Міністерство освіти і науки України

Київський національний університет технологій та дизайну

Кафедра комп'ютерних наук

METHODS AND SYSTEMS OF ARTIFICIAL INTELLIGENCE

SCHERBAN V.Y.
DEMKIVSKIY E.O.
DEMKIVSKA T.I.
SHRAMCHENKO B.L.
REZANOVA V.G.

Methods and systems of artificial intelligence

MINISTRY OF EDUCATION AND SCIENCE OF UKRAINE
KYIV NATIONAL UNIVERSITY OF TECHNOLOGY
AND DESIGN
DEPARTMENT OF COMPUTER SCIENCE

SCHERBAN V.Y., DEMKIVSKIY E.O., DEMKIVSKA T.I., SHRAMCHENKO B.L.,
REZANOVA V.G.

Methods and systems of artificial intelligence

Kyiv 2022

УДК 004.42
ББК 65.9(4Укр)306.4-6

Recommended by the Academic Council of the Kyiv National University of Technology and Design for a wide range of teachers, scientists, graduate students, masters and students of specialized higher education institutions, engineering and technical staff of the computer industry

(Protocol № 6 of 26 January 2022)

Team of authors:

SHCHERBAN V. YU. - Laureate of the State Prize of Ukraine in the field of science and technology, Academician of the International Academy of Computer Science and Systems, Doctor of Technical Sciences, Head of the Department of Computer Science, Kyiv National University of Technology and Design;

DEMKIVSKIY Y.O.- Ph.D., Associate Professor, Associate Professor, Department of Information Systems, Faculty of Computer Science and Cybernetics, Taras Shevchenko National University of Kyiv

DEMKIVSKA T..I. - Ph.D., Associate Professor, Associate Professor, Department of Computer Science and Technology, Kyiv National University of Technology and Design

SHRAMCHENKO B.L.. - Ph.D., Associate Professor, Associate Professor, Department of Computer Science and Technology, Kyiv National University of Technology and Design

REZANOVA V.G - Ph.D., Associate Professor, Associate Professor, Department of Computer Science and Technology, Kyiv National University of Technology and Design

Reviewers:

OPANASENKO V.M. - Doctor of Technical Sciences, Professor, leading researcher Institute of Cybernetics of the National Academy of Sciences of Ukraine

CHUPRINKA V.I. - Doctor of Technical Sciences, Professor of the Department of Computer Science, Kyiv National University of Technology and Design

Щ 610 SHCHERBAN V. Yu. Methods and systems of artificial intelligence / V. YU. Shcherban, Y.O. Demkivskiy, T,I. Demkivska, B. L. Shramchenko, V.G. Rezanova – К.: ТОВ "Фастбінд Україна", 2022. – 210 p.

ISBN 978-617-8237-17-2

The textbook is aimed at the formation of a basic set of knowledge for students of the specialty 122 Computer Science and student-oriented preparation of a bachelor's degree. The tutorial discusses the models and methods used in artificial intelligence systems. Considerable attention is paid to the design of artificial intelligence systems, the consideration of intelligent systems based on knowledge and means of representing and processing knowledge. The material is presented in accordance with the program of the discipline. Contains theoretical material, tasks for independent work of students and examples of their implementation, questions for self-control. The tutorial can be used by teachers, students and graduate students in the field of computer science.

УДК 004.42
ББК 65.9(4Укр)306.4-6

ISBN 978-617-8237-17-2

© V. Yu. Shcherban, 2022
© ТОВ "Фастбінд Україна", 2022

CONTENT

Introduction 5
1. Basic concepts and definitions 6
1.1. The concept of artificial intelligence. History of development. Intellectual task 6
1.2. The concept of intelligent system 9
1.3. List of control questions on the topic № 1 11
2. Ways of presenting an intellectual problem and methods of finding solutions .. 11
2.1. Ways of presenting an intellectual problem, their advantages and disadvantages 11
2.2. Search for solutions to an intellectual problem in space. Methods of blind search and heuristic search 12
2.3. Heuristic search methods 14
2.4. Methods for finding solutions to an intellectual problem in the case of reducing problems to a set of problems 17
2.5. List of control questions on the topic № 2 20
3. Representation of knowledge in artificial intelligence system 21
3.1. Knowledge and models of knowledge representation in artificial intelligence systems 21
3.2. Production models of knowledge representation 28
3.3. Solution management in productive systems 30
3.4. List of control questions on the topic № 3 33
4. Knowledge-based problem solvers 34
4.1. Semantic grids: basic concepts, types, methods of description and logical derivation of semantic grids 34
4.2. Frames: basic concepts, frame structure, frame models 35
4.3. Expert systems: purpose and principles of construction, generalized architecture, classes of problems that are solved with the help of expert systems 37
4.4. Stages of development of expert systems, acquisition of knowledge 40
4.5. Scope of expert systems 41
4.6. Search and explanation of solutions in expert systems 41
4.7. Knowledge engineering 42
4.8. List of control questions on the topic № 4 43

5. Modern trends and approaches to the creation of artificial intelligence systems 43
5.1. Modern software and tools for creating artificial intelligence systems: Visual Prolog, Allegro Common Lips, CLIPS, JESS 43
5.2. An ontological approach to the presentation and interpretation of knowledge in distributed information environments such as the Internet 51
5.3. List of control questions on the topic № 5 62
6. Tasks for practical and independent work 62
6.1. Task № 1. Search for solutions to an intellectual problem in space. Methods of blind search and heuristic search 62
6.2. List of control questions on the task № 1 80
6.3. Task № 2. Construction of autoregressive models in Eviews 80
6.4. List of control questions on the task № 2 93
6.5. Task № 3. Determining the best autoregressive model and building a forecast based on the selected model 93
6.6. List of control questions on the task № 3 97
6.7. Task № 4 Topic: Starting, getting started and basics of Visual Prolog 97
6.8. List of control questions on the task № 4 116
6.9. Task № 5. Unification and search with return 117
6.10. List of control questions on the task № 5 129
6.11. Task №6. Cycle and recursion 129
6.12. List of control questions on the task № 6 145
6.13. Task № 7. Lists and recursion 145
6.14. List of control questions on the task № 7 162
6.15. Task № 8. Internal database of facts 162
6.16. List of control questions on the task № 8 182
Recommended books 183
Additions 184
Additions A. Determining relationships based on facts 184
Additions B. Perceptroni 190
Addition C. Method of group consideration of arguments 207

Introduction

The term intelligence comes from the Latin intellectus – meaning mind; human mental abilities. Accordingly, artificial intelligence (AI) is usually interpreted as the property of automatic systems to assume certain functions of human intelligence, for example, to choose and make optimal decisions based on previous experience and rational analysis of external influences.

But there is no exact definition for this science, because philosophy does not solve the question of the nature and status of human intelligence. There is no exact criterion for the computer to achieve "reasonableness", although a number of hypotheses have been proposed before artificial intelligence, such as the Turing Test or the Newell-Simon hypothesis. Today, there are many approaches to both understanding AI tasks and creating intelligent systems.

Historically, there have been three main directions in AI modeling.

In the first approach, the object of research is the structure and mechanisms of the human brain, and the ultimate goal is to reveal the secrets of thinking. Necessary stages of research in this direction are the construction of models based on psychophysiological data, conducting experiments with them, making new hypotheses about the mechanisms of intellectual activity, improving models, etc.

The second approach as an object of study considers AI. Here we are talking about modeling intellectual activity with the help of computers. The purpose of work in this direction is to create algorithmic and software for computers, which allows you to solve intellectual problems no worse than humans.

The third approach focuses on the creation of mixed human-machine, or, as they say, interactive intelligent systems, a symbiosis of the possibilities of natural and artificial intelligence. The most important problems in these studies are the optimal distribution of functions between natural and artificial intelligence and the organization of dialogue between man and machine.

Artificial intelligence is a very young field of research, founded in 1956. Its historical path resembles a sine wave, each "take-off" of which was initiated by some new idea. Today, its development is in decline, giving way to the application of already achieved results in other areas of science, industry, business and even everyday life.

1. Basic concepts and definitions

1.1. The concept of artificial intelligence. History of development. Intellectual task

Artificial intelligence is a relatively young scientific field that lies at the intersection of various disciplines, such as discrete mathematics, linguistics, psychology, programming, mathematical statistics, and others.

The term intelligence comes from the Latin intellectus – meaning mind; human mental abilities.

Accordingly, artificial intelligence (AI) is usually interpreted as the property of automatic systems to assume certain functions of human intelligence, for example, to choose and make optimal decisions based on previous experience and rational analysis of external influences.

Consider the term "intellectual task". In order to explain how an intellectual task differs from a simple task, it is necessary to introduce the term "algorithm" – one of the cornerstones of cybernetics.

An algorithm is understood as an exact instruction to perform in a certain order a sequence of operations to solve any problem from a given class of tasks.

The term "algorithm" comes from the name of the Uzbek mathematician Al-Khorezmi, who in the IX century proposed the simplest arithmetic algorithms.

Tasks related to finding an algorithm for solving a class of problems of a certain type will be called intelligent.

Therefore, it seems quite natural to exclude from the class of intellectual such tasks for which there are standard methods of solution.

Examples of such problems are purely computational problems: solving a system of linear algebraic equations, numerical integration of differential equations, etc. To solve this kind of problem, there are standard algorithms that represent a certain sequence of elementary operations that can be easily implemented in the form of a computer program.

In contrast, for a wide class of intellectual problems, such as pattern recognition, chess, proof of theorems, etc., on the contrary, this formal division of the solution search process into individual elementary steps is often quite difficult, even if their solution is simple.

Thus, we can paraphrase the definition of intelligence as a universal super-algorithm that can create algorithms for solving specific problems.

The activity of the brain that has intelligence, aimed at solving intellectual problems, we will call thinking, or intellectual activity.

Intelligence and thinking are organically linked to solving problems such as proof of theorems, logical analysis, situation recognition, behavior planning, games and control in conditions of uncertainty.

Characteristic features of intelligence that are manifested in the process of solving problems are the ability to learn, generalize, accumulate experience (knowledge and skills) and adapt to changing conditions in the process of solving problems.

Turing proposed a criterion for determining whether a computer has the ability to think. Its content is as follows. There are people and a car in different rooms. They cannot see each other, but have the opportunity to exchange information (for example, via e-mail). If in the process of dialogue between the participants of the game people fail to establish that one of the participants is a machine, then such a machine can be considered as having intelligence. (2029)

By the way, Turing proposed an interesting plan to imitate thinking: "Trying to imitate the intelligence of an adult," writes Turing, "we have to think a lot about the process by which the human brain reached its true state. Why should we instead of trying to create a program that mimics the intelligence of an adult, do not try to create a program that would mimic the intelligence of a child? After all, if a child's intelligence receives appropriate education, it becomes the intelligence of an adult ... programmed ... Thus, we divide our problem into two parts: the task of building a "child program" and the task of "educating" this program.

Looking ahead, we can say that this is the path used by almost all AI systems. After all, it is clear that it is almost impossible to put all the knowledge in a rather complex system. Thanks to these qualities of intelligence, the brain can solve a variety of problems, as well as easily rebuild from one task to another.

History of AI systems development. Artificial intelligence has a long history. Another Plato, Aristotle, Socrates, Descartes, Leibniz, J. Bull; and later N. Wiener sought to describe thinking as a set of some basic operations, rules and procedures.

As a scientific field, artificial intelligence emerged in the mid – 60's of XX century. Its origin is connected with the direction of automation of human intellectual activity.

There is no exact definition for this science, because philosophy does not address the question of the nature and status of human intelligence. There is no exact criterion for the computer to achieve "reasonableness.

AI is one of the areas of computer science, which aims to develop computer systems capable of performing functions traditionally considered intelligent: language comprehension, inference, use of accumulated knowledge, learning, action planning, etc.

Thus, AI is a young area of research. Its historical path resembles a sine wave, each "take-off" of which was initiated by some new idea. Today, its development is in decline, giving way to the application of the results already achieved in other areas of science, industry, business and even everyday life.

Historically, there have been three main approaches in AI modeling

In the first approach, the object of research is the structure and mechanisms of the human brain, and the ultimate goal is to reveal the secrets of thinking. Necessary stages of research in this direction are the construction of models based on psychophysiological data, conducting experiments with them, making new hypotheses about the mechanisms of intellectual activity, improving models, etc.

The second approach as an object of study considers AI. Here we are talking about modeling intellectual activity with the help of computers. The purpose of work in this direction is to create algorithmic and software for computers, which allows you to solve intellectual problems no worse than humans.

The third approach focuses on the creation of mixed human-machine, or, as they say, interactive intelligent systems, on the symbiosis (interaction and coexistence) of the possibilities of natural and artificial intelligence. The most important problems in these studies are the optimal distribution of functions between natural and artificial intelligence and the organization of dialogue between man and machine.

Thus, AI research began immediately after the advent of the first computers. For 50 years AI has passed in the development of 4 basic stages:

- late 50's – early 70's – research in the field of "general intelligence", attempts to model the general intellectual processes inherent in man: free dialogue, solving various problems, proving theorems, game theory, creating poems and music;

- 70s – 80s – research and development of approaches to the formal presentation of knowledge, attempts to reduce intellectual activity to formal transformations.

- 80s – 90s – development of systems in subject areas that have applied practical value (expert_systems);

- after the 90's – frontal work on the creation of computers of the 5th generation, which would have_intellectual_possibilities.

Stage 1 – the establishment of AI – the period from 1943 to 1956. Work in the field of AI began with the emergence of neurocybernetics. Neurocybernetics is a scientific field that studies the basic laws of organization and functioning of neurons and neuronal formations. The main method of neurocybernetics is mathematical modeling, and the data of a physiological experiment are used as a source material for creating models. Neurocomputers, which, in a broad sense, were perceived as prototypes of the "artificial brain" – an intelligent system that must be built and function similarly to the human brain. The prefix "neuro" emphasized the difference between such a system and a traditional computer and its functional proximity to the brain. In 1943, American scientists McCall and Pitts proposed a model of a formal logic neuron that could be in two stable states. (Neuron (from the ancient Greek νεῦρον –

fiber, nerve) – an electrically excitable cell that processes and transmits information in the form of an electrical or chemical signal).

The period from 1956 to 1969 can be called a period of heuristic search and proof of theorems (heuristic algorithm, or just heuristics – no one guarantees that the solution is accurate. So such algorithms are approximate and inaccurate).

In 1956, Simon created the program "Logic Theorist", which is the first program to prove theorems (this is the realization of the idea that you can make a program that would solve the problem like a man). This program finds a proof of the theorems of mathematical logic. This was the first attempt to penetrate the complex process of thinking through research in the field of artificial intelligence. However, the effectiveness of this program was low. A further solution to this program was the GPS program (general problem solver). The solution of problems in this program was carried out on the basis of search algorithms in the space of possible solutions according to heuristic rules, which directed the search to the desired goal. It was the first program to use human reasoning.

In the early 60's intensive research in the field of neural networks. In 1962, Rosenblatt created the perceptron, a device for recognizing images, including letters of the alphabet. However, its capabilities were limited.

Representation of knowledge (1969 - 1979). By the end of the 1960s, it was found that general knowledge (general strategies for finding solutions) was not enough to solve practically important problems. Successful solution of applied problems is possible only with well-structured special knowledge. This was first implemented in the DENDRAL program (Feigenbaum, Bukhenen, Lederberg), which was used to determine the molecular structure of unknown organic compounds according to data obtained using a mass spectrometer. This program used the empirical knowledge of chemists for the first time, such programs that accumulate the knowledge of experts were called "expert systems"

1.2.The concept of intelligent system

Intelligent system or Intelligent information system is a type of automated information systems, sometimes IIS is called a knowledge-based system. IIS is a set of software, linguistic and logical-mathematical tools for the implementation of the main task: to support human activities and search for information in the mode of extended dialogue in natural language. Tasks solved by the following systems:

1) interpretation of data. This is one of the traditional tasks for expert systems. Interpretation means the process of determining the content of data, the results of which must be consistent and correct. Multivariate data analysis is usually provided;

2) diagnosis. Diagnosis is the process of correlating an object with a certain class of objects and detecting a fault in some system. A fault is a deviation from the norm. This interpretation allows us to consider from a single theoretical standpoint and the failure of equipment in technical systems, and diseases of living organisms, and all sorts of natural anomalies. An important specificity here is the need to understand the functional structure ("anatomy") of the diagnostic system;

3) monitoring. The main task of monitoring is the continuous interpretation of data in real time and signaling the output of certain parameters beyond acceptable limits. The main problems are the "omission" of the alarm situation and the inverse task of "false" operation. The complexity of these problems is in the blurring of the symptoms of anxiety and the need to take into account the temporary context;

4) design. Design is to prepare specifications for Creation of "objects" with predefined properties. The specification means the whole set of necessary documents - drawings, explanatory notes, etc. The main problems here - obtaining a clear structural description of knowledge about the object and the problem of "trace". only the design decisions themselves, but also the reasons for their adoption. Thus, in the design tasks are closely linked two main processes performed within the relevant EC: the decision-making process and the process of explanation;

5) forecasting. Forecasting allows you to predict the consequences of certain events or phenomena based on the analysis of available data. Forecasting systems logically derive probable consequences from given situations. The forecasting system usually uses a parametric dynamic model in which the values of the parameters are "adjusted" to a given situation. The conclusions drawn from this model form the basis for predictions with probable estimates;

6) planning. Planning means finding action plans related to objects capable of performing certain functions. Such ECs use models of behavior of real objects in order to logically deduce the consequences of planned activities;

7) training. Learning means using a computer to teach a discipline or subject. Teaching systems diagnose errors in the study of any discipline using a computer and suggest the right solutions. They accumulate knowledge about the hypothetical "student" and his characteristic mistakes, then in the work they are able to diagnose weaknesses in the knowledge of students and find appropriate means to eliminate them. In addition, they plan an act of communication with the student depending on the student's progress in order to transfer knowledge;

8) management. Management is a function of an organized system that supports a certain mode of operation. This type of EC controls the behavior of complex systems in accordance with specified specifications;

9) decision support. Decision support is a set of procedures that provides the decision maker with the necessary information and recommendations to facilitate the decision-making process. These ECs help professionals to choose

and form the right alternative among the many choices in making responsible decisions.

In the general case, all knowledge-based systems can be divided into systems that solve the problem of analysis, and systems that solve the problem of synthesis. The main difference between analysis tasks and synthesis tasks is that if in analysis tasks many solutions can be listed and included in the system, then in synthesis tasks many solutions are not potentially limited and are built from solutions, components or problems. The tasks of the analysis are: data interpretation, diagnostics, decision support; the tasks of synthesis include design, planning, management. Combined: teaching, monitoring, forecasting.

1.3. List of control questions on the topic № 1

1. Explain the term Intelligence.
2. What is artificial intelligence.
3. Name and describe the directions of AI modeling.
4. Explain the term intellectual task.
5. Examples of intellectual tasks.
6. Name the characteristics of intelligence.
7. Describe the Turing test.
8. Briefly describe the history of AI systems.
9. What is an intelligent system.
10. Describe the problems solved by intelligent systems.

2. Ways of presenting an intellectual problem and methods of finding solutions

2.1. Ways of presenting an intellectual problem, their advantages and disadvantages

All intellectual tasks can be divided into two classes: cognition and transformation. In the tasks of cognition, the goal is to determine the characteristics of objects, situations, processes. Processes of solving cognitive problems are traditionally processes of research, analysis, verification, search. In the tasks of transformation, the goal is to create new objects, situations, processes. Processes for solving transformation problems, as a rule, are processes of synthesis, design, construction. In practice, the elements of cognition and transformation are closely interconnected and intertwined.

The main types of actions performed in solving intellectual problems are: transformation of connections of objects;

- creation of new objects;
- selection of an object that meets the specified requirements;
- calculation of parameter values;

- search for information that satisfies certain conditions;
- withdrawal; search for specified elements in the object model; concept formation.

The disadvantages of intellectual problems are the incompleteness, inaccuracy and inconsistency of knowledge, as well as the large dimension of the solution space, which does not allow to solve them by simple search. In such problems, there are often no clear criteria for choosing the optimal solution, and the problem itself is not always fully formalized. An example of an intellectual task is the recognition of images, ie determining the affiliation of the observed object to one of the predefined categories. The main properties of intellectual tasks are:

- symbolic representation of the conditions of the problem;
- lack of clear statement of the problem;
- lack of a solution acceptable for practical use, which always provides the correct result;
- incompleteness, inaccuracy and inconsistency of knowledge about;
- lack of clear unambiguous criteria for choosing the optimal solution;
- large dimension of decision space.

Intellectual activity is the actions of people that lead to the desired result in situations where there is no algorithm to solve the problem. In other words, it is the process of obtaining the desired result in intellectual tasks.

The advantage is that a person has a certain set of knowledge about the world around him, which allows him to navigate in different situations and make the right decisions. In addition, a person is able to use this knowledge.

2.2. Search for solutions to an intellectual problem in space. Methods of blind search and heuristic search

When planning in the task space, space is formed as a result of the introduction of a set of tasks of the type: "part – whole", "task – subtask", "general case – special case", etc.

In other words, the task space reflects the decomposition of tasks into subtasks (goals on subgoals). The problem is to find the decomposition of the original problem into subtasks, which leads to problems whose solution the system knows. For example, the IC knows how to calculate the values of sin x and cos x for any value of the argument and how to perform the division operation. If it is necessary to calculate tg x, then the solution of the PR-problem will be to present this problem in the form of decomposition

$$tgx = sin\ x / cos\ x \text{ (except } x = \pi / 2 + k\,\pi).$$

Representation of problems in the state space involves the task of a number of descriptions: states, sets of operators and their effects on the

transitions between states, target states. State descriptions can be character strings, vectors, two-dimensional arrays, trees, lists, etc. Operators translate one state into another. Sometimes they are represented as products A => B, which means that state A will turn into state B.

The space of states can be represented as a graph, the vertices of which are denoted by states, and the arcs - by operators.

Thus, the problem of finding a solution to the problem <A, B> when planning by states is presented as a search problem on the graph of the path from A to B. Usually the graphs are not set, but generated as needed.

There are blind and directed methods of finding a way.

The idea of the blind method is very simple and obvious. A point is taken randomly in the admissible domain, and the value of the criterion in it is compared with the current best. If a new random point is worse than the one that is stored as the current best, then take another point. If you find a point where the criterion is better, it is remembered as the current best. It is guaranteed that with an unlimited increase in the number of attempts we approach the global optimum, ie. the current best value found will be so close to the exact solution.

The blind method has two types: search deep and search wide. When searching in depth, each alternative is explored to the end, without considering the other alternatives. The method is bad for "tall" trees, as you can easily slip past the right branch and spend a lot of effort exploring "empty" alternatives.

When searching broadly at a fixed level, all alternatives are explored and only then is the transition to the next level.

The method may be worse than the depth search method if in the graph all the paths leading to the target vertex are located at approximately the same depth.

Both blind methods are time consuming and therefore targeted search methods are needed

The method of branches and boundaries. From the formed in the process of finding unfinished paths is selected the shortest and continues by one step. The obtained new unfinished paths (there are as many of them as there are branches in this vertex) are considered next to the old ones, and the shortest of them continues again by one step. The process is repeated until the first achievement of the target vertex, the decision is remembered. Then from the remaining unfinished paths are excluded longer than the completed path, or equal to it, and the remaining continue according to the same algorithm as long as their length is less than the completed path. As a result, either all unfinished paths are excluded, or a complete path is formed among them, shorter than previously obtained. The latter path begins to play the role of a standard, etc.

Moore's shortest path algorithm. The original vertex X_0 is denoted by the number 0. Suppose that during the operation of the algorithm at the current step we obtain a set of child vertices $X(x_i)$ of the vertex x_i. Then all previously obtained vertices are deleted from it, the remaining ones are denoted by a label

increased by one in comparison with the label of the vertex x_i, and pointers to X_i are drawn from them. Next, on the set of marked vertices, not yet appearing as pointer addresses, the vertex with the smallest label is selected and child vertices are constructed for it. Vertex markup is repeated until the target vertex is obtained.

Dijkstra's algorithm for determining paths with a minimum cost is a generalization of Moore's algorithm by introducing arcs of variable length.

Dora and Mickey search algorithm with low cost. Used when the cost of the search is high compared to the cost of the optimal solution. In this case, instead of selecting the vertices that are least distant from the beginning, as in Moore's and Dijkstra's algorithms, a vertex is chosen for which the heuristic estimate of the distance to the target is the smallest. With a good score, you can quickly get a solution, but there is no guarantee that the path will be minimal.

Hart, Nilsson and Raphael algorithm. The algorithm combines both criteria: the cost of the path to the vertex $g[x)$ and the cost of the path from the vertex $h(x)$ – in the additive estimation function $f\{x\} = g(x\}- h(x)$. Provided $h(x) < hp(x)$, where $hp(x)$ is the real distance to the target, the algorithm guarantees finding the optimal path. The path search algorithms on the graph also differ in the direction of search.

Direct search starts from the initial state and is usually used when the target state is specified implicitly. The inverse search starts from the target state and is used when the initial state is set implicitly and the target state is explicit. Bidirectional search requires a satisfactory solution to two problems: changing the direction of the search and optimizing the "meeting point". One of the criteria for solving the first problem is to compare the "width" of the search in both directions – choose the direction that narrows the search. The second problem is caused by the fact that the forward and reverse paths can diverge and the more the search, the more likely it is.

2.3. Heuristic search methods

The word "heuristics" is interpreted as a method of search, inventions. The foundations of heuristic methods were laid in the philosophical concept of Socrates. However, only in the XX century. this concept has received wide scientific and practical recognition as heuristic thinking, heuristic techniques and methods, heuristic properties. Although today there is no single unambiguous interpretation of heuristics as such, but it is widely associated with creativity, creative search, intelligence, foresight.

Heuristic methods are special methods of analysis based on the use of experience, intuition of the specialist and his creative thinking. Heuristic methods are divided into expert and psychological.

Expert methods are a set of logical and mathematical techniques and research procedures, as a result of which experts receive the information

necessary to make informed rational management decisions. Psychological methods – a set of rules and procedures that provide solutions to problems and solve creative problems.

In addition, all heuristic methods are divided into two groups – methods of undirected search and directional search. The group of methods of undirected search includes methods: brainstorming, expert assessments, associations and analogies, control questions, collective notebook, business games and situations, cyber meetings, etc.

The morphological method, algorithm for solving inventive problems, etc. belong to the group of directed search.

Brainstorming method.

The method of brainstorming is the most common method of generating new ideas as a result of creative collaboration of a group of specialists.

There are certain rules for organizing and conducting brainstorming:

- the definition of the head of the group goal in the form of a problem or task that needs to be solved (increase the profitability of production of a particular product or turn the production of a product from unprofitable to profitable). The task of the leader is to activate the creative thinking of the group members in order to produce as many ideas, proposals for the task;

- strict adherence to the distribution of time as a whole and by stages:

Stage I – the nomination of new ideas,

Stage II – discussion and evaluation of these ideas. In this way it is differentiated in the time of nomination and discussion of ideas. The meeting may not last more than an hour and a half. If necessary, you can hold several meetings on one issue;

- compliance with the established procedure for organizing the creative process. Yes, at the first stage only ideas are put forward, even, at first sight, inappropriate, unrealized, fantastic.

Their quantity is preferred to quality. It is forbidden to criticize ideas so as not to disrupt the creative process. At the II stage there is an active discussion of the put forward offers with the indication of their advantages and lacks. It is useful to combine several ideas. The conclusion of this discussion is the choice of the best of the proposed options.

Experience shows that due to the separation of time generation and discussion of ideas, the number of new ideas is twice as large as when using other traditional methods.

For the most part, the process of brainstorming goes through 5 successive stages: problem identification, generation of ideas, analysis of ideas, search for opportunities for their implementation and completion.

1. Defining the problem. It is expedient to allocate this stage in that case when from the very beginning the problem which needs the decision is not outlined. In this case (especially with a small number of participants in the game), you can study the importance of problems in the rating with the help of

cards, which are distributed to each participant in the game to identify his opinion.

2. Generation of ideas is the most important stage of work, because the quality of the ideas put forward depends on the effectiveness of brainstorming. To do this, game participants can be divided into groups of 5-6 people in each and on separate cards to write down new ideas - one idea on each card. It is necessary to put forward as many ideas as possible and record them. At this stage, criticism of the ideas put forward is not allowed. The duration of the stage of generating ideas is 30 minutes. The groups then inform the facilitator of the number of ideas put forward.

3. Analysis of ideas is an in-depth study, discussion, even critical consideration of the proposals in order to identify a rational grain in each of them. Ideas cannot be rejected. At this stage, too, is given 30 minutes.

4. The search for opportunities for implementation is done by evaluating each idea from two positions – originality and feasibility. 20 minutes are allocated for this purpose.

5. Completion. At this stage, representatives of all groups make reports on ideas that are recognized as a result of the analysis are very successful, original, and which can be really implemented.

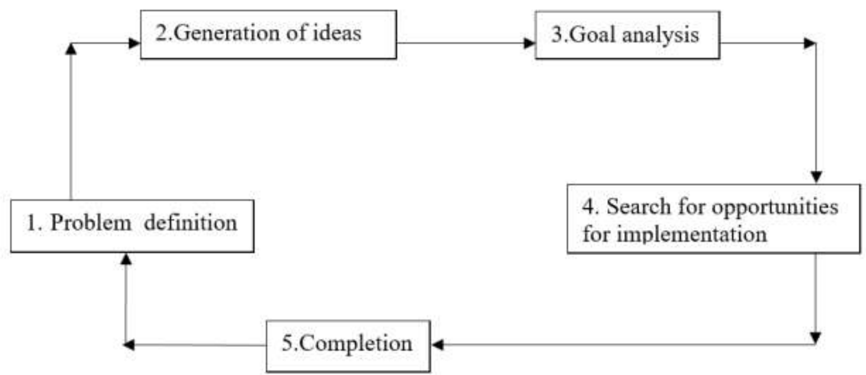


Fig. 2.1. The main stages of brainstorming

The method of brainstorming is best used to solve simple, mostly general, or organizational plan tasks, about which there is enough information, and with which the participants are familiar.

An example of solving such problems is a problem called "Smoking time". The fact is that one of the Japanese companies has a problem of reducing the productivity of turners in the shop due to their frequent distraction during the work shift for smoking breaks. The management could not make an acceptable

decision because it was not possible to place a controller or a video camera near each machine.

Therefore, a brainstorming was carried out with the involvement of managers of different levels of management. In its process, a simple solution was found that did not require large financial or material investments, and took into account the peculiarities of human psychology. So, experts came to the decision to transfer a place for smoking ("smoking room") from a distant room to the center of shop. It was installed on a high pedestal and made of glass. As a result, unproductive losses of working time were sharply reduced and the productivity of turners was increased.

Other heuristic methods

The method of control questions is designed to activate the creative process to solve the problem by providing answers to questions according to a pre-compiled list. This contributes to a comprehensive consideration of the problem and the search for new non-traditional approaches to its solution. The universality of this method is that the analyst can ask questions to himself and look for answers to them, as well as in the process of collective discussion, in particular during brainstorming, conference of ideas and so on.

The method of morphological analysis has several varieties and is designed to generate new unbiased ideas about the possibility and ways to solve the problem. Using this method, the analytical task is described and analyzed as a set of all possible morphological (ie structural) connections and relationships between the constituent elements.

2.4. Methods for finding solutions to an intellectual problem in the case of reducing problems to a set of problems

There are direct, reverse and bidirectional search methods. Direct search starts from the initial state and is usually used when the target state is specified implicitly. The inverse search starts from the target state and is used when the initial state is set implicitly and the target state is explicit. Bidirectional search requires a satisfactory solution to two problems: changing the direction of the search and optimizing the "meeting point". One of the criteria for solving the first problem is to compare the "width" of the search in both directions, the direction that narrows the search is selected. The second problem is caused by the fact that the forward and reverse paths can diverge and the longer the search, the more likely it is.

Task planning. This method leads to good results because often the solution of problems has a hierarchical structure. However, it is not necessary to require that the main task and all its subtasks be solved by the same methods. Reduction is useful for presenting the global aspects of the task, and in solving more specific tasks, the preferred method of planning by state. The method of

planning by states can be considered as a special case of the method of planning by means of reductions, because each application of the operator in the space of states means reduction of the initial problem to two simpler ones, one of which is elementary. In the general case, the reduction of the original problem is not reduced to the formation of such two subtasks, at least one of which was elementary.

The search for planning in the task space is to sequentially reduce the original task to easier and simpler until only elementary tasks are obtained. A partially ordered set of such tasks will be the solution of the original problem. It is convenient to divide the task into alternative sets of subtasks in the form of an AND/OR graph. In such a graph, each vertex, except the final one, has either conjunctively connected child vertices (I-vertex) or disjunctively connected (OR-vertex).

