steep paths). In decision theory usually utilities or reward signals are used. The goal for the agent then becomes to maximize its obtained rewards in its future. So its policy should maximize:

$$\sum_{t=0}^{\infty} \gamma^t R(t) \tag{1.1}$$

Where $0 \leq \gamma \leq 1$ is the discount factor which determines how future rewards are traded off against immediate rewards. E.g. if we find it is important to get a lot of reward during the current day and are not interested in the examination tomorrow, we will set the discount factor to a very low number, maybe resulting in drinking a lot of beer in a bar and failing the examination tomorrow. However, if we are interested in life-long happiness, we should use a high discount factor (close to 1).

1.3.2 The internal state

Often no internal state (IS) is used, but without internal state we can only construct a **reactive agent**. A reactive agent uses a policy which maps the current input to an action. It does not use any memory of previous inputs or actions. For a game like chess, a reactive agent is perfect, because it does not really matter how a particular board-position came about, the best move only depends on the current state of the board which is fully accessible (completely observable) for the agent. However, in case you are looking for a restaurant and someone tells you "go straight until the second traffic light and then turn left." Then you have to use memory, because if you would see a traffic light you cannot know whether to turn left or not without knowing (remembering) that you have seen another traffic light before.

In more complex agents, internal state is very important. Note that we define the internal state as a recollection of past inputs and performed actions and not the knowledge learned by the agent about how to perform (this knowledge is in the adaptive policy). If an agent has to count to ten, it can map the next number using the previous one and does not need to remember what was before. In such cases there is therefore only a previous state which is the input for the policy. If the agent has to remember the capital of the United States, and uses it a long time afterwards, then it uses some kind of internal memory, but in some cases it would use long-term memory that is stored in the policy by learning the response to the question "what is the capital of the US?" Therefore we can speak of long-term and short-term memory, and the long-term memory resides usually in the policy (or knowledge representation) whereas short-term information which needs to be remembered only for a while is stored in short-term memory or the internal state. When we speak about belief (e.g. facts which are believed by the agent with some probability), however, it can also be stored in long-term memory, and therefore it would be better to make a distinction between short-term internal state and long-term belief. For acting one would still use knowledge stored in the policy, although this would usually be procedural knowledge (for learned skills) in contrast to declarative knowledge (knowledge and beliefs about the world). For now we just use the distinction between internal state (to remember facts) which is the short-term changing belief or a policy for acting.

Humans possess a very complex internal state. If you close your eyes and ears, and stop focusing on your senses, then you do not receive any inputs from the environment. But

still, thoughts arise. These thoughts come from the internal state, most often the thoughts are about things which happened not so long ago (like a minute ago, today or yesterday). Of course you can also act and direct your thoughts, in this way your brain becomes the environment and there is an interaction between you and your brain. Therefore when you think about how it would be to walk on the beach, you use your imagination and some policy for choosing what to do next. In that case, the internal state is only there to remind you of the start of the walk on the beach and whether you saw the sun shining or not. In many forms of meditation, one should close her eyes and concentrate on breathing. In this way, there is no information at all in the brain, basically one starts to think about nothing at all. In that case, there is no input and a diminishing internal state until it becomes empty too, and this may cause a very relaxing experience. Note that meditation is not the same as sleeping, some people say that sleeping is inside the inactive consciousness and meditation is in the subconscious where people are still experiencing things, but can concentrate on some thoughts (such as nothingness) much better. Finally, the opposite of a yogi is someone who has schizophrenia. In schizofrenia, one believes very much in the current internal state, and the actions focus on the information present in the internal state. So new inputs which disprove strange ideas residing in the internal state are almost not taken into account, and it is very difficult to convince such people that they are living in a reality set up by themselves without any logic or correspondence to the real world.

1.4 Total System Perspective

An adaptive system (e.g. an agent) interacts with an environment. In principle there may be multiple agents acting in the environment, and it is important to understand the interaction between the agents and their environment. Therefore we usually have to look at the total system which consists of the smaller parts. Looking at the complete system gives different possible views on what the agents are and what they should do. For example, examine forest fire control, the entities which play a role are the trees, fire-men, bulldozers, air-planes, fire, smoke columns, the weather etc. If we examine these entities, we can easily see that only the bulldozers, fire-men, and air-planes can be controlled, and therefore we can make them an agent with their own behavior, goals, etc. Sometimes it is not so easy to abstract from reality; we do not want to model all details, but we want a realistic interaction between the agent and the environment.

Example 1. Examine a restaurant, which entities play a role and which could be modelled as an agent? If we examine possible scenarios we can exploit our creativity on this topic. For example the entities may be the kitchen, tables, chairs, cook, waiter, lights, etc. Now we might consider to make them all agents, e.g. lights which dim if some romantic couple is sitting below them, tables and chairs which can move by themselves so that a new configuration of tables can be made automatically when a large group of people enters the restaurant etc. Would such as futuristic restaurant not be nice to visit?

1.4.1 An example: a room heater with a thermostat

Consider a thermostat for a room heater which regulates the temperature of a room. The heater uses the thermostat to measure the temperature of the room. This is the input of the system. The heater has actions: heat, or do-nothing. The temperature of the room will decrease (until some lower limit value) if the heater does not heat the room, and the

temperature of the room will increase if the heater is on. Figure 1.2 shows the interaction between the heater and the temperature of the room.

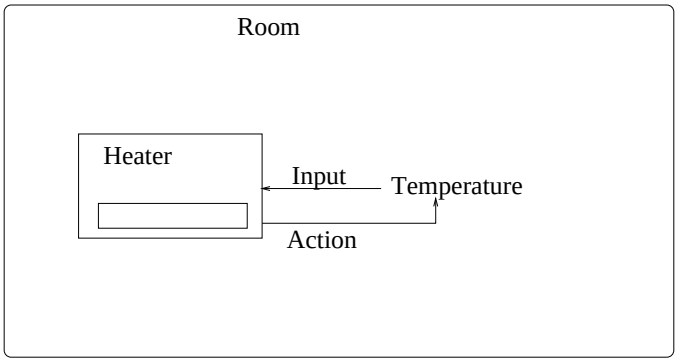

Figure 1.2: The interaction between a heater and the temperature in a room.

Making a model for the heater

The state of the environment which should first be modelled is the temperature of the room at a specific time. Since this is the environmental state, we denote it as $S(t)$. The input of the heater is in this case also the temperature of the room (although it might contain noise due to imprecise measurements), we denote this input as $I(t)$. The internal state of the heater is denoted as $B(t)$ and it can take on values whether the heater is on (heating) or whether it is off (doing nothing). The possible actions of the heater are: heat or do nothing.

Policy of the heater. Now we have to design the policy of the heater which is the most important element, since this is our control objective. Of course we can design the policy in many possible ways, but if there is a reward function, the control policy should be the one which optimizes the cumulative reward over time. The construction of the policy can be done by manual design, although it could also be learned. We will not go into details at this moment how learning this policy should be done, instead we manually design a policy since it is easy enough to come up with a good solution (so learning is not required). An example policy of the heater uses the following if-then rules:

1. If $I(t) \leq 21$ then heat
2. If $I(t) > 21$ and $I(t) \leq 23$ and $B(t) == heat$ then heat
3. If $I(t) > 21$ and $I(t) \leq 23$ and $B(t) == do\_nothing$ then do-nothing
4. If $I(t) > 23$ then do-nothing

If we examine the rules, we can see they are exclusive, at each time-step only one rule can be applied (sometimes the application of a rule is called a firing rule). If rules would overlap, the system would become more complex, since some mechanism should then be constructed which chooses the final decision. Research in fuzzy logic uses membership functions for rules, e.g. if the temperature is warm then do-nothing. The membership function then determines whether it is warm, e.g. is 24 degrees warm, and 27 degrees? This membership function should

be designed (although it may also be learned) and the rules all fire using their activation which is given by the application of the membership functions to the input. After this all actions are integrated using the activations as votes. We will not go into detail into fuzzy logic here, but just mention that it can be used when it is difficult to set absolute thresholds for rules (such as 23 degrees in the above example).

Another issue which is important is that the used policy creates a **negative feedback loop**. This means that if the temperature goes up, the heater will stop to increase the temperature, so that the temperature will go down again. In this way the system remains stable between the temperature bounds. If we would create a policy which would heat the room more when the temperature becomes higher, we would create a **positive feedback loop**, leading to a temperature which becomes very hot until possibly the heater will break down. It is therefore important to note that negative feedback loops are important for stable systems, although positive feedback loops can also be useful, e.g. if one want to have a desired speed very fast, the system can increase the speeds with larger and larger jumps until finally a negative feedback loop would take over.

Another way to construct the policy is to use decision trees. A decision tree makes a choice by starting at the root node of the tree and following branches with choice labels until we finally arrive at a leave node which makes a decision. A decision tree which is equivalent to the set of above rules is shown in Figure 1.3.

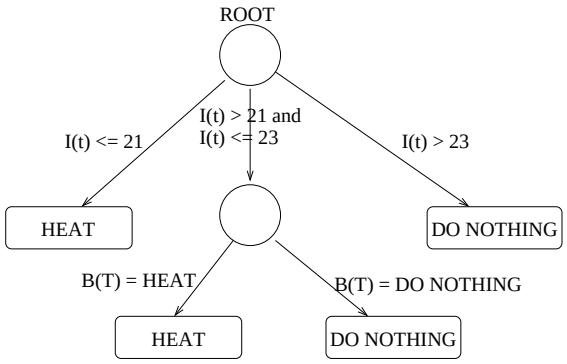

Figure 1.3: The policy of the heater designed as a decision tree.

The update and transition function. To make the model of the system complete, we also have to specify how we update the belief and environmental transition function. In our simple model, these are easily obtained (although the environmental transition function might depend on a lot of different factors such as the temperature outside, whether a door or window is open etc.). The belief update function is modelled as follows:

- $U(*, *, heat) \rightarrow heat$
- $U(*, *, do\_nothing) \rightarrow do\_nothing$

Where $*$ denotes the don't care symbol which can take on any value for the function (or rule) to be applied. So the update function for the internal state or belief just remembers the previous action. We make the following simple transition function of the environment (in reality this transition function does not have to be known, but we construct it here to

make our model complete). If the heater is on then the temperature will increase (let's say that it is a simple linear increasing function, which is of course not true in reality due to the effect that there is an upper limit of the temperature, and that more heat will be lost due to interaction with the outside when the temperature difference is larger. In reality the heat-loss is a linear function of the temperature difference, but in our model we do not include the outside temperature, since then isolation will also be important and we get too many details to model). We also make a simple transition function when the heater is off. So using our simple assumptions we make the following environmental transition function:

- $T(S(t), heat) \rightarrow S(t) + 0.1$
- $T(S(t), do\_nothing) \rightarrow S(t) - 0.05$

The reward function is only needed for self-adapting systems. However, we can also use it as a measurement function on the performance of a policy. Let's say that we want the room's temperature to remain close to 22 degrees, then the reward function may look like:

$$R(I, *, *) = -(I - 22)^2$$

Dynamics of the interaction

When we let the heater interact with the temperature of the room, we will note that there will be constant change or dynamics of a number of variables. The following variables will show dynamics:

- The state of the environment $S(t)$
- The input of the heater (in this case equal to the state of the environment): $I(t)$
- The action of the heater: $A(t)$
- The received reward: $R(t)$
- The internal state of the heater (in this case equal to the previous action of the heater): $B(t)$

If we let the temperature of the room start at 15 degrees, we can examine the dynamics of the room's temperature (the state of the environment). This is shown in Figure 1.4.

1.5 Environments

The interaction with the environment depends a lot on the environment itself. We can make a very simple system which shows very complex behavior when the environment is complex. One good example of this is Simon's ant. Herbert Simon is a well-known researcher in artificial intelligence and he thought about a simple ant which follows the coast line along the beach. Since the waves make different complex patterns on the beach, the ant which follows the coast line will also show complex behavior, although the design of this ant may be very simple.

On the other hand, the environment can also make the design of a system much more complicated. There are some characteristics of environments which are important to study, before we can understand how complex the construction of a well performing system will be. The following characteristics of environments are most important:

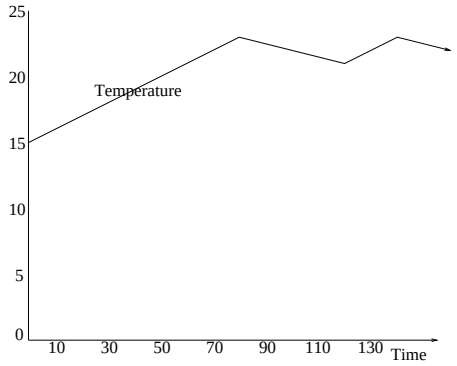

Figure 1.4: The dynamics of the room's temperature while interacting with the heater with the given policy. Note that there is a repetition in the dynamics.

- **Completely / Partially observable.** The question here is about the perception of the agent of the environment. Can it perceive the complete state of the environment through its (virtual) sensors? Then the environment is completely observable, this is for example the case in many board-games (but not in Stratego).

- **Deterministic / Non-deterministic.** If the next state of an environment given the previous state and action of an agent is always unique, then it is a deterministic environment. If the successor state can be one of many possible states, usually a probability distribution is used and then the environment is non-deterministic (also called stochastic).

- **Episodic / Non-episodic.** If the task requires always a single interaction with the environment, then the interaction with the environment is episodic. In case a complete sequence of actions should be planned and executed, the interaction with the environment is non-episodic.

- **Static / Dynamic.** If the environment does not change when we do not regard the action of the agent, then the environment is static. In case the environment changes on its own independently of the action of the agent, we say the environment is dynamic. In case the reward function changes, we say the environment is **semi-dynamic**.

- **Discrete / Continuous.** If the state of the environment only uses discrete variables such as in chess, the environment is discrete. If continuous variables are necessary to accurately describe the state of the environment, the environment is continuous (as is the case with robotics where the position and orientation are continuous).

If we consider these dimensions to characterise the environment, it will not be surprising that the environments that are most complex to perfectly control are partially observable, non-deterministic, non-episodic, dynamic, and continuous. We may always be able to try to simulate these environments, although a good model is also complicated (as for example for weather prediction).

We can make a list of environments and show the characteristics of these environments. Figure 1.5 shows such a mapping of tasks (and environments) to characteristics.

Environment	Completely observable	Deterministic	Episodic	Static	Discrete
Chess with clock	Yes	Yes	No	Semi	Yes
Chess without clock	Yes	Yes	No	Yes	Yes
Poker	No	No	No	Yes	Yes
Backgammon	Yes	No	No	Yes	Yes
Taxi driving	No	No	No	No	No
Medical diagnosis	No	No	No	No	No
Object recognition	Yes	Yes	Yes	Semi	No
Interactive english teacher	No	No	No	No	Yes

Figure 1.5: A mapping from environments and tasks to characteristics.

1.6 Multi-agent Systems

In particular tasks, there are multiple agents which may be working together to solve a problem, or they may be competing to get the best out of the situation for themselves. In the case of multiple agents interacting with each other and the environment, we speak of a **Multi-agent System (MAS)**. In principle the whole MAS could be modelled as one super-agent which selects actions for all individual agents. However, thinking about a MAS as a decentralised architecture has some advantages:

- Robustness. If the super-agent would stop working, nothing can be done anymore, whereas if a single agent of a big group of agents stops to work, the system can still continue to solve most tasks.
- Speed. In case of multiple agents, each agent could easily run on its own computer (distributed computing), making the whole system much faster than using a single computer.
- Simplicity to extend or modify the system. It is much easier to add a new agent running its own policy than to change one big program of the super-agent.
- Information hiding. If some companies have secret information, they do not want other agents to access that information. Therefore this information should only be known to a single agent. If everything runs on a super-agent the privacy rules are much harder to guarantee.

1.6.1 Model of a multi-agent system

If we are dealing with a MAS, we can still model the individual agents with the same formal methods as with single agents, so with inputs, actions, internal state, policy, reward function, and belief update function. In many cases, however, there will also be communication between the agents. In that case the agents possess communication signals (usually some language) and they map inputs and internal states to communication signals which they can send to

individual agents or broadcast to all of them. Communication is important if the agents have to cooperate. Coordination of agents is important to optimize a MAS, since otherwise they might all start to do the same job and go to the same places etc. It is clearly more efficient if the agents can discuss among themselves what role they will play in solving a task. Furthermore there may also be management agents which give roles and tasks to individual agents etc. A current challenging research field is to study self-adaptive structures or architectures of multi-agent organisations.

1.7 Complex Adaptive Systems

Some systems consisting of multiple interacting entities are called complex adaptive systems. The difference between complex adaptive systems and MASs is that in complex adaptive systems, the individual entities do not have a goal, they are just part of the overall system. Basically, these entities are smaller than a complete agent (think about the difference between your body-cells and you as a complete organism). Therefore complex adaptive systems also do not have to be able to control some process or solve some task, they are more important for simulating processes. We do not consider such complex adaptive systems as being rational, although they may still adapt themselves and can be very complex. In complex adaptive systems, simple rules can create complex behavior if multiple simple entities interact. We then often say that the overall system behavior **emerges** from the interaction between the entities. Examples of processes which we can model with complex adaptive systems are:

- Traffic consisting of many vehicles or other users of infrastructures
- Forest fires consisting of trees, grass, etc. which propagate the fire
- Infection diseases consisting of viruses and virus-carriers
- Magnetism consisting of elementary particles which can be positively or negatively charged
- Ecological systems which consist of many organisms which can eat each other and reproduce
- Economical markets which consist of many stocks and investors

In some cases of the above processes, we might also use a MAS to model them and try to optimize the process. This is especially clear in traffic or economical markets.

1.7.1 Predator-Prey systems

A simple example of a system consisting of multiple entities is a predator-prey system. The predator looks for food (prey) to eat and produces offspring. The prey also looks for food, reproduces itself, and tries to circumvent being eaten by predators. The interesting phenomenon is that the population of prey and predators depend on each other. If there are many predators, the population of prey will decrease since many of them will be eaten. But if there are few prey, the population of predators will decrease since there will not be enough food for all of them. If there are then few predators left, the population of prey will increase again, leading to repetitive dynamics.

Lotka-Volterra Equations. Lotka and Volterra captured the predator-prey system with a couple of equations. We will call the size of the prey-population x and the size of the predator-population y. Now the environmental state $S(t) = (x(t), y(t))$. The state will change according to the following two rules:

- $x(t+1) = x(t) + Ax(t) - Bx(t)y(t)$
- $y(t+1) = y(t) - Cy(t) + Dx(t)y(t)$

When we choose starting population sizes: $S(0) = (x(0), y(0))$ and we take some parameter values for A, B, C, D we get a dynamical system which behaves for example as seen in Figure 1.6.

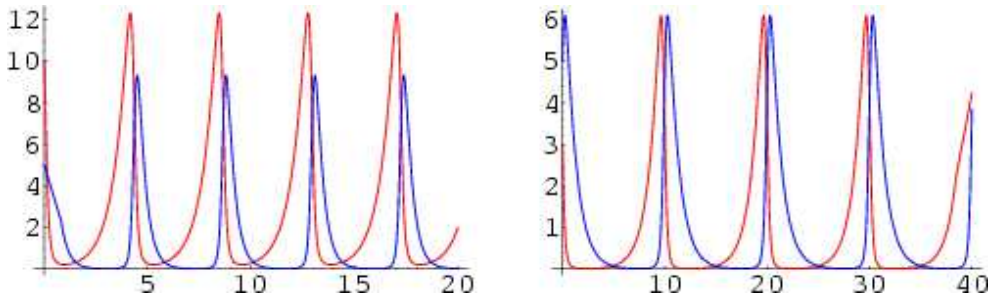

Figure 1.6: The predator-prey dynamics using Lotka-Volterra equations. Note that the predator population y will grow if there is a lot of prey and the prey population will decrease if there are many predators.

1.7.2 State dynamics

We have seen that the state of the environment shows a particular kind of dynamics. We can distinguish between three kinds of dynamics: dynamics to a **Stable point**, dynamics leading to a **periodic cycle**, and **chaotic** dynamics. When the state enters a stable point, it will always stay there, this means that the dynamics basically ends and $S(t+1) = S(t)$ for all $t \geq n$ where n is some time-step where the process enters the stable point. We can compute what the stable point of the dynamics of the Lotka Volterra equations will be depending on the parameters A, B, C, D. Whether the process will enter the stable point may also depend on the initial state. The following should hold for a stable point for the Lotka-Volterra process:

$$(x(t+1), y(t+1)) = (x(t), y(t))$$

Then we can find a stable point $S(*) = (x(*), y(*))$ as follows:

$$\begin{aligned} x(*) &= x(*) + Ax(*) - Bx(*)y(*) && (1.2)\\ 0 &= A - By(*) && (1.3)\\ y(*) &= \frac{A}{B} && (1.4) \end{aligned}$$

$$y(*) = y(*) - Cy(*) + Dx(*)y(*) \quad (1.5)$$

$$0 = -C + Dx(*) \tag{1.6}$$

$$x(*) = \frac{C}{D} \tag{1.7}$$

Periodic Cycle. For a periodic cycle, after some initial transient process, the state-sequence should always repeat itself after some period of fixed length. We have already seen two processes which lead to a periodic cycle, the heater and the Lotka-Volterra equations. Formally for a periodic cycle the following should hold:

$$\begin{aligned} S(t) &= S(t+n) \\ S(t+1) &= S(t+n+1) \\ &\cdot \quad \cdot \quad \cdot \\ S(t+n-1) &= S(t+2n-1) \end{aligned}$$

Here we say that the length of the periodic cycle is n. Note that a stable point is equivalent to a periodic cycle of length 1. Sometimes a process slowly converges to a cyclic behavior. We then say that the final attractor is a limit cycle.

Chaotic dynamics. In case the process does not lead to a stable point or to a periodic cycle (also called a stable limit cycle), the process might be called chaotic although there are some additional conditions for a true definition of chaos explained below. In chaotic dynamics it is very hard to predict what will happen after a long time, although according to the above definition alone it may be simple in some cases, e.g. the equation $S(t+1) = S(t)+1$ would according to the above definition also lead to chaotic dynamics. This is of course very strange, since we always think about chaotic processes as being unpredictable. Therefore we have to include the condition that the process is non-linear and sensitive to initial conditions. This means that when we start with two initial states $S_1(0)$ and $S_2(0)$ which may be very close to each other, that the difference between the trajectories will increase (exponentially) after iterating the process over time. In the case of the equation $S(t+1) = S(t)+1$ the difference between two starting states will not grow but remain the same and the system is clearly linear. But there are processes which are non-linear for which the difference between the state trajectories grows which are still predictable such as $S(t+1) = S(t) \times S(t)$ where $S(0) \geq 1$. Therefore even this requirement may not be strict enough, and to eliminate such trivial cases we have to add the condition that the state trajectory does not go to infinity, but remains bounded in some subspace. This bounded subspace is called a chaotic attractor, and although the state trajectory will remain in the attractor, it is unpredictable where it will be if we do not know the precise initial state and model of the chaotic system. All we can do is to compute a probability function over this subspace to guess in which area the process will be at some time-step.

The requirement that the difference between two initial states will grow makes the prediction problem much harder, since if our measured initial state has some small error ϵ then after some time, the error will have grown drastically so that our prediction of the state will not be valid or useful anymore. Since measuring a state and the change of the state for a complex non-linear system at the same time is impossible (for change we need to look at the difference between two states), we can never have a precise measurement of the current state (where the state includes position and velocity or change). Therefore, when the process is chaotic, it cannot be predicted over time.

Another interesting thought is that chaos is not really possible on a computer, since there are a fixed number of states on the computer. Therefore, since a chaotic system always uses a

deterministic transition function, we will always come back some time to the same state and then go to the next state etc. leading to some periodic cycle of very large period. It is also true that it is often hard to distinguish between chaotic dynamics and a periodic cycle, since the period may be so large that the process appears to be chaotic, but in reality has a very large period which did not appear in the generated state trajectory. Finally we should note that there is a big difference between non-determinism (randomness) or a chaotic process. A chaotic system is deterministic, but may appear random to an observer. On the other hand in non-determinism the process will never follow exactly the same state trajectory, so one might think such processes are chaotic. However, in a chaotic system we could in principle predict future states if the current state is exactly known. The impossibility to predict future states comes from the impossibility to know exactly the current state. On the other hand, in a non-deterministic system, even if we would know the exact initial state, prediction of a trajectory would be impossible since there would be many possible future trajectories. If we examine random-number generators, they are in reality pseudo-random number generators which provide us with seemingly random numbers, but basically it draws the random numbers from a huge periodic cycle of fixed length. Real randomness probably exists in nature, although it is extremely difficult to find out whether it is not deterministic chaos which makes nature to appear random.

1.8 Outline of this Syllabus

This syllabus describes a wide variety of adaptive systems, ranging from artificial life models such as cellular automata to machine learning methods such as artificial neural networks. Since the topic of adaptive systems is so broad, there may not always be an evident connection between the different topics. For example in machine learning, knowledge may be learned from examples. The interaction with the environment may not be very clear in such cases, since the knowledge representation is changing according to the learning dynamics generated by the interaction between the learning algorithm and the examples. Therefore the concept of environment should be considered also very broad ranging from the system itself or examples to a real world environment. In this syllabus the following topics will be covered:

- Cellular Automata which are useful as models for complex adaptive systems and studying artificial life.
- Biological adaptive systems in which systems inspired on swarm (e.g. ants) intelligence are used to solve complex problems
- Evolutionary computation in which a model of evolutionary processes is used to solve complex optimisation problems
- Robotics, where physical robots interact with an environment to solve some specific task
- Machine learning, in which different algorithms such as decision trees, Bayesian learning, neural networks, and self-organising maps are studied in their way of learning knowledge from examples. This knowledge may then be used to solve classification tasks such as mapping mushroom-features to the concept whether they are edible or poisonous.

- Reinforcement learning, which is a part of machine learning, but where the focus is more on an agent which can learn to behave by interacting with some specific environment.

Chapter 2

Artificial Life

Artificial life researchers study computation models of life-like and emergent processes in which complex dynamics or patterns arise from the interaction between many simple entities. Artificial Life is a broad interdisciplinary field where research runs from biology, chemistry, physics to computer science and engineering. The first artificial life workshop was held in Santa Fe in 1987 and after this the interest in this field grew tremendously. One of the most ambitious goals of artificial life is to study the principles of life itself. To study the properties of life there are basically two roads; to study carbon life forms and their development (mainly done in biochemistry) and to examine life forms and their properties using a computer. What both fields have in common is that life emerges from building blocks which cannot be called alive on their own. So the interaction between the elements makes the whole system appear to be alive. Since the interactions are usually not well understood, the study to artificial life is usually holistic in nature, which means that we look at the whole system without being able to make clear separations in smaller modules. Still today many scientists think that life evolved from chemicals in the primordial soup (containing a large number of carbon compounds), although some scientists believe that life may have come from space on a comet. Some assert that all life in the universe must be based on the chemistry of carbon compounds, which is also referred to as "carbon chauvinism".

