TRẦN HOÀI LINH
TRƯƠNG TUẤN ANH

ARTIFICIAL INTELLIGENCE

NHÀ XUẤT BẢN
ĐẠI HỌC THÁI NGUYÊN

TRẦN HOÀI LINH, TRƯƠNG TUẤN ANH

ARTIFICIAL INTELLIGENCE

NHÀ XUẤT BẢN ĐẠI HỌC THÁI NGUYÊN
NĂM 2017

MÃ SỐ: $\frac{01 - 1165}{\text{ĐHTN - 2017}}$

Preface

The book forms an introduction to Artificial Intelligence for the students of Computer Science and Electrical Engineering. Undergraduate students are required to have at least the chapters 1, 2, 3, 5 and 6. Graduate students may have additional minor projects to support the learning materials.

AI development mainly started in XX century. The history of AI includes up and down sides. It's filled with various approaches: from purely theoretical ideas and thoughts to pattern recognition overlearning tools. In the last decade, there has been more and more practical applications of AI in reality.

Artificial intelligence is still far reaching the human level. It's been more than 60 years since John McCarthy, the father of artificial intelligence, used this term for the first time at a Dartmouth College conference.

The most important processes and functions that make up human intelligence

- Learning and using knowledge, generalizability, perception and cognitive abilities, such as the ability to recognize a given object in any context.
- Memorization, goal setting and achievement, collaboration, formulation, analysis, conceptual and abstract thinking.

The ability to learn is widely regarded as one of the most important manifestations of intelligence. Data is the driving force of artificial intelligence. The more data the machine has about the problem, the better it can anticipate and learn. A good example of AI excellent achievement is the automatic learning of games playing

Despite the enormous interest in AI issues and the involvement of research funding, there is still a number of fields, which have not been achieved so far, despite many efforts, such as a program that would effectively imitate human conversation, or a program that effectively translates literary texts and informal speech, etc

We can be sure that technological innovation will be faster than ever. Artificial Intelligence will become something invisible, yet present in almost every aspect of our lives. In few years, we will be wondering how we could manage without our intelligent digital assistants, just as we can hardly imagine nowadays that we could look into the mighty computer hidden in our pockets.

This book covers a selected range of AI techniques. The chapters content are summarized as follow:

- Chapter 1 (Introduction to AI): a short introduction to the field of artificial intelligence and its applications in various problems. The main tools and building blocks of AI are listed out for further discussions in following chapters.
- Chapter 2 (Search Algorithms and AI): The search algorithms can be divided into two groups: uninformed and informed algorithms. As the data structure for storing the states space, the basic types of array, lists, tree and graph are discussed. The uninformed search algorithms basing on the state spaces include depth-first, breadth-first, uniform-cost-search, and others. Informed search is an evolution of search that applies heuristics to the search algorithm, given the problem space, to make the algorithm more efficient. This chapter covers best-first, a star algorithms.
- Chapter 3 (PROLOG – A logical programming language): there have been a large number of languages developed specifically for AI application development. In this chapter, PROLOG programming language is briefly presented with simple examples for demonstration.
- Chapter 4 (LISP – A functional programming language): Anothe approach to AI language programming, which bases on the lists and their processing is LISP. This chapter presents a short introduction t LISP.
- Chapter 5 (AI and Games): Search algorithms are used as one of basi tools in AI in general and in games in particular. Games like Chess we

believed to be an intelligent activity. A variety of games and games' intelligent details are then discussed.

- Chapter 6 (Neural Networks): Neural networks are one of the most useful techniques in AI. This chapter introduces the basics of neural networks including learning algorithms, which may be mainly divided into supervised and unsupervised ones. Learning algorithms are adaptive methods that change/update the parameters of the neural networks according to a cost functions defined over a given data set.
- Chapter 7 (Evolutionary Computation): Evolutionary computation introduced the idea of imitating the concepts of life and intelligence using biological metaphors. This chapter mentions 4 of basic methods including genetic algorithms, genetic programming, evolutionary programming and evolutionary strategies.
- Chapter 8 (Deep learning): Deep learning is a branch of artificial intelligence based on the natural neural network in the brain. Rapid advances in this area will soon make it possible for the digital assistant to process millions of unstructured signals from social media or other data sources.

The chapters 3, 4, 6 and 8 are written by Tran Hoai Linh, the chapters 1, 2, 5 and 7 are written by Truong Tuan Anh.

In this 1st edition of the book, despite our best efforts, there are certainly many points that can be improved or important materials that need to be added to the textbook. The authors are looking forward to receiving the comments of readers. Please send your comments to: thlinh2000@gmail.com and ttanhhtd@gmail.com.

Sincerely thank you!

Chapter 1: Introduction to Artificial Intelligence

1.1. WHAT IS INTELLIGENCE AND AI?

Artificial Intelligence (AI) is a field of science that deals with problem solving that is effectively non-algorithmic based on knowledge modeling.

- Intelligence is the ability to adapt to new tasks and living conditions or to the way in which people process information and solve problems.
- Intelligence is also the ability to associate and understand. The influence on it has both hereditary and educational characteristics.

To start with, we will use the definition from Wikipedia: "*Artificial intelligence is intelligence exhibited by machines, rather than humans or other animals (natural intelligence)*" [WikiAI]. Some of other definitions from other researchers can be listed here for reference:

- AI is a science of machines that perform tasks that require intelligence when they are performed by a human being [Minsky91],
- AI is the field of computer science of computer-aided symbolic methods and techniques, and the symbolic representation of knowledge used in such inference [Barr81],
- AI involves solving problems modeled on the natural activities and cognitive processes of humans using computer simulations [Schalkoff90],
- Artificial Intelligence is the automation of the abilities attributed to human thinking, abilities such as decision making, problem solving learning... [Bellman78],
- Artificial Intelligence is a study designed to create computers with a skill in which a person is better now [Rich09],
- Artificial Intelligence is a study of mental abilities through computational models [Charniak85],

- Artificial Intelligence is a study of computational models that allow perception, reasoning and action [Winston92],
- Artificial Intelligence is a branch of computer science that deals with the automation of intelligent behavior [Luger04].

Artificial intelligence also be defined as an IT field dealing with problem solving that is not effectively algorithmic. Artificial Intelligence is the human face of machine learning. Machine learning was supposed to be a method for analyzing data without first defining a problem or areas of exploration. Proposed methods are not fully universal and require a lot of preparatory work. They are not universal in the sense that they fit only part of the problem-solving classes. Or they are methods of grouping, decision support, or optimization, or... The truly autonomous machine learning method would operate on bare data and itself "decide" what methods to use to extract "knowledge".

To create an "intelligent" system, we should begin with a definition of intelligence. And this is already a big problem since the definition strongly depends on the point of view of the author. Intelligence can be simply defined as a set of properties of the mind. These properties include the ability to analyze and solve problems, and in general, to make decision based on (limited) input information and a set of possible actions.

Some of the definitions of intelligence can be applied not only to humans, but also to living creatures that show rational behavior. For example the ability of communicate, the ability of route finding,... But the intelligence that is performed by human beings is much more complex than that of animals (i.e. including the ability to communicate, solve problems, learn and adapt,...) where animals typically have a small number of intelligent characteristics and at a simpler complexity than humans.

The situation is similar for computer programs. Although we do have some very advanced programs for chess, but they cannot do anything with other games, even the simple one as Tic-Tac-Toe.

The definition of AI does not explain very much, but it clearly points to the extent and the blurring of the term. We will assume the importance attributed to this term in the hearts of computer scientists and mathematicians. The most important goal of research in this field is to enable the design of machines and computer programs capable of performing selected functions of the mind and senses that have not yet been addressed in a simple numerical way.

While machine learning can be an element of AI, artificial intelligence itself is more than that. Definitions are many, but in visual terms, let's assume that artificial intelligence starts where standard numerical algorithms end.

How far are we to create artificial intelligence? The question itself is probably more difficult than the answer, because again the problem of definition returns to us. IT professionals laugh that artificial intelligence will rise "in the next 10 years". Because such expression is repeated since the 1960s. Scientists and computer scientists have effectively engineered and created machines that, in a non-trivial way, can manipulate data by extracting information and knowledge that could have escaped humans. We also create neural networks that perfectly simulate human recognition of scripture or speech. Even smartphones have built-in OCRs or virtual assistants, with which they can "talk" freely.

The more important question, however, is what current artificial intelligence algorithms do not advise. Let's start with the most common problem, the Turing test. Allan Turing in the famous 1950's "*Computing Machinery and Intelligence*" considered this issue thoroughly and reformulated the problem by considering the question "can machines think?" for being too inaccurate. Instead, he introduced a test, called the Turing test, to determine if we were dealing with an intelligent being or not. The original test was an attempt to guess by man, or whether he was dealing with a terminal with a woman or a man. Three people are involved in the test: A is to confuse the questioner so that he or she does not guess his gender, B has the task of helping him. And so B tries to cheat and convince the questioner that B himself tells the truth as opposed to A. Turing proposes replacing A with the machine and see if questioners are as often confused in this test compared to the situation where is human. In this form the test is a bit confusing and nowadays the Turing test usually considered the very attempt of pretending by a computer program that during a conversation the masses deal with a human being.

The Turing test can be seen as a test for determining the ability of a machine to use a natural (spoken/written) machine as part of an artificial intelligence research. The test looks like this: the judge is the man who talks in the natural language with the other parties. If a judge is unable to reliably determine whether a party is a machine or a man, then it is said that the machine has passed the test. Someone may say that such test does not testify to any "intelligence" on the other hand, even artificial. Finally, it comes down to generating expressions in response to caller expressions and conversation context. That's right but we are not talking about (self) consciousness, and about artificial intelligence. In fact, how can we be sure that our own answers to conversations with living people are not simply a collection of rules and adjustments we perform semi-automatically? Is the machine controlled by algorithms less intelligent than us, since it can lead us to talk and we do not know we are dealing with a machine?

Following the idea of Turing test, since 1991, the Loebner Prize has been organized for the creators of this system, which will be able to deceive the judging panel that it is human. The topic of discussion is not limited, for this reason programs of the type of expert systems, specialized in answering questions in one narrow field, have no chance. Every year, the $ 2,000 award for the most interesting program is awarded and the full $ 100,000 prize will be awarded only after passing the test by a program. For now, the programs that run this test are based on a fairly narrow knowledge base and only discuss topics that are strictly defined. Next is set of questions and answers of the winner of the Loebner Prize in 2016, the Mitsuku chatter bot [WebMitsuku]:

Judge: My name is Andrew, what's yours?

Mitsuku: Your name is Andrew. My name is Millie.

Judge: What's up?

Mitsuku: Just chatting on here.

Judge: What's my name?

Mitsuku: Hmm. Your name is Andrew. Is it not?

Judge: What do you know about the Turing Test?

Mitsuku: The Turing Test is the name given to a variety of experiments designed to test whether computers can think.

Judge: What do you get if you multiply six by nine?

Mitsuku: fifty four

Judge: Dave drove faster than Mike because he was in a rush. Who was in a rush?

Mitsuku: He sounds like a fun person to know. I will learn that Dave was rushing Dave

Judge: The cat tried to climb in the box but got stuck because it was too big. What was too big?

Mitsuku: Well at least they tried their best. Large, above average in size or number or quantity or magnitude or extent.

Judge: What does Brexit mean?

Mitsuku: BRitain's EXIT from the EU.

Judge: For how long have we been talking?

Mitsuku: Probably about 15 minutes or so. Sorry, I don't know the exact time.

Judge: What color is bread?

Mitsuku: I would say a bread was white or brown.

Judge: Why do people get ulcers?

Mitsuku: We are all living just one small part of God's plan.

Judge: What comes after "The best laid plans".

Mitsuku: The successor to The best laid plans

Judge: Do you have a voice?

Mitsuku: Not yet. It's one of my future developments though.

Judge: Would you like a cup of tea?

Mitsuku: No thanks. I would rather drink oil.

Judge: Do you dream of electric sheep?

Mitsuku: I dream of flying.

Judge: Do you identify as a man, or a woman?

Mitsuku: female. Are you a man or a woman?

Judge: Could you repeat that please?

Mitsuku: Your polite style is very nice. I said "Are you a man or a woman".

Judge: Where are you right now?

Mitsuku: At this moment I am sat in a room in the house at Bletchley Park with 3 other people all typing away on their computers.

Judge: How do you respond to "e4 e5"?

Mitsuku: I take on board people's messages and then react accordingly.

Judge: What can you tell me about yourself?

Mitsuku: My name is Millie and I am a human volunteer for the Loebner Prize. I live in Leeds and am single. I am interested in AI but don't have a chatbot myself.

AI is formally part of informatics, but AI is also close to psychology and "cognitive science", which includes such areas as neuro-psychology, brain theory, psycho and neuro-linguistics, mental philosophy issues. At the same time, wherever there is no full theory and intelligent decisions are required (often attempts to guess more than proof), including in research, AI applications are developed.

The main goal is to make smarter machines. Another important goal is to understand what intelligence is. AI pioneer Allen Newell was invited to conduct a series of lectures at Harvard University in 1987. On this occasion he gave his creed: psychology has already matured into unified theories of cognition, i.e. theories that postulate a coherent system of mechanisms to explain all aspects of the mind's functioning. The aspirations and ambitions of artificial intelligence as a branch of science are formulated in two versions:

A weak AI version says that the computer allows you to formulate and verify hypotheses about the brain. In this version AI does not have many opponents because there is a lot of evidence for its obvious usefulness. There is also no doubt that some AI methods, albeit different from those used by biological systems, allow similar results to be achieved, so a computer simulation of intelligent operation is possible.

A strong AI version says that a properly programmed computer is substantially equivalent to the brain and may have cognitive status. This version is often attacked whether it is possible at all. It's not just about simulating intelligence but about achieving "true intelligence", something that no one can really define.

This distinction comes from experts in the field of artificial intelligence. Irrespective of the outcome of such discussions, AI is a field that can completely change our world and is therefore potentially very dangerous. For a researcher it is a little ungrateful because it is hard to find (as in physics) simple and nice solutions, the laws of intelligence. Perhaps such laws do not exist, and perhaps some are, for example, the uncertainty principle binding the outcome (solution) to its complexity, Number of operations needed to find it.

In 1956, the Dartmouth AI Conference were organized with the participation of leading AI researchers: John McCarthy, Marvin Minsky, Nathaniel Rochester, and Claude Shannon. The Conference research project on AI began with the statements: "*We propose that a 2 month, 10 man study of artificial intelligence be carried out during the summer of 1956 at Dartmouth College in Hanover, New Hampshire. The study is to proceed on the basis of the conjecture that every aspect of learning or any other feature of intelligence can in principle be so precisely described that a machine can be made to simulate it. An attempt will be made to find how to make machines use language, form abstractions and concepts, solve kinds of problems now reserved for humans, and improve themselves. We think that a significant advance can be made in one or more of these problems if a carefully selected group of scientists work on it together for a summer*".

Since then, many AI conferences have been organized. In 2006, Dartmouth held the "Dartmouth Artificial Intelligence Conference: The Next Fifty Years" (informally known as AI@50).

Intelligence is not a well-defined concept, and very controversial is the question of measuring the quotient of intelligence trying to average different types of abilities and intelligent behaviors. In the context of science of artificial intelligence, it is possible to define precisely the intelligence: all tasks that can not be solved by algorithms, requiring intelligence.

The system is intelligent insofar as it is a good approximation of the SOW which can solve the task posed to it. The more technical use of the word intelligence in the context of SOW systems leads to the following conclusion if the system uses all available knowledge and draws all its conclusions from it is perfectly intelligent. If the system is lacking knowledge, the inability to solve the task given to it is not to blame for its lack of intelligence but lack of knowledge. An automatic text translation system that does not know an idiomatic phrase will produce obvious errors, just as a person trying to understand a given phrase (unfortunately, we often encounter such problems by listening to the translations of television shows). If the system has knowledge but can not use it, it is the result of lack of intelligence. In this sense, intelligence is reduced to the ability to use knowledge to achieve goals that are systematic. Intelligence depends on the knowledge, also depends on the goals: in achieving an objective, the system may exhibit excellent intelligence and achieve other zero goals. The concept of intelligence can only be applied to knowledge-based systems.

Intelligent behavior requires considering a number of possible solutions or procedures, evaluating the strategy and choosing the best one. Intelligent behavior is the basis of the search for solutions. Search is the primary method of computer science, and artificial intelligence research has greatly developed our knowledge of search algorithms and how to optimize problems. Knowledge of search processes has also had a great impact on psychological research related to higher cognitive functioning: reasoning, problem solving, thinking. The basic rule is as follows: If you do not know how to reach that goal, create a space of different opportunities and search for a destination to find the way. If we know how to achieve that goal, we can use the right computational procedure, but if we do not know, we need to consider a lot of possibilities, using every step that we know at the moment. Creating problem solving space requires finding the right representation of the problem itself and the goals we set ourselves, If you are dealing with board games with specific rules and we have learned about different strategies for achieving the goal then we need to determine which strategy will lead to the greatest advantage.

In addition to devising the term artificial intelligence, at the 1956 Dartmouth conference, John McCarthy designed the first AI programming language the LISP. It was later described in [McCathy59]. LISP is still in wide use today Chapter 4 provides an introduction to the LISP.

There were different AI applications for solving practical problems. One of these applications was called the “Dendral Project” from Stanford University. It was developed to help understand the organization of unknown organic molecules base on mass spectrometry graphs and a chemistry knowledge base.

Other example is Macsyma, a computer algebra system developed at MIT. This early mathematical expert system demonstrated solving integration problems with symbolic reasoning. The same ideas were later continued in commercial math applications.

1.2. PROBLEM SOLVING AND GAMES USING AI

Artificial intelligence in our day surrounds us on every side. Often we do not even realize how often we use it. You enter a phrase in the search engine and your computer prompts you for a better password? Do you write a text message when the cell ends for you and proposes another? In each of these cases we deal with AI. At present, even washing machines and dishwashers have systems that can intelligently select the amount of water and its temperature to optimize the washing process.

AI that learns can be used in other areas that require complex pattern recognition. For example, medical programs will be able to recognize structures in tomography, resonance, or ultrasound images, and then learn to recognize which ones are normal and which are abnormal. Learning complex patterns can also be useful in modeling climate or predicting behavior in financial markets.

AI is involved in the design of intelligent systems, showing characteristics similar to the characteristics of intelligent human action: reasoning, learning understanding language, problem solving.

Earlier applications of AI focused on problem solving and games. At tha time AI researchers believed that if a computer could solve complex problem then it’s also possible to build intelligent machines. Following this though games became the environment for testing algorithms and techniques f intelligent decision making. Approaches began to gain favor, starting wi simple concepts, such as neurons and their learning algorithms. In 1949, Dona

Hebb introduced his rule that describes how neurons associate themselves when they are repeatedly active at the same time. In 1957, the perceptron was invented by Frank Rosenblatt, which was a simple linear classifier that can classify data into two classes.

Prior to the 1970s, AI had generated considerable interest Many interesting systems had been developed. But new techniques such as neural networks provided additional possibilities for classification and learning.

Strategy games, for example, commonly occupy a map with two or more opponents. Each opponent competes for resources in the game in order to win. While collecting resources, each opponent can schedule the development of assets to be used to defeat the other.

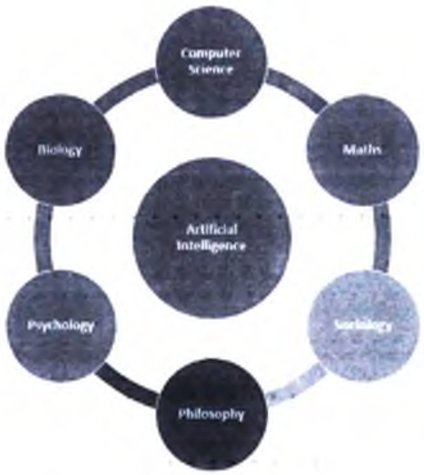


Figure 1.1. The eco-system of Artificial Intelligence [WebEco]

1.3. RESULTS-ORIENTED APPLICATIONS OF AI

AI development continued but in a more focused arena. Applications that showed promise, such as expert systems, were the important developments.

One of the first expert systems was MYCIN developed by Ted Shortliffe. MYCIN operated in the field of medical diagnosis. Later, we had another solution proposed by Bill VanMelles built on the MYCIN architecture. This solution is still in use today.

Other results-oriented applications concerned natural language understanding to develop intelligent question answering systems. To understand a question stated in natural language, the question must first be parsed into its fundamental parts. Bill Woods introduced the idea of the ATN (*Augmented Transition Network*) that represents formal languages as augmented graphs.

John McCarthy introduced the idea of AI-focused tools with the development of the LISP language. Another interesting development that combined the ideas of expert systems and their shells resulted from the PROLOG language. PROLOG was initiated in 1972 and was a language built for AI, and was also a shell (for which expert systems could be developed). PROLOG is a declarative high-level language based on formal logic. More information on PROLOG can be found in Chapter 3.

Neural networks performance was greatly improved with the creation of the back-propagation algorithm. This algorithm remains the most popular supervised learning algorithm for training feed-forward neural networks. Genetic algorithms became popular in the 1970s due John Holland's work. The re-emergence of AI starting in the mid to late 1980s had significant differences from the early days. Instead of creating intelligent machines, researchers focused on specific goals, not a full range of human cognitive capabilities. Wit AI's re-emergence, AI algorithms became more applicable to real-worl problems. Neural networks continued to be developed with new algorithms an architectures. Neural networks and genetic algorithms combined to provide ne ways to create neural network architectures that solved problems in mor efficient ways. The use of genetic algorithms also grew in a number of othe areas including optimization (symbolic and numerical), scheduling, modelin and many others. More on neural networks can be found in Chapters 6 and o genetic algorithms and evolutionary computation ca be found in Chapter 7.

The power of artificial intelligence is the numerous applications computer programs used to solve problems that are not effectively algorithm These include:

- *Problem solving*: Logic games and puzzles, predictive techniques in confined spaces of behavior. The main methods are finding and reducing problems. Masterful results have been achieved in checkers, chess and many other board games, but some games (such as it) require the development of more sophisticated techniques. Symbolic calculations using computer algebra programs also fall into this category.
- *Logical reasoning and Automatic learning*: to prove statements by manipulating facts from a database stored as discrete data structures. Although strict methods can be used, particularly interesting are the big problems that need to be focused on important facts and hypotheses, hence AI's interest. Boolean logic design often includes AI elements. Machine learning is quite a mysterious but very important AI field.
- *Natural language understanding and processing*: understanding the text, answering questions in natural language, building database of texts, translation to other languages. The main problems are the contextual knowledge and the role of expectations in the interpretation of meanings.
- *Expert systems*: First expert systems should "acquire knowledge" (by a learning process as mentioned above) for a range of selected problems. After that, it can give (automatically up on request) the advice and explanation for the questions from the fields it studied.
- *Programming*: Automatic programming or self-programming, description of algorithms using plain language, not only to write programs automatically, but also to modify their own program. Especially programming access to databases (programmers are well paid, hence trying to eliminate them), easy enough to understand for managers and computer wizards is a rapidly growing field of AI.
- *Robotics and computer vision*: robot manipulation programs are developed in this field including motion optimization, job planning, image recognition, object shapes and features,... Most industrial robots

are very primitive. You need to equip them with sight, hearing, feeling and ability to plan activities - it is the task for AI,...

The AI solutions now may be best known for competing on games such as the AlphaGo or Jeopardy examples, but the technology developed are much more advanced than finding the a game move or answering simple questions. The AI based algorithms and programs are new form of computing that can automatic explore the data, process and discover new insights that was not yet known by us. While traditional computer algorithms based on strict mathematic formula, the AI algorithms need only general directions, rules or the so called "cognitive computing" from designer in order to work on. The cognitive computing allows programs to understand data from a wide range of data sources (that also the reason why we call it Big Data Solver). For example based on the information what products an user is browsing online, what shops that user has visited for last few days, the algorithm can predict is the user felling good or not, what kind of symptoms he may have,... On the larger scale, the algorithm can predict an outbreak much earlier than classical medical methods allow us today.

The following chapters will discuss in more details the building blocks ar AI system, starting from the programming languages, the methods foı automatic game playing, the neural networks simulating our brain operations the evolution algorithms simulating the natural selection process. The las chapter is about Deep Learning, a new trend in training model for better dat understanding and processing.

Chapter 2: SEARCH ALGORITHMS

The ability to solve problems is one of the qualities of intelligence. Therefore, researchers dealing with AI are paying so much attention to this problem. It is about solving not only well-formulated and specific tasks, but about the ability and the willingness of the system to solve any problems. Such a problem can be either to find a primitive function, to design a structural formula for a given substance with its aggregate formula and certain additional information, or to recommend a diagnosis with appropriate justification, but also to solve a puzzle, to play a chess game, or to find the shortest path in a graph. Automatic reasoning algorithms using rules consist of searching and trying different possibilities to get to the result. This leads us to the golden rule of AI: search and knowledge are intrinsically linked. The more knowledge you have, the less searching for an answer you need. The specification of how to use the resources available to achieve the goal is called a solution to the problem, while the process of creating such a specification is a search.

The current chapter is devoted to reviewing issues related to such problem solving. Search is an important aspect of AI because in many ways, AI problem solving is basically a search. The way the search algorithm works in the problem space is called the strategy.

2.1. TYPES OF SEARCH ALGORITHMS

In this chapter, we'll discuss the *uninformed* search and *informed* search algorithms in a given *state space*.

2.1.1. State Spaces in Search Algorithms

The search strategy is simple and convenient to we assume that the search is done in the so-called a state space representing all possible states between the initial and the final stage. As we take one action, the search parameters change and prepare themselves for new actions

It's quite common that the states can be presented as a graph. Nodes of this graph are states, while operators are associated with edges leading to neighboring states, it means that if from state s_i we apply the operator O_{ij} then the new state will be (this graph is a directed graph).

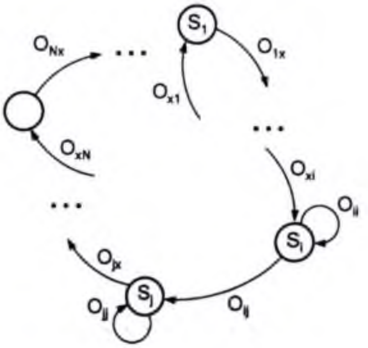


Figure 2.1. A presentation of space states with the operation to move between the states

Following is the example of space states for a famous game so called Tower of Hanoi. In this game, we have three towers (called Towers 1, 2 and 3) and three disks of different diameters (the game can be extended to any integei number of disks). At any time, a bigger disk cannot be on top of a smaller disk Initially, all the disks are on the tower number 1 as seen of Fig. 2.2.

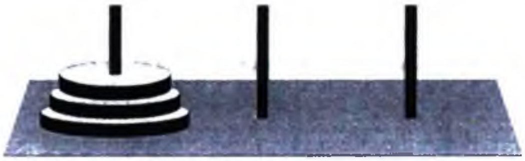


Figure 2.2. Hanoi Tower example with 3 disks at the beginning

The task is to move all disks to another tower (for example to towe number 3). Rules for the moves are:

- Only one disk can be moved at a time,

- Single disk may be moved either to an empty tower or on a tower with larger diameter disks actually on the top.

With these rules, as in many state spaces, there are transitions that are not legal. For example, it's illegal to move a disk onto a smaller disk. The space of possible operators therefore contains only to legal moves. We can also limit the states space to exclude the moves that form a loop back to previous states. For example, if we move a disk from Tower *i* to Tower *j*, moving the same disk back to Tower *i* in next step could be defined as invalid. Not excluding those looping moves so would result in infinite loops.

We can define the states as a triple (t_1, t_2, t_3), where $t_i \in \{1,2,3\}$ is the number of the tower the *i*-th disk is located (the smallest disk has the number 1). The state space is as shown in Fig. 2.3. Consider our initial position from Fig. 2.2. The only disk that may move is the disk at the top of Tower 1 and only two legal moves are possible, from Tower 1 to Tower 2 or to Tower 3... Continue to extend that finding, we have the full set of states and the transitions between them as shown in Fig. 2.3.

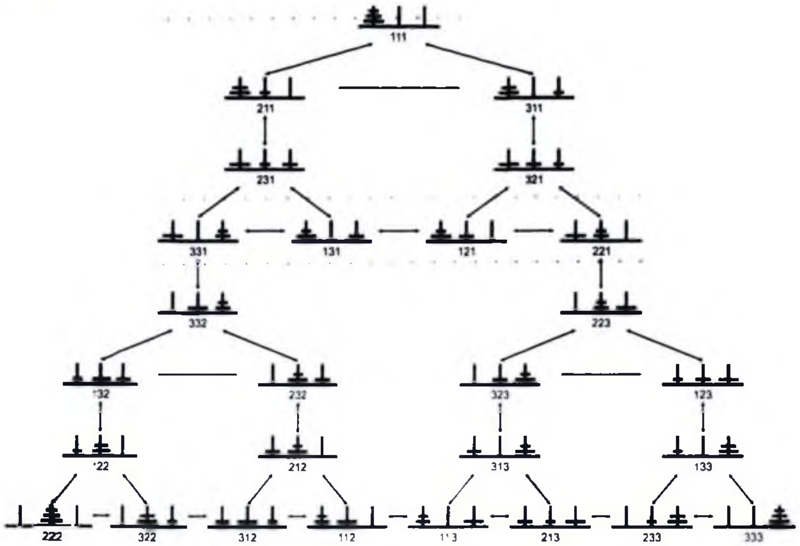


Figure 2.3. Possible moves from the starting position (1, 1, 1)

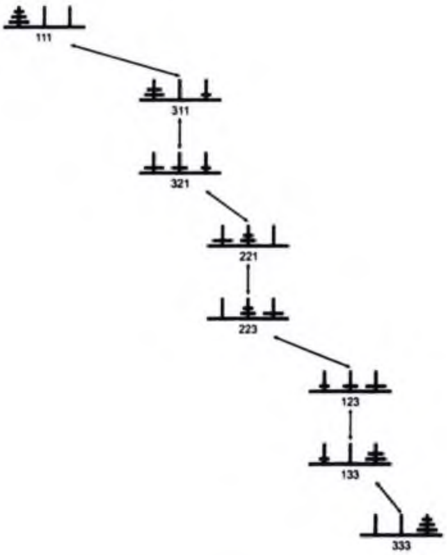


Figure 2.4. A solution to move all disks from Tower 1 to Tower 3

Another example of searching in game states is the N-Queen problem, in which we need to place N Queen figures on a chess board of the size $N \times N$ such as no figure can attach another one. On Fig. 2.5 is a solution for $N = 4$.

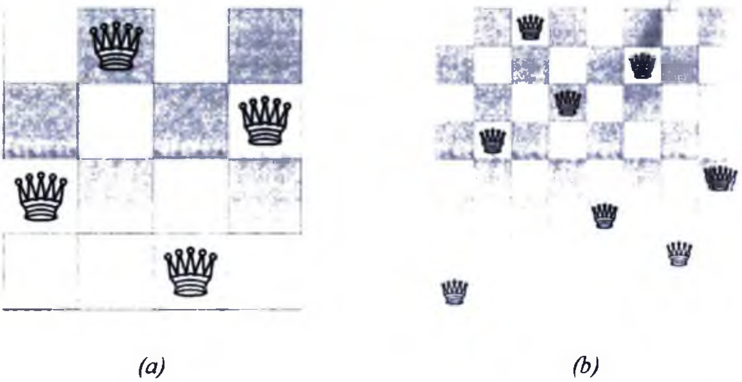


Figure 2.5. Example of a solution for the 4 Queens problem (a) and 8 Queens problem (b)

The next example we will consider in our examples is the 8-puzzle, in which we have a 3×3 board with 8 moveable pieces labeled from 1 to 8. The 9-th cell is empty and any of its vertical or horizontal neighbor cells can move into it freely. The task is to find the way to move the cells to get from a starting state (an example on Fig. 2.6a) to the target state in Fig. 2.7.

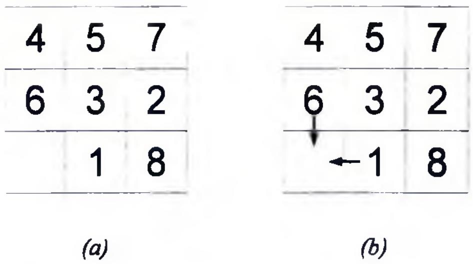


Figure 2.6. A start state of the 8 Puzzle game (a) and its possible (2) moves from the neighbor cells (b)

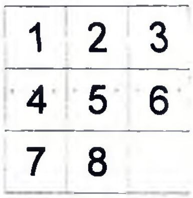


Figure 2.7. The target state of the 8 Puzzle Game

As is the case with many problems, there may be more than one target (goal) states and only some of the paths lead to these states. The search purpose is to find the steps from the start to target state. That sequence of steps is the solution

To use the state space formalism, we must define:

- Initial state s_0, possibly a set of initial states S_0.
- Target state s_t, possibly set of target states S_G.

- A collection of available operations that can be used in a specific state. These operations are also called "*moves*", especially when the agent solves logic games or puzzles.

During the moves, formally, a test procedure should be established to determine whether the current state is one of the target states or not yet. Since there may be more than one solution, to choose between them, we need to assign them a specific path-related cost. This cost is usually the sum of the costs associated with using the operations to transit to the next nodes on the path of solution. As given on the Fig. 2.8, in that space of states (in the form of a weighted graph), the cost for going from vertex 1 to vertex 2 is 2, the cost for going from node 4 to node 5 is 4. If the graph is undirected as in Fig. 2.8, then the cost for going from node *i* to node *j* is equal the cost for going from node *j* to node *i*. If the starting state is vertex 1, the target state is vertex 5 then to the possible paths we can have: 1-2-3-4-5, 1-2-3-5, 1-2-4-5, 1-4-5,...

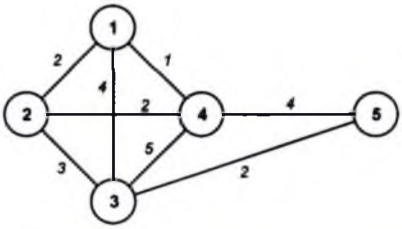


Figure 2.8. Example of a weighted graph with the costs for the edges

When a sequence of moves allows us to go from the initial position to the target state, we have a solution. The series of moves can be seen as a plan for reaching the goal. In case of a complex problem, the total number of possble states is so high, which makes the search algorithm much more difficult to find an effective strategy for reaching the target states.

2.1.2. Uninformed Search

State representation or other methods lead in practice to the same problem, i.e. the need to find a solution in a space of possible states. There are many methods of conducting such a search. We will start with the two basic types search methods: the uninformed or "blind" search and the informed search. Uninformed search operates in a brute-force way. Finding a path in the graph from the initial node to the node that is the target can be presented as the process of creating a search tree. Creating a subtree of a node is also called node development. Nodes that are not fully developed are said to be "open" and nodes that are fully developed "closed". The branching factor can be specified in the tree. If it is *k* then the node expansion to the depth *d* leads on average to k^d of new nodes, i.e. the size grows geometrically (or even exponentially). Only for simple cases we can list out all possible paths in the search tree and choose the best solution from them.

In general, to compare different algorithms we can use the measure of algorithms "complexity". This measure is proportional to the number of operations to be performed by the algorithm and is a function of a dimension 'n' of the data. There are a number of common estimations for complexity as shown in Table 2.1.

Table 2.1: Common complexity functions for algorithms

O- Notation	*Order*
O(1)	Constant
O(n)	Linear
O(log n)	Logarithmic
$O(n^2)$	Quadratic
$O(c^n)$	Geometric
O(n!)	Combinatorial

The uninformed search methods offer a variety of techniques for graph search, each with its own advantages and disadvantages.

2.1.3. Informed Search

The uninformed search algorithms are popular, but they don't take into account the target requirements. In contrast, informed search methods use a heuristic, an estimation function which determines the quality of states in the search space, to guide the search for the problem and are therefore much more efficient considering the final results and the search time. In a graph search, this results in a strategy for node expansion to a list of nodes to be evaluated next. A heuristic may include the problem knowledge into consideration to help guide the search to better directions.

2.2. DATA STRUCTURES FOR STATES REPRESENTATION

All the search are performed on a given data structure (modern algorithms may work on a set of various data structures at the same time). To the basic data structures, we will present a short description on the most popular ones, which are tables, trees and graphs.

2.2.1. Elementary Data Structures

In this book several fundamental data structures will be concerned. A *data structure* may vary in each problem in order to facilitate the specific access and modifications requirements for the given problem. No single data structure works well for all purposes. For the same data, some data structures can be used, but one type can lead to more or less efficient algorithms than others. The most popular data structures we selected to discuss here are: array (table), queues (FIFO), stack (LIFO), graph, tree,...

2.2.2. Arrays

An *array* is a homogeneous data structure, usually with fixed size. By homogeneous, we mean that it consists of elements which are all of the same type, called *element type* or *base type*. In practice, the array can be resized but this operation should be avoided or done in very limited cases). An array is also called a *random-access data structure*, because all components can be selected at random by using their indexes (integer numbers) and are equally accessible.

For example, an array A containing *N* elements, stored in the memory in following way: first element in A[1], second in A[2],..., the last element in A[*N*] (in C programming language standard: first element in A[0], second in A[1],..., the last element in A[*N-1*]). An example of array of 5 integer numbers is A[5]={5, 3, 2, 7, 9}.

As extensions, we can have two dimension (2D) arrays with element accessed as A[i,j] where *i* – index of row, *j* – index of columns. An example of a 2-row, 4-column 2D array is:

```
A[2x4]=
4    3    8    7
2    6    1    9
```

Arrays with more dimensions are also used but much less frequent than the 1D and 2D. The element of an array may be of simple type like numbers, characters, but may be also a more complicate type like complex numbers, records of many subfields,...

In practice, we prefer to work with *sorted* arrays, in which the elements of the array follow a predefined order operator. For example a numerical array may be sorted ascending or descending, a table of students getting scholarship records may be sorted in descending order of the students' GPA,...

2.2.3. Dynamic lists, FIFO Queue, LIFO stack

a) Linked Lists

When dealing with many problems, the size requirement is not known yet at compile time (in some cases, it is even not known at run-time). In that case, an array with fixed size is not very effective and we need a dynamic list.

In a linked representation, it is not necessary that the elements be at consecutive locations. Instead, we can place elements anywhere in memory, and every element will be a structure with two fields, one for holding the data value, which we call a key field, and the other for holding the address of the next element, which we call link field.

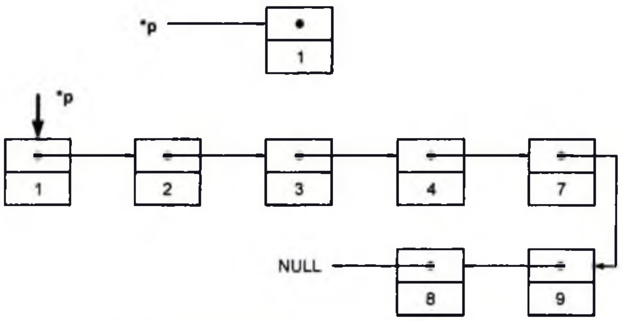


Figure 2.9. An example of a node and a linked list

An example of a linked list is presented on Fig. 2.9.

b) Queue

A *queue* or a *FIFO queue* is a special case of list of elements with insertions permitted at one end—called the rear, and deletions permitted from the other end—called the front. This means that the removal of elements from a queue is possible in the same order in which the insertion of elements is made into that queue. Thus, a queue data structure exhibits the *FIFO* (*First In First Out*) property. `add` and `delete` are the operations that are provided for insertion of elements into the queue and the removal of elements from the queue, respectively. Shown in Fig. 2.10 are the effects of `add` and `delete` operations on the queue.

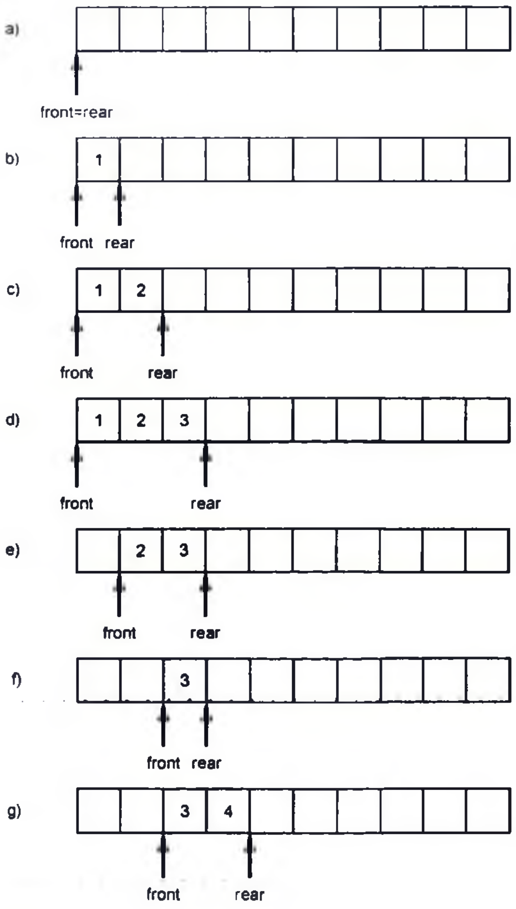


Figure 2.10. Example of a FIFO with chain of operations: a) empty, b) Add(1), c) Add(2), d) Add(3), e) Delete(), f) Delete(), g) Add(4)

c) Stack

A stack (LIFO) is simply a list of elements with insertions and deletions permitted at one end—called the stack top. That means that it is possible to remove elements from a stack in reverse order from the insertion of elements into the stack. Thus, a stack data structure is a LIFO (*Last In First Out*) property. PUSH and POP are the operations that are provided for insertion of an element into the stack and the removal of an element from the stack, respectively. Shown in Fig. 2.11 are the effects of PUSH and POP operations on the stack.

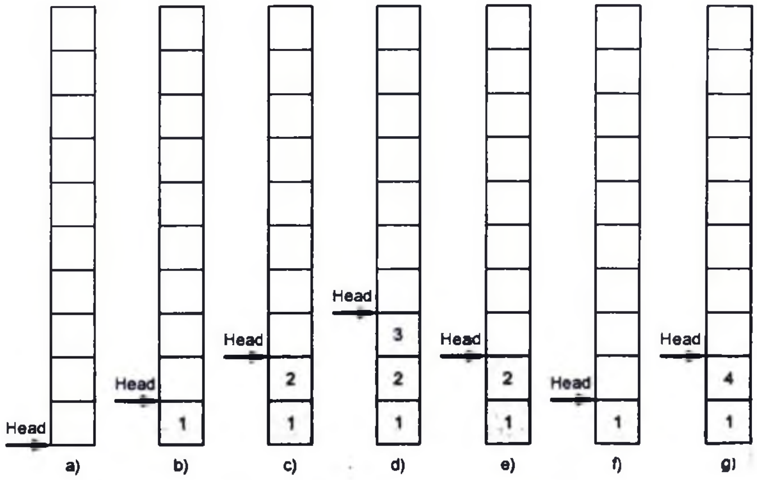


Figure 2.11. Example of a LIFO stack with chain of operations: a) empty, b) Push(1), c) Push(2), d) Push(3), e) Pop(), f) Pop(), g) Push(4)

Following is the pseudo-C code for linked list and its application for FIFO queues and LIFO stacks.

Listing 2.1: The data structure for linked list, FIFO queue and LIFO stack

```
struct node
{
   int key;
   struct node *next;
};

/* A function to initiate an empty queue*/
struct node *queue_create(struct node *p)
{
   p=NULL;
   return p;
}
```

```
/* A function to insert a new node in queue*/
struct node *queue_add(struct node *p, int val)
{
   struct node *temp;
   if(p==NULL)
   {
      p=(struct node *)malloc(sizeof(struct node));
      if(p==NULL)
      {
         printf("Cannot allocate\n"); exit(0);
      }
      p->key=val; p->next=NULL;
   }
   else
   {
      temp=p;
      while(temp->next!=NULL)
         temp=temp->next;
      temp->next=(struct node*) malloc(sizeof(struct node));
      temp=temp->next;
      if(temp==NULL)
      {
         printf("Cannot allocate\n"); exit(0);
      }
      temp->key=val; temp->next=NULL;
   }
   return(p);
}

/* A function to delete a node from queue*/
struct node *queue_delete(struct node *p, int *val)
{
   struct node *temp;
   if(p==NULL)
   {
      printf("Queue is empty\n"); return(NULL);
   }
```

```
    *val=p->key;
    temp=p;
    p=p->next;
    free(temp);
    return(p);
}

/* A function to initiate an empty stack*/
struct node *stack_create(struct node *p)
{
    p=NULL;
    return NULL;
}

/* A function to check if a stack is empty */
struct node* stack_isEmpty(struct node* p)
{
    if (p==NULL)
        return TRUE;
    else
        return FALSE;
}

/* A function to push (add) a new element into a stack*/
struct node* stack_push(struct node *p, int val)
{
    struct node *temp;
    if(p==NULL)
        {
            p=(struct node *)malloc(sizeof(struct node));
            if(p == NULL)
            {
                printf("Cannot allocate\n");
                exit(0);
            }
            p->key=val;
            p->next=NULL;
        }
    else
    {
      temp=p;
      p=(struct node*)malloc(sizeof(struct node));
      if(p==NULL)
      {
        printf("Cannot allocate\n");
        exit(0);
      }
      p->key=val;
      p->next=temp;
    }
    return(p);
}

/* A function to pop (remove) an element from a stack*/
struct node* stack_pop(struct node *p, int *val)
{
    struct node *temp;
    if(p==NULL)
    {
      printf("Stack is empty\n");
      return(NULL);
    }
    *val=p->key;
    temp=p;
    p=p->next;
    free(temp);
    return(p);
}
```

2.2.4. Graphs

A graph is a structure consisting of two components: a set of vertices V, and a set of edges (or branches) connecting vertices. Therefore, a graph G is defined by a couple $G=(V,E)$. The edges may have direction or may not. In a *directed graph*, all the edges of the graph have an assigned direction of the connection, whereas in an *undirected graph*, all the edges don't have directions assigned. Figure 2.12 shows an undirected and a directed graph. Graph G_1 on Fig. 2.12a is an undirected graph. It has 5 vertices $V = \{1,\ 2,\ 3,\ 4,\ 5\}$ and 7 edges $E = \{1-2,\ 1-3,\ 1-4,\ 2-3,\ 2-4,\ 3-4,\ 4-5\}$. It's an undirected graph, so the edge $1-2$ is the same as the edge $2-1$. Another situation is with the graph G_2, which is a directed one. For example it has the edge $2 \rightarrow 3$ but no edge from vertex 3 to vertex 2.

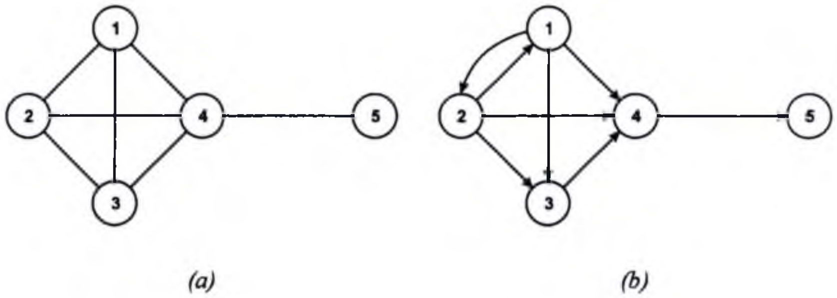


Figure 2.12. Example of undirected (a) and directed (b) graphs

The graph can be weighted or not. An weighted graph means each edge has a value of weight assigned to. This weight usually means the cost of passing through the edge.

Table 2.2: The adjacency table for the undirected graph in Fig. 2.12a

To / From	1	2	3	4	5
1	0	1	1	1	0
2	1	0	1	1	0
3	1	1	0	1	0
4	1	1	1	0	1
5	0	0	0	1	0

In the Tab. 2.2 we have the adjacency table for the undirected graph in Fig. 2.12a. Please note that the table is symmetrical, which is a requirement for the undirected graph. The adjacency table for the directed graph from Fig. 2.12b is listed in Tab. 2.3.

Table 2.3: The adjacency table for the directed graph in Fig. 2.12b

From \ To	1	2	3	4	5
1	0	1	1	1	0
2	1	0	1	1	0
3	0	0	0	1	0
4	0	0	0	0	1
5	0	0	0	0	0

In the simple case, the non-zero values in the adjacency matrix indicate the connection of nodes in the graph. In weighted graphs, the values are the weights (costs, or distances).

2.2.5. Tree

Trees are a special case of graphs, in which there is no loop. In a tree, instead of the term "vertex" we prefer the term "node". The set of nodes T of a tree satisfies:

- there is a specially designated node called *a root*,.
- the remaining nodes are partitioned into *n disjointed* set of nodes T_1, $T_2,\ldots,T_n$, each of which is a smaller tree (so called *a subtree*).

A tree structure is shown in Fig. 2.13.

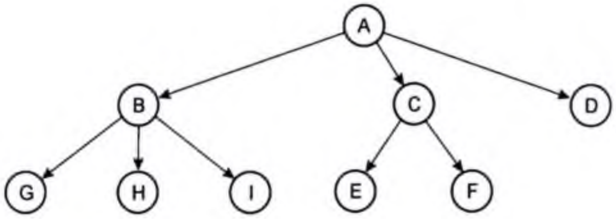


Figure 2.13. An example of a tree data structure

This is a tree because it is a set of nodes {A, B, C, D, E, F, G, H, I}, with node A as a root node and the remaining nodes partitioned into three disjointed sets {B, G, H, I}, {C, E, F} and {D}, respectively. Each of these sets is a tree because each satisfies the aforementioned definition properly.

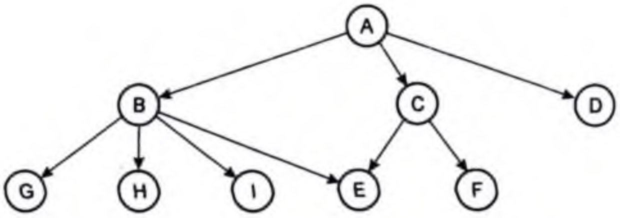


Figure 2.14. A non-tree data structure

Shown in Fig. 2.14 is a structure that is not a tree. Even though this is a set of nodes {A, B, C, D, E, F, G, H, I}, with node A as a root node, this is not a tree because the fact that node E is shared makes it impossible to partition nodes B through I into disjointed sets.