In a separate case, in the absence of I-vertices, there is a graph of state space. The final vertices are either final (they correspond to elementary tasks) or dead ends. The initial vertex (the root of the I/OR graph) represents the original task. The purpose of searching on an I/OR graph is to show that the initial vertex is debugged. Solvable are the final vertices (I-vertices), in which all child vertices are solved, and OR vertices, in which at least one child vertex can be fixed. The permitting graph consists of resolved vertices and indicates the method of solving the initial vertex. The presence of dead-end vertices leads to unsolvable vertices. Insoluble are dead-end vertices, I-vertices in which at least one child vertex is unsolvable, and OR vertices in which each child vertex is unsolvable.

Cheng and Slagel algorithm. Based on the transformation of an arbitrary I/OR-graph into a special OR-graph, each OR-branch of which has I-vertices only at the end. The transformation uses the representation of an arbitrary AND/OR graph as an arbitrary formula of the logic of statements with the subsequent transformation of this arbitrary formula into a disjunctive normal form. Such a transformation allows us to further use the Hart, Nilsson and Raphael algorithm.

The search for planning in the task space is to sequentially reduce the original task to easier and simpler until only elementary tasks are obtained. A partially ordered set of such tasks will be the solution of the original problem. It is convenient to divide the task into alternative sets of subtasks in the form of an AND/OR graph. In such a graph, each vertex, except the final one, has either conjunctively connected child vertices (I-vertex) or disjunctively connected (OR-vertex).

Method of key operators. Suppose that the problem <A, B> is given and it is known that the operator f must be included in the solution of this problem. Such an operator is called a key. Suppose that the state C is required for the application of f, and the result of its application is f (c). Then the I-vertex generates three child vertices:, $<$ C, $f\,(c\}$ $>$ i, of which the mean is an elementary

problem. Key operators are also selected for problems *C* > and *B* >, and the specified reduction procedure is repeated as long as it is As a result, the initial problem B> is divided into an ordered set of subtasks, each of which is solved by the method of planning in the state space.

Alternatives to the choice of key operators are possible, so that in the general case there will be an AND/OR graph. In most tasks it is possible not to select the key operator, but only to specify the set that contains it. In this case, for the problem < *A*, *B* > the difference between A and B is calculated, which corresponds to the operator, which eliminates this difference. The latter is key.

Method of planning a common problem solver (PCP). PCP was the first most famous model of the planner. It was used to solve problems of integral calculus, inference, grammar, etc. PCP combines two basic principles of search: analysis of goals and means and recursive problem solving. In each search cycle, the ARI solves three types of standard problems in a rigid sequence: convert object A to object B, reduce the difference D between A and B, apply the operator f to object A. Solving the first problem determines the difference D, the second – suitable operator f, the third – the required condition for the use of C.

If C does not differ from A, then the operator f is used, otherwise C is represented as another goal and the cycle is repeated, starting with the task of "convert A to C". In general, the PCP strategy performs a reverse search – from a given goal B to the necessary means of achieving it C, using the reduction of the original task to tasks and < C, B >.

Note that the PCP tacitly assumes the independence of differences from each other, hence the guarantee that the reduction of some differences will not lead to an increase in others.

Planning with logical inference. Such planning involves: a description of states in the form of correctly constructed formulas of (CCF) some logical calculus, a description of operators in the form of either CCF, or the rules of translation of some PPF into others. Representation of operators in the form of CCF allows to create deductive methods of planning, representation of operators in the form of translation rules – methods of planning with elements of deductive inference.

Deductive method of planning the QA3 systemd id not live up to expectations, which he hoped for mainly due to unsatisfactory presentation of tasks. An attempt to rectify the situation led to the creation of a question-and-answer QA3 system. The system is designed for any subject area and is able to answer the question by logical inference: is it possible to achieve state B with A? The principle of resolutions is used as a method of automatic output. To guide the logical conclusion, QA3 uses various strategies, mainly syntactic in nature, taking into account the peculiarities of the formalism of the principle of resolutions. Operation of QA3 has shown that the output in such a system is slow, detailed, which is not typical of human reasoning.

STRIPS system product method. In this method, the operator represents the product P, A => B, where P, A and B-sets of CCF calculations of first-order predicates, P expresses the conditions of application of the product core A => B, where B contains a list of added CCF and a list of excluded CCF, that is postum. The method repeats the PCP method with the difference that the standard tasks of determining the differences and the application of the respective operators are solved on the basis of the principle of resolutions. The appropriate operator is selected in the same way as in the PCP, based on the principle of "analysis of means and objectives". The existence of a combined method of planning allowed to limit the process of logical inference to the description of the state of space, and the process of generating new such descriptions to leave the heuristic "from goal to means to achieve it."

Product method using macro operators. Macro-operators are generalized solutions to problems obtained by the STRIPS method. The use of macro operators allows to reduce the search for a solution, but there is a problem of simplification of the applied macro operator, the essence of which is to select a given difference of its required part and exclude from the latter unnecessary operators.

Method of hierarchical system of ABSTRIPS solver products. In this method, the search space is broken down at the hierarchy level by detailing the products used in the STRIPS method. To do this, each letter of the CCF, which is included in the set P of the conditions of use of products, is assigned a weight j, j = 0, k, and at the i-th level of planning, carried out by the STRIPS system, only letters of weight j are taken into account. Thus, at the k-th level of products are described in the least detail, at zero-the most detailed as in the method of the STRIPS system. Such a partition allows for j-th level planning to use the (j +1) -th solution as the skeleton of the j-th solution, which increases the search efficiency as a whole.

Improved method of planning Newell and Simon. The method is based on the following idea of further improvement of the PCP method: the problem is solved first in a simplified (by ranking differences) planning area, and then an attempt is made to clarify the solution to a more detailed, original problem area.

2.5. List of control questions on the topic № 2

1. Name the classes of intellectual problems.
2. Actions in solving intellectual problems.
3. Disadvantages of intellectual tasks.
4. Properties of intellectual problems.
5. What is intellectual activity.
6. In what form can we represent the space of states.
7. What methods do you know of finding a way.
8. The idea of the blind path method.

9. Types of blind search.
10. Describe the method of branches and boundaries.
11. What is the algorithm for the shortest paths of Moore.
12. What is the Dijkstra algorithm used for?
13. When the Dora and Mickey algorithm is used.
14. What is associated with heuristics.
15. What are the heuristic methods based on.
16. Rules for organizing and conducting the method of brainstorming.
17. Stages of the brainstorming process.
18. Describe the stage of brainstorming to determine the problem.
19. Describe the stage of brainstorming generation of ideas.
20. Describe the stage of brainstorming analysis of ideas.
21. Describe the stage of brainstorming the search for opportunities.
22. Describe the stage of brainstorming to determine completion.
23. For which tasks it is better to use the method of brainstorming.
24. What heuristic methods do you know?
25. What methods do you know for finding solutions to an intellectual problem?

3. Representation of knowledge in artificial intelligence system

3.1. Knowledge and models of knowledge representation in artificial intelligence systems

Knowledge – a set of data about the world, including information about the properties of objects, patterns of processes and phenomena, as well as the rules of using this information for decision making. The rules of use include a system of causation. The main difference between knowledge and data is their activity, ie the emergence of new facts in the database or the establishment of new connections can be a source of change in decision-making.

As a separate area of research, knowledge representation has been developing since the mid-1960s.

Knowledge is a set of facts, patterns, attitudes and heuristic rules that reflect the level of awareness of the problems of a particular subject area.

Knowledge can be:

- declarative;
- procedural.

Declarative knowledge contains only an idea of the structure of certain concepts. This knowledge is close to the data, the facts. For example: a higher education institution is a set of faculties, and each faculty in turn is a set of departments.

Procedural knowledge has an active nature. They determine the idea of the means and ways of obtaining new knowledge, testing knowledge. These are

algorithms of various kinds. For example: the method of brainstorming to find new ideas.

Types of knowledge:

- deep knowledge – the result of generalization of primary concepts into abstract structures;
- soft knowledge – allow multiple vague solutions (for example, making recommendations); a set of deep and soft knowledge makes it possible to create powerful knowledge bases;
- superficial knowledge – a set of empirical associations and relationships between the concepts of the subject area for standard situations;
- conceptual knowledge – express the properties of objects, processes and situations through the concepts (basic elements) of the subject area. The description of each concept includes a description of its components, an indication of the relationship with other components, the relationship between the concepts. Conceptual knowledge is rigid. Used in solving analysis problems;
- expert knowledge – the knowledge of specialists in the field, they accumulate experience. This type of knowledge plays the most important role in poorly structured subject areas. They are soft and superficial. The joint use of conceptual and expert knowledge allows to combine logical and associative judgments, to solve the most difficult problems at low computational costs;
- syntactic knowledge – characterize the syntactic structure of the object, which does not depend on the content of the concepts used;
- semantic knowledge – contain information related to the meaning of the analyzed objects;
- pragmatic knowledge – describe objects in relation to the goals of the tasks.

Forms of existence of knowledge:

- in the memory of a person (expert);
- materialized (canonized) – textbooks, monographs, etc.;
- semi-formalized structured model (field of knowledge);
- formalized knowledge of the language of presentation.

Models of knowledge representation

The problem of knowledge representation is central to knowledge-based systems, and in particular to expert systems, because its successful solution depends on the implementation of their main function – the acquisition of new knowledge. It is on this basis that the structure and form of organization of models and methods of knowledge representation are determined, which have a decisive influence on the efficiency of the system, perception of external information, dialogue with the user. As a separate area of research, the presentation of knowledge has been developing since the mid-1960s.

Basic (classical) models of knowledge representation:

- logical;
- algorithmic;
- frame;
- production rules;
- network models;

The basis of logical models of knowledge is the concept of a formal system, an example of which is the calculus of predicates.

In contrast to logical, heuristic models of knowledge use a set of different tools that convey the specific features of the model. Due to this, heuristic models are superior to logical ones in terms of the possibilities of adequate reflection of the subject area and in terms of the efficiency of the rules of logical inference. Heuristic models used in expert systems include production, network and frame systems.

To present knowledge in a computer means to define some initial objects, rules of formation on their basis of new objects and as a result to receive the description of knowledge. The formal way of description is a model of knowledge representation.

Data values act as input indivisible objects. The relationship between the data determines the rules of formation of new objects. Performing separate procedures on the relationship between data, structure data and form knowledge. There are a number of models of knowledge representation: logical, algorithmic, frame, semantic, productive. Consider the main features of each of the models.

Logical models of knowledge representation use the statements of some formal system to describe the problem to be solved. The purpose of the problem is also formulated in the form of a statement, the justice of which must be established or refuted, based on the axioms and rules of derivation of the formal system.

According to the rules established in the formal system, the final statement – the theorem derived from the initial system of statements (axioms, parcels), is assigned a true value, if each parcel or axiom is also assigned a true value.

The set of basic elements of logical models of knowledge representation are logical connections, quantifiers, constants, variables, functional and predicate symbols; syntactic rules. The latter define the concept: term, atom, correctly constructed formula. Derivation rules make it possible to draw some conclusions from existing axioms.

Constants are usually used to denote objects in a subject area, and predicate names are chosen to denote object classes, properties, and relationships between objects. Formulas containing quantifiers and variables describe the general patterns of the subject area.

Logical model of knowledge representation

The knowledge needed to solve, and the problem itself is described by certain statements in logical language. Knowledge is a set of axioms, and the problem to be solved is a theorem that requires proof. The process of proving the theorem is a logical model of knowledge representation. The description of the model is based on constructive logic. Let's set a logical model as a set:

$$M = \langle T, P, A, F \rangle,$$

where T is the set of basic elements, P is the set of rules, A is the set of real expressions (axioms), F is the rule of inference.

Consider in more detail what are the basic elements of T:

$T = T_1 \cup T_2 \cup T_3 \cup T_4 \cup T_5$,

where T1 are the names of tasks and subtasks, T1 = $\{I_1, I_2, ...,\}$; T_2 – determines the structure of their relationship, {ǽ, υ}; T_3 are symbols of reduction of tasks to subtasks, {→}; T4 – auxiliary characters, {(,)}; T_5 - a symbol of truth and falsity of the results of the decision, {t, f}.

Based on the symbols of the alphabet, the formulas of the logical model are built, ie a set of rules P, for example: The name of the problem and its description are given;

Denote the description of problems A, B. Then, if A ǽ B, it means that it is necessary to solve the problem with description A and description B.

2. If A υ B, then this is a description of the problem for which you should solve either a problem with a description of A or a problem with a description of B.

If the description of the problem is its name, then the task is called elementary. If the problem with the name And is reduced to the problem with the description And it is possible to write And → A. Thus elements with the description And are the description of the subtasks which are included in the problem with the name And.

3. Additional symbols 1, Ø mean the description of problems with the results of their solution. The symbol 1 means A = t; the symbol Ø means A = f.

Finding a solution to a problem based on a logical model of knowledge representation is based on the use of a number of axioms, for example:

A υ B = B υ A. The meaning of the axiom is that the solution of the problem of two subtasks A and B is determined. Accordingly, the problem will be solved if one of the subtasks is solved.

(A υ B) υ C = A υ (B υ C), ie, if there are three subtasks A, B, C, the original problem will be solved if one of the subtasks is solved, and any two subtasks can be combined into one subtask.

(Aæ B) æ C = A æ (B æ C). With such an axiom notation, it is assumed that the initial problem includes three subtasks A, B, C. The solution of the problem can be obtained if subtasks A, B are solved. Since there is no symbolic sign between A and B, subtasks can be solved in any sequence.

A total of 10 axioms are used to build a logical model. The logical model corresponds to the graphical mappings in the form of a reduction graph and a graph of spatial states.

For the reduction graph, the vertices are the names of the subtasks, and the arcs indicate the relationship between them. The graph is built from top to bottom, in its final vertices are elementary subtasks that can be solved using a computer. The search for a solution to the original problem is displayed by a sequence of vertices of the graph.

In the state space graph, the vertices are the processes of solving elementary subtasks. This graph must indicate the path from the root vertex to one of the end, ie set the sequence of vertices.

A total of 10 axioms are used to build a logical model. The logical model corresponds to the graphical mappings in the form of a reduction graph and a graph of spatial states.

Algorithmic model of knowledge representation

Algol-like programming languages are often used in the process of formalizing knowledge. The formal system gives a description of the solution of the problem in the form of a calculation program. The basis of the formal system are: the alphabet of the language used, the rules of formation of expressions from the elements of the alphabet, axioms and inference rules.

The alphabet is determined by the set T = T1 UT2 UT3, where: T1 = {A1, A2, ... An} – names of subtasks. Sequence A is a description of the original problem; T2 = {;, case, of, while, do} – includes words that allow you to build syntactic constructions to describe the sequence of the solution (for example, case A of A1, A2, ... An - means that the description of the original problem A, to solve which is enough to solve one of the subtasks); T3 = {begin, end} – auxiliary values.

The algorithmic model can also be represented by a reduction graph, where the original problem is located in the root vertex, subtasks in the intermediate vertices, and elementary subtasks in the final vertices. Arcs reflect programming operations.

Semantic model of knowledge representation

This model allows you to operate with concepts expressed in natural language. An example of the implementation of such a model are expert systems. To build the model using the apparatus of semantic networks, presented in the form of a graph:

$G = \{Y1, Y2, \dots Yn; \beta 1, \beta 2, \dots \beta m\}$, where *Y* are the nodes (vertices) of the graph. They reflect some entities - objects, events, processes, phenomena, etc .; β – arcs of the graph, which denote the relationship between entities, given on the set of vertices.

The vertices reflect the essences of different degrees of commonality. Their ordering is based on the types of relationships.

The subject area is displayed as a set of entities and the relationship between them. If the fundamental concepts of relations and objects of the subject area are adequately formulated (ie there is a comprehensive conceptual knowledge), then the semantic model works very successfully.

Frame model of knowledge representation

This model is based on a person's perception of the world around him, on human psychology. When a person finds himself in a situation, he identifies it with some typical structure present in his memory. This structure is a frame – a declarative representation of a typical situation, supplemented by procedural information about the possibilities and ways to use it.

The frame is represented by a network. The upper levels of the network reflect the essence that is true for a typical situation (frame design). The lower levels end in empty structures – slots. Filling, definition of slots occurs when calling a frame in a specific situation in the subject area. The frame includes a set of slots:

$F = [(C1, d1), (C2, d2), ... (Cn, dn)]$, where F is the name of the frame; C – slot names; d is the value of the slots.

Slots are filled as knowledge of the subject area is obtained.

Production model

The production model contains a set of rules (products) in the form of:

1. IF the condition is an action
2. IF the cause is the consequence
3. IF the situation is a solution.

The essence of the model is that if certain rules of the condition are met, then some action must be taken. Production models can be implemented procedurally and declaratively. Procedural systems must have: a database, a set of production rules and an interpreter (it determines the sequence of activation of products). The database is a variable part of the model, and the rules and interpreter are constant. Only facts (knowledge) can be added and changed.

Production models are used in those subject areas where there is no clear logic and problems are solved on the basis of independent rules (heuristics). Product rules carry information about the sequence of purposeful actions. They reflect well the pragmatic component of knowledge and are used for small tasks.

Product systems

Under the production system is understood a certain method of organizing the computational process, in which the program of transformation of some information structure is set in the form of a set of rules-products.

Each rule is a combination of elements: the condition of suitability – action. The suitability condition specifies some requirements for the current state of the information structure, and the action contains a description of the operations to be performed if these requirements are met.

Production rules are the easiest way to present knowledge. It is based on the presentation of knowledge in the form of rules structured in accordance with the scheme "if – then". Part of the rule "if" is called a parcel (or condition of suitability), and part "then" – a conclusion.

The rule is written as follows: If a1, a2,. . . , an то b.

For example: **If** (1) y is the father of x (2) z is the brother of y **Then** z is the uncle of x

If there are no parcels, then knowledge consists only of a conclusion and is called facts.

In production systems, two main methods of inference are used: forward and reverse.

In direct derivation, the rules are studied one after another in a certain sequence. Based on the initial conditions (data) entered by the user, for each rule the truth or falsity of its condition of suitability is estimated. If the condition is true, the rule is activated, otherwise – no. The derivation procedure is iterative and may require several runs through the whole set of rules until a certain value of the target variable is determined.

The inverse derivation assumes the truth of the consequence (action) of a rule, after which it is necessary, moving a series of rules in the opposite direction, to prove that there are grounds for such a statement.

Advantages of production systems:

- universality of the programming method, which allows the creation of various application systems that differ in the ways of presenting rules and data structures;

- natural modularity of knowledge organization, when each product is a complete piece of knowledge about the subject area, and the set of products is naturally structured into subsets belonging to certain components of knowledge;

- independence of each product from the content of other products provides ease of formulation and modification;

- declarativeness of the production system, which provides a description of the subject area, rather than the relevant processing procedures.

Network models present knowledge in the form of a network, the vertices of which correspond to concepts (objects, events, processes, phenomena), and arcs – the relationships that exist between concepts.

The classification of network models is based on the conditions of the description of vertices and connections. If the vertices do not have their own internal structure, then the corresponding networks are called simple networks. If the vertices of the network themselves have some structure in the form of a network, then such networks are called networks of hierarchical type. The

relationship between the vertices can be the same; in this case, the networks are called homogeneous. If these relations have different meanings, then the network is called heterogeneous.

Depending on the nature of the relationship attributed to the arcs of the network, there are the following types of networks:

- functional networks in which two arc-connected vertices correspond: one to a function, the other to the argument of that function.
- scenarios – homogeneous networks that use a single type of relationship – a relationship of non-strict order. Most often, this relationship determines all possible sequences of events.
- semantic networks that use different types of relationships, and vertices can have different interpretations.

The main structural units from which the semantic network is built are frames.

Compared to other models, semantic networks have advantages:

- more efficient information retrieval, as associations between network objects determine access paths that run through the knowledge base;
- the ability to explicitly reflect the structures inherent in the knowledge of the subject area, for example, the relations "particle-whole", "element-set", "class-subclass", etc.

Frames. The frame is a system-structural description of the subject area (events, phenomena, situations, states, etc.), which consists of empty aspect (role) positions (slots) that correspond to the substantive features of the subject area and after filling with specific data transform the frame on the carrier of specific knowledge.

Translated from English, "frame" means "frame", "frame".

Different types of information can be associated with each frame. One part indicates how to use this frame, the other – what the consequences of its implementation may be, the third – what to do if these expectations are not confirmed.

3.2. Production models of knowledge representation

Production model of knowledge – a model based on rules, allows you to present knowledge in the form of proposals such as "If (condition), then (action)".

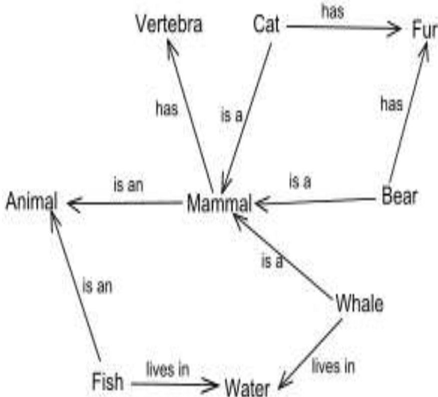


Fig. 3.1 Semantic network

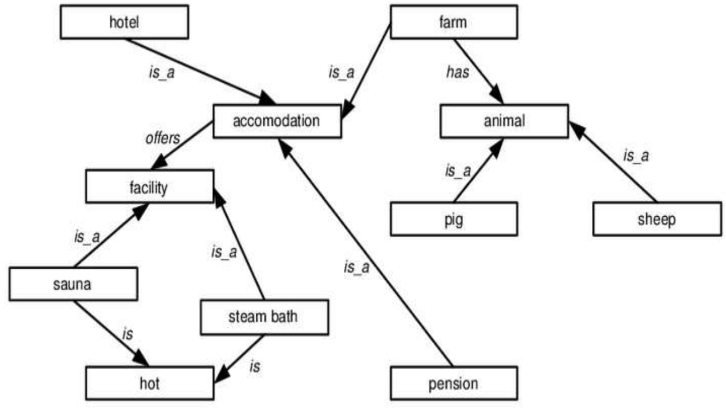


Fig. 3.2 A semantic network example of tourism-related terms

Production model – fragments of the Semantic Network, based on the temporal relationship between the states of objects.

Semantic network, ordered by the relations "whole - part", "genus – species"

Semantic network – an information model of the subject area, which has the form of an oriented graph, the vertices of which correspond to the objects of the subject area, and arcs (edges) define the relationship between them. Objects can be concepts, events, properties, processes.

Thus, the semantic network is one of the ways to present knowledge. The name combines terms from two sciences: semantics in linguistics studies the meaning of language units, and the network in mathematics is a kind of graph – a set of vertices connected by arcs (edges), which are assigned a number. In the semantic network, the role of vertices is performed by the concept of knowledge base, and arcs (and directed) define the relationship between them. Thus, the semantic network reflects the semantics of the subject area in the form of concepts and relationships.

The production model has the disadvantage that with the accumulation of a large number (about several hundred) products, they begin to contradict each other. In the General case, the production model can be represented as follows:

$$H = <I, C1, C2, ..., Cn, Q> ,$$

where I – a set of information elements stored in nodes networks, **C1, C2, ..., Cn** – types of connections between information elements, a mapping that matches multiple types connections and set information elements of the network.

Modifications of the production model

The production model is often supplemented by a certain order, which is introduced on a set of products, which simplifies the mechanism of logical inference. The order can be expressed in the fact that a separate next in order products can be used only after attempts to use previous products. Approximately similar impact on the production model can have the use of product priorities, which means that in the first place should be used products that have the highest priority.

The growing inconsistency of the production model can be limited by the introduction of exceptions and returns mechanisms. The exception mechanism means that special exclusion rules are introduced. They are distinguished by great specificity in comparison with the generalized rules. With an exception, the basic rule does not apply.

The mechanism of returns means that the logical conclusion can continue in the event that at some stage the conclusion led to a contradiction. You just need to abandon one of the previously accepted statements and return to the previous state.

Contradictions in knowledge bases in the Prolog language are detected automatically through the use of automatic proof of theorems with built-in Prolog search mechanisms with returns, organizing the search for information in knowledge bases and output of found information as information search results.

3.3. Solution management in productive systems

The production model attracts developers with its clarity, high modularity, ease of making additions and simplicity of the mechanism of logical inference.

Here are the strengths and weaknesses of product systems. Strengths of product systems:

- modularity;
- uniformity of structure (the main components of the production system can be used to build intelligent systems with different problem orientation);
- naturalness (conclusion of imprisonment in the production system in many ways similar to the process of reasoning of the expert);
- flexibility in the generic hierarchy of concepts, which is supported only as a link between rules (a change in a rule leads to a change in the hierarchy);
- ease of creating and understanding individual rules;
- ease of replenishment and modification;
- simplicity of the mechanism of logical conclusion.

Weaknesses of product systems:

- the derivation process is less efficient than in other systems, as most of the derivation time is spent on unproductive validation of the rules;
- it is difficult to imagine a hierarchy of concepts;
- ambiguity of mutual relations of rules;
- the difficulty of assessing the holistic image of knowledge;
- in contrast to the human structure of knowledge, lack of flexibility in the logical conclusion.

In the production representation, the area of knowledge is represented by a set of production rules If-Then, and the data are represented by a set of facts about the current situation.

The co-output mechanism matches each rule that is stored in the database with the facts contained in the database. When part of the rule If (condition) approaches the fact, the rule works and its part Then (action) is executed. A rule that works can change a set of facts by adding a new fact.

Mapping parts If rules with facts creates an output chain. The output chain shows how the EU applies the rules to obtain an opinion. To illustrate the output method based on a chain, consider a simple example.

Suppose the database initially includes facts A, B, C, D and E, and the database contains only 3 rules;

Rule 1. $Y \& D \rightarrow Z$
Rule 2. $X \& B \& E \rightarrow Y$
Rule 3. $A \rightarrow X$

The output circuit in Fig. 3.4. shows how the EU applies the rules to deduce the fact of Z.

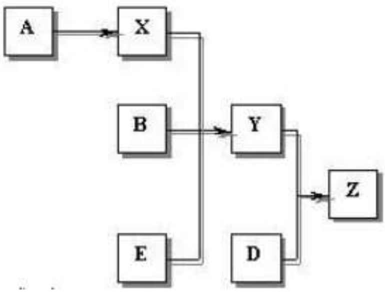


Fig. 3.3**.** Example output circuit

First, Rule 3 is triggered to derive a new fact X from a given fact A. Then Rule 2 is executed to derive a fact B from facts B and E, as well as an already known fact X. And finally Rule 1 applies the known fact D and the early fact. In to calculate the fact Z.

The EC can display its output chain to explain how a separate solution was reached; this is a major part of her explanatory abilities. The output mechanism must decide when the rules should work. There are two basic ways in which rules can be enforced. One is called a straight chain (conditionally – output), and the other reverse chain (target checks displayed)

This example uses a direct output chain. Production systems, in which the antecedent part (conditions) is first analyzed, have a so-called conditionally derived architecture. An example of an expert system of such an architecture is METADENDRAL

An alternative type of architecture, which is often used in expert systems, are purposeful production systems.

For example, a rule of the form A & B & C → D can be interpreted as "The logical conjunction A, B and C causes D" or "To prove D, you must set A, B, C".

In the latter case, the goal must be achieved by deductive inference. To do this, the consequential rules are investigated to find a rule that would achieve the goal. When such a rule is found, all its conditions are checked for truth. If the conditions are true, the product is activated. Otherwise, the search for suitable products continues.

Consider a simplified example of a production system with a consequent – derived architecture. The letters here denote the elements of the database and they are considered true if they are contained in it.

DB: AF

Rule 1: $A \& B \& C \rightarrow D$

Rule 2: $D \& F \rightarrow G$

Rule 3: $A \& J \rightarrow G$
Rule 4: $B \rightarrow C$
Rule 5: $F \rightarrow B$
Rule 6: $L \rightarrow J$
Rule 7: $G \rightarrow H$

Suppose the goal is to prove the truth of N. First of all, it is checked whether H is in the database? Since this is not the case, the system tries to derive the truth of H using rules that have H on the right.

This is rule 7. Now the system tries to derive the truth of G, as the truth of the latter entails the truth of N. Again, the database is checked: the database does not have G, therefore, an overview of the rules containing G in the right part. There are several such rules (two or three). As a strategy for "conflict resolution", we will assume that the rules are ordered by priority, and the rule with the lowest number corresponds to a higher priority.

In this case, rule 2 is chosen, so the goal now is to derive the truth of D and F. To do this, it is sufficient to show that A is true (since it is in the database), B-true (according to rule 5), C-true with rule 4). Since the truth of D and F is proved, from truth 2 follows the truth of G, and from the truth of G follows the truth of H (rule 7).

Thus the goal is achieved. The elements, the truth of which is proved, are added to the database. In this case, these are the elements H, G, D S. V.

3.4. List of control questions on the topic № 3

1. What is knowledge.
2. The main difference between knowledge and data.
3. Types of knowledge.
4. What is declarative knowledge.
5. What is procedural knowledge.
6. Classical models of knowledge representation.
7. What is the basis of logical models.
8. What use heuristic models.
9. Using logical models.
10. What is meant by the production system.
11. Advantages of production systems.
12. How to provide knowledge of network models.
13. Classification of network models.
14. Types of networks.
15. Describe functional networks.
16. Describe the scenarios.
17. Describe semantic networks.
18. Advantages of semantic networks.

19. What are frames.
20. What is the mechanism of exceptions.
21. What is the mechanism of returns.
22. List the strengths and weaknesses of products.

4. Knowledge-based problem solvers

4.1. Semantic grids: basic concepts, types, methods of description and logical derivation of semantic grids

Semantics is the science of making connections between symbols and the objects they denote. That is, it is a science that determines the meaning of signs.

A semantic grid is an oriented graph whose vertices are concepts, and arcs are a relationship between them. Concepts represent abstract or concrete objects, and relationships are connections such as "it", "belongs", "has a part".

For semantic grids, 3 types of relationships are required:
- class – element of the class (flower - rose);
- property – value (color - red);
- class element – example (rose - baccarat).

Classification of semantic grids

By type of relationship:
- homogeneous. With one type of relationship;
- heterogeneous. With different types of relationships.

By the number of relationships:
- binary. Relationship between only 2 objects;
- N-arny. Relationships connect more objects.

Types of relationships;
- relationships such as "whole - part" (class - subclass, set - element);
- functional connections ("follows", "acts", "produces");
- quantitative relationships ("more", "less");
- spatial relationships ("far from", "close to", "for", "under", "above");
- time connections ("before", "later", "during");
- attributive relationships ("has a property", "matters");
- logical connections ("AND", "OR", "NO");
- linguistic connections.

The search for a solution is reduced to finding a fragment of the grid (subnet), where the answer to the query from the knowledge base.

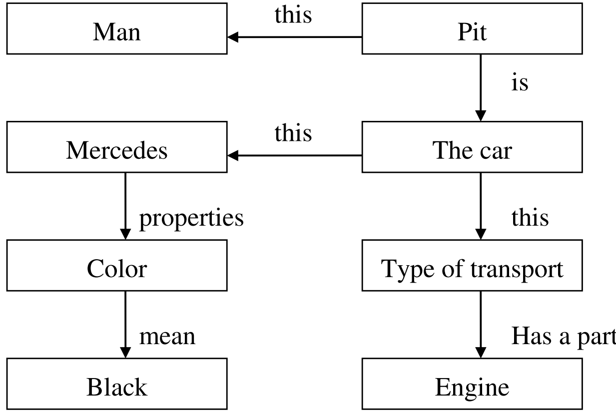


Fig. 4.1. An example of a semantic grid

The semantic grid model was proposed by the American psychologist Culian. It reflects well the idea of the organization of long-term human memory. The disadvantage is the complexity of the organization of the search and retrieval procedure. There are special languages for implementing semantic grids, such as NET. Also, samples of expert systems are known: PROSPECTOR, CASNET, TORUS.

4.2. Frames: basic concepts, frame structure, frame models

In 1975, Marvin Minsky proposed the hypothesis that human knowledge is grouped into modules – frames. He developed a model to denote the structure of knowledge, which is designed for the perception of spatial scenes. A frame here is a data structure that is designed to describe typical situations or concepts.

Definition of the frame for Minsk. A frame is the minimum possible description of a certain entity, such that further reduction of this description leads to the loss of this entity.

For example, a frame "room" is a room with walls, floor, ceiling, window and door. If you remove the "window" from this description, it will no longer be a "room", but a "pantry" and so on. The frame of any concept can be formed by combining all the facts related to this concept.