Thus, artificial life constructs models and simulates them to study living entities or other complex systems in computer systems. Some research questions which it tries to answer are:

- Biology: How do living organisms interact in biological processes such as finding/eating food, survival strategies, reproduction?
- Biochemistry: How can living entities emerge from the interaction of non-living chemical substrates?
- Sociology: How do agents interact in artificial societies if they have common or competing goals?
- Economy: How do rational entities behave and interact in economical environments such as in stock-markets, e-commerce, auctions, etc.?
- Physics: How do physical particles interact in a particular space?
- Artificial Art: How can we use artificial life to construct computer art?

One important goal of artificial life is to understand the source and functionality of life. One particular way of doing that is to make computer programs which simulate organisms using some encoding (might be similar to DNA encoding, but the encoding can range to computer programs resembling Turing machines). The development of artificial creatures which can be called alive also requires us to have a good definition of alive. For this we cite: http://www.wordiq.com/definition/Life

> In biology a conventional definition of an entity that is considered alive has to exhibit all the following phenomena at least once during its existence:
>
> - Growth
> - Metabolism; consuming, transforming and storing energy/mass growing by absorbing and reorganizing mass; excreting waste
> - Motion, either moving itself, or having internal motion
> - Reproduction; the ability to create entities which are similar to itself
> - Response to stimuli; the ability to measure properties of its surrounding environment, and act upon certain conditions
>
> A problem with this definition is that one can easily find counterexamples and examples that require further elaboration, e.g. according to the above definition fire would be alive, male mules are not alive as they are sterile and cannot reproduce, viruses are not alive as they do not grow. One could restrict the definition to say that living organisms found in biology should consist of at least one cell and require both energy and matter to continue living, but these restrictions do not help us to understand artificial life. Finally one could change the definition of reproduction to say that organisms such as mules and ants are still alive by applying the definition to the level of entire species or of individual genes.

As we can see; there are still many possible definitions and just as with the concept intelligence, we may not easily get one unique definition of "alive".

2.1 Genetic Algorithms and Artificial Life

One well-known algorithm in artificial intelligence that is based on evolutionary theory is the genetic algorithm (GA). Darwin speculated (without knowing anything about the existence of genes) that evolution works by recombination of material of parents which pass the selective pressure of the environment. If there are many individuals only some can remain alive and reproduce, this selection is very important for nature since it allows the best apt individuals to reproduce (survival of the fittest). Once parents are selected they are allowed to create offspring and this offspring is slightly mutated so that the offspring will not contain exactly the same genetic material as the parents. Genetic algorithms can be used for combinatorial optimization problems, function optimization, robot control, and the study of artificial life societies. We will not go into detail into genetic algorithms here, since they will be described thoroughly in a separate chapter. Shortly, genetic algorithms are able to mimic the concept of reproduction. Say some artificial organism is stored in some representation, such as a bitstring (a string of 0's and 1's). Then we can take two parents, cutoff their string in two parts and glue these parts together to create a new offspring, which could possibly be better

in the task than its parents. Since parents which are allowed to reproduce are selected on their fitness in the environment, they are likely to possess good blocks of genetic material which may then be propagated to the child (offspring). In combination with artificial life, genetic algorithms allow us to study a wide variety of topics, including:

- Robots which interact with an environment to solve some task
- Competitive evolutionary models such as arm-races studied by Karl Sims. In the arm-races experiment different morphologies and behaviors were evolved in 3D structures where two organisms had to compete against each other by harming the opponent. The winning individual passed the test and was able to reproduce leading to a wide variety of improving morphologies and behaviors.
- Models of social systems such as the study of emerging societies of individuals which work together
- Economical models such as the development of buying and selling strategies
- Population genetics models where one examines which groups of genes remain in the population
- The study of the interaction between learning and evolution

2.1.1 Interaction between evolution and learning

In evolutionary theory, sociology, and psychology one often considers the difference between nature and nurture. Nature is what a newborn organism possesses at its birth. E.g. Chomsky claims that a lot of knowledge for learning a language is already born in the brain of a child when it is born. Nurture is the knowledge, skills, and behaviors which an organism develops through its adaption and learning process while interacting with an environment. The nature/nurture dilemma is often to say whether something was born inside an organism or whether it developed due to the interaction with the environment. Examples of this are whether criminals are born like a criminal or whether they become one due to their education and life. Another example is whether homo-sexuality or intelligence is inborn and stored in the genes or not. Often it is better to say that nature gives a bias towards some behavior or the other, and nurture causes some behaviors to be expressed. E.g. if someone has genes which may be similar to other people having schizophrenia, it is not necessary that such a person would develop the disease, this depends a lot on circumstances but if such a person would suffer from a lot of stress, the genes may be expressed with a much bigger probability.

In artificial life simulations a number of machine learning algorithms can be used which can learn from the interaction with the world. Examples of this are reinforcement learning and neural networks. Although these topics will be discussed in separate chapters, they could also be used together with genetic algorithms in an environment consisting of many entities that interact and evolve. Now if we want to study the interaction between evolution and learning we see that evolution is very slow and takes place over generations of individuals, whereas learning is very fast and takes place within an individual (agent). The combination of these 2 leads to two possible effects:

- **Baldwin effect.** Here an individual learns during its interaction with the environment. This learning may increase the fitness of the individual so that individuals which are

good in learning may receive higher fitness values (are better able to act in the environment) than slow learning individuals. Therefore individuals which are good in learning may reproduce with a higher probability leading to offspring which are potentially also very good in learning. Thus, although the skill of learning is propagated to offspring, learned knowledge is not immediately propagated to the offspring.

- **Lamarckian learning.** Here an individual learns during its life and when it gets offspring it also propagates its learned knowledge to its children which then do not have to learn this knowledge anymore.

Lamarckian learning is biologically not very realistic, but in computer programs it would be easily feasible. E.g. suppose that a group of robots all go to learn to use a language, then if they meet they can create offspring which immediately possess multiple languages. In this way the evolutionary process could become much more efficient. Although Lamarckian learning has not been realistic from a biological point of view until today, research in genetic engineering has currently invented methods to change the DNA of an organism which can then be transmitted to its offspring.

2.2 Cellular Automata

Cellular automata are often used by researchers working in artificial life. The inventor of cellular automata (CA) is John von Neumann who also devised the modern computer and played an important role in (economical) game theory. Cellular automata are decentralised spatial systems with a large number of simple, identical components which are locally connected. The interesting thing of cellular automata is that they are very suited for visualizing processes, and that although they consist of simple components and some simple rules, they can show very complex behaviors. CA are used in a number of fields for biological, social, and physical processes such as:

- Fluid dynamics
- Galaxy formation
- Earthquakes
- Biological pattern formation
- Forest fires
- Traffic models
- Emergent cooperative and collective behavior

2.2.1 Formal description of CA

A cellular automaton consists of two components:

- **The cellular space.** The cellular space consists of a lattice of N identical cells. Usually all cells have the same local connectivity to other cells. Let $\sum$ be the set of possible states for a single cell. Then $k = |\sum|$ is the number of possible states per cell. A cell with index i on time-step t is in state s_i^t. The state s_i^t together with the states of the cells with which i is connected is called the neigborhood n_i^t of cell i.

- **The transition rule.** The transition rule $r(n_i^t)$ gives an update for cell i to its next state s_i^{t+1} as a function of its neigborhood. Usually all cells are synchronously (at the same time) updated. The rule is often implemented as a lookup-table.

2.2.2 Example CA

The following gives an example of a CA consisting of a 1-dimensional lattice of 11 states with periodic boundary conditions. The periodic boundary conditions mean that the most left state has the most right state as its left neighbour and vice versa. Since the neigborhood of a cell consists of itself, the state of the cell to the left and to the right, the size of the neigborhood is 3. Therefore, since the number of possible states of a single cell is only 2 (1 or 0), the transition rule consists of $2^3 = 8$ components; for each neigborhood there is one possible successor state for each cell. Note that in this example there are $2^{11} = 2048$ possible complete state configurations for the CA.

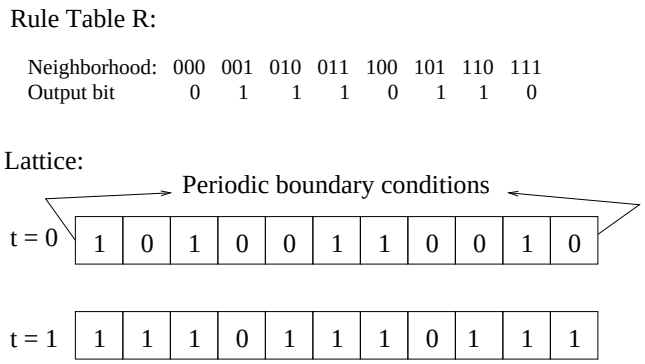

Figure 2.1: A cellular automaton using a 1-dimensional lattice, a neigborhood size of 3, and 2 possible states (0 or 1) per cell. The figure shows the CA configuration at time $t = 1$ computed using the transition rule on the CA configuration at time $t = 0$.

2.2.3 Dynamics of the CA

The CA given in the previous subsection only uses 1 dimension, a neigborhood size of only 3, and 2 possible states per cell. Therefore, it is one of the simplest CA. But even this CA can show complex behavior if we iterate it over time and show the dynamics in the space-time dimensions, see Figure 2.2.

It will not be a surprise that cellular automata with more complex transition rules and a larger number of possible states can even shown much more complex behavior. In principle there are other possible iterative networks or automata networks, cellular automata are just one kind of automata of this family.

2.2.4 Processes in CA

In Chapter one we have already seen that when we have bounded spaces, we can divide a process resulting in a pattern into three different classes; stable, periodic, and chaotic. Since the cellular configuration state space of a CA is bounded, we can divide patterns created by

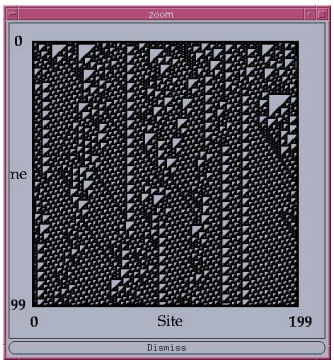

Figure 2.2: The sequence of cellular patterns of the CA given in Figure 2.1 generated by iterating it over 100 time steps.

a CA into these three groups. Note however that the set of possible complete states of a CA is not only bounded, but also finite. The three possible resulting patterns of a CA are:

- A stable state (or point), after entering the stable state, the process remains in the same state and change stops.
- A cyclic pattern. The CA traverses through a repeating pattern of some periodic length. If there are multiple sub-patterns each with their own periodic length, the complete pattern will be periodic but with a larger length (e.g. if two sub-patterns which do not interact in the CA have periodic lengths of 2 and 3, the complete pattern will have periodic length 6).
- Chaotic behavior. The CA always goes to new, unseen patterns. Since the CA is deterministic, chaotic behavior would be possible. However, since the number of possible states on a computer is finite (although it is often huge), there will after finite time always be a state which has been seen before after which the process repeats the same cycle of configurations. Therefore real chaotic behavior in a CA is not possible, only a periodic cycle of very large length will be possible in a finite CA.

It is important to understand that an initial configuration may lead to a sequence of patterns which are all different, after which it may enter a stable state or a periodic cycle. The time until the CA enters a stable state or periodic cycle is called the transient period. Some researchers also like to include structured behavior with the above mentioned three types of behavior. In structured behavior, the behavior seems very structured, but there is no repetitive sequence (at least not for a long time).

The dynamics of CA can be influenced by the transition rules. Some transition rules can lead to very simple behavior, whereas others lead to very complex behavior. Some people find it a sport to make a transition rule which has the longest possible periodic length.

2.2.5 Examples of cyclic processes

A stable state is easy to make, e.g. it can consist of only 1's. Then if we make transition rules which always output a 1, we get the resulting stable state from any possible initial

configuration after one time step. Periodic cycles can be made in many possible ways. Here we show a simple example. Suppose we have a 2-dimensional lattice. The transition rule is: if 2 neighbours (out of 4) are active, then the cell is activated (becomes 1 or black). Otherwise the cell is not activated (becomes 0 or white). Figure 2.3 shows a lattice without boundary conditions (basically we show a small part of the lattice which is everywhere else empty so that we still have identical connectivity for all states), resulting in a periodic cycle of length 2.

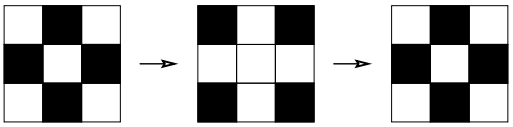

Figure 2.3: A cellular automaton configuration with a repeating pattern (the periodic length is 2).

Problem. Given a 2-dimensional lattice with transition rule: if one neighbour is active and the cell was inactive, then the cell becomes active. Else if the cell was active at the previous time-step keep the cell active in the next time-step. Otherwise the cell remains inactive. Now evolve the CA in Figure 2.4.

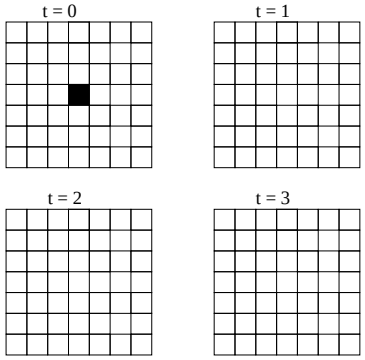

Figure 2.4: The lattice of the CA for the problem. Try to evolve the CA over time with the above given transition rule.

2.2.6 Elimination of basis patterns

When one evolves a CA, there are often some regularities involved, and other parts which are completely unpredictable. Therefore some researchers have tried to use methods for eliminating the basis of the evolutionary transitions in a CA. This basis can consist of walls, singularities, etc. and can then be eliminated from the process.

The importance of eliminating the basis patterns is to get more inside in possible chaotic or turbulent processes. For example take the process from Figure 2.5. If we remove the regularities from this process, we get the process shown in Figure 2.6. We can see that most of the seemingly complex process is removed, but some **embedded particles** move about

in a seemingly random way. It turns out that when these embedded particles hit each other, that they will be destroyed.

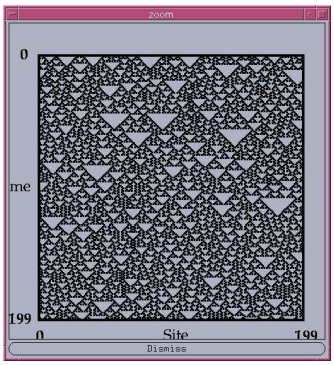

Figure 2.5: A CA process iterated over time.

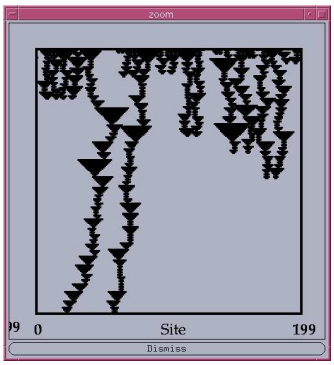

Figure 2.6: The process of Figure 2.5 with the regular basis patterns removed.

2.2.7 Research in CA

One important insight is that cellular automata are universal machines. That means that they can compute any computable function and are therefore just as powerful as Turing Machines. This also means that any algorithm which can be implemented on the usual sequential computer can in principle also be implemented in a CA.

Conway's game of life

The game of life was invented by the mathematician John Conway in 1970. He chose the rules carefully after trying many other possibilities, some of which caused the cells to die too fast and others which caused too many cells to be born. The game of life balances these tendencies, making it hard to tell whether a pattern will die out completely, form a stable population, or grow forever. Conways' game of life uses a 2-dimensional lattice with 8 neighbours for each cell. The transition rule(s) are:

- If a cell is not active (dead, black, or 1) and it has exactly 3 living neighbours, then the cell will become active (rule of birth)
- If a cell is active and it has 2 or 3 neighbours which are active, then the cell stays active (rule of survival)
- In all other cases the cell becomes not active (rule of death due to overcrowding or loneliness).

One of the interesting things about the game of life is that it has universal computing power, even with the three rules given above. This universal computing power relies on particular patterns known as **gliders**. Such gliders are living entities which cross the 2-D lattice and which can pass information so that it becomes possible to make logical AND, and NOT gates. For an example of the behavior of a glider look at Figure 2.7.

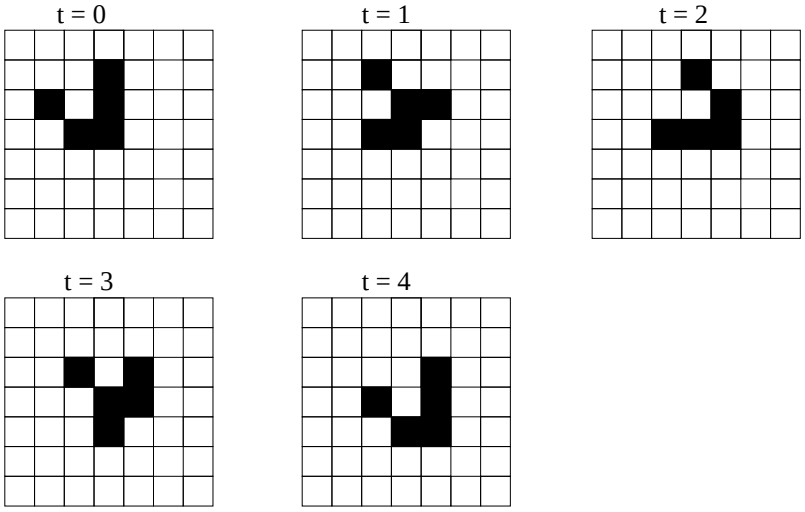

Figure 2.7: A glider moving one step diagonal after each 4 time-steps.

Another important object in the game of life is the use of a **Glider gun.** Glider guns can fire gliders and remain stable, which makes it possible to propagate information at some rate. By using multiple glider guns which shoot gliders, we can make interactions between different patterns which are propagated in the cellular space. An example of this is to have two gliders which collapse after which they will be destroyed. This would be useful to make a NOT gate. Making a CA using the game of life rules to compute arbitrary functions is very complicated, because it requires a very careful development of the initial configuration consisting of glider guns and other patterns, but in principle it would be possible.

Another interesting pattern in the game of life which shows very complex behavior is known as the R-pentomino which looks as shown in Figure 2.8. It is remarkable that such a simple pattern can create complex behavior including gliders and many other patterns.

Development of cellular automata

One goal of artificial life is to make artificial systems which can be called alive. For this reproduction seems necessary, and therefore research investigated whether this was possible

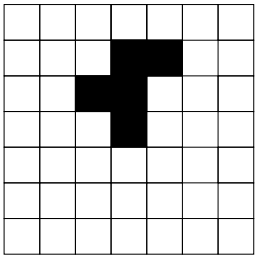

Figure 2.8: The pattern called R-pentomino which creates very complex behavior.

in cellular automata. In 1966, John Von Neumann constructed a cellular automaton which was able to reproduce itself, demonstrating one of the necessary abilities of living systems. Some other researchers examined whether cellular automata could be used for recognizing languages. In 1972, Smith constructed a CA which could recognize context-sensitive languages such as palindromes (palindromes are strings which are the same if you read them from left to right or from right to left). After that, Mitchell et. al (1994) used genetic algorithms to evolve the transition rules of CA. They tried this using the majority problem as a testbed. In the majority problem a bitstring is given of some size and each bit of the string can be on or off. Now the system should tell whether the majority of bits was on or off. The system could indicate this by making all bits on (off) if the majority was on (off) after a number of iterations. Although this problem can of course be simply solved by counting all bits, such a counter would require some form of register or additional memory which was not inside the cellular automaton. Thus, the question was whether the genetic algorithm could evolve transition rules which can solve the problem. The result was that the genetic algorithms found different solutions which are however not optimal for solving all initial problems (with any order of 1's and 0's). Some of the solutions used embedded particles. The reason that no optimal solution was evolved was due to the limited local connectivity which does not allow all bits to communicate to each other.

Other cellular automata

Cellular automata can also be simply and efficiently used for simulating particular processes such as:

- Modelling Traffic. Here a cell is active if there is a car and it is inactive if there is no car. The rules are simple to make too; if the predessor cell is empty, move to that cell, otherwise stop. The CA can be made more complicated by adding in each cell occupied by a car some internal state which models the destination address of the car. Also different speeds can be taken into account.

- Modelling Epidemics. Here a cell can be a sick, healthy, or immune person.

- Modelling Forest Fires. A cell can be a tree on fire, water, a tree without being on fire, grass, sand, etc. It is also possible to include external parameters such as wind-strength and wind-direction, humidity etc. to influence the behavior of the model.

Power laws

There is a lot of research using CA for examining chaotic processes as for example studied in sandpile models. In cellular automata sandpile models a granular material in a gravitational field is used (the model can be two or three dimensional). There are two kinds of cells; immovable ground cells and movable sand grains. Grains fall from a source at the top of the window and proceed down to the ground. Grains pile up and redistribute themselves according to the cellular automata rules (e.g. if two cells on top of each other possess grain, and a neighboring cell does not, then the top grain element will make a transition to the empty neighboring cell). One interesting thing of CA implementations of such physical models is that there will sometimes be long shifts of grain during the redistribution. Such a shift is often called an avalanche. Now the interesting thing is that large avalanches will be much less probable than smaller ones, and that the probability distribution law respects a power law (or Zipf's rule or Pareto distribution). E.g. if we take English words according to their number of occurrences and we rank all the words according to their usage (so rank 1 means the word is used most often), then Zipf's law states that the size y of occurrence of an event (in this example the occurrence of a word) is inversely proportional to its rank r according to:

$$y = ar^{-b}$$

Where a is some constant and the exponential factor b is close to 1. Such a power law has been demonstrated in many research fields, such as in social studies where the number of users of web-pages are counted to examine Website popularity. There are few web-pages with a large number of users, and many web-pages with few users, and the distribution follows Zipf's law. Pareto looked at income and found that there are few millionaires whereas there are many people with a modest income. Also for earthquakes, there are few very heavy earthquakes and many smaller ones, etc. To show whether some data provides evidence for a power law, it can be hard to work with very large values appearing in the data. In that case we can make a log-log plot by taking the logarithm on both sides (note that they should be positive) so that we get:

$$\begin{aligned} \log y &= \log ar^{-b} \\ \log y &= \log a + \log r^{-b} \\ \log y &= \log a - b \log r \end{aligned} \tag{2.1}$$

Thus in a log-lot plot of the data, the resulting function relating two variables should be a line (with negative slope b).

2.3 Ecological Models

In biology and ecology, simulation models often make use of cellular automata due to their insightfulness and easy implementation while still providing interesting and complex behaviors. Ecological models can be used to study social phenomena, immunology and epidemics, population dynamics of different species etc. An artificial ecosystem consists of a number of individuals (agents) which:

- Occupy a position in the environment
- Interact with the environment and with other agents

- Possess some internal state such as amount of energy or money

By examining the evolutionary process in an ecosystem it is possible to research the creation and continuity of processes such as:

- Cooperation: E.g., trading behavior between individuals
- Competition: E.g., fighting behavior between individuals
- Imitation: E.g., an agent learns what he should do by looking at and imitating other agents
- Parasitic behavior: An individual profits from another individual whereas the other individual is harmed by this. Parasitic behavior can be found in many places in nature, a good example of this are viruses.
- Communities: If a large group of individuals are put together they might form communities for the benefit of all. An example of this is fish-schools which can better protect the fish from predators (especially the fish which swim in the middle). Another advantage of communities is that individuals can cooperate and specialise on their own task.

2.3.1 Strategic Bugs

Bedau and Packard developed the artificial life model called Strategic bugs (1992). This model of an ecosystem uses individuals which try to find food and reproduce. The model consists of:

- An environment modelled as a 2-dimensional lattice.
- A cell in the environment can be occupied by food or by a bug or is empty
- Food will grow automatically in the environment; food is added in a cell with some probability if there was no food or bug there
- Bugs survive by finding food
- Bugs use energy to move and die if they do not have any energy anymore
- Bugs can clone themselves or reproduce with another bug if they have sufficient energy.

The behavior of a bug evolves from the interaction of the policy of the bug and the environment. The bug's policy uses a lookup table to map environmental inputs to actions. An example rule is: if there are more than 5 food units in the east, then make a step to the east.

Bedau and Packard tried to come up with a measure for the evolutionary dynamics. If such an ecosystem is simulated and new individuals will be generated all the time, then the question is "What is really new and which individual traits are evolved in the system?" For this they examined the evolutionary activity which looks at the genetic changes in the chromosome strings. The experiments showed that there were waves of evolutionary activity, new genetic material was often found after some time and then stayed in the population for some period. Thus it was seen that new genetic material and therefore behavior was found and exploited during the evolutionary process.

2.4 Artificial Market Models

Financial markets such as stock markets are difficult to predict. Some might think it is completely random behavior, but the investors involved do not seem to make random, but on the contrary, rational decisions. Thus it seems more to be a chaotic process emerging from the large number of investors and unforeseen circumstances.

One important question is to examine under what conditions predictions about the dynamics of financial markets will be possible. To study this question we first have to look at the **efficient market hypothesis (EMH)**. In an information efficient market all price fluctuations are unpredictable if all necessary investment information is taken into account by the investors. The information is taken into account if the expectancies, intentions, and (secret) information of the market participants is incorporated in the prices. From this follows that when a market is more efficient, that the price fluctuations which are generated by the market are more random (and therefore unpredictable). Basically this is caused by the fact that if there would be only a small information advantage by some investors, that the actions of these investors will immediately correct the prices, so that further gain will become impossible.

2.4.1 Are real markets predictable?

Some people tend to make a lot of gain from stock markets. One important case is that of an analyst which has such an importance that (s)he is considered a guru for predicting which stocks will rise and fall. If the guru tells everyone that stock X will increase a lot, then there will be many people buying that stock. The effect is that of a self-fulfilling prophecy; the stock price will increase since the prophet announced it and many people believe it and will buy that stock. Only the buyers which were the last in buying that stock will loose money, the investors which are quickest will gain money and sell them immediately after the price has increased sufficiently. There are other cases and reasons to believe that stock markets can be predictable. One reason is that investors trade-off expected risk and expected gain. This means that a risk-aversive (in contrary to a risk-seeking) investor will sell stocks with a high risk but also with an expected gain. The distribution between risk-aversive and risk-seeking individuals will then cause different price fluctuations, which are therefore not completely random. In fact a number of studies have indicated that price fluctuations are not completely random.

When we examine the efficient market hypotheses, then it requires rational and completely informed investors. However these assumptions are not realistic. Investors are not completely rational and sometimes hard to predict. Furthermore, information is often difficult to interpret, technologies and companies change, and there are costs associated with transactions and information gathering.