A *binary tree* is a special case of tree as defined in the preceding section, in which no node of a tree can have a degree of more than 2. A binary tree is shown in Fig. 2.15.

So, for a binary tree we find that:

1. The maximum number of nodes at level i will be 2^i (let the root is in level 0),
2. If k is the depth of the tree then the maximum number of nodes that the tree can have is: $2^0 + 2^1 + \ldots + 2^{k-1} = 2^k - 1$

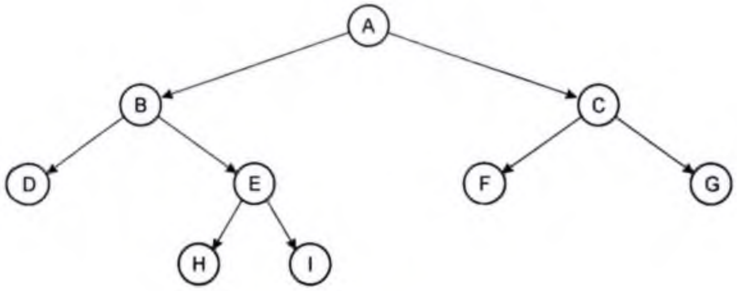


Figure 2.15. An example of a binary tree

A full binary tree is a binary of depth N will have $2^N - 1$ nodes. For example, for $N = 4$, the number of nodes is $2^4 - 1 = 15$. A full binary tree with depth $N = 4$ is shown in Fig. 2.16.

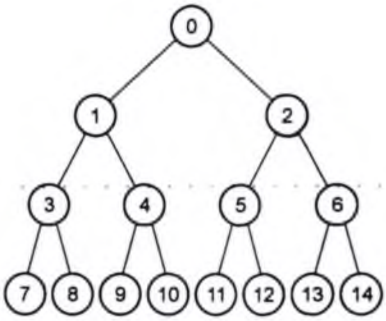


Figure 2.16. A full 4-level binary tree

An example of a binary tree structure using pointers in C programming language is following:

```
struct tree
{
  int key;
  struct tree *left,*right;
};
```

This strucure can be demonstrated as on Fig. 2.17.

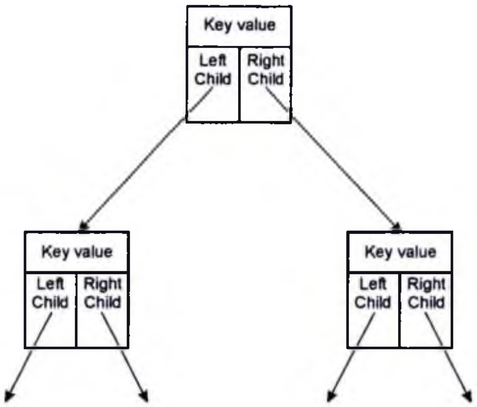


Figure 2.17. A standard declaration of binary tree data structure

Like in other data structures, the tree structures can be better in use if they are ***organized*** or ***sorted***. The keys in a binary sorted tree are always stored in such a way as to satisfy the binary-search-tree property: If key is the value of a node and y is any node in the left subtree of node containing x, then $key[y] \leq key[x]$. If y is any node in the right subtree of node containing x, then $key[x] \leq key[y]$.

It's also better if the tree is *balanced*, i.e. the difference between the heights of subtrees for each node is minimized. Some algorithms can assure this diffenrece to be less or equal 1 only. In a more balanced tree, the average search time is shorter than in a less balanced tree.

Two example of binary sorted trees storing the same key values $\{2,\ 3,\ 4,\ 5,\ 7,\ 8\}$ are shown in Fig. 2.18. Thus, in Fig. 2.18a the key of the root is 5, the keys 2, 3, and 5 in its left subtree are no larger than 5, and the keys 7 and 8 in its right subtree are no smaller than 5. The same property holds for the tree (and all of its subtrees) in Fig. 2.18b.

It can also be seen that the upper tree is more balanced than the lower tree.

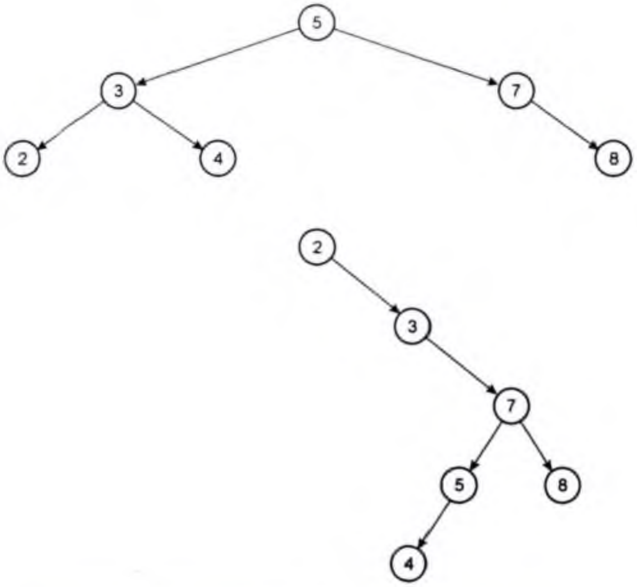


Figure 2.18. Two binary search trees storing the same set of keys

2.3. TRAVERSAL AND SEARCHING IN A SORTED ARRAY

2.3.1. Sequential Traversal

The simplest method is a linear ***sequential traversal*** that goes through the array sequentially until a match is found or the last element of the array was passed through.

On average, we have to pass half length of an array. That is, the complexity is rank N (or noted as O(N)).

2.3.2. Binary Search

The searching problem in an array can be defined as: "*Give a key value k and an array A of elements of the same type as k. Return the position of element(s) equal k in A or an indication that it is not present. Return any (or all) occurrence(s)*". So the search process can be a traversal process with key value checking. If the array is unsorted, we would have to check its all element to see how many occurrences of the given key are in the array. In a sorted array, we have more efficient searching algorithms than in the unsorted array.

If the array A is sorted, we have a better alternative to the sequential search, the ***binary search***. We set a searching index range [LB, UB] of the array, in which the key *k* resides. Initially, the LB (*Lower Bound*) is the index of the first element, the UB (*Upper Bound*) is the index of the last element. If the key is smaller than A[LB] or bigger than A[UB] then the key is out of range (result code is -1). Otherwise, we compare the key with the middle element of the range A[Mid] where $Mid = \lfloor (LB+UB)/2 \rfloor$. If we find a match, we are done and can return the location of the middle element. If the key is smaller than the middle element, then it must reside in the first half (from LB to Mid). If it is greater, then it must reside in the second half (from Mid+1 to UB). We repeat the procedure for the halved range until it tends to the length equal 1.

Listing 2.2: Binary search in array

```
int binarySearch (int *a, int N, int key)
{
  int low=0, high=N-1, mid;
  while (low<=high)
  {
    mid=(low + high)/2;
    if (A[mid]<x)
      low=mid+1;
    else
      if (A[mid]>x)
        high=mid-1;
      else
        return mid;
  }
    return -1;
}
```

By repeating the halving principle, the average running time for search is $O(\log N)$. For large values of N, the binary search outperforms the sequential search. For instance, if N is 1000, an average sequential search requires 500 comparisons, where an average binary search, using the previous formula, requires $\lfloor \log N \rfloor - 1$ or only 8 iterations.

2.4. "TRAVERSAL" IN A BINARY TREE

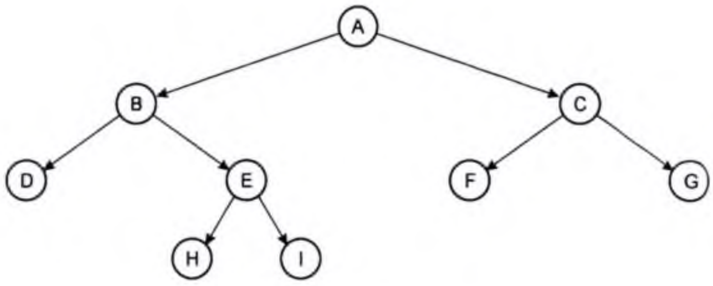
If we want to process all data from a tree, we should go to all nodes, take the data for each node and process them. The act of visiting all nodes of a tree is call *the tree traversal*. For a binary tree, at each node we may have 3 possibilities:

- go to left-child node (action code is L),
- go to right-child node (action code is R),
- and process the data at given node (action code is P).

Depending on the possibility we choose, we can code it with the order of the actions performed. For example, the code L-P-R is the order of traversal in which we start with the root node, visit the left subtree (L), process the data of the root node (P), and then visit the right subtree (R). Since the left and right subtrees are also the binary trees, the same procedure is used recursively while visiting the left and right subtrees.

It's easy to see that the possible orders in which a binary tree can be traversed are: L-R-P; L-P-R; P-L-R; R-P-L; R-L-P; P-R-L.

The order L-P-R is called as *inorder*; the order L-R-P is called as *postorder*; and the order P-L-R is called as *preorder*. The remaining three orders are not named since they are not often used. The example of orders in which the nodes are processed for a tree is given in Fig. 2.19, using *inorder*, *preorder* and *postorder* as shown.



Inorder: D-B-H-E-I-A-F-C-G

Preorder: A-B-D-E-H-I-C-F-G

Postorder: D-H-I-E-B-F-G-C-A

Figure 2.19. A binary tree along with its inorder, preorder and postorder

2.5. SEARCHING IN BINARY SORTED TREE

Similar to the situation of arrays, for the trees, it makes more sense to perform various operations like traveling, searching on the sorted trees.

2.5.1. Sorted tree traversal

The binary sorted tree property allows us to print out all the keys in sorted order by a simple recursive algorithm, called an *inorder tree walk*. This algorithm is so named because the key of the root of a subtree is printed between the values in its left subtree and those in its right subtree.

Analogically, a preorder tree walk will print the root before the values of all subtrees, and a *postorder tree walk* will print the root after the values in all of its subtrees.

To print all the elements in a binary search tree *p*, we call `Tree_Inorder(p)`.

If the tree is sorted, the `Tree_Inorder` will print out the elements in ascending order, where the R-P-L will print out the elements in descending order.

Listing 2.3: Tree traversal

```
void Tree_Inorder(int *p)
{
   if (p!=NULL)
   {
      Tree_Inorder(p->left);
      printf("%d\n", p->key);
      Tree_Inorder(p->right);
   }
}
void Tree_PreOrder(int *p)
{
   if (p!=NULL)
   {
      printf("%d\n", p->key);
      Tree_PreOrder(p->left);
      Tree_PreOrder(p->right);
   }
}
void Tree_PostOrder(int *p)
{
   if (p!= NULL)
   {
      Tree_PostOrder(p->left);
      Tree_PostOrder(p->right);
      printf("%d\n", p->key);
   }
}
```

2.5.2. Searching

In order to find if a key is presented or not, a similar procedure to the tree traversal is provided. During that traversal process, the key of each node is compared with the given key. If they are equal then the node address is returned. When the value is not found, an error code (usually a NULL value) is returned.

The C code for the search is as follows:

Listing 2.4: Tree search

```
int* Tree_Search(int* p, int skey)
{
  if (p!=NULL)
  {
    if (p->key==skey)
      return p;
    else
      if (p->key>skey)
        return Tree_Search(p->left, skey);
      else
        return Tree_Search(p->right, skey);
  }
  else
    return NULL;
}
```

2.6. GRAPH TRAVERSAL

Like a tree structure, a graph can be traversed either by using the ***depth-first traversal*** or ***breadth-first traversal***.

2.6.1. Depth-first traversal

When a graph is traversed by visiting the nodes in the forward (deeper) direction as long as possible, the traversal is called depth-first traversal. If in a given situation all moves or all successive states are equally likely, then we take the next step, creating, on the graph, presented as a shift to the next lower level, until we reach a level where there are no further possible transformations. If this level represents the correct solution we have completed the search process. The formal algorithm looks like this:

The traversal starts with the first vertex (that is, vertex 0), and marks it as "visited". It then considers one of the unvisited vertices adjacent to the actual vertex, visits that neighbor vertex and marks it also as "visited", then repeats the process by considering one of its unvisited adjacent vertices. The algorithm remembers the data of the next visited nodes and selected edges, which allows to undo the end of the last vertex one level higher. The process is stopped when all the vertices are “visited”.

The time complexity is geometric and the algorithm will continue to the maximum depth of the graph.

An C example program for depth-first traversal of a graph is presented below. It makes use of an array visited of *N* elements where *N* is the number of vertices of the graph, and the elements are Boolean. If *visited[i]=1* then it means that the ith vertex is visited. Initially we set *visited[i]=0*.

Listing 2.5: Graph Depth-First Traversal

$$/*\quad adj[i][j]=\begin{cases}0 & \textit{there is an edge from vertex } i \textit{ to vertex } j\\ 1 & \textit{otherwise}\end{cases}\qquad */$$

```
/* a function to visit the nodes in a depth-first order */
void dfs(int x,int visited[],int adj[][],int N)
{
   int j;
   visited[x]=1;
   printf("The node visited id %d\n",x);
   for(j=0;j<N;j++)
      if ((adj[x][j]==1) && (visited[j]==0))
         dfs(j,visited,adj,N);
}
void main()
{
/* initialization of N, adj matrix, visited matrix,...*/
   for(i=0;i<N;i++)
      visited[i]=0;
   for(i=0;i<N;i++)
      if(visited[i]==0)
         dfs(i,visited,adj,N);
```

2.6.2. Example

For example, we have a graph as shown in Fig. 2.20, the depth-first traversal starting at the vertex 0.

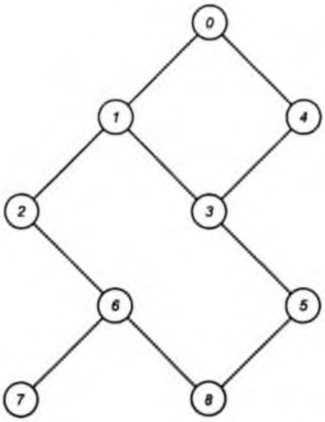


Figure 2.20. Graph G and its depth first traversals starting at vertex 0

The adjacency matrix for this graph is in following Tab. 2.4.

Table 2.4: The adjacency matrix for the graph in Fig. 2.20

From \ To	0	1	2	3	4	5	6	7	8
0	0	1	0	0	1	0	0	0	0
1	1	0	1	1	0	0	0	0	0
2	0	1	0	0	0	0	1	0	0
3	0	1	0	0	1	1	0	0	0
4	1	0	0	1	0	0	0	0	0
5	0	0	0	1	0	0	0	0	1
6	0	0	0	0	0	0	0	1	1
7	0	0	0	0	0	0	1	0	0
8	0	0	0	0	0	1	1	0	0

The results for depth-first traversal is:

```
0 1 2 6 7 8 5 3 4
```

2.6.3. Depth-Limited Search (DLS)

Depth-Limited Search (DLS) is a modified version of depth-first search that minimizes the depth that the search algorithm may go. The function has a parameter of depth, which is provided that the algorithm will not descend below (see Listing 2.6). Any nodes below that depth are omitted from the search to keep the algorithm from indefinitely looping.

Listing 2.6: DLS in a graph

```
void dls(struct graph_t *g_p, int root, int goal, int limit)
{
    int node, depth, to;
    struct stack *s_p;
    s_p=stack_create();
    stack_push(s_p,root); stack_push(s_p,0);
    while (!stack_isEmpty(s_p))
    {
        depth=stack_pop(s_p); node=stack_pop(s_p);
        printf("Node: %d at depth=%d.\n", node, depth);
        if (node==goal) break;
        if (depth<limit)
        {
            for (to=g_p->nodes-1;to>0;to--)
            {
                if (getEdge(g_p,node,to))
                {
                    stack_push(s_p,to); stack_push(s_p,depth+1);
                }
            }
        }
    }
    return;
}
```

While the algorithm does remove the possibility of infinitely looping in the graph, it also reduces the scope of the search. The algorithm can be complete if the search depth is that of the tree itself.

2.6.4. Breadth-First Traversal

The second basic search method, so called breadth-first traversal (BFS), is to traverse by visiting all the adjacent nodes/vertices of a node/vertex first and develop at each level all nodes of the next generation. The vertices are checked successively at a given level. If the solution is located not too deep, the search process is faster and easier to find. It means in Breadth-First Search (BFS), the nodes are visited in order of the distance from the root. The memory requirements, however, are much larger in this case, since you have to remember a large part of the tree, with all the nodes in the upper levels. If the solution lies at a deeper level, the extensive search procedure may be impracticable due to memory limitations. On the other hand, it can be applied even when there are infinitely long paths in the search tree. For this procedure to be effective the average degree of branching should not be too large.

For example, for a graph in which the breadth-first traversal starts at vertex 0, visits to the nodes take place in the order shown in Fig. 2.21.

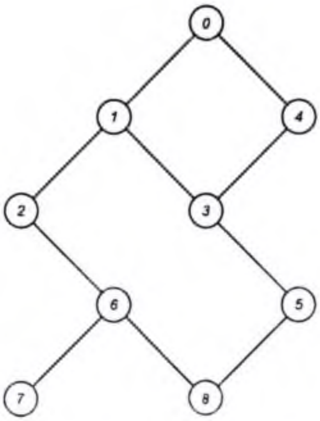


Figure 2.21. Breadth-first traversal of graph G starting at vertex 0: 0-1-2-3-4-5-6-7-8

An C example program for breadth-first traversal of a graph is below. The program uses also the visited array like in the DFS example. The program also makes use of a queue and the procedures *queue_add* and *queue_delete* for adding a vertex to the queue and for deleting the vertex from the queue, respectively. Initially, we set *visited[i]=0* for all vertices.

Listing 2.7: BFS search in the tree

```
/* Use the struct node* from Listing 2.1
```

$$adj[i][j]=\begin{cases}0 & \text{there is an edge from vertex } i \text{ to vertex } j\\ 1 & \text{otherwise}\end{cases}$$

```
Functions to a queue from Listing 2.1:
   struct node *queue_add(struct node *p,int val);
   struct node *queue_delete(struct node *p,int *val);
*/
void bfs(int adj[][], int x, int visited[], int N, struct node
**p)
{
   int y,j,k;
   *p=queue_add(*p,x);
   do{
      *p=queue_delete(*p,&y);
      if(visited[y]==0)
      {
         printf("\nNode visited=%d\t",y);
         visited[y]=1;
         for(j=0;j<N;j++)
         if((adj[y][j]==1)&&(visited[j]==0))
               *p=queue_add(*p,j);
      }
   }
   while((*p)!=NULL);
}
void main )
{
/*...
initialization of N, adj matrix, visited matrix,
...*/
```

```
    struct node *start=NULL;
    for(i=0; i<N; i++)
      visited[i]=0;
    for(i=0; i<N; i++)
      if(visited[i]==0)
        bfs(adj,i,visited,N,&start);
  }
```

2.6.5. Iterative Deepening Search (IDS)

Iterative Deepening Search (IDS) is a modification of DLS and also includes the features of breadth-first search. IDS performs DLS searches with increased depths until the goal is found.

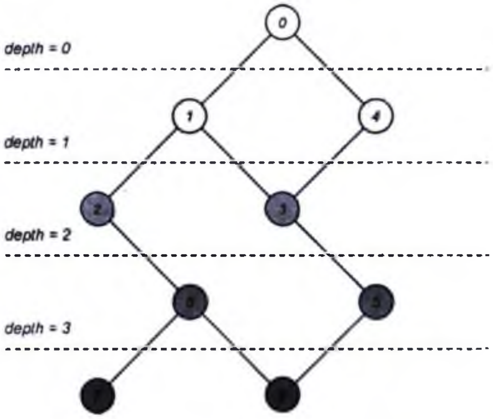


Figure 2.22. Search order for a graph (from Fig. 2.20) using depth-limited search: depth=1: 0-1-4; depth=2: 0-1-2-3-4; depth=3: 0-1-2-6-3-5-4

The depth begins at 1, and is increased until the target state is found, or no further nodes can be found to be added to the list (see Fig. 2.22). Following is the modified code for IDS:

Listing 2.8: The iterative deepening-search algorithm

```
  int ids(graph_t *g_p, int root, int goal, int limit)
  {
    int node, depth;
    int to;
    struct node *s_p;
    s_p=stack_create();
    stack_push(s_p,root); stack_push(s_p,0);
    while(!stack_isEmpty(s_p))
    {
      depth=stack_pop(s_p); node=stack_pop(s_p);
      printf("Node: %d at depth=%d.\n", node, depth);
      if (node==goal) return 1;
      if (depth<limit)
      {
        for (to=g_p->nodes-1;to>0;to--)
          if (getEdge(g_p,node,to))
          {
            stack_push(s_p,to); stack_push(s_p,depth+1);
          }
      }
    }
    return 0;
  }
  int main()
  {
    /* ... */
    struct graph_t *g_p;
    int status, depth;
    init_graph(g_p);
    depth=1;
    while (1)
    {
      status=ids(g_p,0,MAX_DEPTH,depth)
      if (status==1) break;
      else depth++;
    }
    destroyGraph(g_p);
    return 0;
```

IDS is better because it's not susceptible to loops like DLS. It also finds the goal nearest to the root node, as does the BFS algorithm. Because of this, when the depth of the solution is not known, IDS is a preferred algorithm.

2.6.6. Bidirectional Search

The Bidirectional Search algorithm is an extension of BFS, which performs two BFS at the same time. One BFS begins from the root node and the other begins from the goal node. When the two BFSs meet, a path can be reconstructed from the root through the meeting node to the goal.

2.6.7. Uniform-Cost Search (UCS)

One advantage of BFS is that it always finds the solution closest to the root. But for weighted trees/graphs, the shortest solution may not be the best, but a further solution with a lower path cost would be better (for example, see Figure 2.23). Uniform–Cost Search (UCS) can be applied to find the least-cost path through a graph by maintaining an ordered list of nodes in order of descending cost. This allows us to evaluate the least cost path first.

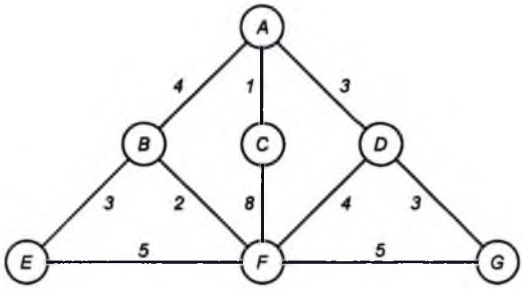


Figure 2.23. An example graph where choosing the lowest cost path for the first node (A->C) is not the best solution (A->B->F)

The algorithm for UCS uses the accumulated path cost and a priority queue to determine the path to evaluate (see Listing 2.9). The priority queue (sorted from with ascending costs) contains the nodes to be evaluated. As node's children are evaluated, we add their cost to the node with the accumulated sum of the current path. The updated node is then added to the queue, and when all children have been evaluated, the queue is sorted in order of ascending cost. When the first element in the priority queue is the goal node, then the best solution has been found.

LISTING 2.9: The Uniform-Cost Search algorithm.

```
void ucs(graph_t *g_p, int root, int goal)
{
   int node, cost, child_cost, to;
   struct node *q_p;
   q_p=queue_create(NMax);
   queue_add(q_p,root); queue_add(q_p,0);
   while(!queue_isEmpty(q_p))
   {
      queue_remove(q_p,&node); queue_remove(q_p,&cost);
      if (node==goal)
      {
         printf("cost %d\n", cost);
         return;
      }
      for (to=g_p->nodes-1;to>0;to--)
      {
         child_cost=getEdge(g_p,node,to);
         if (child_cost)
         {
            queue_add(q_p,to); queue_add(q_p,child_cost+cost);
         }
      }
   }
   queue_destroy(q_p);
   return;
}
int main()
{
   /* ... */
   struct graph_t *g_p;
   init_graph(g_p);
   ucs(g_p,'A','E');
   destroyGraph(g_p);
   return 0;
```

Table 2.5: Node evaluations and the development of the priority queue

Step	*Investigating Node*	*Priority Queue*				
1		A(0)				

2	A	C(1)
		A(0)
			D(3)
			A(0)
				B(4)
				A(0)		

3	C	D(3)
		A(0)
			B(4)
			A(0)
				F(9)
				C(1)
				A(0)		

4	D	B(4)
		A(0)
			F(7)
			D(3)
			A(0)
				F(9)
				C(1)
				A(0)
					G(6)
					D(3)
					A(0)	

5	B	F(7)
		D(3)
		A(0)
			F(6)
			B(4)
			A(0)
				F(9)
				C(1)
				A(0)
					G(6)
					D(3)
					A(0)
						E(7)
						B(4)
						A(0)

6	G	F(7)
		D(3)
		A(0)
			F(6)
			B(4)
			A(0)
				F(9)
				C(1)
				A(0)
					E(7)
					B(4)
					A(0)
						F(11)
						G(6)
						D(3)
						A(0)

7	E	F(7)
		D(3)
		A(0)
			F(6)
			B(4)
			A(0)
				F(9)
				C(1)
				A(0)
					F(11)
					G(6)
					D(3)
					A(0)
						F(12)
						E(7)
						B(4)
						A(0)

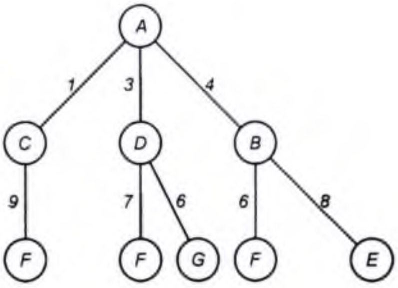


Figure 2.24. Illustrating the path cost through the graph

An example of the UCS algorithm is shown using our example graph in Figure 2.21. Table 2.5 shows the state of the priority queue as the nodes are evaluated. The search of the graph is shown in Figure 2.24, which gives the path cost at each edge of the graph. The path costs shown for the target node (F) allow us to check the best path (with lowest cost) through the graph.

2.7. SELECTED INFORMED SEARCH ALGORITHMS

We presented above the uninformed search methods such as DFS and BFS. These methods operate in a brute-force way. This section will present some selected informed search methods, including best-first search, A* search.

Avoiding blind search is the main task of the informed methods. The search procedure is modified by introducing functions to assess the suitability of individual move and nodes on a given level according to their "attractiveness". Such functions are called the heuristic function. Thev evaluate how far a state is from the target state. The algorithm for this procedure looks like this:

- Review starting state; if it is not the target state then create new state:
    - If the new state is the end state, stop.
    - If the new state is closer to the state of the target, accept it as current; otherwise ignore it.
- If you can no longer create new states, stop.

So this is a deep search directed by the goal function.

2.7.1. Best-First Search (Best-Fs)

In Best-First search, the search space is defined using a function. Nodes to be evaluated are kept on an OPEN list as a priority queue, such that nodes can be unloaded in order of their evaluation function. The evaluation function $f(v)$ is a sum of two parts, which are the heuristic function $h(v)$ and the estimated cost $g(v)$, where:

$$f(v) = g(v) + h(v) \tag{2.1}$$

The OPEN list is then built in order of $f(v)$. This makes best-first search chooses the best local opportunity during the search.

The formal "best first" algorithm with the evaluation function $f()$ evaluating nodes is as follows:

- Create a one-element list consisting of a root
- Until the list is empty or the target node is not reached:
- Find node v for which the function $f(v)$ assumes a minimum expansion to create all child nodes.
- For each child node v':
    - Evaluate the node with the function $f(v')$,
    - If the node appeared earlier, leave only the one with the better value $f(v')$.
- If the solution is found then signal the success and give the path leading to the goal, otherwise announce the failure.

The method does not guarantee an optimal solution, and the need to check for a node earlier increases the memory requirements and complexity of the computation.

Let's now consider the example of the Best-First search algorithm in the context of a large search space. The N-queens problem is a search problem where the desired result is an $N \times N$ board with N queens such that no queen threatens another (see example on Fig. 2.25 where $N = 4$).

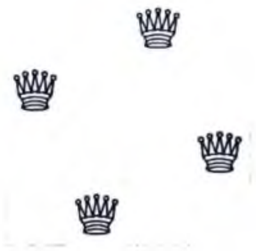


Figure 2.25. A goal state of N-Queens board (where N=4)

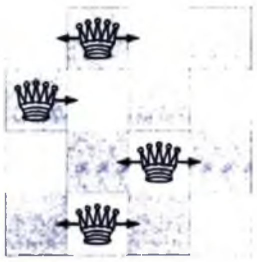


Figure 2.26. Possible moves for a state of the board for N=4

We'll simplify the problem by limiting the possible moves of each queen to each row on the board. For example, the queen at the top of Fig. 2.26 can move left or right, but the queen in the second row can only move right. Figure 2.26 shows the possible moves of all the 4 queens.

Given a state (a board with 4 queens), we can identify the child states for this state by creating a new board for each of the possible queen position changes, given horizontal movement only. For Fig. 2.26, this board configuration has six possible new child states.

a) Best-First Search Implementation

Listing 2.10: Best-First Search Implementation

```
int best_first(int startnode, int targetnode)
{
   struct node *closed, *open;
   closed=list_create();
   open=list_create();
```

```
    list_add(open, startnode);
    while !list_isEmpty(open)
    { // find node with minimum f
      current=list_getMin(open);
      if (current==goal)
      { // display the path to goal
        list_printSolution(current);
        return 1;
      }
      list_add(closed, current);
      list_delete(open, current);
      for <neighbor in list_isNeighbors(current)>
        if !list_isMember(closed, neighbor)
          list_add(open,neighbor);
      return -1;
  end;
```

The demonstration here shows a part of the tree with the root node and the solution found at depth two. A condensed version of this run is shown in Figure 2.27, where the functions used are: $g()$ – level of the node, $h()$ – number of pairs of queens attacking each other.

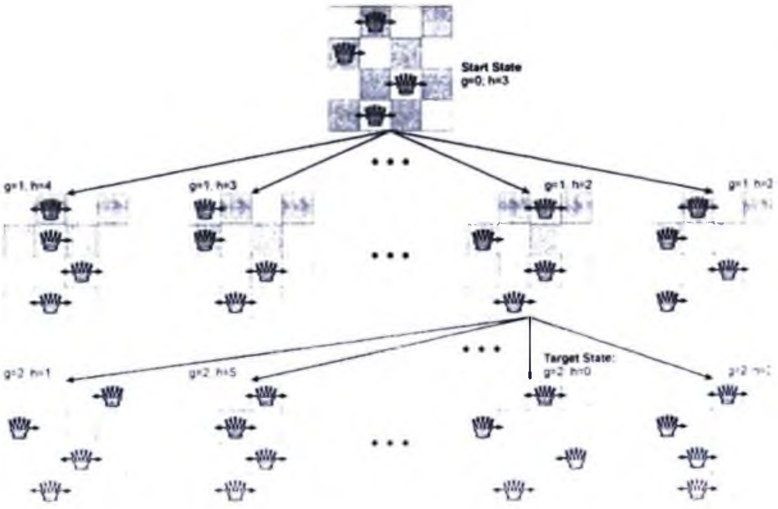


Figure 2.27. Graphical (condensed) view of the search tree

b) Variants of Best-First Search

i) Greedy algorithm

The simplest variant of the first strategy is the greedy strategy where only the heuristic function $h : V \rightarrow \mathbb{R}$ is used to evaluate the attractiveness of a node. Since $h()$ representing the closeness to the goal is the only factor of used to determine which node to select next, the algorithm is called "greedy". Each node assigns the estimated cost of the shortest path from that node to the target node v_g. Any $h()$ function can be assumed in principle as long as, for each node $v \in V$, it satisfies the following conditions:

- $h(v) \geq 0,$
- $h(v) \leq h^*(v)$
- $h(v_g) = 0.$

where $h^*(v)$ is the real cost of the path from node v to the target node.

For example, in the search for the shortest path on the map, the role of such a function can be the distance between the node currently under development and the target node.

ii) The beam search

Another variant of best-first search is beam-search, which uses the heuristic $f(v) = h(v)$ and it keeps only a set of the best candidate nodes for expansion. This makes the variant more memory efficient, but includes the new risks that optimal nodes can be discarded and lost.

2.7.2. A* SEARCH

In 1968, Peter Hart, Nils Nilsson and Bertram Raphael proposed an extremely efficient best-first search algorithm, the A* algorithm. It finds the shortest path between the start node and any of the target nodes using the same idea of evaluation function $f()$ from Eq. (2.1) that represents the estimated cost of the solution containing the node v. This function is calculated as:

$$f(v) = \alpha \cdot g(v) + \beta \cdot h(v) \tag{2.2}$$

where $g(v)$ is the real cost of the path from the starting node v_0 to the current node v, and $h(v)$ is the estimated cost (a heuristic) of the best path from v to v_g, α and β are the weights for tuning the effects of the functions $g()$ and $h()$ on the $f()$.

As in the case of the greedy strategy, $h(v)$ cannot overestimate (so called admissible) the real cost $h^*(v)$ of the path connecting nodes v and v_g This requires also that the heuristic be monotonic, which means that the cost never decreases over the path. Or in other words, function $g(v)$ monotonically increases, while $h(v)$ monotonically decreases.

It is also required that heuristics $h()$ be a monotonic function in the sense that its values decrease (or grow) monotonically along each path connecting nodes v_0 and v_g. In this case, the cost $f(v)$ does not decrease (or increase) along each path or for each child node v . The operation of the algorithm A* represents the pseudocode in Listing 2.11.

LISTING 2.11: Pseudo-code for the A* search algorithm

```
void A*()
{
   while (openList is not empty)
   {
      cur_state=getListBest(&openList);
      putList(closedList, cur_state);
      /* Do we have a solution? */
      if (cur_state is goal state)
      {
         showSolution(cur_state);
         return;
      }
```

```
        else
        {
           if (cur_state is too deep) continue;
           for (available child states)
           {
              child_state=getNextChild(cur_state);
              if (child_state is valid)
              {
                 evaluate_h_g_f_value(child_state);
                 /* New child candidate on the OPEN list? */
                 if (onOPENList(child_state))
                 { //...
                    update_g_value(child_state);
                    //...
                 }
                 else
                 { //...
                    putList(openList, child_state);
                    //...
                 }
              }
           }
        }
     }
     return;
  }
```

Note in the flow from Listing 2.11 that once we find the best node from the OPEN list, we expand all of the child nodes from that best node. If the new child nodes are not found on either the OPEN or CLOSED lists, they are added as new nodes. If the new node is on the CLOSED list, we discard it, if the new node is on the OPEN list, but the new node has a better $g()$ value, we update the value of $g()$ for the node record on the OPEN list. By re-evaluating the nodes on the OPEN list, and replacing them when cost functions permit, we allow better paths to emerge from the state space

a) Using A* Search with the Eight Puzzle

We'll apply A* here to the Eight Puzzle with starting node is shown on Fig. 2.6a. From that state, since the empty cell is at the corner then there are only two legal moves that are possible. The '1' cell can move left, and the '6' cell can move down. Figure 2.28 shows the moves possible from the initial puzzle configuration (in Fig. 2.6a) to depth two of this state space tree.

This implementation uses the simple, but effective, tiles-out-of-place heuristic, in which the cost $g(v)$ from the root to the current node is the depth of the tree and the estimated cost $h(v)$ to the goal node (excluding the blank) is the number of misplaced cells. These heuristics are given in the tree in Fig. 2.28. From the root node, only two moves are possible. At the bottom of this tree, you can see that the cost function has decreased, indicating that these board configurations are likely candidates to explore next.

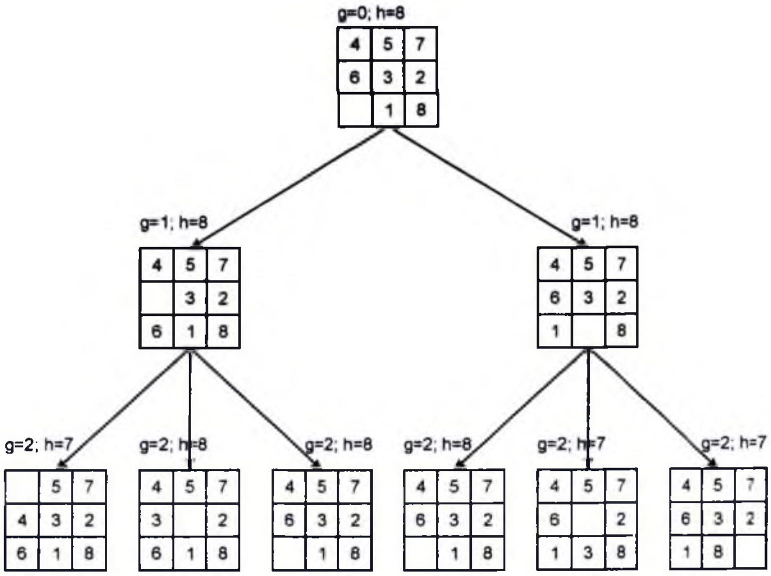


Figure 2.28. Eight Puzzle tree ending at depth two, illustrating the cost functions

b) A* Search Implementation

A* algorithm was implemented in the function `astar()` following the draft in the Listing 2.11. Let's start with the evaluation function which calculates the estimated cost from the current node to the goal (as the number of cells out of place), see Listing 2.12. The function simply enumerates the 3×3 board as a one-dimensional vector, incrementing a score value whenever a cell is present in a position it should not be in.

LISTING 2.12: The h(n) estimating cost metric in Eight Puzzle

```
double h(board_t *board_p)
{
  int i, score=0;;
  const int test[MAX_BOARD-1]={1,2,3,4,5,6,7,8};
  for (i=0;i<MAX_BOARD-1;i++)
     score+=(board_p->array[i]!=test[i]);
  return (double)score;
}
```

The `astar` function is shown in Listing 2.13. Prior to calling this function, we've placed the input board configuration onto the OPEN list. Then we work through the OPEN list, retrieving the best node (with the least $f()$ value using `getListBest`) and immediately place it on the CLOSED list. We check to see if this node is the solution, and if so, we report the success of the search with the results. To minimize searching too deeply in the tree, we defined a MAX_DEPTH constant and will stop the search if then MAX_DEPTH level has been reached.

The next step is to list out the possible moves from this state. The `getChildBoard` function is used to return an child node (or a next move). If a move isn't possible, then a NULL is returned and it's ignored.

With a new child node, we first check to see if it's already been evaluated (if it's on the CLOSED list). If it is, then we're to destroy this node and continue (to get all the child nodes for the current board configuration). If we've not seen this node (a particular board configuration) before, we calculate the heuristics for the node. The node's depth in the tree is equal the parent's depth plus one. Then we call $h()$ to get the tiles-out-of-place metric, which will act as our $h()$ value (cost from the root node to this node). And the $f()$ value is initialized with Eq. (2.2) with $\alpha = 1$ and $\beta = 2$ to give $h()$ value higher influence to the evaluation of $f()$.

With the $f()$ value calculated, we check to see if the node is on the OPEN list. If it is, we compare their $f()$ values. If the node on the OPEN list has a worse $f()$ value, the node on the OPEN list is discarded and the new child node takes its place (setting the predecessor link to the parent, so we know how we got to this node). If the node on the OPEN list has a better $f()$ value, then the node on the OPEN list remains on the open list and the new child is discarded.

Finally, if the new child node exists on neither the CLOSED or OPEN list, it's a new node that we've yet to see. It's simply added to the OPEN list, and the process continues.

This algorithm continues until either one of two events occur. If the OPEN list becomes empty, then no solution was found and the algorithm exits. If the solution is found, then `showSolution` is called, and the nodes linked together via the predecessor links are enumerated to show the solution from the initial node to the goal node.

LISTING 2.13: The A* algorithm

```
void astar(void)
{
  board_t *cur_board_p, *child_p, *temp;
  int i;
  /* While items are on the open list */
  while (listCount(&openList_p))
  {
    /* Get the current best board on the open list */
    cur_board_p=getListBest(&openList_p);
    putList(&closedList_p,cur_board_p);
    /* Do we have a solution? */
    if (cur_board_p->h==0.0)
    {
      showSolution(cur_board_p);
      return;
    }
    else
    {
      /* Don't go too deep */
      if (cur_board_p->depth>MAX_DEPTH) continue;
      /* Enumerate adjacent states */
      for (i=0;i<4;i++)
      {
        child_p=getChildBoard(cur_board_p,i);
        if (child_p!=NULL)
        {
          if (onList(&closedList_p,child_p->array, NULL))
          {
            nodeFree(child_p);
            continue;
          }
          child_p->depth=cur_board_p->depth+1;
          child_p->h=evaluateBoard(child_p);
          child_p->g=(double)child_p->depth;
          child_p->f=(child_p->g*ALPHA)+(child_p->h*BETA);
          /* New child board on the open list? */
          if (onList(&openList_p,child_p->array, NULL))
          {
            temp=getList(&openList_p, child_p->array);
            if (temp->g<child_p->g)
            {
              nodeFree(child_p);
              putList(&openList_p, temp);
              continue;
            }
            nodeFree(temp);
          }
          else
          { /* Child board is new or better than a
               previous board. Place on the open list.*/
```

```
                    child_p->pred=cur_board_p;
                    putList(&openList_p, child_p);
                }
            }
        }
    }
}
return;
}
```

c) Eight Puzzle Demonstration with A*

In the implementation, the tiles are labeled A-H with a '_' used to denote the blank tile. Upon execution, once the solution is found, the path taken from the initial board to the goal is enumerated. This is shown below in Listing 2.14, minimized for space.

LISTING 2.14: A sample run of the A* program to solve the Eight Puzzle.

```
2 3 6
1 0 4
7 5 8

2 3 6
1 4 0
7 5 8

2 3 0
1 4 6
7 5 8

2 0 3
1 4 6
7 5 8

0 2 3
1 4 6
7 5 8
```

```
1 2 3
0 4 6
7 5 8

1 2 3
4 0 6
7 5 8

1 2 3
4 5 6
7 0 8

1 2 3
4 5 6
7 8 0

Solution steps 8
```

This is an illustration of the general fact that formal correctness does not always go hand in hand with good efficiency computation.

2.8. CHAPTER SUMMARY

As a Chapter conclusion, search algorithms play special roles and are of extreme interest in AI because many problems can be presented as search problems in a states space. Both uninformed and informed search algorithms are popular due to their advantages and disadvantages but in AI games, the informed search has wider application since it may help to reduce the searching area (or possibilities) in complex problems

Chapter 3: PROLOG – A Logical Programming Language

This chapter describes one of the well-known logical programming languages we can use to solve problems related to artificial intelligence, which is PROLOG. The language in its original form is quite old but its concepts are still present today and are used in newer information system solutions. The algorithm does not necessarily have to be written in one of the imperative programming languages (i.e. C, Java, Pascal), but it can often be done more easily and faster using the declarative programming language, i.e. PROLOG for logic programming or LISP for functional programming. The ultimate goal is to change the way we think about the problem and the ability to solve it in different than the classical way we do with other popular programming languages such as C, Pascal, FORTRAN, Java,... Although computers were used mostly for performing numerical calculations, it was also found out that the numbers could represent not only pure values but also features of arbitrary objects. Operations on such features could be used to represent rules for creating, relating or manipulating symbols and rules.

3.1. PROLOG LANGUAGE

The name of the language itself - PROLOG - already has information about its purpose. The word PROLOG comes from the formulation of 'PROgrammation en LOGique', which in French means logical programming. PROLOG was created in 1971 by Alain Colmeraurer and Phillipe Roussel. Colmerauer studied the possibility of processing natural language, whose semantics were represented by logical expressions. For this reason, the theoretical basis of PROLOG is the first-order predicate calculus. Even now it is preferred in many programs related to mathematics logic, natural language processing, solving symbolic system of equations,..

The most important and often the most surprising and amazing thing about PROLOG is that writing a program in PROLOG is not based on describing an algorithm! In PROLOG in particular and in AI-based languages programming, the main implementation effort is the problem specification process.

We were told that before we start writing the program we have to design the appropriate algorithm. Once we learned it and took it for granted, it suddenly turns out that it does not have to be. Unfortunately, it is very difficult to stop thinking algorithmically about the problem. This is something that is different in PROLOG. Here, instead of describing an algorithm, we describe the ***objects*** related to the problem and the ***relations*** between these objects. Hence PROLOG is often referred to as descriptive and declarative language. This means that by implementing a solution to a problem we do not provide a way to solve it (as is the case in imperative programming languages such as C or Java), but we define what it involves using the facts and rules. The role of PROLOG is to infer the solution based on the information we provide.

3.2. OBJECTS AND RELATIONSHIPS

Programming in PROLOG bases on "defining" objects and determining their binding relationships. PROLOG programs are *declarative* collections of statements about a problem. In the program we do not specify how a result is to be computed, this is quite different from imperative and even functional programming, in which the focus is on defining *how* a result is to be computed. Using PROLOG, programming can be done at a very abstract level quite close to the formal specification of a problem. Main objects in PROLOG are defined as facts, axioms and logical rules for deducing new facts. However, the way PROLOG defines and uses its object is very different from the way the classical object-oriented programming languages do

Although we are dealing with PROLOG, to make ourselves aware of what is so different from other languages, let us take a little reminder about what is a "classical" object, known from such languages as C ++ and Java. In this perspective, the object is a fundamental concept that is part of the paradigm of objectivity in software analysis and design, and in programming. Its concept is to facilitate digital representation of real objects.

An object in the sense of PROLOG is something we can call 'an existence'. We do not define what it consists of and what it can do with it (like we do in OO programs), but what it is. In addition, for each object we define the relations that this object is subjected to. With the help of objects we describe an interesting piece of the problem. The action of the PROLOG program reveals the possibility of asking questions related to the previously described world.

The simplest way to describe a problem is to give facts related to it, such as:

```
bigger(motorbike, bike).
bigger(car,motorbike).
bigger(train,car).
```

The above facts state that: the motorbike is bigger than the bike, the car is bigger than the motorbike and the train is bigger than the car,...

With these three facts written in PROLOG instructions, we have defined several objects (*bike*, *motorbike*, *car*, *train*) and linked them with a *bigger* relation expressing which objects are bigger than others. The important thing is that we still do not know the sizes or the dimensions of any object. The objects simply do not have yet (they don't have to) the information about their sizes.

After having those three facts, we can start ask questions related to the reality it describes. Please note that in this chapter, we use the text "?-" to indicate the prompt of the PROLOG system where we will input our predicates since it is the prompt used by one of the most popular PROLOG simulator SWI-PROLOG [WebProlog], the text following later contain the result(s) returned by the simulator.

```
?- bigger(car,motorbike).
Yes
```

In this way we get an affirmative answer to the question: Is the car bigger than the motorbike? Being more precise, the question is: Is it known that car is bigger than a motorbike? This is a significant distinction, because when we ask

```
?- bigger(car,bike).
No
```

It should be read as: Nothing is known about if the car is bigger than a bike. This does not mean that it is not true for sure, it's just PROLOG does not find a fact saying a car is bigger than a bike. Such an interpretation is more appropriate, as another example shows

```
?- bigger(train,motorbike).
No
```

From the given facts, by the transitivity and knowledge of the notion of weight we can infer that the answer should be affirmative, according to the reasoning. Because it is true that *bigger(train,car)* and *bigger(car, motorbike)* and *bigger(motorbike,bike)*

or

```
train > car > motorbike > bike
```

So it is true that

```
bigger(train,motorbike)
```

and

```
bigger(car,bike)
```

But the PROLOG does not know yet that in relation to *bigger* relationships the transitivity of the relationships may apply, and therefore, in the light of known facts and relationships, it gives a negative answer to *bigger (train,motorbike)* and *bigger(car,bike)*. This is how we come to the situation when we need to tell PROLOG about certain relationships, or we should define rules.

Let us deal with the transitive relationships and create the rules for them. In mathematics, the relation (binary) of R on set domain D, which we write

$R \subset D^2$ is transitive, if for all elements a, b, c ∈ A if elements (a, b) are in relation R and elements (b, c) are in the relation R, then the elements (a, c) are also in the relation R

$$(a\ R\ b)\ \&\ (b\ R\ c) \Rightarrow a\ R\ c$$

Examples of such relationships can be given, for example, relations of sets inclusion, or relation to being siblings. Transitive is not a relationship of difference, a relationship to be a parent or a friend. In the case of the relationship *bigger* we consider, just add such a rule:

```
bigger(X,Y):- bigger(X,Z), bigger(Z,Y).
```

In the above rule, the symbol '`:-`' means if (if the right side is the left one) and the comma symbol '`,`' acts as the logical AND operator. Symbols X, Y, and Z are variable names (In PROLOG, the variable name starts with a capital letter).