Formally, the concept within the frame model is described as follows:

Frame name
Attribute 1, value 1
Attribute 2, value 2
Attribute N, value N

That is, a frame is an aggregate description of all the basic characteristics of an object. Data structures that are designed to describe individual attributes in a frame are called slots.

Sample frames describe generalized concepts (classes) of the same type of objects with the same characteristics. Specific objects are called instances of frames. The description of frame instances is formed as a result of concretization of frames, is during filling of slots with concrete values. If certain slots are filled with specific values when describing a sample frame, they are passed to all instance frames.

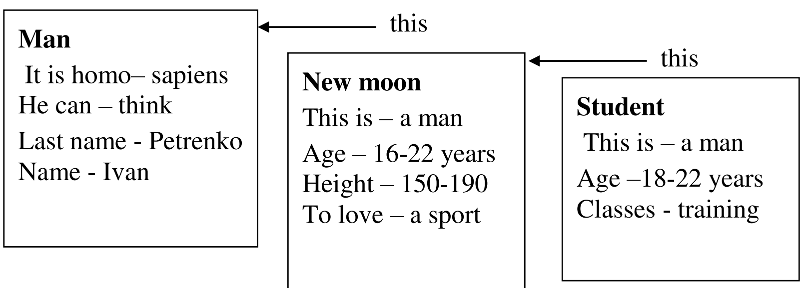


Fig.4.2. Description of frame instances

Frame models are characterized by a hierarchy of concepts and inheritance of properties**.** "Student" is derived from "new moon" and "man", so you can not specify the "last name" and "first name" slots, but only the values for specific slots.

A slot «this» indicates the frame of the highest level of the hierarchy, from which the values of the slots are inherited. For example, when asked if a student loves sports, the answer will be "yes", because it is common to all young people. Inheritance of properties can be partial, for example, "age" for the student is not inherited as it is specified explicitly in own frame.

Table 4.1. Structure of the frame

Frame name			
Slot name	Slot value	The method of obtaining the value	Connection procedure

The value of the slot can be the name of another frame, then frame networks are formed.

Ways to get the value in the instance frame:

- Default from the sample frame.

- Due to inheriting properties from the frame specified in the "it" slot.
- According to the formula specified in the slot.
- Through a connected procedure.
- Explicitly from the dialogue with the user.
- From the database.

The frame model is universal because it provides an opportunity to reflect the diversity of knowledge about the subject area.

Types of frames:

- Frame-structures are used to denote objects and concepts (university, school, audience).
- Frame-roles (teacher, student, dean).
- Scenario frames (lecture, exam, diploma defense).
- Frames-situations (revolution, sudden check of presence of students).

The main advantage of the frame model is its flexibility and clarity, which reflects the conceptual basis of the organization of human memory.

To create frame models use special languages: FRL, KRL. Known expert systems: ANALYST MODIS, TRISTAN, ALTERID.

4.3. Expert systems: purpose and principles of construction, generalized architecture, classes of problems that are solved with the help of expert systems

Purpose and principles of construction

Expert systems are a class of computer programs that offer recommendations, analyze, classify, advise, and diagnose. They are focused on solving problems that require examination by a specialist.

Unlike programs that use procedural analysis, expert systems solve problems in a narrow subject area (a specific area of expertise) based on logical reasoning. Such systems can often find solutions to problems that are unstructured and inaccurate. Through the use of heuristics, they compensate for the lack of structure, which is useful in situations where insufficient data or time eliminates the possibility of a full analysis.

The basis of the expert system is a set of knowledge that is structured to simplify the decision-making process. For artificial intelligence professionals, the term "knowledge" means the information that a program needs to behave intellectually. This information takes the form of facts or rules. Facts and rules are not always true or false, sometimes there is some degree of inaccuracy in the

validity of a fact or the accuracy of a rule. If doubt is expressed clearly, it is called the coefficient of confidence.

Generalized architecture

The basis of expert systems is knowledge.

Knowledge is a holistic and systematized set of concepts about the laws of nature, society and thinking, accumulated by mankind in the process of active transformational activity and aimed at further knowledge and change of the objective world.

Knowledge from the subject area is called the knowledge base. The knowledge base of the expert system contains facts (data) and rules (ways of presenting knowledge). The conclusion mechanism contains: an interpreter, which determines how to apply the rules to derive new knowledge, and dispatchers, who establish the procedure for applying these rules.

The expert system contains three types of knowledge:

- structured knowledge about the subject area – after this knowledge is identified, it does not change;
- structured dynamic knowledge – variable knowledge from the subject area, which is updated as new information is discovered;
- work knowledge used to solve a specific problem or conduct a consultation.

All this knowledge is stored in the knowledge base. To build it, you need to conduct a survey of specialists who are experts in a particular subject area, and then systematize, organize and index the information for ease of use.

The composition of the expert system

The fully designed EC includes 4 important components:

- knowledge base;
- conclusion machine;
- command interpreter;
- interface (explanation system).

The core of the EU is the BZ and the conclusion procedure. They should be considered together, because knowledge on the basis of which conclusions cannot be drawn does not make sense.

Knowledge base

The knowledge base is a set of all knowledge contained in the expert system.

Usually databases are described at the logical and physical levels:

- logical level of data: conceptual scheme, which in a structured form describes the subject area and its quantitative characteristics in the form of data;

- physical layer of data: a diagram showing the addresses of data in the external memory of the computer (file structures with different access).

Conclusion machine

In the ES, there is controversy between proponents of a "direct chain of reasoning" and a "reverse chain of reasoning" as a strategy for logical reasoning in general. The first is a chain of reasoning that leads from data to hypotheses, and the second is an attempt to find data to prove or disprove a particular hypothesis.

The direct chain often leads to an uncontrolled mode of questions in the dialogue, and the reverse – to the persistent repetition of questions about the goal. For this reason, the most successful systems use a combination of both circuits. But whichever way the procedure works, it will deal with unreliable data, which is more in line with the real world than previous abstractions, although the latter are more conveniently squeezed into the rigid framework of the computer.

Command interpreter

The command interpreter determines how to apply the rules to output new knowledge, and the dispatchers determine how to apply these rules.

Interface (explanation system)

The specialist uses the interface to enter information and commands into the expert system and obtain source information from it. Teams contain parameters that guide the process of knowledge processing. Information is usually given in the form of values assigned to certain variables.

The technology of expert systems provides an opportunity to receive as initial information not only decisions, but also **necessary explanations.**

There are two types of explanations*:*

- explanations issued on request. The user may at any time request an explanation of his actions from the expert system;

- explanation of the obtained solution to the problem. After receiving the decision, the user can request an explanation of how it was obtained. The system must explain each step of its reasoning that leads to the solution of the problem. Although the technology of working with an expert system is not simple, the user interface of these systems is friendly and usually does not cause difficulties in dialogue.

Classes of problems that are solved with the help of expert systems

ES classification according to the problem to be solved:

- interpretation of data;
- diagnosis;

- monitoring;
- design;
- forecasting;
- planning;
- training;
- management;
- decision support.

Classification of ES by real-time connection:
- static EC;
- quasidynamic EC;
- dynamic EC.

ES classification by learning objectives:
- systems in which it is problematic to formulate learning objectives;
- systems in which the purpose of training can be formulated, but it is not known how to do it;
- systems with known learning goals and strategies.

4.4. Stages of development of expert systems, acquisition of knowledge

- Stage of problem identification – tasks to be solved are identified, development goals are identified, experts and user types are defined.
- Stage of knowledge extraction – a meaningful analysis of the problem area, identifies concepts and their relationships, identifies methods for solving problems.
- Stage of knowledge structuring – IPs are selected and ways of presenting all types of knowledge are defined, basic concepts are formalized, ways of interpretation of knowledge are defined, system work is modeled, adequacy to system goals of fixed concepts, methods of decisions, means of representation and manipulation of knowledge is estimated.
- Formalization stage – the expert fills the knowledge base. Due to the fact that the basis of the EC is knowledge, this stage is the most important and most time-consuming stage in the development of the EC. The process of acquiring knowledge is divided into the extraction of knowledge from the expert, the organization of knowledge that ensures the effective operation of the system, and the presentation of knowledge in a form understandable to the EC. The process of acquiring knowledge is carried out by a knowledge engineer based on an analysis of the activities of an expert to solve real problems.
- Implementation of the EC – is the creation of one or more prototypes of the EC that solve the problem.

- Testing stage – the assessment of the chosen way of presenting knowledge in the EU as a whole is assessed.

4.5. Scope of expert systems

Expert systems have long been used in diagnostics, in particular in medical and automotive. ECs are also used in forecasting, planning, monitoring, management and training. For example, expert systems are already used in banking in the following areas:

- investment project analysis programs;
- currency, money and stock market analysis programs;
- programs for analyzing the creditworthiness or financial condition of enterprises and banks.

The process of creating expert systems has changed significantly in recent years. Due to the appearance of special tools for building expert systems, the time and considerability of their development has been significantly reduced.

Expert systems are quite young - the first systems of this kind, MYCIN and DENDRAL, appeared in the United States in the mid-70's. Currently, there are several thousand industrial ECs in the world, giving advice:

- when operating complex control panels, such as electricity distribution networks;
- when making medical diagnoses – ARAMIS;
- when troubleshooting electronic devices, diagnostics of failures of control and measuring equipment - Intelligence Ware;
- for the design of integrated circuits – DAA for traffic management – AIRPLAN;
- for forecasting hostilities – ANALYST;
- on the formation of the investment portfolio, assessment of financial risks – RAD, taxation – RUNE, etc.

4.6. Search and explanation of solutions in expert systems

To find solutions, you need to plan the process of asking questions or taking tests. The effectiveness of the search depends on the strategy of processing existing knowledge.

Transparency of the decision is required to explain the decision – the ability of the system to explain the methodology of decision-making. The importance of transparency for individual users:

- users of the system need to confirm that in each case the conclusion made by the program is correct;
- knowledge engineers should be able to verify that the knowledge they have formed has been applied correctly;

- subject matter experts should monitor the system's reasoning to determine how well the system is working.

4.7. Knowledge engineering

Knowledge engineering is a field of artificial intelligence related to the development of expert systems and knowledge bases. Studies methods and tools for obtaining, presenting, structuring and using knowledge.

Knowledge engineering was invented by Feigenbaum, and McCordak (1983) as: "KE – section (discipline) of engineering, aimed at introducing knowledge into computer systems to solve complex problems that usually require the participation of human experience."

Currently, this also involves the construction and maintenance of similar systems (Kendel, 2007). KE is also closely related to software engineering, and is used in many information studies, such as artificial intelligence research including databases, data collection, expert systems, decision support systems, and geographic information systems.

Example of operation of a system based on KE:

- consideration of the problem;
- query databases for this task;
- entering and structuring the received information;
- creating a database of structured information;
- testing of the received information;
- amendments and evolution of the system.

Being an art rather than a purely engineering task, IZ does not have much practical application. A subdivision of IZ is metaengineering of knowledge, which is suitable for the development of artificial intelligence.

Principles of KE

Since the mid-1980s, several principles, methods, and tools have emerged in IS that have facilitated the process of acquiring and working with knowledge. Here are some key ones: There are different types of knowledge and you need to use specific methods and techniques to work with them.

There are different ways to present, use, understand knowledge and work with them can help rethink and use old knowledge in a new way. IZ uses methods of structuring knowledge to speed up the process of obtaining and working with knowledge

There are 2 different points of view on KE

- broadcast – it is traditional, which involves the direct transfer of human knowledge into the machine;
- model is an alternative view. It involves modeling the problem and ways to solve it by the system itself.

4.8. List of control questions on the topic № 4

1. What is semantics?
2. What is a semantic grid.
3. Mandatory types of semantic grid relationships.
4. Classification of semantic grids.
5. Types of relationships.
6. Give an example of a semantic grid.
7. Disadvantages of the semantic grid.
8. Who proposed the hypothesis of frames.
9. What is a frame?
10. What are sample frames?
11. What are instances of frames.
12. Methods of obtaining value in the instance frame.
13. Types of frames.
14. What are expert systems.
15. Types of knowledge of the expert system.
16. The composition of the expert system.
17. What is a knowledge base.
18. What determines the command interpreter.
19. Classification of expert systems according to the tasks to be solved.
20. Real-time EU classification.
21. EU classification by learning objectives.
22. Stages of development of expert systems.
23. Scope of the EU.
24. Search for EU solutions.
25. What is knowledge engineering.
26. Principles of knowledge engineering.
27. Perspectives on knowledge engineering.

5. Modern trends and approaches to the creation of artificial intelligence systems

5.1. Modern software and tools for creating artificial intelligence systems: Visual Prolog, Allegro Common Lips, CLIPS, JESS

Logical programming is a programming paradigm, as well as a section of discrete mathematics that studies the methods and possibilities of this paradigm, based on the derivation of new facts from these facts according to given logical rules. Logic programming is based on the theory of mathematical logic. The most famous logic programming language is Prolog, which is essentially a

universal inference machine that works under the assumption of a closed system of facts.

The first logical programming language was the Planner language, which provided the ability to automatically derive the result from the data and set rules for the search for options (the set of which was called a plan). Planner was used to reduce the requirements for computing resources (using the backtracking method) and to provide the ability to derive facts without the active use of the stack. Then the Prolog language was developed, which did not require a plan to search for options and was, in this sense, a simplification of the Planner language.

From the Planner language also came the logical programming languages QA-4, Popler, Conniver, and QLISP. The programming languages Mercury, Visual Prolog, Oz and Fril were built from the Prolog language. Based on the Planner language, several alternative logic programming languages that are not based on the backtracking method have also been developed, such as Ether (see Shapiro's review [1989]).

The programming paradigm is the basic principles of programming. The programming paradigm determines how a programmer views a program. For example, in object-oriented programming, the programmer considers the program as a set of interacting objects, while in functional programming the program can be represented as a sequence of computing functions without states.

Visual Prolog is a logic programming language. The prologue is based on first-order predicate theory. The name of the programming language is deciphered as – "Programming in logic".

The main concepts in the Prologue language are facts, inference rules and queries that allow you to describe the knowledge base, inference procedures and decision making.

Facts in Prologue are described by logical predicates with specific meanings. The rules in the Prologue are written in the form of rules of logical inference with logical conclusions and the list of logical conditions.

A special role in the Prologue interpreter is played by specific queries to knowledge bases, to which the logical programming system generates answers "truth" and "lie". For generalized queries with variables as arguments, the Prolog system outputs specific data to confirm the truth of the generalized information and output rules.

Facts in knowledge bases in the Prologue language represent specific information (knowledge). Generalized information and knowledge in the Prologue language are given by the rules of logical inference (definitions) and sets of such inference rules (definitions) over specific facts and generalized information.

The beginning of the history of the language dates back to the 1970s. As a declarative programming language, Prolog takes as a program a specific description of a task or knowledge base and draws its own logical conclusions,

as well as finding solutions to problems using the mechanism of backtracking and unification**.**

Allegro Common Lisp – implementation of the Common Lisp language (Common Lisp) – a functional programming language designed to standardize the various versions of the Lisp language that existed before the standard. This dialect is not an implementation, but instead only a specification of a programming language.

The Common Lisp programming language implements several paradigms, including:

- supports imperative, functional and object-oriented programming paradigms;
- dynamic programming language that accelerates program development through iterative compilation;
- contains CLOS, an object system that supports multimedia methods and method combinations;
- can be expanded through standard macro mechanisms.

Data types

Common Lisp data types are objects that are stored in variables, not the variables themselves (which corresponds to a dynamic typing system). Each variable can have as value any Lisp object. To improve performance, it is allowed to declare certain restrictions on possible types of variable values.

The set of all objects is denoted by the symbol t. An empty set of objects or an empty list is denoted by the nil symbol, which corresponds to the logical "incorrect". Any value other than nil is considered logical "true".

Numbers: Common Lisp has data types for integers of any size (limited by the amount of available memory), rational numbers (formed by dividing integers), floating-point numbers, and complex numbers. Letters: Representations of both printed letters and special purpose symbols.

Symbols (atoms): named data. Each character has a list of attributes, and can contain any Lisp object. Can be used as variables or functions. Lists: sequences represented as linked cons cells. Lists are created recursively, adding a new element to an existing list creating a new cons. cons is a Lisp object that has two fields: car can have any value, and cdr pointer to the previous cons. Arrays: collections of Lisp objects of a certain dimension. Any Lisp objects can be stored in arrays. There are, to improve efficiency, arrays that can only contain elements of a certain type. A one-dimensional array with elements of any type is called a vector, an array of letters string, bits bit-vector.

Hash tables: provide a mechanism for effectively comparing any object (key) with another object (value). Packages: collections of characters used as namespaces.

File names: match file names in a way that is as independent as possible from the file system implementation. Threads: Used for I/O operations, and for reading information from strings.

Random structures: data structures used to store information about the state of the built-in random number generator. Records: user-defined data structures. Entries have named components.

Conditions: used as signals to control the operation of the program. Similar to Exceptions in some programming languages. In addition to these data types, CLOS defines data types for classes, methods, general methods.

Macros

In Common Lisp, macros are operators that are implemented by code conversion. The macro is determined by how the code that calls it is converted.

The transformation, also called macro expansion, is performed by the compiler automatically. As a result, the code generated by the macro becomes the same part of the code as the user-entered program code.

Typical uses of macros include:

- new execution order management structures (cycles, branches, etc.);
- design of the scope and binding of variables;
- simplification of complex code fragments that are often repeated;
- determination of higher level forms with side effects of compilation time;
- data-driven programming;
- built-in subject-oriented programming languages (SQL, HTML, Prologue, etc.).

Macros specify a defmacro macro. A special macrolet operator allows you to define local macros. You can also specify macros for characters using define-symbol-macro and symbol-macrolet. Paul Graham's On Lisp is a detailed look at the possibilities of macros in Common Lisp.

Examples

For example, the following macro determines the operator aif, which receives two or three forms, calculates the value of the first, stores it in the variable it and if the value true (t) performs the second form, and if the value is not true and the third form – then third (so-called anaphoric if):

```
(defmacro aif (test then &optional else)
  `(let ((it ,test))
    (if it ,then ,else)))
```

In this case, the value of the first form stored in the variable it is available during the execution of both of the following:

```
(aif (long-and-complicated-calculation)
  (print it))
```

which corresponds to the following pseudocode:

```
it := long-and-complicated-calculation ()
if it then
  print it
```

Object-oriented programming

Read more: CLOS. Common Lisp Object System (CLOS) is a Common Lisp extension that adds support for object-oriented programming capabilities to Common Lisp. This extension is based on common functions, multiple inheritance, declarative combination of methods, and meta-object protocol.

The fundamental objects of CLOS are classes, instances of classes, general functions, and methods. A general function is a function whose behavior depends on the classes or values of the passed arguments. A common function object contains a set of methods, a lambda list, a method combination method, and other information. Methods determine the behavior of general functions depending on the classes of arguments passed; in other words, methods specialize in general functions. When called, the general function performs a subset of its own methods depending on the classes of arguments. [4] The usual Common Lisp function has a single "body" (instruction list) that is always executed when a function is called. Unlike ordinary functions, common functions have a set of "bodies", only a subset of which is executed when calling a common function. The "bodies" selected and the method of their combination depend on the classes of parameters of the general function and the method of combination.

CLIPS

CLIPS, (from English C Language Integrated Production System) – software environment for the development of expert systems. The syntax and name are suggested by Charles Forgy in OPS (Official Production System). The first versions of CLIPS were developed in 1984 at the Johnson Space Center, NASA (as an alternative to the then-existing ART * Inference system), until funding was suspended in the early 1990s.

CLIPS is a production system. The main idea is to present knowledge in the form of the following form:

```
 Rule 1:
  if
    (conditions 1 are met)
  then
    (perform actions 1)
Rule 2:
  if
    (conditions 2 are met)
  then
    (perform actions 2)
```

...

This representation is close to human thinking and differs from programs written in traditional algorithmic languages, where actions are ordered and performed strictly by the algorithm.

CLIPS is one of the most widely used tool environments for developing expert systems due to its speed, efficiency and freeness. As a public domain, it is still updated and maintained by its original author, Gary Riley.

CLIPS includes a full-fledged object-oriented COOL language for writing expert systems. Although it is written in C, its interface is much closer to the LISP programming language. Extensions can be created in C, and CLIPS can be integrated into C programs.

CLIPS is designed for use as a direct inference language. Like other expert systems, CLIPS deals with rules and facts.

Facts

The information on the basis of which the expert system makes a logical conclusion is called facts. CLIPS has 2 types of facts: ordered and template. Template facts have a template specified by the deftemplate construct. The ordered ones do not have an explicit deftemplate construction, however it is implied. The pattern fact resembles a structure in C or an entry in Pascal, the fields are called slots and are declared by the slot construct. For example, the following template declares a template with the name cars and fields: model, color and number.

```
(deftemplate cars
 (slot model)
 (slot color)
 (slot number)
)
```

The facts are stored in working memory. The new facts are placed in working memory by the assert command. For example, the following command

(assert (cars)) will add to the working memory the ordered fact of cars.

The following command will place a template fact with three attributes.

```
(assert
   (cars
     (model "Audi")
     (color Black)
     (number "WY 2576")
   )
 )
```

CLIPS does not allow the placement in the memory of facts with the same values of the slots, although, if necessary, this can be allowed with the appropriate settings.

Rules

Subject area knowledge is presented in CLIPS in the form of rules that have the following structure:

```
(condition) {synonyms: antecedents in logic,
             left part – LHS in CLIPS terms}
=>
(action) {synonyms: consequent in logic,
          right part – RHS in terms of CLIPS}
```

The left part of the rule is a condition of its operation, and the right part is the actions that must be performed if the conditions are met. Character => special character separating LHS and RHS.

Rules are announced using the defrule command

Example rule:

```
(defrule search-black-audi
  (cars (model "Audi") (color Black))
  =>
  (printout  " We have black Audi!" crlf)
 ).
```

Variables

When a fact is stored in memory, its fields can only be changed by deleting and entering a new instance of the fact, even the modify command sequentially deletes and adds a new instance of the fact.

In contrast to facts that are static, variables can take on different values. The name of the variable must always be preceded by a "?" To associate a variable with a fact, use the entry:

? var <- (fact_name (field value))

The defglobal construct allows you to describe variables that are global in the context of the CLIPS environment. That is, the global variable is available anywhere in the CLIPS environment and retains its value independently of other constructs. The following is used to declare a global variable:

(defglobal [<defmodule-name>]? * <global variable name> * = <expression>).

Logic inference machine

The process of adding rules to the working list and their execution is controlled by a logic output machine (LIM).

Tabl. 5.1. **Reaction** LIM to certain events

Event	**Action**

Adding facts to working memory	# Comparison of facts with rules from the knowledge base # Comparison of facts with rules from the worksheet of rules
Delete facts from working memory	# Comparison of facts with rules from the knowledge base # Comparison of facts with rules from the worksheet of rules
The comparison found rules that correspond to the facts of working memory	Add found rules to the rule worksheet
New rules have been added to the worksheet	The working list of rules is sorted according to the chosen conflict resolution strategy
When comparing the facts with the working list of rules, irrelevant rules were revealed	Outdated rules (the conditions of which do not satisfy the facts) are removed from the work list
Executing command (RUN)	The actions (right part) of the rule, which is the first in the working list of rules, are performed.
The worksheet has become empty	Execution of rules from the working list stops

Conflict resolution strategies

A person cannot always set complete conditions that would satisfy reality. There is a legend, according to which Diogenes of Sinope, in defining Plato's "Man is an animal on two legs, deprived of feathers", plucked a chicken and brought it to school, declaring: "Here is a Platonic man!" To which Plato was forced to add to his definition "... and with wide nails." When rules appear in the knowledge base that satisfy the facts but perform opposite actions, a conflict of rules arises. For example, there are two rules:

- (If a person pushes another person – punish the person for hooliganism;

- (If a person pushes another person on whom the truck was driving – reward the person for saving a life).

These two rules will conflict with each other. The first rule is more general and it is always activated if the second is activated. But the second rule must be fulfilled first. CLIPS has several strategies for resolving such conflicts. But even if it is not possible to choose the right strategy for all cases, you can prioritize the rules. High priority rules will be followed first.

Jess

Jess is a system for developing expert systems, which is a descendant of CLIPS and is written entirely under JAVA. This system was developed at Sandia National Laboratories in Livermore, California. With Jess, you can create Java software that treats knowledge as declarative rules.

Its powerful scripting language gives access to all Java APIs. Jess includes a full-featured Eclipse-based development environment. Jess uses an extended version of the Rete algorithm for the rule process. Rete is a very effective mechanism for solving difficult tasks such as "many-to-many". It also has many unique features, including a reverse chain, can directly manage and evaluate Java objects.

Jess is also a powerful Java scripting environment from which you can create Java objects, Java methods, and implement Java interfaces without compiling Java code.

5.2. An ontological approach to the presentation and interpretation of knowledge in distributed information environments such as the Internet

The World Wide Web is rapidly entering literally all walks of life. The Internet is becoming an increasingly powerful and important source of information. Data processing tools on the network are finding it increasingly difficult to deal with the avalanche of information that already exists and is being added to the network on a daily basis. In addition, data on the Internet is organized very spontaneously and not systematically. Many technical problems must be solved for the effective sharing of information. The problem of connecting heterogeneous and distributed computer systems is known as the interoperability problem. Interoperability must be provided at both technical and informational levels. Problems that could arise due to data heterogeneity are already known within the community of distributed database systems: structural heterogeneity (schematic heterogeneity) and semantic heterogeneity (data heterogeneity). Structural heterogeneity means that different information systems store their data in different structures. Semantic heterogeneity considers the content of the information element. To achieve semantic interoperability in a heterogeneous information system, the content of the information exchanged must be clear in all systems. Semantic conflicts occur whenever two contexts do not use the same interpretation of information. The following reasons for semantic heterogeneity can be identified:

- conflicts of confusion occur when information elements seem to have the same meaning but differ in reality, for example due to different temporal contexts;

- scaling conflicts occur when different reference systems are used to measure values. Examples are different currencies;

- naming conflicts occur when the schemes of information notation are significantly different. A common phenomenon is the presence of homonyms and synonyms.

The use of ontologies to explain implicit and hidden knowledge is a possible approach to overcome the problem of semantic heterogeneity.

Therefore, the purpose of this work was:

- understand what the term "ontology" means;
- investigate the purpose of ontologies and thus understand how to use them for problems related to information integration;
- get acquainted with the current state of technology, in particular:
  - basic ideas or models of modern ontologies;
  - methods and tools to support the process of building and using ontologies;
- have an idea of the potential use of ontologies.

What is an ontology?

Origin, history of development and basic definitions. The term ontology came from philosophy (derived from Aristotle's attempt to classify objects in the world), where it is used to denote a system of knowledge relating to the world around (as opposed to a system of knowledge about the inner world of man). In other words, ontology is the science of being, the science of the nature of things and the relationships between them. In the context of information technology of knowledge representation, the term ontology can define a mechanism, a method used to describe a certain area of knowledge (subject area), in particular the basic concepts of this area, their properties and relationships between them.

There are many definitions of ontology, some of which contradict each other, but the most widely used is the definition: "Ontology is a clear specification of conceptualization." Here conceptualization means an abstract representation of the subject area. The definition is also common: "Ontology is a common understanding of an area of interest."

The word "ontology" is often perceived almost as a synonym for knowledge engineering in artificial intelligence, conceptual modeling in databases, and subject area (domain) modeling in object-oriented design.

Today, the ontology can be understood as:

- a solid semantic basis in determining the content;
- general logical theory, consisting of a dictionary and a set of statements in a language of logic;
- the basis for communication between people and computer agents.

Used in the context of intelligent systems development, ontologies have been classified into applied ontologies, domain ontologies, problem ontologies, and top-level ontologies covering various aspects relevant to intelligent systems modeling (Figure 5.1).

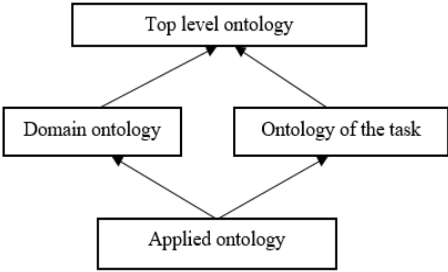


Fig. 5.1. Types of ontologies, according to their level of dependence on a particular task or point of view

Top-level ontologies describe very general concepts, such as place, time, substance, object, event, action, etc., that are independent of a particular problem or domain: then, it seems, at least theoretically, reasonable to have unified top-level ontologies for larger user communities. Domain ontologies and task ontologies, respectively, describe vocabulary related to a typical field (eg, medicine, automotive) or a typical task or activity (eg, diagnostics, sales) by specializing the terms presented in the top-level ontology. Applied anthologies describe concepts that depend on both the ontology of problems and the ontology of the domain. These concepts often correspond to the roles that the essence of the domain plays when performing an activity, such as replacing a part or reserving a component.

In the late nineties, the role and influence of ontologies was rather limited In the late 1990s, the role and influence of ontologies was rather limited, and the understanding that the conceptual, yet feasible application domain model provided significant additional value for all types of application scenarios, such as knowledge management or e-commerce, changed dramatically. Obviously, the main impetus for ontologies, gave the prediction of SemanticWeb. In SemanticWeb, ontologies provide conceptual reinforcement to create metadata semantics.

Ontologies allow us to present new concepts so that they become suitable for machining. With the help of ontology, you can "bridge the gap" between new concepts that the system has not yet encountered, and descriptions of already known classes, relationships, properties and objects of the real world.

Motivation. Here are some reasons for developing ontologies:

- sharing a common understanding of the structure of information among people or software agents;
- the ability to reuse domain knowledge;
- the ability to make explicit domain assumptions;
- separation of domain knowledge from operational knowledge;

- domain knowledge analysis.

Sharing a common understanding of the structure of information among people or software agents is one of the most common goals in developing ontologies. For example, suppose several different Internet pages contain medical information or provide medical e-commerce services. If these Internet pages share and publish the same basic ontology of terms they use, then computer agents can extract and combine information from these different Internet pages. Agents can use this connected information to respond to user requests or as input to other applications.

Providing reuse of domain knowledge has been one of the driving forces of the recent wave in ontology research. For example, models for many different domains must represent the concept of time. This representation includes the concepts of time intervals, points in time, relative measures of time, etc. If one group of researchers has developed such a detailed ontology, others can simply reuse it for their domains. Additionally, if you want to build a large ontology, you can integrate several existing ontologies that describe parts of a large domain.

Making the domain assumptions underlying the implementation clear makes it easy to change these assumptions if our knowledge of the domain changes. Hard coding of world assumptions in programming language code makes these assumptions not only difficult to find and understand but also difficult to change, especially for someone without programming experience. In addition, explicit domain knowledge specifications are useful for new users who need to learn what terms mean.

Separating domain knowledge from operational knowledge is another common use of ontologies. We can describe the task of configuring a product from its components according to the required specification and implement a program that makes this configuration, regardless of the product and components directly.

As soon as a declarative specification of terms becomes available, domain knowledge analysis is possible. Formal analysis of terms is extremely valuable when attempting to reuse existing ontologies and extend them.

Applications and examples. Ontologies have proven that they can be essential elements of many applications. They are used in agent systems, knowledge management systems and e-commerce. They can also combine intelligent information, provide semantically based access to the Internet, and extract information from texts in addition to what they use in many other applications to explicitly identify the knowledge contained in them. However, ontologies are useful not only for applications in which knowledge plays a key role, but they can also cause a significant change in the current content of the Web in accordance with the concept of SemanticWeb. Examples of the use of ontologies on the Internet can be e-commerce sites, search engines, Web-services.

For a more complete picture of what an ontology is, we give an interesting example of an approach to the visualization of ontologies. In the process of working with ontologies, the user often needs a brief overview of the entire hierarchy, so that a quick and easy transition from one class of hierarchy to another is possible. These requirements are met by a representation scheme based on hyperbolic geometry. This visualization technique allows a quick transition to a class that is far from the center, as well as a more detailed study of the classes and their environment. In Fig. 5.2 presents an image of the scientific ontology of KA2 using hyperbolic geometry.

Presentation of ontologies

The components that make up ontologies depend on the representation paradigm. But almost all models of ontologies to some extent contain concepts (other options: concepts, classes, entities), properties of concepts (other options: slots, attributes, roles), the relationship between concepts (other options: connections, dependencies, functions) and additional restrictions (defined by axioms, in some paradigms facets).

Concepts are used in a broad sense. They can be abstract or concrete, elementary or complex, real or fictitious. In other words, a concept can be anything to which something is asserted and, therefore, could also be a description of a task, function, action, strategy, thought process, etc.