One seemingly efficient method for trading stocks is to examine the relative competitive advantage between different markets. When one compares some market to other markets, one can see that one market (such as a market in obligations) was more promising during the previous period, so that it will be likely that more investors will step to that relatively more advantageous market which leads to more profit on that market. Comparing markets (e.g. between countries, or kind of markets — e.g. obligations versus stocks) can therefore be a good option.

2.4.2 Models of financial theories

Already for a long time there have been people trying to come up with financial theories, since if it would work one could get a lot of money out of it. It should be said, however, that if you would ever find a theory which works, that you should not tell it to other people. The reason is that your advantage will be lost in using this theory if everyone knows it. People could even trade in such a way that you will loose money with your once so well working theory. Therefore we can only show general approaches that have been invented to come up with models to predict the price fluctuations:

- Psychological models. Here the model tries to analyse the risk-taking behavior of investors and examines how human-attitudes to the market influences the stock prices.
- Learning models. Here data about the stock prices of the past is used to train a model to predict its development in the future.
- Agent models. Here investors are modelled as agents which use particular strategies. By letting the modelled agents interact the complex dynamic of stock markets can be simulated.
- Evolutionary algorithms for developing strategies. Here the evolution of strategies of investors is mimicked. Competitive strategies could be used to create other strategies. Finally a strategy which was observed to gain most money in the past could be used to trade in the future.

2.5 Artificial Art and Fractals

Iterating a simple function can create very complex, artistic, patterns. This was shown by Bernoit Mandelbrot who discovered the Mandelbrot set, which is a fractal. A fractal is a pattern which is self-similar to different scales, so if we look at a zoomed out picture of some details of the fractal we can recognize features which were also shown in the bigger pattern. It should be said that a fractal can be very complex and not all small scale components look similar to the whole pattern. So how can we get the Mandelbrot set? First of all consider the function:

$$x_{k+1} = x_k^2$$

If we look at the starting values for x_k for which the iteration converges to a single point, we can see that these are the values $-1 < x_0 < 1$, and the final point will be $x_\infty = 0$. If $x_0 < -1$ or $x_0 > 1$ then the value after many iterations goes to infinity. If x_0 is -1 or 1 then the point will stay in 1, but this point is unstable, since small perturbations (changes of x_k) will let the value go to 0 or ∞. In principle the values for which the iteration stays bounded is called the **Julia set**, although more interesting Julia sets are associated to Mandelbrot sets as we will see later. So for the function $f(x) = x^2$, the Julia set would be the region between -1 and 1.

In the space of real numbers, not so many interesting things can happen. But now let's consider the use of complex numbers. Complex numbers consist of a real and an imaginary part, so we write them as: $x = ai + b$, where i is defined as $i = \sqrt{-1}$. We can add, subtract, multiply and divide complex numbers just as we can with real numbers. For example if we take $x = 3i$, then $x^2 = -9$. Complex numbers are used in many sciences such as in quantum mechanics and electric engineering, but we will not go into details about them here.

Now consider the functions of the type:

$$x_{k+1} = x_k^2 + C$$

The question is: if we start with $x_0 = 0$, for which complex numbers C will the iteration of this function not become infinite? This set of complex numbers for which the iterations will stay bounded is called the Mandelbrot set, and it is displayed in Figure 2.9. We can see its complex shape in the complex plane (the real part is depicted on the x-axis and the imaginary part of the points belonging to the set are shown on the y-axis). The points in black belong to the Mandelbrot set, and the others do not. This is an example of a fractal, a self-similar structure. The word fractal was also invented by Mandelbrot.

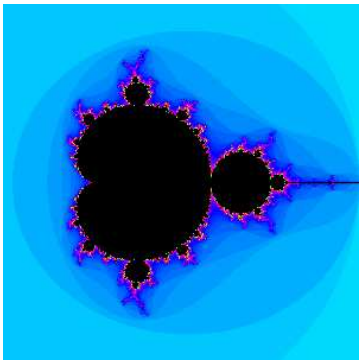

Figure 2.9: The Mandelbrot fractal

Now look what happens if we zoom-in in the picture. The zoomed in figure of the lower part of Figure 2.9 is shown in Figure 2.10. Note that this pattern is very similar to the original Mandelbrot set, and we already see that there are much more self-similar structures to be found in the picture.

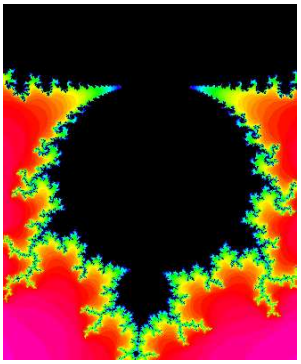

Figure 2.10: A zoomed in pattern of the Mandelbrot fractal

Now, consider again the iterated function

$$x_{k+1} = x_k^2 + C$$

But, now we have chosen a value for C which is an element of the Mandelbrot set. Then another question we can ask is; which initial values x_0 in the complex plane cause the iteration to remain bounded? This set which belongs to a particular value of C is called the Julia set for C. An example pattern from the Julia set is shown in Figure 2.11.

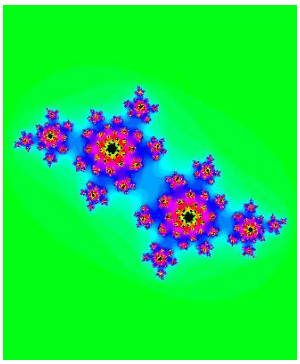

Figure 2.11: An example pattern from the Julia set

Computer artists like to use fractals, since although the equations are simple, as long as they are non-linear (linear maps cannot produce interesting patterns like fractals) they can produce a large variety of complex patterns, and zooming in in the pictures creates many other patterns. This is just another example of using simple rules to create very complex patterns.

2.6 Conclusion

Artificial life is useful for simulating many biological, physical, sociological, and economical processes. One goal of artificial life is to understand the principles underlying living entities and the emergence of life forms. Artificial life can be combined with genetic algorithms for optimizing individual behaviors by adapting them to the (changing) environment. If multiple individuals adapt themselves and also adapt the environment, the resulting dynamics can be very complex and unpredictable. Even with simple entities such as used in cellular automata, complex behavior can result from the interaction between simple components. Cellular automata are very useful for modelling and visualizing spatial processes such as forest fires and can be used to study the behavior of many different complex processes. One interesting thing is that cellular automata are just as powerful as Turing machines which means that any computable function can be implemented using a cellular automaton.

Another aspect in artificial life is the study of price-dynamics in financial markets. Although an efficient market would be completely unpredictable, in reality there are many reasons to believe that price-fluctuations are not completely random. Making models for predicting price changes is a challenging research topic, although found theories may never be published, since they would eliminate their usefulness if they are known by many investors.

Finally we have shown that using the complex plane, simple iterative functions can create complex patterns, called fractals. Examples of these are the Mandelbrot and Julia sets. Computer artists like to use fractals, because they look complex, but are easy to make. Fractals also play a role in chaotic systems as we will see in a later chapter.

Chapter 3

Evolutionary Computation

Inspired by the success of nature in evolving such complex creatures as human beings, researchers in artificial intelligence have developed algorithms which are based on evolution theory. The class of these algorithms are called evolutionary algorithms and consists among others of genetic algorithms, evolutionary strategies, and genetic programming. Genetic algorithms (GAs) are the most famous ones and they were invented by John Holland. Evolutionary algorithms are optimisation algorithms that are inspired on Darwin's evolution theory, known as natural selection or survival of the fittest and they were developed during the 1960's and 1970's. One of their strengths is that they can find very good solutions in very large search spaces, where exhaustive search (trying out all possible solutions) would cost much too much time. The principle of evolutionary algorithms is that solutions are evaluated after which the best solutions are allowed to reproduce most offspring (children). If the parent individuals form good solutions, they are likely to possess good building blocks of genetic material (the genetic material makes up the solution) that may be useful for creating new individuals. Genetic algorithms usually take two parent individuals and they recombine their genetic material to produce a child that inherits genetic material from both parents. If the child performs well on the evaluation test (evaluating an individual and measuring how well an individual performs is commonly done by the use of a fitness function), it will also be selected for reproduction and in this way the genetic material can again be propagated to new generations. Since the individuals themselves will usually die (they are often replaced by individuals of the next generation), Richard Dawkins came with the selfish gene hypothesis. This hypothesis says that basically the genes are alive and use the mortal individuals (e.g. us) as hosts so that they are able to propagate themselves further. Some genes may be found in many individuals, whereas other genes are only found in a small subset of individuals. In this way, the genes seem to compete for hosts, and genes which occupy well performing individuals are likely to be able to reproduce themselves. The other way around we can say that genes which occupy well performing individuals give advantages for the individual and therefore it is good if they are allowed to reproduce.

In this chapter we will look at evolutionary algorithms in general and focus on genetic algorithms, although most issues involved also play a role for other evolutionary algorithms. We first describe optimisation problems and then examine which steps should be pursued for constructing an evolutionary algorithm, and what kind of representations are useful for the algorithm for solving a particular problem. Finally we will examine some other evolutionary algorithms.

3.1 Solving Optimisation Problems

A lot of research in computer science and artificial intelligence has been devoted to solving optimisation problems. There are many different optimisation problems; e.g. one of them is shortest path-planning which requires the algorithm to compute the shortest path from a state to a particular goal state. Well known applications for such algorithms are planners used by cars (e.g. the Carin system) or for train-passengers. In principle shortest path problems are simple problems, and can be solved efficiently by algorithms such as Dijkstra's shortest path algorithm or the A* algorithm. These algorithms can compute the shortest path in a very short time for problems consisting of more than 100,000 cities (or nodes if we formalise the problem as a graph using nodes and weighted edges representing the distances of connections between nodes). On the other hand, there also exist combinatorial optimisation problems which are very hard to solve. One example is the traveling salesman problem (TSP). This problem requires that a salesman goes to N customers which live in different cities, so that the total tour he has to make from his starting city to single visits to all customers and back to his starting place should be minimal. This problem is known to be NP-complete and therefore unless $P = NP$ not solvable in polynomial time. For example if we use an exhaustive search algorithm which computes and evaluates all possible tours, then it has to examine about $N!$ tours, which increases exponentially with N. Thus for a problem with 50 cities, the exhaustive search algorithm would need to evaluate 50! solutions. Let's say that evaluating one solution costs 1 nanosecond (which is 10^{-9} second), then evaluating all possible solutions would cost about 9.6×10^{47} years, which is therefore much longer than the age of the universe. Clearly exhaustive search approaches cannot be used for solving such combinatorial optimisation problems and heuristic search algorithms have to be used which can find good solutions in a short time, although they do not always come up with the optimal solution. There is a number of different heuristic search algorithms such as Tabu search, simulated annealing, multiple restart local hill-climbing, ant colony algorithms, and genetic algorithms. Genetic algorithms differ from the others in the way that they keep a population of solutions and use recombination operators to form new solutions.

3.1.1 Formal description of an optimisation problem

Optimisation problems consist of two components; the representation space and the evaluation (or fitness) function. The representation space denotes all possible solutions. For example if we want to solve the TSP, the representation space consists of all possible tours which are encoded in some specific way. If we want to throw a spear at some target and can select the force and the angle to the ground, the representation space might consist of 2 continuous dimensions which take on all possible values for the force and angle. On the other hand, one could restrict this space by allowing only angles between 0 and 360 degrees and positive forces which are smaller than the maximum force one can use to throw the spear. Let's call the representation space S and a single solution $s \in S$.

The evaluation function (which in the context of evolutionary algorithms is usually called a fitness function) compares different solutions to each other. Although solutions could be compared on multiple criteria, let's assume for now that there is a single fitness function $f(.)$ which maps a solution s to a specific fitness value $f(s) \in \Re$. The goal is to find the solution s_{max} which has the maximal fitness:

$$f(s_{max}) \geq f(s) \; \forall \; s$$

It may happen that there are multiple different solutions with the same maximal fitness value. We may then require to find all of them, or only one (which is of course simpler).

So the goal is to search through the representation space for a solution which has the maximal possible fitness value given the fitness function $f(.)$. Since the representation space may consist of a huge number of possible solutions or may be continuous, the optimal solution may be very hard to find. Therefore, in practice algorithms are compared by their best found solutions within the same amount of computational time. Among these algorithms there could also be a human (expert) which tries to come up with a solution, but if the fitness function gets more complicated and the representation space becomes bigger, the advantage of computers in their ability to try out millions of solutions within a short period of time outcompetes the ability of any human in finding a good solution.

3.1.2 Finding a solution

Heuristic search algorithms usually start with one or more random solutions which are then evaluated. For example local hill-climbing starts with a random solution and then changes this solution slightly in some way. Then, this new solution is evaluated and if it is a better one than the previous one, it is kept and otherwise the previous one is kept. This simple process is repeated until the solution is good enough or time is expired. The local hill-climbing algorithm looks as follows:

- Generate initial solution s_0; t = 0
- Repeat until stop criterium holds:
- s_{new} = change(s_t)
- if $f(s_{new}) \geq f(s_t)$ then $s_{t+1} = s_{new}$
- else $s_{t+1} = s_t$.
- t = t +1

Using this algorithm and a random initial solution s_0, a sequence of solutions $s_0, s_1, \ldots, s_T$ is generated, where each later solution has a larger or equal fitness value compared to all preceding solutions. The most important function in this algorithm is the function change. By changing a solution, we do not mean to generate a new random solution, since if we would generate and evaluate random solutions all the time, there would not be any progressive search towards a better solution. Instead random search would probably work just as good as exhaustive search and is not a heuristic search algorithm. So it should be clear than the function change should keep some part of the old solution in the new solution and change some other part. As an example consider a representation space consisting of bitstrings of some specific length N. It is clear that the representation space in this case is: $S = \{0, 1\}^N$. Now we could make a function change which changes a single bit (i.e. mutating it from 0 to 1 or from 1 to 0). In this case a solution would have N neighbours with this change operator. Now one possible local hill-climbing algorithms would try all solutions in the neighbourhood of the current solution and then select the best one as s_{new}. Or, alternatively, it could select a single random solution from the neighbourhood. In both cases, for many fitness functions, the local hill-climbing algorithm could get stuck in a local optimum. A local optimum is a solution which is not the global optimum (the best solution in the representation space), but

one which cannot be improved using the specific change operator. Thus, a local optimum is the best one in a specific subspace (or attractor in the fitness landscape). Since the local hill-climbing algorithm would not generate a new solution if it has found a local optimum, the algorithm gets stuck and will not find the global optimum. This could be avoided of course by changing the change operator, however this is not trivial. Since if we allow the change operator to change two bits, the neighbourhood would become bigger, but since still not all solutions can be reached, we can again easily get trapped in a local optimum. Only if we allow the change operator to change all bits, we may eventually always find the global optimum, but as mentioned before changing all bits amounts up to exhaustive or random search. A solution to the above problem is to change bits with a specific small probability. In this way, usually small changes will be made, but it is always possible to escape from a local minimum with some probability. Another possibility is used by algorithms such as simulated annealing that always accepts improving solutions, but also can select a new solution with lower fitness value than the current one, albeit with a probability smaller than 1. In specific, simulated annealing accepts a new solution with probability:

$$\min(1, e^{(f(s_{new})-f(s_t))/T})$$

where T is the temperature which allows the algorithm to explore more (using a large T) or to only accept improving solutions (using $T = 0$). Usually the temperature is cooled down (annealed) starting with a high temperature and ending with a temperature of 0. If annealing the temperature from infinity to 0 is done with very slow steps, the algorithm will finally converge to the global optimum. However, in practice annealing should be done faster and the algorithm usually converges to a local maxima just like local hill-climbing. A practical method to deal with this is to use multiple restarts with different initial solutions and finally selecting the best found solution during all runs.

3.2 Genetic Algorithms

In contrast to local hill-climbing and simulated annealing, genetic algorithms use a population of individuals to search for solutions. The advantage of a population is that the search is done in a distributed way and that individuals are enabled to exchange genetic material (in principle the individuals are able to communicate). Making the search using a population also allows for parallel computation, which is especially useful if executing the fitness function costs a long time. However, it would also be possible to parallellize local hill-climbing or simulated annealing, so that different initial solutions are brought to different final solutions after which the best can be selected. Therefore the real advantage lies in the possibility of individuals to exchange genetic material by using recombination operators and by the use of selective pressure on the whole population so that the best individuals are most likely to reproduce and continue the search for novel solutions. A genetic algorithm looks as follows in pseudo-code:

1. Initialize a population of N individuals
2. Repeat:
   (a) Evaluate all individuals in the population using the fitness function
   (b) Repeat N times:
      - Select two individuals for reproduction according to their fitness values

- Recombine these two parent individuals to create one offspring
- Mutate the offspring
- Insert the offspring in a new population

(c) Replace the population by the new population

There is a state of every individual and since a population consists of N individuals, the population also has a state. Therefore after each iteration of this algorithm (usually called a generation), the population state makes a transition to a new state. Finally after a long time, it may happen that the population contains the optimal solution. Since the optimal solution may get lost, we always store the best solution found so far in some place (or alternatively the Elitist strategy may be used that always copies the best found solution to the new population).

3.2.1 Steps for making a genetic algorithm

For solving real world problems with genetic algorithms, such as a time-tabling problem which requires us to schedule for example busses to drivers so that all busses have one driver and no driver has to drive when (s)he indicated that (s)he does not want to drive, the question arises how to make a representation of the problem. This is often more art than science, and research has indicated that particular representations allow better solutions to be found much earlier. For other problems, making a representation does not need to be hard but the chosen representation can influence how fast good solutions are found. Take for example the colouring problem which is also a NP hard problem. In a colouring problem multiple cities may be connected to each other and we want to assign different colors to cities if they are connected. The goal is to find a feasible solution while minimizing the amount of used colors. To solve this problem we may choose a representation which consists of N numbers where N is the number of cities and the number indicates the assigned color to the city. On the other hand, we could also design a representation in which we have a maximum of M colors and NM binary states in which each element of the list of NM states indicates whether the city has that color or not. One should note that the second representation is larger, although it requires only binary states. Furthermore in the second representation it is much easier that false solutions (solutions which do not respect the conditions of the problem) are generated, since it allows for cities to have multiple or 0 colors. Therefore, the first representation should be preferred.

Except for constructing a representation, we also need to find ways to initialize a population, to construct a mapping from genotype to phenotype (the genotype is the encoding in the chromosome on which the genetic operators work, whereas the phenotype is tested using the fitness function), and also to make a fitness function for evaluating an individual (some fitness functions would favour the same optimal solution, but one of these can be more useful for the genetic algorithm to find it).

There are also more specific steps; we need to design a mutation operator, a recombination operator, we have to determine how parents are selected for reproduction, we need to decide how individuals are used to construct a new population, and finally we have to decide when the algorithm has to stop. We will explain these steps in more detail below.

3.2.2 Constructing a representation

The first decision we have to make when we want to implement a genetic algorithm for solving a specific problem is the representation we want to use. As mentioned above, there are often many possible representations, and therefore we have to examine the problem to choose one. Although the representation is often the first decision, we also have to take into account a possible fitness function and which genetic operators (mutation and crossover) we would like to use. For example, if we want to evolve a robot which drives as fast as possible without hitting any obstacles, we could decide to use a function which maps sensory information of the robot to actions (e.g. left motor speed and right motor speed). The obvious representation used in this case would consist of continuous parameters making up the function. Therefore, we may prefer to use particular representations which allow for continuous numbers, although this is not strictly necessary since we may also construct the genotype to phenotype mapping in some way that converts discrete symbols to continuous numbers.

Binary representations and finite discrete sets

The most often used representation in genetic algorithms uses binary values, encoding a chromosome using a bitstring of N bits. See Figure 3.1 for an example. Of course it would also be possible to use a different set of discrete values, e.g. like the one used by biological DNA: $\{C, G, A, T\}$. It depends on the problem whether a binary representation would be more suitable than using different sets of values. It should be said that by concattenating two neighboring binary values, one could also encode each value from a set containing 4 different values. However, in this case a binary encoding would not be preferred, since the recombination operator would not respect the primitive element being a single symbol and could easily destroy such symbols through crossover. Furthermore, a solution in which primitive symbols would be mapped to a single gene would be more readable.

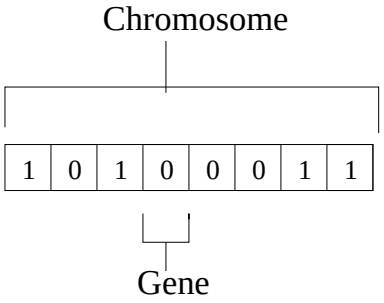

Figure 3.1: A chromosome which uses a binary representation and which is therefore encoded as a bitstring.

If we have a binary representation for the genotype, we can still use it to construct different representations for phenotypes. It should be said that search using the genetic operators takes place in the genotype space, but the phenotype is an intermediary representation which is easier to evaluate by the fitness function. Often, however, the mapping from genotype to phenotype can be an identity mapping meaning that they are exactly the same.

For example, using the 8-bit phenotype given before, we can construct an integer number by computing the natural value of the binary representation. E.g. in the example genotype of

Figure 3.1 we could convert the genotype to the integer: $2^7+2^5+2^1+2^0 = 163$. Alternatively, if we want a phenotype which is a number between 2.5 and 20.5 we could compute $x = 2.5 + \frac{163}{256}(20.5 - 2.5) = 13.9609$.

Thus, using a mapping from phenotype to genotype gives us additional freedom. In the first example, small changes of the genotype (e.g. mutating the first bit) would correspond to big changes in the phenotype (changing from 163 to 35). We note, however, that in the second example, not all solutions between 2.5. and 20.5 can be represented using the limited precision of the 8-bit genotype.

Representing real numbers

If we want to construct a phenotype of real numbers, it is a more natural way to encode these real numbers immediately in the genotype and to search in the space of real numbers. We have already seen that this can lead to more precise solutions, since the binary encoding would have a limited precision unless we use a very long bitstring. Another advantage is that the encoding is much smaller, although this comes at the cost of creating a continuous search space.

Thus, if our problem requires the combined optimisation of n real numbers we could use a genotype $X = (x_1, x_2, \ldots, x_n)$ where $x_i \in \Re$. The representation space would therefore be $S = \Re^n$. For real numbered representations, we have to use a fitness function which maps a solution to a real number, therefore the fitness function is a mapping $f : \Re^n \to \Re$. This encoding is often used for parameter optimisation, e.g. when we want to construct a washing machine which has to determine how much water to consume, how much power to use for turning the cabinet, etc. The fitness function could then trade-off costs versus the quality of the washing machine.

Representing ordering problems

For particular problems there are natural constraints which the representation should obey. An example is the traveling salesman problem which requires a solution that is a tour from a starting city to a last city while visiting all cities in between exactly once. A natural representation for such an ordering problem is to use a list of numbers where each number represents a city. An example is the chromosome in Figure 3.2.

3	4	8	6	1	2	7	5

Figure 3.2: A chromosome which uses a list encoding of natural numbers to represent ordering problems.

3.2.3 Initialisation

Before running the genetic algorithm, one should have an initial population. Often one does not have any a-priori knowledge of the problem so that the initialisation is usually done using a pseudo-random generator. As with all decisions in a GA, the initialisation also depends on the representation, so that we have different possible initialisations:

- Binary strings. Each single bit on each location in the string of each individual receives 50% probability to become a 0 and 50% probability to become a 1. Note that the whole string will likely possess as many 0's and 1's, if we would have a-priori knowledge, we might want to change the a-priori generation constant of 50%. For discrete sets with more than 2 elements, one can choose uniform randomly between all possible symbols to initialize each location in a genetic string.

- Real numbers. If the space of the real numbers is bounded by lower and higher limits, it would be natural to generate a uniform number in between these boundaries. If we have an unbounded space (e.g. the space of real numbers) then we cannot generate uniform randomly chosen numbers, but have to use for example a Gaussian function with a mean value and a standard deviation for initialisation. If one would not have any a-priori information about the location of fit individuals, initialisation in this case would be difficult, and one should try some short runs with different initialisations to locate good regions in the fitness landscape.

- Ordered lists. In this case, we should take care that we have a legal initial population (each city has to be represented in each individual exactly one time). This can be easily done by generating numbers randomly and eliminating those numbers that have been used before during the initialisation of an individual coding a tour.

Sometimes, one possesses a-priori knowledge of possible good solutions. This may be through heuristic knowledge or from previous runs of the genetic algorithm or another optimisation algorithm. Although this has the advantage that the starting population may have higher average fitness, there are also some disadvantages to this approach:

- It is more likely that genetic diversity in the initial population is decreased, which can make the population converge much faster to a population of equal individuals.

- Due to the initial bias which is introduced in this way, it is more difficult for the algorithm to search through the whole state space, possibly making it almost impossible to find a global optimum which is distant from the individuals in the initial population.

3.2.4 Evaluating an individual

Since most operations in a genetic algorithm can be executed in a very short time, the time needed for evaluating an individual is often a bottleneck. The evaluation can be done by a subroutine, a (black-box) simulator, or an external process (e.g. robots). In some cases evaluating an individual can be quite fast, e.g. in the traveling salesman problem the evaluation would cost at most a number of computations which is linear in the number of cities (i.e. one can simply sum all the distances between cities which are directly connected in the tour). In other cases, especially for real world problems, evaluating an individual can consume a lot of time. For example if one wants to use genetic algorithms to learn to control a robot for solving some task, even the optimal controller might already take several minutes to solve the task. Clearly in such a case, populations can not be very large and the number of generations should also be limited. One method to reduce evaluation time for such problems is to store the evaluations of all individuals in memory, so that a possible solution which has already been evaluated before, does not need to be re-evaluated.

If evaluating time is so large, that too few solutions can be evaluated in order for the algorithm to come up with good solutions starting with a random initial population, one could try to approximate the evaluation function by a model which is much faster albeit not as accurate as the real evaluation function. After evolving populations using this approximate fitness function, the best individuals may be further evolved using the real fitness function. A possibility for computing an approximate fitness function is to evaluate a number of solutions and to use a function approximator (such as a neural network) to learn to approximate the fitness landscape. Since the approximate fitness function often does not approximate the real one accurately, one should not run too many generations to find optimal solutions for this approximate fitness function, but only use it to come up with a population which can perform reasonably in the real problem. In case of robotics, some researchers try to come up with very good simulators which makes the evolution much faster than executing the robots in the real world. If the simulator accurately models the problem in the real world, good solutions which have been evolved using the simulator often also perform very well in the real world.

Another function provided by the fitness function is to deal with constraints on the solution space. For particular problems there may be hard or soft constraints which a solution has to obey. Possibilities to deal with such constraints are:

- Use a penalty term which punishes illegal solutions. A problem of this solution is that in some cases where there are many constraints a large proportion of a population may consist of illegal solutions, and even if these are immediately eliminated, they make the search much less efficient.