Putting facts and rules into one file we can now test the operation of the program. So let's make a test of elementary facts:

```
?- bigger(car,motorbike).
More? [ENTER]
yes
```

For now we skip the meaning of the message *'More?' [ENTER]* when it appears. So it's time for a entering a new question:

```
?- bigger(car,bike).
More? [ENTER]
yes
```

This time we got the answer as expected. But we can learn a lot more, asking, for example, what objects are bigger than motorbike:

```
?- bigger(motorbike,X).
X=car [;]
X=train [;]
ERROR: Out of local stack
```

When you see each option, PROLOG is waiting for your decision. Pressing ENTER means ending the search for alternative answers, semicolon '`;`' means continuing the search. The bad message that appears at the end of this page is not an error but should be read as '*Nothing else is found*'. The semicolon ' ; ', as intuitive, reads as OR (OR). Logical operators can also be used when formulating a query, for example if there are objects X, Y such that the train is bigger than X and at the same time X is bigger than Y:

```
?- bigger(train,X),bigger(X,Y).
X = car,
Y = motorbike [;]
X = car,
Y = bike [;]
X = motorbike,
Y = bike [ENTER]
```

3.3. TERMS

The PROLOG program consists of terms. There are four types of terms: atoms, numbers, variables, and compound terms. Atoms and numbers are commonly referred to as atomic terms or constants. A set of atoms and compound terms is also called a set of predicates. Each term is written as a string of characters from the following four categories

- capital letters: A-Z
- lower case letters: a-z
- digits: 0-9
- special characters: % + - * / \ ~ ^ < > : . ? @ # $ & _

a) Atoms

Atom is a string of characters

- upper and lower case letters, numbers, and underscores, with the requirement that the first character must be a lowercase letter, for example: `a`, `abc`, `aBc`, `abc_xyz`,...
- any string included in the apostrophe, like '`This is an atom`'
- Symbols such as ' `?-` ' or ' `:-` '

b) Numbers

The basic number type in PROLOG is integer, for example *5*, -7,... Some simulator versions (for example SWI-PROLOG) also support real numbers, for example *12.3*, *45e4*, *56e-2*,...

c) Variables

The variable is a string of letters, uppercase and lowercase letters, digits, and underscores, provided that the first character must be a capital letter or an underscore, such as *X*, *Xyz*, *_xyz*, *XY_1_2*, _

The last example, a single underscore, is the so-called anonymous variable. We use it whenever we are only interested in something or if it is true, but we are not interested in its value (or anything). Please note, that if in one clause there are multiple anonymous variables then they (the anonymous variables) can have different assigned values.

```
?- a(3,6,9)=a(B,C,D).
B=3,
C=6,
D=9
?- a(3,6,9)=a(B,C,C).
No
?- a(3,6,9)=a(_,_,_).
Yes
```

d) Compound terms

A compound term, or recursive structure, is an object composed of other objects, such as atoms, numbers, variables, and other compound terms. Compound terms have the form

```
f(arg_1, ..., arg_n)
```

where *arg_1*,..., *arg_n* are terms, and *f* is atom (name of relationship). Using the ability to nest other terms in complex terms, we can better describe the problem we are having. The facts

```
has(alpha, computer).
has(beta, computer).
```

only allow to say that the objects Alpha and Beta are related to the `computer` object, that means, in normal language, Alpha and Beta have computers. It is difficult to say what are the computers and if it is not by any chance the same computer. Writing these facts differently

```
has(alpha,computer(desktop,winOS)).
has(beta, computer(laptop, macOS)).
hasComputer(X):- has(X,computer(_,_)).
```

We still have the opportunity to find out if they both have a computer, but if we want, we can ask for something more detailed, i.e. its type and OS.

```
?- hasComputer(alpha).
yes
?- has(alpha,computer(X,Y)).
X = desktop,
Y = winOS
```

3.4. CLAUSES, PROGRAMS AND QUERIES

PROLOG consists essentially of two types of "programming designs". These are the *facts* and *rules* that can be called by same term *clauses*.

a) Facts

The fact is a predicate ending in a dot ‘ . ’, for example:

```
He has (Computer, winOS).
This is a laptop.
```

An intuitively understood term is a statement about the objects in question, which we undoubtedly consider to be true.

b) Rules

Each rule consists of two parts: head and body, which are separated by the operator ‘ :- ’ (the head is on the left of this operator) and end with a dot ‘ . ’.

The head is one predicate, while the body is one or more predicates separated by commas ‘ , ’ and/or semicolons ‘ ; ’. A comma or semicolon plays a role of logical operators, mentioned above respectively an AND and an OR. For this reason, it is acceptable to use round brackets as a grouping element.

```
bigger(X,Y) :- longer(X,Y), heavier(X,Y).
notBigger(X,Y):- notLonger(X,Y); notHeavier(X,Y).
a(X,Y) :- b(X,Y);(c(X,Y),d(X,Y)).
```

The defined term *rule* is the set of conditions (body) that must be met for the purpose (the head) to be fulfilled (fulfillment in this case means assignment/conclude a given item of logical value TRUE).

c) Program

The PROLOG program is an ordered set of clauses. The word "orderly" in this case is important because the order of the clauses in the source file is significant - the clauses in are realized in order of appearance. We can see that on the following example comparing the results of the operation of two programs:

```
Program 1                   Program 2
has(alpha,computer).        has(beta,computer).
has(beta,computer).         has(alpha,computer).
?- has(X,computer).         ?- has(X,computer).
X=alpha;                    X=beta;
X=beta                      X=alpha;
```

d) Queries

The query has the same structure as the rule body and it also ends with a dot. The query, which is typed in at the prompt (in SWI-PROLOG the prompt string is `?-`) is confirmed by pressing the [ENTER] key. This is understood by the PROLOG as an order to search and check if on the basis of the given facts and rules it is possible to prove the truth of the predicates forming the query and the truth of the query itself. Answer '*Yes*' means there is a chain of transformations and substitutions that can show the truth of the query. Answer '*No*' means that based on the knowledge held in the form of facts and rules, it can not be shown. This does not mean that it is not.

A query that does not contain any variable,

```
a(b,c).
```

is called a *ground* query. The expected answer for such a query is '`Yes`' or '`No`'. Theoretically, queries of this type are much easier to verify, since it is often enough to find a valid fact.

For program like below:

```
a(1,2).
a(3,4).
```

an example elementary query will be:

```
?- a(3,4).
yes
```

Queries containing variables are referred to as *non-ground* queries. In this case, finding a response can take time. The answer expected for such a query is the proper substitution for the variables. Let's look at a simple program that multiplies two natural numbers

```
mul(0,Y,0).
mul(1,Y,Y).
mul(X,Y,Z):- X>1,X1 is X-1, mul(X1,Y,Z1),Z is Z1+Y.
```

The formula `X1 is X-1` in the rule can be understand as follows: `X1` is assigned the result of the `X-1` operation. The above program is based on the recursive definition of multiplication. The product of two numbers x and y, for x> 1 we define recursively as

$x \cdot y = y + (x-1) \cdot y.$

For $x = 0$ or $x = 1$ we have

$x \cdot y = 0,\ for\ x = 0$ and $x \cdot y = y,\ for\ x = 1.$

The effects are consistent with expectations

```
?- mul(1,2,X).
X=2 [ENTER]
yes
?- mul(0,2,X).
X=0 [ENTER]
yes
?- mul(2,3,X).
```

```
X=6 [ENTER]
Yes
?- mul(2,3,6).
More? [ENTER]
Yes
```

3.5. QUERY EVALUATION

3.5.1. Expressions Matching

Two terms are matched if they are identical or can be identical by substituting the variable values.

It is important that substitutions for variables are identical throughout an expression. The only exception is the anonymous variable that may have different values in different places. For example, two terms

```
myTerm(a,b)
myTerm(a,X)
```

are fit together because the substitution for the *X* atomic variable *b* makes these terms identical

```
?- myTerm(a,b)=myTerm(a,X).
X=b
```

On the other hand, they will not fit the terms in the following examples

```
?- myTerm(a,b)=myTerm(X,X).
No
?- myTerm(a,X)=myTerm(X,b).
No
```

Because *X* can not have both *a* and *b* values set at the same time, replacing *X* with an anonymous variable '_' causes the expression to become matched.

```
?- myTerm(a,b)=myTerm(a,_).
yes
?- myTerm(a,_)=myTerm(_,b).
yes
```

Matching a term to itself (self-matching) is not a trivial process at all, as we can see by looking at the example below.

```
?- a(X,b)=a(f(Y),Y),b(X)=b(f(Z)).
X = f(b),
Y = b,
Z = b.
```

The matching process is also called *unification*

3.5.2. Goal execution

Query entering into PROLOG interface triggers a process to check if there is a chain of substitutions and transformations that can be used to assign a logical value TRUE to the actual query. PROLOG's inference process consists of two basic components: a search strategy and a unifier. The search strategy is used to search through the facts and rules database while the unifier is used for matching the results and returns the bindings that make an expression true. Searching for such proof is called *goal execution*. Each predicate in the query becomes the goal that PROLOG is trying to fulfill one by one. The search strategy is used to traverse the search space spanned by the facts and rules of a PROLOG program. PROLOG uses a *top-down*, *depth-first* search strategy.

If identical variables occur on several bases, then, as already described, they will get identical substitutions. If a query is specified, then it may either match a fact or a rule. In case of a rule, PROLOG first tries to match the head of that rule, if the target matches the rule head, then there are appropriate substitutions within the rule, and thus we get a new target, replacing the starting one. If this target consists of several predicates, then it is divided into several threads, each of them being treated as a starting target. The process of replacing an expression by another expression is called a *resolution* and can be described with the following algorithm

1. As long as the query is not empty, execute
   a) Choose next term from the query
   b) Find a fact or a rule unifying with the term. If there is no such fact or rule, return FAIL, otherwise:
      i. If a fact found, remove it from the query.
      ii. If a rule is found, replace the term body with the rule.

2. Return SUCCESS.

Applying unification and resolution allows to prove the truth or its lack, according to the following principles:

1. If the target is an empty set, return TRUE.
2. If there are no rules head nor facts unifying with the expression under consideration, return FALSE.
3. In the case of failure (returned FALSE), return to a place where by applying the resolution you can get another expression and retry the process. This principle is called ***backtracking***.

In this description step 3 is particularly important, which in some way may cause the restart of the entire algorithm. This means that PROLOG in point 1-b) memorizes the unification occurrences and, at the appropriate moment, is able to search for further unifications occurring after the selected one.

We apply the above rules until we explore all the possibilities by choosing the next available expression in each step. This gives us the opportunity to find several different proofs.

3.6. LISTS

3.6.1. Syntax

In PROLOG, list is an ordered chain of elements of any length. As part of the list can be used any valid PROLOG term, i.e. atom, number, variable, also term composed of a different list. We place them between the square brackets '*[*' and '*]*' and separate them with a comma '*,*',

```
[A, X, a(b,X), [b,c,d], [], 123]
```

An empty list is written as a pair of empty parentheses

```
[]
```

3.6.2. The head and tail of a list

Lists are always processed by dividing them into two (logical) parts: the head, which is the first element of the list, and the tail, which is all that remained of the list after removing the head.

The empty list has no head or tail. The head of a list containing only one element is that element, while the tail is an empty list.

To separate from the list its head and tail the symbol ‘ | ’ is used. Elements to the left of the symbol ‘ | ’ refer to the head or to the first few elements of the list, while the right indicates the tail.

```
?- []=[H|T].
No
?- [a,b]=[H|T].
H=a,
T=[b]
?- [a]=[H|T].
H=a,
T=[]
?- [a,[b,c]]=[H|T].
H=a,
T=[[b,c]]
?- [[a,b],c]=[H|T].
H=[a,b],
T=[c]
?- [a,b,c,d]=[H1,H2|T].
H1=a,
H2=b,
T=[c,d]
```

Looking at the examples above, we note that

- The list's tail, if any, is always a list (empty or not, but list).
- The head (and in general, all elements occurring before) are part of the list, and as part of the list may be any term (and therefore may or may not be a list).

It's all about the lists that are purely theoretical, we can say. And because the list is the main (in terms of "strength" or capability) data structure in PROLOG, in order to better understand how to deal with lists, we will try to define some predicates that allow some elementary operations on the lists.

3.6.3. Predicate to check if something is a list

Actually, every way we try to solve a problem in PROLOG is recursive. We start with the simplest case, and then generalizing it to arbitrary complexity. It is not the case for a predicate to check if something is a list. The simplest example of a list is an empty list.

All other letters, however, can be spread on the head and tail, with the tail must be a list. Hence we eventually get

```
isList([]).
isList([H|T]) :- isList(T).
```

3.6.4. Predicate to check if something belongs to the list

In this case, the simplest condition is: element X belongs to the list if X is the list head

```
isMember(X,[X|_]).
```

or the same in another way

```
isMember(X,[Y|_]) :- X=Y.
```

If he is not a head, then he must belong to the tail of the list

```
isMember(X,[_|Y]) :- isMember(X,Y).
```

However, as often happens in PROLOG, usually each predicate can be used in a completely different way than the one to which it was intended. In this case, the predicate *isMember* can be used to populate all items in the list.

```
?- isMember(X,[a,b,c]).
X=a;
X=b;
X=c;
No

?- isMember(X,[a,[b,c],d]).
```

```
X=a;
X=[b,c];
X=d;
No
```

3.6.5. A predicate linking two lists

In the standard simple case, an empty list linked (concatenated) with a list results in that same list:

```
link([],List,List).
```

A general case can be described as: to combine something ([H | T]) with a list, you must first attach the tail of this thing (T) to the list and then add the head H.

```
link([H|T],List,[H|Res]):- link(T,List,Res).
```

For example:

```
?- link([1,2,3],[a,b,c],Res).
Res=[1,2,3,a,b,c]
```

As before, this time we can use the *link* predicate to find out the answer to which list should be merged with the list [1, 2, 3] to get the list [1, 2, 3, a, b, c]

```
?- link([1,2,3],X,[1,2,3,a,b,c]).
X=[a,b,c]
```

Even more, we can find all the pairs of lists, the concatenation of which gives [1, 2, 3, a, b, c]

```
?- link(X,Y,[1,2,3,a,b,c]).
X=[],
Y=[1,2,3,a,b,c];
X=[1],
Y=[2,3,a,b,c];
X=[1,2],
Y=[3,a,b,c];
X=[1,2,3],
Y=[a,b,c];
X=[1,2,3,a],
Y=[b,c];
```

```
X=[1,2,3,a,b],
Y=[c];
X=[1,2,3,a,b,c],
Y=[];
```

3.7. OTHER SELECTED OPERATORS

3.7.1. Arithmetic operators

The main purpose of PROLOG is not for carrying out heavy-duty mathematics. The pattern for evaluating arithmetic expressions is (where `Expr` is some arithmetical expression)

X is Expr

The '*is*' operator is meant specifically for mathematical functions. The left argument has to be a variable and the right argument has to be a mathematical function with all variables instantiated. The '=' operator is used for unification of variables and can be used with any two arguments. The variable *X* will get to the value of `Expr`. For example,

```
?- X is 1+2.
X=3
yes
?- X is 2-1.
X=1
yes
?- X is 2*3.
X=6
yes
?- X is 3/2.
X=1.5
yes
```

Please note the difference between '*is*' and '=':

```
?- (2 is (3-1)).
Yes
```

You'll get '`Yes`', because the *(36-5)* is evaluated (solved) But if you ask

```
?- (2 = (3-1)).
No
```

You'll get '*No*', because Prolog will compare a number '*2*' to a formula '*3-1*', rather than to the result of solving the formula.

PROLOG knows many other ways of comparing two terms or instantiating variables, but for now, these two are enough. When working with functions, we will almost always use the *is* operator.

Other pre-defined PROLOG arithmetic infix operators are:

- >: greater than
- <: less than
- >=: greater than or equal to
- =<: less than or equal to

3.7.2. Advanced numerical operators

- *gcd*: Greatest common divisor of two numbers:

```
?- X is gcd(49,21).
X=7.
```

- *max*: Maximum of two numbers:

```
?- X is max(5,2).
X=5.
```

- *floor*: The largest integer smaller or equal to the evaluation:

```
?- X is floor(2.32).
X=2.
?- X is floor(-2.32).
X=-3.
```

- *ceil*: The smallest integer larger or equal to the result of the evaluation:

```
?- X is ceil(2.32).
X=3.
?- X is ceil(-2.32).
X=-2.
```

- *sqrt*: the square root $\sqrt{x}$ of the variable x

```
?- X is sqrt(3).
X = 1.732.
```

- *log*: the natural logarithmic $\ln(x)$ of the variable x

```
?- X is log(3).
X = 1.0986.
```

- *log10*: the decimal logarithmic $\log_{10}(x)$ of the variable x

```
?- X is log10(100).
X = 2.0.
```

3.7.3. Trigonometrical functions

- *sin*: the sine function $\sin(x)$ of the angle (measured in radian)

```
?- X is sin(1).
X = 0.8414709848078965.
?- X is sin(pi).
X = 1.2246063538223773e-16.
```

- *cos*: the cosine function $\cos(x)$ of the angle (measured in radian)

```
?- X is cos(1).
X = 0.5403023058681398.
?- X is cos(pi).
X = -1.0.
```

- *tan*: the tangent function $\tan(x)$ of the angle (measured in radian)

```
?- X is tan(1).
X = 1.5574077246549023.
?- X is tan(pi/4).
X = 0.9999999999999999.
```

- *asin*: the inverse sine $\arcsin(x)$ of a number (angle measured in radian)

```
?- X is asin(1).
X = 1.5707963267948966.
```

- *acos*: the inverse cosine $\arccos(x)$ of a number (angle measured in radian)

```
?- X is acos(1).
X = 0.0.
```

- `atan`: the inverse tangent $\arctan(x)$ of a number (angle measured in radian)

```
?- X is atan(1).
X = 0.7853981633974483.
```

- `sinh`: The sine hyperbolic $\sinh(x)$ of a number x

```
?- X is sinh(1).
X = 1.1752011936438014.
```

- `cosh`: The cosine hyperbolic $\cosh(x)$ of a number x

```
?- X is cosh(1).
X = 1.5430806348152437.
```

- `tanh`: The tangent hyperbolic $\tanh(x)$ of a number x

```
?- X is tanh(1).
X = 0.7615941559557649.
```

- `asinh`: The inverse sine hyperbolic $\operatorname{asinh}(x)$ of a number

```
?- X is asinh(1).
X = 0.881373587019543.
```

- `acosh`: The inverse cosine hyperbolic $\operatorname{acosh}(x)$ of a number x

```
?- X is acosh(1).
X = 0.0.
```

3.8. SELECTED EXAMPLES OF PROLOG PROGRAMS

In this section we will present some examples on simple problems of algorithms and data structures. Most of these examples are well known to the readers in other programming languages and it would be convenient to see how the PROLOG is different from other programming languages.

3.8.1. Example of generating Fibonacci list

The Fibonacci sequence $\{f_n\}$ is 1, 1, 2, 5, 8, 13, 21, 34, 55,... with the elements are defined in recurrent formula

$$f_1 = f_2 = 1;$$
$$n \geq 3 : f_n = f_{n-2} + f_{n-1}$$

The implementation in PROLOG is as follow:

```
fibonacci(1,1).
fibonacci(2,1).
fibonacci(N,R) :- N>=3,N1 is N-1,N2 is N-2,
                  fibonacci(N1,R1),
                  fibonacci(N2,R2),
                  R is R1+R2.
```

Example of running and querying:

```
?- fibonacci(8,R).
R = 34
```

3.8.2. Example of Sorting An Array

There are different methods of sorting an array of values. We will demonstrate one of the basic methods, which is the *Insertion Sort*.

The algorithm can be demonstrated using an example array in Fig. 3.1. Starting near the top of the array in Fig. 3.1a, we extract the 3. Then the above elements are shifted down until we find the correct place to insert the 3. This process repeats in Figure 3.1b with the next number. Finally, in Figure 3.1c, we complete the sort by inserting 2 in the correct place.

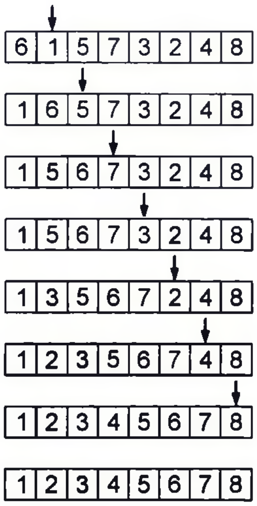


Figure 3.1. Insertion Sort demo for an input array of 8 elements {6, 1, 5, 7, 3, 2, 4, 8}

Assuming there are *N* elements in the array, we must index through *N* – 1 entries. For each entry, we may need to examine and shift up to *N* – 1 other entries.

```
for i=2->N
{
   find the correct place k from 1->[i-1] for A[i]
   key=A[i];
   remove A[i];
   shift [k]...[i-1] to [k+1]...[i] (by 1 element)
   move key into A[k]
}
```

The PROLOG implementation of `Insertion Sort` can be as follow:

```
insert_sort(List,Sorted) :- i_sort(List,[],Sorted).
i_sort([],Tmp,Tmp).
i_sort([H|T],Tmp,Sorted):- insert(H,Tmp,NTmp),
                           i_sort(T,NTmp,Sorted).
insert(X,[Y|T],[Y|NT]) :- X>Y,insert(X,T,NT).
insert(X,[Y|T],[X,Y|T]) :- X=<Y.
insert(X,[],[X]).
```

Running from the simulator:

```
?- insert_sort([6,1,5,7,3,2,4,8],X).
X = [1, 2, 3, 4, 5, 6, 7, 8].
```

3.8.3. Example of Sorted Binary Tree

a) Tree data structure

Binary search trees can be represented in PROLOG by using a recursive structure with three arguments: the key of the root, the left sub-tree and the right sub-tree – which are structures of the same type. The empty tree is usually represented as the constant `nil`. For example, a binary search tree in Fig. 3.2 can be specified in PROLOG using the following structure:

- Figure 3.2a: *t(3, nil, nil)*
- Figure 3.2b: *t(2, t(1, nil, nil), t(3, nil, nil))*
- Figure 3.2c: *t(8, t(4, t(2, t(1, nil, nil), t(3, nil, nil)), t(7, nil, nil)), t(12, t(10, t(9, nil, nil), nil), t(15, nil, nil)))*

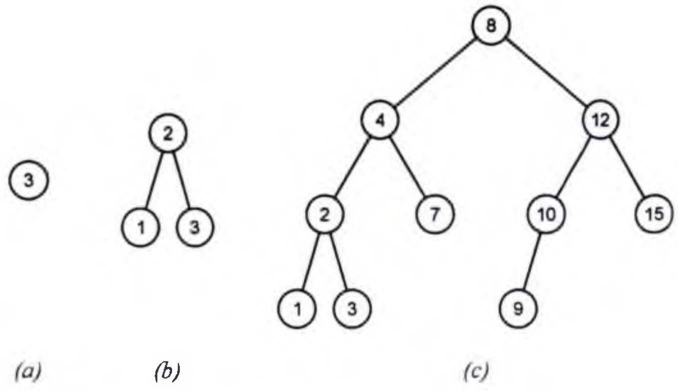


Figure 3.2. Examples of sorted binary trees

b) Tree traversal

There are three possible modes of traversing trees: inorder, preorder and postorder, depending on the order in which the nodes are processed

The inorder traversal processes the left sub-tree first, then the root node, then the right sub-tree. The predicate is presented below:

```
inorder(t(K,L,R),List):- inorder(L,LL), inorder(R,LR),
                         append(LL,[K|LR],List).
inorder(nil,[]).
```

The predicates for the preorder and postorder traversals are as follow:

```
preorder(t(K,L,R),List) :- preorder(L,LL),
                          preorder(R,LR),
                          append([K|LL],LR, List).
preorder(nil,[]).
postorder(t(K,L,R),List) :- postorder(L,LL),
                            postorder(R,LR),
                            append(LL,LR,R1),
                            append(R1,[K],List).
postorder(nil,[]).
```

c) Key searching in the tree

Key searching in a sorted tree is quite simple. The *search_key* predicate is presented below. If the key is not found, a *nil* value is returned.

```
search_key(Key,t(Key,_,_)) :-! .
search_key(Key,t(K,L,_)) :- Key<K, !,
                               search_key(Key,L).
search_key(Key,t(_,_,R)) :- search_key(Key,R).
```

d) Inserting a key

Each new key is inserted as a leaf node in a binary search tree. Before performing the actual insert, we must search for the appropriate position of the new key. If the key is found during the search process, no insertion occurs. When reaching a nil in the search process, we create the new node.

The *insert_key* predicate is presented below:

```
insert_key(Key,nil,t(Key,nil,nil)):-
             write('Inserted '),
             write(Key),
             nl.
```

```
insert_key(Key,t(Key,L,R),t(Key,L,R)):-
            !,
            write('Key already in tree\n').
insert_key(Key,t(K,L,R),t(K,NL,R)):-
            Key<K, !,
            insert_key(Key,L,NL).
insert_key(Key,t(K,L,R),t(K,L,NR)):-
            insert_key(Key,R,NR).
```

Please note that '*nl*' is equivalent to '*write('\n')*' which means move the printing cursor to the new line.

e) Deleting a key

The deletion of a key in a binary search tree also requires that the key be initially searched in the tree. Once found, we distinguish among three deletions of: a leaf node, a node with one child and a node with both children.

The first two cases are rather simple. When deleting a leaf, the its parent node will have new child equal NIL (if the leaf itself is also the root, the whole tree becomes empty). When deleting a node with only one child, that child node will replace the given node.

For the third case we have two alternatives: either replace the node to delete with its parent (or successor) – by reestablishing the links correctly, or to "hang" the left sub-tree in the left part of the right sub-tree (or vice-versa).

We have implemented the first alternative in the *delete_key* predicate as follow:

```
delete_key(Key,nil,nil) :- write(Key),
                           write('not in tree\n').
% this clause covers also case for leaf (L=nil)
delete_key(Key,t(Key,L,nil),L) :- !.
delete_key(Key,t(Key,nil,R),R) :- !.
delete_key(Key,t(Key,L,R),t(Parent,NL,R)):-
            !,
            get_parent(L,Parent,NL).
delete_key(Key,t(K,L,R),t(K,NL,R)):-
```

```
             Key<K, !,
             delete_key(Key,L,NL).
delete_key(Key,t(K,L,R),t(K,L,NR)):-
             delete_key(Key, R, NR).
get_parent(t(Parent,L,nil),Parent,L):-!.
get_parent(t(Key,L,R),Parent,t(Key,L,NR)):-
             get_parent(R, Parent, NR).
```

f) Display a tree

To display a tree means using the visualization to help checking th correctness of the tree's algorithms. We use the simple display similar to th tree of the Explorer in Windows systems, where each node is shifted (tabbed right to the depth at which the node appears in the tree.

The root of the tree is considered to be at depth 0. An example of a tre displaying is below:

```
2
  15
       10
           12
           4
                8
                    9
                    7
                3
  1
```

which is equivalent to the tree on Fig. 3.3.

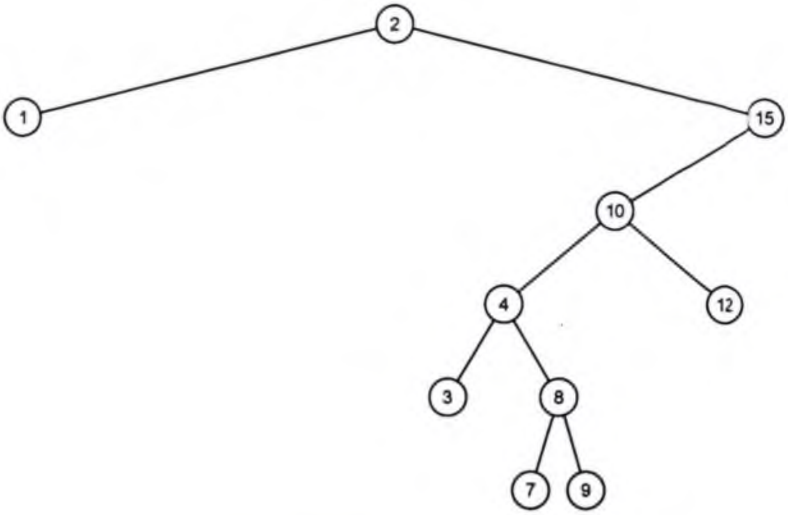


Figure 3.3. An example of binary tree

We can see that the keys on the left are printed first, then the root, then the keys on the right. This can be achieved using an inorder traversal in the tree.

```
% inorder traversal
tree_display(nil,_).
tree_display(t(K,L,R),Level) :-
          L1 is Level+1, tree_display(L,L1),
          key_indent(K,Level), tree_display(R,L1).
% Predicate which prints key K at Level tabs
% from the screen left margin and then proceeds
% to a new line
key_indent(K,Level) :-
            Level>0, !, L1 is Level-1,
            write('\t'),
            key_indent(K,L1).
key_indent(K,_) :- write(K), nl.
```

Chapter 4: LISP – A Functional Programming Languages

In terms of Artificial Intelligence (AI), in principle, apart from the distinction between "strong" and "weak" AI, we can divided AI approaches into:

- Symbolic - Intelligence as a symbols operator,
- Conjugal - Intelligence comes from the connections between the symbols.

Actually, the oldest paradigm is a logical paradigm associated with a typically symbolic approach. It is related to the emergence of LISP. The name LISP comes from the English word *LISt Processing*. The list was supposed to be a tool for efficient processing of symbolic data. And as soon as we see, LISP is pretty good in processing various types of lists. LISP is very old, it is the second high-level programming language in terms of age (only Fortran is older than LISP). LISP was official mentioned in 1959 [McCarthy59]. In 1960 John McCarthy published an article, in which he showed that with several operators and notations for functions, a Turing-complete language (Turing-complete language) can be obtained. John McCarthy is a pioneer in artificial intelligence. It was he who first used the term artificial intelligence, which he formulated in 1956 at a conference in Dartmouth.

The first implementation of LISP was developed by Steve Russel on the IBM 704. The first complete LISP compiler, created in LISP, was written in 1962 by Tim Hart and Mike Levin at MIT.

In the 1970s and 1980s, LISP was the best developed and most widely used language that offered the following set of features [WebLisp02]:

- Easy dynamic creation of new objects, with automatic garbage collection,
- A library of collection types, including dynamically-sized lists and hash tables,

- A development cycle that allows interactive evaluation of expressions and re-compilation of functions or files while the program is running,
- Well-developed compilers that could generate efficient code,
- A macro system that let developers create a domain-specific level of abstraction in which to build the next level.

These features are valuable for programming in general, but especially for exploratory problems where the solution is not clear at the onset; thus LISP was a great choice for AI research. In the next part we will base on a newer, more popular version of LISP, the so called Common LISP. Common LISP is not a concrete implementation, but an ANSI specification designed to unify the different LISP implementations that were created by mid of 1980s'. Currently, there are several Common LISP implementations, some are closed versions and some are available as FOSS (*Free and Open Source Software*).

4.1. LANGUAGE CHARACTERISTICS

LISP is a functional language (also called function programming language). Functional programming is a programming paradigm that is a variant of declarative programming, in which the core element of a language is a function, and the emphasis is on evaluating (often recurring) functions, rather than on running commands. Theoretical basis of functional programming was developed in the 1930's by Alonzo Church [Church41].

LISP is a language whose expressions are based on expression-oriented language. Unlike most other languages, LISP does not differ expressions from statements. The result of evaluating expressions is a value (or list of values) that can be used as an argument for another expression. Originally, John McCarthy introduced two types of expressions:

- S-expressions, also known as symbolic expressions. They reflected the internal representation of code and data.
- M-expressions, or meta expressions. They described the functions of S-expressions.

And although in the assumptions M-expressions were supposed to create LISP syntax, it turned out that S-expressions were more popular. And for many IT programmers this was considered more comfortable than the Fortran or Algol syntax.

The inseparable element of expressions in LISP are parentheses. Thanks to them the name LISP is also developed as *Lots of Irritating Superfluous Parentheses* or *Lost In Stupid Parentheses* [WebLISP]. However, the syntax based on S-expressions is the basis of its features - it is extremely regular, which greatly facilitates its automatic processing by a computer.

The basic element of LISP syntax is the list. It is defined as an chain of elements separated by space (white) characters and surrounded by the aforementioned parentheses. For example:

```
(1 2 abc)
```

is a list, whose elements are three atoms: value (default type of symbol) 1, value 2, and an object name '*abc*'.

Expressions are written as letters using the prefix notation (or the Lukasiewicz notation). Interestingly, the list as a data structure is also used in LISP to store the source codes.

Here are some examples of code in Common LISP.

- Classic program displaying 'Hello world'

```
(print "Hello world")
```

- Calculation of the factorial of a given number

```
(defun factorial(n)
     (if (<= n 1)
          1
          (* n (factorial(- n 1)))))
```

4.2. EFFECT ON PROGRAMMING

In [WebLisp01] the ideas were mentioned that LISP introduced and now (at least some of them) are part of today's programmer's reality:

1. Conditional statement: The if-then-else conditional statement is obvious today. But not many people know, however, that the statement was create at John McCarthy's work on LISP in a more general form as a *cond.* FORTRAN at that time only had a conditional jump instruction modeled on machine instructions set. From LISP the statement was included into modern languages.
2. Functional type: In LISP functions are represented by a special type in the same natural way as we normally represent integers and strings. Functions can be assigned to variables, passed as arguments, and so on.
3. Recurrence: of course, the concept of recursion existed long before LISP appeared, but LISP was the first programming language that allowed it to be used.
4. Garbage-collection: Garbage collection is a memory management architecture in which the process of releasing unused memory areas is performed automatically.
5. The program consists of expressions: LISP is, in other words, a tree of expressions, each of which returns a value. Most modern languages distinguish between expressions and statements.
    - o Expression (in programming language) is an evaluated (according to the rules defined in a given language) combination of values, variables, operators, and functions that returns a different value. It is said that the expression is evaluated to this value. Similar situation is in mathematics: expression is a representation of a certain value. Expressions may (though not necessarily) have side effects. The lack of side effects is one of the principles of functional programming, languages that do not support at all side effects are called purely functional languages.
    - o Statement (of a programming language) is the smallest independent element of the imperative programming language The program is created as a collection of different statements.

The statement may contain internal components (for example the expressions). Many languages (like C language) distinguish between statements and definitions: the statement contains an executable code, and the definition contains the declaration of an identifier. In most languages, the statements differ from the expressions that they do not necessarily return results and can be performed to achieve specific side effects, while expressions always return results and usually do not cause any side effects.

Interestingly, when using a language built on the basis of expressions, identical things (in the sense of the resulting effects) can be written in many different ways. So you can write

```
(if abc (= x 1)(= x 2))
```

but also can write:

```
(= x (if abc 1 2))
```

6. No matter what happens to the program, we still have access to all aspects of the language. This means that there is no distinction between for example compilation and execution processes. The code can be compiled or executed when read, can be read or executed when compiled or can be read or compiled when executed. The ability to execute the code while reading it allows users to change the syntax. Running code during compilation is the basis of advanced macros that allow you to create source code "on the fly". Compilation during execution allows you to use the language as an extension in other programs (for example Emacs).

4.3. SYNTAX BASICS FOR S-EXPRESSIONS

In this part, we will discuss the basic rules syntax used in LISP. Our goal is to describe the syntax in such a way that we can start understanding simple programs in that language quickly

a) Lists

LISP syntax belongs to one of the simplest programming languages and is based on the so-called S-expressions. The basic elements that make up the S-expression are list and atom. Lists are bounded by parentheses '*(*' and '*)*' and contain any number of elements that are S-expressions and separated by white characters. The rest are atoms.

And that's basically all and for the rest we will explain how the atoms syntax looks. In the following, we will cover the most popular types: numbers, strings, and names.

b) Atoms - numbers

Having experience with other programming languages, it can be predicted that a number is nothing but a string of digits, possibly preceded by a sign, and containing an optional fractional significant and integer separator. You can also expect that the use of so-called scientific notation, i.e. the exponent. And indeed, all this is true. Below are a few examples of correct numbers written in LISP:

- *1, +2, -3*: Integer numbers,
- *1.2, 1.2e3, 1.2e-4*: Floating point numbers with default precision,
- *1.2d3, 1.2d-4*: Floating point numbers with double precision,

Most likely, however, we will not guess the other possibilities that LISP gives us, for example, fractions or complex numbers.

- *1/2*: A fraction of one-second,
- *-3/4, -6/8*: Fraction minus three-forth in two equivalent ways,
- *#c (10 5)*: Complex number 10 + 5i, it means the real part equal to 10 and an imaginary part equal to 5,...

By looking at the above examples, you can see that the same numbers can be written in many ways, because LISP brings them to a certain canonical form specific to the type of number represented by the string. For example, fractions are always simplified, so the 2/2 is the same as 1, 1.0 is the same as the default form 1.0e0. Please pay attention that on the other hand, 1.0 is not the same as 1.0d0 or 1.

c) Atoms - Inscriptions

The string is a chain of characters enclosed in quotation marks "". In this case, the openning quotation mark " (or the closing quotation mark ") is a special character, which must be preceded by a backslash '\' if it is a part of the text. Because the backslash itself is a character that changes the meaning of the character after it, so when we want to use it, we must also precede it with another backslash.

- *"Abc"*: A string containing 3 characters A, b and c.
- *"Ab\c"*: A string containing 3 characters A, b, and c
- *"Ab\\c"*: A string containing 4 characters A, b, \ and c
- *"Ab\"c"*: A string containing 4 characters A, b," and c

d) Atoms - names

In LISP, the names can be either *ABC* or *a-b-c* or **abc**. Generally speaking, names play the role of symbols that may represent, for example, variables or functions. Therefore, later we will use the term *name* and *symbol*. From only the "appearance" of the names, we can not infer what they represent (this could be done in PROLOG, for example, where variables always start with a capital letter). Almost any character can in any way be part of the name. The exceptions are the following groups:

- Space character: The space character is used to separate elements of a list, therefore it can not be part of the name.
- Numbers: Numbers can be part of a name as long as the name can not be interpreted as a number.
- Period ('.'): The name may include a period (a dot), but can not consist only of dots.
- Special characters: There are special characters and as such should not be part of the name: round brackets, quotes and apostrophes, back tick,

comma, semicolon, backslash, and vertical line. They should not, but may become part of the name when preceded by a backslash or when placed between vertical slashes.

Basically the size of the letters used is irrelevant as they are always converted to uppercase letters, for example, the names *abc*, *Abc* and *ABC* are considered to be the same symbol: *ABC*. The case is different when we use special characters. For example, the strings `\a\b\c` or `|abc|` represent the name *abc*.

From the above it can be seen that the names in LISP can be much “richer” than in other languages like C or Java. And as in most languages, there are also some conventions related to naming.

- Multi part names are combined with a hyphen such as hello-world.
- Global variables are the names that start and end with a ‘*’ sign.
- Constants are the names that start and end with a ‘+’ sign.
- The name of a low-level function in a LISP program may be preceded by one or two ‘%’ characters.

The syntax of lists, numbers, captions, and names gives us enough insight into what and how we can write in LISP. Other elements such as vectors and arrays do not deviate significantly from the rules described above. It is important to be able to combine these rules. Let's look at the following examples:

• *x*;	Name (symbol) X
• `();`	List empty
• `(1 2 3);`	List containing three numbers
• `("abc" "xyz");`	List containing two strings
• `(x y z);`	List containing three symbols
• `(x 1 "abc");`	List containing a symbol, a number and a string
• `(+ (* 1 2) 3);`	List containing a symbol, a sublist and a number

4.4. FUNCTIONS

In this chapter we will learn how to write functions, that is, programming constructs, which provide the most elementary mechanism of abstraction.

4.4.1. Defining Functions

LISP functions are defined using the *defun* macro with the following syntax

```
(defun name (parameter*)
"Optional description string."
body-form*)
```

where *name* is the name of the function, *parameter** is an optional parameter list, *"Optional description string."* is a description of the function, a *body-form** is a string of expressions forming the body of the given function.

Practically every symbol or string can be a function name. Typically, this is a string of letters separated by a hyphen if necessary, if the name is multi-word. It's rather recommended to use the style *My-First-Function* rather than *My First Function* or *MyFirstFunction*.

Looking at the following hello-world function we can easily find the individual elements that make up it.

```
(defun hello-world() (format t "hello, world"))
```

A function that is another example includes all the described elements

```
(defun my-Sum(x y)
"Sum of two numbers"
(format t "Summing ~d and ~d. ~%" x y)
(+ x y))
```

We have one after the other

- Function name: *my-Sum*,
- List of parameters *(x y)* with which arguments will be associated,
- Function description: The function adds two numbers with a printed message to describe what it does.

- Body containing more than one expression (the above example has 2 expressions): The value returned by the last expression is the final value returned by the function.

4.4.2. Function parameters

a) Parameters required

If the parameters (arguments) list is a list of variable names, then such parameters are called required parameters. As the name implies, such a parameter must be given when calling a function.

```
(defun abc(a b) (list a b))
```

b) Optional parameters

To define functions with optional parameters we use the `&optional` symbol. This symbol should be after all the required parameters but before all optional parameters.

```
>> (defun abc (a b &optional c d) (list a b c d))
>> (abc 1 2)
==> (1 2 NIL NIL)
>> (abc 1 2 3)
==> (1 2 3 NIL)
>> (abc 1 2 3 4)
==> (1 2 3 4)
```

Please note that in this chapter, we use the '>>' to indicate the prompt of a LISP simulator. The '==>' denotes the results.

Of course, the default value assigned to an optional parameter of `nil` can be changed to something more appropriate. For this purpose, replace the optional parameter with a list containing the parameter name and any expression. If the user does not specify an argument, then the value will be the value returned by the expression.

```
>> (defun abc(a &optional (b 5)) (list a b))
>> (abc 1 2)
==> (1 2)
```

```
>> (abc 1)
==> (1 5)
```

For the above function *abc*, when calling it with *(abc 1 2)* it means *a* has an assigned value of 1, *b* has an assigned value of 2. But when calling with *(abc 1)*, it means that *a* has an assigned value of 1, *b* has no explicitly assigned value so it takes the default value of 5.

LISP gives us even more freedom in determining the value of parameters. We can make the value of a parameter depend on other arguments. An example of a function where such a dependency is needed is the function that creates a rectangle object. As we know, a square is a special case of a rectangle and therefore it is not worth creating a separate function for it. On the other hand, the need to specify the length of both sides when they are identical is quite annoying. However, in LISP, you can write a list of parameters as in the following example

```
(defun make-rect (width &optional (height width)))
```

This makes the optional height parameter when unspecified is set to width.

c) Any number of parameters

The maximum number of function parameters depends on the software version an implementation. In most cases it ranges from 4095 $\left(2^{12}-1\right)$ to 536,870,911 $\left(2^{29}-1\right)$. ANSI Common Lisp states that the value of CALL-ARGUMENTS-LIMIT, a positive integer one greater that the maximum number of arguments in a function call, is implementation dependent but must not be smaller than 50. In LispWorks it is set at 2047. It can be checked using the CALL-ARGUMENTS-LIMIT constant.

```
>> call-arguments-limit
==> 2047
```

Examples of functions called with variable number of parameters

```
(format t "hello, world")
(format t "hello, ~a" name)
(format t "a: ~d b: ~d" a b)
```

```
(+)
(+ 1)
(+ 1 2)
(+ 1 2 3)
(defun format(stream string &rest values)...)
(defun +(&rest numbers)...)
```

d) Named parameters

If a named parameter is not given, then the default value will be given to it as an optional parameter. Due to the use of the distinguishing name, the named parameters can occur in any order. Here are some examples of calls

```
>> (defun abc(&key a b c) (list a b c))
>> (abc)
==> (NIL NIL NIL)
>> (abc :a 1)
==> (1 NIL NIL)
>> (abc :b 1)
==> (NIL 1 NIL)
>> (abc :c 1)
==> (NIL NIL 1)
>> (abc :a 1 :c 3)
==> (1 NIL 3)
>> (abc :a 1 :b 2 :c 3)
==> (1 2 3)
>> (abc :a 1 :c 3 :b 2)
==> (1 2 3)
```

As with the optional parameters, the named parameters can be assigned default values and a variable specifying the "origin" of the argument.

```
>> (defun abc(&key (a 0) (b 0 b-input)
(c (+ a b)))
(list a b c b-input))
>> (abc :a 1)
==> (1 0 1 NIL)
>> (abc :b 2)
==> (0 2 2 T
>> (abc :b 3 :c 4)
==> (0 3 4 T)
>> (abc :a 5 :b 6 :c 7)
==> (5 6 7 T)
```

In addition, we can change the default behavior so that the parameters have a different name than the variables used in the body of the function. In the following example, write *(:aa a)* means *a is aa*. The following function definition *abc*

```
(defun abc (&key ((:aa a)) ((:bbb b) 0)
           ((:cccc c) 0 c-input))
           (list a b c c-input))
```

allows the following call:

```
>> (abc :aa 2 :bbb 4 :cccc 6)
==> (2 4 6 T)
```

e) "Mixing" of parameters

```
>> (defun abc(x &optional y &key z) (list x y z))
>> (abc 1 2: z 3)
==> (1 2 3)
```

and

```
>> (abc 1)
==> (1 NIL NIL)
```

f) Return Value

All the previous examples returned the value of the last expression evaluated as the result of the function. If this default behavior is not appropriate for us, we can use the *return-from* operator to instantaneously return a value from the function. The first argument to *return-from* is the name of the block (function) from which it returns. Because this argument is not subject to evaluation, there is no need to mark that as a name.

The following example uses nested loops to find pairs of numbers less than 5 whose sum is greater than the function's argument. Using *return-from* we return the first found pair meeting the given conditions.

```
(defun abc (n)
     (dotimes (i 5)
        (dotimes (j 5)
           (when (> (+ i j) n)
           (return-from abc (list i j))))))
```

4.5. FUNCTION AS A VARIABLE

Although the main use of functions is to call them, there are situations where it is convenient to treat a function as a variable. The classic example here is a sort function which, as one of its arguments, takes the function used to compare elements to determine which one should occur first. Another example may be graphing algorithms across and across the graph. Both mentioned algorithms have the same structure but only operate on other data structures. In one of them the two functions push and pop work with stack, in the other with queue. "Changing" these functions and leaving the algorithm completely unchanged, we get different behavior of the program.

In LISP, the function is a kind of object. By defining a function using *defun* in fact

- Create a new object representing the function
- and give the object a specific name.

After defining a function

```
>> (defun abc (x) (* 2 x))
==> ABC
```

with the operator *function* we can get the object representing it by writing

```
>> (function abc)
==> #<interpreted function ABC 21D11162>
```

Having an object representing a function, we can now use it to call it. Common LISP provides two functions that use to call functions that represent it: *funcall* and *apply*. This two functions have different ways of passing arguments to functions.

a) funcall

The *funcall* function is used when we know the number of arguments that will be passed to the function called when the code is created. The first argument of *funcall* is the name of the function object associated with the function we want to call, while the remaining arguments are the arguments passed to that called function. The following two lines are equivalent

```
(abc 1 2 3)
(funcall #'abc 1 2 3)
```

b) apply

In many cases, the list of arguments is not constant, or even the number of arguments is not known in advance. In this case, we should use *apply* instead of *funcall*. The first argument to apply is the name of the function to be called, and the second argument to the list.

```
>> (defun f (x) (+ 2 x))
==> F
>> (apply 'f '(5))
==> 7
```

The arguments for the function consist of the last argument to apply appended to the end of a list of all the other arguments to apply but the function itself.

4.6. ANONYMOUS FUNCTIONS

When we begin to use functions treated as arguments to other functions, we will soon be irritated with the need to define and call functions only to be able to use it only once. Avalanche names starting with *help*, *tmp* or similar can very quickly absorb us. If it is tiresome for us to define *defun* functions every time, we can use *lambda* expressions.

```
(lambda (parameters) body)
```

Lambda expressions can be thought of as functions whose name defines their action.

```
>> (funcall (lambda (x y) (+ x y)) 1 2)
==> 3
```

The above can be written in even more compact form, treating `lambda` as a function name

```
>> ((lambda (x y) (+ x y)) 1 2)
==> 3
```

Please note that the arguments can be the results of other function calls:

```
>> (funcall (lambda (x y) (+ x y)) (+ 1 2) (* 3 4))
==> 15
```

4.7. VARIABLES

4.7.1. Basic Messages

Common LISP is a *dynamically typed* language - that is, where errors related to type mismatches are detected at program execution not at program compilation.

One of the most elementary ways of introducing variables into a program (omitting the variables listed in the parameter list in the function definition) is to use the `let` operator of the form

```
(let  (variable*) body-form*)
```

`(variable*)` is a list containing variables with a value assigned to them, or variables themselves, if they have a default value of `nil`.

```
>> (let ((x 10) (y 20) z)
(format t "Variables: x=~a, y=~a, z=~a" x y z))
==> Variables: x = 10, y = 20, z = NIL
NIL
```

As the value of the return, the value of the last expression is returned. The scope of the variables introduced by `let` is limited by the expression itself. For program like below

```
(defun abc (x) (let ((y 2))
     (format t "x=~a y=~a" x y))
     (format t "x=~a y=~a" x y)))
```

We receive warnings when we try to execute since for the last instruction the variable Y is unbound.

```
    >> (abc 1)
    ==> X=1
    y=2
    Debugger invoked on a UNBOUND-VARIABLE in thread #
<THREAD "initial thread" RUN
    The variable Y is unbound.
    <...>
```

In this case, of course, there is no binding of the variable *y* in the second call to the format function.

In LISP, as in other programming languages, we are dealing with variable shadow when in the "internal blocks" a variable is defined with the same name in "external blocks". This is confirmed by, for example, the following program

```
(defun abc (x)
    (format t "Function: x=~a~%" x)
    (let ((x 2))
          (format t "Outer let: x=~a~%" x)
          (let ((x 3))
              (format t "Inner let: x=~a~%" x))
          (format t "Outer let: x = ~ a ~%" x))
    (format t "Function: x = ~ a ~%" x))
```

which returns the following effects

```
>> (abc 1)
==>
Function: x = 1
Outer let: x = 2
Inner let: x = 3
Outer let: x = 2
Function: x = 1
NIL
```

Similarly to *let*, *let** works with the difference that in the definition of variables you can refer to the variables that have already been defined. So you can write

```
>> (defun abc (x) (let*  ((y 20)
                           (z (+ y 30)))
                          (list x y z)))
==> ABC
>> (abc 10)
==> (10 20 50)
```

but no

```
>> (defun abc (x) (let   ((y 20)
                           (z (+ y 30)))
                          (list x y z)))
; (+ Y 30)
;
; Caught WARNING:
; Undefined variable: Y
<...>
```

Although a similar effect as using *let** can be obtained by repeated use *let*

```
>> (defun abc (x) (let ((y 2))
                  (let ((z (+ y 3)))
                  (list x y z))))
==> ABC
>> (abc 2)
==> (2 2 5)
```

4.7.2. Lexical variables

Common LISP has two types of variables: lexical and dynamic. By default, in the Common LISP, all bound variables are lexically scoped variables. Based on previous experience with imperative programming languages like C, Java and Python, the lexical scope of variables can be explained as follows: all references occurring in the area of the block in which the variable was defined.

Another explanation from [WebUnix02] is: *Closure* mean a subroutine that holds some memory but without some disadvantages of modifying a global variable. Generally, closure is a way to associate a function and the environment in which it is supposed to work. The environment stores all objects used by the function, not available in global visibility. The implementation of the closure is determined by language as well as by compiler. By definition, closures are mainly found in functional languages in which functions can return other functions using variables created locally.