Concepts in ontology are usually organized in taxonomy. Taxonomies are sometimes considered complete ontologies, although ontologies cannot be limited to this. Taxonomies are widely used to organize ontological knowledge of a subject area, using generalization/specialization relationships through which single/multiple inheritance could be applied.

Relationships represent a type of interaction between the concepts of the subject area. Examples of binary relationships are "part-of" and "connected-to".

Axioms are used to model statements that are always true. They can be included in the ontology for several purposes, such as limiting the information contained in the ontology, checking the correctness or outputting new information.

The term instance is used to represent elements in the subject area, ie the element of this concept. The ontology, together with many individual instances, forms the knowledge base.

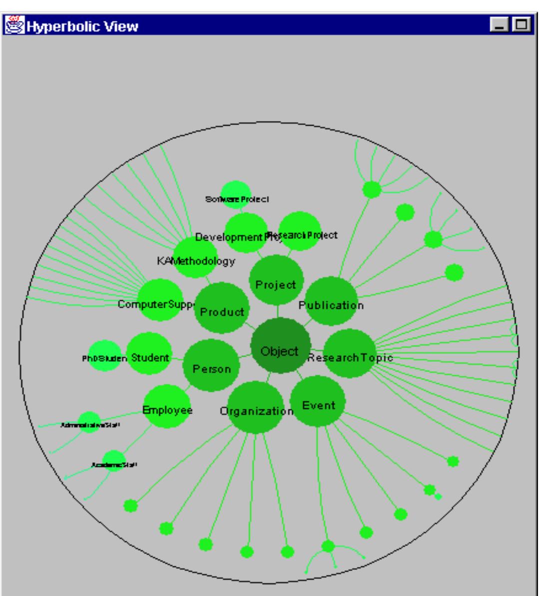


Fig. 5.2 An example of ontology visualization using hyperbolic geometry

It should be noted that today the question of a clear distinction between ontology and knowledge base remains open. Because instances themselves can be concepts and often in modeling you need to decide what to refer to the concept and what to the instance, and what to both the concept and the instance. Therefore, some authors, for example, include a copy in the elements of formalization.

As mentioned above, ontologies consist of hierarchical descriptions of important concepts in the domain along with descriptions of the properties of each concept. The degree of formalism used in fixed descriptions can vary from natural language to logical formalisms, but it is clear that the growth of formalism and regularity facilitates machine interpretation of information.

Ontology models can be classified as follows:

- simple models (for example, having only concepts);
- based on frames (have only concepts and properties);
- based on logic (for example, Ontolingua, DAML + OIL).

Informal linguistic models are often used to specify ontologies. In such models, ontological concepts are defined according to verbal definitions, like an explanatory dictionary. Some kinds of the most important connections can be established between concepts. A glossary of terms in some subject area, a thesaurus with its concepts and connections that define the terms of natural language can be considered as ontologies. Information methods are used to

establish a connection between verbally defined concepts, as well as to search for concepts relevant to the request.

In contrast to verbal models, ontologies based on logic are formally defined and have the ability to make formal judgments. One of the first among the proposed models was Ontolingua.

Ontolingua is a language based on KIF and FrameOntology. KIF (Knowledge Interchange Format) was developed to solve the problem of heterogeneity of languages for the presentation of knowledge. It provides the definition of objects, functions and relationships. KIF has declarative semantics based on numerous first-order predicates with prefix notation. In fact, because KIF is an exchange format, its use is too time consuming to specify ontologies. However, FrameOntology, formed on top of KIF, allows you to define the ontology according to the frame paradigm, including terms such as class, instance, subclass, etc. But it is at the same time less expressive than KIF. Ontolingua allows you to include in the definition of KIF expressions based on FrameOntology. Ontolingua was intended as an intermediate language for the interaction of heterogeneous ontologies.

The OKBC (Open Knowledge Base Connectivity) ontological model, formerly known as the Generic Frame Protocol, uses a frame model. It defines a protocol for accessing knowledge bases stored in frame-based knowledge representation systems and is considered as a supplement to language specifications that are designed to support knowledge sharing. The GFP knowledge model is an implicit formalism of representation that underlies OKBC and provides a set of constructs such as constants, frames, slots, facets, classes, and knowledge bases. The OKBC model serves as an intermediate ontology language that maintains communication using OKBC.

The XOL (XML-based Ontology Exchange Language) model is designed to meet the need for a language with the semantics of object-oriented knowledge representation systems, but with XML syntax. The definitions of the ontology that XOL is designed to encode include both schematic information (metadata), such as class definitions, and non-schematic information (basic facts) such as object definitions in object databases. XOL is a format for exchanging ontological definitions and is therefore not intended for ontology development. It serves as an intermediate language for transferring ontologies among various database systems, ontology development tools or applications. XOL is similar to other ontology exchange languages, in particular, it uses the OKBC model.

FLogic (Frame Logic) – a language that integrates languages based on frames and first-order predicate calculus. It takes into account such structural aspects of object-oriented and frame-based languages as object identity, complex objects, inheritance, polymorphic types, query methods, encapsulation, and more. In a sense, FLogic is in the same relation to the object-oriented paradigm as the classical predicate calculus to relational programming.

Another class of formal ontological models is based on different types of descriptive logics (DL). The family of formalities for the representation of knowledge based on logic is well suited for the representation and reasoning of terminological knowledge and ontologies. They are mainly characterized by a set of constructors that allow you to form complex concepts and roles from elementary concepts.

These models are interpreted in the tasks of checking the categorization between concepts and acceptability, in terms of definitions of concepts like sets of interdependent concepts. KL-ONE, LOOM, CLASSIC, FaCT are examples of reasoning systems that are often used to model ontologies. They provide typical reasoning services – classification, compatibility and instance control.

To consider the concepts of descriptive logic in the form of a set of objects, the concept of interpretation is introduced. The feasibility test consists of proof that there is at least one interpretation according to which the concept corresponds to a non-empty set. Categorization between concepts means that for any interpretation, a set that interprets one concept is a subset of a set that interprets another concept. Proof of categorization comes down to such feasibility.

The first DL system was KL-ONE, which claimed a transition from a semantic network to more sound terminological (descriptive) logics. The impact was very significant, so KL-ONE is considered the ancestor of a whole family of languages. KL-ONE presents the most key concepts explored in subsequent DL work, such as concepts and roles and their interactions, important ideas on "value constraint" and "numerical constraint" that changed the use of roles in concept definitions, and the most important inference. categorization and classification.

The idea of creating CLASSIC was based on providing expressive language and effective reasoning. CLASSIC, supports the description of objects in terms of their relationship to other objects, as well as in terms of the level of content structure. A significant disadvantage of systems of this type – incomplete reasoning algorithms. But today CLASSIC is widely used by various systems, such as OBSERVER.

LOOM belongs to the same type of models as CLASSIC. It is a high-level programming language and environment designed for use in building expert systems and other intelligent applications. LOOM achieves tight integration between rule-based and frame-based paradigms. LOOM supports descriptive language for modeling objects and relationships, and statement language for defining constraints on concepts and relationships.

FaCT (Fast Classification of Terminologies) is a system that provides reasoning support for ontology design, integration, and verification. FaCT is a DL classifier that can also be used to check for inconsistencies in modal and other similar logics. The most interesting feature of FaCT is its expressive logic (including the SHIQ reasoning mechanism), its optimized tabular

implementation (which has become the standard for DL systems) and its CORBA-based client-server architecture. FaCT optimizations, aimed at improving system performance when classifying realistic ontologies, have made it the fastest compared to previous DL systems.

The most significant development of ontological models based on logic is part of the development of the Semantic Web. Let's pay attention to such Web standards as XML and RDF.

XML (eXtensible Markup Language) is a metalanguage derived from SGML (Standard General Markup Language). It was developed by the W3C, for ease of implementation and interoperability with SGML and HTML. As a language for the Web, the main advantages of XML are the following: it is easy to analyze, the syntax is well defined and it is suitable for human perception. Because XML is widely used, there are many software tools for parsing and managing it. XML allows users to define their own tags and attributes, define data structures (embedding them), extract data from documents. XML itself does not have any special capabilities for ontology specification, as it offers only a simple but powerful way to define the syntax for the ontology specification language. Therefore, XML can be used for purposes such as providing the syntax of a language set, such as XOL or OIL, and to cover the needs of ontology exchange.

RDF (Resource Description Framework) was developed by the W3C to create metadata describing Web resources. This makes it possible to specify the semantics of XML-based data in a standardized, interoperable manner. The RDF data model consists of three types of objects: resources-objects that refer to an address on the Web; properties that define certain aspects, characteristics, attributes or relationships used to describe the resource; and instructions that assign values to a property in a particular resource.

The RDF data model is not provided with mechanisms for determining the relationship between properties (attributes) and resources. This is the role of RDF Schema (RDF Schema Specification language), a declarative language used to define RDF schemas. It is based on some ideas of knowledge representation (semantic networks, frames and predicate logic), but is much easier to implement (but also less expressive) than complete predicate number languages such as CycL and KIF. The main classes are class, resource, and property, and hierarchies and type constraints can be defined. Some basic limitations are also identified. However, RDF Schema lacks functions and axioms for ontological definitions, but concepts, relationships, and instances can be easily defined.

Next, consider languages based on these standards.

OIL (Ontology Interchange Language) is a proposal for a common standard for describing and exchanging ontologies. It is the first language for presenting ontologies in the W3C. This is the development of existing proposals such as OKBC, XOL, RDF and the first Web-based language, which is designed

to form ontologies with formal semantics and reasoning services provided by descriptive logic. In OIL, an ontology is a structure consisting of several components organized into three levels: an object layer (which deals with instances), a first meta-level (which contains definitions of ontologies), and a second meta-level or ontological container (which, contains information on the features of the ontology, such as authorship). Concepts, relations, functions and axioms can be defined using OIL ontological definitions.

DAML + OIL (DARPA Agent Markup Language + OIL) was developed by a joint committee with the United States and the European Union (IST) in the context of DAML, a DARPA project to provide semantic interoperability in XML. It is the result of a merger of DAML and OIL languages and is based on earlier W3C standards such as RDF and RDF Schema, extending them to richer modeling primitives. This model was designed as a basis for Web Ontology Language (OWL).

OWL extends RDF and RDF Schema, providing additional vocabulary alongside formal semantics, which is the basis of DAML + OIL, as well as built-in support for ontology mapping. OWL has three dialects: OWL Lite, OWL DL and OWL Full. They are complex and can be used in different applications depending on the need or ease of output, or the formality of the descriptions.

SHOE (Simple HTML Ontology Extension), developed at the University of Maryland, was the first extension of HTML to include machine-readable semantic knowledge in HTML or other Web documents. It has recently been adapted to support XML. The intent of this language is to make it possible for agents to gather meaningful information about Web pages and documents by improving search engines and knowledge collection. This process consists of two phases: defining the ontology, annotating the HTML page with ontological information to describe yourself and other pages.

OML (Ontology Markup Language), developed at the University of Washington, is based in part on SHOE. In fact, the XML SHOE transformation was first considered. Therefore, OML and SHOE have many similar features. There are four different levels of OML: The core of OML, which is related to the logical aspects of language, includes all other levels; Simple OML displayed directly in the RDF Scheme; Abbreviated OML, which includes conceptual graphics capabilities, and Standard OML are the clearest version of OML.

Ontology design methodologies

Previous sections have provided information on the use and languages of ontologies. But it is also important to support the process of developing ontologies. In this section, we will describe which methodologies provide support for the ontology engineering process.

The process of constructing or developing ontologies used in information systems remains an art rather than a science, and there is no one right way or methodology to develop ontologies. This situation can be changed only through

an understanding of how to start building ontologies, ie a good methodology is needed to develop ontologies. To date, there have been several proposals for such a methodology, where people reflect their experience in building ontologies. Here are the most promising of them.

The METHONTOLOGY methodology was developed in the Artificial Intelligence Laboratory of the Madrid Polytechnic University. METHONTOLOGY begins with the identification of the following actions that are included in the development of ontologies: specifications, knowledge collection, conceptualization, integration, implementation, evaluation, documentation. The life cycle of the ontology, based on the improvement of the prototype, allows the construction of ontologies at the level of knowledge and includes: identification of the process of ontology development, life cycle based on the development of prototypes and private methods of each action. METHONTOLOGY is one of the most complete; however, recommendations for pre-development processes are needed, and some actions and methods need to be defined in more detail.

The TOVE methodology is based on the TOVE (Toronto Virtual Enterprise) design experience. It mainly involves the construction of a logical model of knowledge, which must be determined through the ontology. This model is not created directly. First, an informal description is made of the specifications that ontologies must meet, and then this description is formalized. There are no life cycle guidelines in this methodology. In addition, although ontologies have been developed using this methodology, and there are applications that use these ontologies, their scope is limited to business. The TOVE approach is most interesting for distinguishing an ontology estimate, especially since the means of performing this estimate are provided in the form of completeness theorems. These theorems are useful in many ontology support problems.

SENSUS-based methodology. This ontology is used in natural language processing and was developed at ISI (Institute of Informatics) by a natural language group to provide a comprehensive conceptual framework for the development of machine translators. The actual content was obtained by extracting and combining information from various electronic sources. To create the basis of the ontology, the process began with the manual merging of the PENMAN and ONTOS top-level models (two very high-level linguistic ontologies) and the semantic categories from the dictionary. Then the WordNet ontology was connected (again manually) to the ontological database. The connection tool was then used to merge WordNet with the English dictionary. After that, to support machine translation, the result of this merger was supplemented by Spanish and Japanese lexical entries from Collins' Spanish-English Dictionary and Kenkyusha's Japanese-English Dictionary. SENSUS has more than 50,000 concepts organized in a hierarchy according to the level of

abstraction. Includes terms with high and medium levels of abstraction, but does not cover terms from special domains.

CommonKADs is a widely used methodology for developing knowledge base systems in which ontologies play an important role. The KACTUS project was the next project focused on the problem of ontology design. This methodology emphasizes the design, redesign and reuse of modules. The ontology is created from a library of small-scale ontologies, which requires mapping between the different ontologies included in the development of the new ontology. The selection of relevant ontologies from the library is supported by the indexing scheme. This approach to the development of ontologies is due to the development of applications. Yes, every time an application is built, an ontology is built that represents the knowledge needed to create the application. This ontology can be developed using the reuse of other ontologies and can also be integrated into the ontology of the following applications.

5.3. List of control questions on the topic № 5

1. What is logical programming.
2. What is the programming paradigm.
3. Describe the language Visual Prolog.
4. Basic concepts of Visual Prolog.
5. Describe the language of Common Lips.
6. Common Lips paradigms.
7. Data types in Common Lips.
8. Maktoso in Common Lips.
9. Describe the CLIPS environment.
10. What is structural heterogeneity.
11. What is semantic heterogeneity.
12. Name the causes of semantic heterogeneity.
13. What is an ontology.
14. Types of ontologies according to the level of dependence on a specific task.
15. Presentation of ontologies.
16. Ontology design methodologies.

6. Tasks for practical and independent work

6.1. Task № 1. Search for solutions to an intellectual problem in space. Methods of blind search and heuristic search

When planning in the task space, space is formed as a result of the introduction on a set of tasks of relations such as: "part - whole", "task - subtask", "general case - special case", etc.

In other words, the task space reflects the decomposition of tasks into subtasks (goals on subgoals). The problem is to find the decomposition of the original problem into subtasks, which leads to problems whose solution the system knows. For example, it is known how the values of *sin (x)* and *cos (x)* are calculated for any value of the argument and how the division operation is performed. If it is necessary to calculate *tg (x),* then the solution will be to present this problem in the form of decomposition *tg (x) = sin (x)/cos (x) (except x = π/2 + k π).*

Representation of tasks in the state space involves setting a number of descriptions: states, sets of operators and their effects on transitions between states, target states. State descriptions can be character strings, vectors, two-dimensional arrays, trees, lists, etc. Operators translate one state into another. Sometimes they are represented as products A => B, which means that state A will turn into state B.

The space of states can be represented as a graph, the vertices of which are denoted by states, and the arcs – by operators. Thus, the problem of finding a solution to the problem <A, B> when planning by states is presented as a search problem on the graph of the path from A to B. Usually the graphs are not set, but generated as needed.

There are blind and directed methods of finding a way. The idea of the blind search method is very simple and obvious. A point is taken at random in the admissible area, and the value of the criterion in it is compared with the current best. If a new random point is worse than the one that is stored as the current best, then take another point. If you find a point at which the criterion is better, it is remembered as the current best. It is guaranteed that in an unlimited increase in the number of attempts we can get closer to the global optimum, ie. the current best value found will be arbitrarily close to the exact solution.

The blind method has two types: search deep and search wide. When searching in depth, each alternative is explored to the end, without considering the other alternatives. The method is bad for "tall" trees, as you can easily slip past the right branch and spend a lot of effort exploring "empty" alternatives.

When searching broadly at a fixed level, all alternatives are explored and only then is the transition to the next level.

The method may be worse than the depth search method if in the graph all the paths leading to the target vertex are located at approximately the same depth. Both blind methods are time consuming and therefore targeted search methods are needed

The method of branches and boundaries. In the process of finding unfinished paths, the shortest one is selected and continues by one step. The obtained new unfinished paths (there are as many of them as there are branches in this vertex) are considered next to the old ones, and the shortest of them continues again by one step. The process is repeated until the first achievement of the target vertex, the decision is remembered. Then from the remaining

unfinished paths are excluded longer than the completed path, or equal to it, and the remaining continue according to the same algorithm as long as their length is less than the completed path. As a result, either all unfinished paths are excluded, or a complete path is formed among them, shorter than previously obtained. The latter path begins to play the role of a standard, etc.

Moore's shortest path algorithm. The output vertex *X0* is denoted by the number 0. Let the set of child vertices *X (x(i)* vertices be obtained during the operation of the algorithm at the current step). Then all previously obtained vertices are deleted from it and marked with a label increased by one compared to the label of the vertex x i, and from them are pointers to *Xi*. Next, on the set of marked vertices, which do not yet appear as pointer addresses, the vertex with the smallest label is selected and child vertices are constructed for it. Vertex markup is repeated until the target vertex is obtained.

Dijkstra's algorithm for determining paths with a minimum cost is a generalization of Moore's algorithm by introducing arcs of variable length.

Dora and Mickey search algorithm with low cost. Used when the cost of the search is high compared to the cost of the optimal solution. In this case, instead of selecting the vertices that are least distant from the beginning, as in Moore's and Dijkstra's algorithms, a vertex is chosen for which the heuristic estimate of the distance to the target is the smallest. With a good score, you can quickly get a solution, but there is no guarantee that the path will be minimal.

Hart, Nilsson and Raphael algorithm. The algorithm combines both criteria: the cost of the path to the vertex g (x) and the cost of the path from the vertex *h (x)* – in the additive function *f {x) = g (x}-h (x). h (x) < hp (x),* where *hp (x)* is the actual distance to the target, the algorithm guarantees finding the optimal path.

Algorithms for finding the path on the graph also differ in the direction of search. There are direct, reverse and bidirectional search methods. Direct search starts from the initial state and is usually used when the target state is specified implicitly. The inverse search starts from the target state and is used when the initial state is set implicitly and the target state is explicit. Bidirectional search requires a satisfactory solution to two problems: changing the direction of the search and optimizing the "meeting point". One of the criteria for solving the first problem is to compare the "width" of the search in both directions – choose the direction that narrows the search. The second problem is caused by the fact that the forward and reverse paths can diverge and the more the search, the more likely it is.

Task. Construct a tree of problem solutions and find an integer solution of the linear programming task using the method of branches and boundaries graphically and using the Find Solutions function in Excel. Compare the results obtained.

Example of execution

Thus, the essence of the method of branches and boundaries is a consistent search of options, considering only those of them that on certain grounds are promising, and rejecting unpromising options. When using the branch and boundary method, the domain of admissible solutions of the initial problem is divided into subsets in a certain way, and subtasks are solved, ie problems on these subsets with the same objective function and without taking into account the integer condition (as linear programming task). If the result is the optimal non-integer solution, the ODR of the subtask is again broken into parts and this process continues until the optimal integer solution of the original problem is found.

For example, if in the problem to the maximum at the decision of subtasks we receive optimum integer decisions those of them which correspond to increasing values of CF are remembered. If the "obtained" solution of the subtask is not better than the stored integer solution, then such a subtask is excluded from the list of tasks. The name of this method is explained by the fact that in the process of solving the problem consistently "branch", breaking into simpler subtasks.

$$Z = 3x_1 + 5x_2 \rightarrow \max$$
$$5x_1 + 2x_2 \leq 14 \qquad (6.1)$$
$$2x_1 + 5x_2 \leq 16$$

x_1 , x_2 – integer, non-negative

Let's construct the range of admissible values, ie we will solve graphically system of inequalities. To do this, construct each line and determine the half-planes, which are given by the inequalities (half-planes are marked by a dash).

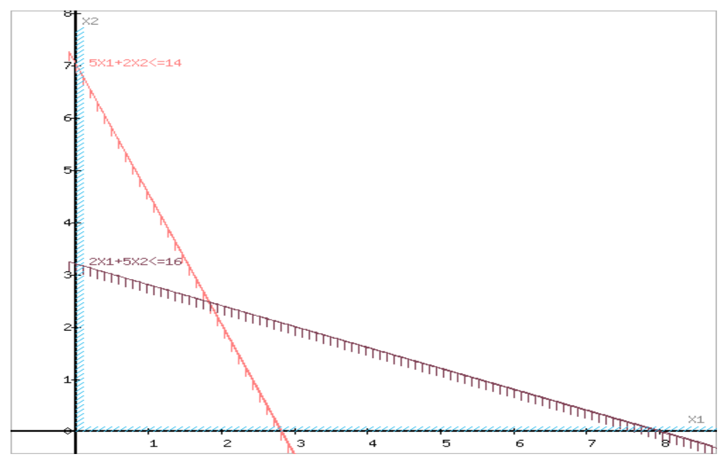


Fig. 6.1. The boundaries of the scope of acceptable solutions

The intersection of half-planes will be an area whose coordinates of points satisfy the condition of inequalities of the system of constraints of the problem.

Let's mark the boundaries of the area of the solution polygon.

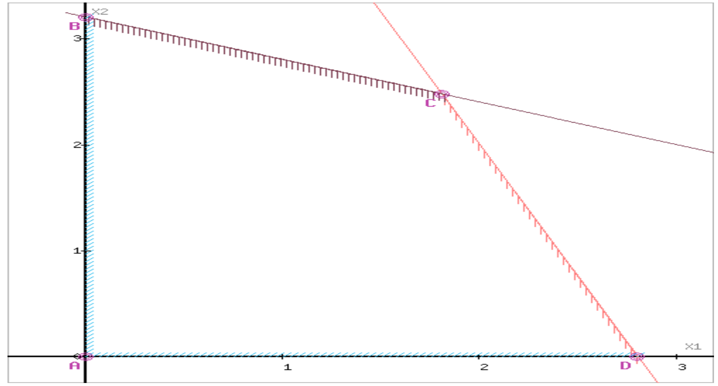


Fig. 6.2. Polygon of solutions of the task 1

Consider the objective function of the problem F = 3x1 + 5x2 → max. Construct a line that corresponds to the value of the function F = 0: F = 3x1 + 5x2 = 0 (construct a gradient vector and a level line). Consider the objective function of the problem $F = 3x_1 + 5x_2 \rightarrow max$.

Construct a line that corresponds to the value of the function F = 0: F = $3x_1 + 5x_2 = 0$ (construct a gradient vector and a level line). We will move the level line parallel to the gradient vector (because we are interested in the maximum), move the line to the last touch of the level line with the allowable area. In the graph, this line is marked by a dotted line.

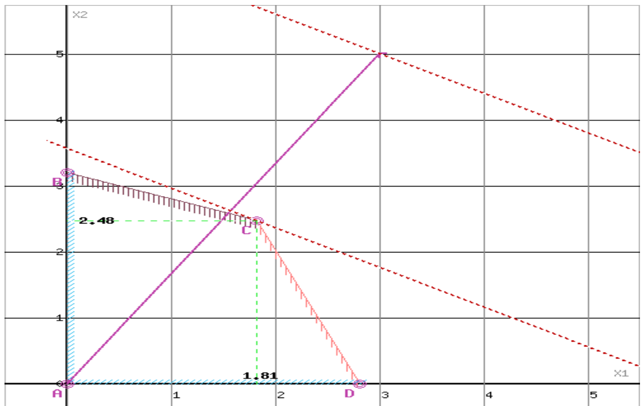


Fig. 6.3. Point C is the optimal point task 1

The area of valid solutions is a polygon. The line F (x) = const intersects the region at the point C. Since the point C is obtained by the intersection of the lines (1) and (2), its coordinates are satisfied by the equation of these lines:

$$5x_1+2x_2\leq 14$$
$$2x_1+5x_2\leq 16$$

Solving the system of equations, we obtain: $x_1 = 1.8095$, $x_2 = 2.4762$. From here we find the maximum value of the objective function:

$$F(X) = 3 * 1.8095 + 5 * 2.4762 = 17.8095$$

The optimal value of the variable $x_1 = 1.81$ was non-integer. Divide problem 1 into two subtasks 11 and 12. In the first of them, the condition $x_1 \geq 2$ is added to the conditions of problem 11, and the condition $x_1 \leq 1$ is added to problem 12. This procedure is called branching by the variable x_1.

Let's solve graphically task 11 as a task of LP.

$$5x_1+2x_2\leq 14 \qquad \textbf{(1)}$$
$$2x_1+5x_2\leq 16 \qquad \textbf{(2)}$$
$$x_1\geq 2 \qquad \textbf{(3)}$$

$$x_1 \geq 0 \qquad \textbf{(4)}$$
$$x_2 \geq 0 \qquad \textbf{(5)}$$

The area of acceptable solutions is a triangle. The line F (x) = const intersects the region at point B. Since point B is obtained by crossing lines (1) and (3), its coordinates satisfy the equation of these lines:

$$5x_1+2x_2 \leq 14$$
$$x_1 \geq 2$$

Solving the system of equations, we obtain: x1 = 2, x2 = 2. From here we find the maximum value of the objective function:

F (X) = 3 * 2 + 5 * 2 = 16

Got an integer solution. The new value of the current record will be equal to F (X) = 16.

Since the point found is the first integer solution, it is necessary to remember the corresponding value of the CF. The point itself is called the current integer record or simply the record, and the optimal value of the integer problem is called the current value of the record. This value is the lower limit of the optimal value of the problem Z*.

Let's solve graphically problem 12 as a task of LP.

$$5x_1+2x_2 \leq 14 \qquad \textbf{(1)}$$
$$2x_1+5x_2 \leq 16 \qquad \textbf{(2)}$$
$$x_1 \leq 1 \qquad \textbf{(3)}$$
$$x_1 \geq 0 \qquad \textbf{(4)}$$
$$x_2 \geq 0 \qquad \textbf{(5)}$$

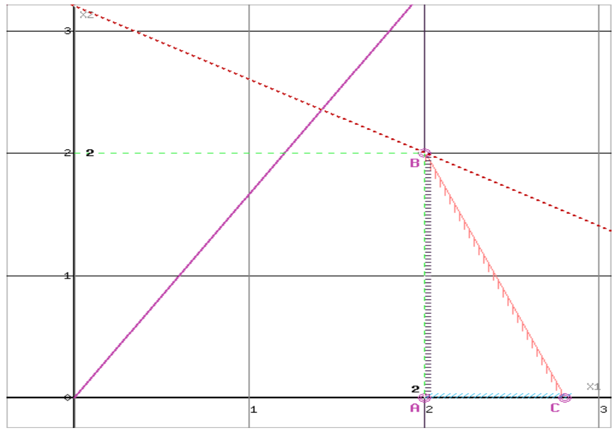


Fig. 6.4. Point B is the optimal point task 11

The area of valid solutions is a polygon. The line F (x) = const intersects the region at the point D. Since the point D is obtained by crossing the lines (2) and (3), its coordinates satisfy the equation of these lines:

$$2x_1+5x_2\leq 16$$
$$x_1\leq 1$$

Solving the system of equations, we obtain: x_1 = 1, x_2 = 2.8. From here we find the maximum value of the objective function:

$$F(X) = 3*1 + 5*2.8 = 17.$$

The optimal value of the variable x_2 = 2.8 was non-integer. Divide problem 12 into two subtasks 121 and 122. In the first of them, the condition $x_2 \geq 3$ is added to the conditions of problem 121, and the condition $x_2 \geq 2$ is added to task 122.

Let us solve graphically problem 121 as a task of LP.

$5x_1+2x_2\leq 14$	**(1)**
$2x_1+5x_2\leq 16$	**(2)**
$0x_1\leq 1$	**(3)**
$x_2\geq 3$	**(4)**
$x_1\geq 0$	**(5)**
$x_2\geq 0$	**(6)**

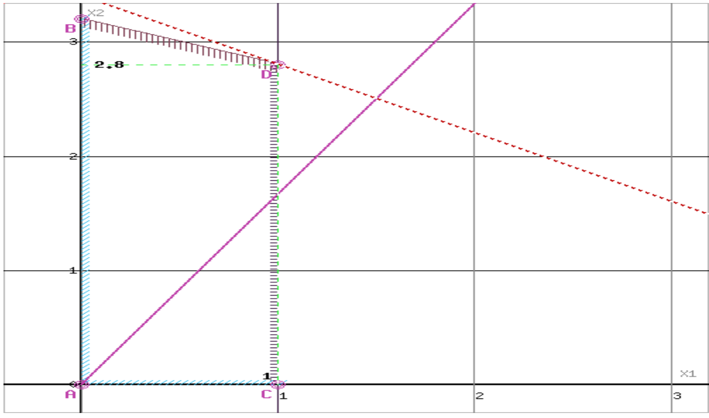


Fig. 6.5. Point D is the optimal point task 12

The area of acceptable solutions is a triangle. The line F (x) = const intersects the region at the point C. Since the point C is obtained by the intersection of lines (2) and (4), its coordinates satisfy the equation of these lines:

$$2x_1+5x_2\leq 16$$
$$x_2\geq 3$$

Solving the system of equations, we obtain: $x_1 = 0.5$, $x_2 = 3$. From here we find the maximum value of the objective function:

$$F(X) = 3 * 0.5 + 5 * 3 = 16.5$$

Let's solve graphically problem 122 as a problem of LP

$$5x_1+2x_2\leq 14 \qquad \mathbf{(1)}$$
$$2x_1+5x_2\leq 16 \qquad \mathbf{(2)}$$
$$x_1\leq 1 \qquad \mathbf{(3)}$$
$$x_2\leq 2 \qquad \mathbf{(4)}$$
$$x_1\geq 0 \qquad \mathbf{(5)}$$
$$x_2\geq 0 \qquad \mathbf{(6)}$$

The area of valid solutions is a polygon. The line F (x) = const intersects the region at the point D. Since the point D is obtained by crossing the lines (3) and (4), its coordinates satisfy the equation of these lines:

$x_1 \le 1$
$x_2 \le 2$

Solving the system of equations, we get: $x_1 = 1$, $x_2 = 2$

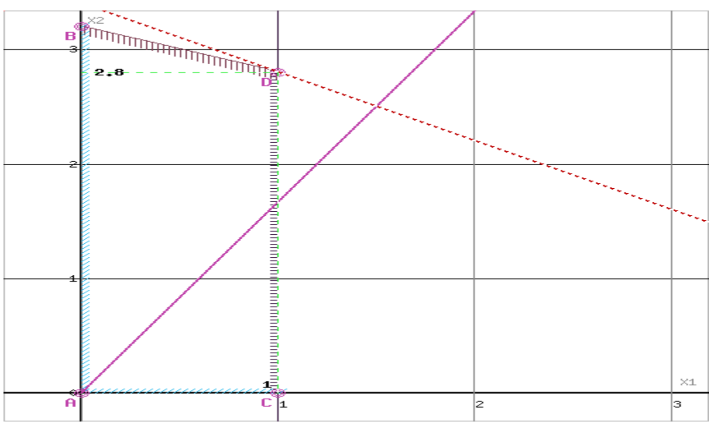


Fig. 6.5. Point D is the optimal point task 12

From here we find the maximum value of the goal function: F(X) = 3*1 + 5*2 = 13.