- Use specific evolutionary operators which make sure that all individuals form legal solutions. This is often preferred, but can be harder to implement, especially if not all constraints in the problem are known.

3.2.5 Mutation operators

In genetic algorithms there are two operators which determine the search for solutions in the genotype space. The first one is mutation. Mutation is used to perturbate (slightly change) an individual so that a new individual is created, but which still resembles the previous one (in genetic algorithms mutation is often performed after recombination so that the previous one is already a new individual). Mutation is an important operator, since it allows us to explore the representation space. Without it, it would become possible that the whole population contains the same allele (value on some locus or location in the genetic string), so that different values for this locus would never be examined. Mutation is also useful to create more diversity and to escape from a converged population which otherwise would not explore different solutions anymore. It is possible to use different mutation operators for the same representation, but it is important that:

- At least one mutation operator should make it possible to search through the whole space of solutions

- The size of the mutation operator should be controllable

- Mutation should create valid (legal) individuals

Mutation for binary representations

Mutation on a bitstring usually is performed by changing a bit to its opposite ($0 \rightarrow 1$ or $1 \rightarrow 0$). This is usually done on each locus of a genetic string with some probability P_m. Thus the mean number of mutations is NP_m where N is the length of the bitstring. By increasing P_m the algorithm becomes more explorative, but may also lose more important genetic material that was evolved before. A good heuristic to set P_m is to set it as $\frac{1}{N}$ which creates a mean number of mutations of 1. Figure 3.3 shows schematically how mutation is done on a bitstring.

Figure 3.3: A chromosome represented as a bitstring is changed by mutation.

In case of multi-valued discrete representations with a finite number of elements, mutation is usually done by first examining each locus and using the probability P_m to choose whether mutation should occur, and if a mutation should occur, each possible symbol has equal probability to replace the previous symbol on that location in the chromosome.

Mutation for real numbers

If a representation of real numbers is used, we also need a different mutation operator. We can use the same way as before to select a locus which will be mutated with probability P_m. But now the value of the locus is a real number. We can perturb this number using a particular form of added randomness. Usually Gaussian distributed zero-mean noise is used with a particular standard deviation, so that we get for the chosen value of the gene x_i in a chromosome:

$$x_i = x_i + N(0, \sigma)$$

Mutation for ordered representations

For mutating ordered representations we should try to make sure that the resulting individual respects the constraints of the problem. That means that for a traveling salesman problem all cities are used exactly one time in the chromosome. We can do this by using a swap of two values on two different loci. Thus we generate two locations and swap their values as demonstrated in Figure 3.4.

3.2.6 Recombination operators

The advantage of using recombination operators is that it becomes possible to combine useful genetic material from multiple parents. Therefore, if one parent has particular good building

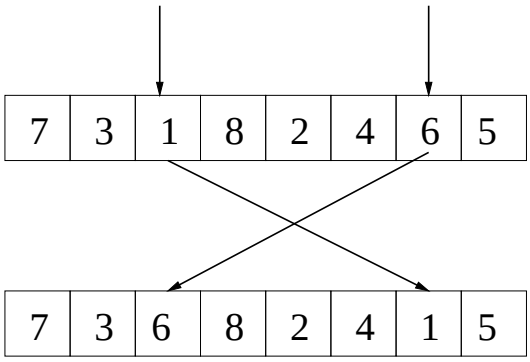

Figure 3.4: A chromosome represented as an ordered list is mutated by swapping the values of two locations.

blocks, and another parent has different good building blocks, the offspring by recombining these parents may immediately possess all good building blocks from both parents. Of course this is only the case if recombination succeeds very well, an offspring may also contain those parts of the parents which are not useful. However, good individuals will be kept in the population and the worse ones will die, so that it is often still useful to use recombination.

A recombination operator usually maps two parent individuals to one or two children. We can use one or more recombination operators, but it is important that:

- The child must inherit particular genetic material from both parents. If it only inherits genetic material from one of the parents, it is basically a mutation operator
- The recombination operator must be designed together with the representation of an individual and the fitness function so that recombination is not often a catastrophe (generating bad individuals)
- The recombination operator should generate legal individuals, if possible

Recombination for binary strings

For binary strings there exist a number of different crossover operators. One of them is 1-point crossover in which there is a single cutting point that is randomly generated after which both individuals are cut at that point in two parts. Then these parts are combined, resulting in two possible children of which finally one or both will be kept in the new population (usually after mutating them as well). Figure 3.5 shows how 1-point crossover is done on bitstrings.

Instead of using a single cutting point, one could also use two cutting points and take both sides of one parent together with the middle part of the other parent to form new solutions. This crossover operator is known as 2-point crossover. Another possibility is to use uniform crossover, here it is decided by a random choice for each location separately whether the value of the first individual or of the second individual is used in the offspring. We can see the different effects of a generated crossover operator using crossover masks. Figure 3.6 shows a crossover mask which is used to create two children from two parents.

Note that these recombination operators are useful for all finite discrete sets and thus wider applicable than only for binary strings.

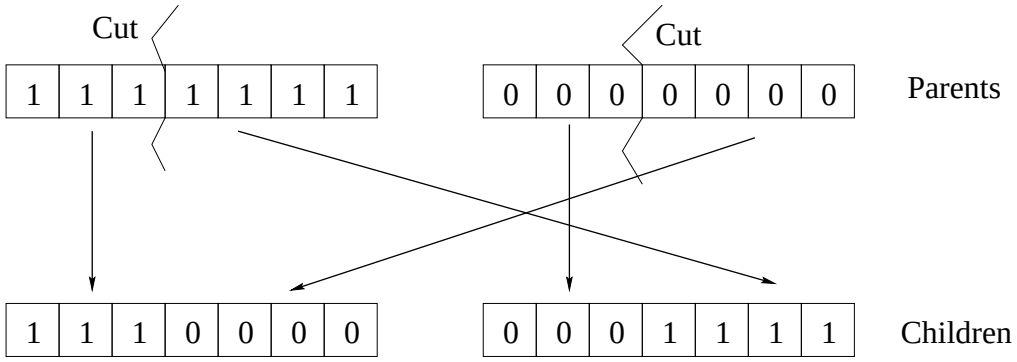

Figure 3.5: The recombination operator known as 1-point crossover. Here the part left to the cutting point of the first parent is combined with the part right to the cutting point of the second parent (and vice versa).

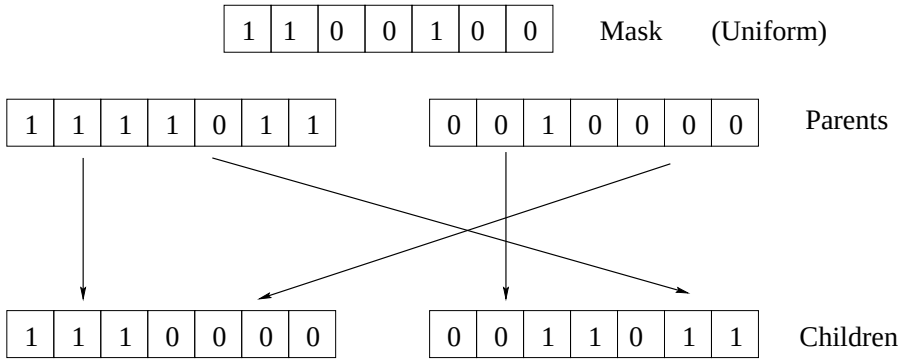

Figure 3.6: The effect of a recombination operator can be shown by a crossover mask. Here the crossover mask is uniformly generated, after which this mask is used to decide which values on which location to use from both parents in the offspring.

Recombination for real numbered representations

If we have representations which consist of real numbers, one might also want to use the recombination operators that are given above for binary strings. However, another option is to average the numbers on the same location, so that we get:

$$(x_1^c = \frac{x_1^a + x_1^b}{2}, \ldots, x_n^c = \frac{x_n^a + x_n^b}{2})$$

The two different recombination operators for real numbers can also be used together by randomly selecting one of them each time.

Recombination for ordered representations

Designing recombination operators for ordered representations is usually more difficult, since we have to ensure that we get children that respect the constraints of the problem. E.g. if we would use 1-point crossover for the TSP, we will almost for sure get children which have

some cities twice and some other cities no time in their representation, which would amount to many illegal solutions. Penalising such solutions would also not be effective, since almost all individuals would become illegal. There has been a lot of research for making recombination operators for ordered representations, but we only mention one possible recombination operator here.

Since the constraint on a recombination operator is that it has to inherit information from both parents, we start by selecting a part of the first parent and copy that to the child. After this, we want to use information from the second parent about the order of values which is not yet copied to the child. This we do by looking at the second parent, examining the order in the second parent of the cities which are not yet inside the child, and attaching these cities in this order to the child. Figure 3.7 shows an illustration of this recombination operator for ordered lists.

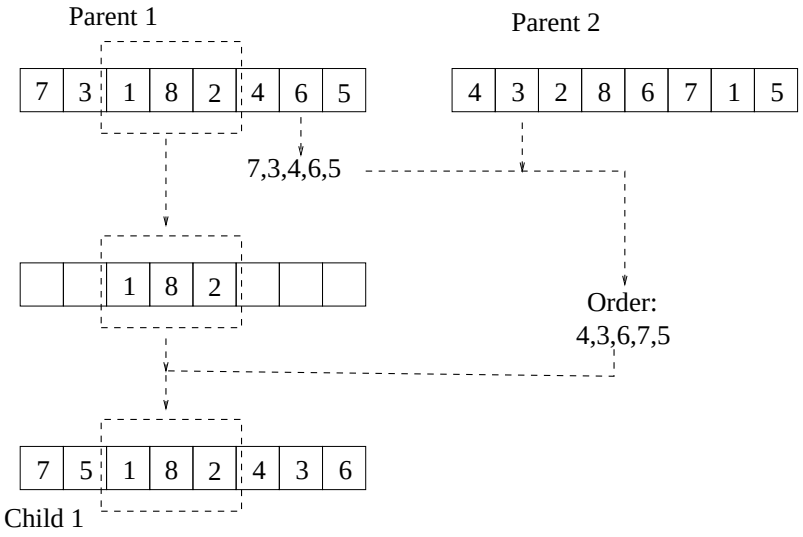

Figure 3.7: A possible recombination operator for ordered representations such as for the TSP. The operator copies a part of the first parent to the child and attaches the remaining cities to the child while respecting their order in the second parent.

3.2.7 Selection strategies

Another important topic in the design of GAs is to select which parents are allowed to create children. If one would always randomly choose parents for creating children, there would not be any selective pressure for obtaining better individuals. Thus, good individuals must have a larger probability for generating offspring than worse individuals. The selection strategy determines how individuals of a population are chosen for generating offspring. Often the selection strategy allows bad individuals to generate offspring as well, albeit with a much smaller probability, although some selection strategies only create offspring with the best individuals. The reason for using less than average fit individuals for creating offspring is that they can still contain good genetic material and that the good individuals may resemble each other very much. Therefore, using bad individuals may create more diverse populations. In the following we will describe a number of different selection strategies.

Fitness proportional selection

In fitness proportional selection, parents which are allowed to reproduce themselves are assigned a probability for reproduction that is based on their fitness. Suppose all fitness values are positive, then fitness proportional selection computes the probability p_i that individual i is used for creating offspring as:

$$p_i = \frac{f_i}{\sum_j f_j}$$

where f_i indicates the fitness of the i^{th} individual. If some fitness values are negative, one should first subtract the fitness of the worst individual to create only new fitness values which are positive. There are some disadvantages to this selection strategy:

- There is a danger of premature convergence, since good individuals with a much larger fitness value than other individuals can quickly take over the whole population
- There is little selection pressure if the fitness values all lie close to each other
- If we add some constant to all fitness values, the resulting probabilities will become different, so that similar fitness functions lead to completely different results

A possible way to deal with some of these disadvantages is to scale all fitness values, for example between values of 0 and 1. For this scaling one might use different functions such as the square root etc. Although this might seem a solution, the scaling method should be designed ad-hoc for a particular problem and therefore requires a lot of experimental testing.

Tournament selection

Tournament selection does not have the problems mentioned above, and is therefore used much more often, also because it is very easy to implement. In tournament selection k individuals are selected randomly from the population without replacing (so each individual can only be selected one time), and then the best individual of this group of k individuals is used for creating offspring. Here, k is known as the tournament size, and is usually set to 2 or 3 (although the best value also depends on the size of the population). Very high values of k cause a too high selection pressure and therefore can easily lead to premature convergence. Figure 3.8 shows how this selection strategy works.

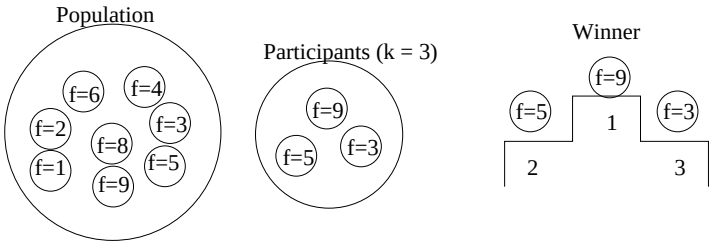

Figure 3.8: In tournament selection k individuals are selected and the best one is used for creating offspring.

Rank-based selection

In rank-based selection all individuals receive a rank where higher ranks are assigned to better individuals. Then this rank is used to select a parent. So if we have a population of N individuals, the best individual gets a rank of N, and the worst one a rank of 1. Then we compute probabilities of each individual to become a parent as:

$$p_i = \frac{r_i}{\sum_j r_j}$$

where r_i is the rank of the i^{th} individual.

Truncated selection

In truncated selection the best $M < N$ individuals are selected and used for generating offspring with equal probability. The problem of truncated selection is that it does not make distinctions between the best and the M^{th} best individual. Some researchers have used truncated selection where the best 25% of the individuals in the population are used for creating offspring, but this is a very high selection pressure and can therefore easily lead to premature convergence.

3.2.8 Replacement strategy

The selective pressure is also influenced by the way individuals of the current population are eliminated to make place for new individuals. In a generational genetic algorithm, one usually kills the old population and replaces it by a completely new population, whereas in a steady-state genetic algorithm at each time one new individual is created which replaces one individual of the old population (usually the worst one). Generational GAs are most often used, but sometimes part of the old population is kept in the new population. E.g. one well-known approach is to always keep the best individual and copy it to the next population, this approach is called Elitism (or elitist strategy). We recall that even if the elitist strategy is not used, we always keep the best found solution so far in memory.

3.2.9 Recombination versus mutation

The two search operators used in genetic algorithms have different usage. The recombination operator causes new individuals to depend on the whole population (genetic material of individuals is mixed). Its utility relies on the schemata-theorem which tells us that if the crossover operator does not destroy good building blocks too often, they can be quickly mixed and stay in the population, since an individual consisting of two good building blocks (schemata) is likely to have a higher fitness value and therefore more likely to propagate its genetic material. In principle, the crossover operator exploits previously found genetic material and leads to faster convergence. In case the whole population has converged to the same individual, the crossover operator will not have any effect anymore. Thus, with less diverse populations, the effect of crossover diminishes.

On the other hand the mutation operator possesses different properties. It allows a population to escape from a single local minimum. Furthermore it allows values of locations which have been lost to be reinserted again. Thus we should regard it as an exploration operator.

Genetic algorithms and evolutionary strategies

Independently on the development of genetic algorithms, Rechenberg invented evolutionary strategies (ES). There is a number of different evolutionary strategies, but in principle ES resemble GA a lot. Like GAs they rely on reproducing parents for creating new solutions. The differences between GA and ES are that ES usually work on real numbered representations and that they also evolve their own mutation parameter σ. Furthermore, most ES do not use crossover, and some ES only use a single individual whereas GAs always use a population.

The choice whether to use crossover or not depends on:

- Is the fitness function separable in additive components (e.g. if we want to maximize the number of 1's in bitstring, then the fitness function is the addition of the fitness of each separate location). In case of separable fitness functions, the use of recombination can lead to much faster search times for optimal solutions.

- Are there building blocks? If there are no real building blocks, then crossover does not make sense.

- Is there a semantically meaningful recombination operator? If recombination is meaningful it should be used.

3.3 Genetic Programming

Although genetic algorithms can be used for learning (robot) controllers or functions mapping inputs to outputs, the use of binary representations or real numbers without a structure does not provide immediate means for doing so. Therefore in the late 1980's Genetic Programming (GP) was invented and made famous by the work and books of John Koza. The main element of genetic programming is the use of functional (or program) trees which are used to map inputs to outputs. E.g., for robot control the inputs may consist of sensory inputs and the outputs may be motor commands. By evolving functional program trees, those programs which work best for the task at hand will remain in the population and reproduce.

A program tree may consist of a large number of functions such as cos, sin, $\times, +, /$, exp, and random constants. These functions usually require a fixed number of inputs. Therefore a program tree must obey some constraints which make it legal. To make a program tree legal, functions which require n arguments (called n-ary functions), should have n branches to child-nodes where each child-node is filled in by another function or variable. The leaf nodes of the tree are input-variables or random constants. Figure 3.9 shows an example of a program tree.

Genetic programming has been used for a number of different problems among which; supervised learning (machine learning) to map inputs to outputs, learning to control robots, and pattern recognition to distinguish between different objects from pixel-data.

Genetic programming is quite flexible in its use of functions and primitive building blocks. Loops, memory registers, special random numbers, and more have been used to solve particular tasks. Like in genetic algorithms, one has to devise mutation and crossover operators for program trees. The other elements of a genetic programming algorithm can be equal to the ones used by genetic algorithms.

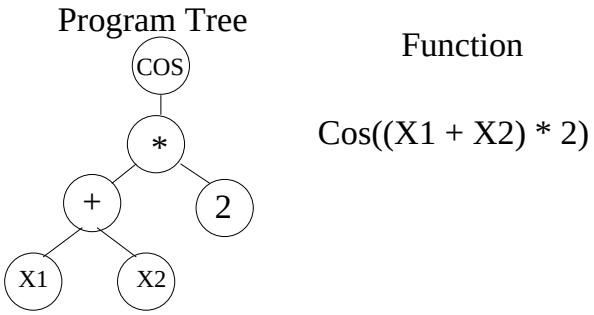

Figure 3.9: A program tree and its corresponding function.

3.3.1 Mutation in GP

The mutation operator can adjust a node in the tree. If the new function in the node will have the same number of arguments, it is easy, but otherwise some solutions have to be found. In the case of point-mutations one only allows mutating a terminal to a different terminal and a function to a different function of the same arity. Other researchers have used mutation of subtrees, in which a complete subtree is replaced by a randomly created new subtree. Figure 3.10 shows an example of a point mutation in GP.

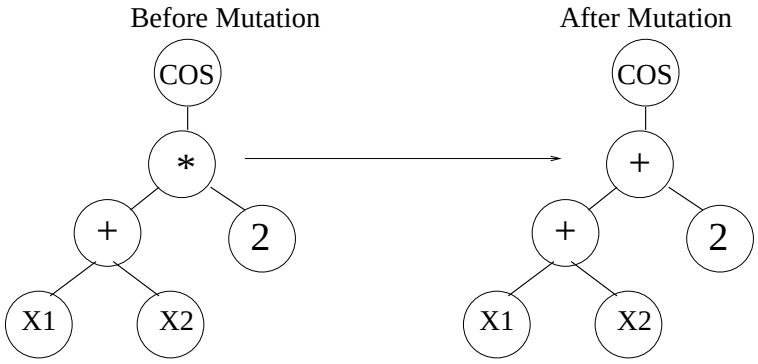

Figure 3.10: Point mutation in genetic programming. A function in a node is replaced by a different function with the same number of arguments.

3.3.2 Recombination in GP

The recombination operator also works on program trees. First particular subtrees are cut from the main program trees for both parent individuals and then these subtrees are exchanged. Figure 3.11 shows an example of the recombination operator in GP.

3.3.3 Probabilistic incremental program evolution

Instead of using a population of individuals, one could also use generative prototypes which generate individuals according to some probability distribution. Baluja invented population based incremental learning (PBIL) which encodes a chromosome for generating bitstrings. For

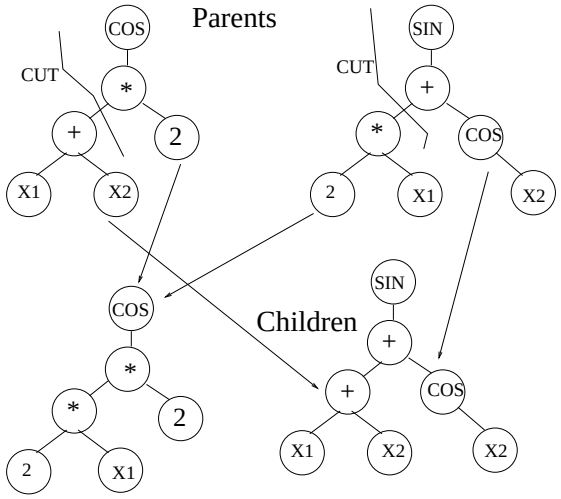

Figure 3.11: Recombination in genetic programming. A subtree of one parent is exchanged with a subtree of another parent.

this the chromosome consists of probabilities for generating 1 on a specific location (and 1 minus that probability for generating a 0). Using this prototype chromosome, individuals can be generated and evaluated. After that the prototype chromosome can be adjusted towards the best individual so that it will generate solutions around the best individuals with higher probability.

This idea was pursued by Rafal Salustowicz for transforming populations of program trees in a representation using a probabilistic program tree (PPT). The idea is known as probabilistic incremental program evolution (PIPE) and it uses probabilities to generate functions in a particular node. The probabilistic program tree which is used for generating program trees consists of a single large tree consisting of probabilities of functions in each node, as shown in Figure 3.12.

The PPT is used to generate an individual as follows:

- Start at the root node and select a function according to the probabilities
- Go to the subtrees of the PPT to generate the necessary arguments for the previously generated functions
- Repeat this until the program is finished (all leaf nodes consist of terminals such as variables or constants)

For learning in PIPE, it is requested that the PPT is changed so that the individuals which are generated from it obtain higher fitness values. For this PIPE repeats the following steps:

- Generate N individuals with the prototype tree
- Evaluate these N individuals
- Select the best individual and increase the probabilities of the functions and terminals used by this best individual

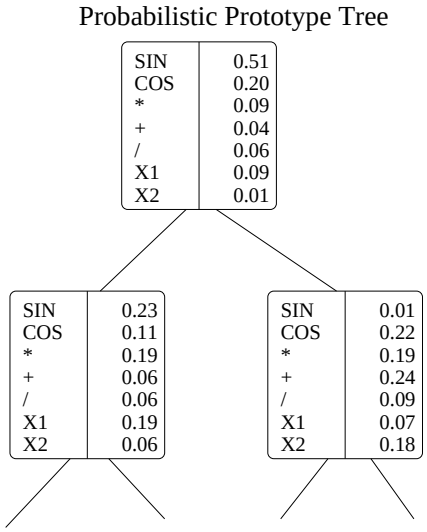

Figure 3.12: The probabilistic prototype tree used in PIPE for generating individuals.

- Mutate the probabilities of the PPT a little bit

PIPE has been compared to GP and it was experimentally found that PIPE can find good solutions faster than GP for particular problems.

3.4 Memetic Algorithms

There is an increasing amount of research which combines GA with local hill-climbing techniques. Such algorithms are known as memetic algorithms. Memetic algorithms are inspired by memes [Dawkins, 1976], pieces of mental ideas, like stories, ideas, and gossip, which reproduce (propagate) themselves through a population of meme carriers. Corresponding to the selfish gene idea [Dawkins, 1976] in this mechanism each meme uses the host (the individual) to propagate itself further through the population, and in this way competes with different memes for the limited resources (there is always limited memory and time for knowing and telling all ideas and stories).

The difference between genes and memes is that the first are inspired by biological evolution and the second by cultural evolution. Cultural evolution is different because Lamarckian learning is possible in this model. That means that each transmitted meme can be changed according to receiving more information from the environment. This makes it possible to locally optimize each different meme before it is transmitted to other individuals. Although optimisation of transmitted memes before they are propagated further seems an efficient way for knowledge propagation or population-based optimisation, the question is how we can optimize a meme or individual. For this we can combine genetic algorithms with different optimisation methods. The optimisation technique which is most often used is a simple local hill-climber, but some researchers have also proposed different techniques such as Tabu Search. Because a local hill-climber is used, each individual is not truly optimized, but only brought to its local maximum. If it would be possible to fully optimize the individual, we would not need a genetic algorithm at all.

The good thing of memetic algorithms compared to genetic algorithms is that genetic algorithms usually have problems in fine-tuning a good solution to make it an optimal one. E.g. suppose that a bitstring contains perfect genetic material except for a single bit. In this case there are much more possible mutations which harm the individual than mutations which bring it to the true global optimum. Memetic algorithms do not have this problem and they also have the advantage that all individuals in the population are in local maxima. However, this also involves a cost, since the local hill-climber can require many evaluations to bring an individual to a local maximum in its region.

Memetic algorithms have already been compared to GAs on a number of combinatorial optimisation problems such as the traveling salesman problem (TSP) [Radcliffe and Surry, 1994] and experimental results indicated that the memetic algorithms found much better solutions than standard genetic algorithms. Memetic algorithms have also been compared to the Ant Colony System [Dorigo et al., 1996], [Dorigo and Gambardella, 1997] and to Tabu Search [Glover and Laguna, 1997] and results indicated that memetic algorithms outperformed both of them on the Quadratic Assignment Problem [Merz and Freisleben, 1999].

3.5 Discussion

Evolutionary algorithms have the advantage that they can be used for solving a large number of different problems. For example if one wants to make a function which generates particular patterns and no other learning method exists, one could always use an evolutionary algorithm. Furthermore, evolutionary algorithms are good in searching through very large spaces and can be easily parallellized.

A problem with evolutionary algorithms is that sometimes the population converges prematurely to a suboptimal local minimum. Therefore a lot of research effort has come up with methods for keeping diversity during the evolution. Another problem is that many individuals are evaluated and then never used anymore, which seems a waste of computer power. Furthermore, the learning progress can be quite slow for some problems and if many individuals have the same fitness value there is not much selective pressure. E.g. if there is only a good/bad final evaluation, it is very hard to come up with solutions which are evaluated good if in the beginning all individuals are bad. Therefore, the fitness function should be designed in a way to provide maximal informative information.