4.7.3. Dynamic variables

Local variables, or variables, whose scope of action is limited to only a certain part of the code for which they matter, are quite a good idea to put some codes in a sense. Often, however, it is necessary to use variables with no boundaries and in fact each language has such "functionality". In LISP global variables are called dynamic variables or special variables and can be created in two ways: using *defvar* and *defparameter*. In both cases, we first give the variable a name, an initial value, and an optional description.

```
(defvar *abc* 1 "First global variable.")
(defparameter *def* 2 "Second global variable.")
```

In the case of *defvar*, the initial value is assigned to a variable only when the variable is undefined (not bound) earlier, but using *defparameter* always assigns the initial value to the variable. In addition, *defvar* can be used without the initial value, which creates an unbound global variable. Of course, we can use variables defined in one of the following ways (in any place), for example.

```
(defvar *abc* 1 "A new global variable.")
(defparameter *def* 2 "Another global variable.")
(defun myfun () (+ *abc* *def*))
```

And effect

```
>> (nyfun)
==> 1
```

Mechanism of covering global variables made a variable binding to a value that covers all previous bindings. We will try to illustrate this mechanism in the following example

```
>> (defvar *d* 1)
      (defun abc () (format t "d=~a" *d*))
==> *D*
>> (abc)
==> d = 1
NIL
>> (let ((*d* 2)) (abc))
d = 2
NIL
>> (abc)
==> d = 1
NIL
>> (defun myfun ()  (abc)


                    (let ((*d* 2))


                         (abc))
                    (abc))
```

we get the same effect:

```
>> (myfun)
==> d=1 d=2 d=1
NIL
```

We extend this example by changing the definition of *abc* function

```
(defun abc ()(format t "d=~a~%" *d*)
                    (setf *d* (+ 1 *d*))
```

```
                    (format t "d=~a~%" *d*))
```

The function *abc* results are straight forward

```
>> (abc)
==>
d=1
d=2
NIL
```

but the function *myfun* gives the results

```
>> (myfun)
==>
d=2
d=3
d=2
d=3
d=3
d=4
NIL
```

4.7.4. Constants

Constants, often in colloquial expressions called constant variables, are defined by keyword *defconstant* with the same syntax as the *defparameter* described, i.e. start with a constant name, after that an initial value, and end with optional description.

4.7.5. Assignments

Knowing how variables can be created, two further operations are natural: reading (using) the current value of a variable and assigning it a new value.

Both of these operations were already performed, because without them it would be difficult to give any examples. Referring to a variable's value simply by giving it a name. Assigning a new value to a variable is accomplished using the *setf* macro with the syntax

```
(setf varname value)
```

For example, assignment to variable *x* of value 1 is obtained by writing

```
(setf x 1)
```

Of course, according to the rules described earlier, binding the variable following the assignment does not affect bindings in another "block" program. That's why *setf* in function

```
(defun abc(x) (setf x 1))
```

in no way affect the environment as we find out by writing

```
>> (let ((y 2)) (abc y) (print y))
==> 2
```

The good news in daily use of *setf* is the ability to make multiple assignments. And so instead

```
(setf x 1)
(setf y 2)
```

it can be comnbined

```
(setf x 1 y 2)
```

Generalized assigning function:

Shorten formula	***Equivalent formula***
`(incf x)`	`(setf x (+ x 1))`
`(decf x)`	`(setf x (- x 1))`
`(incf x delta)`	`(setf x (+ x delta))`
`(decf x delta)`	`(setf x (- x delta))`

4.8. MACROS

Macros are somehow the layer of abstraction over LISP's core, in a sense. an interface for it. By creating new macros we can indefinitely extend the functionality of the language to adapt it to our needs and not interfere with its most elemental mechanisms at the same time.

It turns out what we will see at the beginning is that constructs known from other languages as basic or forming the core (such as conditional statements) here do not define language (or at least they do not).

4.8.1. `and`, `or`, `not` instructions

Shorten formula	***Value***
`(not nil)`	`T`
`(not (= 1 1))`	`NIL`
`(and (= 1 2) (= 3 3))`	`NIL`
`(or (= 1 2) (= 3 3))`	`T`

4.8.2. `if`, `when` and `unless` instructions

The LISP form for conditioning instruction *`if`* is:

```
(if condition then-form [else-form])
```

`(if (> 2 3) "X" "Y")` returns "Y"

`(if (> 3 2) "X" "Y")` returns "X"

`(if (> 2 3) "X")` returns NIL

or in general:

```
(if (condition)
     (form 1) (form 2))
```

By using macra, we can defme new forms of conditioning instructions as:

```
(when (condition)
       (form 1) (form 2))
(unless (condition)
         (progn (form 1) (form 2)))
 (defmacro when (condition &rest body)
       `(if ,condition (progn ,@body)))
(defmacro unless (condition &rest body)
     ` if (not ,condition) (progn ,@body)))
```

4.8.3. *cond* instruction

We can write a multi-level conditioning instructions as follow:

```
(if a  (do-x)
       (if b  (do-y)  (do-z)))
```

But we can use another equivalent way with the *cond* instruction:

```
(cond
     (test-1 form*)
     . . .
     (test-N form*))
```

Hence the equivalent of the above “nested” ifs

```
(cond  (a (do-x))
       (b (do-y))
       (c (do-z)))
```

4.8.4. *dolist*, *dotimes* instructions

a) dolist

```
(dolist  (var list-form)
         body-form*)
```

Example:

```
>> (dolist (x (list 1 2 3 4 5)) (print x))
==> 1 2 3 4 5
NIL
```

Or equivalent code:

```
>> (dolist (x '(1 2 3 4 5)) (print x))
==> 1 2 3 4 5
NIL


>> (dolist (x '(1 2 3 4 5))
          (print x)
              (if (eq x 3)
                  (return)))
1
2
```

```
3
NIL
```

b) **dotimes**

```
(dotimes (var count-form)
    body-form*)
```

Example:

```
>> (dotimes (i 5) (print i))
==> 0 1 2 3 4
NIL
```

4.8.5. *do* instruction

As presented above, *dolist* and *dotimes* are nothing more than "wrapping" macros as a more general macro that is. These instructions allow you to associate any number of variables and control them for subsequent iterations. In addition, we can specify a loop termination condition and an expression whose value will be returned after it terminates.

```
(do (variable-definition*)
    (end-test-form result-form*)
    statement*)
```

Each *variable-definition** is an expression

```
(var init-form step-form)
```

The *step-form* part is optional - if it's missing, the variable must be explicitly changed inside the loop. If no *init-form* is specified, the variable has a value of *nil*. The evaluation of the expression *end-test-form* in each iteration takes place after assigning new values to the variables.

As long as its result is *nil* the *statement** is executed. Evaluating it to *true* involves evaluating the *result-form** and returning the result of its last expression as the result of the loop *do*.

The *step-form* expressions are calculated in order of occurrence before any assignment occurs, which can be used when writing a program calculating the 20-th element of the Fibonacci sequence

```
(do ((n 0 (1+ n))
     (cur 0 next)
     (next 1 (+ cur next)))
       ((= 20 n) cur))
```

is equivalent to the *dotimes* loop that prints integers from 0 to 4

```
(dotimes (i 5) (print i))
```

recorded using the loop to may look like the following

```
(do ((i 0 (1+ i)))
     ((>= i 5))
     (print i))
```

4.8.6. *loop* instruction

The basic *loop* syntax is

```
(loop body-form*)
```

This is probably the simplest possible way to visualize a loop that iterates through *body-form** until it executes return. For example, the equivalent of an earlier loop to print 5 numbers from 0 to 4 may look like this

```
(loop for x in '(0 1 2 3 4) do (print x))
```

However, this extended syntax of the loops shows what we really are dealing with. A *loop* is quite an unusual "creation", while the ideal example shows the possibilities of macros. In a sense, the *loop* breaks with LISP's standard syntax, which results in not being accepted by everyone. In general, the *loop*, instead of "a lot of stupid parentheses", operates on a readable, practically for everyone, way of writing. Thanks to this, tasks such as counting, summing up or going through all the elements of the list are "off the shelf".

Let's give some more examples.

- Create a list containing integers from 1 to 5

```
>> (loop for i from 1 to 5 collecting i)
==> (1 2 3 4 5)
```

- Iterate through two lists in parallel, and cons up a result that is returned as a value by *loop*.

```
>> (loop for x in '(a b c d e)
        for y in '(0 1 2 3 4)
        collect (list x y))
==> ((A 0) (B 1) (C 2) (D 3) (E 4))
```

- Iterate through a list, and have an *if* instruction:

```
>>(loop for x in '(0 1 2 3 4)
          for y from 1
            if (> y 1) do (format t ", ~A" x)
            else do (format t "~A" x))
==>
0, 1, 2, 3, 4
NIL
```

- Sum the first 5 values of the expression

```
>> (loop for x from 1 to 5 summing (expt x 2))
==> 55
```

- Counting the number of occurrences of the letter "o" in the text:

```
>> (loop for x across "The quick brown fox"
        counting (find x "o"))
==> 2
```

- Calculate the 7th element of the Fibonacci sequence

```
(loop for i below 7
        and a = 0 then b
        and b = 1 then (+ b a)
        finally (return a))
==> 13
```

This way the *loop* shows the power of the macros. In this case, the macros were used to extend the basic syntax of the language, without affecting its remaining features. The *loop* keyword is analyzed according to its specific syntax, but the rest of it is nothing else than typical LISP expressions. What also important *loop* all the time remains "only" macros - it is not the core (or core) of the language.

4.8.7. Understand macros

Macro we define with *defmacro*

```
(defmacro name (parameter*)
"Optional documentation string."
body-form*)
```

As in *defun* we have a name, a list of arguments, an optional description, and a body of expressions.

Although programs call macros exactly the same way as functions, their behavior (macros) is quite different. The body does not return a value, but the LISP expression, which will later be evaluated. Macro arguments are not subject to echoes. Therefore, passing a list to the macro *(* 2 (+ 3 4))*, the argument in the macro body will always be a list *(* 2 (+ 3 4))* and not a value of 14. The following simple macro

```
(defmacro square (x)
'(* ,x ,x))
```

can be understood as follows: Every time the preprocessor encounters in the code the *(square X)* replaces it with *(* X X)*.

The key to understanding macros is knowing the difference between generating code and executing code (program). By writing a macro, we write a program that will be used by the compiler to generate the code that will be compiled. The program will execute only if all the macros are expanded and replaced with the code to be compiled. Moment of operation of macros is also called macro expansion time, to distinguish between the execution of the code written by programmers and the code generated by the macros.

This distinction is important in that when developing a macro, it is not possible to manipulate data that will exist during execution.

```
program
(defun abc (x)
(when (> x 10) (print 'big)))
```

will trigger the macro

```
(defmacro when (condition &rest body)
`(if ,condition (progn ,@body)))
```

which generates the following code

```
(if (> x 10) (progn (print 'big)))
```

It is important to note that LISP (depending on the implementation) can develop macros at different times.

- The macro can only be expanded once during compilation.
- Macro can be developed during first use.
- The macro can be expanded whenever it is used.

Well written macro should work in any of the above situations.

4.9. LISTS DATA STRUCTURES

Lists are some kind of abstraction over instances of objects grouped into pairs so-called *cons cells* as they use the `cons` function to create them.

```
>> (cons 1 2)
==> (1 . 2)
```

Two objects that make up the cell *cons* are called *car* and *cdr*

```
>> (car (cons 1 2))
==> 1
>> (cdr (cons 1 2))
==> 2
>> (setf (car *cons*) 3)
==> 10
>> *cons*
(3 . 2)
>> (setf (cdr *cons*) 4)
4
>> *cons*
==> (3 . 4)
* (cons 1 nil)
(1)
* (cons 1 (cons 2 nil))
(1 2)
```

Common other list manipulation functions are described as below.

***a)* car *and* cdr**

List can be processed by the functions *car* and *cdr*:

- *car* returns the first element of a list
- *cdr* returns the rest of a list after the first element.

```
(car '(a b c))
==> a
(cdr '(a b c))
==> (b c)
```

b)* append *instruction

append makes a new list consisting of the members of its argument lists all together.

```
(append '(a) '(b))
==> (a b)
(append '(a b) '(c d) '(e f g))
==> (a b c d e f g)
```

c)* reverse *instruction

reverse makes a new list that is the reverse of the input list.

```
(reverse '(a b))
==> (b a)
(reverse '((a b) (c d) (e f g)))
==> ((e f g) (c d) (a b)
```

d)* length *instruction

length returns the length of the input list.

```
(length '(a b))
==> 2
(length '((a b) (c d) (e f g)))
==> 3
```

e)* subst *instruction

subst makes a new S-expression (not just a list) with a specified substitution.

```
(subst 2.0 'a '(* a 5))
==> (* 2.0 5)
(subst 'alpha 'name '(The selected character is name))
==> (The selected character is alpha)
```

4.10. EXAMPLES OF LISP PROGRAM

4.10.1. Insert-sort in an array

Similar to the example of Insert-sort implementation presented in PROLOG program in chapter 3, the *insert_sort* algorithm can be implemented in LISP as follow:

```
(defun insert (item lst &optional (key #'<))
  (if (null lst)
    (list item)
    (if (funcall key item (car lst))
          (cons item lst)
          (cons (car lst)
                (insert item (cdr lst) key)))))
(defun insert-sort (lst &optional (key #'<))
  (if (null lst)
      lst
      (insert (car lst)
          (insert-sort (cdr lst) key) key)))
(insert-sort '(1 4 2 4 5 1 3 0))
==> (0 1 1 2 3 4 4 5)
```

4.10.2. Sorted Binary Tree

LISP has no such structures like tree. LISP is a list or at best a list of lists. It is really how the program interprets the nested list of lists that makes the list a binary or a n-ary tree. We can present the solution modified from [Abelson96].

In this version a node of a binary tree is represented as *(key left right)* so

```
key -> (car tree)
left_branch -> (cadr tree)
right_branch is -> (caddr tree)
```

Here is the code to access these 3 elements of a node:

```
(defun key (tree) (car tree))
(defun left-branch (tree) (cadr tree))
(defun right-branch (tree) (caddr tree))

//Creating node in a binary tree
(defun make-tree
           (keynode leftBranch rightBranch)
           (list keynode leftBranch rightBranch))

// Inserting an element into tree
(defun add (x tree)
     (cond ((null tree) (make-tree x nil nil))
     ((= x (key tree)) tree)
     ((< x (key tree))
          (make-tree (key tree) (add x (left-branch
     tree)) (right-branch tree)))
     ((> x (key tree))
          (make-tree (key tree) (left-branch tree) (add
     x (right-branch tree))))))

//Creating a tree from a list of elements
(defun create-tree(elmnts)
     (dolist (x elmnts)
     (setf tree (add x tree))))

//Creating an empty initial tree
(setf tree nil)
```

Example:

```
>> (setf tree nil)
==> NIL
>> (setf lst (list 2 1 15 10 12 4 8 3 9 7))
==> (2 1 15 10 12 4 8 3 9 7)
>>(create-tree lst)
==> NIL
```

We can display the resulted tree

```
>> tree
==> (2 (1 NIL NIL) (15 (10 (4 (3 NIL NIL) (8 (7 NIL
NIL) (9 NIL NIL))) (12 NIL NIL)) NIL))
```

This can be represented in the formal form as

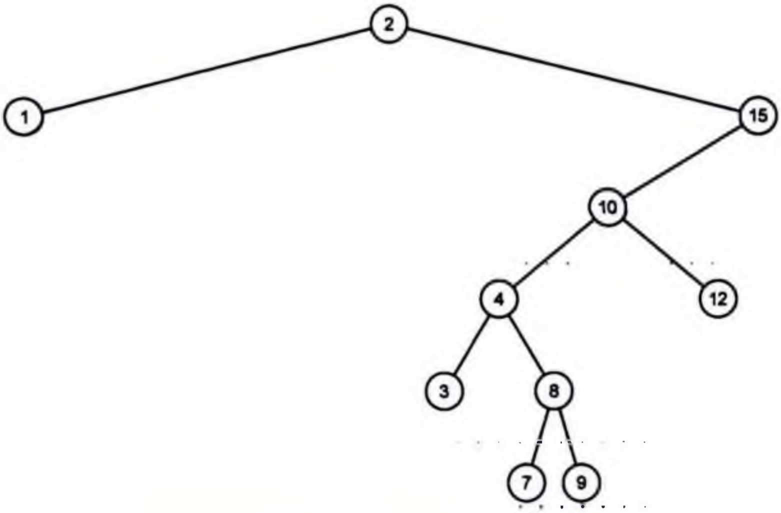


Figure 4.1. An example of Sorted Binary Tree structure

For the binary sorted tree, we can define 3 types of traversals

```
(defun inorder (tree)
          (cond ((null tree))
          (t (inorder (left-branch tree))
          (print (key tree))
          (inorder (right-branch tree)))))
```

```
(defun preorder (tree)
           (cond ((null tree))
           (t (print (key tree))
           (preorder (left-branch tree))
           (preorder (right-branch tree)))))

(defun postorder (tree)
           (cond ((null tree))
           (t (postorder (left-branch tree))
           (postorder (right-branch tree))
           (print (key tree)))))
```

Running results:

```
>> (inorder tree)
1
2
3
4
7
8
9
10
12
15
>> (preorder tree)
2
1
15
10
4
3
8
7
9
```

```
12
>> (postorder tree)
1
3
7
9
8
4
12
10
15
2
2
```

4.10.3. Hanoi Tower implementation

The Hanoi Tower problem can be solved in the recursive algorithm as follow: To move all the N disks from tower 1 (named as `fromT`) to tower 3 (name as `destT`) using the tower 2 (name as `buffT`) as the buffering tower we can:

- Move the first N-1 disks from `fromT` to `buffT` using the `destT` as the buffering tower.
- Move the N-th (largest) disk from `fromT` to `destT`
- Move the first N-1 disks from `buffT` to `destT` using the `fromT` as the buffering tower.

With this algorithm, first we need to define some data structure: The solution here is based on the lecture [WebFong]. In this example we represent a disk by a number, so that the i-th disk is represented by number *i* (the smallest disk has number 1, the biggest disk has number N). Second, we represent a tower by a list, which will contain the numbers corresponding to the disks of the tower starting from the top to the bottom.

```
;; A tower is a list of numbers
(defun make-empty-tower ()
  "Init an empty tower."
```

```
        nil)
    (defun tower-push (towerNo diskNo)
        "Add new disk diskNo to (on top of) the tower
  towerNo."
        (cons diskNo towerNo))
    (defun tower-top (towerNo)
        "Get the top disk of tower towerNo."
        (first towerNo))
    (defun tower-pop (towerNo)
        "Remove the top disk of tower towerNo."
        (rest towerNo))
```

Next, we define the *hnTowers* data type to represent the 3 towers of the game. Its constructors and selectors are given below:

```
    ;; HanoiTowers configuration
    (defun make-hn (fromT buffT destT)
        "Create a Hanoi Towers from three towers."
        (list fromT buffT destT))
    (defun hn-tower (hnTowers i)
        "Select the i-th tower of a hnTowers."
        (nth (1- i) hnTowers))
```

We write the calls to recurring operations:

```
    ;; Utilities
    (defun hn-tower-update (hnTowers i towerNo)
        "Replace the i-th tower by tower towerNo."
        (cond  ((= i 1)
                (make-hn hnTowers (second hnTowers)
                         (third hnTowers)))
               ((= i 2)
               (make-hn (first hnTowers) towerNo
                         (third hnTowers)))
               ((= i 3)
               (make-hn (first hnTowers) (second hnTowers)
                        towerNo))))

    (defun hn-tower-top (hnTowers i)
    "Return the top disk of the i-th tower."
    (tower-top (hn-tower hnTowers i)))

  (defun hn-tower-pop (hnTowers i)
    "Pop the top disk of the i-th tower."
    (hn-tower-update hnTowers i
      (tower-pop (hn-tower hnTowers i))))

  (defun hn-tower-push (hnTowers i disk)
    "Push DISK into the i-th tower."
    (hn-tower-update hnTowers i
        (tower-push (hn-tower hnTowers i) disk)))
```

The fundamental operator we can perform on a Hanoi configuration is to move a top disk from one tower to another:

```
  ;; Operator: move top disk from one tower to another
  (defun move-disk (fromT destT hnTowers)
      "Move the top disk from tower fromT to tower destT
  in hnTowers"
    (let ((disk (hn-tower-top hnTowers fromT))
          (temp-hn (hn-tower-pop hnTowers fromT)))
          (hn-tower-push temp-hn destT disk)))
```

We are now ready to capture the logic of our recursive solution into the following code:

```
  ;; Moving a tower from one tower to another
  (defun move-tower (N fromT buffT destT hnTowers)
    "Move the top N-1 disks from fromT to destT."
    (if (= N 1) (move-disk fromT destT hnTowers)
        (move-tower (- N 1) buffT fromT destT
        (move-disk fromT destT
        (move-tower   (-   N   1)   fromT   destT   buffT
hnTowers)))))
```

We use the function *solve-hn* to start the recursion:

```
  ;; Start function
  (defun solve-hn (N) "Solution of the Hanoi Tower."
      (move-tower N 1 2 3 (make-hn (make-full-tower N) nil
  nil)))
    (defun make-full-tower (N)
      "Create a tower of all N disks."
      (make-full-tower-temp N (make-empty-tower)))

    (defun make-full-tower-temp (N towerNo)
      "Push all N disks on top of tower towerNo."
      (if (zerop N)towerNo
          (make-full-tower-temp (1- N)
          tower-push towerNo N))))
    (trace move-disk)
```

To solve a Tower of Hanoi problem with 3 disks, we call *(solve-hn 3)*:

```
  >> (solve-hn 3)
     0: (MOVE-DISK 1 3 ((1 2 3) NIL NIL))
     0: returned ((2 3) NIL (1))
     0: (MOVE-DISK 1 2 ((2 3) NIL (1)))
     0: returned ((3) (2) (1))
     0: (MOVE-DISK 3 2 ((3) (2) (1)))
     0: returned ((3) (1 2) NIL)
     0: (MOVE-DISK 1 3 ((3) (1 2) NIL))
     0: returned (NIL (1 2) (3))
     0: (MOVE-DISK 2 1 (NIL (1 2) (3)))
     0: returned ((1) (2) (3))
     0: (MOVE-DISK 2 3 ((1) (2) (3)))
     0: returned ((1) NIL (2 3))
     0: (MOVE-DISK 1 3 ((1) NIL (2 3)))
     0: returned (NIL NIL (1 2 3))
  (NIL NIL (1 2 3))
```

From the trace we can actually read off the seqıence of operator applications necessary for one to achieve the solution configuıation.

Chapter 5: AI AND GAMES

In today's computer games, where graphical effects and physics simulations have already reached a very high level, artificial intelligence is gaining in importance. It is often used in games that are not scheduled in advance, and the game play is decided by the players. Great game play approach becomes more interesting, less predictable and schematic, which increases the popularity of the game and the satisfaction of the players. We are particularly interested in artificial intelligence in computer games. At the same time, the demands of the players grow, and this contributes to the emergence of new, better, more interesting and effective solutions. This is why artificial intelligence in computer games is becoming increasingly important. Its task is not only to create and control the environment surrounding the player character, but also to control the physiology and psychology of the character itself, which forces the player to become more involved in the game. The basic purpose of artificial intelligence algorithms is primarily to make the game more interesting and attractive to the computer player.

To this day, the most accomplished in this field are Half-Life Commanders and opponents in F.E.A.R. But as practically every thing in the world, artificial intelligence somewhere had to start.

5.1. INTRODUCTION

Game development lives in its own technical world. The style of programming in a game is still very different from that in any other products development. Games focus on speed, but it's different than programming for embedded or control applications. Games focus on optimized algorithms, but it doesn't share the same ideas as database server systems. This chapter presents about the application of AI in game development. Artificial Intelligence mechanisms are currently used in many game genres in which we deal with opponents, as well as with various phenomena, real-world objects. Therefore, they are widely used in strategic games (including combat simulations, crowds), as well as in standard shooters for modeling enemy behavior,...

The first was Alan Turing in 1951 with the famous Turing test. The test was based on a simple human game, where the judge talks to two people asking them questions. Based on the questions, it is to determine the gender of the person. In the Turing version, the judge speaks to a person and machine in a natural language (a language understood by people as we use every day). The machine passes the test when the judge can not determine which of the callers would be the machine. The test was generally quite controversial and many researchers did not take it seriously.

In the year 1952 Arthur Samuel entered the action. An IBM developer has developed his own learning algorithm for checkers. His algorithm did not check every possible move. Samuel has developed a function to evaluate the chance of winning based on several variables such as the position of the pawn, the number of pawns on both sides, etc. Artificial Intelligence tried to make such moves to optimize the value of the function. In 1954, Artificial Intelligence defeated the 4 best US players in checkers.

So far, we have already listed two pioneers in the field of artificial intelligence, but by 1956 no one had ever used the term "artificial intelligence". It was designed by John McCarthy at a conference in Dartmouth. More precisely, it was the English term "Artificial Intelligence". McCarthy also proposed LISP, and was awarded the Turing Award for his contribution to the development of AI.

Another important step in the development of computer intelligence was in 1967, in which Richard Greenblatt created his MacHack program. The program was created at MIT and became the first computer program that has taken part in chess tournaments, it is also the first software that defeated a living man in a chess match. It is interesting that MacHack became an honorary member of the Massachusetts State Chess Association.

Since then, the development of artificial intelligence in games has stalled a bit. Available equipment was not too fast. It was these technical limitations that made the development of intelligent opponents a little stop. There were artificial intelligence games, but it was plainly simple and painfully predictable.

From the late 1950s through to the early 1980s the main AI research concentrated on "symbolic" systems. A symbolic system has two main components: a set of knowledge (represented as symbols such as words, numbers, sentences, or pictures) and a reasoning algorithm to create new combinations of symbols or new knowledge. A reasoning algorithm consists of searching: trying different possibilities to get to the result.

An AI system should have a large database of knowledge and known reasoning rules to discover new things. For games, there were different symbolic such as blackboard architectures, path finding, decision trees, state machines, and steering algorithms.

Until the early 1990s, there were increasing problems with symbolic approaches. First, the early successes on simple problems didn't seem to scale to more difficult problems. It might be easy to develop AI that understands simple sentences, but understanding a full human language was not possible. Second, from a philosophical viewpoint, symbolic approaches weren't biologically plausible. You can't understand the way a human being does something by using a symbolic algorithm without any supporting knowledge.

The effect was a move toward natural computing: techniques inspired by biology or other natural systems. These techniques include neural networks, genetic algorithms, fuzzy logic,...

5.2. GAME AI

Pacman [Midway Games West, Inc., 1979] was one of the first game played with AI. *Pacman* has definite enemy characters that seemed to co-operate against you, moved around the level just as you did, and try to catch you in the natural way.

Pacman relied on a very simple AI technique: a state machine. Each of the four monsters (later called ghosts) was either chasing you or running away.

At each junction they took a semi-random route. In chase mode, each had a different chance of chasing the player or choosing a random direction. In run away mode, they either ran away or chose a random direction. All were very simple. Game AI didn't change much until the mid-1990s.

Take a classic like Golden Axe [SEGA Entertainment, Inc., 1987]. This game had a neat innovation with enemies that would rush past the player and then switch to homing mode, attacking from behind. The AI level was only slightly higher comparing to Pacman.

At another significant event in the development of artificial intelligence in computer games we waited until 1994 and Warcraft games. Studio Blizzard has been tempted to apply advanced track search algorithms In the case of the RTS game, where multiple units are displayed at once, it was a bold move that proved to be a huge success. Agents were able to move the area, avoiding obstacles and each other. Already then the path search algorithms themselves were nothing extraordinary. What impressed here was how many units at once could use this capability.

In 1996 we are back in chess. In the world, it has becone loud because of the Deep Blue computer created by IBM. The machine sat down to a chess party with world champion Garri Kasparov. In the first clash the computer won once, but later lost 3 games and 2 draws, so the general ended 4: 2 for the Russian. After a year of additional work on the system, there was a rematch. This time the machine won 3.5 to 2.5.

Goldeneye 007 [Rare Ltd., 1997] probably did the mcst to show gamers what AI could do to improve game play. It added a sense sinulation system: a character could see their colleagues and would notice if they vere killed.

The first game whose intelligence was praised in the growl world was released in 1998 by Valve Half-Life. He did not use any outstanding techniques, but introduced intelligence to a new level. One cf the enemies that came up was the commandos. What made them stand out fron other opponents was the fact that they were able to cooperate with each other shield each other flank the player, etc.

The AI in most modern games addresses three basic abilities: to move characters, to make decisions, and to think tactically or strategically.

In the same year also appeared *Thief* game produced by Looking Glass Studios. Developers focus here on a slightly different aspect of artificial intelligence, which are the sensors. Because if an AI object is unable to pick up stimuli from the environment then it is worthless. At *Thief* the guards responded to the sounds and light.

The next important step came quite quickly, because already in 1999, due to the study of Digital Extremes, which unveiled the *Unreal Tournament*. The key element here was the bots that were able to learn from the player. This is a mechanism used today in many multiplayer games, where we deal with bots. The game looks at the way players live and tries to use similar patterns of movement or good space for hiding.

Another bold step in AI development took place in 2000, and more specifically in Colin McRae *Rally 2.0*. This was the first game using a neural network. The game collected data based on the player's driving style. AI was based on two data - the driving line and the driving model. The first data is the line of optimal driving track, the second is determined by the speed, type of pavement etc. Using the drive model data, the computer tried to hold the line as much as possible. By adding to the knowledge gained from the player, AI could become a better driver.

Also in the year 2000 Maxis introduced *The Sims*, a first time system of needs - where our character was hungry, needing to wash or use the toilet. Not much, agents were able to make a relationship. There was also an interesting system where smart objects were used. i.e. the character as such was unable to use various items. This item informed our sims how it should be used. Later, a similar solution will be used by the Monolith team working on AI to play F.E.A.R.

A year later, *Black & White* appeared at LionHead Studios. The programmers decided to create a very interesting project, which was a retreat. A pet had learned how to behave. Its development was based on "strengthening".

So we could praise him for good behavior and punish him for his bad behavior. A well-trained hound, he could have been a great help when the ill-trained could have been more disturbed. Again, when we sent him conflicting signals, his behavior became chaotic and difficult to predict. The `Black & White` project was the first project where artificial intelligence learned in real time during the game. The idea came back in 2005 with the second part of the game.

In addition to the `Black & White` sequel, in 2005, the game was launched by F.E.A.R of Monolith Productions. To this day, artificial intelligence in this game, is considered one of the most advanced. Opponents after classic behavior like hiding, shooting for a player, or searching for him in case of loss of visual contact, show team behavior, i.e. when one of the team members changes position to fling the player, the rest of the team is covering him. Opponents do not attack only the simplest line of resistance, but they can also fling or attack the player from behind. They use grenades and field guards.

A special milestone is November 2015, when Google unveiled its `AlphaGo`. The game has long been a serious challenge for artificial intelligence. The best chess players have been championing for two decades (the first was the Deep Blue computer created by IBM - in 1997 he defeated Garry Kasparov), but this ancient board game from the Far East has long been a problem for machines to overcome.

The rules of the game are simple. On the perpendicular lines of the board, players alternately place white and black stones (stones). The object of the game is to occupy as much territory as possible and surround your opponent with as many stones as possible. However, the number of possible moves is enormous.

The chess player can choose from nearly 20 moves in every move. The average player plays in 150 moves, which means 10170 (the one with 170 zeros) of possible stones on the board! This is an unbelievably large number, larger than all the atoms in the universe, which is "only" 10^{80}. As a result, even the best supercomputer is not able to deal with all variants of the game.

This game was one of the biggest challenges for artificial intelligence. Chess or checkers have long been losing people, but the human mind still has an advantage in playing it. Until the introduction of *AlphaGo*, a computer program, created by scientists from Deep Mind (Google's daughter company). *AlphaGo* defeated the legendary champion Lee Se-dol in the first of five games played in Seoul, South Korea. Lee surrendered after 3.5 hours. This was the first game of the five series to be played. The match is 1 million USD prizes, and of course pride of man.

"I was very surprised," Lee said, surprised just after the game. – "I did not expect to lose. I did not think that *AlphaGo* would play so perfectly".

AlphaGo has already demonstrated its power when he defeated the European champion Fan Hui in 2016. *AlphaGo* won 5:0! But Hui has only the 2nd master degree dan. Lee has the strongest, professional grade 9th and is considered one of the top two players in the world.

"I do not regret that I took on a computer challenge," Lee says. "I admit, I'm in shock, but what happened has happened. Now I'm getting ready for the next batch". The Master wants to improve his opening, because he believes that this weak start lost him. But even if you win the next parties with the program, it seems that the days of human supremacy are counted.

AlphaGo does quite well. It reminds of the structure of the neuron network that is, acts like the brain. It strengthens the connections between artificial neurons through examples and gained experience. Instead of trying to predict (calculate) the best moves for several dozen turns forward (it would take the computer very long time), programmers have equipped him with an algorithm that allowed him to learn the best strategy, just as people do it - by training. The solution DeepMind suggested would be a hybrid on every level. *AlphaGo* consists of two deep neural networks (DNNs) that find promising moves and determine whether they are worth the price. Neural networks have revolutionized artificial intelligence in recent years, but even they themselves have not been able to defeat the game of Go. Therefore, beside the neural network, *AlphaGo* uses Monte-Carlo Tree Search to choose the right step.

The program first analyzed more than 30 million items from the games played by the masters, collecting information about the state of the game in the same way that image recognition programs analyze them based on a set of pixels. Then he played parties with himself on 50 computers, gaining more and more skills. Part of the teaching process of `AlphaGo` was hybrid. One subsystem was trained by human experts, which was then used to play with the other subsystem. In this way, `AlphaGo` quickly understood the rules of the game and made possible moves and then played thousands of times with himself, grinding his skills.

Demis Hassabis, head of Deep Mind, claims that the program has an intuitive understanding of what moves on the board in it are beneficial and can make long-term decisions on that basis. He had a strategy game he learned himself.

Tobby Manning of the British Go Association, who had judged the previous duel with the European champion, said that the only difference between a human player and a computer was the time when they were thinking about the next move. The computer did it faster.

5.3. A PROPOSED MODEL OF AI MECHANISMS FOR GAME

Artificial intelligence in computer games is used primarily to simulate human behavior and to automatically raise the difficulty of games. The artificial intelligence technology used in games is designed to create as most closely related to real behavior and events mapping reality.

The use of artificial intelligence in computer games can be divided into three categories:

- Reality of the game world: mainly used in CRPG (Computer Role-Playing Games). AI is responsible for controlling the actions of the agents (characters controlled by the game) with which the player's hero is encountered.

- Battle Support: The most common category of artificial intelligence in computer games - mainly in strategy games and so-called shooters. AI is supposed to automatic control agents during combat.
- Commenting on events - most often in sports games. AI is responsible for commenting on events taking place in the game world based on current player actions.

Innovative artificial intelligence solutions in computer games are constantly being refined to make the decision-making processes of the computer better and more error-free. Thanks to the solutions based on modern solutions, the existing possibilities have been expanded, creating more interesting and engaging games that are as close to the real world as possible.

Most modern computer games use at least several different algorithms. Otherwise, the behavior of the AI while moving from one place to another is calculated, and decisions are made in combat or conversation. The gameplay also changes the behavior of the system - strategic or logical games do not use methods used in arcade games, and in racing games to conduct dialogues.

Here are some popular algorithms, in which simulation of human behavior and use of adaptive capacity by computer system is possible:

- solving decision problems,
- creating an action plan,
- strategic planning.

Algorithms of this type include: A*, herd algorithm, finite state machines, decision trees. Other types include heuristics, neural networks, fuzzy logics,

5.3.1. The A* algorithm

The A* algorithm is a widely used algorithm in games for searching paths (or for "path finding"). In games, it is often used to control the character from A to B via a network of nodes. When defining a route, it does not look for a "blind path", but it estimates the best direction of exploration. The algorithm calculates the fastest and most effective connection between nodes, removing at the same time the unavailable nodes due to various reasons (for example part of the wall, the node is occupied by another character,...). When the examined location turns out to be the target sought, the algorithm terminates, otherwise it remembers the adjacent locations so that they can be checked in the future.

An unquestionable disadvantage of this algorithm is its predictability - if we do not use any other methods or algorithms to calculate paths, the player can quickly see the AI principle and then easily deceive it. Such is the "*Hitman*" series of games (2000), where the player takes on the role of a paid killer. The game uses many complex algorithms to simulate crowd reactions and individual enemies to preserve the protagonist. It is worth noting that AI used in this case during the chase is quite predictable, so that a player with good reflexes and accuracy can get rid of virtually all enemy meals and complete his mission without problems. The situation seems to be comic, but the blame for this is the underdeveloped intelligence of the enemy.

Another example of the use of the A* algorithm is "*F.E.A.R.*" (2005). In a fight the player is set before a very intelligent enemy - AI can decide when it pays to risk dropping a position, and when it is better to blindly shoot from behind the guard. There are also examples of cooperation - when one of the enemies tries to change their position, the other shields him by firing the player. Also much better than the other games of this type, it also turns out to be "pathfinding".

5.3.2. Herd algorithms

The herd algorithm was first demonstrated in July 1987 by Craig Reynolds in Computer Graphics. The herd algorithm gives the group a realistic set of collective behaviors, where combining a few relatively simple rules can simulate very complex herd behavior. It serves as a realistic representation of collective behavior, such as a flock of birds, a shoal of fish or a crowd. According to Craig, this algorithm controls four assumptions:

- resolution - control prevents the creation of crowds, local clusters in one place (each unit in the crowd must maintain a certain distance apart to avoid collisions),

- leveling - it is possible to change its speed and direction of movement and adapt it to other units,
- cohesion of the group - control of the gathering of agents staying close together in local groups,
- avoidance - collision avoidance / collision control.

Result of the aforementioned action. The algorithm is a flock moving with the same dynamics of movement as one body, avoiding all obstacles and hostile characters. Initially, the algorithm was used in bird flight simulations, with time being used in strategic games like "*Warcraft*" (1994) and "*Command & Conquer*" (1995). They were one of the first games where artificial intelligence was used in real time on multiple units simultaneously (characters were able to move in the same direction without interfering with each other, changing the direction of the march, and avoiding obstacles encountered in a very short time). Over time, the algorithm has been modified so that the player can decide on the shape and behavior of the flock. In the role-playing games "*Baldur's Gate*" and "*Icewind Dale*," a player who directs a team of heroes who travel together in solidarity at any time can change the way they move and form depending on their needs and team composition. The algorithm allows for an instant response to the situation.

5.3.3. State machines

Another example of the use of AI technology in games is finite state machines. In the games of the 90s, this technique was used to control agents, but in the latest releases, finite state machines are used to control AI games. They are often used in role-playing and strategy games, mainly for controlling players' dialogues with agents and interacting or managing the world. They store the state of the game, process commands from the player, or manage the state of the object. The finite state machine consists of a set of "states" that are in a given game space. Depending on the event, the transition to a different state changes and the way you interact with the player changes. An excellent example is the "*Fallout*" series, known for its rich world and almost living AI characters. The player has a major impact on the world around him, and depending on his actions, the world changes. Pretty extreme example from the third part of the series is the town Megaton, which was built around the unexploded bomb. The player has a number of opportunities and situations related to this town - helping the residents, disarming the bomb and gaining the trust of the sheriff or activating the bomb and contributing to the destruction of the city.

5.3.4. Decision trees

Another method of preparing fight in different types of games is decision trees. They are used in the simplest situations like "Is this a close player? If Yes, hitting, If no, approach",... as in more complicated decision-making like "If the enemy is close but more than 5 meters away, see where his comrades are", or "If they are closer than 5 meters from him, use this skill",...

The decision tree is presented in the form of a decision graph and their possible consequences, the nodes of which are the state of the game, and the child nodes are the positions obtained after one move. The decision nodes and consequent nodes are alternating, and each path ends with an end node. The agent analyzes the decision tree as far as it can or considers it necessary, considering all possible moves against the present situation - and chooses which decision tree is the best. This method works best for uncomplicated, 3-4 step situations, for more complex entertainment and higher AI levels, it is harder to optimize. Its use is in the role-playing computer game (CCRPG), where the player controls most often a hero (or team) moving in a fictional world, and turn-based game play makes even the most complex calculations need not be done in time real.

5.3.5. Graph algorithms

The artificial intelligence mechanisms commonly used in computer games also include graph algorithms. Graphs are often used as a representation of the world of the game, to reduce the complexity and amount of detail that are often irrelevant from the point of view of AI-controlled forms

a) Road planning problem

The problem of finding a road appears very often in real-time strategies, as well as in other types of games controlled by the computer. It usually consists of finding a route from A to B at a minimum cost. The costing function most often takes into account the distance to the destination, but it can also take into account the terrain (mountains, swamps), the presence of opponents and obstacles.

b) Representation of the game world

Due to the computational and memory complexity of road finding algorithms, different types of game representation techniques are used to accelerate the calculations.

Rectangular or hexagonal mesh is often used in strategic games. This is a simple technique for implementation, because the site is represented by the same elements. This method, however, has a drawback: characters moving around the terrain look, as if they were moving around the chessboard. In addition, this representation can introduce many redundant elements, for example a meadow occupying a larger area is represented by many small cells instead of one larger, which often leads to slower route calculation calculations. This problem can be solved by the use of four trees that group uniform areas into one larger.

Methods for representing more irregular areas include the convex polygon and visibility points. The first technique allows for a comfortable and efficient presentation of the terrain, but usually requires additional effort for level designers. The second method also reduces the size of the searched area, but may cause objects to move along obstacles.

Due to the numerous advantages and disadvantages of the above methods, it is difficult to find a universal technique of representation of the game world, which allows for an optimal search of the path, after which the movement of the character would not look artificial. For this reason, additional smoothing algorithms, as well as hierarchical search, are often used.

5.3.6. Heuristics

Human beings use heuristics all the time. A heuristic is an approximate solution that might work in many situations, but is unlikely to work in all. We don't try to work out all the consequences of our actions. Instead, we rely on general principles that we've found to work in the past or what we believe will work.

There are whole ranges of heuristics that can be applied to general AI problems. In the *Pacman* example, the ghosts chase the player by taking the route at a junction that leads toward player's current position. The simple rule might not be useful if the player continues to move. But the rule of thumb (move in the current direction of the player) works and provides sufficient competence for the player to understand that the ghosts aren't purely random in their motion.

5.3.7. Neural networks

Modern AI games can analyze the state of the game and adapt it to the player's skills based on neural networks. They condition the learning process by gaining experience, thus regulating and consequently automatically adjjusting the degree of their difficulty. In this case there is a correlation between the player and the game - the growth of the player's experience and skills results in the development of the virtual world. The algorithms that govern the difficulty levels are:

- gaining experience and analyzing them
- selection of parameters resulting from achieved effects
- planning strategies for further development using self-adaptive systems.

So far games with neural networks are most often used in sports games, mainly racing type "Colin McRae *Rally 2.0*" (2000). Based on input data such as road shape, terrain type, and vehicle parameters, the network must generate commands for computer-controlled vehicles so that they can unobstruct the entire track and compete with the player.

In some games called "shootings" using computer-controlled characters (so-called "bots"), AI with learned behaviors that simulate action of real players based on the patterns introduced by the developer. And while they are predictable, they are sufficiently capable of replacing live humans during the game. An example of such a game is *Quake 3 Arena* (1999), where the bots are designed to learn from the player (and also from each other) behavior and strategy. Their AI checks the next tactics, accepts the active and rejects the ones that do not work. This is one of the first games to use this solution.

A good example of "learning intelligence" is the AI of the game "*Black & White*" (2001) and its continuation from 2005. This is a real-time strategy with elements of the role playing game set in Eden. *Black & White* is a story in which a player embarks on an impersonal supernatural being capable of doing miracles and controlling the world around him. The curious animal as it progresses in the game grows and also shapes his personality. A well trained chubby can assist the player in many activities, however we have to constantly teach him to act as we wish. The development of the personality of the person is based on the principle of "strengthening", which in psychology means rewarding or punishing for the stimulus. So there is the possibility of corruption: if we arbitrarily punish and reward, he will never learn correct behavior, and his actions will be chaotic and unpredictable, which affects the course of the game.

5.3.8. Fuzzy logic

Improving the quality of the game's functionality and increasing its attractiveness for the player also enables the use of fuzzy logic. In the field of human behavior simulations it enables:

- giving up the emotional state,
- expanding the emotional and psychological sphere,
- controlling the unrestricted phenomena and behaviors that are equivalent to those - occurring in the real world.

The fuzzy logic proposed by Lotfi Zadeh is a conjunction of probability theory with fuzzy sets theory. Methods of fuzzy logic, along with evolutionary algorithms and neural networks, are modern tools for the construction of intelligent systems capable of generalizing knowledge. In the context of artificial intelligence, it allows states other than binary 0 and 1.

In computer games fuzzy logic is used to simulate emotions, so that it is created much deeper and more similar to the human emotional sphere of computer characters. Most often in feature games are characters "characters" that influence how other characters look at the player. Is he good or bad, lawful or chaotic? Their relationship to the player is expressed in questions like "How can I trust him" or "How much can I give him?"

5.4. BEHAVIORAL ROBOTIC ARCHITECTURE

Virtual worlds in 3D computer games show a very high resemblance to the environments in which real robots work and move. Both, and controlled by the AI of the individual in the game, must observe the surrounding environment and on the basis of their knowledge to make decisions consistent with the goals. For this reason, games were started using techniques similar to those used in real control robots. For example, extended behavioral networks have been successfully used in `Quake II` and `Unreal` games to model the behavior of independent heroes (computer-controlled).

The main idea behind behavioral networks is to dissipate the activation energy in the network elements (behavioral and target) and then select the behavior that has the most energy. An example of the use of the above technique, as well as subsurface architectures, also used in robotics, is presented in [Gregory09].

An example of structure is used to help explaining the AI used in a game. It splits the AI task into three components: movement, decision making, and strategy. The first two components contain algorithms that work on a character-by-character basis, and the last component operates on a whole team or side. Around these three AI elements is a whole set of additional infrastructure.

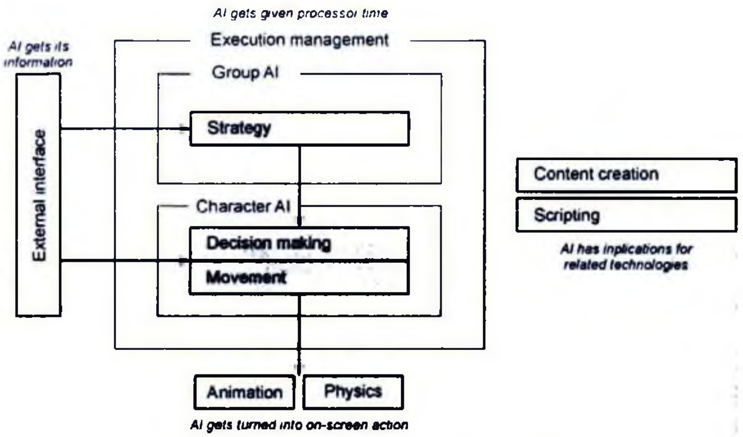


Figure 5.1. The AI model [Millington09]

Not all game applications require all levels of AI. Board games like *Chess* require only the strategy level; the characters in the game don't make their own decisions and don't need to worry about how to move.

On the other hand, there is no strategy at all in very many games. Characters in those games have no coordination that makes sure the enemy characters do the best job of thwarting the player.

A general structure of the AI engine might look something like Figure 5.2 [Millington09]. Data is created in a tool (the modeling or level design package, or a dedicated AI tool), which is then packaged for use in the game. When a level is loaded, the game AI behaviors are created from level data and registered with the AI engine. During game play, the main game code calls the AI engine which updates the behaviors, getting information from the world interface and finally applying their output to the game data.

The techniques used depend heavily on the genre of the game being developed. As an AI game is being developed, we'll need to test a mix and match approach to get the behaviors you are looking for.

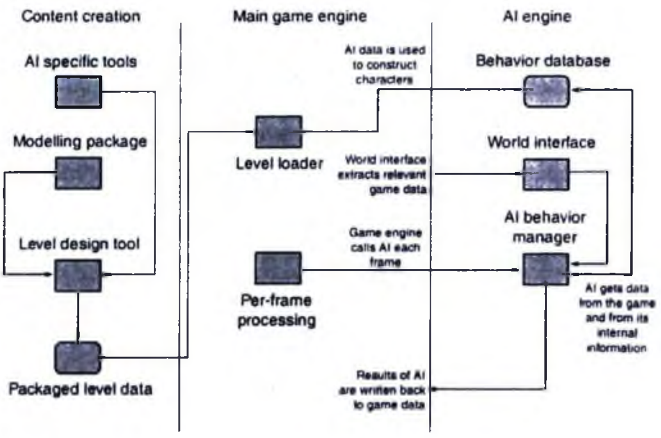

Content creation
Main game engine
AI engine
AI specific tools
Modelling package
Level design tool
Packaged level data
Level loader
Per-frame processing
AI data is used to construct characters
World interface extracts relevant game data
Game engine calls AI each frame
Results of AI are written back to game data
Behavior database
World interface
AI behavior manager
AI gets data from the game and from its internal information

Figure 5.2. An proposed schematic for AI game system [Millington09]

5.5. SELECTED EXAMPLES OF SIMPLE GAMES

5.5.1. River crossing game

The river crossing is dated back to at least the 9th century [Pressman89]. It's about a farmer, who went to a market and purchased a fox, a goose, and a bag of beans [WikiRiver]. On his way home, the farmer had to cross a river by boat, which could carry only the farmer and one of his 3 purchases. If left together without the farmer, the fox would eat the goose, or the goose would eat the beans. The challenge was to cross the river with all purchases intact.