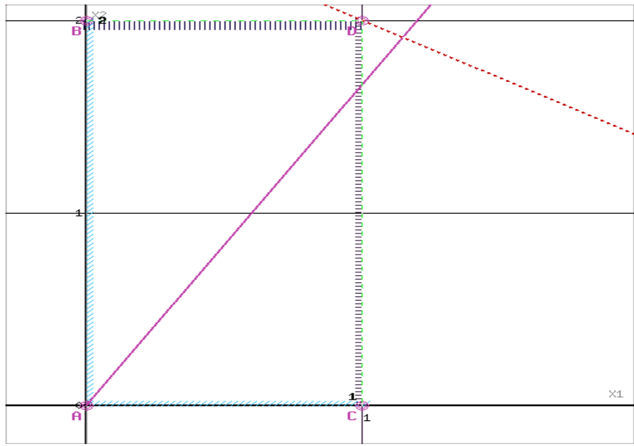


Fig. 6.7. Point D is the optimal point task 122

The current record is Z = 16 ≥13, so we stop branching from this vertex. The optimal value of the variable $x_1 = 0.5$ was incomplete. Divide problem 121

into two subtasks 1211 and 1212. In the first of them, the condition $x_1 \geq 1$ is added to the conditions of problem 1211, and the condition $x_1 = 0$ is added to problem 1212. Let's solve graphically problem 1211 as a problem of LP

$5x_1+2x_2\leq14$ **(1)**
$2x_1+5x_2\leq16$ **(2)**
$x_1\leq1$ **(3)**
$x_2\geq3$ **(4)**
$x_1\geq1$ **(5)**
$x_1\geq0$ **(6)**
$x_2\geq0$ **(7)**

We reduce the system of restrictions to the following type:

$5x_1+2x_2\leq14$ **(1)**
$2x_1+5x_2\leq16$ **(2)**
$x_1=1$ **(3)**
$x_2\geq3$ **(4)**
$x_1\geq0$ **(5)**
$x_2\geq0$ **(6)**

The task has no valid solutions**.** Range of permissible values (RPV) is an empty set.

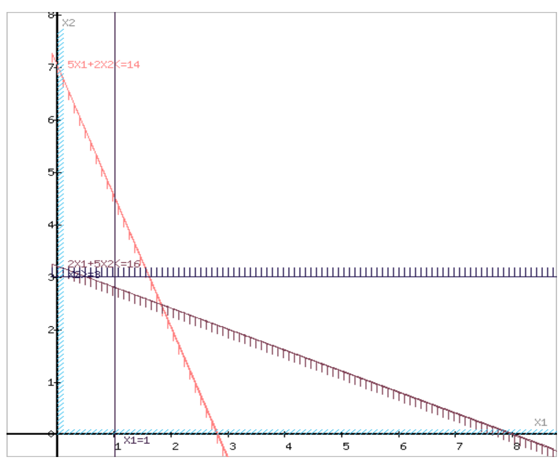


Fig. 6.8. An empty set of solutions to the problem task 1211

Problem 1211 has no solution, so we interrupt the branching process for it. Let's solve graphically problem 1212 as a problem of LP.

$5x_1+2x_2\leq14$	**(1)**
$2x_1+5x_2\leq16$	**(2)**
$x_1\leq1$	**(3)**
$x_2\geq3$	**(4)**
$x_1=0$	**(5)**
$x_1\geq0$	**(6)**
$x_2\geq0$	**(7)**

The area of valid solutions is a polygon. The line F (x) = const intersects the region at the point D. Since the point D is obtained by crossing the lines (2) and (7), its coordinates satisfy the equation of these lines:

$$2x_1+5x_2\leq16$$
$$x_1=0$$

Solving the system of equations, we obtain: $x_1 = 0$, $x_2 = 3.2$. From here we find the maximum value of the objective function:

$$F(X) = 3 * 0 + 5 * 3.2 = 16$$

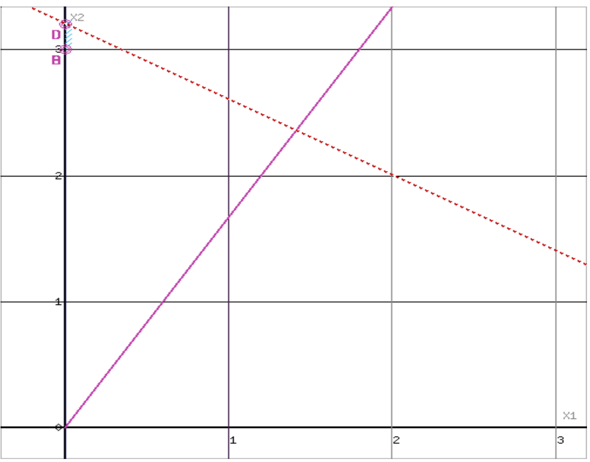


Fig. 6.9. Point D is the optimal point task 1212

The current record is Z = 16 ≥ 16, so we stop branching from this vertex. The optimal value of the variable $x_2 = 2.48$ was non-integer. Divide problem 1 into two subtasks 11 and 12. In the first of them, the condition $x_2 \geq 3$ is added to the conditions of problem 11, and the condition $x_2 \geq 2$ is added to task 12. This procedure is called branching over the variable x_2.

Let's solve graphically task 11 as a task of LP.

$5x_1+2x_2 \leq 14$	**(1)**
$2x_1+5x_2 \leq 16$	**(2)**
$x_2 \geq 3$	**(3)**
$x_1 \geq 0$	**(4)**
$x_2 \geq 0$	**(5)**

The area of acceptable solutions is a triangle. The line F (x) = const intersects the region at the point C. Since the point C is obtained by the intersection of the lines (2) and (3), its coordinates satisfy the equation of these lines:

$$2x_1+5x_2 \leq 16$$
$$x_2 \geq 3$$

Solving the system of equations, we obtain: $x_1 = 0.5$, $x_2 = 3$. From here we find the maximum value of the objective function:

$$F(X) = 3 * 0.5 + 5 * 3 = 16.5$$

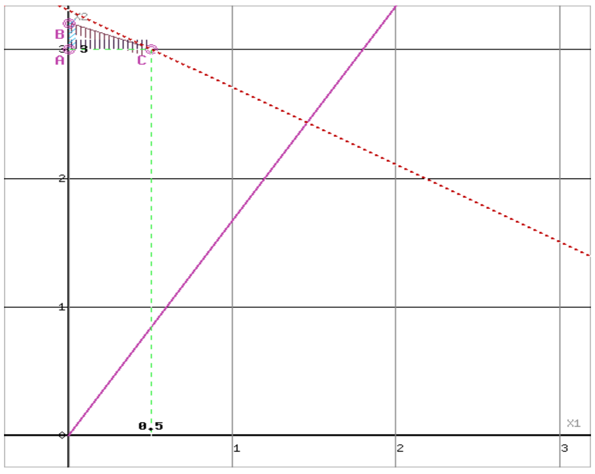


Fig. 6.10. Point C is the optimal point task 1212

Let's solve graphically task 12 as a task of LP

$$5x_1+2x_2\leq 14 \quad \textbf{(1)}$$
$$2x_1+5x_2\leq 16 \quad \textbf{(2)}$$
$$x_2\leq 2 \quad \textbf{(3)}$$
$$x_1\geq 0 \quad \textbf{(4)}$$
$$x_2\geq 0 \quad \textbf{(5)}$$

The line F (x) = const intersects the region at the point C. Since the point C is obtained by the intersection of lines (1) and (3), its coordinates satisfy the equation of these lines:

$$5x_1+2x_2\leq 14$$
$$x_2\leq 2$$

Solving the system of equations, we obtain: x_1 = 2, x_2 = 2. From here we find the maximum value of the objective function:

$$F(X) = 3*2 + 5*2 = 16$$

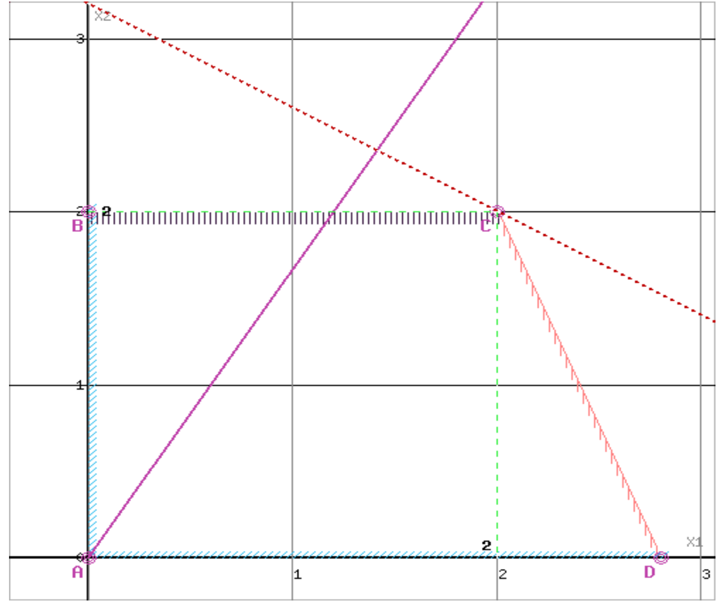


Fig. 6.11. Point C is the optimal point task 12

The current record is Z = 16 ≥16, so we stop branching from this vertex. The optimal value of the variable x_1 = 0.5 was non-integer. Divide problem 11 into two subtasks 111 and 112. In the first of them, the condition $x_1 \geq 1$ is added to the conditions of problem 111, and the condition $x_1 = 0$ is added to problem 112.

Let us solve graphically task 111 as the task of LP

$$5x_1+2x_2\leq14 \qquad \textbf{(1)}$$
$$2x_1+5x_2\leq16 \qquad \textbf{(2)}$$
$$x_2\geq3 \qquad \textbf{(3)}$$
$$x_1\geq1 \qquad \textbf{(4)}$$
$$x_1\geq0 \qquad \textbf{(5)}$$
$$x_2\geq0 \qquad \textbf{(6)}$$

The problem has no valid solutions. PRV is an empty set.

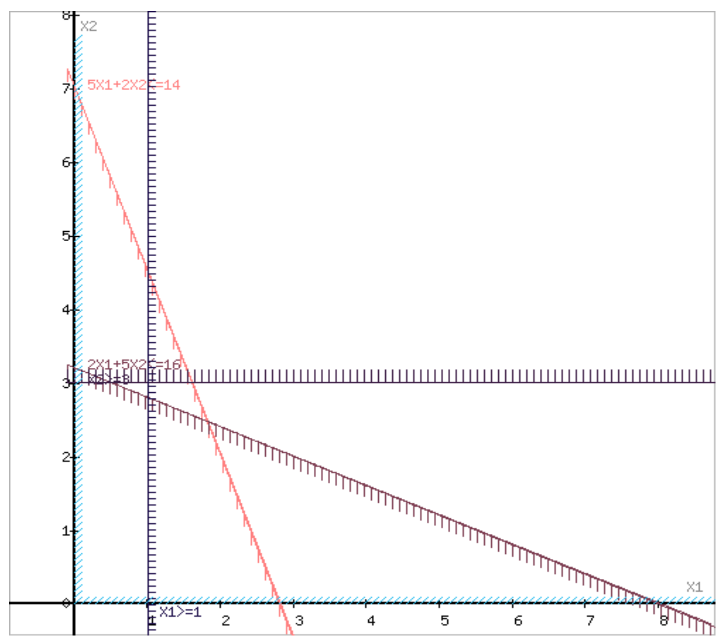


Fig. 6.12. Task 111 has no solution

Task 111 has no solution, so we stop the branching process for it. Let's solve graphically task 112 as a task of LP

$5x_1+2x_2\leq 14$	**(1)**
$2x_1+5x_2\leq 16$	**(2)**
$x_2\geq 3$	**(3)**
$x_1=0$	**(4)**
$x_1\geq 0$	**(5)**
$x_2\geq 0$	**(6)**

The area of valid solutions is a polygon. The line F (x) = const intersects the region at the point D. Since the point D is obtained by crossing the lines (2) and (6), its coordinates satisfy the equation of these lines:

$$2x_1+5x_2\leq 16$$
$$x_1=0$$

Solving the system of equations, we obtain: $x_1 = 0$, $x_2 = 3.2$. From here we find the maximum value of the objective function:

$$F(X) = 3 * 0 + 5 * 3.2 = 16$$

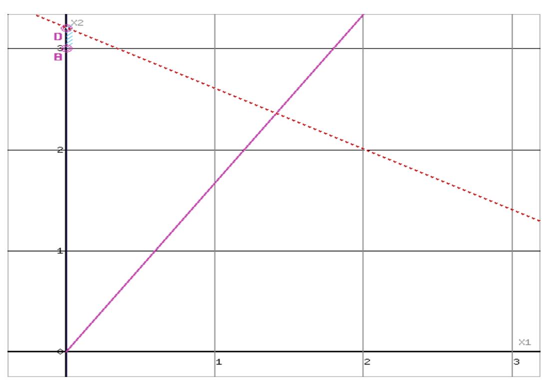


Fig. 6.13. Point D is the optimal point task 112

The current record is Z = 16≥16, so we stop branching from this top.

$$F(X) = 16$$
$$x_1 = 2$$
$$x_2 = 2$$

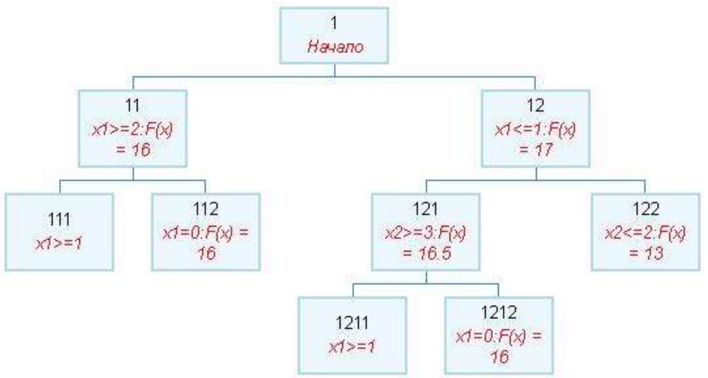


Fig. 6.14. Problem solving tree

Task options

1. $\begin{cases} 8x_1 - 5x_2 \le 16, \\ x_1 + x_2 \ge 1, \\ 2x_1 + 6x_2 \le 9; \end{cases}$
$x_1, x_2 \ge 0;$
$F = 2x_1 + 3x_2 -> \max.$

2. $\begin{cases} 4x_1 - 2x_2 \le 12, \\ -x_1 + 3x_2 \le 6, \\ 2x_1 + 4x_2 \ge 16; \end{cases}$
$x_1, x_2 \ge 0;$
$F = -x_1 + 2x_2 -> \max.$

3. $\begin{cases} 3x_1 + x_2 \ge 10, \\ x_1 + 2x_2 \ge 6, \\ x_1 - x_2 \le 3; \end{cases}$
$x_1, x_2 \ge 0;$
$F = x_1 + x_2 -> \min.$

4. $\begin{cases} x_1 + x_2 \le 14, \\ 3x_1 - 5x_2 \le 15, \\ 5x_1 + 3x_2 \ge 21; \end{cases}$
$x_1, x_2 \ge 0;$
$F = 7x_1 + x_2 -> \max.$

5. $\begin{cases} 2x_1 + x_2 \le 14, \\ -3x_1 + 2x_2 \le 9, \\ 3x_1 + 4x_2 \ge 27; \end{cases}$
$x_1, x_2 \ge 0;$
$F = 3x_1 + 2x_2 -> \max.$

6. $\begin{cases} 2x_1 + x_2 \le 14, \\ -3x_1 + 2x_2 \le 9, \\ 3x_1 + 4x_2 \ge 27; \end{cases}$
$x_1, x_2 \ge 0;$
$F = 3x_1 - 2x_2 -> \max.$

7. $\begin{cases} x_1 - 4x_2 \le 4, \\ -4x_1 + x_2 \le 4, \\ x_1 + x_2 \le 6; \end{cases}$
$x_1, x_2 \ge 0;$
$F = 8x_1 + 2x_2 -> \max.$

8. $\begin{cases} 2x_1 + x_2 \le 11, \\ -3x_1 + 2x_2 \le 10, \\ 3x_1 + 4x_2 \ge 20; \end{cases}$
$x_1, x_2 \ge 0;$
$F = 3x_1 - 2x_2 -> \max.$

9. $\begin{cases} 8x_1 - 4x_2 \le 16, \\ 2x_1 + x_2 \ge 2, \\ 2x_1 + 6x_2 \le 9; \end{cases}$
$x_1, x_2 \ge 0;$
$F = 3x_1 + x_2 -> \max.$

10. $\begin{cases} 3x_1 + 2x_2 \le 6, \\ x_1 - x_2 \ge -1, \\ x_1 + 2x_2 \ge 2; \end{cases}$
$x_1, x_2 \ge 0;$
$F = -x_1 - 2x_2 -> \min.$

11. $\begin{cases} 5x_1 - 2x_2 \le 4, \\ x_1 - 2x_2 \ge -4, \\ x_1 + x_2 \ge 4; \end{cases}$
$x_1, x_2 \ge 0;$
$F = x_1 - 2x_2 -> \max.$

12. $\begin{cases} x_1 + x_2 \ge 2, \\ 2x_1 + x_2 \le 5, \\ 5x_1 - 2x_2 \ge -2{,}5; \end{cases}$
$x_1, x_2 \ge 0;$
$F = x_1 + 3x_2 -> \max.$

13. $\begin{cases} 2x_1 + x_2 \le 11, \\ -3x_1 + 2x_2 \le 10, \\ 3x_1 + 4x_2 \ge 20; \end{cases}$
$x_1, x_2 \ge 0;$
$F = 2x_1 + 3x_2 -> \max.$

14. $\begin{cases} x_1 + x_2 \ge 2, \\ 2x_1 + x_2 \le 5, \\ 5x_1 - 2x_2 \ge 10; \end{cases}$
$x_1, x_2 \ge 0;$
$F = x_1 + 3x_2 -> \max.$

15. $\begin{cases} x_1 + x_2 \ge 9, \\ 2x_1 + x_2 \le 12, \\ 5x_1 - 2x_2 \ge 10; \end{cases}$
$x_1, x_2 \ge 0;$
$F = x_1 + 3x_2 -> \max.$

16. $\begin{cases} -x_1 + 2x_2 \ge 3, \\ 2x_1 + x_2 \ge 3, \\ 2x_1 + 2x_2 \le 5; \end{cases}$
$x_1, x_2 \ge 0;$
$F = 3x_1 + 2x_2 -> \max.$

17. $\begin{cases} -x_1 + x_2 \le 7, \\ 1x_1 + 2x_2 \le 3, \\ 4x_1 + 2x_2 \ge 8; \end{cases}$
$x_1, x_2 \ge 0;$
$F = -x_1 + 2x_2 -> \max.$

18. $\begin{cases} -3x_1 + 2x_2 \le 12, \\ 3x_1 + 4x_2 \ge 9, \\ 2x_1 + x_2 \le 7; \end{cases}$
$x_1, x_2 \ge 0;$
$F = x_1 + 4x_2 -> \max.$

19. $\begin{cases} -3x_1 + 2x_2 \le 3, \\ 3x_1 + 4x_2 \ge 3, \\ 2x_1 + x_2 \le 7; \end{cases}$

$x_1, x_2 \ge 0;$

$F = x_1 + 4x_2 -> \max.$

20. $\begin{cases} -3x_1 + 2x_2 \le 4, \\ 3x_1 + 4x_2 \ge 9, \\ 2x_1 + x_2 <= 7; \end{cases}$

$x_1, x_2 \ge 0;$

$F = x_1 + 4x_2 -> \max.$

21. $\begin{cases} 6x_1 + 3x_2 \ge 8, \\ -2x_1 + 4x_2 \ge 10, \\ x_1 + 3x_2 \le 12; \end{cases}$

$x_1, x_2 \ge 0;$

$F = -x_1 + 2x_2 -> \min.$

22. $\begin{cases} 6x_1 + 3x_2 \ge 12, \\ -2x_1 + 4x_2 \ge 6, \\ x_1 + 3x_2 \le 12; \end{cases}$

$x_1, x_2 \ge 0;$

$F = -x_1 + 2x_2 -> \min.$

23. $\begin{cases} 6x_1 + 3x_2 \ge 4, \\ -2x_1 + 4x_2 \ge 7, \\ x_1 + 3x_2 \le 12; \end{cases}$

$x_1, x_2 \ge 0;$

$F = -x_1 + 2x_2 -> \min.$

24. $\begin{cases} -6x_1 + 4x_2 \le 20, \\ 3x_1 + 4x_2 \ge 20, \\ 2x_1 + x_2 \le 11; \end{cases}$

$x_1, x_2 \ge 0;$

$F = -2x_1 - 3x_2 -> \min.$

25. $\begin{cases} -3x_1 + 2x_2 \le 9, \\ 3x_1 + 4x_2 \ge 24, \\ 2x_1 + x_2 \le 14; \end{cases}$

$x_1, x_2 \ge 0;$

$F = x_1 + 4x_2 -> \max.$

6.2. List of control questions on the task № 1

1. Statement of the problem of mathematical programming ZMP.
2. Transition from ZMP to ZLP.
3. What is a valid area and how to build it?
4. What is a gradient vector?
5. How to find an integer solution of ZLP with the help of "Solution Search"?
6. Describe the branches of the decision tree.
7. What is a task space?
8. What is the idea of blind search? Types of blind search.
9. Describe the method of hylogues and boundaries.
10. What algorithms do you know to teach paths on a graph?

6.3. Task № 2. Construction of autoregressive models in Eviews

Objective: To gain skills in building regression models in the Eviews package and learn to evaluate their parameters

Part 1. Create a working file in Eviews and import data.

Create a working file where the series will be generated. To do this, perform the following actions: File, New, Workfile. In the window

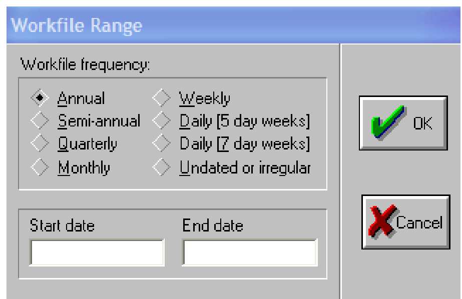


Fig. 6.15. Workfile Range window

turn on Undate or irregular and set the number of the first (Start observation) and last (End observation) observation, for example,

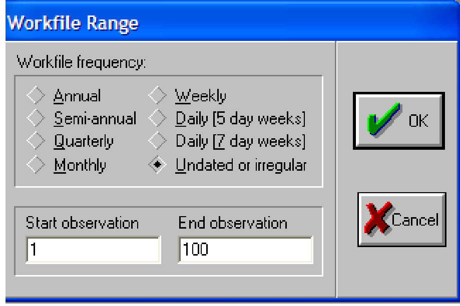


Fig. 6.16. Providing the number of the first and last observation

As a result, we received an empty workbook, in which you can download a number of 100 observations.

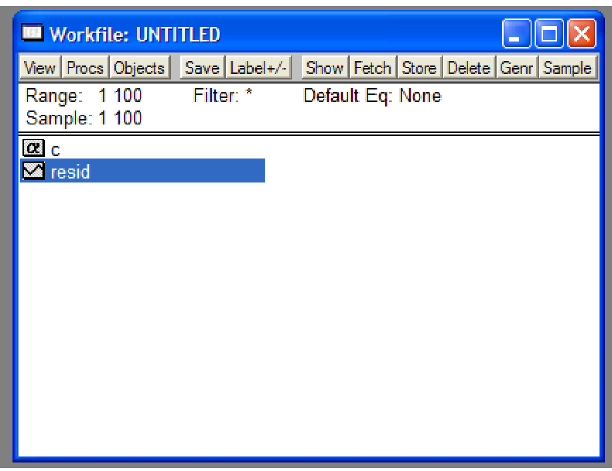


Fig. 6.17. Blank workbook window

To enter data from the file, follow these steps:

File, Import, Read Text-Lotus-Excel, Specify the path where the row file is

Located

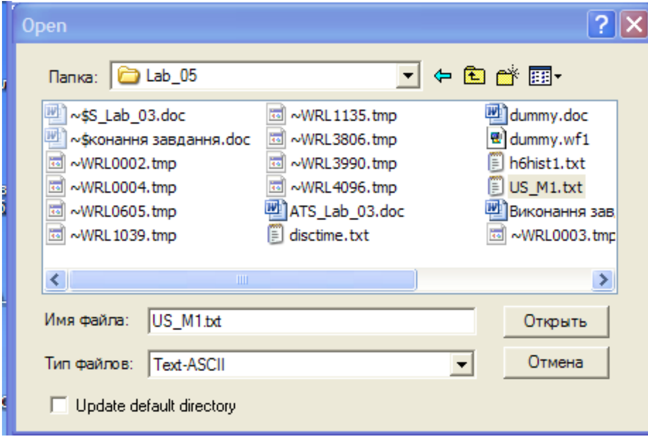


Fig. 6.18. Path to the data file

Activate US_M1.txt and click OPEN. We get a window in which Specify the name of the row in the Name to series tab.

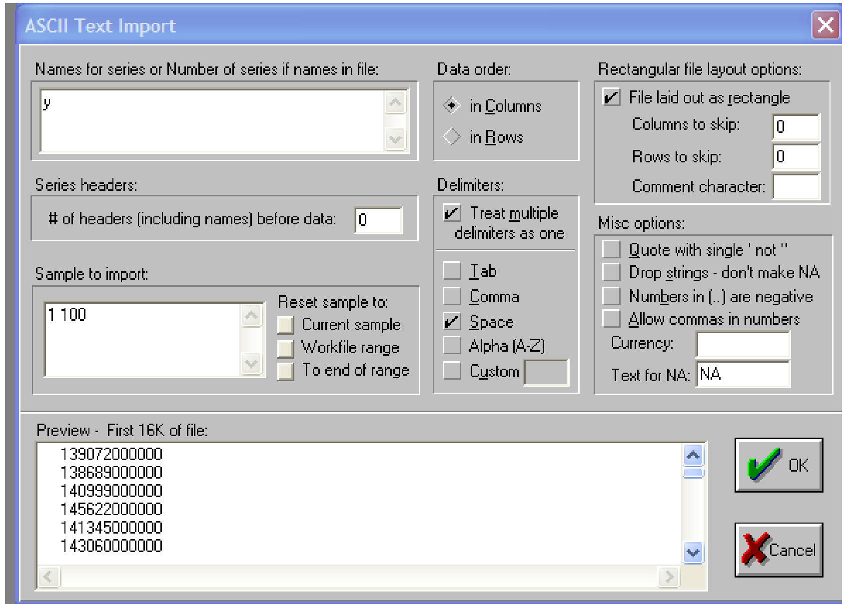


Fig. 6.19. Import data file

2. Creating a difference equation of autoregression with a moving average in Eviews.

1. The difference equation of autoregression with moving average (ARMA) has the form:

$$y(k) = c_1 y(k-1) + c_2 y(k-2) + c_3 e(k-1) + e(k), \quad (6.2)$$

where $y(k)$ – the dependent variable; $e(k)$ – input process variable, generate the value of the dependent variable. The type of equation (6.2) is abbreviated as ARMA (2,1) (ie, the second order autoregression and the first order relative to the moving average). Choose the coefficients of equation (6.2) so that it is a stationary process. This requires that the necessary condition is met

$$\sum_{i=1}^{n} c_i < 1, \quad \text{and sufficient} \quad \sum_{i=1}^{n} |c_i| < 1.$$

To generate an input sequence $e(k)$, you can use the standard function of the Eviews package - the NRND (normally distributed random numbers) operator of the Eviews package. When we have the value of a random variable,

the values of the dependent variable are calculated using equation (6.2). To do this, use the Generate Series option. Execution of point 3: We create a new workbook.

We generate. Quick, General Series, e = nrnd.

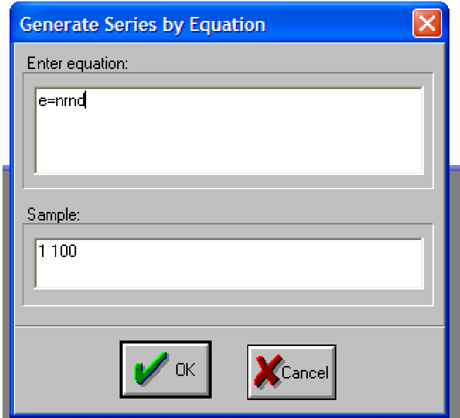


Fig. 6.20. Input sequence generation

- Select the coefficients c1, c2, c3. Their sum must be less than 1. Let c1 = 0.1, c2 = 0.2, c3 = 0.3.
- Generate a second-order process. To do this, first generate an empty row. Quick, General Series

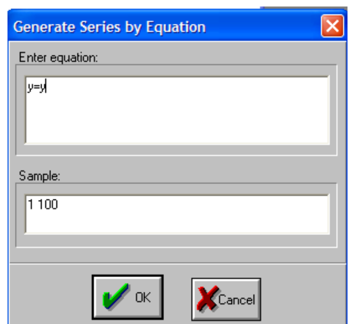


Fig. 6.21. Generation of an empty row

We generate y = y. Opening the generated row by double-clicking, we see a row in all cells whose value is NA, you need to edit the created row by

filling the first cell, for example, enter the number 1. To do this, right-click on the row with merged cells, and in In the drop-down box, click Edit.

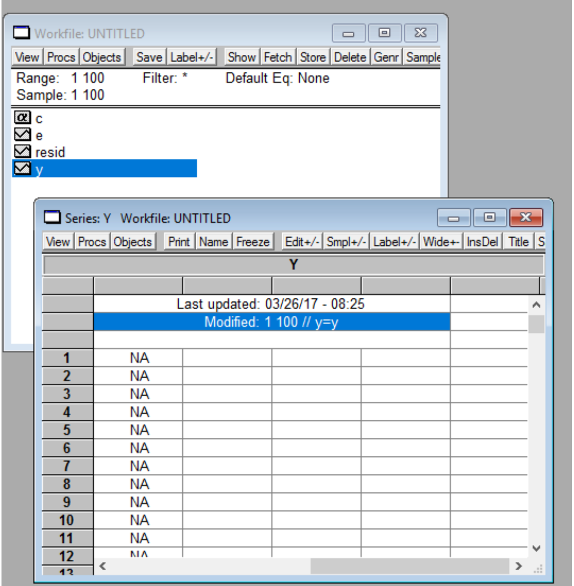


Fig. 6.22. Generation of NA series

Next, activate the cell (1,1) and write there 1.

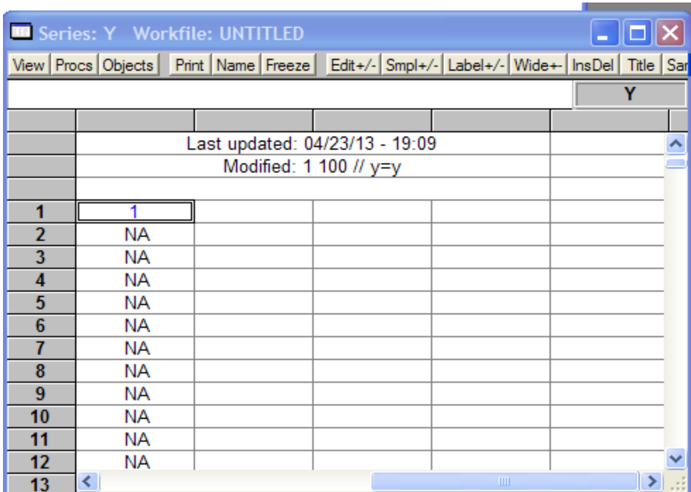


Рис. 6.23. Editing a series *NA*

Fill the first two cells with the numbers 1 and 2. Then perform the following actions: Quick, Generate Series, Generate in:

$$y = 0.1 * y(-1) + 0.2 * y(-2) + 0.3 * e(-1) + e.$$

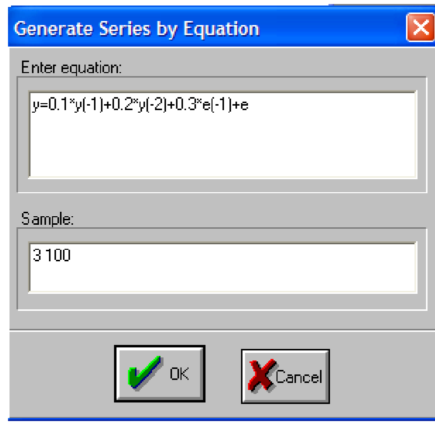


Fig. 6.24. Generation of the difference equation

Thus, we constructed a series that corresponds to model (6.2).