A lot of current research focuses on "linkage learning". We have seen that recombination is a useful operator which can allow for quickly combining good genetic material (building blocks). However, uniform crossover is very disruptive, since it is a random crossover operator it does not keep building blocks as a whole together. On the other hand 1-point crossover may keep building blocks together if the building blocks are encoded on bits which lie nearby on a genetic string (i.e. next to each other). It may happen, however, that a building block is not encoded in a genetic string as material next to each other, but distributed over the whole string. In order to use effective crossover for such problems one must identify the building blocks which is known as linkage learning. Since building blocks can be quite large, finding the complete block can be very difficult, but effective progress in this direction has been made.

Chapter 4

Physical and Biological Adaptive Systems

Before the 16'th century, the Western thinkers believed in a deductive approach to acknowledge truth. For example, Aristotle always thought that heavy objects would fall faster to the ground than lighter objects. It was not until Galileo Galilei (1564-1642) tested this (according to some he did his experiments by dropping objects from the tower of Pisa), that this hypothesis turned out to be false (if we disregard air-resistance). After this Galilei played an important role to use mathematics for making predictive models and he also showed that planets were going around the sun instead of around the earth (this hypothesis he had to retract from the church). This was the start of a natural science where experiments were used to make (predictive) models. Christiaan Huygens also played an important role by his discovery of much better clocks to make measuring time much more precise, his discovery of better lenses and telescopes, and the discovery that light could be described by waves instead of particles. The new science continued with Kepler (1571 - 1630) who approximated the orbits of planets and came up with ellipsoids to predict them instead of the commonly thought hypothesis that the orbits should be approximated using circles.

Isaac Newton (1642-1727) discovered the gravitation laws and laws of mechanics which were the final breakthrough for a new natural science. Newton's gravitation laws tells that two objects (e.g. planets) attract each other based on the multiplication of their masses divided by the square of the distance between them, and it is very accurate for big objects which do not move at very high speed (for very small moving particles quantum mechanics introduced different laws, and for very high speed relativity theory was later invented). Newton's laws of mechanics were also used to predict that planet orbits were ellipsoids and that planets will circle around the sun whose movement is hardly influenced by the planets.

After this fruitful period of scientific revolutions, researchers started to think that the universe worked like a clock and that everything could be predicted. This even led to the idea of a Genius by Laplace which would be an almighty entity which could predict the future and the past based on the current state and the mechanical laws. Although this idea of a universal clock brought many fruitful machines such as computers and television, already in the start of the 19'th century Poincaré had discovered that not everything could be predicted. Poincaré was studying three body problems, like three planets moving around each other, and discovered that there were not enough known equations to come up with a single analytical solution for predicting their movements. This eventually led to chaos theory, where a model

can be deterministic, but still shows unpredictable behavior if we cannot exactly measure the initial state.

Although the word chaos often refers to a state without order, researchers have found remarkable structures in chaotic systems. Even in chaotic systems there seems to be a kind of self-organisation. In this chapter we will look at the path from physics to biology, take a look at chaotic systems, and then we will examine self-organising biological systems such as ants and the use of these systems for solving optimisation problems.

4.1 From Physics to Biology

In Newtonian mechanics, the systems are reversible, which means that we can turn around the arrow of time and compute the past instead of the future. There are specific laws of physical systems such as the conservation of energy which states that the sum of potential and kinetic energy of an objects must remain constant. An example is a ball which we throw in the air. In the beginning the kinetic energy (due to its speed) is maximal, and it will become 0 at the highest point where the potential energy (due to gravitation) is maximal. Then it will fall again while conserving its energy until finally it bounces against the ground and will lose energy due to this (in reality the height of the ball will be damped due to friction which causes a loss of energy. Without loss of energy the ball would continue bouncing forever).

If we have energy preserving systems, the system will continue with its movement. A good example is a pendulum. Suppose a pendulum is mounted at some point, and there is no friction at this point or friction due to air resistance. Then we give the clock a push to the right and it will remain moving to the left and to the right. If we give the pendulum a harder push, it will go around and continue going around. Let's look at the phase diagram in Figure 4.1 that shows possible trajectories in the plane with the angle on the x-axis, and the (normalised) angular speed on the y-axis.

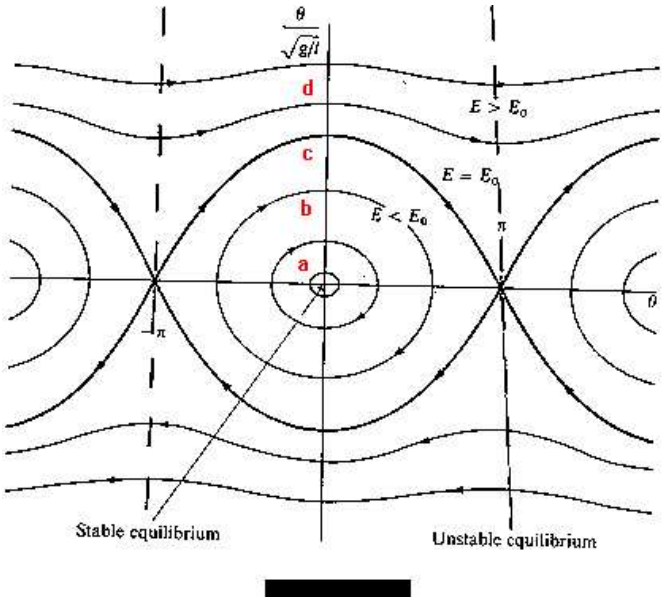

Figure 4.1: The phase diagram of the pendulum

In the middle of the figure a stable equilibrium is shown, the pendulum is not moving at all. Trajectories a and b show periodic cycles (orbits) where the pendulum is moving to the left, right, left, etc. Orbit c leads to an unstable equilibrium in which the pendulum goes to the highest point and there it stops to move. This point is unstable, because a slight perturbation will cause it to move again. Finally, in orbit d the pendulum is going over its head.

The pendulum is an example of a reversible system in which energy is conserved. Ideally, such mechanical systems always conserve their energy. However, there are also many systems which are irreversible, which are thermodynamic objects. After the industrial revolution, many scientists were interested in making the optimal (perpetuum mobile) machine; one which would continue to work forever. But soon they discovered that every machine would lose useful energy due to production of heat. An example of a thermodynamic object is a system which consists of a box with 2 halves. In one half there are N gas-molecules and in the other half there are none. The initial state is very ordered since all the gas-molecules are at the left half. Now we take away the border between the halves and we will observe that after some time both halves will contain roughly the same amount of molecules. This is an example of an irreversible system since if the system would be in a state with the same amount of molecules in both halves it would probably never go to the state with all molecules in one half again. To describe such processes, Boltzmann invented the word **entropy**. Entropy corresponds to the amount of disorder which is caused by the production of useless energy such as heat which cannot be turned back to make energy without a heat potential. Entropy has also been widely used in thermodynamics to explain why heat will always flow from a hot space to a colder space.

Consider the case of the N gas molecules again. Boltzmann used a statistical explanation why the molecules would mix and a state of disorder would arise. Consider now N molecules and the number of permutations that can describe a possible state. For example all N molecules in one half of the box would only have one possible state, one molecule in one half and the rest in the other half would have N possible states. Now if we divide the N molecules in N_1 and N_2 molecules in both halves, the number of permutations would be:

$$P = \frac{N!}{N_1!N_2!}$$

Now its logical that the system will go to an equilibrium with most possible states, that is where $N_1 = N_2$. For this Boltzmann defined entropy of a system as:

$$S = k \log P$$

Where k is called the Boltzmann constant.

So although all microscopic states are equally possible, due to the effect that there are much more microscopic states around the macroscopic situation of having the same number of molecules in both halves, this situation will arise after some time. Of course small deviations from the macroscopic equilibrium can happen, but the system's state will oscillate around this equilibrium. We can see that entropy is maximised and that disorder will be the result. Since entropy production is always positive in a closed system and there is a state with maximal entropy, the system will always converge to such an equilibrium. Since the initial state gets lost in this case, the process is not reversible (many states lead to the same final state). Note the difference with the energy-preserving pendulum which is reversible. It is important to

note that there are many irreversible processes in machines caused by loss of heat, friction etc. so that is is not possible to make a machine which continues forever without receiving additional energy. These processes cause an increase of the entropy of the system.

But how is this with open systems such as living systems? Here the change of entropy of the system is governed by an internal change of entropy dS_i/d_t which is irreversible and always positive, and an exchange of entropy between the system and its environment dS_u/d_t which can be positive or negative. We note that the exchange of entropy of a closed system is not possible (since there is no environment to interact with) so that the entropy can only increase or remain constant (at the maximal value). In this case, the entropy determines the direction of time; for all closed systems their future lies in the direction of increased entropy. This lead to the two laws of the thermodynamics by Clausius in 1865:

- The energy of the world is constant
- The entropy of the world goes to a maximal value

Thus in the thermodynamic equilibrium the entropy and disorder will be at its maximum. However, living systems can exchange entropy with their environment. This allows them to keep their entropy low. E.g. by consuming food and energy, a living system is able to keep its order without having to increase its entropy. This is the essential difference between open and closed systems. An open system can receive useful energy from the environment and thereby it can reduce its disorder and create more order.

4.2 Non-linear Dynamical Systems and Chaos Theory

As mentioned before, Poincaré had already discovered that there are no analytical solutions to be found for the n-body problem with n larger than 2. For 2 planets, there is an analytical solution which determines the position and velocity of both interacting planets given the initial conditions. These planets will move around their point of joint mass as shown in Figure 4.2.

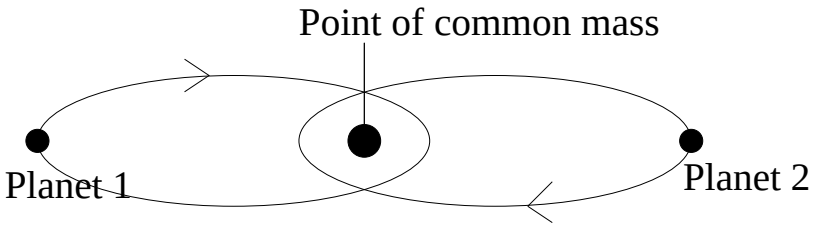

Figure 4.2: The orbits of two interacting planets

For the n-body problem with $n \geq 3$, Poincaré had demonstrated that there were not enough differential equations to be able to compute a solution to the problem. The problem was therefore not integratable to a closed-form analytical solution. Poincaré has also demonstrated that small perturbations could cause large differences of trajectories in this case. This was the first time chaotic dynamics had been mentioned.

After this, for a long time few researchers were studying chaotic systems. One major breakthrough in their understanding came when computers were used which could visualise

such processes. The first famous demonstration of chaos using computer simulations was described by the meteorologist Edward Lorenz who was studying weather prediction. In 1961 he saw an event in his computer simulations. By accident he discovered sensitivity to initial conditions, since he wanted to repeat his simulations, but found completely different results. After some time he discovered that the values he used in his second simulation were rounded to three decimals, whereas the computer used values with 6 decimals during the entire run. These minimal differences quickly caused large deviations as is seen in Figure 4.3.

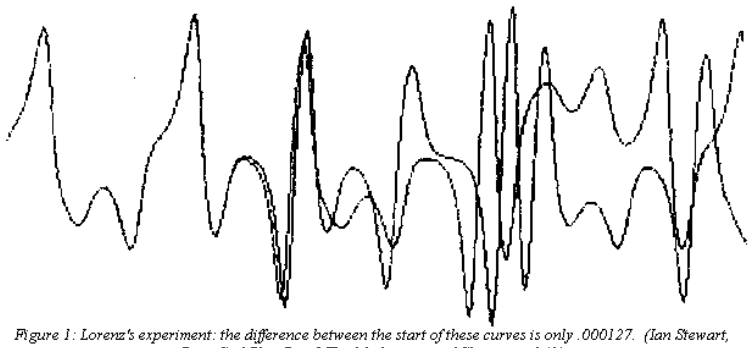

Figure 1: Lorenz's experiment: the difference between the start of these curves is only .000127. (Ian Stewart, Does God Play Dice? The Mathematics of Chaos, pg. 141)

Figure 4.3: The simulations done by Lorenz showed sensitivity to initial conditions. Although the initial values were almost similar, the difference between the trajectories became very large.

In chaos theory it is often said that little causes create big consequences. After simplifying his model to three variables, he first noted something like random behavior, but after plotting the values in a coordinate space, he obtained his famous Lorenz attractor depicted in Figure 4.4. We can see an ordered structure, so again we should not confuse chaotic dynamics with non-determinism.

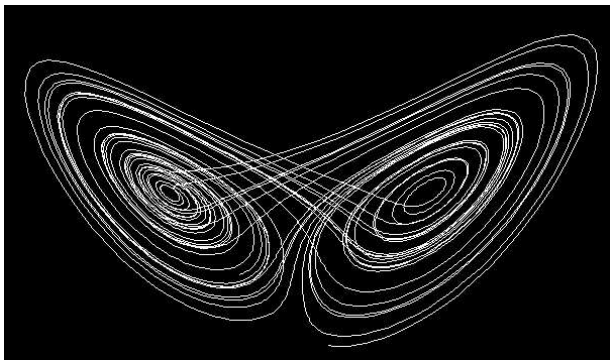

Figure 4.4: The Lorenz attractor

The dynamical system of Lorenz is quite complicated to analyse, and therefore we will use an example from biology to demonstrate chaotic dynamics in an easier way.

4.2.1 The logistic map

Around 1800, T.R. Malthus assumed that the growth of a population would be linear with the number of individuals $x(t)$. The mathematical expression is the differential equation:

$$\frac{dx}{dt} = kx$$

which has as closed-form solution an exponential growing population:

$$x(t) = x(0)exp(kt)$$

In 1844 P.F. Verhulst noted that for a growing population there must arise competition so that the population would stop growing at some time. He noted that the population would grow linearly with the number of individuals and the difference between the number of available sources and the sources needed to sustain the population. This model is known as the following Verhulst equation:

$$\frac{dx}{dt} = Ax(N - x)$$

with AN the maximal number of available sources and Ax the amount needed for x persons. The logistic map equation can be derived from this in which we use discrete time and change variables. The logistic map equation looks as follows:

$$x(t + 1) = rx(t)(1 - x(t))$$

Where x has a value between 0 and 1. For values of r below 1, the population will die out ($x(\infty) = 0$). If r is between 1 and 3, there is one single final state $x(\infty)$. Now if we keep increasing r, there will arise period-2 cycles and higher periodic cycles. Each value for r that causes the period to increase (in the beginning it doubles) is called a bifurcation point. Figure 4.5 shows a period-2 cycle of this map with a value of r a little bit larger than 3.

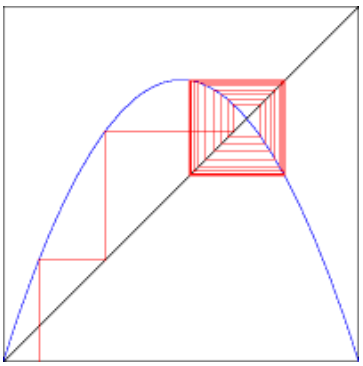

Figure 4.5: A period-2 cycle of the logistic map.

Figure 4.6 shows a larger periodic cycle. Although the periodic attractor is difficult to see, it is important to note that trajectories from different starting points x_0 approach this limit cycle.

Now, look what happens if we plot the value of r to the values which x can take after a long transient period (so we eliminate the initial values $x(t)$ by waiting for 1000 steps). This

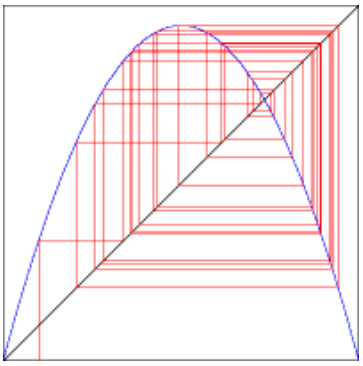

Figure 4.6: A larger periodic cycle of the logistic map.

plot is shown in Figure 4.7. The figure shows a very complicated bifurcation diagram. In the beginning there is a single steady state (for $r \leq 1$ all trajectories go to $x(\infty) = 0$). When $r > 1$ and $r < 3$ there is a single stable state for x, although the final value $x(\infty)$ depends on r. Now if we increase r to a value higher than 3, there is a periodic cycle of length 2, which is shown in the bifurcation diagram by the two branches which determine the multitude of values of x which are part of periodic cycles. Increasing r further leads to periodic cycles of length 4, 8, 16, etc. Until finally the period becomes infinite and the system shows chaotic behavior.

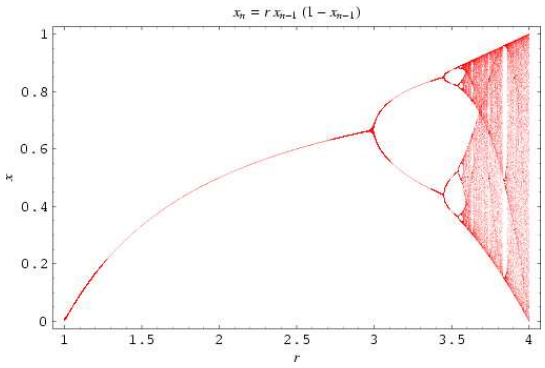

Figure 4.7: A plot of the value of the control parameter r to the values which x will take after some transient period.

In Figure 4.8, we see a more detailed figure of this bifurcation diagram for values of r between 3.4 and 4. It shows that although there are values of r displaying chaotic behavior, for some values of r there are again periodic cycles, which is shown by the bands with only few branches. If we further zoom in in the area of Figure 4.8, we get the figure displayed in Figure 4.9. This figure shows clearly that there are periodic cycles alternating with chaotic dynamics.

A remarkable property of the chaotic dynamics generated by the logistic map is when we further zoom in in the area of Figure 4.9 and get the Figure 4.10. This figure clearly shows that there is a self-similar pattern on a smaller scale. Again we see bifurcation points

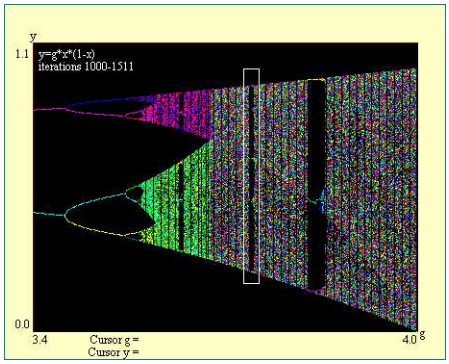

Figure 4.8: A plot of the value of the control parameter r to the values which x will take after some transient period.

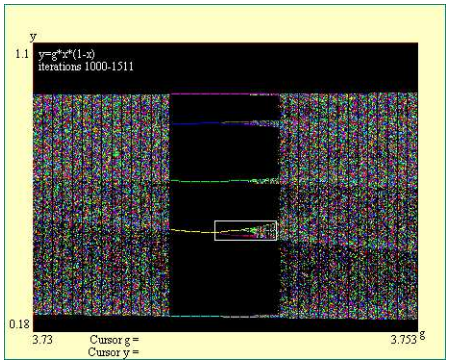

Figure 4.9: A plot of the value of the control parameter r between 3.73 and 3.753 to the values which x will take after some transient period.

and periodic lengths which double, until again it arrives at chaotic dynamics which visit an infinite number of points.

So what can we learn from this? First of all even simple equations can display chaotic behavior. For a map (or difference equation) chaotic dynamics can be obtained with a single variable (the population x). When using differential equations it turns out that there need to be three differential equations which form a non-linear system in order for the system to display chaotic behavior. Furthermore, when chaotic dynamics arise, even a very small difference between two initial states can cause a very different trajectory. This means that if we cannot exactly measure the initial state our hope to predict the future dynamics of the system is lost. Of course, here we have shown simple mathematical equations leading to chaotic behavior, the question therefore is whether chaos also arises in real natural systems. The answer to this is yes; research has turned out that the heartbeat follows an irregular non-periodic patterns, and using a EEG it was shown that the brain also possesses chaotic dynamics. Furthermore, in biological systems the population of particular kinds of flies also shows chaotic behavior. And of course to come back to Lorenz, the weather is unpredictable since it is very sensitive to initial conditions. This sensitivity in chaos theory is often related to the possibility that a butterfly in Japan can cause a tornado in Europe.

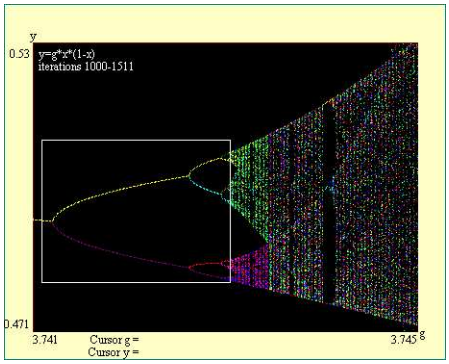

Figure 4.10: A plot of the value of the control parameter r between 3.741 and 3.745 to the values which x will take after some transient period.

Instead of only disorder, we can also see ordered patterns in chaotic systems. One example is the self-similar structure if we look at the pattern of a bifurcation diagram at different scales. Furthermore, if we look at the Lorenz attractor, we can see that not all states are possible; the state trajectories are all on a particular manifold (subspace of the whole space). On the contrary, when we would use a stochastic (non-deterministic) system, the trajectories would fill up the whole phase diagram. In reality chaos therefore also displays order, which is also the statement of Ilya Prigogine; "order out of chaos".

4.3 Self-organising Biological Systems

Adaptive systems can be used fruitfully to model biological systems. We have already seen that the model can consist of mathematical equations, but they can also have a spatial configuration using individualistic models such as cellular automata. The advantage of using individualistic models moving in a particular space is that there is an additional degree of freedom for the physical space and therefore additional emergent patterns. By simulating an individualistic model, it also becomes much easier to visualise processes such as the fire propagation in forest fires. The disadvantage of spatial models compared to mathematical equations is that it is much slower to simulate. Some examples of biological models which can be modelled are:

- Infection diseases
- Forest fires
- Floods
- Volcano eruptions
- Co-evolving species

The first four processes mentioned above show a common aspect; they propagate themselves over paths which depend on the environment. To stop the propagation, such paths should be "closed". This is essential for controlling these natural disasters, but will not be the issue in this chapter.

4.3.1 Models of infection diseases

We will look at two different models for simulating infection diseases. In infection diseases, we can distinguish between three kinds of individuals in the population:

- Healthy individuals (H)
- Infected, sick individuals (S)
- Immune individuals which have had the disease (I)

If a healthy person comes in the neighbourhood of an infected individual, the healthy person will also become infected in our model (although usually this will only happen with some probability). If an infected person has been sick long enough, it becomes an immune individual which is not sick anymore.

Mathematical model of infection diseases

We can make a model using difference equations. We start with a state of the population: $S(0) = (H(0), I(0), S(0))$, and use the following equations to determine the evolution of the system:

$$\begin{aligned} S(t+1) &= S(t) + S(t)H(t) - bS(t) \\ I(t+1) &= I(t) + bS(t) \\ H(t+1) &= H(t) - aS(t)H(t) \end{aligned}$$

Here we have two control parameters a and b. Note that the values H, I, S should not become negative! If we examine the model, we can see that the number of immune individuals is always increasing or stays equal. Therefore a stable attractor point is a situation with all people immune to the disease. However, if the control parameter b is set to a very large value, the population of sick people might decrease too fast and might become 0 before all healthy people became sick. Therefore other possible stable states include a number of healthy and immune people. Also when there are no sick or immune people at all at start, the stable point would consist only of healthy people.

Cellular automaton model of infection diseases

We can also use a cellular automaton (CA) in which we have to make rules to update the state of the CA. Suppose we take the 2-dimensional CA with individuals as shown in Figure 4.11. Cells can be empty or be occupied by a sick, immune, or healthy person.

The CA also needs transition rules to change the state of the system, we can make the following rules:

- If H has a S in a cell next to it, the H becomes a S.
- S has each time step a chance to become a I
- For navigation, all individuals make a random step at each time-step

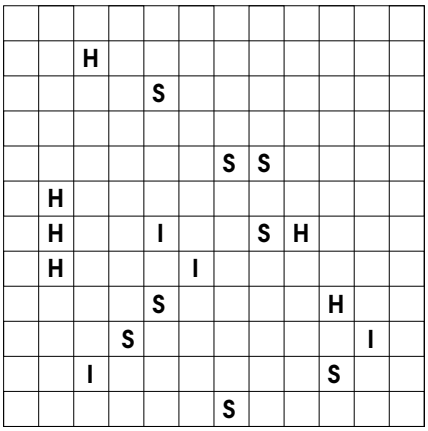

Figure 4.11: The CA for infection diseases. H = healthy person, I = immune person, S = sick person

Step 2 above uses a probability to change a state of a cell and navigation also used randomness, therefore this is an example of a stochastic cellular automaton. Finally, we can also make another navigation strategy so that healthy persons stay away from sick individuals. This could lead to different evolving patterns where healthy persons are in one corner, far away from the sick individuals.

4.4 Swarm Intelligence

It is well known that a large group of simple organisms such as ants or bees can show intelligent behavior. The question is how this collective intelligent behavior emerges from simple individuals. In this section we will see examples of this phenomenon and how this self-organising collective behavior can be used for making optimisation algorithms.

First we will look at some smart collective behaviors:

- Foraging behavior: individuals search for food and bring it to their nest
- Protection of the nest: individuals have received an altruistic and non-producing task which helps the group to survive
- Building of a nest: E.g. how do termites construct their nest or how are honeycombs made by bees.
- Stacking food and spreading it

It is clear that there is no super controller which sends the individuals messages how to do their task. In some ways the behaviors emerge from simple individual behaviors. E.g. if we look at the process of creating honeycombs, then we can see that the structure emerges from local interactions between the bees. Every bee creates a single cell in the wax by hollowing out part of the space of the wax. Whenever a bee makes a cell it takes away parts of the borders of the cell. When it feels that there is another bee working in the cell close next to it, it stops taking wax out of the direction of that bee. In this way a hexagonal pattern emerges with very similar cells (because bees have similar sizes), see Figure 4.12.

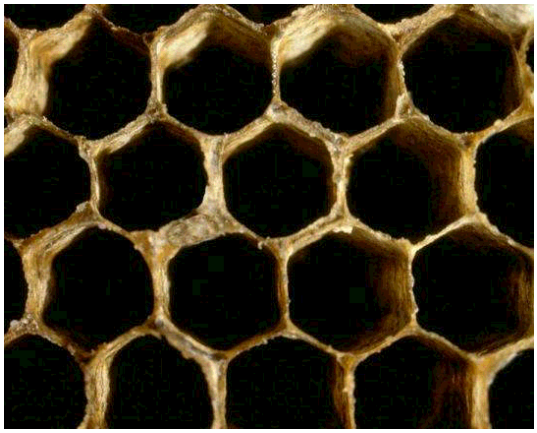

Figure 4.12: A honeycomb

It is also known that ants can solve particular problems, such as finding the shortest path to a food pile, clustering or sorting food, and clustering dead ant bodies. Although a single ant is not intelligent, the whole colony shows intelligent group behavior (super-intelligence).