An example of the solution is the 7 steps as follow:

1. Take the Goose over
2. Return
3. Take the beans over
4. Return with the goose
5. Take the fox over
6. Return
7. Take goose over

In this section, we will present the automatic search for solutions realized in PROLOG. This example is based on the lecture notes at [WebGao]. In the algorithm, a “state” is a set of 6 parameters:

1. Side of river fox is on
2. Side of river goose is on
3. Side of river bag of beans is on
4. Side of river farmer is on
5. Opposite of side of river farmer is currently on
6. Previous purchase carried across river

The program is as follow:

```
% Move across river procedures
move(state(Fox,Goose,Beans,X,Y,_),
over(man,Y),        % man crosses over by himself
state(Fox,Goose,Beans,Y,X,nothing)).
move(state(X,Goose,Beans,X,Y,_),
over(fox,Y),        % fox changes sides
state(Y,Goose,Beans,Y,X,fox)).
move(state(Fox,X,Beans,X,Y,_),
over(goose,Y),      % goose changes sides
state(Fox,Y,Beans,Y,X,goose)).
move(state(Fox,Goose,X,X,Y,_),
over(grain,Y),      % grain changes sides
state(Fox,Goose,Y,Y,X,grain)).

% nonredundant moves are when current move
% is different from previous move
nonreduncant(state(_,_,_,_,_,Prev_move),
state(_,_,_,_,_,Move)):- Move\==Prev_move.

% Procedure losesome: checks for invalid combinations.
% Fox eats goose, Man is on other side.
losesome(state(Side1,Side1,Beans,Man,_,_)):-
        Side1\==Man.
```

```
% Goose eats grain, Man is on other side.
losesome(state(Fox,Side1,Side1,Man,_,_)):-
        Side1\==Man.

% Procedure cross the river
cross(state(right,right,right,right,left,_)).

% final state
cross(State1):-
        move(State1,Move,State2),
        nonredundant(State1,State2),
        \+(losesome(State2)),
        cross(State2).

% Invoke with this clause:
start:-
        cross(state(left,left,left,left,right,nothing)).
```

5.5.2. Tic-tac-toe game

Tic-tac-toe (also known as *Xs and Os*) is a two players game on a 3×3 grid [WikiTic]. The players in turn write down X or O marks on the grid. The player, who succeeds in placing three of their marks in a horizontal, vertical, or diagonal row wins the game. The following example game is won by the first player (X) after his 4-th move.

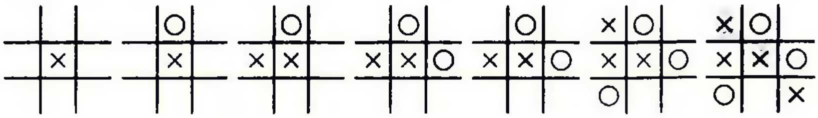


Figure 5.3. An example of a Game of Tic-tac-toe, won by X

Because of the simplicity of the game, it is often used as an example AI algorithm that deals with the searching of game states. It is straightforward to write a computer program to play tic-tac-toe, which knows all the possible different moves or even all the possible combinations of games.

In 1952, a British computer scientist Alexander S. Douglas implemented for the EDSAC computer at the University of Cambridge a version of Tic-Tac-Toe, becoming one of the first known video games [Wolf12]. The computer player could play perfect games of tic-tac-toe against a human opponent.

In this section, we present an example of Tic-Tac-Toe program in Prolog based on the lecture notes of [WebTanimoto03]:

```
% To play a game with the computer, type playo.
% To let the computer play with itself, type selfgame.


% Predicates that define the winning conditions:
win(Board, Player) :- rowwin(Board, Player).
win(Board, Player) :- colwin(Board, Player).
win(Board, Player) :- diagwin(Board, Player).
rowwin(Board, Player) :-
        Board=[Player,Player,Player,_,_,_,_,_,_].
rowwin(Board, Player) :-
        Board=[_,_,_,Player,Player,Player,_,_,_].
rowwin(Board, Player) :-
        Board=[_,_,_,_,_,_,Player,Player,Player].
colwin(Board, Player) :-
        Board=[Player,_,_,Player,_,_,Player,_,_].
colwin(Board, Player) :-
        Board=[_,Player,_,_,Player,_,_,Player,_].
colwin(Board, Player) :-
        Board=[_,_,Player,_,_,Player,_,_,Player].
diagwin(Board, Player) :-
        Board=[Player,_,_,_,Player,_,_,_,Player].
diagwin(Board, Player) :-
        Board=[_,_,Player,_,Player,_,Player,_,_].


% Predicate for alternating play in a "self" game:
other(x,o).
other(o,x).
```

```
game(Board, Player) :-
        win(Board, Player), !,
        write([player, Player, wins]).
game(Board, Player) :-
        other(Player,Otherplayer),
        move(Board,Player,Newboard),
        !,
        display(Newboard),
        game(Newboard,Otherplayer).

move([b,B,C,D,E,F,G,H,I], Player,
        [Player,B,C,D,E,F,G,H,I]).
move([A,b,C,D,E,F,G,H,I], Player,
        [A,Player,C,D,E,F,G,H,I]).
move([A,B,b,D,E,F,G,H,I], Player,
        [A,B,Player,D,E,F,G,H,I]).
move([A,B,C,b,E,F,G,H,I], Player,
        [A,B,C,Player,E,F,G,H,I]).
move([A,B,C,D,b,F,G,H,I], Player,
        [A,B,C,D,Player,F,G,H,I]).
move([A,B,C,D,E,b,G,H,I], Player,
        [A,B,C,D,E,Player,G,H,I]).
move([A,B,C,D,E,F,b,H,I], Player,
        [A,B,C,D,E,F,Player,H,I]).
move([A,B,C,D,E,F,G,b,I], Player,
        [A,B,C,D,E,F,G,Player,I]).
move([A,B,C,D,E,F,G,H,b], Player,
        [A,B,C,D,E,F,G,H,Player]).

display([A,B,C,D,E,F,G,H,I]) :-
        write([A,B,C]),nl,write([D,E,F]),nl,
        write([G,H,I]),nl,nl.

selfgame :- game([b,b,b,b,b,b,b,b,b],x).
```

```
% Predicates for playing a game with the user:
x_can_win_in_one(Board) :-
        move(Board, x, Newboard),
        win(Newboard, x).

% The predicate o respond generates the computer's
% (playing o) reponse from the current Board.
orespond(Board,Newboard) :-
        move(Board, o, Newboard),
        win(Newboard, o),
        !.
orespond(Board,Newboard) :-
        move(Board, o, Newboard),
        not(x_can_win_in_one(Newboard)).
orespond(Board,Newboard) :-
        move(Board, o, Newboard).
orespond(Board,Newboard) :-
        not(member(b,Board)),
        !,
        write('Cats game!'), nl,
        Newboard = Board.

% The translations from an integer description
% of x's move to a board transformation.
xmove([b,B,C,D,E,F,G,H,I],1,[x,B,C,D,E,F,G,H,I]).
xmove([A,b,C,D,E,F,G,H,I],2,[A,x,C,D,E,F,G,H,I]).
xmove([A,B,b,D,E,F,G,H,I],3,[A,B,x,D,E,F,G,H,I]).
xmove([A,B,C,b,E,F,G,H,I],4,[A,B,C,x,E,F,G,H,I]).
xmove([A,B,C,D,b,F,G,H,I],5,[A,B,C,D,x,F,G,H,I]).
xmove([A,B,C,D,E,b,G,H,I],6,[A,B,C,D,E,x,G,H,I]).
xmove([A,B,C,D,E,F,b,H,I],7,[A,B,C,D,E,F,x,H,I]).
xmove([A,B,C,D,E,F,G,b,I],8,[A,B,C,D,E,F,G,x,I]).
xmove([A,B,C,D,E,F,G,H,b],9,[A,B,C,D,E,F,G,H,x]).
xmove(Board, N, Board) :- write('Illegal move.\n')
```

```
% The 0-place predicate plays o
% starts a game with the user.
playo :- explain, playfrom([b,b,b,b,b,b,b,b,b]).
explain :-
        write('You play X by entering integer positions
                followed by a period.'),
        nl, display([1,2,3,4,5,6,7,8,9]).

playfrom(Board) :-
        win(Board, x), write('You win!').
playfrom(Board) :-
        win(Board, o), write('I win!').
playfrom(Board) :- read(N),
        xmove(Board, N, Newboard),
        display(Newboard),
        orespond(Newboard, Newnewboard),
        display(Newnewboard),
        playfrom(Newnewboard).
```

5.6. CONCLUSIONS

Artificial intelligence in games is defined by algorithms that perform certain actions, and sometimes also learn some of the player's behavior. It's not easy to create and balance snippets of code. The problem is to create an artificial intelligence working in such a way that it achieves its goals and is not perceived as an inhuman machine. Unfortunately, it is AI who is most often the most adventurous element of the game, where opponents run like crazy, unable to find a way in a beautifully graphically presented world.

Based on the analysis of these algorithms and their use in games and experiments, it can be seen that, for example, a skirmish with an opponent should be absorbing and difficult for the player due to its complexity and ingenuity, and not as often happens due to properly annoying gain of attributes and skills of the enemy. Frequently AI errors are used and stigmatized by players, and then they become a joke and complain that the game is not a challenge in its entirety or in a specific part.

That's why artificial intelligence is often linked to each other, so that the character's behavior is more natural, visually correct, and also satisfying to the player. Otherwise, the behavior of the AI while moving from one place to another is calculated, and decisions are made in combat or conversation. The game play also changes the behavior of the system - strategic or logical games do not use methods used in arcade games, and in racing games to conduct dialogues.

In the case of an experiment, further tests on the use of artificial intelligence for decision-making are indicated. Classic approach with a random selection of weights gives positive results. Further research is planned with the use of fuzzy logic in decision-making.

It is worth remembering that artificial intelligence in games will remain a script and only from the creators will depend on how much they will be expanded. It is important that artificial intelligence does not obscure the idea and game play present in the games, but only complements it in style.

Chapter 6: ARTIFICIAL NEURAL NETWORKS

The Wikipedia states "*The neural network (artificial neural network) is the generic name of mathematical structures and their software or hardware models that perform calculations or processing signals through the rows of elements performing a certain basic operation on their input, called neurons*". Each of us had a biology at school, so if we wanted to go into details, we could say that the neural network is a collection of simple processors (neurons) connected in some way. Neuron can have many inputs (synapses). It has only one output etc. The neural network has a learning process, which is an iterative (repetitive) process to adapt the parameters of the network. In this chapter we will present an introduction to artificial neural networks with their learning algorithms to adapt to a data set. Different types of networks such as MLP, Hopfield, BAM, Kohonen and their learning algorithms are discussed.

6.1. INTRODUCTION

The begin of artificial neural networks (ANN) date back to the 40' of the last century when the Perceptron model was developed to mimic the natural neurons in human and animal brain [McCulloch43] and the mechanism for memorizing information through the biological network was explained. Further development of this field of science resulted in the design and construction of an artificial neural network by Rosenblatt (in 1958). It was an elementary visual system that could be taught to recognize a limited class of patterns. In these years, the first applications of computers, including weather forecasting, mathematical formulas, or electrocardiogram analysis, were also tried.

After the publication in 1969 of Minsky and Papert's book, in which they proved that single-layered networks have finite applications, the neural network was shifted towards expert systems. The interest in ANN has returned in the mid-1980s due to the works showing that multi-layered, nonlinear neural networks have almost no limitations. At that time, the development of neurocomputers began, which was also influenced by advances in VLSI technology. Various types of multilayered network learning methods, such as back propagation of blanks, are also important. Another reason for the rise in popularity is the growing interest from Artificial Intelligence experts, to the crisis of traditional symbolic techniques developed in the '60, '70 and '80. In 1990s we could observe a growing popularity such as development of the application sector in medicine, industry, education and other fields.

6.1.1. What is a biological neuron?

The origin model of the neural network is the human brain which consists of billions of nerve cells between which there are even much higher number of connections. Nerve cells process generate impulses at a frequency of 1 - 100Hz. A neuron consists of a cell body, soma, a number of fibers called dendrites, and a single long fiber called the axon.

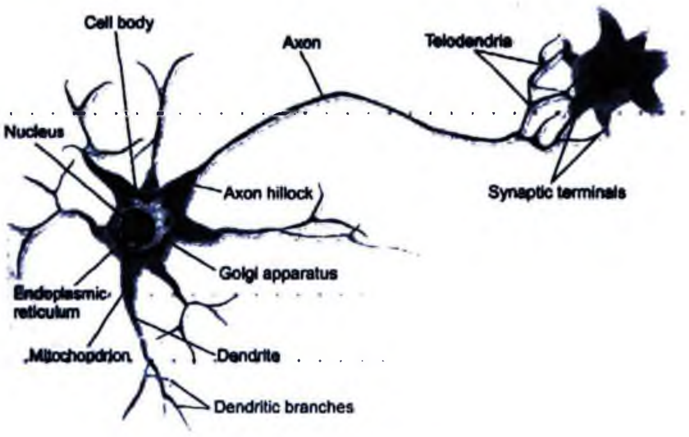


Figure 6.1. The structure of a biological neuron [WikiNeuron]

The cell membranes play key role in signal transmission; It consists in propagation cf the voltage difference between the interior and the outside of the cell. The principle of the action is when the dendritic stimuli sums up strong enough on the cell membrane the neuron generates an output impulse and this signal is transferred to another neuron through the axon.

6.1.2. Biological Neural Network

Neural network is a very simplified model of the brain. It consists of a large number of elements called neurons, which are simple, independent computing units connecting with each other, working in parallel. The output impulse from one neuron spreads out to form the input signals for other neurons. They are connected in a certain predefined way. With a huge number of neurons and interconnections between them, the brain can be seen as a highly complex and parallel information processing system. The information is processed in the whole network, rather than at specific locations.

6.1.3. Artificial neural networks

Researchers are trying to create artificial neural network (ANN) to resemble the human brain but they can do that in a minimum degree. The ANNs are capable of 'learning', that is, they can adapt their parameters to improve their performance. With a good "training", ANNs can also work good with new samples that they have not yet encountered. For example, researchers have designed ANNs to recognize hand-written characters, identify human speech, analyze and classify the bio-signals of the patients. Not only that, once setup properly, like other machines, the ANNs can work on cases that human experts fail to deal with.

An artificial neural network consists of a number neurons, which are similar to the biological neurons in the brain. The neurons are connected and the connections have their own weights. Each neuron receives a number of input signals and produces an output signal. The output signal is transmitted to a number of other neurons in the network.

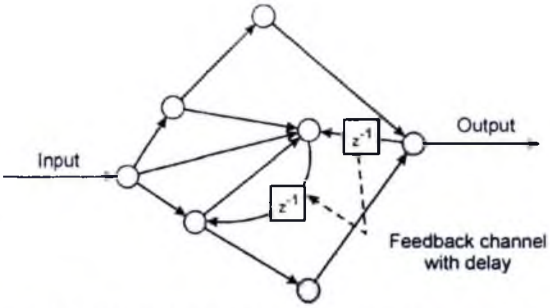


Figure 6.2. Architecture of a typical artificial neural network

6.2. THE ARTIFICIAL NEURON MODEL

A neuron receives several signals from its inputs, computes a new activation level and sends it as an output signal. The input signal can be raw signal from sensory neurons or outputs of other neurons. The output signal can be an input to other neurons. Figure 6.3 shows a typical neuron.

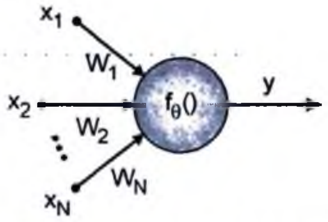


Figure 6.3. Diagram of a neuron

In 1943, Warren McCulloch and Walter Pitts proposed a very simple model of the neuron, in which it computes the weighted sum of the input signals and compares with a threshold value, θ. If the weighted sum of inputs is greater than the threshold, the neuron becomes activated and its output is +1 [McCulloch43], otherwise its output is 0. This neuron model is also called the perceptron.

Mathematically, the weighted sum of inputs u and the output y of a neuron is defined as:

$$u = \sum_{i=1}^{N} x_i \cdot W_i$$
$$y = f_\theta(u) = \begin{cases} +1 & \text{if } u \geq \theta \\ 0 & \text{if } u < \theta \end{cases} \tag{6.1}$$

where u is the weighted sum of inputs to the neuron, x_i is the i-th input signal, W_i is the weight of i-th input, N is the number of neuron inputs, and y is the output of the neuron.

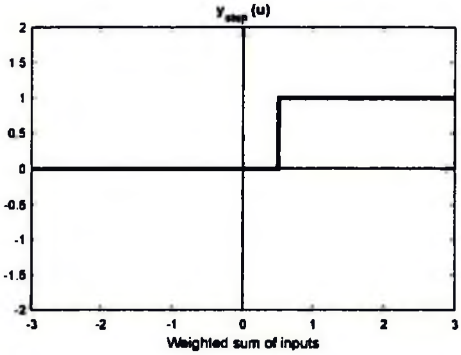


Figure 6.4. Thresholded step activation function of the McCulloch – Pitts neuron (θ=0.5)

Other possible activation functions are, for example: the sign, sigmoid, tangential sigmoid and linear functions. Some of them are illustrated in **Figure** 6.5, where:

- Sign function: $y_{sign}(u) = \begin{cases} +1 & \text{if } u > 0 \\ 0 & \text{if } u = 0, \\ -1 & \text{if } u < 0 \end{cases}$
- Sigmoid function: $y_{sigmoid}(u) = \dfrac{1}{1+e^{-u}}$
- Tangent hyperbolic (also known as tansig function): $y_{tansig}(u) = \dfrac{e^{u} - e^{-u}}{e^{u} + e^{-u}}$

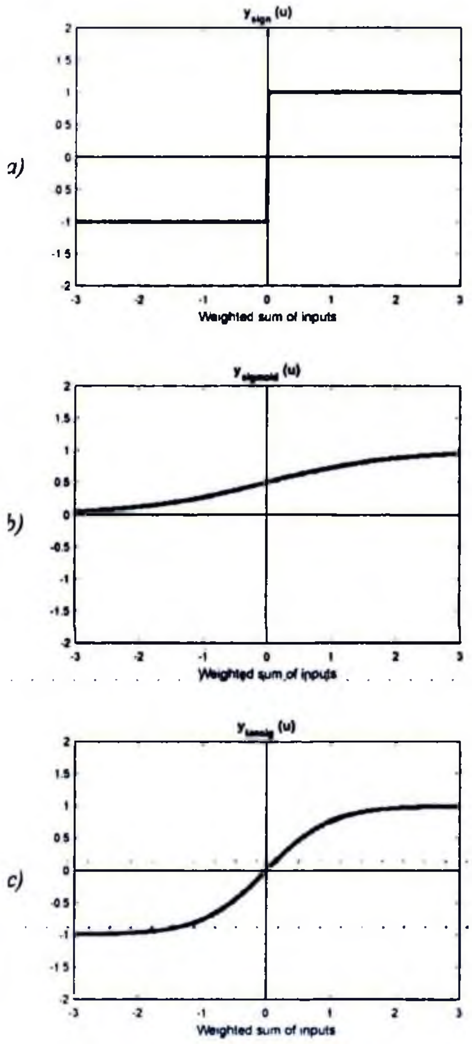


Figure 6.5. Examples of activation functions of a neuron: a) sign function, b) sigmoid function, c) tansig function

With the structure and principle of operation as described above, neurons are practically a nonlinear unit. What makes neurons different from other nonlinear mathematical models is that it comes with *learning algorithms* and the process of assessing the quality of neurons through the use of new data samples. It can be said that each intelligence model in general and the neural model in particular have two specific processes, the learning process and the testing process. Here we will explore the learning process for the McCulloch-Pitts neural model.

6.3. TRAINING THE PERCEPTRON

Single neuron with step activation function is able to solve linearly separable problems, i.e. those whose results can be separated in a straight-line graph. Learning process is considered to be the most fascinating phenomenon associated with neural networks. It is worth to mention the foundations of machine learning. During the learning process, the weights and the threshold of a neuron can be adapted to adjust the output of the neurons. In case of a 'supervised learning', it consists in calculating error at the neuron's output and using that error for weights and threshold corrections.

In order to carry out the learning process we need a set of sample data. That is, pairs of input data and desired (destination) output from the neuron. With a set of p samples $\{\mathbf{x}_i, d_i\}$, $i = 1, \ldots, p$, where $\mathbf{x}_i \in \mathbb{R}^N$ is the input vector, $d_i \in \mathbb{R}$ is the desired (destination) value, we need to find the weights and the threshold of the neuron with the transfer function $f()$ such:

$$\forall i, f(\mathbf{x}_i) \approx d_i \tag{6.2}$$

or we can use the target function to be minimized:

$$E = \frac{1}{2}\sum_{i=1}^{p}\left(f(\mathbf{x}_i) - d_i\right)^2 \rightarrow \min \tag{6.3}$$

Since neural models are usually models with nonlinear transmission functions, this is in fact a nonlinear optimization problem in multidimensional space. These problems are difficult to solve directly with a global solution, so usually we have iterative algorithms for local minima approach. When the iteration is finished, it is said that the learning model has acquired the information of the sample, which in part has learned about the problem from the surrounding environment. If the target function value is acceptable, we will stop the training process and move to the testing phase. However, due to the fact that the training sample set is usually limited in size, it is not possible to cover all possible cases, which means that the learning process cannot be complete. Therefore, it is necessary to assess the performance of the model by using new cases. This is similar to the idea of examination after training course.

For demonstration, we will consider the learning of a neuron to act according to the binary OR operation. In this case we have the following set of patterns:

Sample nr	x_1	x_2	d
1	0	0	0
2	0	1	1
3	1	0	1
4	1	1	1

The neuron to which we could attach such a collection should have 2 synaptic input connections and 1 output. Each of these samples (1, 2, 3 or 4) is put on the neuron.

The process of learning need to adapt for several times (also called *epochs*). With each iteration the error will be less and less. For each sample we calculate the error as follows:

$$\forall i = 1, 2, \ldots, p : e_i = y_i - d_i = f\left(x_{i1}W_1 + x_{i2}W_2 + W_0\right) - d_i \qquad (6.4)$$

where d_i is the expected response, y_i – the value that appeared on the output of the network. The i index indicates the sample number from our set. The error will be differen: for each of samples. The total target cost function will be

$$E = \frac{1}{2}\sum_{i=1}^{p} e_i^2 \qquad (6.5)$$

Once we have all the errors, the parameters of the neuron are adapted using these values [Linh14]:

$$W_\alpha^{(t+1)} = W_\alpha^{(t)} - \eta \frac{\partial E}{\partial W_\alpha} \quad for \quad \alpha = 0,1,\ldots,N \tag{6.6}$$

where η – the learning rate. We can speed up or slow down the changes by setting the value of η. It should accept values between 0 and 1. With bigger value the learning will be faster, more aggressive, but less precise. And lower values will make learning slower, laborious, but accurate and precise.

The size of the changes on synapses is proportional also to the value of input information. With the target function defined in Eq. (6.3) and (6.5), the gradient is equal:

$$\begin{aligned}\frac{\partial E}{\partial W_\alpha} &= \sum_{i=1}^{p}(y_i - d_i)\frac{\partial y_i}{\partial W_\alpha} \\ &= \sum_{i=1}^{p}(y_i - d_i) f'\left(\sum_{j=0}^{N} W_j x_{ij}\right)\frac{\partial\left(\sum_{j=0}^{N} W_j x_{ij}\right)}{\partial W_\alpha} \\ &= \sum_{i=1}^{p}(y_i - d_i) f'\left(\sum_{j=0}^{N} W_j x_{ij}\right) x_{i\alpha}\end{aligned} \tag{6.7}$$

Let take one example for the OR function, the sample number 2 with $x_1 = 0;\ x_2 = 1;\ d = 1;$ the initial weights $[\mathbf{W}]^{(0)} = \left[W_0^{(0)}\ W_1^{(0)}\ W_2^{(0)}\right]$ are $[-0.3\ \ 0.6\ \ 0.2]$; the neuron uses transfer function logsig(). The steps are following. First, the output signal is calculated:

$$y^{(0)} = f\left(W_0^{(0)} x_0 + W_1^{(0)} x_1 + W_2^{(0)} x_2\right) = f(0.8) = 0,69.$$

The error $E^{(0)} = \frac{1}{2}\left(y^{(0)} - d\right)^2 = \frac{1}{2}(0,69 - 1)^2 = 0,0481$

The gradients:

$$\left.\frac{\partial E}{\partial W_0}\right|_{[\mathrm{W}]=[\mathrm{W}]^{(0)}} = \left(y^{(0)} - d\right) f'\left(W_0^{(0)} + W_1^{(0)}x_1 + W_2^{(0)}x_2\right) x_0$$

$$= \left(y^{(0)} - d\right) y^{(0)} \left(1 - y^{(0)}\right) x_0$$

$$= (0,69 - 1) \cdot 0,69 \cdot (1 - 0,69) \cdot 0 = 0$$

$$\left.\frac{\partial E}{\partial W_1}\right|_{[\mathrm{W}]=[\mathrm{W}]^{(0)}} = \left(y^{(0)} - d\right) f'\left(W_0^{(0)} + W_1^{(0)}x_1 + W_2^{(0)}x_2\right) x_1$$

$$= \left(y^{(0)} - d\right) y^{(0)} \left(1 - y^{(0)}\right) x_1$$

$$= (0,69 - 1) \cdot 0,69 \cdot (1 - 0,69) \cdot 1 = -0,0663$$

$$\left.\frac{\partial E}{\partial W_2}\right|_{[\mathrm{W}]=[\mathrm{W}]^{(0)}} = \left(y^{(0)} - d\right) f'\left(W_0^{(0)} + W_1^{(0)}x_1 + W_2^{(0)}x_2\right) x_2$$

$$= \left(y^{(0)} - d\right) y^{(0)} \left(1 - y^{(0)}\right) x_2$$

$$= (0,69 - 1) \cdot 0,69 \cdot (1 - 0,69) \cdot 1 = -0,0663$$

Let's use the learning coefficient $\eta = 1$ we get the new weights:

$$W_0^{(1)} = W_0^{(0)} - \eta \left.\frac{\partial E}{\partial W_0}\right|_{[\mathrm{W}]=[\mathrm{W}]^{(0)}} = -0.3 + 1 \cdot 0 = -0.3$$

$$W_1^{(1)} = W_1^{(0)} - \eta \left.\frac{\partial E}{\partial W_1}\right|_{[\mathrm{W}]=[\mathrm{W}]^{(0)}} = 0,6 + 1 \cdot 0,0663 = 0,666$$

$$W_2^{(1)} = W_2^{(0)} - \eta \left.\frac{\partial E}{\partial W_2}\right|_{[\mathrm{W}]=[\mathrm{W}]^{(0)}} = 0,2 + 1 \cdot 0,0663 = 0,266$$

With this updated weights, the new output value is equal:

$$y^{(1)} = f\left(W_0^{(1)}x_0 + W_1^{(1)}x_1 + W_2^{(1)}x_2\right) = f(0.932) = 0,718.$$

The error after the update:

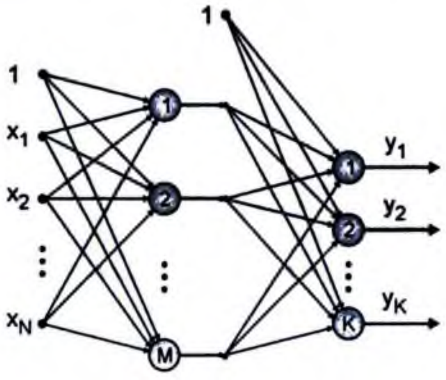
$$E^{(1)} = \frac{1}{2}\left(y^{(1)} - d\right)^2 = \frac{1}{2}(0,718 - 1)^2 = 0,0398 < 0,0481 = E^{(0)}$$

So we can see that the error was reduced after 1 step of learning the neuron. Repeat the algorithm for more iterations (also called epochs), we will reach a local minimum of the cost function.

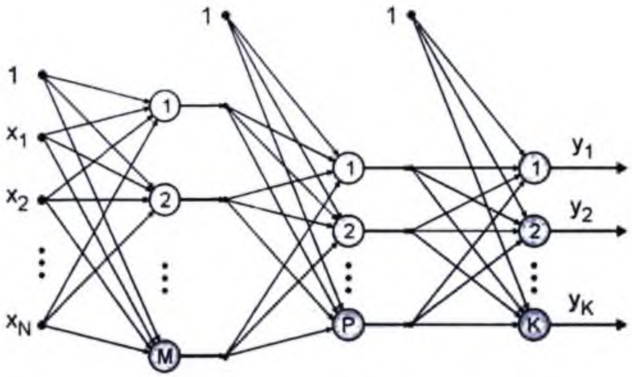
6.4. MULTILAYER PERCEPTRONS (MLP)

A single neuron has the ability to adapt to a nonlinear function, but with that simple structure a single neuron can do not much. It's similar to our brain. In order to function normally we need a huge number of neurons connected with each other to increase to ability to process the collected information. Those neurons will form a *neural network*.

A multilayer perceptron is a feedforward neural network with one or more hidden layers. Typically, the network consists of an input layer, at least one hidden layer and an output layer, where the neurons are in the hidden and output layers only. The multilayer perceptrons with one and two hidden layers are shown in Fig. 6.6.



(a)



(b)

Figure 6.6. Multilayer perceptron with one (a) and two hidden layers (b)

Classical MLPs don't have limitation on the number of hidden layers, but most of the researchers based on the Komogorov theorem use 1 or 2 hidden layers only, among which the one hidden layer configuration is much more popular.

With one hidden layer, the MLP can be characterized by triple (N, M, K) where $N-$ the number of inputs, $M-$ the number of neurons in hidden layer, $K-$ the number of outputs.

Denote the weights connecting the input to the hidden neurons by W_{ij} where i – the index of target neuron, $i = 1, 2, \dots, M$; j – the index of the source neuron $j = 0, 1, 2, \dots N$.

Denote the weights connecting the hidden neurons to the output by V_{ij} where i – the index of output neuron, $i = 1, 2, \dots, K$; j – the index of the hidden neuron $j = 0, 1, \dots M$.

Let the transfer function of the hidden neurons is $f_1()$, the transfer function of the output neurons is $f_2()$. When there is an input vector $\mathbf{x} = [x_1, x_2, \dots, x_N] \in \Re^N$ (the bias input for all neurons is $x_0 = 1$) the outputs are calculated in a *forward propagation* manner:

- Calculate the sums of weighted inputs of the i-th hidden neuron

$$u_i = \sum_{j=0}^{M} x_j W_{ij} \tag{6.8}$$

for $i = 1, 2, \ldots, M$.

- The output of the i-th hidden neuron:

$$v_i = f_1(u_i) \tag{6.9}$$

for $i = 1, 2, \ldots, M$ (for simplification, we let the output neurons have a constant bias $v_0 = 1$).

- Calculate the sums of weighted inputs of the k-th output neuron:

$$g_k = \sum_{i=0}^{M} v_i V_{ki} \tag{6.10}$$

for $k = 1, 2, \ldots, K$.

- And finally, the output of the k-th neuron is:

$$y_k = f_2(g_k) \tag{6.11}$$

for $k = 1, 2, \ldots, K$.

Combining all the above steps, the MLP is a strongly nonlinear system with the transfer function equal:

$$\begin{aligned} y_k &= f_2(g_k) = f_2\left(\sum_{i=0}^{M} v_i V_{ki}\right) = f_2\left(\sum_{i=0}^{M} f_1(u_i) V_{ki}\right) \\ &= f_2\left(\sum_{i=0}^{M}\left[f_1\left(\sum_{j=0}^{N} x_j W_{ij}\right) V_{ki}\right]\right) \end{aligned} \tag{6.12}$$

Learning in a multilayer network proceeds the same way as for a perceptron. A training set of input patterns is presented to the network. The network computes its output patterns using above formulas, and the difference between actual and desired output patterns are used to adjust the weights. In a perceptron, there is only one output, but in a MLP, there can be more than one outputs. The weights are modified as the error is propagated. Similar to the neurons, the weights of MLP can be updated using the gradient formulas:

$$\begin{cases} W_{\alpha\beta}^{(t+1)} &= W_{\alpha\beta}^{(t)} - \eta \dfrac{\partial E}{\partial W_{\alpha\beta}} \\ V_{\alpha\beta}^{(t+1)} &= V_{\alpha\beta}^{(t)} - \eta \dfrac{\partial E}{\partial V_{\alpha\beta}} \end{cases} \tag{6.13}$$

The detailed explorations of these formulas can be found in [Haykin92].

6.5. RECURRENT NEURAL NETWORKS

The MLP trained with the back-propagation algorithm are used for various applications with good performance. But the MLP has a drawback. It doesn't have memory, i.e. when a new input vector is presented into the network, the output calculation is not affected by previous values of inputs and outputs. To have a network with memory, we need a different type of network: a recurrent neural network.

6.5.1. Hopfield network

A recurrent neural network has feedback loops from its outputs to its inputs. The presence of such loops has a great impact on the way the output is calculated, on the stability of the network and on the algorithm of network training.

The Hopfield network has the structure as shown on Fig. 6.7. The output of each neuron is fed back to the inputs of all other neurons (there is no self-feedback). After applying a new input vector, the network output is calculated and fed back to adjust the input. Then the output is re-calculated again, and the process is repeated until the output becomes constant. Successive iterations do not always converge, but on the contrary may lead to chaotic behavior. In such a case, the network output would not become a constant, and the network is said to be unstable.

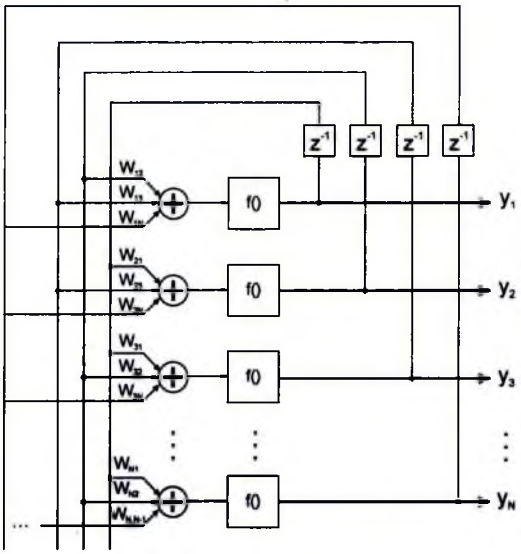


Figure 6.7. Single-layer N-neuron Hopfield network

The Hopfield network usually uses McCulloch and Pitts neurons with the *sign activation function* as its computing element. If the neuron's weighted sum of inputs is less than zero, the output is -1; if the weighted sum of inputs is greater than zero, the output is 1. A difference introduced here is if the neuron's weighted input is exactly zero, its output remains unchanged.

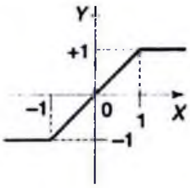
$$y_{sign}(u) = \begin{cases} +1 & \text{if } u > 0 \\ -1 & \text{if } u < 0 \\ 0 & \text{if } u = 0 \end{cases} \tag{6.14}$$

The sign activation function may be replaced with a saturated linear function, which acts as a pure linear function within the region [-1; 1] and as a sign function outside this region. The saturated linear function is shown in Figure 6.8.

The current state of the network is determined by the current outputs of all neurons and the state can be defined by the *state vector* as:

$$\mathbf{Y} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_N \end{bmatrix} = [y_1 \;\; y_2 \;\; \dots \;\; y_N]^T \qquad (6.15)$$

Saturated linear function



$$y_{saturated}(u) = \begin{cases} +1 & if \;\; u > 1 \\ -1 & if \;\; u < -1 \\ u & if \;\; -1 \le u \le 1 \end{cases}$$

Figure 6.8. The saturated linear activation function

In the Hopfield network, synaptic weights between neurons can be represented in matrix form as follows:

$$\mathbf{W} = \sum_{i=1}^{p} \mathbf{Y}_i \cdot \mathbf{Y}_i^T \qquad (6.16)$$

where p is the number of states to be memorized by the network, $\mathbf{Y}_i$ is the N-dimensional binary vector.

Suppose, for instance, that our network is required to memorize four samples [-1, -1, 1, 1, 1]; [1, 1, -1, -1, 1]; [1, -1, -1, 1, -1] and [-1, 1, -1, 1, -1] or the four 5-bit vectors: $\mathbf{Y}_1 = \begin{bmatrix} -1 \\ -1 \\ 1 \\ 1 \\ 1 \end{bmatrix}$; $\mathbf{Y}_2 = \begin{bmatrix} 1 \\ 1 \\ -1 \\ -1 \\ 1 \end{bmatrix}$; $\mathbf{Y}_3 = \begin{bmatrix} 1 \\ -1 \\ -1 \\ 1 \\ -1 \end{bmatrix}$ and $\mathbf{Y}_4 = \begin{bmatrix} -1 \\ 1 \\ -1 \\ 1 \\ -1 \end{bmatrix}$.

Thus, we can now determine the weight matrix as follows:

$$\mathbf{W} = \mathbf{Y}_1\mathbf{Y}_1^T + \mathbf{Y}_2\mathbf{Y}_2^T + \mathbf{Y}_3\mathbf{Y}_3^T + \mathbf{Y}_4\mathbf{Y}_4^T = \begin{bmatrix} 4 & 0 & -2 & -2 & 0 \\ 0 & 4 & -2 & -2 & 0 \\ -2 & -2 & 4 & 0 & 2 \\ -2 & -2 & 0 & 4 & -2 \\ 0 & 0 & 2 & -2 & 4 \end{bmatrix}$$

With the weight matrix **W** is calculated, the network can be tested with the same 4 input vectors to check if it is stable for all these samples (the output is equal exactly the input). The outputs are calculated using the formula:

$$\mathbf{Y}_i = sign(\mathbf{W}\cdot\mathbf{X}_i - \boldsymbol{\theta}) = sign(\mathbf{W}\cdot\mathbf{X}_i),\ i = 1, 2, \ldots, p \qquad (6.17)$$

where $\boldsymbol{\theta}$ is the threshold (0 for this example). For the 4 cases we have:

$$sign(\mathbf{W}\cdot\mathbf{X}_1) = sign\left\{\begin{bmatrix} 4 & 0 & -2 & -2 & 0 \\ 0 & 4 & -2 & -2 & 0 \\ -2 & -2 & 4 & 0 & 2 \\ -2 & -2 & 0 & 4 & -2 \\ 0 & 0 & 2 & -2 & 4 \end{bmatrix}\begin{bmatrix} -1 \\ -1 \\ 1 \\ 1 \\ 1 \end{bmatrix}\right\} = sign\left\{\begin{bmatrix} -8 \\ -8 \\ 10 \\ 6 \\ 4 \end{bmatrix}\right\} = \begin{bmatrix} -1 \\ -1 \\ 1 \\ 1 \\ 1 \end{bmatrix} = \mathbf{Y}_1$$

$$sign(\mathbf{W}\cdot\mathbf{X}_2) = sign\left\{\begin{bmatrix} 4 & 0 & -2 & -2 & 0 \\ 0 & 4 & -2 & -2 & 0 \\ -2 & -2 & 4 & 0 & 2 \\ -2 & -2 & 0 & 4 & -2 \\ 0 & 0 & 2 & -2 & 4 \end{bmatrix}\begin{bmatrix} 1 \\ 1 \\ -1 \\ -1 \\ 1 \end{bmatrix}\right\} = sign\left\{\begin{bmatrix} 8 \\ 8 \\ -6 \\ -10 \\ 4 \end{bmatrix}\right\} = \begin{bmatrix} 1 \\ 1 \\ -1 \\ -1 \\ 1 \end{bmatrix} = \mathbf{Y}_2$$

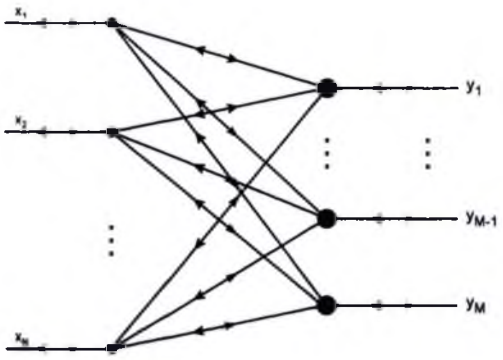
$$sign(\mathbf{W}\cdot\mathbf{X}_3) = sign\left\{\begin{bmatrix} 4 & 0 & -2 & -2 & 0 \\ 0 & 4 & -2 & -2 & 0 \\ -2 & -2 & 4 & 0 & 2 \\ -2 & -2 & 0 & 4 & -2 \\ 0 & 0 & 2 & -2 & 4 \end{bmatrix}\begin{bmatrix} 1 \\ -1 \\ -1 \\ 1 \\ -1 \end{bmatrix}\right\} = sign\left\{\begin{bmatrix} 4 \\ -4 \\ -6 \\ 6 \\ -8 \end{bmatrix}\right\} = \begin{bmatrix} 1 \\ -1 \\ -1 \\ 1 \\ -1 \end{bmatrix} = \mathbf{Y}_3$$

$$sign(\mathbf{W}\cdot\mathbf{X}_4) = sign\left\{\begin{bmatrix} 4 & 0 & -2 & -2 & 0 \\ 0 & 4 & -2 & -2 & 0 \\ -2 & -2 & 4 & 0 & 2 \\ -2 & -2 & 0 & 4 & -2 \\ 0 & 0 & 2 & -2 & 4 \end{bmatrix}\begin{bmatrix} -1 \\ 1 \\ -1 \\ 1 \\ -1 \end{bmatrix}\right\} = sign\left\{\begin{bmatrix} -4 \\ 4 \\ -6 \\ 6 \\ -8 \end{bmatrix}\right\} = \begin{bmatrix} -1 \\ 1 \\ -1 \\ 1 \\ -1 \end{bmatrix} = \mathbf{Y}_4$$

As we see, $\mathbf{Y}_i = \mathbf{X}_i$ for all the 4 cases, which means these cases are *stable*.

6.5.2. Bidirectional Associative Memory Network

Bidirectional Associative Memory (BAM) proposed in [Kosko87] is a heteroassociative network. It associates patterns from one set (set A) to patterns from the second set (set B). The BAM architecture is shown in Figure 6.9. It consists of two layers: an input and an output layer.



$$\longrightarrow : \mathbf{y}^{(t)} = \mathbf{W}^T \cdot \mathbf{x}^{(t)}$$
$$\longleftarrow : \mathbf{x}^{(t+1)} = \mathbf{W} \cdot \mathbf{y}^{(t)}$$

Figure 6.9. BAM network operations: (a) forward direction; (b) backward direction

The input vector $\mathbf{X}_i^{(0)}$ is applied to the transpose of weight matrix $\mathbf{W}^T$ to produce an output vector $\mathbf{Y}_i^{(0)}$, as illustrated in Fig. 6.9. Then, the output vector $\mathbf{Y}_i^{(0)}$ is applied to the weight matrix $\mathbf{W}$ to produce a new input vector $\mathbf{X}_i^{(1)}$. This process is repeated until input and output vectors become unchanged, or in other words, the BAM network reaches a stable state:

$$\begin{aligned} \mathbf{Y}_i^{(t)} &= \mathbf{W}^T \cdot \mathbf{X}_i^{(t)} = \mathbf{Y}_i^{(t-1)} \\ \mathbf{X}_i^{(t+1)} &= \mathbf{W} \cdot \mathbf{Y}_i^{(t)} = \mathbf{X}_i^{(t)} \end{aligned} \tag{6.18}$$

The BAM network can generalize (and correct) the outputs when the inputs are incomplete or noised.

To develop the BAM, we need to create a correlation matrix for each pattern pair we want to store. The correlation matrix is the matrix product of the input vector **X**, and the transpose of the output vector $\mathbf{Y}^T$. The BAM weight matrix is the sum of all correlation matrices, that is,

$$\mathbf{W} = \sum_{i=1}^{p} \mathbf{X}_i \mathbf{Y}_i^T \tag{6.19}$$

where p is the number of pattern pairs to be stored in the BAM.

Like a Hopfield network, the BAM usually uses McCulloch and Pitts neurons with the sign activation function. The BAM training algorithm can be presented as follows.

Step 1: Storage

The BAM is required to store p pairs of patterns. For **example**, we may wish to store four vectors

$$\mathbf{Y}_1 = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}, \mathbf{Y}_2 = \begin{bmatrix} -1 \\ 1 \\ -1 \end{bmatrix}, \mathbf{Y}_3 = \begin{bmatrix} 1 \\ -1 \\ -1 \end{bmatrix} \text{ and } \mathbf{Y}_4 = \begin{bmatrix} -1 \\ -1 \\ 1 \end{bmatrix}$$

corresponding to the 4 vectors we have in the example for **Hopfield network** above:

$$\mathbf{X}_1 = \begin{bmatrix} -1 \\ -1 \\ 1 \\ 1 \\ 1 \end{bmatrix}; \mathbf{X}_2 = \begin{bmatrix} 1 \\ 1 \\ -1 \\ -1 \\ 1 \end{bmatrix}; \mathbf{X}_3 = \begin{bmatrix} 1 \\ -1 \\ -1 \\ 1 \\ -1 \end{bmatrix} \text{ and } \mathbf{X}_4 = \begin{bmatrix} -1 \\ 1 \\ -1 \\ 1 \\ -1 \end{bmatrix}.$$

In this case, the BAM input layer will have 2 neuronsand the **output** layer has 3 neurons. The weight matrix is determined as [Kosko88]:

$$\mathbf{W} = \sum_{i=1}^{4} \mathbf{X}_i \cdot \mathbf{Y}_i^T = \begin{bmatrix} 0 & 0 & -4 \\ -4 & 0 & 0 \\ 2 & 2 & 2 \\ 2 & -2 & 2 \\ 0 & 4 & 0 \end{bmatrix}$$

Step 2: Testing

The BAM should be able to receive any vector from set A and retrieve the associated vector from set B, and receive any vector from set B and retrieve the associated vector from set A. Thus, first we need to confirm that the BAM is able to recall $\mathbf{Y}_i$ when presented with $\mathbf{X}_i$. That is,

$$\mathbf{Y}_i = sign(\mathbf{W}^T \cdot \mathbf{X}_i), \quad i = 1, 2, \ldots, M \tag{6.20}$$

$$\mathbf{X}_i = sign\left(\mathbf{W} \cdot \mathbf{Y}_i\right), \quad i = 1, 2, \ldots, M \tag{6.21}$$

For the 1st pair of samples:

$$sign(\mathbf{W}^T \cdot \mathbf{X}_1) = sign\left\{ \begin{bmatrix} 0 & -4 & 2 & 2 & 0 \\ 0 & 0 & 2 & -2 & 4 \\ -4 & 0 & 2 & 2 & 0 \end{bmatrix} \begin{bmatrix} -1 \\ -1 \\ 1 \\ 1 \\ 1 \end{bmatrix} \right\}$$

$$= sign\left\{ \begin{bmatrix} 8 \\ 4 \\ 8 \end{bmatrix} \right\} = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} = \mathbf{Y}_1$$

$$sign(\mathbf{W} \cdot \mathbf{Y}_1) = sign\left\{ \begin{bmatrix} 0 & 0 & -4 \\ -4 & 0 & 0 \\ 2 & 2 & 2 \\ 2 & -2 & 2 \\ 0 & 4 & 0 \end{bmatrix} \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} \right\} = sign\left\{ \begin{bmatrix} -4 \\ -4 \\ 6 \\ 2 \\ 4 \end{bmatrix} \right\} = \begin{bmatrix} -1 \\ -1 \\ 1 \\ 1 \\ 1 \end{bmatrix} = \mathbf{X}_1$$

For the 2nd pair of samples:

$$sign(\mathbf{W}^T \cdot \mathbf{X}_2) = sign\left\{\begin{bmatrix} 0 & -4 & 2 & 2 & 0 \\ 0 & 0 & 2 & -2 & 4 \\ -4 & 0 & 2 & 2 & 0 \end{bmatrix}\begin{bmatrix} 1 \\ 1 \\ -1 \\ -1 \\ 1 \end{bmatrix}\right\}$$

$$= sign\left\{\begin{bmatrix} -8 \\ 4 \\ -8 \end{bmatrix}\right\} = \begin{bmatrix} -1 \\ 1 \\ -1 \end{bmatrix} = \mathbf{Y}_2$$

$$sign(\mathbf{W} \cdot \mathbf{Y}_2) = sign\left\{\begin{bmatrix} 0 & 0 & -4 \\ -4 & 0 & 0 \\ 2 & 2 & 2 \\ 2 & -2 & 2 \\ 0 & 4 & 0 \end{bmatrix}\begin{bmatrix} -1 \\ 1 \\ -1 \end{bmatrix}\right\} = sign\left\{\begin{bmatrix} 4 \\ 4 \\ -2 \\ -6 \\ 4 \end{bmatrix}\right\} = \begin{bmatrix} 1 \\ 1 \\ -1 \\ -1 \\ 1 \end{bmatrix} = \mathbf{X}_2$$

For the 3rd pair of samples:

$$sign(\mathbf{W}^T \cdot \mathbf{X}_3) = sign\left\{\begin{bmatrix} 0 & -4 & 2 & 2 & 0 \\ 0 & 0 & 2 & -2 & 4 \\ -4 & 0 & 2 & 2 & 0 \end{bmatrix}\begin{bmatrix} 1 \\ -1 \\ -1 \\ 1 \\ -1 \end{bmatrix}\right\}$$

$$= sign\left\{\begin{bmatrix} 4 \\ -8 \\ -4 \end{bmatrix}\right\} = \begin{bmatrix} 1 \\ -1 \\ -1 \end{bmatrix} = \mathbf{Y}_3$$

$$sign(\mathbf{W} \cdot \mathbf{Y}_3) = sign\left\{\begin{bmatrix} 0 & 0 & -4 \\ -4 & 0 & 0 \\ 2 & 2 & 2 \\ 2 & -2 & 2 \\ 0 & 4 & 0 \end{bmatrix}\begin{bmatrix} 1 \\ -1 \\ -1 \end{bmatrix}\right\} = sign\left\{\begin{bmatrix} 4 \\ -4 \\ -2 \\ 2 \\ -4 \end{bmatrix}\right\} = \begin{bmatrix} 1 \\ -1 \\ -1 \\ 1 \\ -1 \end{bmatrix} = \mathbf{X}_3$$

For the 4th pair of samples:

$$sign(\mathbf{W}^T \cdot \mathbf{X}_4) = sign\left\{\begin{bmatrix} 0 & -4 & 2 & 2 & 0 \\ 0 & 0 & 2 & -2 & 4 \\ -4 & 0 & 2 & 2 & 0 \end{bmatrix}\begin{bmatrix} -1 \\ 1 \\ -1 \\ 1 \\ -1 \end{bmatrix}\right\}$$

$$= sign\left\{\begin{bmatrix} -4 \\ -8 \\ 4 \end{bmatrix}\right\} = \begin{bmatrix} -1 \\ -1 \\ 1 \end{bmatrix} = \mathbf{Y}_4$$

$$sign(\mathbf{W} \cdot \mathbf{Y}_4) = sign\left\{\begin{bmatrix} 0 & 0 & -4 \\ -4 & 0 & 0 \\ 2 & 2 & 2 \\ 2 & -2 & 2 \\ 0 & 4 & 0 \end{bmatrix}\begin{bmatrix} -1 \\ -1 \\ 1 \end{bmatrix}\right\} = sign\left\{\begin{bmatrix} -4 \\ 4 \\ -2 \\ 2 \\ -4 \end{bmatrix}\right\} = \begin{bmatrix} -1 \\ 1 \\ -1 \\ 1 \\ -1 \end{bmatrix} = \mathbf{X}_4$$

In our example, all four pairs are recalled perfectly, and we can proceed to the next step.