4. It is necessary to calculate the coefficients of the model (6.2).

Execution: Objects / New Object / Equation

In the Equation Specification window, specify the type of equation that describes a series of data:

$$y = c(1) * y(-1) + c(2) * y(-2) + c(3) * e(-1) + e \quad (6.2)$$

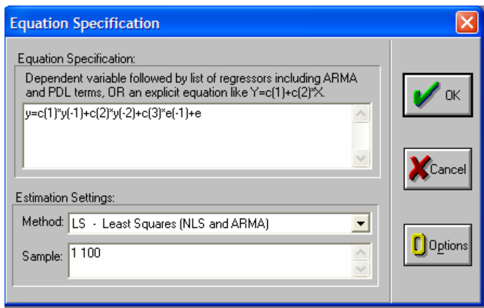
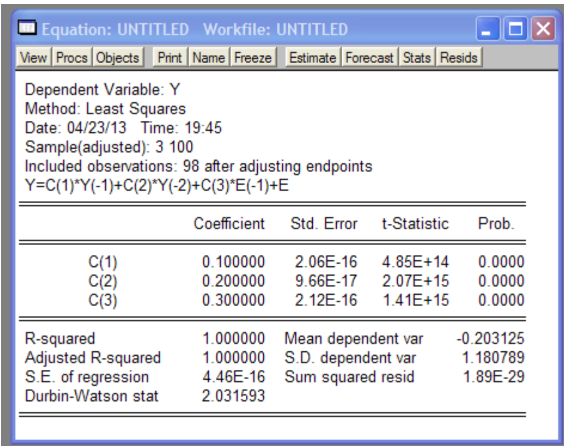


Fig. 6.25. Equation for calculating the coefficients of the model

This equation exactly matches the form used in the source data generation program. As a result, a new Equation window will appear: UNTITLED Workfile: UNTITLED, in which you will find estimates of the coefficients of the equation and related statistical characteristics.



Equation: UNTITLED Workfile: UNTITLED

View | Procs | Objects | Print | Name | Freeze | Estimate | Forecast | Stats | Resids

Dependent Variable: Y
Method: Least Squares
Date: 04/23/13 Time: 19:45
Sample(adjusted): 3 100
Included observations: 98 after adjusting endpoints
Y=C(1)*Y(-1)+C(2)*Y(-2)+C(3)*E(-1)+E

	Coefficient	Std. Error	t-Statistic	Prob.
C(1)	0.100000	2.06E-16	4.85E+14	0.0000
C(2)	0.200000	9.66E-17	2.07E+15	0.0000
C(3)	0.300000	2.12E-16	1.41E+15	0.0000

R-squared	1.000000	Mean dependent var	-0.203125
Adjusted R-squared	1.000000	S.D. dependent var	1.180789
S.E. of regression	4.46E-16	Sum squared resid	1.89E-29
Durbin-Watson stat	2.031593		

Fig. 6.26. Estimates of the coefficients of the equation and related statistical characteristics of the model

To save this window, you must activate Name and specify a name. Compare the obtained values of the coefficient estimates with the exact values that were used in generating the data sample.

Building a model of non-stationary process

A) Use the data stored in the US_M1.txt file to build a non-stationary process model. This file contains data for the M1 aggregate from the first quarter of 1960 to the fourth quarter of 1991 for the United States (128 values). Use the Eviews package to organize your work file and enter data from disk. Data processing is performed in the following sequence:

- plot a graph of the introduced series and visually determine the presence of nonstationarity (trend);
- from the graph we conclude that the series has a tendency to rise, so the expectation is growing, so the series has a trend, so it is non-stationary.

B) Find the equation that describes the trend:

$$\log(us) = a_0 + a_1 \cdot k + a_2 \cdot k^2 + \varepsilon(k), \qquad (6.3)$$

where k – discrete time (it is in the file *discrtime.txt*; $\varepsilon(k) = resid$, ie the remainder obtained after estimating equation (6.3), this value will be used later,

so form an additional series $e(k)$ of values *resid* (mark it, for example *ek*). The equation for estimating the trend of prologarithmic values of the series has the form:

$$LOG(us) = c(1) + c(2) * k + c(3) * k^2 ,$$

where $a_0 = c(1)$, $a_1 = c(2)$, $a_2 = c(3)$ values (discrete time) must be taken from the file *disctime.txt* (link: https://drive.google.com/drive/folders/1cDvuOS1rpAhd8nUZuDOAJEH-Z9N1Mkjt?usp=sharing).

How adequate is the obtained model?

Procedure:

1. Create a new workbook for 128 members of the series.
2. Import the series US_M1.txt (link: https://drive.google.com/drive/folders/1cDvuOS1rpAhd8nUZuDOAJEH-Z9N1Mkjt?usp=sharing).
3. Build a graph of the series US (to build a graph, you must open the series and perform the actions Viev, Line Graph.) The visual graph can visually determine the presence of nonlinearities.
4. Calculate and print the parameters of descriptive statistics. To find the parameters of descriptive statistics, perform the following actions: View, Descriptive Statistic, Histogram and Stats (make a report).
5. Review the correlogram (ACF) – Autocorrelation and PC – Partial Correlation. To do this, open the row, View tab, Corelogram.
6. Import the number k (time) (disctime file).

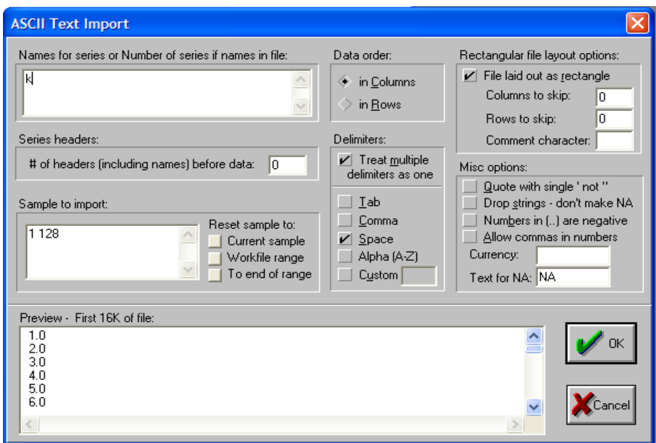


Fig. 6.27. Import series k (time)

1. Normalize the series *US_M1.txt*. by logarithm. Generate a series lm1 = log (us) – (Quick, General Series).

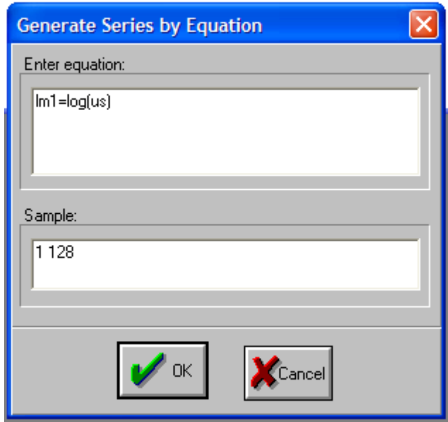
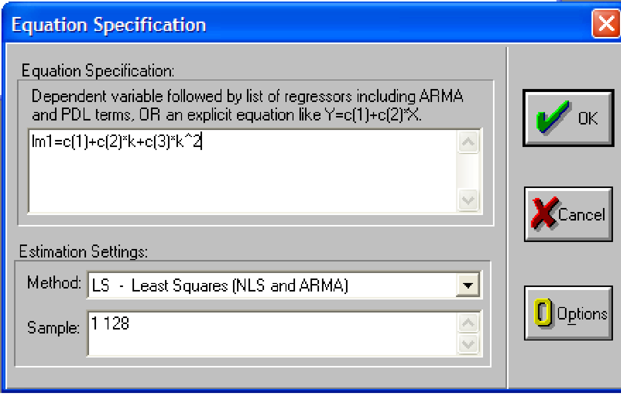
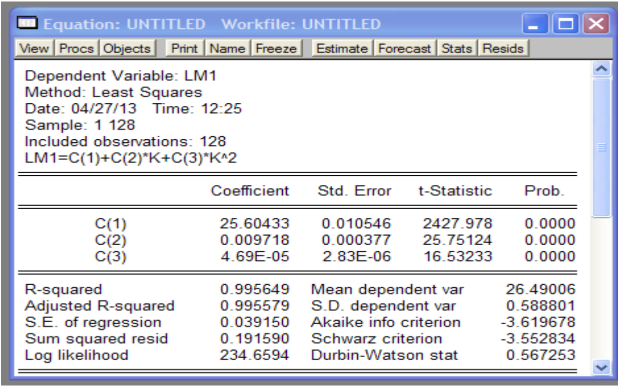


Fig. 6.28. Logarithmization of the series US_M1.txt

2. Model the trend using a polynomial of degree 2: Quick, Estimate Equation lm1=c(1)+c(2)*k+c(3)*k^2.





Equation: UNTITLED Workfile: UNTITLED

Dependent Variable: LM1
Method: Least Squares
Date: 04/27/13 Time: 12:25
Sample: 1 128
Included observations: 128
LM1=C(1)+C(2)*K+C(3)*K^2

	Coefficient	Std. Error	t-Statistic	Prob.
C(1)	25.60433	0.010546	2427.978	0.0000
C(2)	0.009718	0.000377	25.75124	0.0000
C(3)	4.69E-05	2.83E-06	16.53233	0.0000

R-squared	0.995649	Mean dependent var	26.49006
Adjusted R-squared	0.995579	S.D. dependent var	0.588801
S.E. of regression	0.039150	Akaike info criterion	-3.619678
Sum squared resid	0.191590	Schwarz criterion	-3.552834
Log likelihood	234.6594	Durbin-Watson stat	0.567253

Fig. 6.29. Trend modeling and statistical characteristics and estimates of the lm1 model

In the last window we can see the estimated coefficients of the model and a set of statistical characteristics of this model. Characteristic *R*-squared (coefficient of determination).

In order to preserve the comfort of the latest model, correct resid in ek (right mouse button on the file resid, object copy). We conclude that the model is adequate for *R* ^ 2 – the coefficient of determination (for an adequate model it is close to 1).

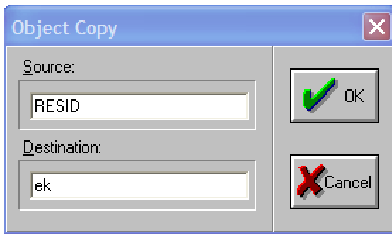


Fig. 6.30. Save model remnants

Part 2. Calculation of ACF and PC.

The main characteristics of the time series are mathematical expectation, variance, autocorrelation function, partial autocorrelation function. The time

series is given, where is the number of observations of the series. It is necessary to find selective estimates of autocorrelations. Find the ACF of a given series.

The autocorrelation coefficient is calculated by the formula:

$$r_k = \frac{c_k}{c_0}, \tag{6.4}$$

де $c_k = \frac{1}{N-1}\sum_{t=1}^{N-k}(z_t - \bar{z})(z_{t+k} - \bar{z})$, where k - delay, shift, lag, delay.

Suppose $k = 0$, that is, it is necessary to calculate c_0, then the formula (6.4) матиме will look like:

$$c_0 = \frac{1}{N-1}\sum_{i=1}^{N}(z_t - \bar{z})(z_t - \bar{z}) = \frac{1}{N-1}\sum_{t=1}^{N}(z_t - \bar{z})^2 \tag{6.5}$$

The first-order autocorrelation coefficient is calculated by the formula:

$$k = 1:\ c_1 = \frac{\frac{1}{N-1}\sum_{t=1}^{9}(z_t - \bar{z})(z_2 - \bar{z})}{\frac{1}{N-1}\sum_{t=1}^{10}\overline{(z_t - z)^2}} \tag{6.6}$$

The second-order autocorrelation coefficient is calculated by the formula:

$$k = 2:\ c_2 = \frac{\frac{1}{N-1}\sum_{t=1}^{8}(z_t - \bar{z})(z_3 - \bar{z})}{\frac{1}{N-1}\sum_{t=1`}^{10}(z_t - \bar{z})^2}. \tag{6.7}$$

and so on.

In practice, to obtain a useful estimate of ACF, we need at least 50 observations and sample autocorrelations should be calculated for k = 1, 2,… K, where K <= N / 4.

To clarify the order of the autoregressive component allows partial autocorrelation function (PC), which is calculated according to the formulas:

$$\Phi_{11} = \rho(1),\ \Phi_{22} = \frac{\rho_2 - \rho_1^2}{1-\rho_1^2}, \ldots,\ \Phi_{ss} = \frac{\rho_s - \sum_{j=1}^{s-1}\Phi_{s-1,j}\rho_{s-j}}{1 - \sum_{j=1}^{s-1}\Phi_{s-1,j}\rho_j} \tag{6.8}$$

where ρ_i – the value of the autocorrelation function

$$\Phi_{s,j} = \Phi_{s\text{-}1,j} - \Phi_{s,s}\Phi_{s\text{-}1,s\text{-}j}, \tag{6.9}$$

s – ordinal number of the coefficient, $\boldsymbol{j} = s - 1$.

Let’s calculate

$$\Phi_{33} = \frac{\rho_3 - (\Phi_{21}\rho_2 + \Phi_{22}\rho_1)}{1 - (\Phi_{21}\rho_1 + \Phi_{22}\rho_2)} \tag{6.10}$$

$(s = 2, j = 1) \Rightarrow \Phi_{21} = \Phi_{11} - \Phi_{22}\Phi_{11}$.

Let's calculate

$$\Phi_{44} = \frac{\rho_4 - (\Phi_{31}\rho_3 + \Phi_{32}\rho_2 + \Phi_{33}\rho_1)}{1 - (\Phi_{31}\rho_1 + \Phi_{32}\rho_2 + \Phi_{33}\rho_3)} \tag{6.11}$$

Need to find Φ_{31} та Φ_{32}. For Φ_{31} s=3, j=1.

$$\Phi_{31} = \Phi_{21} - \Phi_{33}\Phi_{22}. \tag{6.12}$$

For Φ_{32} s=3, j=2.

$$\Phi_{32} = \Phi_{22} - \Phi_{33}\Phi_{21} \quad . \tag{6.13}$$

PC more accurately reflects the order of the AR model due to the lack of influence of intermediate correlation coefficients on selected values of the variable, ie, the coefficient characterizes the degree of relationship between adjacent (in time) variable values and characterizes the relationship between variable values distant at two periods sampling. It is considered expedient to consider the coefficients of PC and those numbers of their values that stand out among others (is the largest in modulus) and they are candidates for inclusion in the delay numbers of the autoregressive model.

Task*:* Calculate the ACF and PC for a given series, compare the results with the results of the calculation of these indicators in Evies.

Task options:

1. 47; 65; 22; 70; 37; 64; 55; 41; 55; 34.
2. 44; 60; 22; 59; 37; 55; 41; 59;70; 48.
3. 59; 44;71;35;57;40;58;44;80;55.
4. 48; 71; 35; 57; 40; 58; 44; 80; 55; 70.
5. 71; 35; 57; 40; 58; 64; 71; 55; 74; 50; 58.
6. 35; 57; 40; 58; 50; 71; 56; 74; 60; 44.
7. 44; 80; 55; 26; 37; 74; 51; 57; 50; 66.
8. 50; 71; 56; 74; 50; 58; 45; 54; 36; 53.
9. 62; 44; 43; 52; 38; 59; 55; 41; 53; 49.
10. 44; 43; 52; 38; 59; 55; 41; 53; 49; 56.
11. 56; 74; 50; 58; 45; 54; 36; 53; 59;44.
12. 50; 58; 45; 54; 36; 53; 59;44; 70; 67.
13. 52; 38; 59; 55; 41; 53; 49; 56; 45; 57.

14. 55; 26; 37; 74; 51; 57; 50; 66; 35; 67.
15. 26; 37; 74; 51; 57; 50; 66; 35; 67; 26.
16. 71; 56; 74; 50; 58; 45; 54; 36; 53; 45.
17. 56; 74; 50; 58; 45; 54; 36; 53; 45; 50.
18. 55; 41; 53; 38; 58; 41; 53; 49; 35; 57.
19. 53; 38; 58; 41; 53; 49; 35; 57; 68; 60.
20. 38; 58; 41; 53; 49; 35; 57; 68; 60; 57.
21. 58; 41; 53; 49; 35; 57; 68; 60; 57; 73.
22. 41; 53; 49; 35; 57; 68; 60; 57; 73; 40.
23. 53; 49; 35; 57; 68; 60; 57; 73; 40; 56.
24. 49; 35; 57; 68; 60; 57; 73; 40; 56; 70.
25. 5; 57; 68; 60; 57; 73; 40; 56; 70; 51.

6.4. List of control questions on the task № 2

1. How to obtain estimates of the coefficients of the equation of the type AR or ARKS using the package Eviews?
2. Why logarithm the values of the time series?
3. What is the difference between stationary and non-stationary processes?
4. Define a stationary process?
5. Why is the process represented by the data in the file US_M1.txt non-stationary?
6. Tell us what is the sequence of building a model of non-stationary process?
7. What are the first and higher order differences used for?

6.5. Task № 3. Determining the best autoregressive model and building a forecast based on the selected model

Types of time series models.

Depending on the nature of the behavior of these coefficients are divided into the following processes:

- ***AR (k)*** – (autoregressive model) – autoregressive model of order ***k***;
- ***MA (m)*** – moving average – moving average of order ***m***;
- ***ARMA (k, m)*** – autoregression model of sliding mean order ***k***;
- ***ARIMA (k, d, m)*** – integrated model of autoregression-moving average, ***k***-order of autoregression, ***m*** – order of moving average, ***d*** – order of integration.

Autoregression model of order *k* - *AR (k)*.

Let there be a time series $y_1, y_2, \ldots, y_n$, or $t = 1,2,\ldots,n$ where y_t is the current level value.

The basic assumption is that the current value of the series equation is a linear combination of k previous values and a random error.

General autoregression model:

$$y_t = \alpha_1 y_{t-1} + \alpha_2 y_{t-2} + \ldots + \alpha_k y_{t-k} + \varepsilon_t,$$

where $k \le t$ i $\alpha_1,\ldots,\alpha_k$ – model parameters or model coefficients – random error or "white noise". When building the AR model, it is necessary to solve two problems: what order of the model should be chosen and why the coefficients of the model are equal. Addressing these issues is called a model evaluation procedure.

The most accurate estimates of the coefficients for the model can be obtained *AR(1)*, *AR(2)*.

Model AR(1)

View $y_t = \alpha_1 y_{t-1} + \varepsilon_t$.

Model AR(2)

View $y_t = \alpha_1 y_{t-1} + \alpha_2 y_{t-2} + \varepsilon_t$.

Models of moving average *MA (m)*

These models are based on the assumption that the current value of the level of the series is represented as a linear combination of current and past error values, ie, $y_t = \varepsilon_t + \beta_1 \varepsilon_{t-1} - \beta_2 \varepsilon_{t-2} - \ldots - \beta_m \varepsilon_{t-m}$, where β_i – **model parameters**, ε_t – **white noise**, $m \le t$ – **order of the model**.

MA(1)

View $y_t = \varepsilon_t - \beta \varepsilon_{t-1}$.

Moving average order models 1 can be used only to describe the process with ACF, which ends after the first delay and such that $r < 0{,}5$, where *r* is the sample autocorrelation coefficient.

Models *ARMA* (*p*, *q*) or *ARIMA* (*p*, *d*, *q*)

Generalization of the ARMA (p, q) or ARIMA (p, d, q) model – autoregressive models of the moving average ARMA (p, q) = ARIMA (p, 0, q)

These models are based on the assumption that the current level of a series is a linear combination of p of its previous levels and q of its previous errors. When identifying the ARMA model (*p, q*) use the fact that their autocorrelation functions fade smoothly along the exponent or sine wave.

General view of the model:

$$y_t = \alpha_1 y_{t-1} + \alpha_2 y_{t-2} + \ldots + \alpha_p y_{t-p} + \varepsilon_t - \beta_1 \varepsilon_{t-1} - \beta_2 \varepsilon_{t-2} - \ldots - \beta_q \varepsilon_{t-q}$$

Methods of integration.

1. Taking the final differences. Let the input row $y_1, y_2, \ldots, y_n$ not be stationary. Let's build a series $x_1, x_2, \ldots, x_{n-1}$ where $x_t = y_t - y_{t-1}$. If this series satisfies the stationary condition $I(1)$s, then the input series $y_1, y_2, \ldots, y_n$ is denoted, and it is concluded that the input series is close to linear.

Otherwise go to the series $z_1, z_2, \ldots, z_{n-1}$ where $z_t = x_t - x_{t-1} = y_y - y_{t-1} - (y_{t-1} - y_{t-2}) = y_t - 2y_{t-1} + y_{t-2}$. It is similarly denoted, and it is concluded that this series is close to quadratic.

2. Logarithm of chain indices, ie. Occurs if the time series y_t is close to the exponent $y_t \approx a_0 e^{-a_1 t}$.

The integration procedure has to be applied quite often, but after that in the stationary series we can talk about the constancy of the second-order autocorrelation coefficients.

2. Procedure for work

2.1. Import a series from Excel according to the option in the Evies environment.

2.2. Find ACF and ACF for a given series.

2.3. From the analysis of CHAKF to define numbers of delays which can be used at modeling of a number.

2.4. Build models and analyze statistical characteristics, determine the best model that describes the process. To do this, use the formula

$$KK = e^{1-R^2} + \frac{RSS}{N} + \begin{cases} \ln(AIC + BSC), & AIC + BSC > 0 \\ e^{AIC+BSC}, & AIC + BSC \le 0 \end{cases} + e^{2-DW} + e^{U}. \quad (6.14)$$

For the best QC model the least.

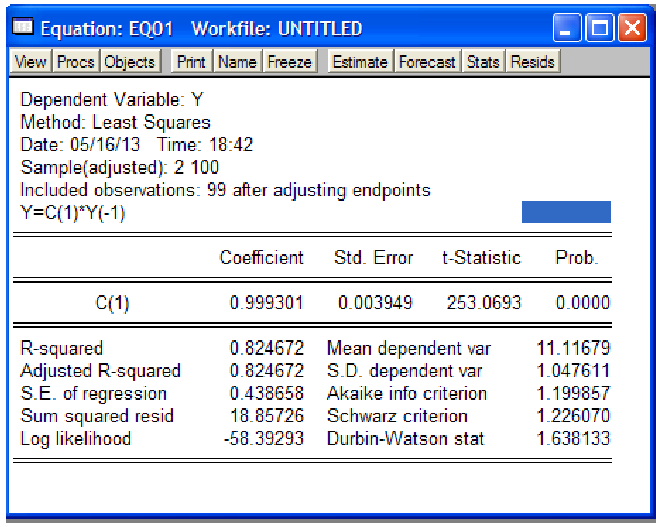
2.5. Determine the adequacy of the model.

2.6. Draw conclusions.

Explanation:

For example, in the analysis of CHAKF it was found that the most significant numbers of delays – 1, 2, 5. We build the following models: Quick, Estimate equation

```
y=c(1)*y(-1)
y=c(1)*y(-1)+c(2)*y(-2)
y=c(1)*y(-1)+c(2)*y(-2)+c(3)*y(-5)
```



Equation: EQ01 Workfile: UNTITLED

View | Procs | Objects | Print | Name | Freeze | Estimate | Forecast | Stats | Resids

Dependent Variable: Y
Method: Least Squares
Date: 05/16/13 Time: 18:42
Sample(adjusted): 2 100
Included observations: 99 after adjusting endpoints
Y=C(1)*Y(-1)

	Coefficient	Std. Error	t-Statistic	Prob.
C(1)	0.999301	0.003949	253.0693	0.0000

R-squared	0.824672	Mean dependent var	11.11679
Adjusted R-squared	0.824672	S.D. dependent var	1.047611
S.E. of regression	0.438658	Akaike info criterion	1.199857
Sum squared resid	18.85726	Schwarz criterion	1.226070
Log likelihood	-58.39293	Durbin-Watson stat	1.638133

Fig. 6.31. Statistical characteristics of the first-order autoregressive model

To get the Tayle factor U in the window click Forecast and in the window

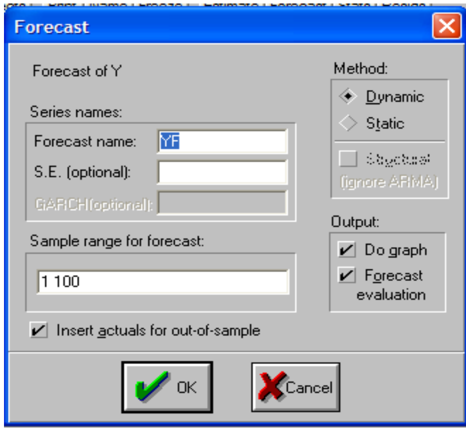


Fig. 6.32. Forecast window

click OK. We get a window

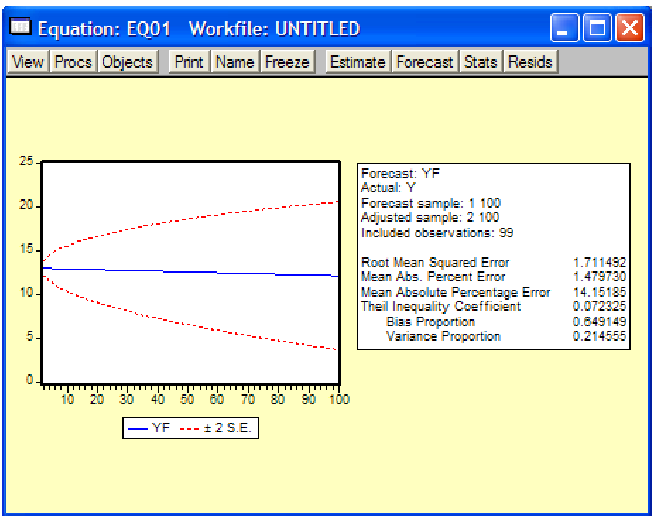


Fig. 6.33. Statistical indicators of the forecast

We find the indicator Theil Inequality Coefficient = 0.0723, so u = 0.0723. (The closer the Tayle ratio is to 0, the more accurate the prediction).

6.6. List of control questions on the task № 3

1. How to build a model AR ()?
2. What is the order of the model?
3. How to build a model MA ()?
4. In which cases is the ARIMA () model built?
5. To predict which series are used the above models?
6. What is the autocorrelation function?
7. What is a partial autocorrelation function?
8. What is lag, delay?
9. How to build an autoregressive model based on the analysis of ACF and CHAKF?

6.7. Task № 4 Topic: Starting, getting started and basics of Visual Prolog

Launch and get started in the visual environment of Visual Prolog application development. To start Visual Prolog, follow these steps: Start →

Programs → Visual Prolog 5.2 → Vip32. This opens the main window, called the Task window (Fig. 6.34).

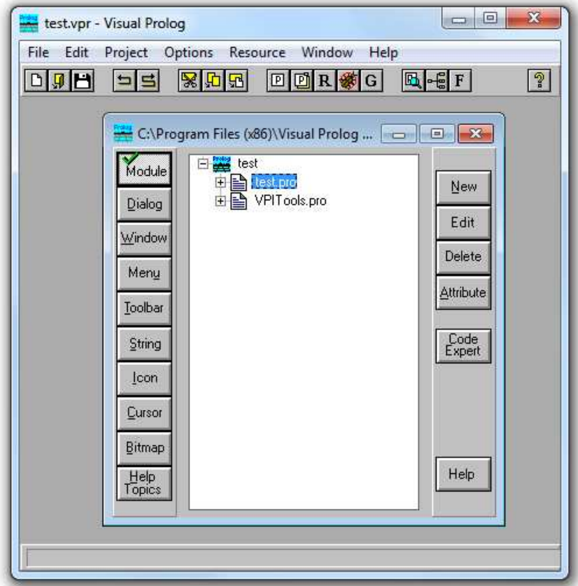


Fig. 6.34. Task window

The File, Edit, Project, Options, Help, and Window menus are usually available in the Task window, but additional menu items may appear when you activate some other windows. Frequently used menu commands are implemented on the toolbar in the form of buttons (Fig. 6.35).

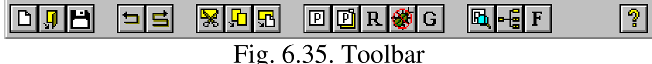


Fig. 6.35. Toolbar

The correspondence of the buttons located on the toolbar to the menu commands is given in table. 6.1.

Table 6.1. Toolbar buttons and corresponding menu commands

Toolbar buttons	Menu commands
	File → New
	File → Open

	File → Save
	Edit → Undo
	Edit → Redo
	Edit → Cut
	Edit → Copy
	Edit → Paste
	Project (Compile file)
	Project → Build
	Project → Run
	Project → Debug
	Project → Test Goal
	Project → Browse
	Project → Tree
	Options → Temporary → Font
	Help → Local Help

At the bottom of the Task window is a tooltip. It is divided into two parts (Fig. 6.36). The left field is used to display context-sensitive information, such as tooltip tooltips on the toolbar or information about the current control in the dialog editor, and so on. The right field is used by the builder to display the states of generation, compilation, layout of the current resource.

Fig. 6.36. Hint bar

Creating a project. To create a new project, you need to select some Visual Prolog compiler options. To do this, follow these steps:

1. Start the Visual Prolog visual development environment (Start → Programs → Visual Prolog 5.2 → Vip32). The first time you start the screen, the window shown in Fig. 1.4. You will also be notified that the initialization file for Visual Prolog VDE (Visual Develop Environment) has been created by default.

2. Create a new project. To do this, select the command Project → New Project, the result is activated dialog Arrlication Expert (Fig. 1.4).

3. Specify the base catalog and project name. In the Field Name field, enter the name Test, then left-click in the Name field. VRP File. After clicking, in the Name field of. The VRP File should appear: test.vpr. You also need to check the box for Multiprogrammer Mode and left-click in the Name of RPY File field. Then the project file name Test.prj will appear there (Fig. 6.37).

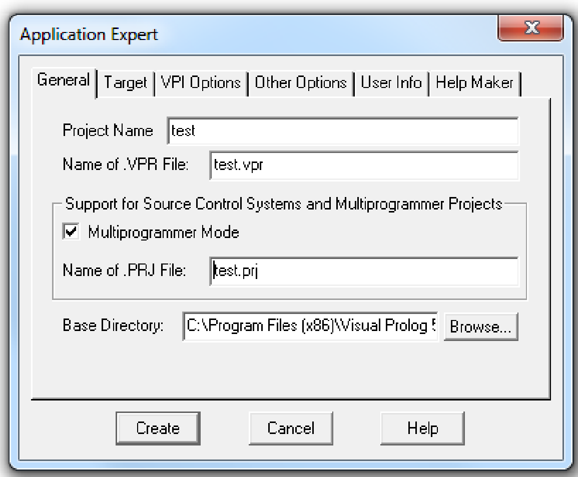


Fig. 6.37. General settings of the Expansion dialog box

1. Define the purpose of the project. On the Target tab, it is recommended to select the parameters shown in Fig. 6.38. and click the Create button to create the project files.

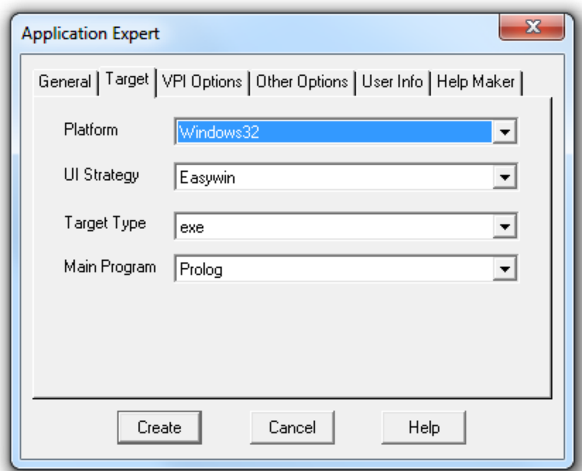


Fig. 6.38. Options on the Target tab of the Application Expert

2. Set the necessary compiler parameters for the created project. To activate the Compressor Options dialog box, select Options → Project → Compressor Options. Then go to the Warnings tab. To set the necessary parameters, perform the following steps:

- set the Nondeterm switch. This is required so that the Visual Program compiler accepts by default that all user-defined predicates are nondeterministic (may have more than one solution);

- uncheck the following options: Not Quoted Symbols, Strong Type Conversion Check and Check Type of Predicates. These steps will reduce the number of warnings from the compiler;

- click OK to save the compiler options settings.

As a result of the above steps, the Compiled Options dialog box will have shown in Fig. 6.39 view.

Running and testing the program. To verify that the system is configured properly, follow these steps:

1. In the project window (Fig. 6.34,6.40) double-click the left mouse button to open the file test.pro.

2. In the goal section, type the following from the keyboard instead of test (): write ("Hello world!"), Nl.

3. Press the button on the toolbar (either <Ctrl> + <g>, or activate the Project → Test Goal command). In Prolog terminology, it is called GOAL and this is enough for the program to run. If your system is set up properly, a Hello world window will appear on the monitor screen! under which it will be written

yes (Fig. 6.41). To proceed to testing other GOALs, you must close the program results window.