4.4.1 Sorting behavior of ant colonies

When many ants die at the same time, the living group makes cemeteries of the dead ants by stacking them all on the same place. How can this be done if single ants are not intelligent enough to know where to put the dead ant they may be carrying? To explain this, we can make a simple model with three rules:

- An ant walks in arbitrary directions
- Whenever an ant does not carry anything and finds a dead ant, it takes it and will carry it to some different place
- Whenever an ant carries a dead ant and sees a pile of dead ants, it will drop the ant near that pile

These three simple rules can explain the group-behavior of sorting ants. A similar model can be made to let ants make piles of sugar and chocolate. Since each ant is very simple, it would take a long time until some organisation would emerge using a single ant. However, when many ants are used, the self-organisation of matter in the space can occur at very short time periods. This is also a reason why some researchers investigate collective swarm robotics to make many simple small robots collaborate together to perform different tasks, instead of a single large robot which has to do everything alone.

4.4.2 Ant colony optimisation

A new kind of multi-agent adaptive system for combinatorial optimisation has been invented by Marco Dorigo in the 90's. In this algorithm, a colony of ants works together to find solutions to difficult path-planning problems such as the traveling salesman problem. The algorithm is inspired by how ant colonies work in reality. The foraging ants leave a chemical

substance known as **pheromone** on the ground when they go from their nest to a food source and vice versa. Other foraging ants follow the paths with most pheromone according to a probability distribution. While following these paths they strengthen them by leaving additional pheromone. This collective foraging behavior enables an ant colony to find the shortest path between the nest and a food source. Optimisation algorithms which are inspired by the collective foraging behavior of ants are called ant colony systems or simply ant algorithms.

We will first examine combinatorial optimisation problems which determines the class of problems which are hard to solve and for which ant colony systems can be applied.

Combinatorial optimisation

Particular problems cost exponential amount of time to solve. To get an idea of an exponential problem, consider a solution that consists of n states and the time to solve it is 2^n or $n!$. An example is to find a bitstring of only 1's when the fitness is 0 for all solutions except for the state with all 1's which gets higher fitness (known as a needle in a haystack problem). Exponential time problems grow much faster than polynomial time problems:

$$\lim_{n\to\infty} \frac{n^p}{e^n} \to 0$$

Where p is the degree of some polynomial function and e is the natural exponent. A number of well known mathematical problems are called combinatorial optimisation problems, a subset of these are NP-complete problems which cannot be solved in polynomial time unless P = NP. The question P = NP is known as one of the open and most important questions in computer science and optimisation. The interesting thing is that if one of these NP-complete problems can be solved by some algorithm in polynomial time, all these problems can be solved in polynomial time. So far no polynomial time algorithm has been found to solve one of these problems, however.

Since computer power cannot increase faster than exponential time (Moore's law states that computer power doubles every two years), some big combinatorial optimisation problems can never be solved optimally. Some examples of combinatorial optimisation problems are:

- The traveling salesman problem: find the shortest tour through n cities
- Quadratic assignment problem: minimize the flow (total distance which has to be travelled) if a number of employees has to visit each other daily in a building according to some frequency. So the total cost is the product of the distance matrix and the frequency matrix. The problem requires to assign locations to all people to minimize the total cost. This often involves putting people who meet each other frequently in nearby locations.
- 3-satisfiability: Find truth-values for n propositions to make the following kind of formula true:
$$\{x_1 \vee \neg x_2 \vee x_4\} \wedge \ldots \wedge \{x_1 \vee \neg x_5 \vee x_7\}$$
- Job-shop scheduling: Minimize the total time to do a number of jobs on a number of machines where each job has to visit a sequence of machines in a specific order and each machine can only handle one job at a time.

We will elaborate a bit on the traveling salesman problem here, since ant algorithms were first used to solve this kind of problem. In the traveling salesman problem (TSP) there is a seller which wants to visit n cities and come back to his starting city. All cities i and j are connected with a road of distance $l(i, j)$. These lengths are represented in a distance matrix. The agent must compute a tour to minimize the length of the total tour. An example of a tour with 8 cities with distance 31 is shown in Figure 4.13.

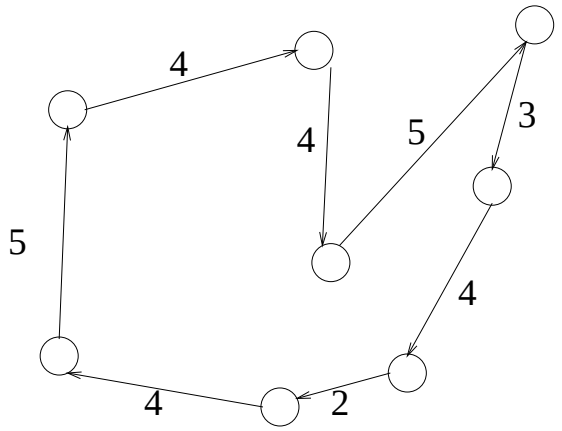

Figure 4.13: A tour in a traveling salesman problem

How can we generate a tour for the traveling salesman problem? The constraints are that all cities have to be visited exactly once and that the tour ends at the starting city. Now we keep a set of all cities which have not been visited: $J = \{i \mid i \textit{ is not visited}\}$. In the beginning J consists of all cities. After visiting a city, we remove that city from the set J. The algorithm for making a tour now consists of the following steps:

1. Choose an initial city s_0 and remove it from J
2. For $t = 1$ to n:
   (a) Choose city s_t out of J and remove s_t from J
3. Compute the total length of the tour:
   $L = \sum_{t=0}^{N-1} l(s_t, s_{t+1}) + l(s_N, s_0)$

Of course the most important thing is to make the rule for choosing the next city given the current one and the set J. Different algorithms for computing tours can be compared to the final value L returned by the algorithm (note that for very large problems, it is extremely hard to find an optimal solution, so that an algorithm should just find a good one).

4.4.3 Foraging ants

One algorithm for making an adaptive rule for selecting the next city given the current one and the set J is inspired on the collective foraging behavior of ants. We will first examine why ants can find shortest paths from the nest to a food source. Let's have a look at Figure 4.14. It shows two paths from the left to the right and ants approaching the point where they

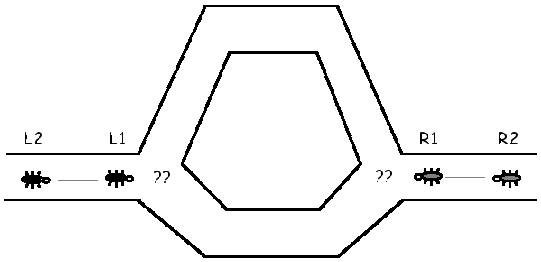

Figure 4.14: In the beginning the ant colony does not have any information about which path to take

need to choose one of them. In the beginning their choice will be completely random, so 50% will take the upper path and the other 50% the lower path.

Now in Figure 4.15 it becomes clear that ants which took the lower path will arrive at the destination earlier than those which took the upper path. Therefore, as we can see in Figure 4.16, the lower path will accumulate more pheromone and will be preferred by most ants, leading to more and more strengthening of this path (see Figure 4.17).

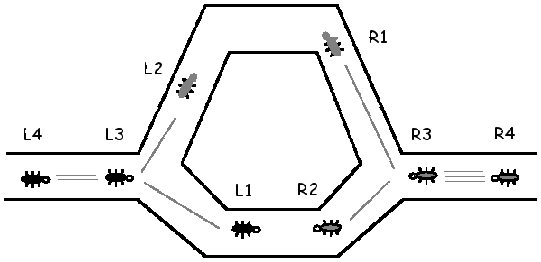

Figure 4.15: Ants which take the lower path will arrive sooner at the destination

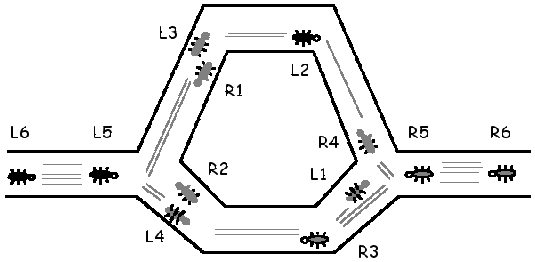

Figure 4.16: This causes more ants to follow the lower part

4.4.4 Properties of ant algorithms

There are multiple different ant algorithms, but they all share the following properties:

- They consist of a colony of artificial ants

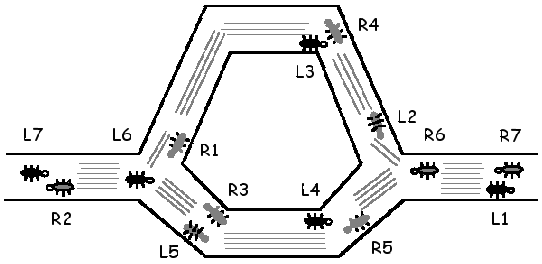

Figure 4.17: The amount of pheromone keeps on strengthening more along the lower path than along the upper path, therefore finally almost all ants will follow the lower path.

- Ants make discrete steps
- Ants put pheromone on chosen paths
- Ants use the pheromone to decide which steps to make

Ant algorithms have been used for a wide variety of combinatorial optimisation problems such as the traveling salesman problem, the quadratic assignment problem, and network routing. The idea to let individuals interact, because one of them changes the environment is called **stigmercy**. The first ant algorithm, the ant system, was initially tested on the TSP. It works as follows:

- All N ants make a tour for which they use pheromone between cities to select the next city
- All not followed edges lose a bit of pheromone due to evaporation
- All followed edges receive additional pheromone where edges belonging to shorter tours receive more pheromone.

The ant-system was thereafter changed in some ways and this led to the ant colony system. We will now give a formal description of the ant-colony system used for the traveling salesman problem (although the ant-colony system is often called a meta-heuristic that includes many possible algorithms and can be used for different problems).

The ant colony system consists of K ants. The amount of pheronome between 2 cities i and j is denoted as $m(i,j)$. For choosing the next city an additional heuristic is used which is the inverse of the length between two cities: $v(i,j) = \frac{1}{l(i,j)}$.

Now every ant: $k = 1 \ldots k$ makes a tour:

- Choose a random starting city for ant $k : i = random(0, N)$ and take the city out of the set J_k of unvisited cities for ant k
- Choose next cities given the previous one according to:

$$j = \begin{cases} \arg \max\limits_{h \in J_k} \{[m(i,h)] \cdot [v(i,h)]^\beta\} & \text{if } q \leq q_0 \\ S & \text{else} \end{cases} \tag{4.1}$$

  Here β is a control parameter, $0 \leq q \leq 1$ is a random number, and the control parameter $0 \leq q_0 \leq 1$ determines the relative importance of exploration versus exploitation. If

exploration is used, we generate S which is a city chosen according to the probability distribution given by the following equation:

$$p_{ij} = \begin{cases} \frac{[m(i,j)]\cdot[v(i,j)]^\beta}{\sum_{h\in J_k}[m(i,h)]\cdot[v(i,h)]^\beta} & \text{if } j \in J_k \\ 0 & \text{else} \end{cases} \tag{4.2}$$

Now all ants have made a tour and we can update the pheromone trails as follows. First we compute which generated tour was the best one during the last generation, let's call this tour S_{gb} for global-best solution. This tour has length: L_{gb}. Now the update rule looks as follows:

$$m(i,j) = (1-\alpha)\cdot m(i,j) + \alpha \cdot \Delta m(i,j)$$

$$\text{where } \Delta m(i,j) = \begin{cases} (L_{gb})^{-1} & \text{if edge (i,j)} \in S_{gb} \\ 0 & \text{else} \end{cases}$$

Here, α is a control parameter similar to the learning-rate. Note that the added pheromone depends on the length of the best tour, and that pheronome on other edges evaporate.

This is one possible ant colony system, it is also possible to let the pheromone be adapted to the best tour ever found, instead of the best tour of the last cycle. Other possibilities of choosing paths are also possible, but the method given above usually works a bit better. Note also that there are many parameters to set: α, β, q_0 and the initial values for the pheromone.

4.5 Discussion

Biological systems differ from mechanical systems or thermodynamic systems since they are able to take energy from the environment in order to decrease their internal entropy (state of disorder). We have seen that there are dynamic systems which look very simple, but which can lead to chaotic dynamics. An example is the logistic map and its operation depends on the control parameter r. If we increase r we can see that instead of a single stable state, there will arise bifurcations to periodic cycles of higher order, finally leading to chaotic dynamics. Chaotic dynamics leads to unpredictable systems, since if we do not know the exact initial condition of the system, the evolution of the system will create large discrepancies between the predicted and the real observed behavior. Although chaotic systems are unpredictable, they also show some kind of order which is seen from the emergence of manifolds on which all points lie (such as in the Lorenz attractor) or the self-similar structure when one looks at chaotic dynamics from different scales.

In biology, there are often simple organisms which can fulfil complex tasks. We have seen that this intelligent collective behavior can emerge from simple individual rules. An example of this is when ants build ant-cemeteries. Furthermore, this ability of swarm intelligence has also inspired researchers to develop algorithms for solving complex problems. A well-known example of this is the ant colony system which has been fruitfully used to solve combinatorial optimisation problems such as the traveling salesman problem.

Chapter 5

Co-Evolution

Let us first consider the history of the earth. Using the internet-site: "http:///www.solstation.com/life.htm" the following summary can be extracted:

> Our solar system was born about 4.6 billion years ago. In this time protoplanets agglomerated from a circum-Solar disk of dust and gas. Not long after that the protoplanetary Earth was struck by a Mars-sized body to form the Earth and Moon. Geologists have determined that the Earth is about 4.56 billion years old. Initially, the Earth's surface was mostly molten rock that cooled down due to the radiation of heat into space, whereas the atmosphere consisted mostly of water (H_2O), carbon dioxide (CO_2), nitrogen (N_2), and hydrogen (N_2) with only a little bit of oxygen (O_2). Eventually a rocky crust was formed and some areas were covered with water rich with organic compounds. From these organic compounds, self-replicating, carbon-based microbial life developed during the first billion years of Earth's existence. The microbes spread widely in wet habitats and life diversified and adapted to new biotic niches, some on land, but life stayed single-celled. After some time microbes were formed which produced oxygen and these became widespread. Chemical reactions caused the production of ozone (O_3) which protected carbon-based life forms from the Sun's ultraviolet radiation. Although the large concentration of CO_2 caused the Earth to warm-up, the produced O_2 caused a chilling effect and as a result the Earth's surface was frozen for large parts, although some prokaryotic microbial life survived in warm ocean seafloors, near volcanos and other warm regions. Due to a large volcanic activity, the Earth warmed up again, but leading to a different niche which led to heavy evolutionary pressure. About 2.5 billion years ago some microbes developed a nucleus using cellular membranes to contain their DNA (eukaryotes), perhaps through endosymbiosis in which different microbes merged to new life-forms. The first multi-cellular life-forms (e.g. plants) evolved after 2.6 billion years of Earth's existence. This multi-cellularity allowed the plants to grow larger than their microbial ancestors. Between 3.85 and 4.02 billion years after the birth of the solar system, there may have been a cycle between ice climates and acid hothouses, leading to strong selective pressure. After a massive extinction, intense evolutionary pressure may have resulted in a burst of multi-cellular evolution and diversity leading to the first multi-cellular animals. After this Dinosaurs were created and may have become extinct 65 millions years ago by the assistance of a large cometary impact.

> The extinction of the Dinosaurs created ecological conditions which eventually led to the creation of modern Human (Homo sapiens sapiens) which originated only 100,000 years ago.

What we can observe from the history of the Earth is that life adapts itself to the biological niche. If environmental circumstances are good for some organisms they can multiply, but there have been many species which became extinct due to environmental conditions or cometary impacts. The way that evolution works is therefore really governed by environmental selection; there is no optimisation but only adaptation to the environment.

5.1 From Natural Selection to Co-evolution

No biologist doubts that natural evolution has occurred and created the diversity of organisms alive today. For the evolutionary theory there are enough indicative facts such as the existence of DNA, organisms which have been shown to mutate themselves to cope with changing environments, and observed links between different organisms in the phylogenetic tree.

The current debate is more on the question how evolution has come about and which mechanisms play a role in evolutionary processes on a planetary scale. In Darwin's evolutionary theory survival of the fittest plays an eminent role to explain the evolution of organisms. We can explain the birth of this competitive mechanism by looking at a planet which is initially populated by some organisms of a specific type with plenty (though finite) amount of nutricients for them to survive. As long as the initial circumstances are good, the population will grow. However, this growth will always lead to a situation in which there are so many organisms that the resources (space, food) will become limited. If the resources are scarce, not all individuals will be able to get enough food and multiply themselves. In such a situation there will arise a competition for the resources and those organisms which are best able to get food will survive and create offspring. The question is which organisms will survive and reproduce. For this we have to examine their genetic material. The existence of particular genes in an individual will give it an advantage and this allows such genes to be reproduced. Therefore there will be more and more offspring which will consist of these genes in their genetic material. Since the resources will usually not grow very much, the population will not grow anymore and only the genetic material inside individual organisms will change. Finally, it may happen that all organisms of the same population will resemble each other very much, especially if the environmental conditions are the same over the whole planet. However, if there are different biological niches, individuals may have adapted themselves to their local niche, so that individuals of the same population will remain somewhat different. Since mutation keeps on occurring during reproduction, it may happen that many mutations after many generations create a new organism which does not look alike the original one. In this way, multiple organisms can evolve and keep on adapting to their local niche.

Since evolution through natural selection is just a mechanism we can implement it in a computer program. A known example of artificial evolution is the use of genetic algorithms. In genetic algorithms, a fitness function is used to evaluate individuals. Such a fitness function is designed a-priori by the programmer and determines how many children an individual can obtain in a given population. Although these genetic algorithms are very good for solving optimisation problems, they do not exactly look alike natural evolution. The problem is that the fitness function is defined a-priori, whereas in natural evolution there is nobody who determines the fitness function.

In reality the (implicit) fitness of an individual depends on its environment in which other species interact with it. Such a fitness function is therefore non-stationary and changes according to the adaptions of different populations in the environment. Here we speak of **co-evolution**. Co-evolutionary processes can be quite complex, since everything depends on each other. Therefore we have to look at the whole system or environment to study the population dynamics.

5.2 Replicator Dynamics

We have already seen two different models for studying the dynamics of interacting species:

- With differential equations (mathematical rules which specify how the variables change). An example of this are the Lotka-Volterra equations.
- With cellular automata

We can also generalise the Lotka-Volterra equations to multiple organisms, this is done using the model of **Replicator dynamics**. We will first study a model in which the fitness of an organism (phenotype) is fixed and independent of its environment. The replicator equation describes the behavior of a population of organisms which is divided in n phenotypes $E_1, \ldots, E_n$. The relative frequencies of these phenotypes are denoted as $x_1, \ldots, x_n$, and so we obtain a relative frequency vector $\vec{x} = (x_1, x_2, \ldots, x_n)$, where $\sum_i x_i = 1$. The fitness of a phenotype E_i is fixed and is denoted as $f_i(\vec{x})$.

Now we can first compute the average fitness of a population using:

$$\hat{f}(\vec{x}) = \sum_{i=1}^{n} x_i f_i(\vec{x})$$

The change of the frequency of phenotype E_i is related to the difference in fitness of E_i and the average of the population:

$$\frac{\partial x_i}{x_i} = f_i(\vec{x}) - \hat{f}(\vec{x})$$

Now we get the replicator equation with adaption speed α (α can be seen as a time-operator dt after which we recompute the relative frequencies):

$$\Delta x_i = \alpha x_i (f_i(\vec{x}) - \hat{f}(\vec{x}))$$

If the fitness values of the existing phenotypes are different, the replicator equation will also change their relative frequencies. If the environment does not change from outside and the fitness values of phenotypes remain constant, then the phenotype with the largest fitness will overtake the whole population. This assumption is of course unrealistic: the environment and fitness values will change due to the changing frequencies.

Now we will look at a model for co-evolutionary replicator dynamics. Here we make the fitness of a phenotype dependent on other existing phenotypes and the relative frequencies. We do this by computing the fitness value at some time-step as follows:

$$f_i(\vec{x}) = \sum_{j=1}^{n} a_{ij} x_j$$

Here the values a_{ij} make up the fitness value of phenotype E_i in the presence of E_j. We can immediately see that phenotypes can let the fitness of other phenotypes increase or decrease. It can therefore happen that both a_{ij} and a_{ji} are positive and quite large. The result will be that these species co-operate and obtain a higher fitness due to this co-operation. Since we always compute relative frequencies with replicator dynamics, we do not always see this co-operation in the values x_i. However, in reality we may assume that both populations will grow, although one may grow faster than the other.

On the other hand when a_{ij} and a_{ji} are negative and quite large, these species are competitive, and the one with the largest frequency will dominate and can make the other one extinct (dependent on the rest of the environment of course).

Instead of two cooperating or competitive organisms, there can also be whole groups of cooperating organisms which may compete with other groups. In this sense we can clearly see the dependence of an organism of its environment.

5.3 Daisyworld and Gaia

In 1983, James Lovelock presented his Daisyworld model which he presented to explore the relationship between organisms and their environment. Daisyworld is a computer model of an imaginary planet consisting of white and black daisies. Daisies can change their environment, reproduce, grow, and die. There is a global variable: the temperature of the planet which may slowly increase due to the radiation of an imaginary sun.

Now the temperature of the planet has an influence on the growth, reproduction, and death of daisies. White daisies have a favourite temperature in which they grow and reproduce fastest and this temperature is higher than the favourite temperature of black daisies. This has as a consequence that if the temperature of the planet would increase, that the population of white daisies would become bigger than the population of black daisies. If the planet's temperature would not stop increasing, however, the temperature would become too hot for any living organism to survive leading to a planet without life-forms.

Due to the albedo effect of white daisies, however, the solar radiation is reflected which causes the temperature of the planet to decrease when there are enough white daisies. Therefore when the planet is warmed up and there are many white daisies the planet will cool down. If the white daisies would continue to decrease the planet's temperature, the planet would become too cold and all life forms would also become extinct.

However, black daisies absorb the heat of the sun and therefore they increase the temperature of the planet. Therefore, if the planet becomes colder, the number of black daisies would become larger than the number of white daisies (since the black daisies' favourite temperature for growth is lower), and the planet would become warmer again. This again leads to a temperature which is closer to the favourite temperature of the white daisies so that the population of white daisies would grow again and thereby cool down the planet.

Thus, we can see that in Daisyworld the daisies influence the environment, and the environment has an influence of the population growth of the daisies. The daisies are also related, since if there would only be black daisies, the temperature could only increase so that life becomes impossible. By increasing and decreasing the temperature of the planet, the different daisy populations are linked to each other, leading to cooperative co-evolutionary dynamics. Furthermore, since the daisies make the temperature suitable for both to survive, they regulate the temperature, like a thermostat of a heater would regulate the temperature

of a room. Therefore we can see that there is a **self-regulating feedback loop**.

5.3.1 Cellular automaton model for Daisyworld

We can use a cellular automaton as a spatial model for Daisyworld. Each cell can be a black or white daisy or a black or white daisy-seed. Furthermore, each cell has its local temperature. Each cycle we can increase the temperature of all cells with for example one degree (of course we can also decrease the temperature). If the temperature of each cell continues to increase, the temperature would become 100 degrees and all life-forms would die.

The rules of the CA look as follows:

- Black daisies have most probability to survive at a temperature of 40 degrees, and white daisies at 60 degrees. Each 20 degrees away from their favourite temperature, the survival probability decreases with 50%.
- Black daisies increase the temperature of 49 cells around their cell with 3 degrees. White daisies cool down the 49 cells around them with 3 degrees.
- White daisies reproduce 6 seeds in random location of their 25-cell neighbourhood with most probability (40%) at 60 degrees, and black daisies do the same at 40 degrees.
- Daisy seeds have a probability of 10% to die each cycle. White (black) seeds become white (black) daisies with most probability at 60 (40) degrees.

We can see the Cellular Automaton model of Daisyworld in Figure 5.1.

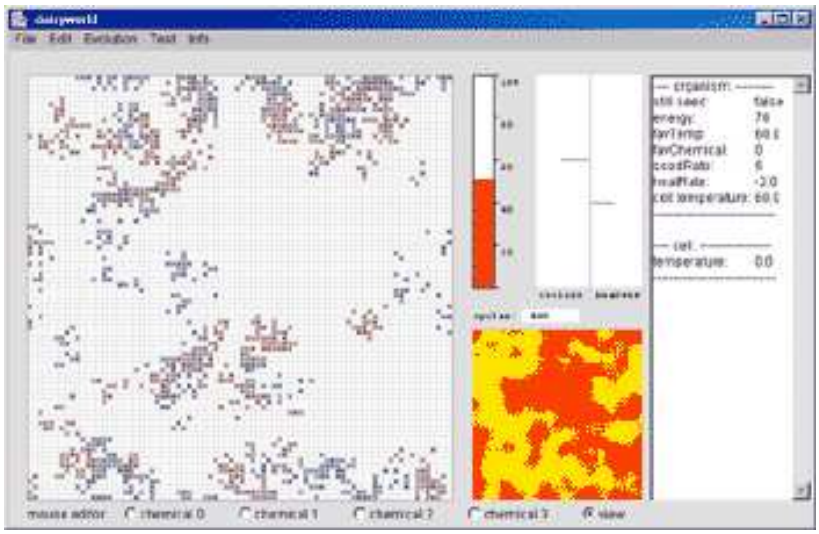

Figure 5.1: A cellular automaton model of Daisyworld. At the right the average temperature of the planet is shown and the temperature in all cells.

Now there are two evolutionary processes in this model: natural selection and self-regulation. Natural selection in Daisyworld takes place becomes there is competition between the different daisy types, since there are limited sources (cells or space to grow). Now let's examine what happens if we use mutation in the model. Mutation is an arbitrary small change

of a genotype of an organism. Such a small change results in a small change of the color which means a difference in the absorbing or reflection of solar energy and therefore different cooling or heating behaviors. In general a mutation can be good for an individual organism, although most mutations are damaging or neutral. However, even if a mutation only gives an advantage one in a million times, once it occurred the new organism may quickly propagate through the environment.

The most interesting aspect of Daisyworld is the self-regulation which appears to be at a higher level than natural selection. This self-regulation is good for all individuals, because it keeps the temperature of a planet at a level which makes life possible. Because this self regulation is good for all individuals, we might think that is is on its own caused by natural selection. However, in Daisyworld self-regulation is not participating in a competitive or reproductive mechanism and therefore is not created by some form of higher level evolutionary process. We can better say that natural selection prefers daisy properties and patterns which lead to self-regulating dynamics.