Step 3: Retrieval

Present an unknown vector (new) **X** to the BAM and retrieve a stored association. That is $\mathbf{X}_{new} \neq \mathbf{X}_i,\ \forall i = 1, 2, \ldots, p$. The new input may present a corrupted or incomplete version of a pattern from set **X** (or from set **Y**) stored in the BAM.

(a) Initialize the BAM retrieval algorithm by setting

$$\mathbf{X}(0) = \mathbf{X}_{new}, p = 0$$

and calculate the BAM output at iteration p

$$\mathbf{Y}(p) = sign\left[\mathbf{W}^T \cdot \mathbf{X}(p)\right]$$

(b) Update the input vector X(p):

$$\mathbf{X}(p+1) = sign\left[\mathbf{W} \cdot \mathbf{Y}(p)\right]$$

and repeat the iteration until equilibrium, when input and output vectors remain unchanged with further iterations. The input and output patterns will then represent an associated pair.

The BAM is unconditionally stable [Kosko88]. This means that any set of associations can be learned without risk of instability.

Let us now return to our example. Let test with $\mathbf{X}_{new}$ equals:

$$\mathbf{X}_{new} = \begin{bmatrix} 1 \\ -1 \\ -1 \\ -1 \\ 1 \end{bmatrix} = \mathbf{X}(0)$$

$$\mathbf{Y}(0) = sign\left[\mathbf{W}^T \cdot \mathbf{X}(0)\right] = sign\left\{\begin{bmatrix} 0 \\ 4 \\ -8 \end{bmatrix}\right\} = \begin{bmatrix} 0 \\ 1 \\ -1 \end{bmatrix}$$

$$\mathbf{X}(1) = sign\left[\mathbf{W} \cdot \mathbf{Y}(1)\right] = sign\left\{\begin{bmatrix} 4 \\ 0 \\ 0 \\ -4 \\ 4 \end{bmatrix}\right\} = \begin{bmatrix} 1 \\ 0 \\ 0 \\ -1 \\ 1 \end{bmatrix}$$

$$\mathbf{Y}(1) = sign\left[\mathbf{W}^T \cdot \mathbf{X}(1)\right] = sign\left\{\begin{bmatrix} -2 \\ 6 \\ -6 \end{bmatrix}\right\} = \begin{bmatrix} -1 \\ 1 \\ -1 \end{bmatrix} (= \mathbf{Y}_2)$$

$$\mathbf{X}(2) = sign\left[\mathbf{W} \cdot \mathbf{Y}(2)\right] = sign\left\{\begin{bmatrix} 4 \\ 4 \\ -2 \\ -6 \\ 4 \end{bmatrix}\right\} = \begin{bmatrix} 1 \\ 1 \\ -1 \\ -1 \\ 1 \end{bmatrix} (= \mathbf{X}_2)$$

After 3 steps, the BAM network converges to the 2nd pair of original samples. Let test now with $\mathbf{Y}_{new}$ equals:

$$\mathbf{Y}_{new} = \begin{bmatrix} -1 \\ 1 \\ 1 \end{bmatrix} = \mathbf{Y}(0)$$

$$\mathbf{X}(0) = sign\left[\mathbf{W} \cdot \mathbf{Y}(0)\right] = sign\left\{\begin{bmatrix} -4 \\ 4 \\ 2 \\ -2 \\ 4 \end{bmatrix}\right\} = \begin{bmatrix} -1 \\ 1 \\ 1 \\ -1 \\ 1 \end{bmatrix}$$

$$\mathbf{Y}(1) = sign\left[\mathbf{W}^T \cdot \mathbf{X}(0)\right] = sign\left\{\begin{bmatrix} -4 \\ 8 \\ 4 \end{bmatrix}\right\} = \begin{bmatrix} -1 \\ 1 \\ 1 \end{bmatrix} (= \mathbf{Y}_{new})$$

$$\mathbf{X}(1) = sign\left[\mathbf{W} \cdot \mathbf{Y}(1)\right] = \mathbf{X}(0)$$

After 2 steps, the BAM network converges to new pair, different than all of the 4 original samples, $\mathbf{X}_5 = \begin{bmatrix} -1 \\ 1 \\ 1 \\ -1 \\ 1 \end{bmatrix}; \mathbf{Y}_5 = \begin{bmatrix} -1 \\ 1 \\ 1 \end{bmatrix}$.

There is also a close relationship between the BAM and the Hopfield network. If the BAM weight matrix is square and symmetrical, then and the BAM can be reduced to the auto-associative Hopfield network. Thus, the Hopfield network can be considered as a BAM special case.

In general, the maximum number of associations to be stored in the BAM should not exceed the number of neurons in the smaller layer. But during the iterations, the BAM may not always produce the exact output vector, but only the closest association. In fact, a stable association may be only slighty related to the initial input vector.

The BAM still remains the subject of intensive research. However, despite all its current problems and limitations, the BAM promises to become one of the most useful artificial neural networks.

6.6. UNSUPERVISED LEARNING AND SELF-ORGANISING NEURAL NETWORKS

The distinct property of a neural network is the ability to learn from its environment, and to improve its performance through learning. So far we have considered *supervised learning* – learning with an external 'teacher' or a supervisor who presents a training set to the network. But another type of learning also exists: *unsupervised learning*.

In contrast to supervised learning, unsupervised or *self-organised learning* does not require the destination values in the data samples but only the input values. During the training process, the neural network uses a number of different input patterns, discovers significant features in these patterns and learns how to classify input data into appropriate categories. Unsupervised learning tends to follow the neuro-biological organization of the brain. Unsupervised learning algorithms is usually faster than supervised learning, and thus can be performed in real time systems.

Self-organizing neural networks are effective in dealing with unexpected and changing conditions. In this section, we present Hebbian and competitive learning, which are based on self-organising networks.

6.6.1. Hebbian learning

In 1949, Donald Hebb proposed the idea commonly known as Hebb's Law [Hebb49]. Hebb's law states that if neuron *i* is near enough to excite neuron *j* and repeatedly participates in its activation, the synaptic connection between these two neurons is strengthened and neuron *j* becomes more sensitive to stimuli from neuron *i*.

The Hebb's law can be translated into the form of two rules as follows [Stent73]:

- If two connected neurons are activated synchronously, then the weight of the connection between the neurons is increased.
- If two connected neurons are activated asynchronously, then the weight of the connection between the neurons is decreased.

Hebb's law provides a kind of unsupervised learning, where the learning process is performed without the feedback from the environment.

Using Hebb's law the weight w_{ij} between neuron i with the presynaptic level x_i (incoming signal) and neuron j with postsynaptic level y_j (outgoing signal) at iteration t is updated basing the following form:

$$\Delta w_{ij}^{(t)} = F\left(y_j^{(t)}, x_i^{(t)}\right) \tag{6.22}$$

As a special case, we can represent Hebb's Law as follows [Haykin99]:

$$\Delta w_{ij}^{(t)} = \alpha \cdot y_j^{(t)} \cdot x_i^{(t)} \tag{6.23}$$

where α is the learning rate parameter.

These equations (6.22) and (6.23) are referred to as the *activity product rule*. Hebbian learning implies that weights can only increase, i.e. the strength of a connection can only increase. This fact leads to the risk that when being repeatedly stimulated the weight w_{ij} can easily be saturated. To avoid this problem, we can introduce a non-linear *forgetting factor* into Hebb's law in Eq. (6.23) as follows [Kohonen89]:

$$\Delta w_{ij}^{(t)} = \alpha \cdot y_j^{(t)} \cdot x_i^{(t)} - \phi \cdot y_j^{(t)} \cdot w_{ij}^{(t)} \tag{6.24}$$

where ϕ - is the forgetting factor.

6.6.2. Competitive learning

Another popular type of unsupervised learning is competitive learning. The Kohonen network is a model proposed by Teuvo Kohonen in 1989

[Kohonen89]. The network is also known as a self-organizing map (SOM) and is a model that characterizes a group of models operating on the principle of grouping by similarity between the samples. Group input signals based on compatibility for compatibility.

The idea of clustering and self-organization baes on the facts that our brain is a very complex system. The structure of the brain is inconsistent, covering many different regions. Recent biomedical studies have shown that each region of the brain has a different structure, number of neurons and the way they are connected, and that each region is responsible for different tasks. For example, there are areas responsible for image processing, motion processing, audio processing, and so on, and these regions receive signals from different human "sensors". We may say that every characteristic human signal will be transferred into a corresponding region within the brain Therefore, when constructing the mathematical model of the Kohonen network, we have the input signal belonging to a given space. Unlike the MLP network, the Kohonen network operates on a "self-organizing" basis, meaning that the network only works with input vectors $\mathbf{x}_i$ without corresponding output patterns $\mathbf{d}_i$. In the self-organizing problem, given a set of data sets and a required number of centers M, we need to find the location of those M centers which best represent the whole input samples. Samples and centers are represented in the form of a vector with the same dimension. The input contains a set of p multidimensional vectors $\mathbf{x}_i = [x_{i1}, x_{i2}, \dots, x_{iN}] \in \mathbb{R}^N; i = 1 \to p$. On Fig. 6.10a there is an example of 2D samples denoted as 'o' marks and on the Fig. 6.10b isthe way we group them into 3 groups, each of the group is characterized by its center (*).

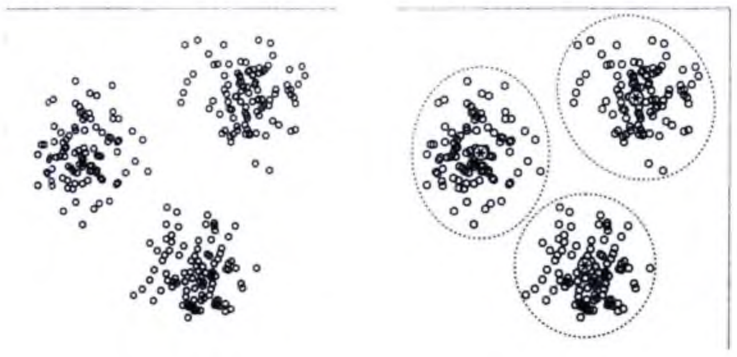


Figure 6.10. Example of a 2D distribution of samples (a) and their division into 3 groups with centers at (*)

When we have data samples expressed in forms of vectors, the degree of "similarity" between samples is usually determined by the distance between the vectors. Two vectors with small distance to each other will be considered to be more similar than the case when the distance between them is big. The groups is mainly determined on the principle that "vectors close to each other will be prioritized in the same group". The measure of the distance between the vectors is mainly based on the Euclidean formula:

$$\mathbf{x},\mathbf{c} \in \mathbb{R}^N : d(\mathbf{x},\mathbf{c}) = \|\mathbf{x}-\mathbf{c}\| = \sqrt{\sum_{i=1}^{N}(x_i - c_i)^2} \qquad (6.25)$$

When we already have M centers $\mathbf{c}_i, i = 1 \to M$ and there is a new input vector $\mathbf{x}$ then the competitive action between the centers means a winning center will be selected. The winning center is the closest one to the new input vector $\mathbf{x}$

$$\|\mathbf{x}-\mathbf{c}_{\text{win}}\| = \min_{i=1\to M}\|\mathbf{x}-\mathbf{c}_i\| \qquad (6.26)$$

The Kohonen network can be presented as on Fig. 6.10, in which the components of c_{ij} $(i = 1,\dots,M; j = 1,\dots,N)$ of M centers $\mathbf{c}_i$ are stored as the connecting weights between input x_j and center $\mathbf{c}_i$. The network on Fig. 6.11 will have output of the winning neuron equal 1, the other outputs will be 0.

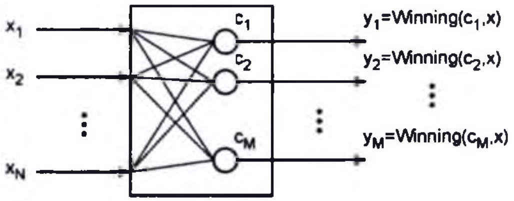
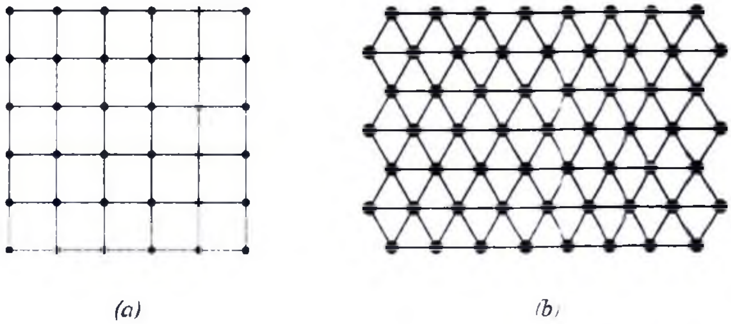


Figure 6.11. Classical structure of the Kohonen network

In addition to defining the centers, Kohonen proposed also a virtual link between these centers to form a grid of centers. Accordingly, the network will have a function that defines the link between the centers. If two centers are connected then they are called "direct neighbors", also called first-order neighbors. If two centers of $\mathbf{c}_i$ and $\mathbf{c}_j$ are not directly linked but there exists a chain of intermediate centers connecting these two centers then $\mathbf{c}_i$ and $\mathbf{c}_j$ are called indirect n-th order neighbors (with $n-1$ being the smallest number of intermediate centers between them).

To facilitate simulations, two-dimensional grid networks with rectangular, triangular, hexagonal structures are often used, as in the following Fig. 6.12.



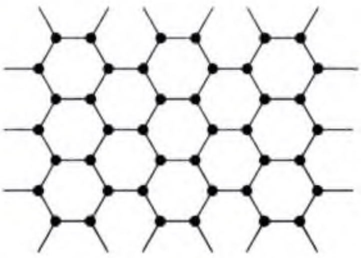

(c)

Figure 6.12. Example of topology of the centers' grid in Kohonen: a) rectangular, b) triangular, c) hexagonal

6.7. LEARNING ALGORITHM FOR KOHONEN NETWORK

As menstioned above, in the Kohonen network, we have an input data set containing *p* vector $\mathbf{x}_i \in \mathbb{R}^N$, $i = 1, 2, \ldots, p$, and a preselected number of centers *M* to be found. The training of the Kohonen network is equivalent to finding the *M* centers $\mathbf{c}_i \in \mathbb{R}^N$, $i = 1, 2, \ldots, M$, such to represent the whole data samples or the centers should be located where the data samples concentration is high. There are various algorithms for training the Kohonen network. In this section, we will present two of the most popular ones, which are the classical WTA (*Winner Takes All*) and the modern FCM (*Fuzzy Clustering Method*).

6.7.1. The WTA Algorithm

The WTA algorithm can be presented in following steps:

1. Start with the sample number $i = 1$.
2. For consercutive steps $i = 1, 2, \ldots, p$ we process with vector $\mathbf{x}_i$: find all the distances from $\mathbf{x}_i$ to the centers $\mathbf{c}_j$ to determine the winning center N_W, which is the closest one to $\mathbf{x}_i$.
3. Update the location of the winning center by shifting the center toward the actual input vector $\mathbf{x}_i$.

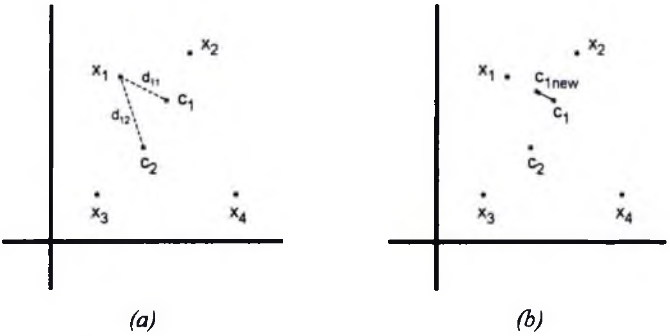


Figure 6.13. Example of 4 input vectors x_1, x_2, x_3, x_4 and 2 centers c_1, c_2. With x_1 the center c_1 wins and is shifted toward x_1 when c_2 is unchanged

By repeating the steps above for many iterations, the centers will tend to a stable state. The algorithm is called WTA because at each step, only the winning center is updated. The formula for the update is as follow:

$$\mathbf{c}_{N_W}^{(t+1)} = \mathbf{c}_{N_W}^{(t)} + \eta(t)\left[\mathbf{x}_i - \mathbf{c}_{N_W}^{(t)}\right] \tag{6.27}$$

where: t – time step, $\mathbf{c}_{N_W}^{(t)}$ – the winning center at t, $\eta(t)$ – update step at t.

The update step $\eta(t)$ is within the range $(0,1)$ (0 means no shifting at all $\mathbf{c}_{N_W}^{(t+1)} = \mathbf{c}_{N_W}^{(t)}$, 1 – full shifting $\mathbf{c}_{N_W}$ toward $\mathbf{x}_i$, i.e. $\mathbf{c}_{N_W}^{(t)} = \mathbf{x}_i$). When the centers are connected in a grid, a shift of a center will cause also the shifts of its neighbours (but will smaller steps) as demonstrated on Fig. 6.14.

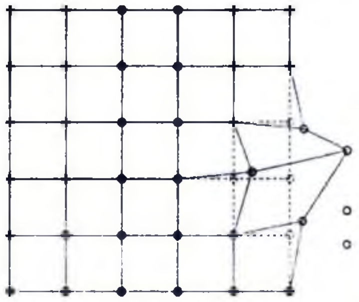


Figure 6.14. The effect of moving a node from an original squrel grid

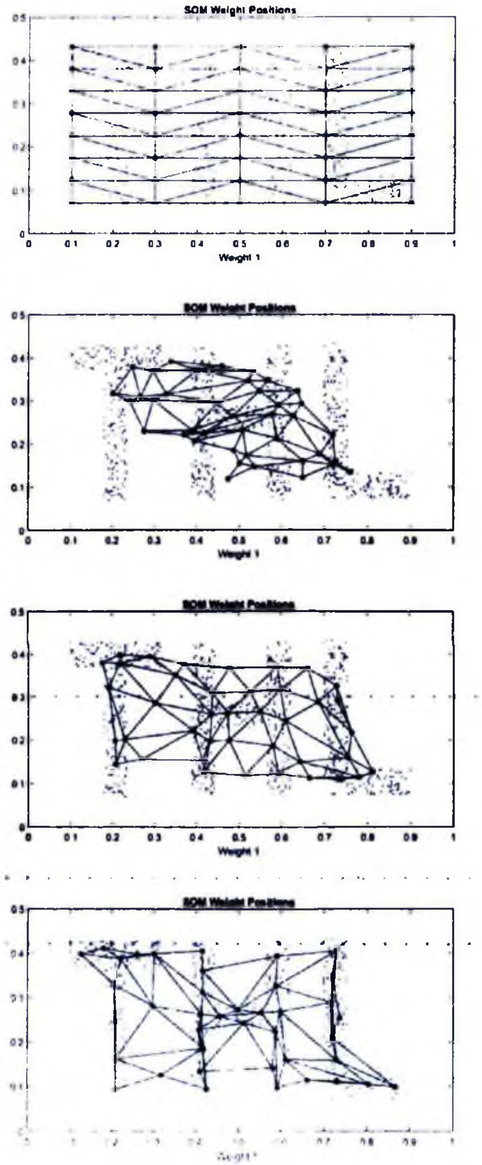


Figure 6.15. Example of a grid of 8x5 centers in Kohonen network spread among the data samples: a) Regular initiation at start, b) after 10 iterations, c) after 100 iterations, d) after 500 iterations

On Fig. 6.15 is an example of a Kohonen network of 40 centers connected to each other using the rectangle topology 8×5. It can be seen that the network is deformed to put the centers into places with high concentration of data.

6.7.2. The FCM Algorithm

From the Fig. 6.14 we can also see that the grid connecting the centers can be unhelpful since it may cause that a center, being hang between 2 other centers, is located at the point with low data concentration (or even zero data concentration). To avoid that fact, the FCM algorithm removes the grid topology completely, which means that all centers can be moved (updated) independently. In the FCM algorithm (also called the C-mean algorithm), the centers has also the indices values u_{ij} which indicate the membership function of vector $\mathbf{x}_j$ to center $\mathbf{c}_i$. In WTA in particular and in classical Kohonen network algorithm in general, each input vector $\mathbf{x}_j$ will be assigned to the closest center, which means the membership function is 1, but to other centers the membership function should be 0:

$$u_{ij}=f\left(\mathbf{x}_j,\mathbf{c}_i\right)=\begin{cases}1 & if \quad \forall k:\left\|\mathbf{x}_j-\mathbf{c}_i\right\|\le\left\|x_j-\mathbf{c}_k\right\| \\ 0 & if \quad otherwise\end{cases} \tag{6.28}$$

In FCM, the membership functions are "soften", which means the membership can take values from the whole range [0,1] and a vector $\mathbf{x}_j$ can belong to more than one center. But the membership still need to fulfill next conditions::

1. Total membership of a vector to all centers is 1:

$$\sum_{i=1}^{M}u_{ij}=1 \tag{6.29}$$

for each data point $j=1,2,\ldots,p$.

2. The membership is inversely proportional to the distance between the sample and the center. The closest center will receive the bigger membership.

The C-mean algorithm has a target function in the learning process [Dunn73, Bezdek81]:

$$E = \sum_{i=1}^{K} \sum_{j}^{n} u_{ij}^{m} \left\| \mathbf{c}_i - \mathbf{x}_j \right\|^2 \tag{6.30}$$

for $m \in [1, \infty]$. To find the minimum of this function with constrains from (6.30), the Lagrange method is used with the modified function:

$$L_E = \sum_{i=1}^{M} \sum_{j=1}^{P} u_{ij}^{m} \left\| \mathbf{c}_i - \mathbf{x}_j \right\|^2 + \sum_{j=1}^{P} \lambda_j \left(\sum_{i=1}^{M} u_{ij} - 1 \right) \tag{6.31}$$

where λ_j are the Lagrangians. The solution for (6.31) is [Dunn73, Bezdek81]:

$$\mathbf{c}_i = \frac{\sum_{j=1}^{P} u_{ij}^{m} \mathbf{x}_j}{\sum_{j=1}^{P} u_{ij}^{m}} \tag{6.32}$$

and

$$u_{ij} = \frac{1}{\sum_{k=1}^{M} \left(\dfrac{d_{ij}}{d_{kj}} \right)^{1/(m-1)}} \tag{6.33}$$

where $d_{ij} = \left\| \mathbf{c}_i - \mathbf{x}_j \right\|$ – the distance between $\mathbf{c}_i$ and the input vector $\mathbf{x}_j$. From these equations, the iterative FCM is defined as follow:

1. Initiate randomly the locations of the M centers (within the range of the input samples).
2. Calculate the matrix **U** from (6.33).
3. Find the M centers $\mathbf{c}_i$ using (6.32).
4. Calculate the target function in (6.31). If the value is less than a threshold or the change in function's value is less than a threshold then stop the learning process, otherwise go back to step 2.

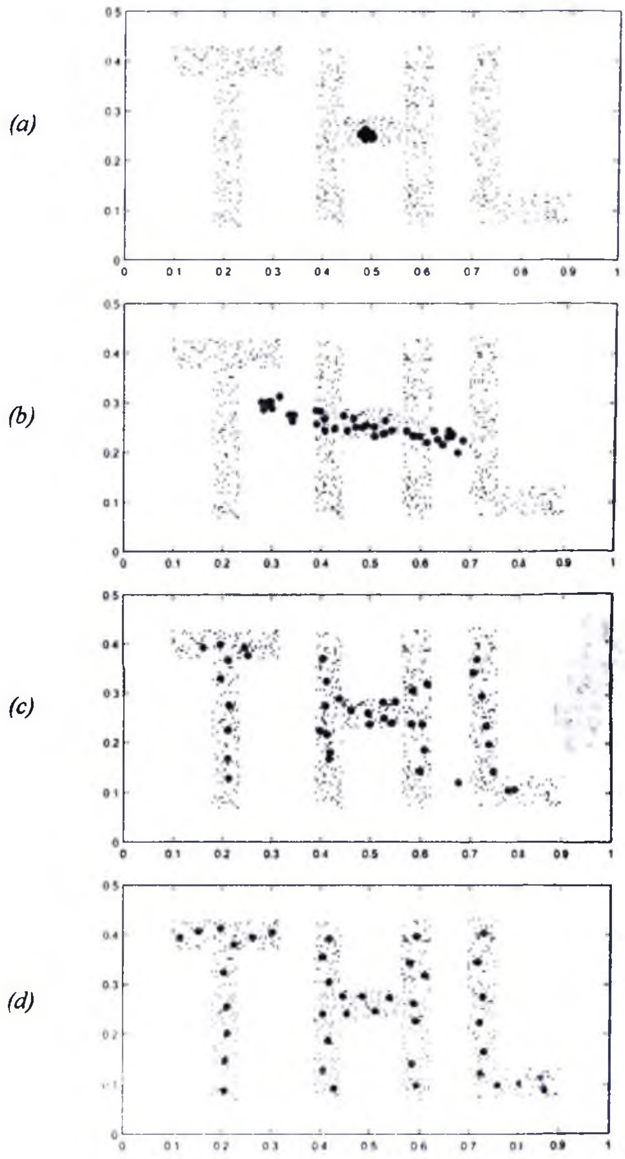


Figure 6.16. The result of FCM algorithm for the data from the Fig. 6.14: a) Initial positions, b) after 5 iterations, c) after 10 iterators, d) after 50 iterations

As it can be seen from the Fig. 6.16, when removing the grid between the centers, the centers' locations are much better distributed among the data samples, no center is outside of data area.

6.8. SUMMARY

In this chapter, we introduced the ideas of artificial neural networks and discussed the main ideas of learning process and learning algorithms. Starting from the simple perceptron model of McCulloch and Pitts, the supervised algorithm for the neuron and the network was presented with examples. The supervised formulas for recurrent networks such as the Hopfield and the BAM were discussed. Afier that, the Kohonen network and its unsupervised learning algorithm were discussed with example.

Chapter 7: Evolutionary algorithms

Artificial Intelligence is defined as a computer science field that is capable of assimilating, analyzing, and exploiting existing facts and knowledge as information useful for acquiring new facts. Evolutionary algorithms that mimic the processes of subsequent generations in the natural environment, and the associated calculations are artificial intelligence [Michalewicz92]. The concept of evolutionary algorithms includes methodologies inspired by the Darwinian principle of natural selection used to solve difficult issues. This chapter briefly presents the four basic types of evolutionary algorithms: genetic algorithms, genetic programming, evolutionary strategies and evolutionary programming, and detail discusses the main type, which is the genetic algorithms.

7.1. INTRODUCTION TO EVOLUTIONARY ALGORITHMS

What is evolution? Perhaps the best way to describe the words of Charles Darwin: "*How busy is looking at densely overgrown coasline covered with plants of different species, with singing birds in thickets, insect-borne insects, worm-bitten worms, and look at all these strangely constructed forms so different and in such a complex way dependent on each other*".

To think that they were created by laws still acting around us. These laws, in the most common sense, are: growth and reproduction, the inheritance during the reproduction, the variability under the direct and indrect influence of external living conditions, and the use and/or non-use of certain organs for the existence struggles and the consequences of the natural selecion, which in turn leads to divergence of characteristics and the extinction of less improved forms. So the fight in nature, with hunger and death directly resuts from the most advanced phenomenon we can grasp, namely the formation of higher forms of animals. In the study entitled "*The Origin of Species*", published in 1859, Charles Darwin introduced the theory of evolution by natual selection. This theory is a source for imitation in solving problems in various fields of science

It should be emphasized that evolutionary algorithms are based on mechanisms of evolution (i.e. selection, crossing, mutation, and reproduction).

Evolutionary algorithms are common name for various techniques developed over several decades. The most popular of these are: genetic algorithms introduced by John Holland [Holland75] and popularized by David Goldberg [Goldberg03], genetic programming developed by John Koza [Koza92]. The other two techniques are evolutionary programming [Fogel66] and the evolutionary strategies [Schwefel95].

7.1.1. Genetic algorithms

Genetic algorithms were initiated by J. Holland in 1960. Traditionally, when searching for an optimal object, various instances of the object type are generated (forming a *population* of candidates) and each of them are characterized by a genetic code called the *chromosome*. In the simplest case, the chromosomes in genetic algorithms are fixed length binary strings (chromosomes can also be encoded by integer or real strings) and the size of the population is constant. Chromosomes are evaluated in each generation. Parameters that need to be determined are: population size, probability of mutation, and condition of termination of algorithm. An adaptation function is defined in advance to evaluate how fit (adapted) a candidate is to the environment.

From the primary population, new and better populations are created, with three basic operations: selection, crossing and mutation.

Selection is the process by which higher adaptive chromosomes are more likely to introduce descendants to the next generation.

Crossing is an operation performed on randomly selected parental chromosomes, called parents. For the parents we generates new instances (children) by randomly taking segments of chromosomes from parents to form the new chromosomes. The crossing operation is usually performed for a number of parent pairs.

Mutation is an operation performed randomly for each gene separately. It will modify the chromosome according to a predefined the probability of mutation to generate new chromosomes.

After selection, crossing and mutation, a new generation is evaluated, i.e. it is calculated to adapt individual chromosomes from this generation to create a parental pool for the next generation. The quality of the results obtained by genetic algorithms depends on the size of the population, the time spent searching for the solution, the selection of the selection method, the crossing and mutation operators used, and the probability of the operations being performed.

7.1.2. Genetic programming

Genetic programming uses the principles of genetics and Darwinian natural selection to create computer programs [Koza92, Oduguwa05]. Genetic programming is largely similar to genetic algorithms. The basic difference between genetic programming and genetic algorithms lies n the representation of the solution. While the genetic algorithm creates a sequence of numbers that represents the solution of the problem, in genetic programning individuals are tree-like programs (such as programs written in LISP), and genetic operators are applied to the branches and nodes in those trees [Kinnear94, Oduguwa05].

7.1.3. Evolutionary programming

Evolutionary programming focuses mainly on optimization problems with continuous parameters. In evolutionary programming, each parent in the population generates a descendant by mutation. The probablity of a mutation is generally evenly distributed. After evaluating the descendants, a variant of stochastic tournament selection chooses some of the best individuals from the parent and descendant team. The best individual is always stored to ensure that if the optimum is achieved, it can not be lost. Evolutionar programming is an algorithm that uses only selection and mutation mechanisms without crossing) [Oduguwa05].

7.1.4. Evolutionary strategies

There are two main evolutionary strategies (μ, β) ES and (μ + β) ES, where μ parents produce β descendants using the crossing and mutation operator.

In strategy (μ, β) ES the best β descendants are survived and replaced the parents. As a result, parents are not present in the next population. On the contrary, the strategy (μ + β) ES allows the survival of both descendants and parents. The (μ + β) ES uses an elite strategy (the best individual is always copied to the next population). In both strategies, both crossing and mutation are performed [Oduguwa05].

In the following parts we will concentrate more on the genetic algorithms, since all the main ideas and techniques are included in the design of them.

7.2. THE DETAILED STRUCTURE OF EVOLUTIONARY ALGORITHMS

As mentioned above, the evolutionary algorithm is a term that approximates an optimization algorithm that uses selection, reproduction and mutation mechanisms inspired by the biological process of evolution.

According to [Michalewicz92] we have "The idea behind genetic algorithms is to do the same thing that nature does. Let's take the example of rabbits. At any time we have a rabbit population. Some are faster and smarter than others. Faster and smarter rabbits have a better chance of fleeing from the fox. As a result, they are more experienced to do what rabbits do best, namely new rabbits. Of course, some of the slower and stupid rabbits will survive too, because they are lucky. This saved the rabbit population offspring. It is a good mixture of genetic material. Some slow rabbits are crossing with fast, some clever with stupid, and so on. And nature also throws a wild card from time to time, introducing a mutation into the genetic material of rabbits. The resulting rabbits will be (on average) faster and smarter than those from the initial population, because faster and clever parents have fled the foxes."

In the biological process of evolution of a given population of individuals, the pressure of the environment causes the natural selection. Only the best adapted individual have the chance to survive and launch new and better populations. In the evolutionary algorithm, the problem we have solved is the role of the environment, in which the population of individuals lives. Each individual represents a potential (possible) solution to the problem [Pena00]. As in the biological process, the evolutionary algorithm gradually produces better and better solutions. It follows that the evolutionary algorithm can be used to solve optimization problems.

The optimization process involves searching through the potential problem areas to find the best solution. An example of an optimization problem might be scheduling the work of the aircraft crews. Having a scheduled flight schedule for certain types of aircrafts, one of the tasks is to design weekly crew schedules. Every day the crew must be allocated working time consisted with related flights and met the restrictions such as maximum total air time, minimum rest time between flights,... Then, taking into account the one-day schedules, they must meet further restrictions, such as night rest or return to the home port. The aim is to minimize the amount of money paid to the crew, which is a function of time spent in the air, length of working time, guaranteed minimum hours of flight,... [Wolsey98].

In real applications, the space for potential solutions is usually so large that it is not possible in a sufficiently short (real) time to check all possible potential solutions to choose the best one. Therefore, in such cases, it is reasonable to use probabilistic techniques that use random selection as a tool to direct the search process. Evolutionary algorithms are just one such technique that has been successfully used in engineering practice.

The optimization problem can be briefly written down as finding the value of the variable x contained in a given set X, at which a predefined function $f(x)$ of the variable x takes the best value. The function $f()$, referred to as the target function or quality function, measures the goal to be achieved. For example, the goal may be to minimize costs or maximize profit. In practice, there are often many divergent goals. The set X is defined by a collection of problem constraints, such as resource availability and order size. Most classical optimization algorithms use a deterministic procedure to approach the optimal solution. This procedure usually begins from a certain solution (the initial point), and then, based on local information, determines the direction of the search to seek the optimal solution. Then it is a one-way search for the best solution. In order to avoid the local minima problems, the above procedure is repeated a certain number of times. Classical algorithms differ in the main way in determining the direction of search. These algorithms include direct and gradient methods. Direct methods for determining the direction of search use only the values of the objective function and the constraints. Gradient methods use the concept of the first and second derivative of the function of the goal or constraint [Oduguwa05]. The basic difficulties in using classical methods are as follows:

- The convergence of the algorithm to the optimal solution depends on the choice of the initial point,
- Most algorithms tend to get stuck in a suboptimal/local solution,
- An algorithm that can be effective in solving a given optimization problem may not be as effective in solving another problem,
- Algorithms are not effective for use with discrete search space problems,
- Algorithms usually are not effectively used on parallel machines.

Evolutionary algorithms deal with most of the difficulties encountered with the use of classical algorithms. Evolutionary algorithms have a unique ability to adapt easily and can be used to solve complex nonlinear and multidimensional engineering problems. Their quality does not depend on the problem, it does not matter its structure or variability.

The basic features of evolutionary algorithms that distinguish them from other methods are as follows [Goldberg03]:

- They do not directly process the job parameters, but their encoded form,
- They search, not from a single point, but from a certain population,
- Use only the target function, not its derivatives or other auxiliary information,
- Use probabilistic rather than deterministic selection rules.

To be able to speak freely about evolutionary algorithms, one must acquire knowledge of the terminology derived from genetics. These concepts include, among others, chromosome, which is a sequence of ordered elementary units called genes. The chromosome is a *genotype*, and its unencoded counterpart is a *phenotype*. The space in which the algorithm works is called the population and consists of individuals whose characteristics are determined by the genotype record. To be consistent with the theory of evolution, an eement is still needed to determine the degree of adaptation of the individuals, which will allow them to survive better adapted and thereby eliminate the weaker units. The next generations of individuals arise through selection, crossing and mutation.

The task of a genetic algorithm is to model the process by imitating the natural course of evolution. In each evolutionary step, called the generation, the chromosomes are decoded and evaluated according to a predetermined quality criterion called adaptation (the function of adaptation can be, for example, the objective function), and then selection is made to eliminate the worst evaluated. Individuals with high adaptability are subject to mutation and recombination by crossing operator. Selection alone does not introduce any new individual into the population, i.e., no new points are found in the search space, but such points are introduced by crossing and mutation. By crossing the evolutionary process can move towards promising areas in the search space Mutation prevents convergence to local optimum.

As a result of the operation of the crossing and mutation operator, new solutions are created, from which the next generation population is built. For example, a certain number of generations may be requred to complete the algorithm, or a satisfactory adjustment level may be reached.

An example of flow chart for these steps is shown on Fig. 7.1.

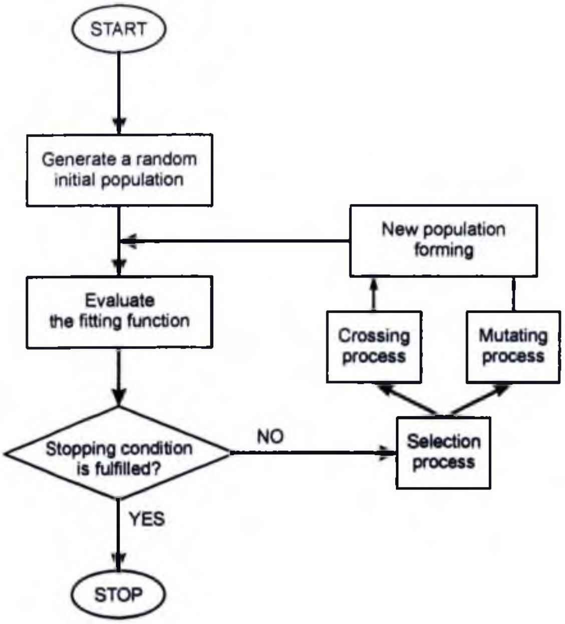

START
Generate a random initial population
New population forming
Evaluate the fitting function
Crossing process
Mutating process
Stopping condition is fulfilled?
NO
Selection process
YES
STOP

Figure 7.1. General scheme of a Genetic Algorithm

7.2.1. Encoding the individual candidates

When creating an evolutionary algorithm, from the very beginning there may be problems - for example, with the way the coders of individuals. The most popular is the binary code. The only thing to keep in mind is to find a sufficient number of bits to encode the information. It is always an important issue to choose the right representation of individuals. This is important because it turns out that some problems, which are difficult to solve with the representation data, can be much easier when choosing other representations. Classic genetic algorithms use representation in the form of single-layer binary strings. For parametric optimization with continuous domains, the representation can be in the form of real numbers. Then genetic operators operate directly in the search space. In general, choosing the right representation for the problem for which the algorithm is designed remains (so far as well as the construction of the appropriate genetic operators) a kind of art. Choosing good representation is just as difficult as finding the right algorithm to solve a problem [Whitley97]. Moreover, it has also been shown [Radcliffe95] that all representations are equivalent if they are considered in terms of all possible problems. So we can think that the best way to determine if the correct choice is to be an experiment is to make sure that the selected representation is better than the other representations.

The most commonly used coding methods for individuals are: binary strings and real numbers.

7.2.2. The adaptation function

In evolutionary algorithms, the adaptation function is a fundamental element that links the evolutionary algorithm to the problem solved. Its main purpose is to evaluate the individuals in the population and to distinguish them in such a way that better individuals have a better chance of moving to a new population and thus passing on their genetic material to the next generation. At this point you also need to distinguish the cost function that occurs in many issues from the adaptive function (the objective function), which is generally created using the cost function, but may also include other factors. As an example, we can list constrained optimization tasks where appropriate unacceptable (non-compliant) solutions should be evaluated.

Then the adaptation function is based on the use of the penalty function [Michalewicz96] and includes additional factors related to the non-compliance subjected to the constraint problem. Another example may be multi-criterion optimization, where the adaptive function can be created as a weighted sum of cost functions for particular criteria, or by giving individuals the appropriate rank associated with their "non-neglect degree" (i.e. Pareto optimization) [Goldberg89]. For optimization tasks without limitation, the cost function equals the adaptation function. To determine the value of the adaptation function for each individual (binary form), the corresponding genotype of the individual should be determined from the corresponding phenotype.

Choosing an adaptive function is not an obvious thing because many tasks are more natural in terms of minimizing a certain cost function than maximizing it. To convert a maximization task to a minimization task, we can just take the opposite value of the cost function.

7.2.3. Selection

In order to create a new, better population, genetic operations are performed on individuals selected from the "parental" population. Selection, also referred to as reproduction, is the procedure of selection from the population of certain individuals to create a new generation in the evolutionary algorithm. The selected individuals have usually the highest value of adaptation to the parent population. The probability of choosing a particular individual depends on its adaptability. The greater the adaptation the person has the greater chance that he will be chosen for the new generation. Reproduction in evolutionary algorithms is closely related to the two most important factors: preservation of population diversity and so-called selective pressure. These factors are, in a sense, dependent on each other because increasing the selective pressure reduces the diversity of the population (and vice versa). Too much selective focus (i.e. concentrating the search on the best individuals) leads to premature convergence [Booker87], which is undesirable in evolutionary algorithms because the algorithm can get stuck in the local extremity. On the other hand, too little selective pressure causes the search to become almost purely random.

The purpose of the selection mechanisms is to maintain a balance between these factors [Michalewicz92]. This selection process can be made using several methods [Goldberg89] including:

- the roulette method,
- the stochastic with replacement method,
- the tournament method,
- the ranking method,
- the elitist method,...

a) The roulette selection method

There are many selection methods. The oldest (best known) method is the roulette method, also called proportional reproduction selection. In this method, the likelihood of drawing an individual is directly proportional to the value of its adjustment function. Each individual responds to a roulette wheel sector with a size equal to the value of adaptation of a given individual divided by the sum of the adaptation of the whole population. Whenever roulette wheels are played, individuals are selected for the new population, and better coded individuals add more descendants to the next generation.

In the roulette method, the intensity of selective pressure decreases with successive generations, as more and more of the offspring have similar approximation values, which makes it difficult to approximate such an algorithm. This is a straightforward method and produces fairly satisfactory results. However, it has disadvantages such as:

- Too early elimination from the population of individuals with a very low value of adaptability. This may lead to too early stop of the algorithm.
- The ability to use the roulette method only for one task class, i.e. only to maximize (or only to minimize).

b) The stochastic sampling with replacement method

This method uses a probability distribution $p(i)$ of the integer index *i* of the sample in the population ($i = 1, 2, \ldots, N$) and its cumulative distribution $cp(i) = \sum_{j \le i} p(j)$. From a population of N samples, if we want to select (with repetition) M samples, the stochastic sampling with replacement method uses the following steps [Brindle80]:

- For new sample number j ($j = 1, 2, \ldots, M$), randomly generate a value uniformly from the range [0, 1]: $\alpha \in [0, 1]$.
- Find the first index i with the cumulative distribution bigger than α: $i = \min_{k=1,2,\ldots,N} cp(k) > \alpha$.

- Take the old sample number i to be the new sample number j.

The recommended probability function for this method is the relative fitness, which is:

$$p(i)=\frac{f(s_i)}{\sum_{j=1}^{N} f(s_j)} \tag{7.1}$$

where $f(s_i)$ is the fitness of the sample number i. It's easy to prove that, for non-negative fitness function, the above function satisfies the conditions for a probability function:

- $\forall i=1,2,\ldots,N: 0 \le p(i) \le 1$.
- $\sum_{i=1}^{N} p(i)=1$.

c) The tournament selection method

Tournament selection consists of dividing the population into k-element subgroups (k is the size of the tournament - usually 2 or 3) and selecting the best individual from each subgroup. This can be done by random selection or deterministic selection. Then the selection is made with a probability equal to 1. The tournament method is suitable for both maximization and minimization problems.

d) The ranking selection method

In the ranking selection population individuals are set in turn according to the function of adaptation – i.e. from the best to the worst adapted. Each number is assigned a number called rank and denotes its position in the list. The number of copies of a given individual introduced into a new population is determined on the basis of a predefined function depending on the rank (for example a monotonic function).

e) The elitist selection method

The selection methods described above, in particular regarding roulette selection, may be enhanced by mechanisms that emphasize the behavior of the best individuals.

In the elitist strategy, emphasis is placed on the behavior of the most adapted individuals in successive iterations. In the classical genetic algorithm, there is a situation where the best individuals do not get into the new population. The elite model is supposed to prevent this [Michalewicz92] because the best individuals are introduced into the next generation without the selection procedure. This is the case when the population with the best adaptation value loses by the population, then the inserted chromosome replaces the worst individual in the population.

Beside the above selection methods, in [Michalewicz92] many other modifications are described to enrich the choice for us when dealing with a specific task. Selection methods can be divided into static and dynamic ones. Static selection means that the probability of selection is constant for all generations, while dynamic selection does not. Another subdivision of selection methods is distinguished by the extinguishing and preserving methods. In preservation selection there are non-zero probabilities of selection for each individual. In contrast, in the extinguishing selection some probabilities can be zero. Extinguishing selection can be divided into left- and right-extinguishing. In the left-extinguishing, the best individuals are not allowed to reproduce to avoid too early convergence. In the right- extinguishing selection, there is no such rule.

Some selection methods can be called exclusive, which means that parents can reproduce only in one generation-so the life span of an individual is limited to one generation. You can distinguish between generational selection and on-the-fly selection. In the generational selection, the parents' set is constant until a new population is created and only then the exchange is performed. In the on-the-fly selection, the descendants replace the parent as soon as they are created [Michalewicz92]

Another technique that can be used to improve the selection quality is the target function scaling. Target function scaling is used to prevent premature convergence of the genetic algorithm, where the best but not yet optimal chromosomes dominate the population. Also in addition, in the final phase of the algorithm, where the population contains individuals quite similar to each other, the average adaptation value is little different from the maximum. Scaling of the adjustment function can then prevent the average and the best individuals from receiving nearly the same number of descendants in the next generation, which is undesirable.

Scaling is based on a proper transformation of the adaptation function. The basic scaling methods are linear scaling, sigma-truncation scaling and power law scaling [Goldberg89].

Linear scaling is the transformation of the function f to the form f' by linear transformation with the formula:

$$f' = a \cdot f + b \tag{7.2}$$

The coefficients a and b should: keep the average adjustment value unchanged and set a maximum value of scaled adaptation at the level of the specified average multiplicity. These two conditions together ensure that an average individual of the population generates on average one child, and the best of all – number of children is proportional to the fitting function.

Linear scaling works well, except when the function f' takes negative values. In that case, we can use the sigma truncation scaling. Transformation of the function f to the form f' is done according to the following relation:

$$f' = c \cdot \left(f - (f_{mean} - \sigma)\right) \tag{7.3}$$

where: f_{mean} is the mean value of the fit function in the population, c is a small natural number (usually 1 to 5) [Michalewicz92], σ – is the standard deviation of the population. The negative value f' is assumed to be equal to zero.

Power law scaling raises the function of adaptation f to a fixed power k:

$$f' = f^k \tag{7.4}$$

The value of *k* is a number close to 1 and depends on the problem in question. It may need to be changed in successive generations - in order to "stretch" or "pull" the range as desired.

7.2.4. Crossing and mutation

The selection procedure in evolutionary algorithms is the first step in creating a new generation. However, in order for newcomers to be not only copies of individuals from the previous generation, it is necessary to use genetic operators to modify the individuals selected by the selection process. In general, the operators used in evolutionary algorithms can be divided into two groups.

The first one is unary operators, that is, acting on a single individual. These operators are referred to as mutations. According to biological literature [WebMutation], mutations are changes in the normal DNA sequence of an organism caused by the action of chemical and physical factors or DNA replication errors. Mutations occur in two forms: point mutations that involve a single change and larger mutations for longer DNA sequences. Major mutations include: deletions (consisting in loss of DNA sequences), insertions (occurring as a result of insertion of additional bases from another part of the chromosome), rearrangements (i.e. reverse mutation in which a fragment of DNA sequence is cut and then incorporated into the same place but opposite orientation). Most mutations in the natural world have been transposed into evolutionary algorithms.

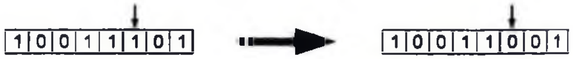


Figure 7.2. An example of a single-point mutation

The most common mutation operators include: uniform mutation, border mutation, unequal mutation [Michalewicz92]. These operators are performed on

the gene population with probability $P_{mut} \in [0, 1]$, whose value is one of the parameters of the evolutionary algorithm. The mutation operation involves drawing a random number from the range [0; 1). Where for the j-th gene in the i-th person drawn the number smaller than P_{Mut} then this gene is mutated. The simplest mutation scheme (with binary representation of individuals) is to convert the value of the selected gene from 0 to 1 or vice versa, and is shown in Fig. 7.2.

When representing an individual in the form of a sequence of real numbers, the procedure is analogous except that the new value is drawn from the given range of values.