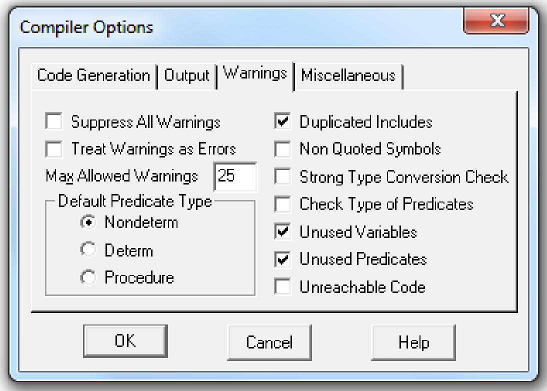


Fig. 6.39. Compiler options

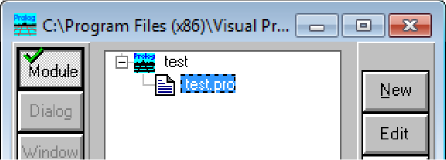


Fig. 6.40. Project window

Comments on the properties of the Test Goal utility. The visual development utility interprets GOAL as a special program that is compiled, compiled, generated into a file, and Test Goal runs it. This utility internally extends the specified GOAL code so that the generated program finds all possible solutions and shows the values of all variables. The Test Goal utility compiles this code using the compiler options set for the open project (the recommended compiler options for the TestGoal project we defined earlier). It should be noted that the Test Goal utility compiles only the code that is defined in the active editor window (code in other open editors or project modules, if any, is ignored). The EASYWIN strategy is used when composing a Test Goal file. It is not possible to define any layout options for Test Goal, as any Make Options set for the open project are ignored. Therefore, Test Goal cannot use any global predicates defined in other modules. The utility has a limit on the number of variables that can be used in GOAL. There are 12 for a 32-bit visual

development environment, but this number may vary depending on the version of Visual Prolog.

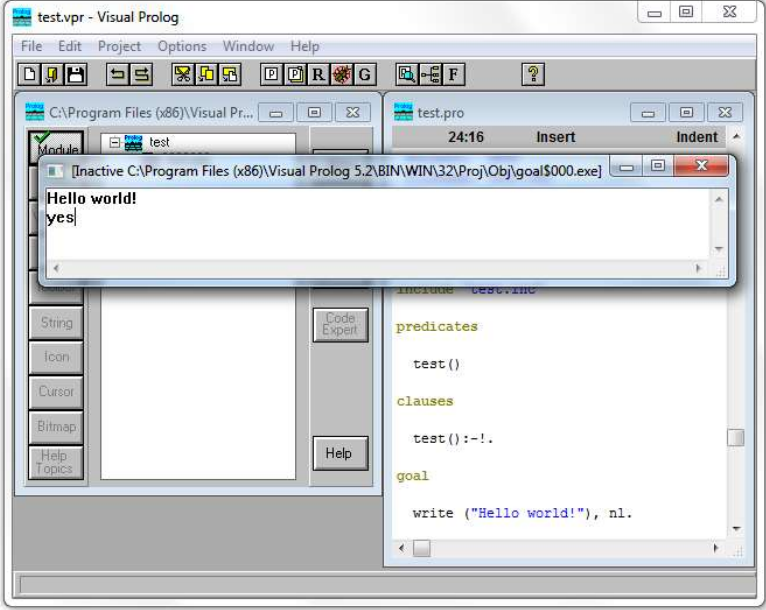


Fig. 6.41. Hello world test program

Error handling. If there are errors in the program, then during compilation the visual development environment will display the Errors (Warnings) window, which will contain a list of error messages. If you double-click on one of the messages, the environment will move the cursor to the line of code in which the corresponding error in the source code. You can use the <F1> key to display the Visual Prolog online help system. When the help window opens, you need to click on the Search button, dial the error number, and the corresponding help window will appear on the screen with more complete information about the error.

Construction command. Project → Compile Module command. This command (corresponding to <Ctrl> + <F9>) attempts to compile the module contained in the currently edited file. Execution of the command depends on the following properties of the file:

- if the file has the extension pro and is a module of the current project, then VDE tries to compile this file;

- if the file is not a module of the current project and its extension – pro, pre, inc, con or dom, then VDE tries to find the project module that includes this file and compile the first module found;

- in all other cases, VDE tries to compile the module selected in the project window.

VDE cannot compile a file that is not part of an open project. Instead, the VDE file will compile the module selected in the project window. If no project is open in VDE, no files will be compiled. The Project → Compile Module menu command is locked and the <Ctrl> + <F9> key combination does not work. The only possible action is to run the Test Goal utility.

Project → Build command. If any resources have changed since the last project build, code experts can update some sections in the source files before building. This command (corresponding to <Alt> + <F9>) builds the project by checking the timestamps of all source files in the project, so if the source files (or files included in them) are newer than the dependent OBJ files, then the relevant project modules will be recompiled.

The Build command also builds resource files and a help file (if needed). The project is then composed to generate a target module (program or DLL).

Project → Rebuild All command. This command (corresponding to <Ctrl> + <Alt> + <F9>) performs the same action as Project → Build, and all files will be re-generated or compiled and compiled regardless of their timestamps.

Project → Stop Building command. This command (corresponding to <Alt> + <F10>) is used to stop compiling / composing.

The Project → Run command. If necessary, this command (corresponding to the <F9> key) will perform the action Project → Build and then run the generated file.

Project → Link Only command. This command (corresponding to <Shift> + <F9>) is used to perform the layout. In this case, the program builder calls the compiler and does not check whether any project modules need to be recompiled (or even compiled for the first time).

Project → Test Goal command. This command (corresponding to <Ctrl> + <G>) is used to test Goals. The program is compiled and composed in a special mode, and then run the appropriate file. The Test Goal utility searches for all solutions for the purpose defined in the program. For each solution, the Test Goal displays the values of all variables in the GOAL section and the number of solutions. This feature is a convenient way to check local predicates in the module.

For example, the following goal: GOAL X = 2; X = 1, Y = X+1. For example, the following goal:

Resource → Build Resource Only command. When the project window is activated, the Resource command appears in the Project menu. Selecting this

item (or pressing <Alt> + <F8>) generates the selected files with the extensions rc and res and the required constant files.

Example 6.1.

1. Create a new project. Specify the base directory and project name (lab1ex1). Define the purpose of the project. Set the necessary compiler parameters for the created project.
2. In the file with the extension lab1ex1.pro enter the following program:

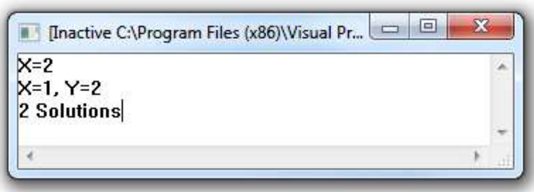


Fig. 6.42. Derivation of the target testing mode

```
predicates
        likes (symbol, symbol)
clauses
        likes (ellen, tennis).
        likes (john, football).
        likes (tom, baseball).
        likes (eric, swimming).
        likes (mark, tennis).
        likes (bill, Activity):- likes (tom, Activity).
goal
        likes (bill, baseball).
```

3. Run the program with the Test Goal utility. The Test Goal utility will answer in the program window: yes.

4. Try the following query in the GOAL section: likes (bill, tennis). The Test Goal utility must answer: no.

Debug commands.

Project → Debug command. Starts the debugging process. You can also start the debugger by pressing **<Strl> + <Shift> + <F9>.**

You can use the View dialog to open additional information windows that display different environment states and debugging variables:

- view → Call Stack - opens the call stack information window;

- **view → Local Variables** – opens the information window of local variables.

The following commands are used to perform debugging steps:

- **run → Trace Intro** (or key **<F7>);**
- **run → Step Over** (or the **<F8> key**);
- **run → Run to Cursor** (or the **<F4> key**).

Basic principles, mechanisms and features of programming in the Prolog language. In Prolog (Programming Logic) the solution of the problem is obtained by a logical conclusion from previously known statements. Usually, a program in the Prologue language is not a sequence of actions, it is a set of facts with rules that provide conclusions based on these facts. Therefore, the Prologue is known as a declarative language. The prologue is based on Horn's sentences, which is a subset of the formal system called predicate logic. The prologue includes a derivation mechanism that is based on matching samples. By selecting answers to queries, he receives known information, ie knowledge of the Prologue about the world - is a limited set of facts (and rules) set in the program. One of the most important features of the Prologue is that, in addition to the logical search for answers to questions, it can deal with alternatives and find all possible solutions. Instead of the usual work from the beginning of the program to its end, the Prologue can go back and review more than one "path" in solving all components of the problem. The programmer on the Prologue describes objects and relations, and then describes the rules under which these relations are true.

The main sections of Visual Prolog programs. Typically, a program on Visual Prolog consists of four main program sections. These include:

- **clauses** section;
- section **predicate**s;
- section **domains (domains);**
- section **Goal (goals)**

Clauses. The clauses section contains all the facts and rules that make up the program. Facts are attitudes or properties that are known to have the meaning of "Truth." Fact represents either a property of an object or a relationship between objects. The fact is self-sufficient. No additional information is required to confirm the fact, the fact can be used as a basis for a logical conclusion.

The fact in Visual Prolog consists of the name of the relationship and the object or objects placed in parentheses. The fact ends with a dot (). That is, the sentence in the natural language "John loves football" (John likes football), in the syntax of Visual Prolog will look like (john, football).

Facts, in addition to relationships, can also express properties. For example, the sentence in natural language "Grass is green" on the Visual Prolog, expressing the same properties, looks like this: green (grass).

Rules are related relationships, they allow you to logically deduce one piece of information from another. The rule takes the value "True, if it is proved that the specified set of conditions is true. A rule is a property or relationship that is valid when a number of other relationships are known to be valid. Syntactically, these relationships are separated by commas. All rules have 2 parts: title and body, separated by a special sign ": -".

The title is a fact that would be true if several conditions were true. This is called a conclusion or dependent relationship.

The body is a set of conditions that must be true in order to prove that the title of the rule is true. The following is a generalized rule syntax in Visual Prolog:

title: - <subpurpose>, <subpurpose>, ..., <subpurpose>.

The body of a rule consists of one or more sub-goals. Sub-goals are separated by commas, defining the conjunction, and after the last sub-goal of the rule a full stop is placed. Each subgoal calls another Visual Prolog predicate, which can be true or false. After the program has made this call, Visual Prolog checks the truth of the called predicate, and if it is true, the work continues, but with the following sub-goal. If in the process of such work a point was reached, then the whole rule is considered true, if at least one of the sub-goals is wrong, then the whole rule is wrong. The following are the rules that correspond to the "likes" link:

(Ann likes everything that likes John).

(Kate likes everything that is green).

Using these rules, you can find out from the previous facts some things that Anya and Katya love:

(Ann likes football).

(Kate likes grass).

To translate these rules into Prolog, you need to change the syntax as follows:

likes (ann, Something):- likes (john, Something).

likes (kate, Something):- green (Something).

The symbol ": -" is equivalent to "if". However, if in the Prologue is different from if written in other languages, such as Pascal, where the condition contained in the if statement must be specified before the body of the statement that can be executed. This type of operator is known as a conditional if / then operator.

Visual Prolog uses a different form of logic in such rules. The conclusion about the truth of the title of the Prolog rule is made if (after) the body of this rule is true, ie the Prologue rule corresponds to the conditional form then / if.

All sentences for each particular predicate in the clauses section must be placed together. A sequence of sentences describing one predicate is called a procedure.

The rule can be considered as a procedure. In other words, the following rules:

likes (ann, Something):- likes (john, Something).

likes (kate, Something):- green (Something).

also means "To prove that Anya loves something, prove that John loves it" and "To prove that Katya loves something, prove that it is something green."

Predicates section. If any predicate is described in the clauses section of the program on Visual Prolog, it must be declared in the predicates section, otherwise Visual Prolog will not understand what you are "telling" it. As a result of declaring a predicate, the programmer reports which domains (types) the arguments of this predicate belong to. Visual Prolog comes with a large set of built-in predicates (no need to announce them), and an online help guide provides a complete description.

Predicates set facts and rules. In the predicates section, all predicates are simply listed with the types (domains) of their arguments. Declaring a predicate begins with the name of that predicate, followed by a parenthesis ("followed by zero or more domains (types) of predicate arguments, followed by a comma after each domain (type) of the argument, and closing the parenthesis") "after the last argument type). . Example:

predicates name (argument_type1 OptionalName1, argument_type2 OptionalName2,...,argument_typeN OptionalNameN)

It should be noted that, unlike the sentences in the clauses section, the predicate declaration does not end with a period. The domains (types) of

predicate arguments can be either standard domains or domains declared in the domains section.

The name of the predicate must begin with a letter, which can be followed by a sequence of letters, numbers and underscores. The case is irrelevant, but capital letters should not be used as the first letter of the predicate name. The predicate name can be up to 250 characters long.

It is forbidden to use a space, minus sign, asterisk and other alphanumeric characters in the names of predicates. Predicate arguments must belong to domains known to Visual Prolog. These domains can be either standard domains or some of those declared in the domains section. That is, if the predicate my_predicate (symbol, integer) is declared in the predicates section as follows:

```
predicates
        my_predicate (symbol, integer),
```

it is not necessary to declare domains of its arguments in the domains section because symbol and integer are standard domains. However, if the same predicate is declared as follows:

```
predicates
        my_predicate (name, number),
```

then you must declare that name (character type) and number (integer type) belong to the standard domains symbol and integer:

```
domains
        name = symbol
        number = integer
predicates
        my_predicate (name, number).
```

Domains section. In the traditional Prologue there is only one type - therm. In Visual Prolog, you must declare the domains of all predicate arguments. Domains allow you to specify different names for different types of data, which, otherwise, will look exactly the same. In Visual Prolog programs, objects in relations (predicate arguments) belong to domains, and these can be both standard (Table 6.2) and manually described special domains..

Table 6.2. Basic standard domains

Domain	Description	Realization
short	Short, symbolic, quantitative	All platforms 16 біт (from -32768 to 32767)

ushort	Short, unsigned, quantitative	All platforms 32 біт (from 0 to 65535)
long	Long, significant, quantitative	All platforms 32 біт (from -2147483648 to 2147483647)
ulong	Long, unsigned, quantitative	All platforms 32 біт (from 0 to 4294967295)
integer	Significant, quantitative, has platform-dependent size	All platforms 16 біт (from -32768 to 32767) platforms 32 біт (from -2147483648 to 2147483647)
unsigned	Unsigned, quantitative, has platform-dependent size	Platforms 16 біт (from 0 to 65535) Platforms 32 біт (from 0 to 4294967295)
byte		All platforms 8 біт (from 0 to 55)
word		All platforms 16 біт (from 0 to 65535)
dword		All platforms 32 біт (from 0 to 4294967295)

Domains of types **byte, word and dword** are most convenient when working with machine numbers. Integer and unsigned types are mainly used, as well as short and long (and their unsigned analogues) for more specialized applications.

In domain ads, the keywords signed and unsigned can be used in conjunction with standard byte, word, and dword domains to build new base domains. For example:

```
domains
        i8 = signed byte
```

creates a new base domain in the range of -128 to +127. Other base domains are shown in table.6.3.

Table 6.3. Basic standard domains

Domain	Description and implementation
char	Character implemented as an unsigned byte. Syntactically, it is a symbol, placed between two single paws: 'a'
real	Floating point number implemented as 8 bytes according to the IEEE agreement; equivalent to the double type in the C language. Valid range of numbers: from 1x10-307 to 1x10 + 308 (from 1e-307 to 1e + 308). If necessary, goals are automatically converted to real

string	A sequence of characters that is implemented as a pointer to a byte array ending in zero, as in C. Two strings are allowed for strings: 1. A sequence of letters, numbers, and underscores, with the first character being a lowercase letter. 2. A sequence of characters enclosed in double quotes. The lines written in the program can reach a length of 255 characters, while the lines that Visual Prolog reads from a file or builds within itself can reach (theoretically) up to 4 GB on 32-bit platforms
symbol	A sequence of characters implemented as a pointer to an input in an ID table that stores strings of IDs. Syntax - as for strings

The domains section serves two useful purposes. First, it is possible to give domains meaningful names, even if internally these domains are similar to existing standard ones. Second, special domain declarations are used to describe data structures that are missing from standard domains.

Sometimes it is very useful to describe a new domain - especially when there is a need to clarify parts of the predicates section. Declaring one's own domains, by assigning meaningful names to argument types, helps to document predicates.

An example of how domain declarations help document predicates is:

Frank is a 45-year-old man.

Using the following domains, you can declare the corresponding predicate:

```
domains
        name, sex = symbol
        age = integer
predicates
        person (name, sex, age).
```

Goal section. This section is similar to the body of the rule, ie contains a list of sub-goals. The goal differs from the rule as follows:

- the keyword goal is not followed by ": -";

- Automatically executes the target when you start Visual Prolog.

This is as if Visual Prolog calls a goal, thus running a program that tries to solve the body of the goal rule. If all sub-goals in the goal section are true, the program completes successfully. If any sub-goal from the goal section is wrong, it is considered that the program is not completed successfully (although outwardly there is no difference in these cases, the program will simply terminate its work).

Example. Having given Visual Prolog a few facts at once, it is possible to ask questions about the relationship between them. In natural language, you can ask the question: Does Bill like football? (Bill loves football?). According to the rules of the Prologue, this question is as follows: likes (Bill, football). Upon receiving such a request, Visual Prolog will answer: yes, because Visual Prolog has a fact that confirms that it is.

Other sections of programs. Facts section. The program on **Visual Prolog** is a set of facts and rules. Sometimes in the process of the program there is a need to modify (change, delete or add) some of the facts with which it works. In this case, the facts are considered as a dynamic or internal database, which may change during the execution of the program. To announce the facts of the program, which are considered part of a dynamic (or variable) database, Visual Prolog includes a special section – facts.

The facts keyword announces the facts section. It is in this section that it is possible to announce the facts that are included in the dynamic database. Visual Prolog has several built-in predicates that make it easier to use dynamic facts**.**

Constants section. Visual Prolog programs can declare and use symbolic constants. The section for declaring constants is denoted by the keyword constants, followed by the declarations themselves, which use the following syntax:

<Id> = < Macrodefinitions >

where **<Id>** is the name of the symbolic constant, and **<macrodefinition>** is the value assigned to this constant. Each **<macrodefinition>** ends with a newline character and, therefore, there can be only one description of a constant in one line. The constants declared in this way can later be used in programs.

Example:

```
constants
        zero = 0
        pi = 3.141592653
```

Before compiling, Visual Prolog will replace each constant with the appropriate string. The following restrictions are imposed on the use of symbolic constants:

- the description of the constant cannot refer to itself:

my_number = 2*my_number/2

such a record is not allowed, its use will result in an error message "Recursion in constant definition" (recursion in the constant description);

- in the descriptions of constants, the system does not distinguish between upper and lower case. Therefore, when using a constants identifier in a clauses section, its first letter must be lowercase to avoid confusion between constants and variables;
- the program may have several sections of constants, but the declaration of the constant must be made before using it;
- Constant IDs are global and can only be declared once. Multiple declaration of the same identifier will result in the error message "Constant identifier can only be declared once" (constant identifier can be declared only once).

Global sections. Visual Prolog allows you to declare some domains, predicates, clauses global (not local). This can be done by announcing special sections in the program global domains, global predicates and global facts.

Variables. In Visual Prolog, variables allow you to write down general facts and rules and ask general questions. Variable names in Visual Prolog must begin with a capital letter (or underscore), followed by any number of letters (uppercase or lowercase), numbers, or underscores. It is convenient to use letters of different case in the name of a variable. For example: IncomeAndExpenditureAccount**.**

In a simple query, you can use variables to find "someone who loves tennis". For example: likes (X, tennis). In this query, the letter "X" is used as a variable to find an unknown person.

Meaningful choice of variable names makes the program more readable. For example, a name:

```
likes (Person, tennis).
```

better than

```
likes (X, tennis).
```

тому що ім’я «Person» має більше сенсу, ніж «X».

The English sentence "Bill likes the same thing as Kim" can be written on the Visual Prolog as follows:

```
likes (bill, Thing):- likes (kim, Thing).
```

where Thing is a variable.

If you only need certain query information, you can use anonymous variables to ignore unnecessary values. In Visual Prolog, anonymous variables

are denoted by an underscore ("_"). An anonymous variable can be used in place of any other variable and is never assigned a value.

Anonymous variables can also be used in facts. The following facts of the Prologue:
eats (_).

could be used to express the following statements in natural language:

Everyone has shoes. (Everyone owns shoes)
Everyone eats. (Everyone eats)

Anonymous variables are compared to any data.

Comments. A good style of programming is to include in the program comments that explain everything that may be incomprehensible to someone else (or even the developer of the program, after a while). Multi-line comments must begin with the characters "/ *" and end with the characters "* /". You can use either the same characters or start a comment with a percentage character ("%") to set one-line comments.

Matching. Visual Prolog has several examples of comparing one value to another. It is clear that identical structures can be compared with each other. For example, it is possible to compare: parent (joe, tammy) with parent (joe, tammy).

However, a comparison usually uses one or more free variables. For example, if X is free, then parent (joe, X) can be compared to parent (joe, tammy) and X will take the value (contact) tammy.

If the variable X is already bound, then it acts in the same way as the ordinary constant. Thus, if X is related to tammy, then parent (joe, X) can be compared to parent (joe, tammy), but parent (joe, X) cannot be compared to parent (joe, millie).

Two free variables can be compared with each other. For example, parent (joe, X) is compared to parent (joe, Y), linking the variables X and Y. From the moment of binding, X and Y are treated as one variable, and any change in the value of one of them leads to an immediate corresponding change in the other. In the case of such "binding" of several free variables, they are all called connected free variables.

The Prologue binds variables (with values) in two ways: input and output. The direction in which the values are transmitted is specified in the flow pattern. In the future (for brevity) the word "template" will be omitted and the expression "parameter flow" will be used. When a variable is passed into a sentence, it is considered an input argument and is denoted by the "and" symbol. When a variable is returned from a sentence, it is the original argument and is denoted by the symbol "o".

Example 6.2.

1. Create a new project. Specify the base directory and project name (lab1ex2). Define the purpose of the project. Set the necessary compiler parameters for the created project.
2. In the file with the extension lab1ex2.pro enter the following program:

```
predicates
        phone_number (symbol, symbol)
clauses
        phone_number ("Albert", "222-3665").
        phone_number ("Betty", "555-5233").
        phone_number ("Carol", "909-1010").
        phone_number ("Dorothy", "438-8400").
goal
```

The presented listing is a complete program on Visual Prolog, which serves as a small telephone directory. Because only standard domains are used, the domains section in this program is not required.

Run the program. Before running the program, in turn, you must add the following goals (goal section):

- phone_number ("Carol", Number);
- phone_number (Who, "438-8400");
- phone_number ("Albert", Number);
- phone_number (Who, Number).

Change the sentence as follows: Suppose Kim and Dorothy have the same phone number. Add this fact to the clauses section and enter the target: phone_number (Who, "438-8400"). Visual Prolog must issue two solutions to this request:

```
Who = Dorothy
Who = Kim
2 Solutions
```

Task.

1. Implement the programs given in the examples (example 6.1, example 6.2) in Visual Prolog 5.2.
2. Using the acquired skills, write two programs in Prolog. Types of programs are determined according to the option (Table 6.4). The option number is determined by the serial number of the student in the teacher's journal.
3. Make a report on the research.

Table 6.4. Task options

№ option	Program type 1	Program type 2
1	Address book	Laptop catalog
2	Catalog of cars	Fare in suburban trains
3	Price list for shoes	Catalog of network equipment
4	Catalog of bicycles	The cost of bakery products
5	Price list of books	Catalog of sports equipment
6	Fabric catalog	Price list for candies
7	Price list for stationery	Catalog of utensils
8	Catalog of toys	Price list for mobile services
9	Price list for computers	Power tool catalog
10	Catalog of mobile phones	Price list of ties
11	Price list for mobile phones	Catalog of films
12	Outerwear catalog	Price list for household техніку
13	Price list for vegetables	Directory of sites
14	Catalog of monitors	Price list of mattresses
15	Price list for medicines	Catalog of enterprises
16	Catalog of video cards	The cost of tickets to the theater
17	Price list for clothes	Music catalog
18	Catalog of audio equipment	Price list for TVs
19	Fruit price list	Catalog of motorcycles
20	Catalog of TVs	Price list for the car
21	Price list for repair work	Catalog of watches
22	Catalog of office equipment	Price list for toys
23	Fare in minibuses	Catalog of flowers
24	Catalog of cameras	Price list of fabrics
25	Price list for sporting goods	Catalog of paintings

6.8. List of control questions on the task № 4

1. What is the Test Goal utility?
2. How to solve a problem in Prolog?
3. What is usually a program in Prolog?
4. Name the most important feature of the Prologue language.
5. What sections does a Visual Prolog program usually consist of?
6. What is placed in the section of sentences?
7. Define the terms "fact", "rule" and "title".
8. Explain the meaning of the combination of characters ": -".
9. What is placed in the section of predicates?
10. Name the rules for declaring predicates.
11. Name the rules for naming predicates.

12. List the main domains in Visual Prolog.

13. Explain when to declare domains of predicate arguments and when not.

14. How is a goal different from a rule?

15. What is the facts section used for?

16. Name the restrictions imposed on the use of constants.

17. What are global sections used for?

18. Name the rules for naming variables in the Prologue.

19. Name the purpose of anonymous variables.

20. Name the rule for naming anonymous variables.

21. How to comment on a line or part of the program in the Prologue?

22. Give an example of comparing one value with another.

23. In which case the variables are called conjugate free variables.

6.9. Task № 5. Unification and search with return

Comparison and unification

The process that Visual Prolog uses when trying to map a call (from a sub-goal) to a sentence (in the clauses section of the program) involves associating a particular call with a specific sentence – what is called unification. In the Prologue, unification implements procedures of more traditional languages – such procedures as parameter transfer, case selection, structure creation, structure access, assignment.

Consider the program shown in example 6.3 in terms of how the utility Test Goal will look for all solutions to the following goal: written_by (X, Y). Trying to execute the target statement written_by (X, Y), Visual Prolog must check each sentence written_by in the program. Comparing the arguments X and Y with the arguments of each sentence written_by, Visual Prolog performs a search from the beginning of the program to its end. Once it finds a sentence that matches the target statement, Visual Prolog assigns values to the free variables so that the target statement and the sentence become identical, in which case it is said that the target statement is unified with the sentence. This mapping operation is called unification.

Example 6.3.

```
domains
        title, author = symbol
        pages = unsigned
predicates
        book (title, pages)
        written_by (author, title)
        long_novel (title)
clauses
```

```
written_by (fleming, "DR NO").
written_by (melville, "MOBY DICK").
Book ("MOBY DICK", 250).
Book ("DR NO", 310).
long_novel (Title):-
        written_by (_, Title),
        book (Title, Length),
        Length> 300.
```

Since X and Y are free variables in the target statement, and the free variable can be unified with any other argument (and even with another free variable), the target statement can be unified with the first sentence written_by in the program, as shown below:

```
written_by (    X,          Y    ).
                ↕          ↕
written_by (fleming, "DR NO").
```

Visual Prolog establishes a match, X becomes associated with fleming, and Y - with "DR NO". At this point, Visual Prolog prints:

```
X = fleming, Y = "DR NO"
```

Since **Test Goal** seeks all solutions for a given goal, the goal statement will also be unified with the second sentence **written_by**:

```
written_by (melville, "MOBY DICK").
```

So **Test Goal** also prints a second solution:

```
X = melville, Y = "MOBY DICK"
2 Solution
```

Assume that the program is given a target statement written_by (X, "MOBY DICK"). **Visual Prolog** will match the first sentence **written_by:**

```
written_by (    X,      "MOBY DICK").
               ↕             ↕
written_by (fleming,    "DR NO").
```

Since "MOBY DICK" and "DR NO" do not match, the unification attempt fails. Then **Visual Prolog** will check the following fact in the program:

```
written_by (melville, "MOBY DICK").
```

This fact is really unified, and X becomes associated with melville.
Consider how **Visual Prolog** fulfills the following target statement:

```
long_novel (X).
```

When **Visual Prolog** tries to execute a target statement, it checks to see if the request can really match the fact or title of the rule. In our case correspondence with long_novel (Title) is established. **Visual Prolog** checks the sentence for long_novel, trying to complete the match by unifying the arguments. Since in the objective statement X is a free variable, it can be unified with any other argument. The title is also not related in the title of the long_novel sentence. The target statement matches the title of the rule, and unification is performed. Eventually, **Visual Prolog** will try to reconcile the sub-goals with the rule:

```
long_novel (Title):-
        written_by (_, Title),
        book (Title, Length),
        Length> 300.
```

Trying to match the body of the rule, Visual Prolog will refer to the first sub-goal in the body of the rule – written_by (_, Title). Because the authorship of the book is insignificant, an anonymous variable (_) appears in place of the "author" argument. The appeal written_by (_, Title) becomes the current sub-goal, and the Prologue seeks a solution to this appeal.

The prologue seeks correspondence with this sub-goal from the top to the end of the program. As a result, unification is achieved with the first fact for written_by, namely:

```
written_by (     _,          Title),
                 ↕           ↕
written_by (fleming, "DR NO").
```

The variable Title is associated with "DR NO", and by the next sub-goal book (Title, Length) the call is already performed with this value of the variable.

Next, **Visual Prolog** begins the next search process, trying to match the reference to the book. Since the Title is related to "DR NO", the actual address looks like a book ("DR NO", Length). The search process starts again from the top of the program. In this case, it should be noted that the first attempt to match the sentence book ("MOBY DICK", 250) will fail, and **Visual Prolog** will move to the second sentence book in search of a match. In the second sentence, the title of the book corresponds to the sub-goal, and Visual Prolog associates the variable Length with the value 310.

Now the third sentence in the body long_novel becomes the current subgoal:

Length> 300.

Visual Prolog performs a comparison that completes successfully: 310 is greater than 300. At this point, all subgoals in the rule body are met, and therefore the long_novel (X) call is successful. Because the X in the request was unified with the Title variable in the rule, the value that the Title associates with when confirming the rule is returned and unified with the X variable. :

```
X = "DR NO"
1 Solution.
```

Search with return

Visual Prolog uses the trial and return method to find a solution to a problem. This method is called return search. If, when starting to search for a solution to a problem (or target statement), Visual Prolog has to choose between alternative paths, it puts a marker at the branch point (called the rollback point) and selects the first subgoal to test. If this goal is not met, Visual Prolog will return to the rollback point and try to check another sub-goal. Example 6.4 discusses a simple program that uses Test Goal.

Example 6.4.

```
predicates
        likes (symbol, symbol)
        tastes (symbol, symbol)
        food (symbol)
clauses
        likes (bill, X):-
                food (X),
                tastes (X, good).
        tastes (pizza, good).
        tastes (brussels_sprouts, bad).
        food (brussels_sprouts).
        food (pizza).
```

The rule of the likes states that Bill loves good food. To see how the return search works, let's introduce the following target statement to the program:

likes (bill, What).

Warning! When Prolog tries to match the target statement, it will start the search from the top of the program.

In this case, the Prologue will look for a solution, conducting from the top of the program to match the sub-goal likes (bill, What). It matches the first sentence in the program and the variable "what" is unified with the variable X. A mapping to the rule header causes Visual Prolog to try to satisfy this rule. In doing so, he moves along the body of the rule and turns to the first sub-goal that is in it: food (X).

Warning! If a new request is executed, matching for that request starts again from the top of the program.

In an attempt to reconcile the first sub-goal, Visual Prolog (starting at the top) compares each fact or title with the rules it encounters in the program. It matches the query in the first fact represented by the food relation. Thus, the variable X is associated with the value of brussels_sprouts. Because there is more than one possible response to the food (X) request, Visual Prolog places a return point (marker) next to the food (brussels_sprouts) fact. This return search point indicates where the Prologue will start searching for the next possible match for food (X).

Warning! When the matching is successful, it is said that the return is returned, and another sub-goal can be tested.

Because variable X is associated with brussels_sprouts, the following call will be performed as follows:

tastes (brussels_sprouts, good)

and Visual Prolog will start searching from the top of the program again, trying to reconcile this appeal. As the corresponding sentences are not found, the appeal fails, and now Visual Prolog starts the return mechanism. Starting the search with the return, the Prologue retreats to the last position where the rollback point was set. In this case, the Prologue returns to the fact:

food (Brussels_sprouts).

Warning! The only way to release a variable that is once associated with a sentence is to roll back when searching with a return.

When **Prolog** retreats to the return point with a return, it releases all variables associated after that point and will look for another solution for the original appeal. The appeal was food (X), so the connection of brussels_sprouts with X is canceled. Now the Prologue is trying to find a solution for this appeal

again. It corresponds to the fact of food (pizza). This time the variable X is associated with the value of pizza.

The prologue goes to the next sub-goal in the rule, while having a new related variable. A new appeal is generated, tastes (pizza, good), and the search begins (again from the top of the program). This time the match is found and the target statement is successfully executed.