5.3.2 Gaia hypothesis

In the beginning of the sixties, James Lovelock was working at NASA that wanted to research whether there was life on Mars. Lovelock wondered what kind of tests would be possible to demonstrate the existence of life. Of course it would be possible to check the surface of Mars and to look whether some organisms live there, but it might always be possible that at the place where the spaceship would have landed no life forms existed, whereas life forms might exist at other parts of the planet.

Lovelock thought about examining processes that reduce the entropy of the planet. This can best be explained by looking at a beach. When we sea a sand-castle on the beach, we can see a very ordered object which must be constructed by life forms. On the other hand, if there would not be any life forms on the beach, the surface of the sand on the beach would be completely smooth and not contain any order. But how can this be measured, since not all organisms make sand castles. Lovelock thought about the atmospheric conditions of the planet. If we consider our planet, the Earth, then we can see that the constituents of the atmosphere are very much out of equilibrium. For example, there is much too much oxygen (O_2) and much too little carbon dioxide (CO_2). If we look at Venus, there is 98% carbon dioxide and only a tiny bit oxygen in the atmosphere. On Mars, there is 95% carbon dioxide and 0.13% oxygen. If we compare this to the Earth where there is 0.03% carbon dioxide and 21% oxygen we can see a huge difference. Lovelock explained this difference due to the existence of self-regulatory mechanisms of the biosphere on Earth which he called Gaia. If there would not be any life on Earth, the gases would react with each other and this would lead to an equilibrium similar to that of Mars or Venus. However, since life forms regulate the complete atmosphere it can continuously stay far out of equilibrium and make life possible.

Lovelock predicted that because the planet Mars has an atmosphere which is in a chemical equilibrium, there cannot be any life on Mars, On the other hand, because the atmosphere on Earth is far out of equilibrium there is a complex organising self-regulating force called Gaia which makes life possible. Without this self regulation the amount of carbon dioxide may become much too large and heat up the planet, making life impossible. If one looks at the mechanisms of Gaia, one can see a complex web consisting of bacteria, alges and green plants which play a major role in transforming chemical substances so that life can flourish. In this way Gaia has some kind of metabolism, keeping its temperature constant like a human does.

For example if a human being is very cold, he begins to shake, this causes movements of the body and muscles which makes the body temperature higher. On the other hand if a human being is warm, he will transpirate and thereby lose body heat. These mechanisms therefore keep the temperature of a human more or less constant, and without it (e.g. without feeling cold when it is very cold) people would have died a long time ago.

The name Gaia refers to the Greek goddess Gaea, see Figure 5.2. Since the whole web of organisms creates a self-regulating mechanism, one may speculate that this entire super-organism is alive as well. This led to three forms of the Gaia-hypothesis:

- **Co-evolutionary Gaia** is a weak form of the Gaia hypothesis. It says that life determines the environment through a feedback loop between organisms and the environment which shape the evolution of both.
- **Geophysiological Gaia** is a strong form of the Gaia hypothesis. It says that the Earth itself is a living organism and that life itself optimizes the physical and chemical environment.
- **Homeostatic Gaia** is between these extremes. It says that the interaction between organisms and the environment are dominated by mostly negative feedback loops and some positive feedback loops that stabilize the global environment.

Figure 5.2: The Greek Goddess Gaea, or mother Earth.

There are many examples to demonstrate the homeostatic process of Gaia. Some of these are:

- The amount of oxygen. Lovelock demonstrated that Gaia worked to keep the amount of oxygen high in the atmosphere, but not too high so that a fire would spread too fast and destroy too much.
- Temperature. The average ground temperature per year around the equator has been between 10 and 20 degrees for more than a billion years. The temperature on Mars fluctuates much more and is not suitable for life-forms (-53 degrees is much too cold).
- Carbon-dioxide. The stability of the temperature on the Earth is partially regulated by the amount of carbon dioxide in the atmosphere. The decrease of heat absorption of the Earth in some periods is caused by a smaller amount of carbon dioxide which is regulated by life-forms.

In Figure 5.3 we can see that the temperature of the world has increased during the last century. This may be caused by the large amount of burned fossil fuels during this period, although differences in temperatures are also often caused by the change of the Earth's orbit around the sun. The Gaia hypothesis states that mankind can not destroy life on Earth by e.g. burning all fossil fuels, or using gases which deplete the ozone layer, since the metabolism of the Earth will be much too strong and always some organisms will survive. Even if we would throw all nuclear weapons, we would not destroy all life forms and life will continue albeit without human beings.

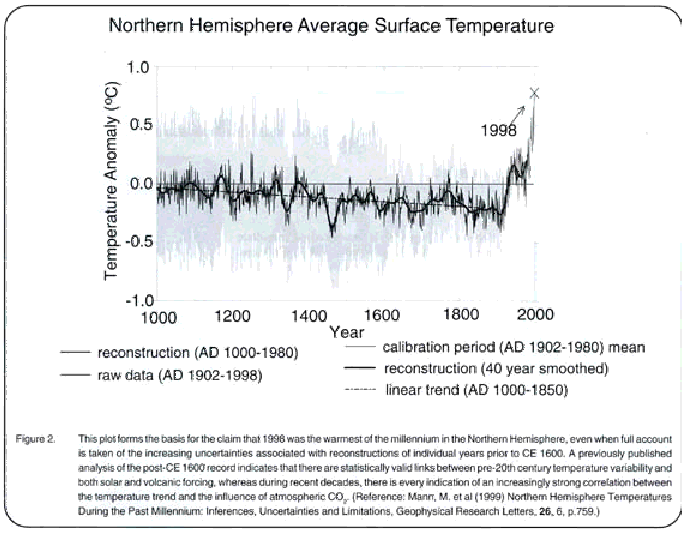

Figure 5.3: The northern hemisphere shows an increasing temperature during the last 100 years.

5.4 Recycling Networks

If there are multiple co-evolving organisms in an environment, they can also interact with the available sources such as chemical compounds. It is possible that these organisms recycle each other's waste so that all compounds remain available for the environment. Of course some of these processes will cost energy which is usually obtained by the sun through e.g. photo-synthesis. Some other processes will create free energy for an organism which it can use to move or to reproduce.

An example of such a process is when we put plants and mammals together in an environment and make a simplified model:

- Plants transform CO_2 into C and O_2 molecules
- Mammals transform C and O_2 into CO_2 molecules
- External chemical reactions transform C and O_2 into CO_2
- Mammals can eat plants and thereby increase their mass with C molecules which they store.

We can implement this model in a cellular automaton consisting of plants, mammals, and molecules. In Figure 5.4 we show the simple model in which we use a layered cellular automaton, one layer of the CA consisting of the positions of molecules and the other layer consisting of plants and mammals. These two layers will interact on a cell by cell basis (for simplicity mammals have the same size as molecules which is of course very unrealistic).

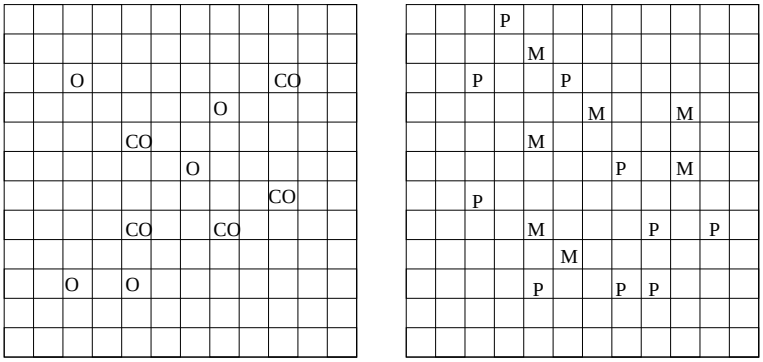

Figure 5.4: A layered cellular automaton for modelling a recycling network.

To make the CA model complete, we also need to model the amount of carbon (C) inside plants and mammals. Therefore, Figure 5.4 does not show us the complete picture, there are internal states of plants and mammals which model the amount of C molecules.

Furthermore, we need to make transition rules to let plants and mammals reproduce and die. Mammals should also have the possibility to navigate on the grid and look for food. We do not model these issues here, although an implementation of these rules would be straightforward. Here we are more interested to examine the feedback loops in the model which will create a recycling network.

If we examine this simple ecology consisting of plants, mammals, and chemical molecules, we can see that the molecules will be recycled under good conditions. If they would not be recycled then the mammals would die since there would not be any O_2 molecules anymore for them. We can see the following dynamics in this model:

- Without plants, all C and O_2 molecules will be transformed to CO_2 molecules. This will lead to a stable chemical equilibrium where no reactions can take place anymore, resulting in the death of all mammals.

- If there are many plants, the number of O_2 molecules would grow, leading to less CO_2 molecules for the plants, possibly also leading to the death of some plants. This will cause the transformation of CO_2 to C and O_2 molecules done by the plants to become much slower, and will give the external reactions and mammals the ability to create more CO_2 molecules, leading to a homeostatic equilibrium. If there would not be any mammals it can be easily seen that there cannot be too many plants, because the speed of the external reactions can be very slow. Therefore, the existence of mammals may be profitable for plants (although the mammals also eat the plants).

- If there are many plants and mammals, they will quickly recycle the molecules. This leads to a situation that even with few molecules, many plants and mammals can survive.

It should be noted that these amounts of plants and mammals depend heavily on each other, but natural processes are likely to create a good situation.

- If there are too many mammals, many plants will be eaten. If this causes few plants to survive, there will not be enough food for all mammals causing many mammals to die. Therefore the growth and decline of the mammal population will not make it easily possible that all plants will be eaten, so that mammals cause their own extinction (this is similar to predator-prey dynamics).

Recycling networks are very important for Gaia and co-evolutionary systems. For example in Gaia many molecules are recycled causing almost optimal conditions for life. One example is the amount of salt in the seas. When this becomes too large, almost all sea life-forms will dry out and die. However, every year a lot of salt is moved from the land to the seas which might easily lead to very large concentrations of salt in the sea. It has been shown by Lovelock that the sea functions as a kind of salt pump keeping the concentration of salt at levels which are advantageous for life forms.

Also in rain-forests the plants and trees cause a very efficient recycling of water molecules. In this way, even with a small amount of H_2O molecules many plants and trees can survive. Furthermore this recycling and co-evolutionary dynamics also causes a profitable temperature for the plants and trees which makes it possible to have rain-forests in hot countries that create their own local redistribution of water.

5.5 Co-evolution for Optimisation

Co-evolutionary processes are not only important for studying population dynamics in ecologies, but can also be used for making optimisation algorithms. We already studied genetic algorithms which can be used for searching for optimal (or near-optimal) solutions for many different problems for which exhaustive search would never work.

There is currently more and more research to use co-evolution to improve the ability of genetic algorithms in finding solutions. The idea relies on evolving a population of individuals to solve some problem which can be described by a large set of tests for which an individual should succeed. If the tests are not clearly specified, they can also be evolved by evolutionary algorithms. An example is to learn to play backgammon. If you want to be sure your individual, which encodes for a backgammon playing program, is very good in backgammon, you want to test it against other programs. When the other programs are not available, you can evolve these programs. The individual which plays best against these test-programs (some call them parasites since they are used to kill individuals by determining their fitness), may reproduce in the learner population. The tests which are good for evaluating learners can also reproduce to create other tests. This is then a co-evolutionary process and makes sense since there is no clear fitness function to specify what a good backgammon player is.

We will now examine a specific problem which requires a solution to be able to solve a specific task such as sorting a series of numbers. In principle there are many instantiations of the sorting problem, since we can vary the numbers, or the amount of numbers, or their initial order, etc. So suppose we take N instantiations of the sorting problem and keep these fixed (like we would do with normal evolutionary computation). Now we can use as a fitness function the amount of instantiations of the sorting task which are solved by an individual. The problem of this is that it can cost a lot of time to evaluate all individuals on all N

tasks if N is large. And if we take the number of instantiations too low, maybe we evolve an individual which can sort these instantiations of the sorting problem, but performs poorly on other sorting problems. Furthermore, it is possible that the best individuals always are able to sort the same $0.7N$ problems and never the others. In this case there is not a good gradient (search direction) for further evolution.

Co-evolution for optimisation. A solution to these problems is to use co-evolution with learners (the individuals which need to solve the task) and problem-instantiations (the parasites or the tests). There are K tests which can be much smaller than the N tests we needed for a complete evaluation in normal evolution, since these K tests also evolve. There are also L learners which are tested on the tests (can be all tests, but might also be a part of all tests). The fitness of a learner is higher if it scores better on the tests it is evaluated on. This creates improving learners, but how can we evolve the test-individuals?

An initial idea would be to make the fitness of a test higher when less learners can solve it (we will later examine the problems of this method for assigning such fitness values to tests). In this way, the learners and tests will co-evolve. The parasites make the tests harder and harder and the individuals have to solve these increasingly difficult tests.

A problem of the above fitness definition of tests is that it becomes possible that only tests remain which cannot be solved by any learner. This leads to all learners having the same fitness and would stop further evolution. Therefore it is much better to let the fitness of a test depend on the way it can differentiate between different learners. In this way when a test is solved by all learners or is not solved by any learner, the test is basically useless at the current stage of evolution and will get a low fitness so that it is not allowed to stay in the population or to reproduce. If two tests make exactly the same distinctions between all learners, it is possible to reduce the fitness of one of them since they would essentially encode the same distinction.

Pareto-front in co-evolutionary GA. If we have a number of learners with their result on all tests, we want to examine which learners are allowed to reproduce themselves. For this we will examine the Pareto-front of individuals which means the set of individuals which are not dominated by any other individual. When a learner passes a number of tests and another learner passes the same number of tests, but also an additional one, it is not hard to see that the second learner performs strictly better than the first one. In this case we say that the first learner is dominated by the second one. We can make this more formal by the following definition, where $f_i(j)$ is the fitness of learner i on test j.

We define:

$$dominates(k,i) = \forall j f_i(j) \leq f_k(j) \wedge \exists l f_i(l) < f_k(l)$$

So dominates(k,i) says that learner i is dominated by learner k. Now we define the Pareto-front as all learners which are not dominated by any other learner. Now we only let the learners in the Pareto-front reproduce and eliminate all other learners which are dominated by some other learner.

This Pareto-front optimisation is also a used and good method for multi-objective optimisation in which there are more criteria to evaluate an individual.

5.6 Conclusion

In this chapter we studied co-evolutionary processes which are important in natural evolution. We have seen that instead of Darwin's survival of the fittest, there can be groups of cooperating organisms which struggle for the same spatial resources, but which may help each other to survive at the same time. We also looked at the methods that life-forms use to alter their environment. There are many mechanisms which keep the environment profitable for life to sustain itself. Lovelock studied this complex web of many coupled processes for the first time and called this entire mechanism of a homeostatic Earth; Gaia. There are many examples of Gaian processes, and in this chapter we only examined a few, but important ones. Gaian homeostasis also relies on recycling networks in which chemical compounds are transformed through a sequence of different organisms so that resources never become depleted. This is very important, since if some compound would get lost, the whole recycling network might starve since their required resources are not available. Finally we have examined how co-evolution can be used in evolutionary computation to make the search for optimal solutions different and for some problems more successful than the search process of normal evolutionary algorithms. Here learners and tests evolve together to allow learners to become better in solving the tests, and the tests to create harder and harder problems while still being able to differentiate between learners.

It is important that we look at the co-evolutionary mechanisms when we use the Earth's resources and kill organisms. Particular organisms may play very important roles to keep the homeostatic equilibrium of the Earth or of a local environmental niche. Since we cannot study a single organism alone, apart from its environment, we again need a holistic approach in which all elements are studied in a total perspective.

Chapter 6

Unsupervised Learning and Self Organising Networks

Unsupervised learning is one of the three forms of machine learning; supervised, unsupervised, and reinforcement learning. The special aspect of unsupervised learning is that there are only input (sensory) signals and no desired outputs or evaluations of actions to specify an optimal behavior. In unsupervised learning it is more important to deal with input signals and to form a meaningful representation of these. E.g. if we look at different objects, we can cluster objects which look similar together. This clustering does not take into account what the label of an object is, and therefore trees and plants may be grouped together in unsupervised learning, whereas in supervised learning we may want to map inputs to the label plant or tree. It is also possible that particular plants form their own cluster (or group) such as cactuses, this grouping is only based on their input representation (e.g. their visual input), and not based on any a-priori specified target concept which we want to learn. We may therefore start to think that unsupervised learning is less important than supervised learning, but this is not true since they have different objectives. Unsupervised learning has an important task in preprocessing the sometimes high-dimensional input and can therefore be used to make the supervised learning task simpler. Furthermore, supervised learning always requires labelled data, but labelled data is much harder obtained than unlabelled data. Therefore unsupervised learning can be applied continuously without the need for a teacher. This is a big advantage, since it makes continual life-long learning possible.

The second topic in this chapter is self-organising networks or often called self-organising maps (SOMs). These SOMs are neural networks and can be applied for unsupervised learning purposes and can be easily extended for supervised and reinforcement learning problems. In principle a SOM consists of a number of neurons that have a position in the input space. Every time a new input arrives, the SOM computes distances between the input and all neurons, and thereby activates those neurons which are closest to the input. This idea of looking at similarities is a very general idea for generalization, since we usually consider objects that look alike (have a small distance according to some distance measure) to be of the same group of objects. To train the SOM, activated neurons are brought closer to the generated inputs, in order to minimize the distance between generated inputs and activated neurons. In this way a representation of the complete set of inputs is formed in which the distance between generated inputs and activated neurons is slowly minimized. By using SOMs in this way, we can construct a lower dimensional representation of a continuous, possibly high-dimensional

input space. E.g. if we consider faces with different orientations as input, the input-space is high-dimensional, but activated neurons in the SOM essentially represent the orientation of the face which is of much smaller dimensionality.

6.1 Unsupervised Learning

In unsupervised learning the program receives at each time-step an input pattern x^p which is not associated to a target concept. Therefore all learned information must be obtained from the input patterns alone. Possible uses of unsupervised learning are:

- Clustering: The input patterns are grouped into clusters where input patterns inside a cluster are similar and input patterns between clusters are dissimilar according to some distance measure.
- Vector quantisation: A continuous input-space is discretized.
- Dimensionality reduction: The input-space is projected to a feature space of lower dimensionality while still containing most information about the input patterns.
- Feature extraction: particular characteristic features are obtained from input patterns.

6.1.1 K-means clustering

One well-known clustering method is called K-means clustering. K-means clustering uses K prototypes which will form K clusters of all input patterns. In principle K-means clustering is a batch learning method, which means that all the data should be collected before and the algorithm is executed one time on all this data. Running the algorithm on this data creates a specific set of clusters. If another input pattern is collected, the algorithm has to be executed again on all data examples which therefore can cost more time than online clustering methods.

K-means clustering is usually executed on input patterns consisting of continuous attributes, although it can be extended on patterns partly consisting of nominal or ordinal attributes.

The K-means algorithm uses K prototype vectors: $w^1, \ldots, w^K$ where each prototype vector is an element of $\Re^N$ where N is the number of attributes describing an input pattern. Each prototype vector w^i represents a cluster C^i which is a set of input patterns which are element of that cluster. So the algorithm partitions the data in the K clusters. We assume that there are n input patterns (examples) denoted as: $x^1, \ldots, x^n$.

The algorithm works as follows:

- Initialize the weight-vectors $w^1, \ldots, w^K$.
- Repeat the following steps until the clusters do not change anymore
  1. Assign all examples $x^1, \ldots, x^n$ to one of the clusters. This is done as follows:
     An example x^i is an element of cluster C^j if the prototype vector w^j is closer to the input pattern x^i than all other prototype vectors:

$$d(w^j, x^i) \leq d(w^l, x^i) \quad \text{For all } l \neq j$$

The distance $d(x, y)$ between two vectors is computed using the Euclidean distance measure:

$$d(x, y) = \sqrt{\sum_i (x_i - y_i)^2}$$

In case the distances to multiple prototype vectors are exactly equal, the example can be assigned to a random one of these.

2. Set the prototype vector to the center of all input patterns in the corresponding cluster. So for each cluster C^j we compute:

$$w_i^j = \frac{\sum_{k \in C^j} x_i^k}{|C^j|}$$

Where $|C|$ denotes the number of elements in the set C.

An example of K-means clustering. Suppose we have four examples consisting of two continuous attributes. The examples are: (1,2); (1,4); (2,3); (3,5).

Now we want to cluster these examples using $K = 2$ clusters. We first initialize these clusters, suppose that $w^1 = (1, 1)$ and $w^2 = (3, 3)$. Now we can see that if we assign the examples to the closest prototypes, we get the following assignment:
$(1, 2) \rightarrow 1$
$(1, 4) \rightarrow 2$
$(2, 3) \rightarrow 2$
$(3, 5) \rightarrow 2$
Now we compute the new prototype vectors and obtain: $w^1 = (1, 2)$ and $w^2 = (2, 4)$. We have to repeat the process to see whether the cluster stay equal after the prototype vectors have changed. If we repeat the assignment process to clusters, we can see that the examples stay in the same clusters, and therefore we can stop (continuing would not change anything).

6.2 Competitive Learning

K-means clustering works on a given collection of data and when the data changes, the algorithm has to be executed again on all examples. There also exist a number of online clustering approaches which are based on artificial neural network models. Competitive learning is one of these methods and partitions the data into specific clusters by iterating an update rule a single time each time a new input pattern arrives. Therefore these online competitive learning algorithms are more suitable for changing environments, since they can change the clusters online according to the changing distributions of input patterns. Again only input patterns x^p are given to the system. The system consists of a particular neural network as a representation of the clustering. The network propagates the input to the top where an output is given which tells us in which cluster an input pattern falls. Like in the K-means algorithm, the number of clusters should be given to the system and usually stays fixed during learning.

In a simple competitive learning network all inputs are connected to all outputs representing the clusters, see Figure 6.1. The inputs describe a specific input pattern and when given these inputs, the competitive learning network can easily compute in which cluster the input falls.

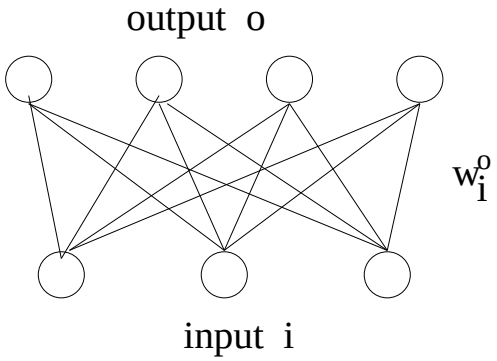

Figure 6.1: In a competitive network all input units are connected to output units through a set of weights.

6.2.1 Normalised competitive learning

There are two versions of the competitive learning algorithm, the normalised and unnormalised versions. We first examine the normalised version which normalises all weight vectors and input vectors to a length of 1. Normalising a vector v means that its norm $||v||$ will be one. The norm of a vector is computed as:

$$||v|| = \sqrt{(v_1^2 + v_2^2 + \ldots + v_N^2} = \sqrt{\sum_{i=1}^{N} v_i^2}$$

Basically the norm of a vector is its Euclidean distance to the origin of the coordinate system. This origin is a vector with only 0's. Normalising a vector is then done by dividing a vector by its norm:

$$x^{norm} = \frac{x}{\|x\|}$$

So if all vectors are normalised, all weight vectors (each output unit has one weight vector which determines how it will be activated by an input pattern) will have length 1, which means that they all fall on a circle when there are 2 dimensions ($N = 2$). Therefore, the weights can only move on the circle.

So, how do we adapt the weight vectors? Just as in the K-means algorithm, we initialize the weight vectors for the chosen number of clusters (represented by as many output units). Then, the normalised competitive learning algorithm performs the following steps after receiving an input pattern:

- Each output unit o computes its activation y^o by the dot- or inner-product:

$$y^o = \sum_i w_i^o x_i = w^o x$$

- Then the output neuron k with the highest activation will be selected as the winning neuron:

$$\forall o \neq k : \qquad y^o \leq y^k$$

- Finally, the weights of the winning neuron k will be updated by the following learning rule:

$$w^k(t+1) = \frac{w^k(t) + \gamma(x(t) - w^k(t))}{\|w^k(t) + \gamma(x(t) - w^k(t))\|}$$

  The divisor in the fraction makes sure that the weight vector remains normalised.

The mechanism of normalised competitive learning causes the winning weight-vector to turn towards the input pattern. This causes weight-vectors to point to regions where there are many inputs, see Figure 6.2.

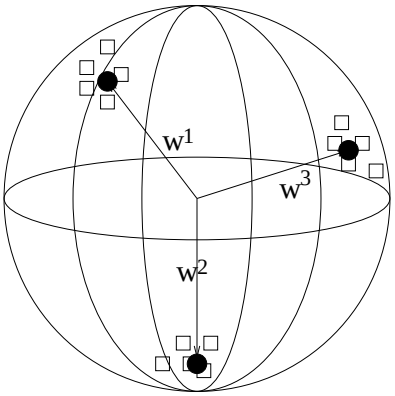

Figure 6.2: In a normalised competitive network, the weight-vectors will start to point to clusters with many inputs.

When we would not use normalised weight vectors, there would be a problem with this algorithm which is illustrated in Figure 6.3. Here it is seen that if weight-vectors are different in size, larger vectors would win against smaller weight vectors, since their dot-product with input vectors is larger, although their (Euclidean) distance to an example is larger.

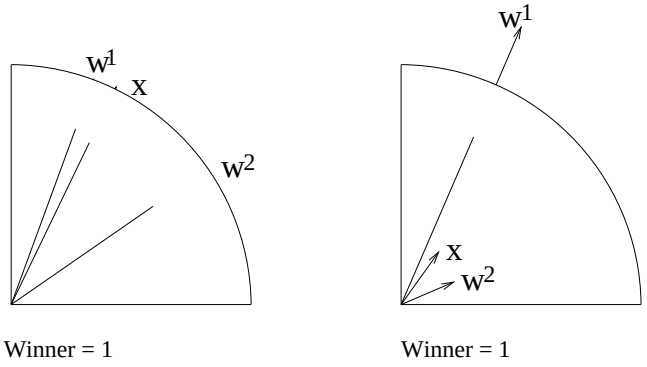

Figure 6.3: (A) With normalised weight vectors the algorithm works appropriate. (B) When weight vectors would not be normalised, we would get undesirable effects, since larger weight vectors would start to win against small weight vectors.

6.2.2 Unnormalised competitive learning

Instead of using the dot-product between two vectors to determine the winner for which we need normalised vectors, we can also use the Euclidean distance to determine the winning neuron. Then we do not need normalised weight vectors anymore, but we will deal with unnormalised ones. So in this case all weight-vectors are again randomly initialised and we determine the winner with the Euclidean distance:

$$\text{Winner} \quad k : \|w^k - x\| \leq \|w^o - x\| \quad \forall o.$$

So here we take the norm of the difference between two vectors, which is the same as taking the Euclidean distance $d(w^k, x)$. The neuron with the smallest distance will win the competition. If all weight-vectors are normalised, this will give us the same results as computing the winner with the dot-product, but if the vectors are not normalised different results will be obtained.