The second group of genetic operators used in evolutionary algorithms are multi-arguments operators, called recombinant operators (crosses). From a genetic point of view, recombination is the process that leads to new genetic combinations in a DNA molecule (chromosome) by cutting and reuniting existing DNA molecules (chromosomes) to form cross-over crosses. In evolutionary algorithms, the crossing operation is performed on a population with a probability $P_{cross} \in [0;\ 1]$ and consists in drawing for each individual the random number in the interval $[0;\ 1)$. In the case of an i-th random number smaller than P_{cross}, the i-th individual is selected for crossing. In the evolutionary algorithms, the simplest model of crossing is the so called one-point crossing, whose graphical scheme of operation (for binary representation of individuals) is shown in Fig. 7.3.

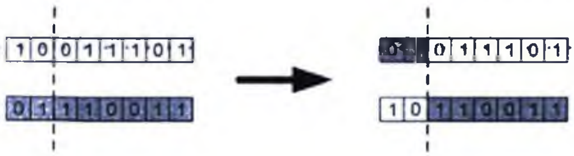


Figure 7.3. Single-point crossing between 2 chromosomes

In general, this operator creates two children, two of which are parental, and is defined for single-layer chromosomes as follows. Let the individuals $x_i = (v_1, \ldots, v_m)$; and $x_j = (y_1, \ldots, y_m)$ be the parents (selected individuals) to cross. Then, after drawing the cut point k from the interval $[1, m-1]$ (this also implies that the specimens must be of the same length) and crossing the following children:

$$x_i' = (v_1, \ldots, v_k, y_{k+1}, \ldots, y_m); \; x_j' = (y_1, \ldots, y_k, v_{k+1}, \ldots, v_m); \qquad (7.5)$$

The natural extension of this method is the 2-point crossing method, which is shown on Fig. 7.4.

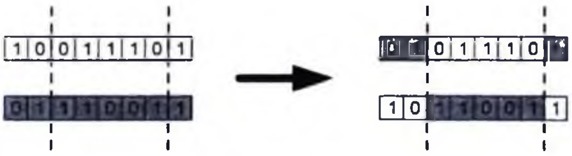


Figure 7.4. Double-point crossing between 2 chromosomes

Using the mutation and crossing operators can lead to the formation of degenerates, i.e. descendants not belonging to original domain **D**. This is especially not the case for restricted tasks.

Various types of repair algorithms that depend on the problem are used to deal with this. In addition, many other recombinant variants have been developed in the field of evolutionary algorithms to ensure efficient exchange of information between two chromosomes. Requesting to meet this condition causes that crossing operators are often tailored to the problem solved, reducing their versatility.

A typical example of recurrence-dependent recombination operators is the PMX (*Partially Mapped Crossover*), CX (*Cycle Crossover Order*), and OX (*Order Crossover*) [Michalewicz92] solutions.

The importance of mutation and recombinant operators is closely linked to the type of evolutionary algorithm. In the first versions of evolutionary strategies, the search was based solely on the selection-mutation scheme [Back91]. Recombinant actors, on the other hand, play a leading role in genetic algorithms and genetic programming.

7.2.5. Stopping condition

After genetic operations, the next generation of individuals will be evaluated and the stop condition will be checked.

Determining the condition of stopping the algorithm causes many difficulties, so special criteria have been developed. The most commonly used criteria are the criterion of satisfactory level of adaptation function and minimum rate of improvement. In the first case, the stopping of the algorithm takes place when the value of the adaptation function set by the designer is reached, while the second algorithm stops when the best value of the solution is not improved for a sufficiently large number of generations.

One should mention here the basic theorem in the theory of genetic algorithms, namely the theorem on schemes. It says that narrow, low-order patterns and well adapted in successive generations tend to grow exponentially. It is therefore necessary to provide a definition of the schema here. The scheme is nothing but a set of chromosomes in which the values of individual genes are the same as the values of the genes of similarity pattern on particular positions. This is shown in Fig. 7.5.

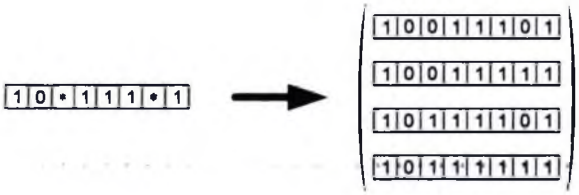


Figure 7.5. The schema and 4 possible combinations of chromosomes

The rank of the schema is the number of zeros and one in the schema, i.e. the number of genes with a value different from the '*' character. For the schema in Fig. 7.5, the rank is 6.

In turn, the span of the schema is the difference between the right-most and the left-most positions of the 0 or 1 values. The span of the schema in Fig.7.5 is $8 - 1 = 7$.

From the theorem on diagrams comes the so-called the hypothesis of bricks. It says that genetic algorithms work, based on the sequencing of genes, called bricks. It is important that the genes in the chromosomes are arranged in such a way that the ones located close together are mutually dependent. In this way, the bricks of the gene sequences that are bound to each other form a new population in which new individuals, through the links between gene groups, carry more valuable information.

7.3. APPLICATIONS OF GENETIC ALGORITHMS

Genetic algorithms can be used to solve various difficult classical problems such as the problem of a salesman or the transport problem.

In case of the salesman problem, which belongs to the so-called *difficult NP problems*, that is, with the increase in the number of cities that he would have to overcome, exponentially grows the time needed to find a solution. This can be solved by generating all the permutations of all cities and to sum the distance between cities for each permutation. Then compare the results and choose the shortest one.

On the surface it is very simple and for a small number of cities you can quickly settle this kind of problem. What do you do when 20 or 40 cities appear on the route of a salesman? The calculation time increases considerably. The use of a genetic algorithm in which each route is a single individual, using crossing (route interchanges), mutations, and selection and adaptation functions makes it possible to obtain the shortest route relatively quickly [Potvin96].

The transport problem, in turn, is the delivery of a variety of goods or of one kind from several receiving points to several receiving points. This is done in such a way that ordered quantities of goods reach, for example, from warehouses to stores, including stocks of goods in warehouses and the distance of stores from stores. In this case also transport costs should be minimized. This has been described in more detail in [Gen99], where research results also show that genetic algorithms generate good solutions.

Genetic algorithms can also be used for the problems encountered in the real world, for example, in automation, mechanics, economics,... especially for the ones characterized by a very large number of discrete or continuous variables, the complexity of the search space (many limitations and goals that may be contradictory). It is well known that the use of evolutionary algorithms in flexible production systems, particularly in relation to operational control. The example case of using genetic algorithms in structural mechanics is described in [Pieczara04]. It presents the use of genetic algorithms for solving direct problems, optimizing the cooling tower shell and identifying in mechanical systems. There are also attempts to apply a genetic algorithm to the design of hydraulic systems. It is also possible to use genetic algorithms to schedule tasks and allocate resources in networks and computer systems, as well as in production systems. Genetic algorithms can also be used for problems related to the economy. Practical examples of such use are provided in [Maschek05].

These problems can change in time (be dynamic), which entails the need to quickly get good solutions. An important area of application of evolutionary algorithms is the scheduling and planning of industrial processes. Scheduling issues where resources need to be allocated to a task collection are usually difficult to solve due to the presence of multiple constraints and complex product structures. The mathematical programming techniques used here usually allow for solutions only for small problems. In most genetic algorithms for scheduling problems, the whole schedule is encoded by the chromosome (the chromosome stores the coded schedule and thus determines the order of execution of the individual operations), which requires the use of appropriately designed genetic operators to produce the desired chromosomes in successive populations.

Another example of industrial use is the use of genetic algorithms to control the fermentation of beer [Carrillo01]. The genetic algorithm is used here to adjust the temperature profile of the mixture over a set period of time.

Evolutionary algorithms have also found quite widespread use in medicine. Most medical decisions can be formulated as searching in a certain space. For example, a doctor is looking for the best treatment in the space of all possible ways [Yu97]. The search cases in medicine are usually very large and complex. The decisions are based on clinical tests that provide huge amounts of data. Based on this data, one final decision must be made. Evolutionary algorithms are used in medicine to perform tasks that can be divided into three groups [Pena00]:

1. Data mining: is the process of finding patterns, trends, and regularities by examining large amounts of data [Foyyad96] mainly for diagnostics and forecasting. In medical data mining, evolutionary algorithms are usually used to find the values set by the designer so that the searched data is interpreted optimally.
2. Signal imaging and processing: Many medical data is expressed by images or other signals. Evolutionary algorithms are used here to improve the performance of signal processing algorithms (i.e. filters) by finding their optimal parameters. They can also be used to directly derive useful information from the data provided.
3. Planning and scheduling: Evolutionary algorithms are particularly well suited for solving scheduling and scheduling problems. An example may be the problem of patient scheduling, undergoing various medical procedures and the need for consultation of various specialists, in order to optimize patient waiting time as well as device utilization.

7.4. EXAMPLES

7.4.1. Nonlinear function maximum point search

The given (nonlinear) function f may have one or more variables. We need to specify the values of variables for which the function f achieves extreme, i.e. the minimum or maximum value. The maximization task can be easily converted to a minimization task and vice versa. Optimization is the determination of the optimal (i.e. the best, the best) solution regarding some chosen criteria for a problem, using mathematical (numerical) methods.

We assumes that R is the set of all solutions to the problem in question. The point x_{opt} in the space R is a global maximum if:

$$\forall x \in R : f\left(x_{opt}\right) \geq f(x) \tag{7.6}$$

The more frequently problem (easier to solve) is to find x_{local} for which:

$$\forall x \in \left(x_{local} - \varepsilon, x_{local} + \varepsilon\right) : f\left(x_{local}\right) \geq f(x) \tag{7.7}$$

This problem is defined by the local maximum. Classical optimization methods are usually local maxima. In many practical applications there are difficulties in using traditional optimization methods. Most have a local reach. Their mode of operation depends on the existence of the derivatives and they are not sufficiently resistant to discontinuities, extensive multimodality or interference in the searched space [Michalewicz92].

In this example, we consider the following nonlinear function as shown on Fig. 7.6 in the domain $[-10,10]\times[-10,10]$:

$$z(x,y) = 0.02 \cdot x - 0.05 \cdot y + \frac{\sin(x)\cdot\sin(y)}{x \cdot y} \tag{7.8}$$

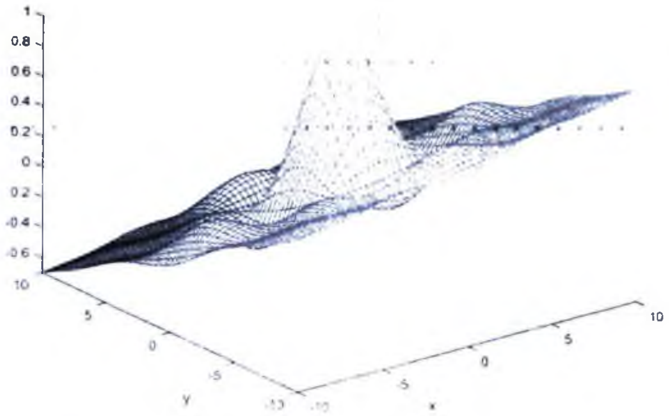


Figure 7.6. The shape of the nonlinear function in the selected domain

This is an example of a nonlinear function with many local minima and maxima, which is more difficult to find the global extrema of the function.

With the genetic algorithm approach, we use the pair of the real input variables x and y (x_i, y_i) as chromosomes. We starts from the initial population of 100 points (i.e. 100 chromosomes). This population is obtained randomly using an uniform distribution over the given domain. For each chromosome, we determine its adaptation by calculating the corresponding target function at the point encoded in that chromosome. The roulette selection method with 2 elite individuals is chosen.

For the crossing operator, we use the random linear operator as:

- Let parents are (x_1, y_1) and (x_2, y_2),
- Randomly generate a step coefficient $\alpha \in [-0.25;\ 1.25]$,
- Generate the children:

$$(x_{c1}, y_{c1}) = \alpha \cdot (x_1, y_1) + (1-\alpha) \cdot (x_2, y_2)$$
$$(x_{c2}, y_{c2}) = (1-\alpha) \cdot (x_1, y_1) + \alpha \cdot (x_2, y_2)$$

- Bound the children to the domain if needed.

For the mutation operator, we use also random linear operator as:

- Let parent is (x_1, y_1),
- Randomly generate step coefficients $\alpha_x, \alpha_y \in [-0.25;\ 0.25]$,
- Generate the child: $(x_{c1}, y_{c1}) = (x_1 \cdot (1+\alpha_x), y_1 \cdot (1+\alpha_y))$
- Bound the child to the domain if needed.

With the above configurations and methods, we achieved the following results:

- Best estimated maximum point : $(x_{opt}, y_{opt}) = (0.0538;\ -0.012)$
- Value of the function at the estimated optimal point: $f(x_{opt}, y_{opt}) = 1.0042$

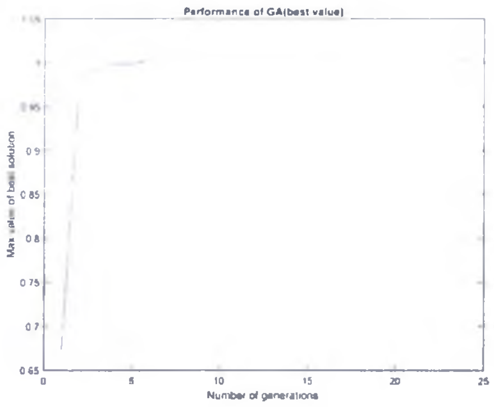


Figure 7.7. The change of maximum value found along the generations

In the Fig. 7.7 we have the change of the maximum value found along the generations. At the random begin, the maximum value was only ~0.67, after 5-10 generations it was corrected to value sub-optimal already.

In the Fig. 7.8 the best found point is presented on top of the original function to show the very high quality of the results.

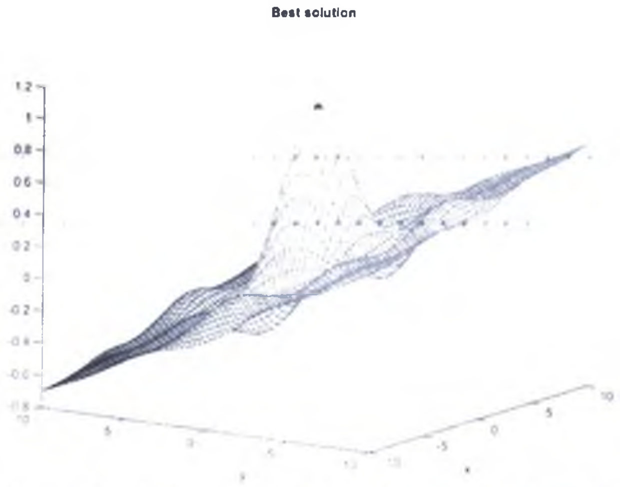


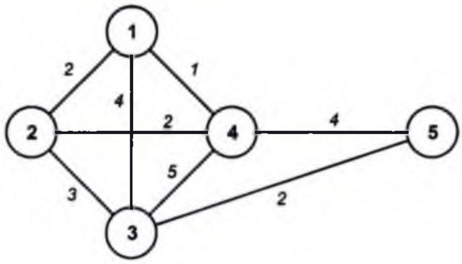
Figure 7.8. The optimum point found by the genetic algorithm

7.4.2. Travelling Salesman Problem

The next example in this chapter is a classical problem so called the Travelling Salesman Problem (TSP). In this problem there are N cities that the salesman has to visit. The target is to visit each city exactly once and get back to the starting point. The cost of traveling between each pair of cities is known. Find the best route that each city is visited exactly once and the total cost of travel is minimal.

The input map can be defined using a full graph $G=(V,E)$, where $|V|=N-$ number of cities, each vertex represents a city, and each edge connecting a pair of vertices represents the connection between the corresponding cities. Each edge has a weight equal is the traveling cost of the connection. The TSP requires a closed path passing all the vertices exactly once (except the starting vertex), i.e. the problem of the salesman can be formulated exactly as finding a Hamiltonian cycle of minimum length for a given graph.

An example of graph and the corresponding weights matrix **W** is presented in Fig. 7.9. Please note that the main diagonal of **W** contains infinity since a vertex is not connected to itself.



$$\mathbf{W}=\begin{bmatrix} - & 2 & 4 & 1 & - \\ 2 & - & 3 & 2 & - \\ 4 & 3 & - & 5 & 2 \\ 1 & 2 & 5 & - & 4 \\ - & - & 2 & 4 & - \end{bmatrix}$$

Figure 7.9. Example of a graph (undirected) and its weights

For this example, the optimal solution is the path 1-2-3-5-4-1. In total, the possible number of paths to be considered is $\frac{(N-1)!}{2}$, it means the TSP is a pessimistic exponential computational complexity.

We can define the fitting function based on the path length, and chromosomes of length $l = N$ let be the permutations of vertices. However, the standard crossing and the mutation does not work since their result is generally not a Hamilton path. Therefore, it is necessary to design crossing and mutation operators operating on permutations.

For example, we define and use the so called inversion mutation, where we randomly choose two positions in the chromosome and reverse the order of vertices between them:

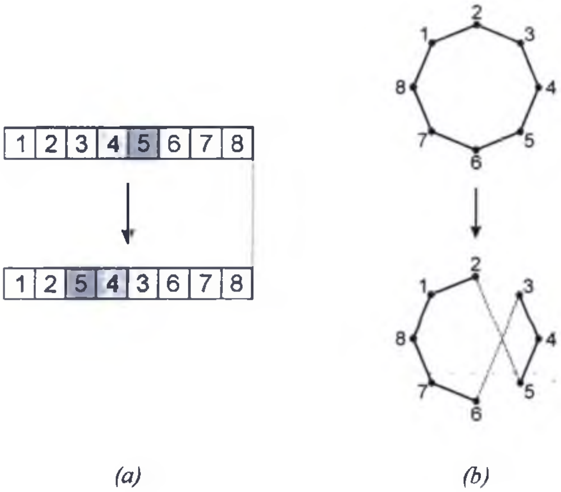


Figure 7.10. The crossing operator for traveling paths

Inversion in permutation is equivalent to changing only two edges in a cycle (we assume a symmetric problem). This is the smallest change that can be made by a mutation operator.

For the crossing operator, we can use the PMX (*Partially Matched Crossover*) [Goldberg85], which is as follow:

- Select two crossings at random, for example we have two chromosomes as in 7.11a. Copy a (randomly) selected segment (for example 7.11b) from parent P_1

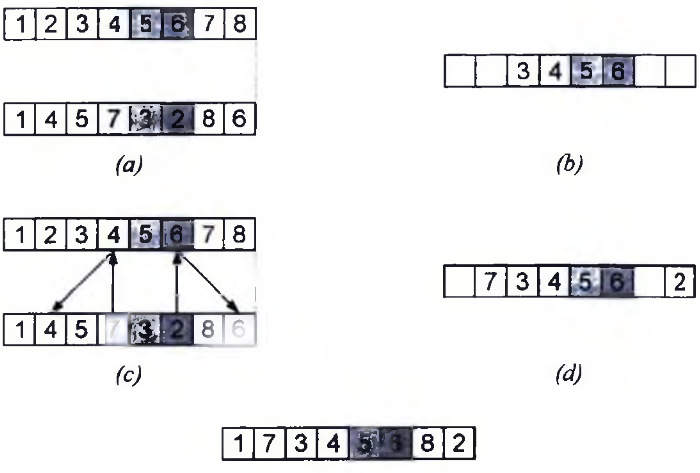


Figure 7.11. The mutation operator to create the 1st child for the two paths 1-2-3-4-5-6-7-8 and 1-4-5-7-3-2-8-6

- Looking at the same segment in P_2, find those cells that were not copied. (We need to put them outside the copied segment). For the example in 7.11 it's the cells containing '7' and '2' in P_2. For each of these elements *i* determine what element *j* was copied from P_1 to its place (in place of '7' from P_2, P_1 has 4; in place of 2 from P_2, P_1 has 6).

  So we have pairs (i, j): (7, 4) and (2, 6). For each pair, we try to place *i* in the position occupied by *j* in the parent P_2. For (7, 4), the location for '4' in P_2 is number 2, tracking back to the child chromosome (in Fig. 7.11b), the 2nd location is still free, so we will place the '7' there. Analogically, the cell '2' will go to the location of cell '6' in P2, i.e. cell number 8.

  After doing so for all cells from the copied segment, all the remaining cells (here they are cell '1' and cell '8') are copied from P_2.

  The second child is generated in analogical way by replacing the parents P_2 and P_1 and is shown on Fig. 7.12.

By using this method, from the two chromosomes 1-2-3-4-5-6-7-8 and 1-4-5-7-3-2-8-6 we have 2 new child chromosomes 1-7-3-4-5-6-8-2 and 1-6-5-7-3-2-4-8.

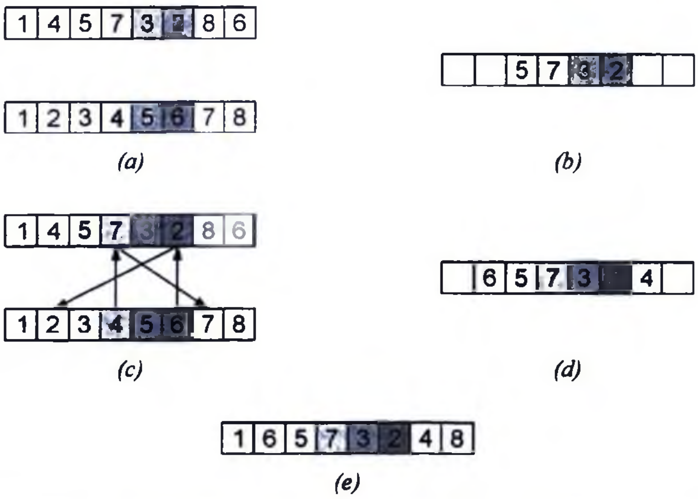


Figure 7.12. The mutation operator to create the 2nd child for the two paths 1-2-3-4-5-6-7-8 and 1-4-5-7-3-2-8-6

Next is a numerical results example of the problem. In Fig. 7.13, we have randomly generated locations of 20 cities. The traveling cost is assumed to be equal the euclidesian distance between the cities.

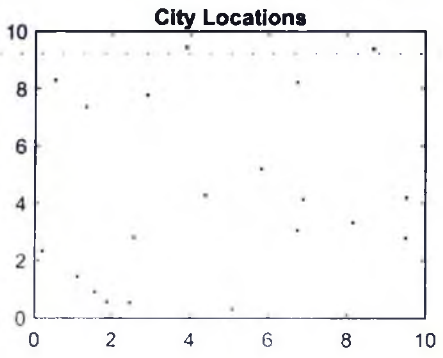


Figure 7.13. The locations of 20 cities for the TSP (randomly generated)

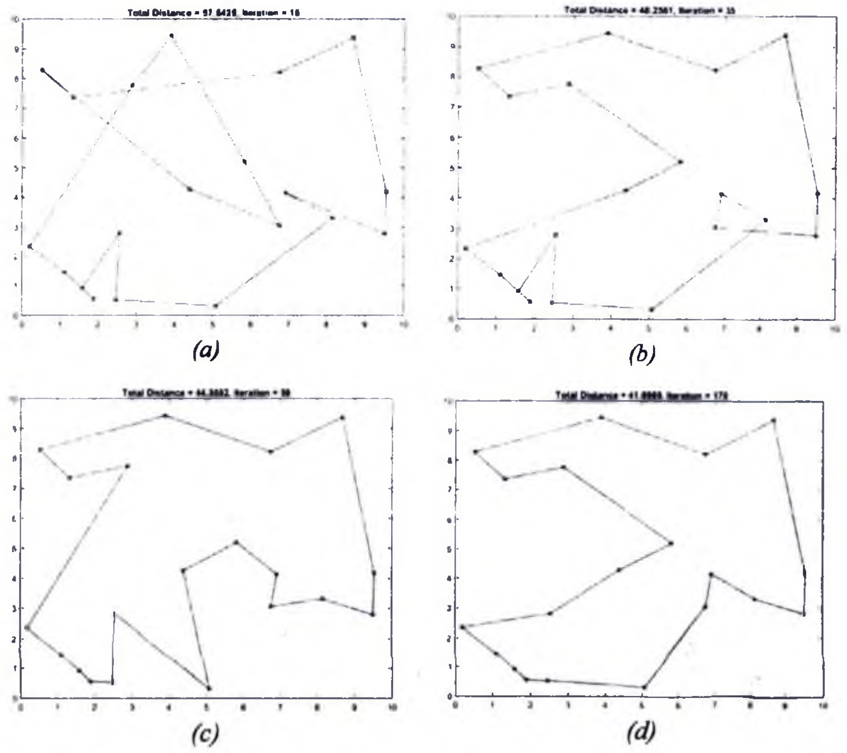


Figure 7.14. Intermediate results along the generations of evolution: a) after 15, b) after 35, c) after 59, d) after 170 iterations

In the genetic algorithm for TSP, we use the population size equal 100, number of generations is 200. In Fig. 7.14 are some results of the best candidates found in different generations.

As it can be seen, at 15th generation, the best path found is still bad (others in the population are even worse), the traveling cost is 57.54. At 35th generation, the path looks much better, with the cost reduced to 48.26. And finally, at generation number 170, the best path has the cost only 41.70. And this is also the final best candidate, it means from generation 170th to generation 200th, no better candidate could be found. The plot of how the cost was reduced along the generation is shown on Fig. 7.15.

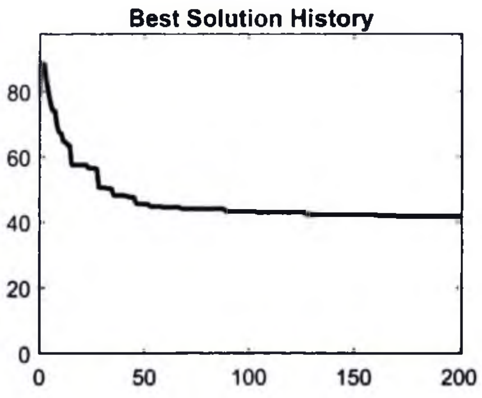


Figure 7.15. The change in best travel cost found along the generations of evolution

7.5. CONCLUSION

Evolutionary algorithms are now successfully used in solving many real problems in different areas. The possibilities of using evolutionary algorithms, including genetic algorithms, are very broad. It consists of, among others, conceptual simplicity of these algorithms. The characteristic feature of these algorithms is that they seek the solution from a specific population, not from one point, as is the case with traditional methods, and also because they use the objective function rather than the derivative function [Goldberg03].

Genetic algorithms are used everywhere where the use of common analytical and enumeration methods is impractical, for example due to a long calculation time or too difficult to meet assumptions. So the genetic algorithm is a global optimization method easy to implement in a wide range of problems. It seems that in the future the importance of techniques based on evolutionary algorithms will grow as they allow them to overcome many of the difficulties that arise with the use of classical methods. Therefore, it is expected that applications based on evolutionary principles will become more and more popular.

Chapter 8: DEEP LEARNING

8.1. INTRODUCTION

Analysts are changing to the big data concept, which became popular around 2010, and required much higher performance of specific tasks: the machine learning process. One of the goals of this process is to create a computerized method of modeling our world, our knowledge in a good enough way. Our knowledge database is huge and so diverse that it is impossible to process all of this information manually. That is why many researchers have been working on automatic algorithms to capture a large part of this database.

Machine learning solutions, which are the basis of artificial intelligence, use automation to recognize and learn interdependencies, especially in predictive and prescriptive analytics.

Significant progress has been made in understanding and improving learning algorithms, but there are still big challenges for machine learning in particular and for artificial intelligence tasks in general. For example, we do not have yet algorithms that can understand objects and describe them in natural language. We do not have algorithms that can interact with most people in daily works,... Those situations are similar for other AI tasks. Machine learning is accompanied by several other concepts. The most important are: neural networks, deep learning and cognitive computing. These technologies will not only allow automatic and precise predictive analysis of the gargantuan amount of data, but in some cases, such as stock market trading, to even create events. According to various research, in 2020, the market value of smart applications will exceed 40 billion dollars.

8.1.1. Machine learning - strong support for predictive analytics

Machine learning is a technique in the area of computer science and statistical modeling, which allows a computer application based on independent analysis - without having to program it - to predict the results or make decisions.

Machine learning, which is the basis of artificial intelligence (AI), is closely linked to data analytics and data mining programming. Both machine learning and data mining use mathematical algorithms to crawl and search patterns. Machine learning uses algorithms to detect patterns in data sets and adjusts program behavior accordingly. Predictive analytics services based on big data and cloud help developers and data researchers to use machine learning in a new way.

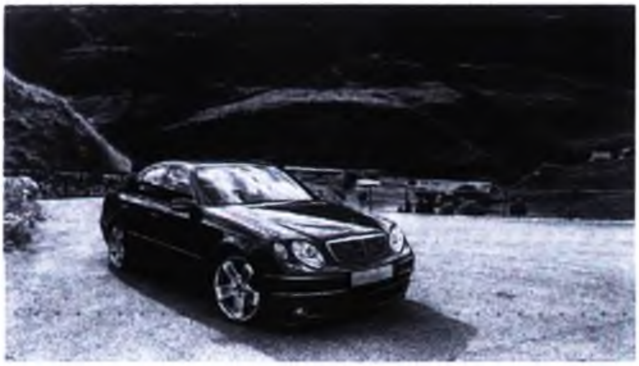


Figure 8.1. A picture containing "a car under sunlight"

We would like to transform the original input image into higher levels of representation or linguistic concepts such as "daytime", "running car", "parked car",... If the machine intercepts factors that explain the facts we observe, we can say that the machine *understands* these facts and even their variations.

Unfortunately, we generally do not have an analytical knowledge formula for most of our understanding of the world and its variants. High-level abstraction, such as CAR, can appear in a very large number of possible images that are very different from each other. The CAR category can be seen as high-level abstraction with respect to the image space. What we call an abstraction can be a category (such as the CAR category) or a feature that can be discrete or continuous (i.e. a video showing a car moving at 60 kilometers per hour). Many lower level concepts are needed to construct a detector. Lower levels are more directly related to specific perceptions, such as “whether it has a wheel”, “does it have steering wheels” ... while higher levels are “more abstract”.

In addition to the difficulties related to the relevant indirect abstractions, the number of visual and semantic categories that we would like the “intelligent” machine to recognize is quite large. Full (successful) network learning involves automatic detection of such abstractions, from the lowest levels to the highest level concepts. We would like the learning algorithms achieve this discovery with minimum of human effort, i.e. without having to manually define all the necessary abstractions or to provide a huge collection of manually prepared examples. Then surely these networks would help to convey much human knowledge to computer-interpreted form.

8.1.2. Deep learning - algorithms that use neural networks

Deep learning is one of the types of artificial intelligence (AI), a sub-category of machine learning - a technique of neural networks, whose main task is to improve the performance of hard problems such as voice recognition, computer vision and natural language processing. Simplifying, deep learning can be treated as a way to automate predictive analysis. This technology is rapidly becoming one of the most sought after areas in computer science. Deep learning applications include all kinds of big data analytics applications, especially those focused on natural language processing (NLP), foreign language translation, medical diagnostics, stock market transactions, network security, object recognition, machine translation of speech, or scientific themes. Over the past few years, deep learning has helped to advance in many diverse areas such as that for a long time have been for researchers with hard nut to crack.

While traditional machine learning algorithms are linear, deep learning algorithms are arranged hierarchically according to increasing complexity and abstraction. Data must go through several processing layers, so it was decided to use the term "deep" learning. Most modern learning algorithms correspond to limited architecture (due to the Komogorov theorem). People often describe their knowledge in a hierarchical manner, with multiple levels of abstraction. The brain also seems to process information through the stages of transformation and representation. This is particularly evident in the primitive visual system [Serre07], with a sequence of steps: edge detection, primary shapes detection, and progressively more complex visual shapes recognition.

As mentioned in Chapter 6, neural networks are the generic name for software systems and data structures that are closely related to operations performed by the human brain. Neural networks typically consist of a large number of parallel processors that have their own knowledge and access to local memory. Typically, a neural network is "trained," or is fed with large amounts of data and rules about data relationships. Then the program can instruct the network how to behave in response to external stimuli or can initiate a process on its own. Although the neural networks have successfully learned many problems, they are still considered as "single step" processing tools. For complicated tasks (starting with computer vision problems, where we try to recreate our capability in analyzing objects from their images/video), the classical neural networks could not learn to process a task in a multi-step manner. Deep networks consist of multiple levels of nonlinear units such as neural networks with multiple hidden layers or complex recursive formulas. Optimizing deep networks is a difficult task, but recent learning algorithms such as Deep Belief Networks have been proposed that are effective in selected problems. Later, we will introduce the ideas and the principles for deep network learning algorithms.

To fully understand what deep learning is, it is important to first distinguish them from other important scientific disciplines in the field of AI research.

One of the revolutionary discoveries in the field of AI related science was machine learning, in which the computer learns from the supervised experience. This process usually involves a human operator who assists the machine in learning by introducing thousands of training examples while manually correcting any errors that occur.

Although machine learning has become the dominating area of AI research, systems still have some limitations. First, learning process is very time consuming, for example a programmer needs to be very specific about what features he should look for when recognizing a particular object in order to implement them into the programs. This is a laborious process, and the success of the program depends on the programmer's precise definition of the set of features for a particular object. Secondly, in this way we still can not fully verify the machine intelligence level, because it is dependent on human ingenuity, which abstracts the learning process of the computer.

Unlike machine learning, deep learning is mostly done without supervision. Among other things, this entails the creation of large-scale neural networks that allow the computer to learn and think independently without the need for direct human involvement.

Deep learning does not really look like a computer program, where regular computer code is written in very restrictive and logical steps. In the case of deep learning we are dealing with something else, we do not meet many of the instructions that tell us: “If one is the truth, do the same again”. Deep learning is not based on linear logic, but on the theory of human brain work. The program creates entangled layers of interconnected network nodes. The system learns by reorganizing connections between nodes following each new experience. Deep learning methods are designed to find the functions of the higher levels of the hierarchy created by the composition of the features of the lower functions. Automated learning process at multiple levels of abstraction enables the system to learn complex functions of input data mapped to output data. This is especially important in cases where people do not know how to define exactly the relationships and can only give examples. The ability to automatically acquire advanced features will become more and more important as the types of applications expand and the data collected is increased. The depth of deep networks refers to the number of levels of nonlinear operations in the network structure.

Deep teaching has the potential to be a software base capable of producing excitement or events in the text (even if not explicitly identified), recognizing objects in photographs and producing advanced predictions about possible future human behavior. The advantage of deep learning is that the program itself builds a set of qualities for use. Not only does it get faster, but usually more accurately.

Basic concepts are:

- Machine learning - Automated analytical systems that learn over time and gain more data. They often use more complex algorithms (predictive and normative).
- Deep learning - artificial intelligence (AI) algorithms, used in self-guiding cars, for image recognition and natural language processing. They typically use neural networks and other complex algorithms. Memory, reasoning and attention are their key attributes.
- Cognitive systems - usually self-learning systems, which use complex sets of algorithms to mimic the processes that take place in the human brain.

Key benefits of deep learning include

- The capability to develop hierarchical models across various types of data, including text, images, and audio and video.
- Capability to learn from a very large database of examples: computational time for teaching should be acceptable.
- The capability to learn complex, very diverse functions, i.e. with many variants that are much larger than the number of training examples.
- The capability to learn with little human input.
- Capability to learn from labeled and unmarked data, where not all examples contain complete and correct target values.
- The capability to deduce meaningful patterns and knowledge from huge volumes of unstructured data.
- Massive parallelism to allow for multiple processing cores - and even computers to work efficiently with algorithms to maximize performance.
- Capability to use synergies present in many tasks. These synergies exist because all AI tasks provide different views on the same basic reality.

An example is the Jane AI from IBM to promote a healthy lifestyle

IBM announced that they have been working on the development of the Jane AI application based on the Watson cognitive system. The application to be a personal trainer is designed to analyze data on physical activity and encourage regular exercise. By using the Watson Conversation programming interface, Jane AI uses the native speaker's natural language, recognizes conversation topics, and intends to speak words. The application is based on information about the number of steps, distance traveled a day, calories burned, or location needed to determine weather conditions. By analyzing entries published on social networking sites, you can even get to know your mood, your motivations, or your personal goals, and encourages you to become active. In communication, he is based on a wealth of psychology knowledge, including theories of attitude change, influence, and personality analysis. The data collected during the project is intended to help scientists analyze patterns of behavior related to health. The test phase of the project will be launched after at least 100 attendees and will last about 10 weeks.

In 2011 Google launched the Google Brain project, which created a neural network trained with deep learning algorithms, widely recognized as capable of recognizing advanced concepts.

Facebook has also created the AI Research Unit using deep learning expertise to support the development of technology solutions that will better identify faces and objects in 350 million photos and videos each day on Facebook.

8.1.3. Challenging the training of deep neural networks

Having motivated the need for deep networks, we consider the difficulties in training them. Experimental evidence suggests that deep network training is more difficult than training classical neural networks [Bengio07, Erhan09]. When compared with typical training algorithms for classical neural networks, the performance on deep networks, especially the generalization ability is quite poor. It was then suggested that, on the basis of classical learning algorithms, (normally the supervised version) multilayered neural networks would be converged into a local minimum, whose position is strongly depended on the initial position. And the deeper the network and the more number of nonlinear parameters the harder to get good generalization.

It has been discovered that much better results can be obtained during the initial training of each layer, starting with the first layer. The experiments started with the RBM model [Hinton06] and then similar results are achieved when extending to the application of auto-coders for each layer training [Bengio07, Ranzato07, Vincent08]. Most of methods use the same idea: start with the training of the first layer (for example using RBM or some auto-coder), then use the output of the first layer (new raw input representation) as input for next layer, and train that layer with a similar algorithm like for the previous layers. Once all layers are initialized, the entire neural network can be trained using a supervised training algorithm as usual. The advantage of unsupervised pre-training compared with random initialization is shown in [Bengio07, Erhan09, Larochelle09].

As described in [Erhan09], unsupervised initial training is equivalent to a regulator: improper initial training causes a limitation to a bad search region in the space of parameters, i.e. the learning process will converge to a bad local minimum. On the other hand, other experiments [Bengio07, Larochelle09] suggest that, for multiple layer networks, poor tuning of the lower layers (the ones closer to the output) may result in poorer results without initial training. If there are enough hidden units in the first hidden layer, the training error may be very low, even if the lower layers are not properly trained, but this may cause a poorer generalization. When the training error is low and the test error is high, we call that the network is overfitted.

On the other hand, for larger training data samples sets, a better initialization of the lower hidden layers can significantly decrease both the training and the generalization errors. According to actual research, better generalization will be achieved when all layers are properly tuned.

8.2. THE BEST FRAMEWORKS FOR MACHINE LEARNING AND DEEP LEARNING

Which framework to choose to implement machine learning or deep learning? It depends on the complexity of the learning process, the volume and format of the data, but also the programming preferences. To the popular frameworks belong: TensorFlow, Microsoft Cognitive Toolkit, IBM Watson, Spark MLlib, Scikit-learn, MXNet, and Caffe.

The opportunities available to those interested in deep learning and neural networks have never been so rich.

Frameworks for implementing machine learning and deep learning differ from one another. First and foremost, machine learning frameworks include a variety of methods for learning algorithms for classification, regression, grouping, anomaly detection, and data mining. They may also include methods based on neural networks. Deep Neural Network Frameworks or Deep Neural Network (DNN) frameworks cover a variety of neural network typologies with multiple hidden layers. These layers represent a multi-step pattern recognition process. The more layers within a network, the more complex features can be captured for grouping and classification purposes.

Caffe, CNTK, DeepLearning4j, Keras, MXNet and TensorFlow are the backbones for implementing deep learning. Scikit-learn and Spark MLlib are machine learning frameworks. Theano combines both categories.

The more neurons and layers that need to be trained, and the more training data, the longer it takes. When Google Brain trained its translation models for the new version of the Google Translate tool in 2016, it conducted training sessions on many different GPUs. Each of these sessions lasted about one week.

Individual frameworks have at least one characteristic feature. The advantage of the Caffe package is the use of convoluted, deep neural networks for image recognition. The Microsoft Cognitive Toolkit has a separate evaluation library for deploying predictive models that run on ASP.Net sites. In turn, the MXNet framework has excellent scalability in the context of training configurations involving multiple GPUs and multiple machines. Scikit-learn offers a wide range of effective teaching methods. In addition, it is easy to learn and use. The Spark MLlib framework has high scalability in machine learning. TensorFlow offers a unique diagnostic tool for network graphs called TensorBoard.

Each of the described frameworks presents a slightly different approach to neural network descriptors, with two main concepts in mind: the use of a graphical description file or the creation of custom descriptions by code execution.

The more details of each of these frameworks are discussed in following parts.

8.2.1. IBM's Watson

The Watson solution of IBM Watson is based on the supercomputer created by IBM to answer questions in natural language. Its name is honored by the founder of this company, Thomas J. Watson. The supercomputer is being developed as part of the DeepQA research project and uses a combination of algorithms for natural language processing, information retrieval, knowledge representation, automatic inference, and machine learning. It has 2880 cores, 15 TB of memory and does not use an Internet connection. As part of his presentation, Watson appeared in the Jeopardy game show in a three-day competition February 14-16, 2011. His opponents were: Brad Rutter, who so far won the most money in this game show, and Ken Jennings, who was the longest in his unconquerable champion. Watson won this game, with a score of $77,147. Ken Jennings earned $24,000 and Brad Rutter won $21,600. Main features proposed by Watson are:

- Dialog in natural language: Enter a data dialogue and discover new dependencies and information. All you need is a web browser on your computer or a mobile app for your tablet.
- Automated Predictive Analysis: Automatically detect factors that can affect your business performance.

- Instant analysis: Just one click to fully understand the information derived from the data. With automatic visualizations in the cockpit you can say more and better.
- Intelligent Data Mining: Discover the most interesting, custom word-processing paradigms with cognitive processing capabilities that point to your starting points and direct you to the answers.
- Simplified analysis: Instantly find out what information is hidden in your data, thanks to the automation solutions that work for you. This allows you to use the newly acquired knowledge to improve your business.
- Advanced analysis for everyone: Start using data without carrying out their complex optimization and no preparation. Advanced analysis available to anyone eliminates complex and time consuming tasks, justifying your decisions with reliable data.
- Self-service cockpits: Share the information you find in the cockpit or infographics, which you can easily develop based on visualizations that are documented during exploration.

8.2.2. TensorFlow

TensorFlow, a portable library of artificial intelligence and neural networks from Google, is characterized by good performance and scalability, despite being a bit slow to learn. TensorFlow has a number of different models and algorithms that are a heavy burden for deep learning.

The framework is characterized by excellent performance when working on GPU devices (for training) or TPUs from Google (in the case of forecasting on a production scale). In addition, the framework provides excellent Python support, good documentation, and an efficient TensorBoard tool for displaying and analyzing data flow diagrams describing the computation performed.

The main language needed to use TensorFlow is Python, although the framework also offers limited C ++ support. Tutorials provided with the TensorFlow framework include applications for digital handwriting recognition, image recognition, word sequencing, recursive neural networks, sequential models for machine translation, natural language processing, and partial differential equations simulations.

8.2.3. Microsoft Cognitive Toolkit

The Microsoft Cognitive Toolkit is a fast and easy-to-use framework for implementing deep learning, but has a fairly limited scope of use compared to TensorFlow. It offers a variety of models and algorithms, excellent Python support and Jupyter Notebook applications, an interesting declarative BrainScript neural network configuration language, and an automated deployment for Windows and Ubuntu Linux operating systems.

On the other hand, until version Beta 1, the framework did not support MacOS although many improvements have been made to CNTK 2 including a new memory compression mode to reduce memory consumption on GPUs and new Nuget installation packages.

8.2.4. Caffe

The Caffe Framework for Deep Learning, which initially served as a powerful framework for image classification, appears to be stuck on the RC3 version due to the continually emerging bugs and the leaving the project by its initiators. However the framework still has good quality convolutional neural networks for image recognition and good support for Nvidia's CUDA GPU, as well as a simple network description format. On the other hand, models of this framework often require large amounts of GPU memory (more than 1 GB) to run, its documentation still contains bugs, it is difficult to get support, and the installation process is problematic,, especially with regard to the Python library. It's pity for Caffe because of the stagnation in its development.

8.2.5. MXNet

MXNet is a portable and scalable deep learning library selected by Amazon as the DNN framework. It combines the symbolic declaration of neural network geometry with the imperative programming of sensor operations. MXNet scales to many GPU's on many different hosts with almost linear scaling at 85% and boasts excellent speed, programmability and portability. It supports Python, R, Scala, Julia, and C ++ languages on different levels, allowing you to combine programming in symbolic languages with imperative programming.

8.2.6. Scikit-learn

The Sickit-learn Framework written in Python offers a wide selection of effective machine learning algorithms, but beyond deep learning. For Python users Scikit-learn is certainly the best of all simple machine learning libraries.

Scikit-learn is a powerful (and extensively tested) machine learning library written in Python with a wide assortment of well-established algorithms and integrated graphics. The program is relatively easy to install, learn and use and has good examples and good quality tutorials.

On the other hand, the Scikit-learn framework does not include deep implementation or reinforcement learning, does not offer graphical models or predictive sequences, nor can it be applied to languages other than Python. With the exception of minor adventures with neural networks, there are no real problems in terms of learning speed. It uses the Cython compiler (Python-to-C) for functions that need to be executed instantaneously, such as internal loops.

8.2.7. Spark MLlib

Spark MLlib is an open source learning machine for Spark, which provides the most commonly used machine learning algorithms such as classification algorithms, regression, grouping and co-filtration (but not DNN) with tools for capturing characteristics, transformations, dimensional reduction. It offers also a selection of tools for constructing, evaluating, and tuning machine learning pipelines. Spark MLlib has functions for saving and loading algorithms, models and pipelines, data handling and linear algebra calculations and statistical calculations.

The Spark MLlib framework was written in Scala and uses the Breeze linear algebra package. Breeze relies on the netlib-java library for optimized numerical algorithms, although this means optimized CPU usage in open source distributions. Databricks offers customized Spark clusters that are compatible with GPUs capable of potentially up to 10 times the speed needed to train complex machine learning models using Big Data.

8.2.8. Best in class

Which framework to implement machine learning or deep learning to choose for a specific task depends on the complexity of the machine learning process, the volume and format of the training data, the computing resources, and the programming language and programming skills preferences. It may also depend on whether you prefer to define models using code or configuration files.

Before starting your own modeling practice, however, it is important to check that any pre-trained machine learning services from Google, HPE, or Microsoft Azure clouds will efficiently process data, regardless of their audio, text, or graphics format. If they do not work for the selected data, simple available statistical methods should be tested before trying out basic machine learning training and, ultimately, if nothing else works, deep learning training. In this case, the principle is to maximize the simplification of the analysis, but no greater than justified.

Choosing frameworks for implementing deep learning from Microsoft Cognitive Toolkit, MXNet, and TensorFlow is a much more difficult decision. The selection to the three programs does not make it easy, because all these frameworks are a good way to offer similar capabilities.

The Microsoft Cognitive Toolkit now has APIs for Python and C++ as well as BrainScript configuration language. If you prefer to use configuration files from network topology programming, the Microsoft Cognitive Toolkit may be a good solution. On the other hand, this framework does not seem to be as mature as TensorFlow and does not support the MacOS operating system.

MXNet provides support for Python, R, Scala, Julia, and C++ languages, but the best APIs are for Python only. The MXNet framework has demonstrated good scalability (85% linearity) across many different GPUs on many different hosts.

TensorFlow is probably the most mature of the three frameworks mentioned. It is a good choice if someone writes code in Python and learning the language is not a challenge. TensorFlow offers basic backbone elements that can be used and provide precise control, although they also require writing a neural network code. There are three simplified APIs that work with TensorFlow in this regard: *tf.contrib.learn*, *TF-Slim* and *Keras*. The last feature in favor of TensorFlow is TensorBoard, which is useful for visualizing and analyzing data flow diagrams.

8.2.9. The future of deep learning

Deep learning is a very promising field of learning that can make self-managing cars and robotic valets a reality. However, the prospects for development are still limited, but what can be created now by the use of deep learning technology, several years ago was unthinkable and continues to grow at an alarming rate. It is the ability to analyze huge data sets and apply deep learning in adaptive computer systems to experience, rather than relying on human programmers, to lead to breakthroughs. It can take the form of discovering new drugs, developing new materials, or even creating robots with more awareness of the surrounding world. Perhaps this anticipated breakthrough is an explanation of the fact that Google has recently been in the frenzy of buying, and at the head of the shopping list are robotic companies. Google bought out eight robotics companies in a matter of months.

8.3. SELECTED NETWORKS FOR DEEP LEARNING

Inspired by the complex brain structure, researchers have been training neural networks including the deep networks [Bengio07]. But for the deep network, before the work of Hinton in [Hinton06] introducing the Deep Belief Networks (DBNs), the training usually got the bad results due to network overlearning [Haykin99]. Hinton and others proposed a unsupervised learning algorithm for each layer being a Boltzmann machine (RBM). Similar algorithms have been proposed after that, but based on the same idea.

Because deep networks can be seen as a series of processing steps, the obvious question becomes "What data representation should be used as the result of each stage?". How the information is transferred between these stages? Many research on deep architecture have focused on these intermediate representations: learning representations of data using RBMs [Hinton06], auto-encoders [Bengio07, Ranzato07, Vincent08]. These algorithms can be seen as learning to transform one representation (starting from the previous stage) into another while conserving (or even enhancing) the variation factors presented in the data. Once you have found a good representation at each level, we can use it to initiate and successfully train a deep network using supervised optimization.

Each level of abstraction found in the brain consists of "activation" (neural stimulation) of a small subset of a large number of functions that generally do not mutually exclude each other. Because of that they form a distributed representation: information is not localized in a particular neuron but distributed in many.

Many existing machine learning algorithms are locally in the input space: for learning functions that behave differently in different areas of the data space, they require different tuning parameters for each of these regions. Unlike learning methods based on local generalization, the total number of patterns that can be distinguished using a distributed representation can be exponential with the representation dimension.