After determining the winning neuron for an input vector, we move that neuron closer to the input vector using the following learning rule:

$$w^k(t+1) = w^k(t) + \gamma(x(t) - w^k(t)) \tag{6.1}$$

where $0 \leq \gamma \leq 1$ is a learning rate which determines how much the neuron will move to the pattern (if $\gamma = 1$ the point will jump to the input vector, and therefore when continuing learning there will be a lot of jumping around. When the learning rate decreases while more updates have been done, a real "average" of the represented input patterns can be learned).

Example unnormalised competitive learning. Suppose we start with $K = 2$ neurons with initialized weight-vectors: $w^1 = (1, 1)$ and $w^2 = (3, 2)$. Now we receive the following four examples:
$x^1 = (1, 2)$
$x^2 = (2, 5)$
$x^3 = (3, 4)$
$x^4 = (2, 3)$
When we set the learning rate γ to 0.5, the following updates will be made:
On $x^1 = (1, 2) \rightarrow d(w^1, x^1) = 1,\ d(w^2, x^1) = 2$. Therefore: Winner $w^1 = (1, 1)$. Application of the update rule gives:
$w^1 = (1, 1) + 0.5((1, 2) - (1, 1)) = (1, 1.5)$.
$x^2 = (2, 5) \rightarrow d(w^1, x^2) = \sqrt{13.25},\ d(w^2, x^2) = \sqrt{10}$. Therefore: Winner $w^2 = (3, 2)$. Application of the update rule gives:
$w^2 = (3, 2) + 0.5((2, 5) - (3, 2)) = (2.5, 3.5)$.
$x^3 = (3, 4) \rightarrow d(w^1, x^3) = \sqrt{10.25},\ d(w^2, x^3) = \sqrt{0.5}$. Therefore: Winner $w^2 = (2.5, 3.5)$ Application of the update rule gives:
$w^2 = (2.5, 3.5) + 0.5((3, 4) - (2.5, 3.5)) = (2.75, 3.75)$.
Now try it yourself on the fourth example.

Initialisation

A problem of the recursive (online) clustering methods which also holds for the K-means clustering algorithm is a possible wrong initialisation of the weight vectors of the neurons. Therefore it can happen that some neuron never becomes a winner and therefore never learns. In that case we are basically dealing with a dead (or silent) neuron and have one cluster less in our algorithm. To deal with this problem, there are two methods:

- Initialise a neuron on some input pattern
- Use "leaky learning". For this we let all neurons adapt on all examples, although we use a very small learning rate for this adaption so that this will only make a difference in the long run. The leaky learning rule adapts all neurons (except for the winning neuron) to the current example with a very small learning rate $\gamma' << \gamma$:

$$w^l(t+1) = w^l(t) + \gamma'(x(t) - w^l(t)), \forall l \neq k$$

Minimising the cost function

The goal of a clustering method is to obtain a clustering in which the similarities between inputs of the same cluster are much larger than similarities between inputs of different clusters. The similarity between two inputs can be computed using the inverse of the (Euclidean) distance between the two inputs. Therefore if we minimize the distances between a neuron and all the examples in the cluster, we will maximize the similarities between the inputs in a cluster.

A common measure to compute the quality of a final obtained set of clusters on a number of input patterns is to use the following quadratic cost function E:

$$E = \frac{1}{2}\sum_p \|w^k - x^p\|^2 = \frac{1}{2}\sum_p \sum_i (w_i^k - x_i^p)^2$$

In which k is the winning neuron on input pattern x^p.

Now we can prove that competitive learning searches for the minimum of this cost function by following the negative gradient of this cost function.

Proof that the cost function is minimized. The cost-function for pattern x^p:

$$E^p = \frac{1}{2}\sum_i (w_i^k - x_i^p)^2$$

in which k is the winning neuron is minimized by Equation 6.1.

We first examine how the weight-vectors should be adjusted to minimize the cost-function E^p on pattern x^p:

$$\Delta_p w_i^o = -\gamma \frac{\partial E^p}{\partial w_i^o}$$

Now we have as the partial derivative of E^p to the weight-vectors:

$$\begin{aligned} \frac{\partial E^p}{\partial w_i^o} &= w_i^o - x_i^p, \quad \text{If unit } o \text{ wins} \\ &= 0, \quad \text{else} \end{aligned} \tag{6.2}$$

From this follows (for winner o):

$$\Delta_p w_i^o = \gamma(x_i^p - w_i^o)$$

Thus we demonstrated that the cost-function is minimized by repetitive weight-vector updates. Some notes on this are:

- If we continue the updating process with a fixed learning rate, the weight-vectors will always make some update step, and therefore we do not obtain a stable clustering. To obtain a stable clustering we should decrease the learning-rate γ after each update according to the conditions of stochastic approximation: (1) $\sum_{t=1}^{\infty} \gamma_t = \infty$ and (2) $\sum_{t=1}^{\infty} \gamma_t^2 < \infty$. The first condition makes sure that the weight-vectors are able to move an arbitrarily long distance to their final cluster-point, and the second condition makes sure that the variance of updates goes to zero which means that finally a stable state will be obtained. A possible way of setting the learning rate which respect these conditions is: $\gamma_t = \frac{1}{t}$.

- It is important to note that the cost-function is likely to contain local minima. Therefore the algorithm does not always obtain the global minimum of the cost-function. Although the algorithm will converge (given the conditions on the learning-rate), convergence to a global minimum is not guaranteed. Better results can therefore be obtained if we execute the algorithm multiple times starting with different initial weight-vectors.

- Choosing the number of cluster-points (or neurons) is an art and not a science. Of course the minimum of the cost-function can be obtained if we use as many cluster-point as input-patterns and set all the cluster-points on a different input-pattern. This would result in a cost of 0. However, using as many cluster-points as input-patterns does not make any sense since we want to obtain an abstraction of the input data. It is also logical that increasing K leads to a smaller minimal cost, so how should we then choose K? Often we need to trade off the complexity of the clustering (the number of used cluster-points) and the obtained error-function. Thus, we like to minimize a new cost-function:

  $$E_f = E + \lambda K$$

  where the user-defined parameter λ trades off complexity versus clustering cost. E_f can then be minimized by running the algorithm with different K.

6.2.3 Vector quantisation

Another important use of competitive learning is vector quantisation. In vector quantisation we divide the whole input space into a number of non-overlapping subspaces. The difference with clustering is that we are not so much interested in the clusters of similar input-patterns, but more in the quantisation of the whole input space. Vector quantisation uses the same (unnormalised) competitive learning algorithm as described before, but we will finally examine the subspaces and not the clusters. It should be noted that the distribution of input-patterns is respected by competitive learning; more inputs in a region lead to more cluster-points. An example of an obtained vector quantisation is shown in Figure 6.4.

Vector quantisation combined with supervised learning

Vector quantisation can also be used in a preprocessing phase for supervised learning purposes. In this case, each neuron corresponds to some output value which is the average of the output values for all input-patterns for which this neuron wins the competition. The output-values for multiple outputs belonging to some neuron that represents a subspace of the input space is usually stored in the weights from this neuron to the output neurons. Thus we can denote the value for output o which is computed when neuron h is activated as w_o^h. If there is only one

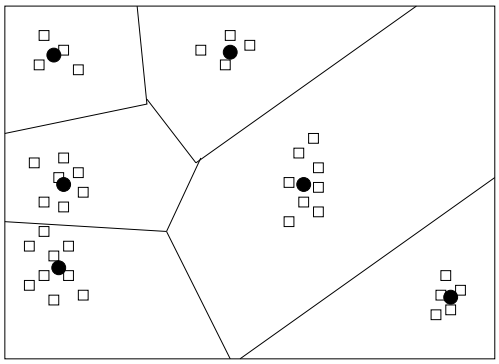

Figure 6.4: A final set of clusters (the big black dots) corresponds with a quantisation of the input space into subspaces.

output, we sometimes write y^h to indicate that this value is the output of neuron h. Figure 6.5 shows a supervised vector quantisation network in which vector quantisation in the first layer is combined with supervised learning in the second layer.

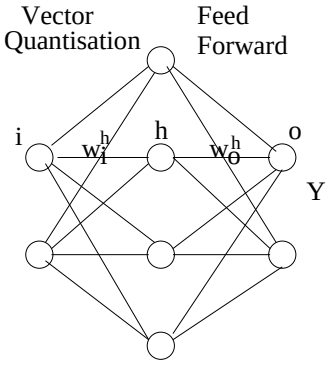

Figure 6.5: A supervised vector quantisation network. First the input is mapped by a competitive network to a single activated internal neuron. Then this neuron is used for determining the output of the architecture.

For learning this network we can first perform the (unsupervised) vector quantisation steps with the unnormalised vector quantisation algorithm and then perform the supervised learning steps, but is is also possible to perform these two updates at the same time. The supervised learning step can simply be done with a simple version of the delta-rule. The complete algorithm for supervised vector quantisation looks as follows:

- Present the network with input x and target value $D = f(x)$
- Apply the unsupervised quantisation step: determine the distance of x to the (input) weight-vector of each neuron and determine the winner k, then update the (input) weight-vector of neuron k with the unsupervised competitive learning rule (Eq. 6.1).
- Apply the supervised approximation step, for all outputs o do:

$$w_o^k(t+1) = w_o^k(t) + \alpha(D_o - w_o^k(t))$$

This is a simple version of the delta-rule where α is the learning-rate and k the winning neuron.

This algorithm can work well for smooth functions, but may have problems with fluctuating functions. The reason is that inside a subspace in which a single neuron is activated, the generated network output is always the same. This means that large fluctuations will cause problems and can only be approximated well when enough neurons are used. For smooth functions, however, the target values inside a subspace are quite similar so that the approximation can be quite good. Given a vector quantisation and input-patterns with their target outputs, we can compute to which values w_o^k the network converges. First we define a function $g(x,k)$ as:

$$\begin{aligned} g(x,k) &= 1, \qquad \text{If } k \text{ is the winner} \\ &= 0, \qquad \text{Else} \end{aligned}$$

Now it can be shown that the supervised vector quantisation learning rule converges to:

$$w_o^h = \frac{\int_{\Re^n} D_o(x) g(x,h) p(x) dx}{\int_{\Re^n} g(x,h) p(x) dx}$$

where $D_o(x)$ is the desired output value of output o on input-pattern x and $p(x)$ is a probability density function which models the probabilities of receiving different inputs. Thus, each weight from neuron h to output o converges to the average target output value for output o for all the cases that neuron h wins.

Example of supervised vector quantisation. The winning neuron moves according to the same update rule as normalised competitive learning. Since there is only a single output in the example below, we will write y^k to denote the output value of neuron k. The value y^k for the winning neuron w^k is adapted after each example by the following update rule:

$$y^k = y^k + \alpha(D^p - y^k)$$

Suppose we start again with 2 cluster-points and set their output-values to 0 :
$w^1 = (1,1), y^1 = 0$ and $w^2 = (3,2), y^2 = 0$.
Now we receive the following learning examples:
$(x^1 \to D^1) = (1, 2 \to 3)$
$(x^2 \to D^2) = (2, 5 \to 7)$
$(x^3 \to D^3) = (3, 4 \to 7)$
$(x^4 \to D^4) = (2, 3 \to 5)$

Suppose we set the learning-rate γ to 0.5 and the learning rate for the supervised learning step $\alpha = 0.5$. Now if we update on the four learning examples, the following updates are made:
$x^1 = (1,2) \to d(w^1, x^1) = 1,\ d(w^2, x^1) = 2$. Thus: Winner $w^1 = (1,1)$. Application of the update rule gives:
$w^1 = (1,1) + 0.5((1,2) - (1,1)) = (1, 1.5)$.
This is just the same as in the example of unnormalised competitive learning before.
The only difference in computations is that we also adjust the output values of the winning neuron:
$y^1 = 0 + 0.5(3 - 0) = 1.5$

Since the weight-vectors w^i are adjusted in the same way as in the example of competitive learning, we only show the updates of the neurons' output values:
$x^2 = (2, 5)$. Winner is neuron 2.
$y^2 = 0 + 0.5(7 - 0) = 3.5$.
$x^3 = (3, 4)$. Winner is neuron 2.
$y^2 = 3.5 + 0.5(7 - 3.5) = 5.25$.

Now try it yourself on the fourth example.

6.3 Learning Vector Quantisation (LVQ)

Learning vector quantisation is basically a supervised learning algorithm, since the neurons have labels associated to them and therefore can classify inputs into a fixed number of categories. Using the training examples, which in this case consist of an input pattern and an associated discrete label (or output), LVQ learns decision boundaries which partition the input space into subspaces with an associated label. The goal is that each input patterns falls into a subspace with the same associated label.

The algorithm looks as follows:

- Initialize the weight-vectors of a number of neurons and label each neuron o with a discrete class label y^o
- Present a training example (x^p, d^p)
- Use the distance measure between the weight-vectors of the neurons and the input vector x^p to compute the winning neuron k_1 and the second closest neuron k_2:

$$\|x^p - w^{k_1}\| < \|x^p - w^{k_2}\| < \|x^p - w^i\| \; \forall i \neq k_1, k_2$$

- The labels y^{k_1} and y^{k_2} are compared to the desired label of the example d^p from which an update is computed

The update rule causes the winning neuron to move closer to the input example when its label corresponds to the desired label for that example. In case the labels are not the same, the algorithm looks at the second-best neuron and when its label is correct it is moved closer and in this case the winning neuron is moved away from the input example. Formally, the update rules look as follows:

- If $y^{k_1} = d^p$: Apply the weight update rule for k_1:

$$w^{k_1}(t+1) = w^{k_1}(t) + \gamma(x^p - w^{k_1}(t))$$

- Else, if $y^{k_1} \neq d^p$ and $y^{k_2} = d^p$: Apply the weight update rule for k_2:

$$w^{k_2}(t+1) = w^{k_2}(t) + \gamma(x^p - w^{k_2}(t))$$

  and move the winning neuron away from the example:

$$w^{k_1}(t+1) = w^{k_1}(t) - \gamma(x^p - w^{k_1}(t))$$

The algorithm does not perform any update if the labels of the winning and second-best neurons do not agree with the label of the example. One could make an algorithm which would move the closest neuron with the correct label to the example (and possibly move all others away from it), but this is not done in LVQ. A possible problem of this would be strong oscillation of the weight-vectors of the neurons due to noise.

LVQ example. In LVQ, we use K cluster-points (neurons) with a labelled output. We compute the closest (winning) neuron w^{k_1} and the second closest neuron w^{k_2} for each training example and apply the weight update rules.

Suppose we start with 2 cluster-points: $w^1 = (1, 1)$ with label $y^1 = A$, and $w^2 = (3, 2)$ with label $y^2 = B$. We set the learning rate γ to 0.5.

Now we receive the following training examples:
$(x^1 \rightarrow D^1) = (1, 2 \rightarrow A)$
$(x^2 \rightarrow D^2) = (2, 5 \rightarrow B)$
$(x^3 \rightarrow D^3) = (3, 4 \rightarrow A)$
$(x^4 \rightarrow D^4) = (2, 3 \rightarrow B)$

Then we get the following update rules: For $(1, 2 \rightarrow A)$, the winner is neuron 1 and the second best is neuron 2. The label of neuron 1 $y^1 = D^1$. Therefore neuron 1 is moved closer to the example:
$w^1 = (1, 1) + 0.5((1, 2) - (1, 1)) = (1, 1.5)$.

$x^2 = (2, 5)$. Winner is neuron 2. Second closest is neuron 1. The label of neuron 2 is the same as the label D^2, therefore neuron 2 is moved closer to the example:
$w^2 = (3, 2) + 0.5((2, 5) - (3, 2)) = (2.5, 3.5)$.

$x^3 = (3, 4)$. Winner is neuron 2. Second closest is neuron 1. The label of neuron 2 is not the same as the label D^3. The label of neuron 1 is the same as D^3. Therefore we move neuron 1 closer to the example, and neuron 2 away from the example:
$w^1 = (1, 1.5) + 0.5((3, 4) - (1, 1.5)) = (2, 2.75)$.
$w^2 = (2.5, 3.5) - 0.5((3, 4) - (2.5, 3.5)) = (2.25, 3.25)$.

Now try it yourself on the fourth example. An example partitioning of a 2-dimensional input space is shown in Figure 6.6. The structure of the decision boundaries of such a partitioning is often called a Voronoi diagram.

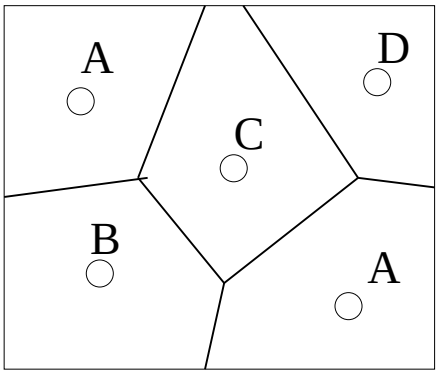

Figure 6.6: An example of a partitioning created by LVQ.

6.4 Kohonen Networks

Kohonen networks or Kohonen maps are self-organising maps (SOMs) in which the neurons are ordered in a specific structure such as a 2-dimensional grid. This ordering or structure determines which neurons are neighbours. Input patterns which are lying close together are mapped to neurons in the structure S which are close together (the same neuron or neighbouring neurons). The learning algorithm causes the structure of the neurons to get a specific shape which reflects the underlying (low dimensional) manifold of the input patterns received by the algorithm. The structure of a Kohonen network is determined before the learning process, and often a structure is used which has lower dimensionality than the dimensionality of the input space. This is very useful to visualise the structure of inputs which fall on a subspace of the input space, see Figure 6.7. The structure used here is a 2-dimensional structure consisting of 4×4 neurons.

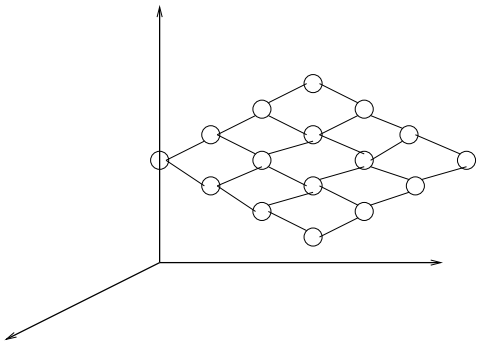

Figure 6.7: In this example, the 2-dimensional 4×4 structure of the Kohonen network covers a manifold of lower dimensionality than the input space.

6.4.1 Kohonen network learning algorithm

Again we compute the winning neuron for an incoming input pattern using some distance measure such as the Euclidean distance. Instead of only updating the winning neuron, we also update the neighbours of the winning neuron for which we use a neighbourhood function $g(o, k)$ between two neurons. Here we define $g(k, k) = 1$ and with a longer separation distance in the structure we decrease the value of the neighbourhood function $g(o, k)$. So the update is done using:

$$w^o(t+1) = w^o(t) + \gamma g(o,k)(x(t) - w^o(t)) \quad \forall o \in S.$$

Where k is the winning neuron and we have to define a function $g(o, k)$. For example we can use a Gaussian function defined as:

$$g(o,k) = exp(-distance_S(o,k))$$

Where $distance_S(o, k)$ computes the distance in the structure S between two neurons. This distance is the minimal number of edges which have to be traversed in the structure to arrive at neuron o from winning neuron k.

By this collective learning method input patterns which lie close together are mapped to neurons which are close together in the structure. In this way the topology which can be

found in the input signals is represented in the learned Kohonen network. Figure 6.8 shows an example of the learning process in which input patterns are drawn randomly from the 2-dimensional subspace.

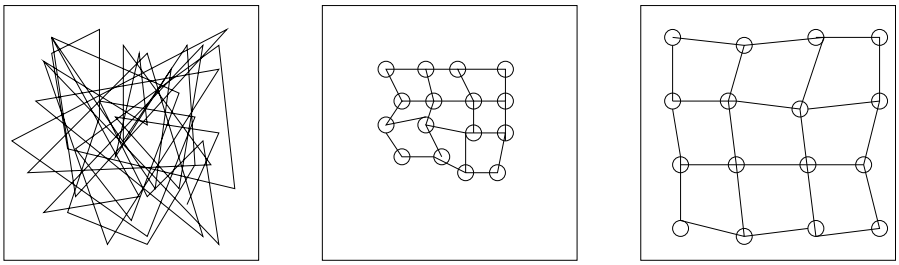

Figure 6.8: The Kohonen network learns a representation which preserves the structure of the input patterns.

If the intrinsic dimensionality of the structure S is smaller than the dimensionality of the input space, the neurons of the network are "folded" in the input space. This can be seen in Figure 6.9.

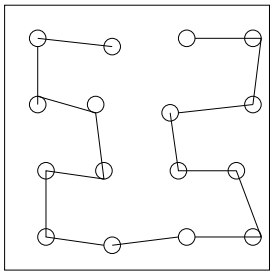

Figure 6.9: If the dimensionality of the structure is smaller than the manifold from which input patterns are generated, the resulting Kohonen map is folded in the input space. Here this folding is shown for a 1-dimensional structure in a 2-dimensional input-space.

Example Kohonen network. Suppose we use a Kohonen network with 3 neurons connected in a line (thus 1-dimensional) structure. We use a neighbourhood relation as follows: $g(k,k) = 1$ and $g(h,k) = 0.5$ if h and k are direct neighbours on the line, else $g(h,k) = 0$.

Again we always compute the winning neuron on each input pattern, and then we update all neurons as follows:

$$w^i = w^i + \gamma g(i,k)(x^p - w^i)$$

We initialise: $w^1 = (1,1)$, $w^2 = (3,2)$, $w^3 = (2,4)$. We set $\gamma = 0.5$. Now we obtain the examples:
$x^1 = (1,2)$
$x^2 = (2,5)$
$x^3 = (3,4)$
$x^4 = (2,3)$

On $x^1 = (1, 2)$ neuron 1 wins the competition. This results in the update:
$w^1 = (1, 1) + 0.5 * 1((1, 2) - (1, 1)) = (1, 1.5)$.
We also have to update the neighbours. $g(2, 1) = 0.5$ en $g(3, 1) = 0$. So we update neuron 2:
$w^2 = (3, 2) + 0.5 * 0.5((1, 2) - (3, 2)) = (2.5, 2)$.
On $x^2 = (2, 5)$ neuron 3 wins. This results in the update:
$w^3 = (2, 4) + 0.5 * 1((2, 5) - (2, 4)) = (2, 4.5)$.
We also have to update the neighbours. $g(2, 3) = 0.5$ en $g(1, 3) = 0$. So we update neuron 2:
$w^2 = (2.5, 2) + 0.5 * 0.5((2, 5) - (2.5, 2)) = (2.375, 2.75)$.
On $x^3 = (3, 4)$ neuron 3 wins. This results in the update:
$w^3 = (2, 4.5) + 0.5 * 1((3, 4) - (2, 4.5)) = (2.5, 4.25)$.
We also have to update the neighbours. Again $g(2, 3) = 0.5$ en $g(1, 3) = 0$. So we update neuron 2:
$w^2 = (2.375, 2.75) + 0.5 * 0.5((3, 4) - (2.375, 2.75)) = (2.53, 3.06)$.

Try it yourself on the last example.

6.4.2 Supervised learning in Kohonen networks

A Kohonen network can also be used for supervised learning. For this we use outputs w_o^h for each neuron h and each output o. In case there is only a single output we can denote the output of a neuron h as y^h. To determine the overall output on a training example, we use the outputs of all activated neurons (neurons are activated if $g(h, k) > 0$. So we obtain the output y_o by the following formula which weighs the neuron outputs by their activations:

$$y_o = \frac{\sum_{h \in S} g(h, k) w_o^h}{\sum_{h \in S} g(h, k)}$$

This is basically a weighted sum and causes smoother functions when larger neighbourhood function values are used.

Now each neuron can learn output values in two different ways. The first possibility is to let neurons learn the average output weighted by its activation using:

$$w_o^h = w_o^h + \alpha(D_o - w_o^h) \frac{g(h, k)}{\sum_{i \in S} g(i, k)}$$

Where D_o is the target value for output o.

We can also let each neuron learn to reduce the overall error of the network. In this case neurons collaborate more. The following learning rule does this:

$$w_o^h = w_o^h + \alpha(D_o - y_o) \frac{g(h, k)}{\sum_{i \in S} g(i, k)}$$

Furthermore for supervised learning in Kohonen networks, the unsupervised steps can be changed so that neurons with small errors are moved faster to the input pattern than neurons with larger errors.

6.5 Discussion

In this chapter we examined unsupervised learning methods which can be used for clustering data, vector quantisation, dimensionality reduction, and feature extraction. The K-means

algorithm is a well-known method for clustering, but is a batch learning method meaning that it has to be executed on all input patterns. In competitive learning, updates are made online. The neurons compete for becoming activated based on their distance to the input pattern. Unsupervised learning methods can also be extended with additional output weights to make supervised learning possible. In this case we can simply use the delta rule for learning outputs of each neuron. The shown algorithms are well able in dealing with continuous inputs, for discrete inputs some adaptions may be necessary to improve the algorithms. All learning algorithms respect the locality principle; inputs which lie close together in the input space are grouped together. For supervised learning, the shown algorithms can be very suitable if the function is smooth. By using additional neurons a good approximation of a fluctuating target function can be learned, but finding the winning neuron becomes slow if many neurons are used.

Bibliography

[Dawkins, 1976] Dawkins, R. (1976). *The Selfish Gene*. Oxford University Press.

[Dorigo and Gambardella, 1997] Dorigo, M. and Gambardella, L. M. (1997). Ant colony system: A cooperative learning approach to the traveling salesman problem. *Evolutionary Computation*, 1(1):53–66.

[Dorigo et al., 1996] Dorigo, M., Maniezzo, V., and Colorni, A. (1996). The ant system: Optimization by a colony of cooperating agents. *IEEE Transactions on Systems, Man, and Cybernetics-Part B*, 26(1):29–41.

[Glover and Laguna, 1997] Glover, F. and Laguna, M. (1997). *Tabu Search*. Kluwer Academic Publishers.

[Merz and Freisleben, 1999] Merz, P. and Freisleben, B. (1999). A comparison of memetic algorithms, tabu search, and ant colonies for the quadratic assignment problem. In et al., P. J. A., editor, *Proceedings of the Congress on Evolutionary Computation*, volume 3, pages 2063–2070.

[Radcliffe and Surry, 1994] Radcliffe, N. J. and Surry, P. D. (1994). Formal memetic algorithms. In *Evolutionary Computing, AISB Workshop*, pages 1–16.