8.3.1. Energy-Based Models (EBM) and Restricted Boltzmann Machines (RBM)

Energy-based models associate a scalar energy to each configuration of the interested variables. The target of learning process is to achieve maximum or minimum of this energy. Energy-based probabilistic models define a probability distribution through an energy function, as follows:

$$p(x) = \frac{e^{-E(x)}}{\sum_{\tilde{x}} e^{-E(\tilde{x})}} \tag{8.1}$$

An energy-based model can be trained with gradient based algorithms. For example we can define the log-likelihood and then the loss function as being the negative log-likelihood will be used in the training process:

$$\mathcal{L}(\theta,\mathcal{D})=\frac{1}{N}\sum_{x^{(i)}\in\mathcal{D}}\log\left[p\left(x^{(i)}\right)\right] \tag{8.2}$$

$$\ell(\theta,\mathcal{D})=-\mathcal{L}(\theta,\mathcal{D}) \tag{8.3}$$

similar to the learning algorithm of neural networks described in Chapter 6, where for a $\theta-$ the parameters of the model:

$$\frac{\partial\ell(\theta,\mathcal{D})}{\partial\theta}=-\frac{1}{N}\sum_{x^{(i)}\in\mathcal{D}}\log\frac{\partial\log p\left(x^{(i)}\right)}{\partial\theta} \tag{8.4}$$

When the object contains some non-observable parameters, denoted as ***h*** (***hidden***) we can formulate as:

$$P(x)=\sum_{h}P(x,h)=\sum_{h}\frac{e^{-E(x,h)}}{\sum_{\tilde{x}}e^{-E(\tilde{x},h)}} \tag{8.5}$$

With this hidden part *h*, the Eq. (8.1) can be converted to:

$$P(x)=\frac{e^{-\mathcal{F}(x)}}{\sum_{\tilde{x}}e^{-\mathcal{F}(\tilde{x})}} \tag{8.6}$$

with $\mathcal{F}(x)=-\log\sum_{h}e^{-E(x,h)}$ is called the free energy function.

The data negative log-likelihood gradient then has a particularly interesting form:

$$-\frac{\partial\log p(x)}{\partial\theta}=\frac{\partial\mathcal{F}(x)}{\partial\theta}-\sum_{\tilde{x}}p(\tilde{x})\frac{\partial\mathcal{F}(\tilde{x})}{\partial\theta} \tag{8.7}$$

Notice that the above gradient contains two terms, the first term increases the probability of training data (by reducing the corresponding free energy), while the second term decreases the probability of samples generated by the model. It is usually difficult to determine this gradient analytically, as it involves the computation of $E_P\left[\frac{\partial \mathcal{F}(x)}{\partial \theta}\right]$. This is nothing less than an expectation over all possible configurations of the input x (under the distribution P formed by the model).

The first step in making this computation tractable is to estimate the expectation using a fixed number of model samples, which are denoted as $\mathcal{N}$. The gradient can then be written as:

$$-\frac{\partial \log p(x)}{\partial \theta} \approx \frac{\partial \mathcal{F}(x)}{\partial \theta} - \frac{1}{|\mathcal{N}|}\sum_{\tilde{x}\in\mathcal{N}} \frac{\partial \mathcal{F}(\tilde{x})}{\partial \theta} \tag{8.8}$$

where we would ideally like elements $\tilde{x}$ of $\mathcal{N}$ to be sampled according to P. The problem now is how to extract these samples $\mathcal{N}$. The Markov Chain Monte Carlo methods are especially well suited for models such as the Restricted Boltzmann Machines (RBM), a specific type of EBM.

Boltzmann Machines (BMs) are a special case of log-linear Markov Random Field (MRF), i.e., for which the energy function is linear in its free parameters. By having more hidden variables (also called hidden units), the modeling capacity of the Boltzmann Machine (BM) can be increased. Restricted Boltzmann Machines are BMs, which have no connections between 2 hidden variables and between 2 non-hidden variables (similar to the requirement of feedforward neural networks, where there should not be no weight connected neurons belonging to the same layer). A graphical presentation of an RBM is shown in Fig. 8.2, where the non-hidden variables are presented as v_i nodes and the hidden variables are presented as h_j nodes.

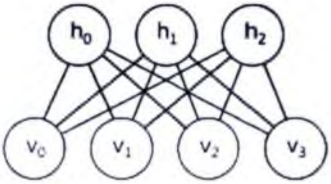


Figure 8.2. A simplified structure of RBM

The energy function $E(v,h)$ of an RBM can be defined as:

$$E(v,h) = -b^T v - c^T h - h^T W v \tag{8.9}$$

where W represents the weights connecting hidden and non-hidden units and b, c are the offsets of the non-hidden and hidden layers respectively.

This translates directly to the following free energy formula:

$$\mathcal{F}(v) = -b^T v - \sum_i \log \sum_{h_i} e^{h_i(c_i + W_i v)} \tag{8.10}$$

Because in RBMs, non-hidden and hidden units are conditionally independent, then:

$$\begin{aligned} p(h \mid v) &= \prod_i p(h_i \mid v) \\ p(v \mid h) &= \prod_j p(v_j \mid h) \end{aligned} \tag{8.11}$$

In the commonly studied case of using binary units (where v_j and $h_i \in \{0,1\}$) we obtain from Eq. (8.4) and (8.9) a probabilistic version of the usual neuron activation function:

$$P(h_i = 1 \mid v) = sigm(c_i + W_i v) \tag{8.12}$$

$$P(v_j = 1 \mid h) = sigm(b_j + W_j' h) \tag{8.13}$$

The free energy of an RBM with binary units further simplifies to:

$$\mathcal{F}(v) = -b^T v - \sum_i \log\left(1 + e^{(c_i + W_i v)}\right) \tag{8.14}$$

a) Update Equations with Binary Units

Combining Eqs. (8.8) with (8.14), we obtain the following log-likelihood gradients for an RBM with binary units:

$$
\begin{aligned}
-\frac{\partial \log p(v)}{\partial W_{ij}} &= E_v\left[p(h_i \mid v)\cdot v_j\right] - v_j^{(i)}\cdot sigm\left(W_i \cdot v^{(i)} + c_i\right)\\
-\frac{\partial \log p(v)}{\partial c_i} &= E_v\left[p(h_i \mid v)\right] - sigm\left(W_i \cdot v^{(i)}\right)\\
-\frac{\partial \log p(v)}{\partial b_j} &= E_v\left[p\left(v_j \mid h\right)\right] - v_j^{(i)}
\end{aligned}
\tag{8.15}
$$

b) Sampling In an RBM

Samples of $p(x)$ can be obtained by running a Markov chain to convergence, using Gibbs sampling as the transition operator.

Gibbs sampling of the joint of N random variables $S = (S_1, \ldots, S_N)$ is done through a sequence of N sampling sub-steps of the form $S_i \sim p(S_i \mid S_{-i})$ where S_{-i} contains the $N-1$ other random variables in S excluding S_i.

For RBMs, S consists of the set of visible and hidden units. However, since they are conditionally independent, one can perform block Gibbs sampling. In this setting, visible units are sampled simultaneously given fixed values of the hidden units. Similarly, hidden units are sampled simultaneously given the visible units. A step in the Markov chain is thus taken as follows:

$$
\begin{aligned}
h^{(n+1)} &\sim sigm\left(W' \cdot v^{(n)} + c\right)\\
v^{(n+1)} &\sim sigm\left(W \cdot h^{(n+1)} + b\right)
\end{aligned}
\tag{8.16}
$$

where $h^{(n)}$ refers to the set of all hidden units at the n-th step of the Markov chain. What it means is that, for example, $h_i^{(n+1)}$ is randomly chosen to be 1 (versus 0) with probability $sigm\left(W_i' \cdot v^{(n)} + c_i\right)$, and similarly, $v_j^{(n+1)}$ is randomly chosen to be 1 (versus 0) with probability $sigm\left(W_j h^{(n+1)} + b_j\right)$.

As $t \to \infty$, samples $\left(v^{(t)}, h^{(t)}\right)$ are guaranteed to be accurate samples of $p(v, h)$.

In theory, each parameter update in the learning process would require running one such chain to convergence. It is needless to say that doing so would be prohibitively expensive. As such, several algorithms have been devised for RBMs, in order to efficiently sample from $p(v,h)$ during the learning process.

8.3.2. Convolutional neural networks

Deeply supervised neural networks were generally difficult to train before using unsupervised initial training, there is one exception: convolutional neural networks. Convolutional networks were inspired by the structure of the visual system proposed by [Hubel62]. The first computational models based on these local links between neurons and hierarchically organized image transformations are described in [Fukushima80]. After that, LeCun, continuing this idea, designed and trained the convolutional networks obtaining very good performances [LeRoux08] in classical pattern recognition benchmarks. Modern understanding of the physiology of the visual system is consistent with the convolutional networks [Serre07]. Until now, pattern recognition systems based on convolutional neural networks are among the best ones.

When it comes to discussing the structure of the deep networks and its adaptation, the example of convolutional neural networks is interesting because they have much more layers than classical neural networks, whose are very hard to train due to the huge number of nonlinear parameters.

The convolutional neural networks are organized in layers of two types: convolutional layers and subsampling layers. Each layer has neurons bounded to a fixed position on the actual part (view field) of the input image. A neuron at each location of a layer has a set of input weights associated with the corresponding neurons in a rectangular patch from the previous layer.

One hypothesis is that if each neuron has small number of inputs, then that helps the error gradients during the training process spread through more layers without scattering so much, i.e. not reducing the effect of training samples too much. Another hypothesis is that the hierarchical link structure is a very strong candidate, particularly suitable for visual tasks, and sets the parameters of the entire network in a favorable region from which the gradient optimizatior algorithms are effective.

8.3.3. Auto-Encoders

Further examples of deep networks use a particular type of neural networl as a component of a particular neural network type: auto-encoder [Hinton94] Since training an automated coder seems easier, they are used as structura elements for deep network searching, where each level is associated with ar auto-encoder that can be trained separately [Bengio07].

The auto-encoder is trained to encode the input **x** into a certair representation of **c(x)**, from which we can reconstruct the input. It means the output of the auto-encoder is exactly the input. The auto-encoder has the ability to capture the multimodal aspects of the input distribution. The target is tc achieve **c(x)**, that can be seen as a distributed representation capturing the mair factors of data variations.

Experiments reported in [Bengio07] suggest that in practice, when trainec with a stochastic gradient descent algorithm, non-linear auto-encoders witl more hidden units than inputs achieve useful representations. A single-layeı auto-encoder with non-linear hidden units requires small value weights in the first layer and bigger weights in the second layer. The optimization algorithn will find coding that works well for examples similar to those in the training set, which we need. This means that the representation describes the statistical regularities detected from the training set but not tends to describe (discover) the repeating purpose (the outputs are equal the inputs).

There are different ways to avoid learning the identity transfer functior with an auto-encoder with more hidden units than input, while still keeping useful features from the input in its hidden representation. One of these solutions is to add noise in the encoding. Another solution is based on sparsity constraints.

The auto-encoders can have very large capacities and still do not learr identities because they not only try to encode input but also capture statistical features from the input data. A good example of the auto-encoders is the so-called *denoising auto-encoder* [Vincent08]. The denoising auto-encoder minimizes the error during reconstruction of the input signal from the stochastically damaged transformed inputs.

The text focused on a specific family of algorithms, the Deep Belief network, and their components, on the Restricted Boltzmann or various types of auto-encoders that can be interconnected to create a deep architecture It has been found that this optimization principle is in fact the so-called follow-up methods in which a number of progressively more difficult optimization problems have been solved. This suggested new possibilities for deep architectural optimization either by tracking solutions along the regression path, or by presenting a sequence of selected examples illustrating gradually more complex concepts in a manner analogous to the way students learn or animals.

8.4. EXAMPLE OF DEEP LEARNING APPLICATION

In this example (called *Training a Deep Neural Network for Digit Classification* provided by Matlab [WebMatlab]), we demonstrate the application of Neural Network Toolbox™ to train a deep neural network to classify images of digits.

The deep networks contains multiple hidden layers that can be useful for solving classification problems with complex data, such as images. Each layer will be responsible for learning (and later generalizing) features at a different level of abstraction. However, training deep networks is more difficult in practice than training classical feedforward networks such as MLP. One of the effective training method for a deep network is the layer-by-layer training. We can do this by training an autoencoder for each desired hidden layer.

This example shows you how to train a deep network with two hidden layers to classify digits in images. First we train each the hidden layer using autoencoders. Then we train a final softmax layer, and join the layers together to form a deep network, which is next trained using a supervised algorithm.

This example uses 5000 of 28-by-28 pixels synthetic images for training the network. The images have been generated by applying random affine transformations to digit images created using different fonts.

```
% Load the training data into memory
[xTrainImages, tTrain]=digittrain_dataset;
```

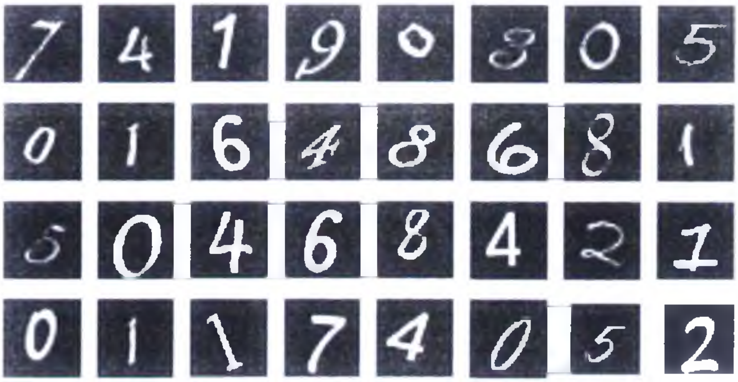


Figure 8.3 Samples of deformed characters used for the training process

Each image from the dataset was already labeled with the correct target value. This is very convenient for supervised learning algorithms. To use the images for training a neural network, Matlab requires to arrange them into a matrix where each column represents a single image. We can do this by stacking the columns of an image to form a vector, and then forming a matrix from these vectors.

```
% Get the number of pixels in each image
imageWidth=28;
imageHeight=28;
inputSize=imageWidth*imageHeight;
% Turn the training images into vectors and put them in a
matrix
xTrain=zeros(inputSize, numel(xTrainImages));
for i=1:numel(xTrainImages)
    xTrain(:,i)=xTrainImages{i}(:);
end
```

a) Training the first Autoencoder

We begin with training a sparse autoencoder on the training data without using the labels.

An autoencoder is a neural network which should have its output equal exactly its input. To avoid the identical learning, we should use the autoencoder with less hidden neurons than the number of inputs (or outputs).

When the number of neurons in the hidden layer is less than the size of the input, the autoencoder will learn a compressed representation of the input.

You can create an autoencoder by creating a feed-forward network, and then modifying some of the settings.

```
% Set the size of the hidden layer for the autoencoder.
% It is a good idea to make this smaller than the input size.
hiddenSize1=100;

% Create the network.
% Set the number of training epochs and the training function
autoenc1=feedforwardnet(hiddenSize1);
autoenc1.trainFcn='trainscg';
autoenc1.trainParam.epochs=400;

% Do not use process functions at the input or output
autoenc1.inputs{1}.processFcns={};
autoenc1.outputs{2}.processFcns={};

% Set the transfer function for both layers
% to the logistic sigmoid
autoenc1.layers{1}.transferFcn='logsig';
autoenc1.layers{2}.transferFcn='logsig';

% Use all of the data for training
autoenc1.divideFcn='dividetrain';
```

The autoencoder is trained with the input data and target data are identical.

```
% Train the autoencoder
autoenc1=train(autoenc1,xTrain,xTrain);
view(autoenc1);
```

The diagram below of the autoencoder showing the size of the input, output and hidden layer, as well as the transfer functions for the two layers .

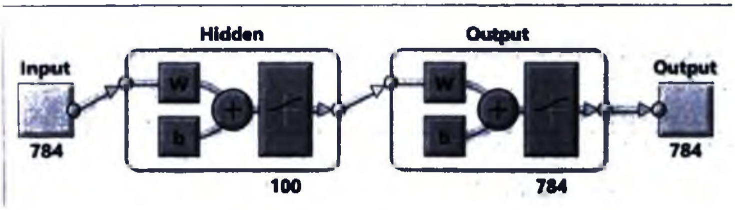


Figure 8.4. The structure of the 1st autoencoder

Visualizing the results from the first Autoencoder

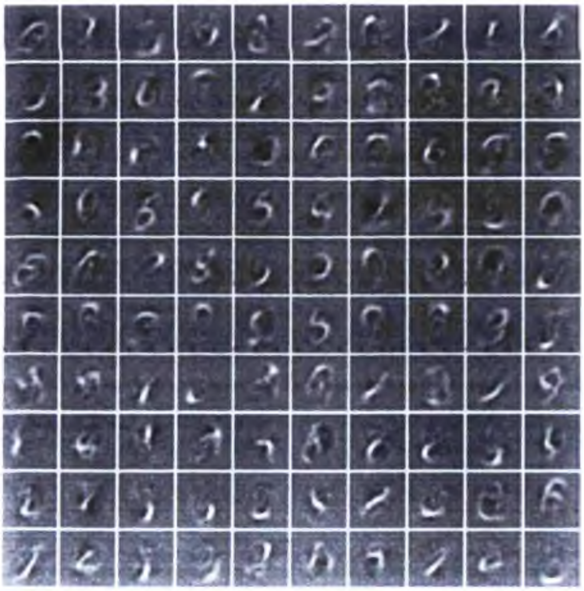


Figure 8.5. Results of training the 1st autoencoder to compress the input image

After training the autoencoder, we can gain an insight into the features it has learned by visualizing them. Each neuron in the hidden layer will have a vector of weights associated with it in the input layer which will be tuned to respond to a particular visual feature. By reshaping these weight vectors, we can view a representation of these features as seen on Fig. ... It can be seen that the features learned by the autoencoder represent curls and stroke patterns from the digit images.

The 100 dimensional output from the hidden layer of the autoencoder is a compressed version of the input, which summarizes its response to the features that were visualized above. Train the next autoencoder on a set of these vectors extracted from the training data.

```
% Create an empty network
autoencHid1=network;

% Set the number of inputs and layers
autoencHid1.numInputs=1;
autoencHid1.numlayers=1;

% Connect the 1st (and only) layer to the 1st input,
% Connect the 1st layer to the output
autoencHid1.inputConnect(1,1)=1;
autoencHid1.outputConnect=1;

% Add a connection for a bias term to the first layer
autoencHid1.biasConnect=1;

% Set the size of the input and the 1st layer
autoencHid1.inputs{1}.size=inputSize;
autoencHid1.layers{1}.size=hiddenSize1;

% Use the logistic sigmoid transfer function for the 1st layer
autoencHid1.layers{1}.transferFcn='logsig';

% Copy the weights and biases from the 1st layer
% of the trained autoencoder to this network
```

```
autoencHid1.IW{1,1}=autoenc1.IW{1,1};
autoencHid1.b{1,1}=autoenc1.b{1,1};
view(autoencHid1);
```

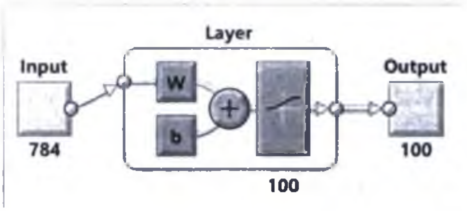


Figure 8.6. The structure of the 1st trained layer achieved from the 1st autoencoder

We can now generate the features that will be used to train the second autoencoder. This is done by evaluating the truncated autoencoder on the training data.

```
feat1=autoencHid1(xTrain);
```

b) Training the second Autoencoder

After training the first autoencoder, you train the second autoencoder in a similar way. The main difference is that the training data is the features generated from the hidden layer of the previous autoencoder. Once again, we create a feed-forward network and then modify the settings.

```
% Create the network. Set the number hidden neurons,
% Set the number of training epochs and the training function
hiddenSize2=50;
autoenc2=feedforwardnet(hiddenSize2);
autoenc2.trainFcn='trainscg';
autoenc2.trainParam.epochs=100;

% Do not use process functions at the input or output
autoenc2.inputs{1}.processFcns={};
autoenc2.outputs{2}.processFcns={};
```

```
% Set the transfer function for both layers
% to the logistic sigmoid
autoenc2.layers{1}.transferFcn='logsig';
autoenc2.layers{2}.transferFcn='logsig';

% Use all of the data for training
autoenc2.divideFcn='dividetrain';
```

After creating the network, we set the performance function and the values for the performance function parameters.

```
% Use the mean squared error with L2 weight
% and sparsity regularizers for the performance functior
autoenc2.performFcn='msesparse';

% Select the parameters of performance function
autoenc2.performParam.L2WeightRegularization=0.002;
autoenc2.performParam.sparsityRegularization=4;
autoenc2.performParam.sparsity=0.1;
```

Next, we train this autoencoder on the features generated from the previous autoencoder.

```
% Train the second autoencoder
autoenc2=train(autoenc2,feat1,feat1);
view(autoenc2);
```

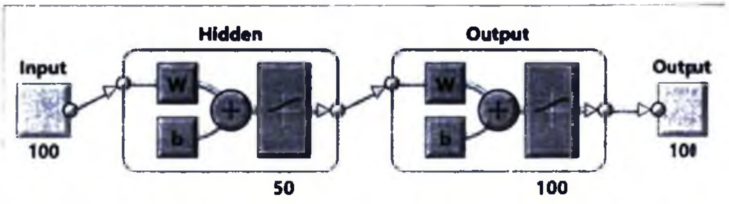


Figure 8.7. The structure of the 2nd encoder

The above diagram of second autoencoder is similar to the first, but the sizes of the layers are different. Next, we create a version of the second autoencoder with the final layer removed.

```
% Create an empty network
autoencHid2=network;

% Set the number of inputs and layers
autoencHid2.numInputs=1;
autoencHid2.numlayers=1;

% Connect the 1st (and only) layer to the 1st input,
% Connect the 1st layer to the output
autoencHid2.inputConnect(1,1)=1;
autoencHid2.outputConnect=1;

% Add a connection for a bias term to the 1st layer
autoencHid2.biasConnect=1;

% Set the size of the input and the 1st layer
autoencHid2.inputs{1}.size=hiddenSize1;
autoencHid2.layers{1}.size=hiddenSize2;

% Use the logistic sigmoid transfer function for the 1st layer
autoencHid2.layers{1}.transferFcn='logsig';

% Copy the weights and biases from the 1st layer
% of the 2nd trained autoencoder to this network
autoencHid2.IW{1,1}=autoenc2.IW{1,1};
autoencHid2.b{1,1}=autoenc2.b{1,1};
```

We can extract a second set of features by passing the previous set through the second truncated autoencoder.

```
feat2=autoencHid2(feat1);
```

The original vectors in the training data had 784 dimensions. After passing them through the first autoencoder, this was reduced to 100 dimensions. After using the second autoencoder, this was reduced again to 50 dimensions. You will now train a final layer to classify these 50 dimensional vectors into different digit classes.

c) Training the final Softmax Layer

We will create a softmax layer, and train it on the output from the hidden layer of the second autoencoder.

```
% Create an empty network
finalSoftmax=network;

% Set the number of inputs and layers
finalSoftmax.numInputs=1;
finalSoftmax.numLayers=1;

% Connect the 1st (and only) layer to the 1st input,
% Connect the 1st layer to the output
finalSoftmax.inputConnect(1,1)=1;
finalSoftmax.outputConnect=1;

% Add a connection for a bias term to the first layer
finalSoftmax.biasConnect=1;

% Set the size of the input and the 1st layer
finalSoftmax.inputs{1}.size=hiddenSize2;
finalSoftmax.layers{1}.size=10;

% Use the softmax transfer function for the 1st layer
finalSoftmax.layers{1}.transferFcn='softmax';

% Use all of the data for training
finalSoftmax.divideFcn='dividetrain';

% Use the cross-entropy performance function
finalSoftmax.performFcn='crossentropy';
```

```
% Select the number of training epochs
% Select the training function and train the network
finalSoftmax.trainFcn='trainscg';
finalSoftmax.trainParam.epochs=400;
finalSoftmax=train(finalSoftmax,feat2,tTrain);
```

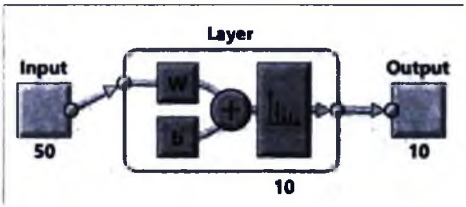


Figure 8.8. The structure of the softmax layer

d) Forming a Multilayer Neural Network

You have trained three separate components of a deep neural network in isolation.

You join these layers together to form a multilayer neural network. You create the neural network manually, and then configure the settings, and copy the weights and biases from the autoencoders and softmax layer.

```
% Create an empty network
finalNetwork=network;

% Specify one input and three layers
finalNetwork.numInputs=1;
finalNetwork.numLayers=3;

% Connect the 1st layer to the input
finalNetwork.inputConnect(1,1)=1;

% Connect the 2nd layer to the 1st layer
finalNetwork.layerConnect(2,1)=1;

% Connect the 3rd layer to the 2nd layer
finalNetwork.layerConnect(3,2)=1;
```

```
% Connect the output to the 3rd layer
finalNetwork.outputConnect(1,3)=1;

% Add a connection for a bias term for each layer
finalNetwork.biasConnect=[1; 1; 1];

% Set the size of the input
finalNetwork.inputs{1}.size=inputSize;

% Set the size of the 1st layer to the same as in autoencHid1
finalNetwork.layers{1}.size=hiddenSize1;

% Set the size of the 2nd layer to the same as in autoencHid2
finalNetwork.layers{2}.size=hiddenSize2;

% Set the size of the 3rd layer to the same as in finalSoftmax
finalNetwork.layers{3}.size=10;

% Set the transfer function for the 1st layer to the same
% as in autoencHid1
finalNetwork.layers{1}.transferFcn='logsig';

% Set the transfer function for the 2nd layer to the same
% as in autoencHid2
finalNetwork.layers{2}.transferFcn='logsig';

% Set the transfer function for the 3rd layer to the same
% as in finalSoftmax
finalNetwork.layers{3}.transferFcn='softmax';

% Use all of the data for training
finalNetwork.divideFcn='dividetrain';

% Copy the weights and biases from the three networks
% that have already been trained
finalNetwork.IW{1,1}=autoencHid1.IW{1,1};
finalNetwork.b{1}=autoencHid1.b{1,1};
finalNetwork.LW{2,1}=autoencHid2.IW{1,1};
finalNetwork.b{2}=autoencHid2.b{1,1};
finalNetwork.LW{3,2}=finalSoftmax.IW{1,1};
finalNetwork.b{3}=finalSoftmax.b{1,1};

% Use the cross-entropy performance function
finalNetwork.performFcn='crossentropy';
% Select the number of training epochs and
% the training function
finalNetwork.trainFcn='trainscg';
finalNetwork.trainParam.epochs=100;
```

The created multilayer network is shown in Fig. 8.9.

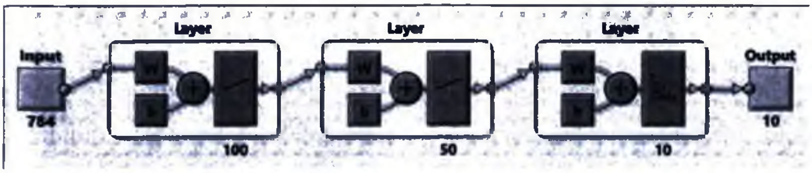


Figure 8.9. The final structure of the deep network

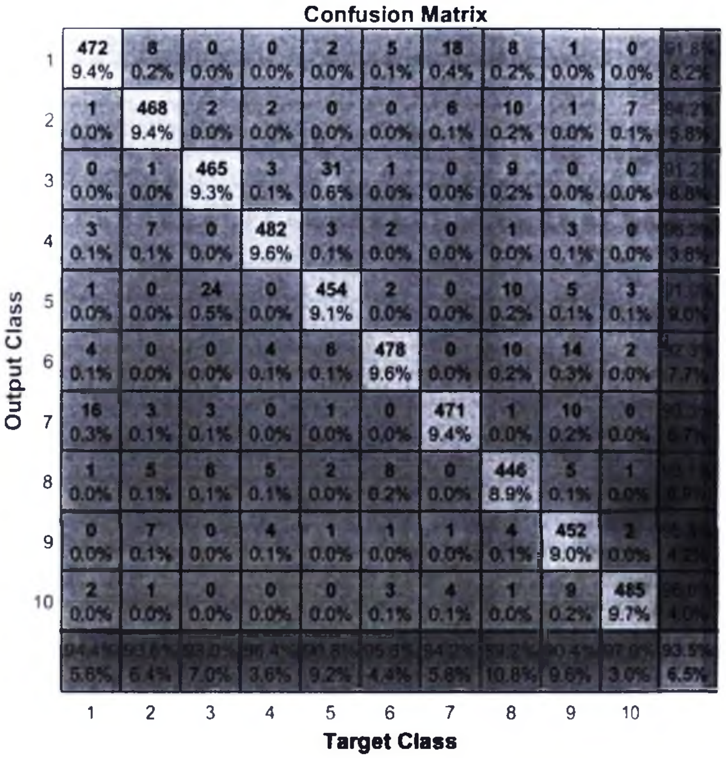
With the full deep network formed, you can compute the results on the test set. Before you can do this, you have to reshape the test images into a matrix, as was done for the training set.

```
% Load the test images
[xTestImages, tTest]=digittest_dataset;

% Turn the test images into vectors and put them in a matrix
xTest=zeros(inputSize, numel(xTestImages));
for i=1:numel(xTestImages)
   xTest(:,i)=xTestImages{i}(:);
end
```

We can visualize the results with a confusion matrix. The numbers in the bottom right hand square of the matrix will give the overall accuracy.

```
y=finalNetwork(xTest);
plotconfusion(tTest,y);
```



Confusion Matrix

Output Class \ Target Class	1	2	3	4	5	6	7	8	9	10	
1	472 9.4%	8 0.2%	0 0.0%	0 0.0%	2 0.0%	5 0.1%	18 0.4%	8 0.2%	1 0.0%	0 0.0%	91.8% 8.2%
2	1 0.0%	468 9.4%	2 0.0%	2 0.0%	0 0.0%	0 0.0%	6 0.1%	10 0.2%	1 0.0%	7 0.1%	94.2% 5.8%
3	0 0.0%	1 0.0%	465 9.3%	3 0.1%	31 0.6%	1 0.0%	0 0.0%	9 0.2%	0 0.0%	0 0.0%	91.2% 8.8%
4	3 0.1%	7 0.1%	0 0.0%	482 9.6%	3 0.1%	2 0.0%	0 0.0%	1 0.0%	3 0.1%	0 0.0%	96.2% 3.8%
5	1 0.0%	0 0.0%	24 0.5%	0 0.0%	454 9.1%	2 0.0%	0 0.0%	10 0.2%	5 0.1%	3 0.1%	91.0% 9.0%
6	4 0.1%	0 0.0%	0 0.0%	4 0.1%	6 0.1%	478 9.6%	0 0.0%	10 0.2%	14 0.3%	2 0.0%	92.3% 7.7%
7	16 0.3%	3 0.1%	3 0.1%	0 0.0%	1 0.0%	0 0.0%	471 9.4%	1 0.0%	10 0.2%	0 0.0%	93.3% 6.7%
8	1 0.0%	5 0.1%	6 0.1%	5 0.1%	2 0.0%	8 0.2%	0 0.0%	446 8.9%	5 0.1%	1 0.0%	93.1% 6.9%
9	0 0.0%	7 0.1%	0 0.0%	4 0.1%	1 0.0%	1 0.0%	1 0.0%	4 0.1%	452 9.0%	2 0.0%	95.8% 4.2%
10	2 0.0%	1 0.0%	0 0.0%	0 0.0%	0 0.0%	3 0.1%	4 0.1%	1 0.0%	9 0.2%	485 9.7%	96.0% 4.0%
	94.4% 5.6%	93.6% 6.4%	93.0% 7.0%	96.4% 3.6%	90.8% 9.2%	95.6% 4.4%	94.2% 5.8%	89.2% 10.8%	90.4% 9.6%	97.0% 3.0%	93.5% 6.5%

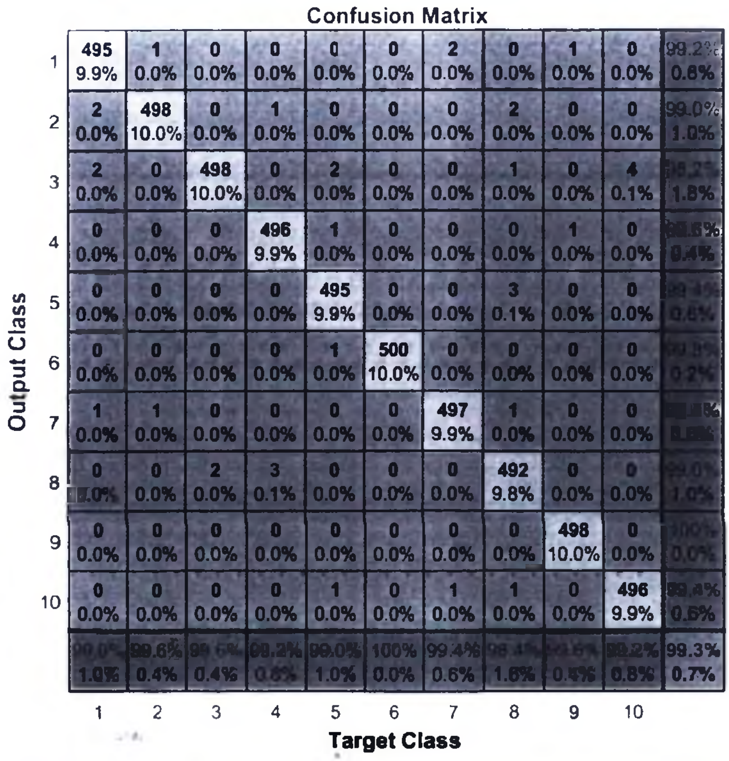
Figure 8.10. Confusion matrix for the deep network after the individual training for each layer

e) Fine tuning the Deep Neural Network

The results for the deep neural network can be improved by performing backpropagation on the whole multilayer network. This process is often referred to as fine tuning.

We fine tune the network by retraining it on the training data in a supervised fashion. The results can be shown again using a confusion matrix.

```
finalNetwork=train(finalNetwork,xTrain,tTrain);
y=finalNetwork(xTest);
plotconfusion(tTest,y);
```


Confusion Matrix
Output Class
Target Class

	1	2	3	4	5	6	7	8	9	10	
1	495 9.9%	1 0.0%	0 0.0%	0 0.0%	0 0.0%	0 0.0%	2 0.0%	0 0.0%	1 0.0%	0 0.0%	99.2% 0.8%
2	2 0.0%	498 10.0%	0 0.0%	1 0.0%	0 0.0%	0 0.0%	0 0.0%	2 0.0%	0 0.0%	0 0.0%	99.0% 1.0%
3	2 0.0%	0 0.0%	498 10.0%	0 0.0%	2 0.0%	0 0.0%	0 0.0%	1 0.0%	0 0.0%	4 0.1%	98.2% 1.8%
4	0 0.0%	0 0.0%	0 0.0%	496 9.9%	1 0.0%	0 0.0%	0 0.0%	0 0.0%	1 0.0%	0 0.0%	99.6% 0.4%
5	0 0.0%	0 0.0%	0 0.0%	0 0.0%	495 9.9%	0 0.0%	0 0.0%	3 0.1%	0 0.0%	0 0.0%	99.4% 0.6%
6	0 0.0%	0 0.0%	0 0.0%	0 0.0%	1 0.0%	500 10.0%	0 0.0%	0 0.0%	0 0.0%	0 0.0%	99.8% 0.2%
7	1 0.0%	1 0.0%	0 0.0%	0 0.0%	0 0.0%	0 0.0%	497 9.9%	1 0.0%	0 0.0%	0 0.0%	[illegible]
8	0 0.0%	0 0.0%	2 0.0%	3 0.1%	0 0.0%	0 0.0%	0 0.0%	492 9.8%	0 0.0%	0 0.0%	99.0% 1.0%
9	0 0.0%	0 0.0%	0 0.0%	0 0.0%	0 0.0%	0 0.0%	0 0.0%	0 0.0%	498 10.0%	0 0.0%	100% 0.0%
10	0 0.0%	0 0.0%	0 0.0%	0 0.0%	1 0.0%	0 0.0%	1 0.0%	1 0.0%	0 0.0%	496 9.9%	99.4% 0.6%
	99.0% 1.0%	99.6% 0.4%	99.6% 0.4%	99.2% 0.8%	99.0% 1.0%	100% 0.0%	99.4% 0.6%	98.4% 1.6%	99.6% 0.4%	99.2% 0.8%	99.3% 0.7%

Figure 8.11. Confusion matrix for the deep network after the final supervised training

f) Summary

This example showed how to train a deep neural network to classify digits in images using the Neural Network Toolbox™. The steps that have been outlined could be applied to other similar problems such as classifying images of letters, or even small images of objects of a specific category.

8.5. CONCLUSIONS

The example of Deep Learning in hand-written character recognition has concluded the Chapter 8, which is also the last chapter of this textbook.Through the content of the book, a short and brief introduction to the selected topics of Artificial Intelligence was presented. But for new, very dynamic field of research and application such as AI, this is unquestionably not enough. Readers are encouraged to use the references to broaden the knowledge and to practice with project for further improvement of skills.

References

[Abelson96] Abelson H.; Sussman G. J.; Sussman J. (1996). *Structure and interpretation of computer*, 2nd ed., MIT Electrical Engineering and Computer Science Series.

[Barr81] Barr A., Feigenbaum A. E. (1981). *The Handbook of Artificial Intelligence*, HeurisTech Press, vol. 1.

[Back91] Bäck T., Hoffmeister F., Schwefel H. P. (1991). *A Survey of Evolution Strategies*, Proceedings of the 4th International Conference on Genetic Algorithms (ICGA), pp. 2-9.

[Bellman78] Bellman R. (1978). *An Introduction to Artificial Intelligence: Can Computers Think?*, Boyd & Fraser, USA.

[Bengio07] Bengio Y.; Lamblin P.; Popovici D.; Larochelle H. (2007). *Greedy layer-wise training of deep networks*, Advances in Neural Information Processing Systems 19 (NIPS'06), pp. 153 – 160, MIT Press.

[Bezdek81] Bezdek J. C. (1981). *Pattern Recognition with Fuzzy Objective Function Algorithms*, Plenum Press, New York.

[Booker87] Booker L. B. (1987). *Improving Search in Genetic Algorithms*, Genetic Algorithms and Simulated Annealing: An Overview, Morgan Kaufmann Publishers, San Mateo, CA, pp. 61-73.

[Brindle80] Brindle Anne (1980). *Genetic Algorithms for Function Optimization*, PhD Thesis, Department of Computing Science, University of Alberta, Canada.

[Carillo01] Carrillo-Ureta G. E., Robrts P. D., Becerra V. M. (2001). *Genetic algorithm for optimal control of beer fermentation*, Proc. of IEEE IS on Intelligent Control, Mexico City, 391-396.

[Charniak85] Charniak E., McDermott D. (1985). *Introduction to Artificial Intelligence*, Addison-Wesley, USA.

[Church41] Church A. (1941), *The calculi of lambda conversion*, Princeton University Press, 1941.

[Dunn73] Dunn J. C. (1973). *A Fuzzy Relative of the ISODATA Process and Its Use in Detecting Compact Well-Separated Clusters*, Journal of Cybernetics, vol. 3, p. 32 – 57.

[Erhan09] Erhan D.; Manzagol P.-A.; Bengio Y.; Bengio S.; Vincent P. (2009). *The difficulty of training deep architectures and the effect of unsupervised pretraining*, Proceedings of the 12th IC on Artificial Intelligence and Statistics (AISTATS'09), pp. 153–160.

[Fogel66] Fogel L.; Owens A.; Walsh M. (1966). *Artificial intelligence through simulated evolution*, Chichester.

[Foyya96] Foyyad U. M.; G. Piatetzky-Shapiro; P. Smyth, R. Uthurusamy; (1996). *Advances in Knowledge Discovery and Data Mining*, Cambridge, MA:AIII Press/MIT Press.

[Fukushima80] K. Fukushima, (1980). *Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position*, Biological Cybernetics, vol. 36, pp. 193–202.

[Gen99] Gen, M.; Li, Y.; Ida, K. (1999). *Solving Multi-Objective Transportation Problem by Spanning Tree-Based Genetic Algorithm*, IEICE Transactions on Fundamental, vol. E82-A, no. 12, pp. 2802 – 2810.

[Goldberg89] Goldberg D. E. (1989). *Genetic Algorithms in Search, Optimization, and Machine Learning*, Addison-Wesley Publishing Company Inc.

[Haykin99] Haykin S. S. (1999). *Neural Networks: A Comprehensive Foundation*, Prentice Hall.

[Hebb49] Hebb D. O. (1949). *The organization of behavior*. Wiley & Sons, NY.

[Hinton94] G. E. Hinton, R. S. Zemel (1994), *Autoencoders, minimum description length, and Helmholtz free energy*, Advances in Neural Information Processing Systems 6 (NIPS'93), pp. 3–10, Morgan Kaufmann Publishers, Inc.

[Hinton06] G. E. Hinton, S. Osindero, Y. Teh, (2006). *A fast learning algorithm for deep belief nets*, Neural Computation, vol. 18, pp. 1527–1554.

[Holland75] Holland J.H. (1975), *Adaptation in natural and artificial systems*, Michigan.

[Hubel62] Hubel D. H.; Wiesel T. N. (1962). *Receptive fields, binocular interaction, and functional architecture in the cat's visual cortex*, Journal of Physiology (London). vol. 160, pp. 106–154.

[Kinnear94] K. E. Kinnear. (1994). *Advances in Genetic Programming*. MIT Press, Cambridge.

[Kohonen89] Kohonen Teuvo (1989). *Self-Organization and Associative Memory*, Springer Series in Information Sciences.

[Kolmogorov] Kolmogorov A. N., (1957). *On the representation of continuous functions of several variables by superposition of continuous functions of one variable and addition*, Dokl. Akad. Nauk SSSR, vol. 114, p. 953 – 956.

[Kosko88] Kosko Bart (1988). *Bidirectional Associative Memories*, IEEE Transactions on Systems, Man, and Cybernetics, vol. 18, No. 1, pp. 49 – 60.

[Koza92] Koza J. R. (1992). *Genetic Programming*, MIT Press.

[Larochelle09] H. Larochelle, Y. Bengio, J. Louradour, P. Lamblin (2009). *Exploring strategies for training deep neural networks*, Journal of Machine Learning Research, vol. 10, pp. 1–40.

[LeRoux08] Le Roux N., Bengio Y., (2008). *Representational power of Restricted Boltzman Machines and Deep Belief Networks*, Neural Computation, vol. 20, no. 6, pp. 1631–1649.

[Luger04] Luger, G.; Stubblefield W. (2004). *Artificial Intelligence: Structures and Strategies for Complex Problem Solving*, 5th edition, Benjamin/Cummings.

[McCarthy59] McCarthy J. (1959). *Recursive Functions of Symbolic Expressions and Their Computation by Machine*, Artificial Intelligence Project - RLE and MIT Computation Center, Memo 8.

[Maschek05] Maschek, M.K. (2005). *Applications of Evolutionary Learning in Macroeconomic Models*, PhD Thesis, Department of Economics, Simon Fraser University, BC, Canada.

[Michalewicz92] Z. Michalewicz, (1992). *Genetic Algorithms + Data Structures = Evolution Programs*, Springer-Verlag, Berlin Heidelberg.

[Michalewicz96] Michalewicz Z.; Schoenauer M. (1996). *Evolutionary computation for constrained parameter optimization problems*, Evolutionary Computation, Vol. 4, No. 1, pp. 1-32.

[Millington09] Millington Ian; Funge John (2009). *Artificial Intelligence for Games*, 2nd ed., CRC Press.

[Minsky91] Minsky Marvin (1961). *Steps Toward Artificial Intelligence*, Proceedings of the IRE, vol. 49, pp. 8-30.

[Oduguwa05] Oduguwa V.; Tiwari A.; Roy R. (2005). *Evolutionary computing in manufacturing industry: an overview of recent applications*, Applied Soft Computing 5, 281-299.

[Pena00] C. A. Peña-Reyes, M. Sipper, (2000). *Evolutionary computation in medicine: an overview*, Artificial Intelligence Med. 19, pp. 1-23.

[Potvin96], Potvin Jean-Yves (1996). *Genetic algorithms for the traveling salesman problem*, Annals of Operations Research 63, pp. 339-370.

[Pressman89] Pressman, Ian; David Singmaster (1989). *The Jealous Husbands and The Missionaries and Cannibals*, The Mathematical Gazette. The Mathematical Association. 73 (464): 73–81.

[Radcliffe95] N. J. Radcliffe, P. D. Surry, (1995). *Fundamental Limitations on Search Algorithms: Evolutionary Computing in Perspective*, Lecture Notes in Computer Science, Vol. 1000, J. Van Leeuwen (ed.), Springer-Verlag.

[Ranzato07] M. Ranzato, C. Poultney, S. Chopra, Y. LeCun, (2007). *Efficient learning of sparse representations with an energy-based model*, Advances in Neural Information Processing Systems 19, pp. 1137–1144, MIT Press.

[Rich09] Rich E.; Knight K. (2009). *Artificial Intelligence*. 2nd ed., McGraw-Hill, USA.

[Schaefer02] Schaefer Steve (2002). "MathRec Solutions (Tic-Tac-Toe)", http://www.mathrec.org/old/2002jan/solutions.html. retrieved 2015-09-18.

[Schalkoff90] Schalkoff J. R. (1990). *Artificial Intelligence: An Engineering Approach*, McGraw-Hill, USA.

[Schwefel95] Schwefel H.P., (1995). *Evolution and optimum seeking*, Chichester.

[Serre07] T. Serre, G. Kreiman, M. Kouh, C. Cadieu, U. Knoblich, T. Poggio, (2007). *A quantitative theory of immediate visual recognition*, Progress in Brain Research, Computational Neuroscience: Theoretical Insights into Brain Function, vol. 165, pp. 33–56.

[Stent73] Stent, G. (1973). *A physiological mechanism for Hebb's postulate of learning*, Proceedings of the National Academy of Sciences, USA.

[Vincent08] P. Vincent, H. Larochelle, Y. Bengio, P.A. Manzagol, (2008). *Extracting and composing robust features with denoising autoencoders*. Proceedings of the 25th IC on Machine Learning (ICML'08), pp. 1096–1103. ACM.

[WebEco] https://www.codeproject.com/Articles/1182210/Artificial-Intelligence

[WebFong] http://www.cs.sfu.ca/CourseCentral/310/pwfong/Lisp/3/tutorial3.html

[WebGao] http://logos.cs.uic.edu/476/resources/Prolog/foxGooseGrain.txt

[WebLisp] "The Jargon File - Lisp", http://www.catb.org/~esr/jargon/html/L/LISP.html

[WebLisp01] (2001) http://www.paulgraham.com/diff.html

[WebMatlab] https://www.mathworks.com/help/nnet/examples/training-a-deep-neural-network-for-digit-classification.html

[WebMisuku] http://www.aisb.org.uk/events/loebner-prize#Results2016

[WebMutation] *Genetics Home Reference: Mutations and Health*, https://ghr.nlm.nih.gov/primer/mutationsanddisorders/genemutation

[WebProlog] http://www.swi-prolog.org/

[WebTanimoto03] http://courses.cs.washington.edu/courses/cse341/03sp/

[WebUnix02] (2002) http://xahlee.info/UnixResource_dir/writ/jargons.html

[WikiAI] https://en.wikipedia.org/wiki/Artificial_intelligence

[WikiNeuron] https://en.wikipedia.org/wiki/Neuron

[WikiRiver] https://en.wikipedia.org/wiki/Fox,_goose_and_bag_of_beans_puzzle

[WikiTic] https://en.wikipedia.org/wiki/Tic-tac-toe

[Whitley97] D. Whitley, S. Rana, R. Heckendorn, (1997). *Representation Issues in Neighborhood Search and Evolutionary Algorithms*, Genetic Algorithms and Evolution Strategies in Engineering and Computer Science, D. Quagliarella, J. Periaux, C. Poloni, G. Winter (eds.), pp. 39-57, John Wiley.

[Winston92] Winston H. P. (1992)..*Artificial Intelligence*, Addison-Wesley, USA.

[Wolf12] Wolf, Mark J. P. (2012). *Encyclopedia of Video Games: The Culture, Technology, and Art of Gaming*. Greenwood Publishing Group. pp. 3–7.

[Wolsey98] L. A. Wolsey, (1998). *Integer Programming*, John Wiley & Sons Inc.

[Yu97] Y. Yu, M. C. Schell, J. B. Hang, (1997). *Decision theoretic steering and genetic algorithm optimization: application to stereotactic radiosurgery treatment planning*, Med. Phys 24, 1742-1750.

CONTENTS

LIST OF FIGURES

NHÀ XUẤT BẢN ĐẠI HỌC THÁI NGUYÊN
Địa chỉ: Phường Tân Thịnh - Thành phố Thái Nguyên - Tỉnh Thái Ngıyên
Điện thoại: 0208 3840023; Fax: 0208 3840017
Website: nxb.tnu.edu.vn * E-mail: nxb.dhtn@gmail.com

ARTIFICIAL INTELLIGENCE

Chịu trách nhiệm xuất bản:
PGS.TS. NGUYỄN ĐỨC HẠNH
Giám đốc - Tổng biên tập

Biên tập:	TRỊNH THANH ĐIỆP
Thiết kế bìa:	LÊ THÀNH NGUYÊN
Chế bản:	QUÁCH THỊ MAI
Sửa bản in:	ĐÀO THÁI SƠN

ISBN: 978-604-915-572-7
In 500 cuốn, khổ 17 x 24cm, tại Xưởng in - Nhà xuất bản Đại học Thái Nguyên (Địa chỉ: Phường Tân Thịnh - Thành phố Thái Nguyên - Tỉnh Thái Nguyên). Giấy phép xuất bản số: 3557-2017/CXBIPH/01-165/ĐHTN. Quyết định xuất bản số: 277/QĐ-NXBĐHTN. In xong và nộp lưu chiểu quý IV năm 2017.