Because the What variable in the target statement is unified with the X variable in the likes rule, and the X variable is associated with the pizza value, the What variable is now associated with the pizza value, and **Visual Prolog** displays the solution:

```
 What = pizza
1 Solution
```

There are four basic rules for returning search:

- sub-goals must be agreed in order, from top to bottom;

- predicative sentences are checked in the order in which they appear in the program, from top to bottom;

- when the goal corresponds to the title of the rule, the body of this rule must be agreed next: the body of the rule now forms a new set of sub-goals for coordination;

- the target statement is considered consistent when the relevant fact is found for the target tree leaf.

1. Performing sub-goals, **Visual Prolog** begins the search with the first sentence defining the predicate. Then one of two things can happen: 1. **Visual Prolog** finds the appropriate sentence, then:

- if there is another sentence that may reconcile the sub-goal, **Visual Prolog** sets a marker (to indicate the return point) and associates all free variables in the sub-goal (which correspond to the values in the sentence) with the corresponding values;

- if the given sentence is the title of the rule, then the body of this rule is evaluated.

The sub-goals in the body of the rule must be met for the successful completion of the appeal.

2. Visual Prolog cannot find the corresponding sentence. The target statement is inconsistent, and Visual Prolog performs a return search in an attempt to reconcile the previous subtask. When the process reaches the last return point, Visual Prolog releases all variables that have been assigned new values (after the return point has been set), and tries to reconcile the original request again.

Visual Prolog starts the search from the top of the program. When it returns to the appeal, the new search process begins with the last set rollback point. If the search is unsuccessful, the return search is performed again. If the

return search process has exhausted all sentences for all sub-goals, it means that the target statement is not consistent.

Solution management

The built-in search engine with return in the Prologue can lead to finding unnecessary solutions, resulting in lost efficiency (for example, when it is desirable to find only one solution). In other cases, it may be necessary to continue to seek additional solutions, even if the target statement has already been agreed.

The prologue provides two tools that allow you to control the return search engine:

- the fail predicate, which is used to initialize the return search;
- predicate cut or clipping (denoted by the symbol "!") – to prohibit the possibility of return.

Use the predicate fail

The prologue starts the search with a return when the call fails. The language supports a special fail predicate, which causes a failed completion, and initials a return. The action of the fail predicate is equivalent to the effect of comparing 2 = 3 or another impossible sub-goal. Example 6.5 illustrates the use of this special predicate.

Example 6.5.

```
domains
        name = symbol
predicates
        father (name, name)
        everybody
clauses
        father (leonard, katherine).
        father (carl, jason).
        father (carl, marilyn).

        everybody:-
                father (X, Y),
                write (X, "is", Y, "'s father\n"),
                fail.
```

Let's find all the solutions to everybody's goal. Using the Test Goal utility, you can record a goal as follows:

```
goal
        everybody.
```

Test Goal will find all solutions to everybody's goal and display the following:

leonard is katherine's father
carl is jason's father
carl is marilyn's father
no

When compiling the program, Visual Prolog will find only the first suitable solution for father (X, Y). After the goal statement defined in the goal section has been fulfilled for the first time, nothing tells the Prologue about the need to continue the search with a return. Therefore, appealing to the father will lead to only one decision. The everybody predicate in the program uses fail to support return searches. Everybody's task is to find all the solutions for the father and give a complete answer.

Fail cannot be reconciled (it is always unsuccessful), so Prolog is forced to repeat the search with a return. When searching with a return, it returns to the last request, which can give multiple solutions. This treatment is called indeterminate. It is the opposite of a deterministic appeal that can only give one solution.

It is not necessary to place subgoals after fail in a rule body. The fail predicate fails all the time, there is no way to achieve the subgoal located after fail.

Interrupt search with return

The **Prolog** provides a clipping capability that is used to interrupt the return search. The clipping is indicated by an exclamation mark ("!"). The clipping is simple: it is impossible to roll back (search with return).

The clipping is placed in the program in the same way as the subgoal in the rule body. When the process passes through the clipping, the appeal to the cut is immediately satisfied and the appeal to the next sub-goal is performed (if any). After passing through the clipping, it is no longer possible to roll back to the sub-goals that are located in the processed sentence before the clipping, and it is impossible to return to other sentences that define the predicate to be processed (predicate containing clipping).

There are two main uses for clipping:

-I f it is known that certain links will never lead to meaningful decisions (clipping is applied), the program becomes faster and more economical. This technique is called green clipping.

- if the clipping is required by the logic of the program itself to exclude alternative sub-goals. This is a red clipping.

Prevent the search from returning to the previous sub-goal in the rule

```
r1:- a, b,
        !,
        c.
```

Such a record is a way to inform the Prologue that the first solution he found for subgoals a and b is sufficient. Having the ability to find multiple solutions to "c" by searching with return, Prolog can not roll back (search with return) through clipping and find an alternative solution to appeals "a" and "b". He also cannot return to another sentence that defines the predicate r1. As an example, consider the program shown in example 6.6.

Example 6.6.

```
predicates
        buy_car(symbol,symbol)
        car(symbol,symbol,integer)
        colors(symbol,symbol)
clauses
        buy_car(Model,Color):-
                car(Model,Color,Price),
                colors(Color,best),
                !,
                Price < 25000.
        car(maserati,green,25000).
        car(corvette,black,24000).
        car(corvette,red,26000).
        car(porsche,red,24000).
        colors(red,best).
        colors(black,mean).
        colors(green,preppy).
goal
        buy_car(corvette,Y).
```

In this example, the goal is to find a corvette (Corvette) of a pleasant color (best), with a suitable price. Clipping in the buy_car rule means that since the database contains only one nicely colored Corvette, albeit at a very high price, there is no need to look for another car.

After receiving the target statement buy_car (corvett, Y) the program will perform the following steps:

1. The Prolog refers to car, the first subgoal for the predicate buy_car.
2. Performs a test for the first machine, Maserati, which fails.
3. Then check the following car sentences and find a match by associating the Color variable with the value black.

4. Proceed to the next request and check whether the selected machine has a pleasant color. Black is not pleasant in this program, so the test fails.

5. Performs a car return search and searches again for a Corvette that satisfies this criterion.

6. Finds the match and checks the color again. This time the color is nice. The prologue goes to the next sub-goal in the rule: to clipping. Clipping is performed immediately, "freezing" all variables previously associated in this sentence.

7. Moves to the next (and last) sub-goal in the rule, to compare Price <25000.

8. The test fails, and Visual Prolog tries to do a return search to find another machine to test. Clipping prevents an attempt to solve the last sub-goal, and the target statement fails

Prevent search by returning to the next sentence

Clipping can be used as a way to tell Visual Prolog that it has chosen the correct sentence for a particular predicate. For example, consider the following fragment of the program:

```
r (1):-
        !,
        a, b, c.
r (2):-
        !,
        d.
r (3):-
        !,
        c.
r (_):-
        write ("This is a catchall clause.").
```

The use of clipping makes the predicate "r" deterministic. In this case, the Prologue appeals to "r" with a single integer argument. Assume that r (1) is used. Visual Prolog scans the program for matching matches. It corresponds to the first sentence defined for "r". Because there is more than one possible solution to this sentence, Visual Prolog puts a return point next to this sentence. Now the Prologue begins processing the body of the rule, goes through the clipping and eliminates the possibility of returning to another sentence "r". This eliminates return search points, improving program performance, and ensures that a sentence that catches errors will only be executed if none of the conditions match the "r" statement. As another example, consider the program shown in example 6.7.

Example 6.7.

```
predicates
        friend (symbol, symbol)
        girl (symbol)
        likes (symbol, symbol)
clauses
        friend (bill, jane):-
                girl (jane),
                likes (bill, jane),
                !.
        friend (bill, jim):-
                likes (jim, baseball),
                !.
        friend (bill, sue):-
                girl (sue).
        girl (mary).
        girl (jane).
        girl (sue).
        likes (jim, baseball).
        likes (bill, sue).
goal
        friend (bill, Who).
```

If there was no clipping, the Prologue would suggest two solutions: "Bill is a friend of both Jim and Sue." However, the clipping in the second sentence, which defines friend, suggests that if this sentence is agreed, then Bill's friend has already been found, and there is no need to continue searching for other candidates.

Determinism and clipping

If the friend predicate defined in Example 6.7 did not contain clippings, it would be a nondeterministic predicate (capable of issuing multiple solutions using a return search). The prologue checks for indeterminate sentences itself.

In Visual Prolog, there is a compiler directive check_determ. If you add this directive to the beginning of the program, the Prologue will issue a warning if undetermined sentences are detected during compilation. It is possible to convert non-deterministic sentences into deterministic ones, this can be done by inserting clippings into the body of the rules that define this predicate.

Predicate not

The program in Example 6.8 demonstrates how the predicate not can be used to identify the student with the best grade: that is, the student who has a GPA of at least 3.5 and who is not currently on probation.

Example 6.8.

```
domains
        Name = symbol
        GPA = real
predicates
        honor_student(Name)
        student(Name,GPA)
        probation(Name)
clauses
        honor_student(Name):-
                student (Name, GPA),
                GPA>=3.5,
                not(probation (Name)).
        student("Betty Blue", 3.5).
        student("David Smyth", 2.0).
        student("John Jonson", 3.7).
        probation("Betty Blue").
        probation("David Smith").
goal
        honor_student(X).
```

Warning! The predicate not will be successful if the truth of this subgoal cannot be proved.

This prevents binding inside unbound variables. When calling a sub-target with free variables from within, Visual Prolog will return an error message: "Free variables not allowed in not or retractall". This is due to the fact that to bind free variables in a sub-goal, the sub-goal must be unified with some other sentence and executed. The correct way to manage incoherent subgoal variables inside not is to use anonymous variables.

Here are some examples of correct and incorrect sentences:

```
likes (bill, Anyone):-                    % Anyone – initial argument
        likes (sue, Anyone),
        not (hates (bill, Anyone)).
```

In this example, Anyone connects with likes (sue, Anyone) before the Prologue concludes that hates (bill, Anyone) is not true. This sentence will work correctly.

If you change the sentence so that you do not call first, you will get an error message: "Free variables not allowed in not".

Приклад:

```
likes (bill, Anyone):-                    % This will not work properly
```

```
        not (hates (bill, Anyone)),
        likes (sue, Anyone).
```

Even if you replace any (hates (bill, Anyone)) Anyone with an anonymous variable, and the sentence will thus not return an error, the wrong result will still be displayed. Example:

```
likes (bill, Anyone):-          % This will not work properly
        not (hates (bill, _)),
        likes (sue, Anyone).
```

This sentence states that Bill is liked by anyone, if nothing is known about whom Bill hates, and if this "someone" is liked by Sue. The real sentence was that Bill liked someone who liked Sue, and Bill didn't hate that man.

Tasks

1. Implement the programs given in the examples (example 6.3 - example 6.8) in Visual Prolog 5.2.
2. For programs created in the course of task № 4 implement the following predicates:
2.1. Output of all records.
2.2. Search for all records that satisfy the condition.
2.3. Search for the first record that does not satisfy the condition.
3. Make a report on the research.

6.10. List of control questions on the task № 5

1. Define the concept of "unification" and explain the process of unification of the target statement with the sentence.
2. Explain the essence of the search method with return.
3. Explain why the return search point is used.
4. Name the basic rules of search with return.
5. Explain the mechanism of sub-goals.
6. Name the tools for managing the return search engine.
7. Name the main cases when cutting is used.
8. Explain how to turn non-deterministic sentences into deterministic ones in the Prologue.

6.11. Task №6. Cycle and recursion

Visual Prolog has no For, While, Repeat constructs. There is no direct way to express repetition. The prologue provides only two types of iterations –

rollback, which searches for many solutions in one query, and recursion, in which the procedure calls itself. Visual Prolog recognizes a special recursion case, namely tail recursion, and compiles it into an optimized iterative loop.

Visual Prolog can express iterations in both procedures and data structures. Prolog allows you to create data structures whose size is not known at the time of creation.

Using rollback with loops

A return search is a good way to find all possible solutions to a target statement. Even if the problem does not have many solutions, you can use return search to perform iterations. To do this, determine the predicate with two sentences:

```
repeat.
repeat:- repeat.
```

This technique demonstrates the creation of a management structure of the Prologue, which generates an infinite number of decisions. The purpose of the repeat predicate is to allow infinity of search with return (infinite number of rollbacks). Example 6.9 demonstrates the use of repeat to save entered characters and print them until the user presses the <Enter> key.

Example 6.9.

```
predicates
        repeat
        typewriter
clauses
        repeat.
        repeat:- repeat.
        typewriter:-
                repeat,
                readchar (C), Read the symbol, assign its meaning to C
                write (C),
                C = '\r',       Carriage return (Enter) or failure symbol!.
goal
        typewriter (), nl.
```

The program shown in example 3.1 shows how repeat works. The typewriter rule: describes the process of receiving characters from the keyboard and displaying them on the screen until the user presses the <Enter> key.

The **typewriter rule** works as follows:

1. Executes repeat (which does nothing but puts a rollback point).
2. Assigns a character value to the variable C.

3. Displays C.

4. Checks that C corresponds to the carriage return code.

5. If so, then – completion. If not, return to the rollback point and look for alternatives. Since neither write nor readchar are alternatives, there is a constant return to repeat, which always has alternative solutions.

6. Processing progresses: the program reads the next character, displays it and checks for compliance with the return code of the carriage.

Warning! All variables lose their values when processing rolls back to the position preceding the predicate calls that set those values.

Recursive procedures

Another way to organize repetitions is recursion. A recursive procedure is a procedure that causes itself. In a recursive procedure, there is no problem with remembering the results of its execution, because any calculated values can be passed from one call to another as arguments of a predicate that is recursively called.

For example, you need to find the factorial of the number N: if N is 1, then the factorial is 1, otherwise find the factorial N-1 and multiply it by N. To find factorial 3, you need to find factorial 2, and to find factorial 2, you need to calculate factorial 1 Factorial 1 can be found without referring to other factors, so repetitions will not begin. If we have a factorial of 1, then we need to multiply it by 2 to get a factorial of 2, and then multiply the result by 3 to get a factorial of 3.

In **Visual Prolog**, it looks like this:

```
factorial (1, 1):-! .
factorial (X, FactX):-
        Y = X-1,
        factorial (Y, FactY),
        FactX = X * FactY.
```

The full implementation of this approach to the task is presented in example 6.10.

Example 6.10.

/ * Recursive program for calculating factors. Recursion is normal, not tail. * /

```
predicates
        factorial (unsigned, real)
clauses
        factorial (1,1):-
```

```
            !.
        factorial (X, FactX):-
            Y = X-1,
            factorial (Y, FactY),
            FactX = X * FactY.
goal
        X = 3,
        factorial (X, FactX).
```

How does a computer execute a factorial predicate in the middle of factorial predicate processing? The fact is that the computer creates a new copy of the predicate factorial in such a way that factorial becomes able to call itself as a completely independent procedure. Of course, the execution code will not be copied, but all arguments and intermediate variables are copied.

The information is stored in an area of memory called the stack frame, or simply the stack that is created each time a rule is called. When the rule is executed, the memory occupied by its stack frame is freed (unless it is a non-deterministic rollback), and execution continues in the stack frame of the parent rule.

Advantages of recursion

Recursion has three main advantages:

- it can express algorithms that cannot be conveniently expressed in any other way;
- it is logically simpler than the iteration method;
- it is widely used in list processing.

Recursion is a good way to describe tasks that include tasks of the same type. For example, a search in a tree (a tree consists of smaller trees) and recursive sorting (to sort a list, it is divided into parts, the parts are sorted and then combined together).

Tail recursion optimization

Recursion has one major drawback – it "eats" memory. Each time one procedure calls another, information about the execution of the calling procedure must be saved so that the calling procedure can resume execution after the execution of the called procedure at the same place where it stopped. This means that if the procedure calls itself 100 times, then 100 different states must be written simultaneously (decision execution states are stored in the stack frame).

Consider a special case where the procedure can cause itself without saving information about your condition. Assume that the procedure is called for the last time, ie when the called procedure terminates, the calling procedure will not resume its execution. This means that the calling procedure does not need to

save its state, because this information is no longer needed. As soon as the called procedure is completed, the processor should go in the direction specified for the calling procedure after its execution.

For example, suppose that procedure A calls procedure B and B calls C as its last step. When B causes C, B should do nothing more. Therefore, instead of storing information about the current state C in the procedure stack, it is necessary to rewrite the old saved information about the state B (which is no longer needed) to the current information about C, making appropriate changes to the stored information. When C finishes execution, it will assume that it is called directly by procedure A.

Assume that in the last step, procedure B calls itself instead of procedure C. It turns out that when B calls B, the stack (state) for calling B must be replaced by the stack for called B. This is a very simple operation, the arguments are simply assigned new values and then the process returns to the beginning of procedure B. Therefore, from a procedural point of view, what is happening is very similar to just updating the control variables in the loop.

This operation is called tail recursion optimization or last call optimization. Last call optimization is not applicable to recursive functions.

How to set tail recursion

In the Prologue, the phrase "one procedure calls another in its last step" means:

- challenge is the last sub-goal of the sentence;
- previously there were no return points in the sentence.

The following is an example that satisfies both conditions:

```
count (N):-
        write (N), nl,
        NewN = N + 1,
count (NewN).
```

This procedure is a tail recursion that calls itself without reserving a new stack frame, and therefore does not deplete memory. As shown in the program shown in example 6.11, if it is given the target statement "count (0)." then the count predicate will print integers starting with 0 and will never stop. Eventually there will be an integer overflow, but a stop due to memory depletion will not occur.

Example 6.11.

```
/* A program with tail recursion that does not deplete memory */
        predicates
                count (ulong)
        clauses
```

```
        count (N):-
                write ('\r', N),
                NewN = N + 1,
                count (NewN).
goal
        nl,
        count (0).
```

Tail recursion is not optimized

Three erroneous ways to organize tail recursion:

1. If a recursive call is not the last step, the procedure is not a tail recursion. Example:

```
badcount1 (X):-
        write ('\ r', X),
        NewX = X + 1,
        badcount1(NewX),
        nl.
```

Each time badcount1 calls itself, the stack must be saved so that processing can be returned to the calling procedure, which must be performed before nl. Therefore, it will make only a few thousand recursive calls before the end of free memory.

Another way to make tail recursion not optimized is to leave some possible alternative untested until the recursive call is executed. The stack must then be saved, because if the recursive call fails, the calling procedure may roll back and start testing this alternative. Example:

```
badcount2 (X):-
        write ('\r', X),
        NewX = X + 1,
        badcount2 (NewX).
badcount2 (X):-
        X <0,
        write ("X is negative.").
```

The first sentence badcount2 calls itself, while the second sentence is not yet executed. Again, the program runs out of memory after a certain number of calls.

To lose the optimization of tail recursion, it is not necessary to have an untested alternative as a separate sentence of the recursive procedure. An untested alternative can be in any called predicate. Example:

```
badcount3 (X):-
        write ('\ r', X),
        NewX = X +1,
        check (NewX),
        badcount3 (NewX).
check (Z):- Z >= 0.
check (Z):- Z < 0.
```

Assume that X is a positive value. When badcount3 calls itself, the first check sentence reaches the goal, and the second check sentence has not yet been checked. Therefore, badcount3 must keep a copy of its stack frame to be able to go back and start checking the second check sentence in case the recursive call fails (example 6.12).

Example 6.12.

% In a 32-bit architecture, these examples will run long enough to take

% a lot of memory and significantly reducing the overall performance of the system.

```
predicates
        badcount1 (long)
        badcount2 (long)
        badcount3 (long)
        check (long)
clauses
```

% badcount1: **recursive call is not the last step**

```
        badcount1 (X):-
                write ('\r', X),
                NewX = X+1,
                badcount1 (NewX),
                nl.
```

% badcoun2: **it is a sentence that is not executed during implementation**

% recursive call

```
        badcount2 (X):-
                write ('\r', X),
                NewX = X + 1,
                badcount2 (NewX).
        badcount2 (X):-
                X <0,
                write ("X is negative.").
```

% badcount3: **untested alternative to the procedure that**

% called before recursive call.

```
        badcount3 (X):-
```

```
        write ('\ r', X),
        NewX = X +1,
        check (NewX),
        badcount 3 (NewX).
check (Z):-
        Z >= 0.
check (Z):-
        Z < 0.
```

It should be noted that badcount2 and badcount3 are worse than badcount1 because they generate return points. Clipping allows you to discard all possible unnecessary alternatives. To set the cut, you must use the compiler directive check_determ.

You can fix badcount3 as follows (by modifying its name):

```
cutcount3 (X):-
        write ('\r', X),
        NewX = X +1,
        check (NewX),
        !,
        cutcount3 (NewX).
```

"Clipping" is also effective in badcount2 if you move the test from the second sentence to the first:

```
cutcount2 (X):-
        X>= O,
        !,
        write ('\r', X),
        NewX = X + 1,
        cutcount2 (NewX).
cutcount2 (X):-
        write ("X is negative.").
```

Clipping is used whenever we are not interested in alternatives. In the original version of the previous example, the second sentence had to leave a choice because the first sentence did not contain an X test. By moving the test to the first sentence and denying it, a decision can be made there and the cutoff set according to the statement: "Now I know I should not write that X is negative.

The same goes for cutcount3. The check predicate shows the situation when it is necessary to perform some additional operation on X based on the sign. However, the code for check is indeterminate, and clipping after calling it

is all that needs to be done. However, the above is a bit artificial - it might be better for the check to be deterministic:

```
check (Z): -
        Z> = 0,
        !,
        ...             % using Z
check (Z):-
        Z <0,
        ...             % using я Z
```

Check in the second sentence check - complete denial of check in the first, so check can be rewritten as:

```
check (Z):-
        Z >= 0,
        !,
         ...
```

If clipping is performed, the computer assumes that there are no untested alternatives and does not create a stack frame. The program shown in example 6.13 contains modified versions of badcount2 and badcount3.

Example 6.13.

/* Shows how badcount2 and badcount3 can be improved by declaring a cut to exclude unverified sentences. These versions use optimized tail recursion.. */

```
predicates
        cutcount2 (long)
        cutcount3 (long)
        check (long)
clauses
        cutcount2 (X):-
                X >= 0,
                !,
                write ('\r', X),
                NewX = X + 1,
                cutcount2 (NewX).
        cutcount2 (_):-
                write ("X негативно.").
                cutcount3 (X):-
                write ('\r ', X),
                NewX = X + 1,
                check (NewX),
                cutcount3 (NewX).
```

```
check (Z):- Z >= 0.
check (Z):-Z <0.
```

Unfortunately, clipping will not help with badcount1, in which the need to copy stack frames is not associated with untested alternatives. The only way to improve badcount1 is to do the calculation so that the recursive call occurs at the end of the sentence.

Using arguments as loop variables

Consider a small transformation from Pascal to Prolog. In the section "Recursive procedures" the calculation of the factorial using a recursive procedure was demonstrated. Another way to calculate the factorial is to use iteration for this. In Pascal it would look like this:

```
P: = 1;
for I: =        1 to N do P: = P *I;
FactN: = P;
```

The ": =" operator is an assignment operator and is pronounced as "assign". 4 variables were used. N is the number whose factorial will be calculated; FactN – the result of the calculation; And - cyclic variable (from 1 to N), P – summing variable.

The first step in translating to Prolog is to replace it with a simpler wording for the loop, which more accurately defines what happens to I at each step. The **while** definition is used for this:

```
P: = 1;                          /* Initialization «P» and «I» */
I: = 1;
while I <= N do     /* Cycle assignments */
        begin
                P: = P * I              / * Renewal «P» and«I» */
                I: = I + 1
        end;
FactN: = P;           / * Show result * /
```

The program shown in example 6.14 shows the Pascal while loop translated into Prolog.

Example 6.14.

```
predicates
        factorial (unsigned, long)
        factorial_aux (unsigned, long, unsigned, long)
% Numbers that are likely to become larger are announced   long.
```

```
clauses
	factorial (N, FactN):-
		factorial_aux (N, FactN, 1,1).
	factorial_aux (N, FactN, I, P):-
		I <= N,
		!,
		NewP = P * I,
		Newl = I +1,
		factorial_aux (N, FactN, Newl, NewP).
	factorial_aux (N, FactN, I, P):-
		I> N,
		FactN = P.
```

Consider the program in more detail. There are only two arguments in the sentence for the factorial predicate – N and FactN. They are the "entrance" and the "exit" from the point of view of the one who calculates the factorial. Sentences for factoriai_aux (N, FactN, I, P) actually provide recursion. Their arguments are four variables that must be passed from one step to another. Therefore, factorial simply calls factorial_aux, passing it N and FactN with initial values for I and P:

```
factorial (N, FactN):-
	factorial_aux (N, FactN, 1, 1).
```

So I and P are initialized. But how does factorial convey FactN, since it doesn't matter yet? The answer is that conceptually Visual Prolog here unifies a variable called FactN in one sentence with a variable called FactN in another sentence. In the same way, factoriai_aux passes itself to FactN as an argument in a recursive call. Eventually, the last FactN will get the value, and then all other FactNs that are unified with it will get the same value. "Conceptually", because in reality there is only one FactN. Visual Prolog can determine from the source code that FactN is not actually used before the second sentence factorial_aux, and the same FactN is passed all the time.

Now about the work of factorial_aux. Usually this predicate checks the sentence "I is less than or equal to N" for cyclic calculation, and then recursively calls itself with new values for I and P. Here is another feature of Visual Prolog. In Prologue, the arithmetic expression is correct: p = p + 1 is not a definition of assignment (as it should be in most other programming languages).

Warning! You cannot change the value of a variable in Visual Prolog

In the Prologue it is as absurd as in algebra. Instead, you need to create a new variable and give it the value you want. Example:

```
        NewP = P + 1
```
Therefore, the first sentence is as follows:
```
        factorial_aux (N, FactN, I, P):-
                I <= N,
                !,
                NewP = P * I,
                Newl = I +1,
                factorial_aux (N, FactN, Newl, NewP).
```
As in the case of cutcount2, in this sentence the clipping will optimize the tail recursion, although it is not the last sentence in the predicate. Ultimately I will exceed N; the current values of P and FactN are unified and recursion stops. This is realized in the second sentence, which is executed when the check I <= N in the first sentence is unsuccessful.

```
        factorial_aux (N, FactN, I, P):-
                I > N,
                FactN = P.



        factorial_aux (_, FactN, _, FactN).
```
Task

1. Implement the programs given in the examples (example 6.9 - example 6.14) in Visual Prolog 5.2.
2. Using the acquired skills, according to the option, write a program in the language of Prolog, which implements calculations according to the formula shown in table 3.1. The option number is determined by the serial number of the student in the teacher's journal.

Table 6.5. Task options

№ option	Formula
1	$m = \prod_{i=1}^{n} (i + \sin(i))$
2	$r = \prod_{k=1}^{m} k / \cos(k)$
3	$v = \prod_{s=1}^{k} \sin(s) / \cos(s+1)$

4	$g=\prod_{t=1}^{k}(t+3)*t$
5	$a=\prod_{j=1}^{l}(j+1/\cos(j))$
6	$f=\prod_{i=1}^{n}(i+\sin(i)/\cos(i))$
7	$b=\prod_{s=1}^{k}s/(1+\cos(s)$
8	$q=\prod_{f=1}^{r}(f+2)/(f+3)$
9	$h=\prod_{p=1}^{k}(2*p+\sin(p))$
10	$w=\prod_{x=1}^{z}(x+2*x)/x$
11	$n=\prod_{i=1}^{m}(\sin(i)-\cos(i))$
12	$c=\prod_{s=1}^{k}(s+10)/s$
13	$t=\prod_{a=1}^{b}(a-1/(a+2))$
14	$p=\prod_{n=1}^{m}(n/\sin(n)+1)$
15	$b=\prod_{c=1}^{f}(\cos(c)/2+3/\sin(c))$
16	$v=\prod_{m=1}^{n}(m+e^{m+1})$
17	$y=\prod_{n=1}^{m}(\cos(n)+\sin(n)/5)$
18	$x=\prod_{a=1}^{b}e^{\lvert\sin(a)+\cos(a)\rvert}$

19	$z = \prod_{b=1}^{c} \sin(c) * \cos(c) / 5$
20	$u = \prod_{h=1}^{z} \lvert h - \sin(h) \rvert / 3$
21	$r = \prod_{i=1}^{k} (e^{\cos(i)} + e^{\sin(i)}) / 2$
22	$w = \prod_{j=1}^{k} \sin(\lvert h - 2h \rvert)$
23	$n = \prod_{k=1}^{l} \cos(l - e^{\sin(l)})$
24	$k = \prod_{s=1}^{h} (\cos(s) + \sin(s)) / (\sin(s) * \cos(s))$
25	$f = \prod_{i=1}^{n} \sin(i) * e^{\cos(i+2)}$

3. According to the variant of the task (table 6.6), write the program in the Prologue language. The option number is determined by the serial number of the student in the teacher's journal.

Table 6.6. Task options

№ option	**Task**	**Additional requirements**
1	Calculate the sum of n terms of the arithmetic progression *. The program must calculate a series from the smallest member of the progression to the larger.	n=10, d=3, a_1=1
2	Derive n members of the increasing geometric progression **. The program should display a series from the larger member of the progression to the smaller.	n=8, q=2, b_1=15
3	Print n members of the arithmetic progression *. The program should display a series from the larger member of the progression to the smaller.	n=20, d=2, a_1=5
4	Calculate the sum of n terms of the geometric progression **. The program must calculate a series from the smallest	n=5, q=3, b_1=1

	member of the progression to the larger.	
5	Calculate the sum of n terms of the arithmetic progression *. The program must calculate a series from the larger member of the progression to the smaller.	n=7, d=5, a_1=15
6	Derive n members of the increasing geometric progression **. The program should output a series from the smallest member of the progression to the larger.	n=5, q=5, b_1=2
7	Print n members of the arithmetic progression *. The program should display a series from the larger member of the progression to the smaller.	n=20, d=2, a_1=5
8	Calculate the sum of n terms of the increasing geometric progression **. The program must calculate a series from the larger member of the progression to the smaller.	n=10, q=2, b_1=10
9	Calculate the sum of n terms of the arithmetic progression *. The program must calculate a series from the smallest member of the progression to the larger.	n=15, d=4, a_1=3
10	Derive n members of the increasing geometric progression **. The program should display a series from the larger member of the progression to the smaller.	n=6, q=7, b_1=3
11	Print n members of the arithmetic progression *. The program should output a series from the smallest member of the progression to the larger.	n=8, d=8, a_1=10
12	Calculate the sum of n terms of the increasing geometric progression **. The program must calculate a series from the smallest member of the progression to the larger.	n=5, q=6, b_1=1
13	Calculate the sum of n terms of the arithmetic progression *. The program must calculate a series from the larger member of the progression to the smaller.	n=12, d=3, a_1=4
14	Derive n members of the increasing geometric progression **. The program should output a series from the smallest member of the progression to the larger.	n=15, q=2, b_1=5

15	Print n members of the arithmetic progression *. The program should display a series from the larger member of the progression to the smaller.	n=14, d=6, a_1=1
16	Calculate the sum of n terms of the increasing geometric progression **. The program must calculate a series from the larger member of the progression to the smaller.	n=12, q=3, b_1=15
17	Calculate the sum of n terms of the arithmetic progression *. The program must calculate a series from the smallest member of the progression to the larger.	n=16, d=2, a_1=20
18	Derive n members of the increasing geometric progression **. The program should display a series from the larger member of the progression to the smaller.	n=4, q=8, b_1=1
19	Print n members of the arithmetic progression *. The program should output a series from the smallest member of the progression to the larger.	n=11, d=5, a_1=-10
20	Calculate the sum of n terms of the increasing geometric progression **. The program must calculate a series from the smallest member of the progression to the larger.	n=7, q=4, b_1=4
21	Calculate the sum of n terms of the arithmetic progression *. The program must calculate a series from the larger member of the progression to the smaller.	n=17, d=5, a_1=-15
22	Derive n members of the increasing geometric progression **. The program should output a series from the smallest member of the progression to the larger.	n=10, q=2, b_1=1
23	Print n members of the arithmetic progression *. The program should display a series from the larger member of the progression to the smaller.	n=22, d=2, a_1=-5
24	Calculate the sum of n terms of the increasing geometric progression **. The program must calculate a series from the larger member of the progression to the smaller.	n=7, q=7, b_1=7

25	Calculate the sum of n terms of the arithmetic progression *. The program must calculate a series from the smallest member of the progression to the larger.	n=10, d=8, a_1=0

* – arithmetic progression: $\forall n > 1,\ a_n = a_1 + (n-1)d$.

** – geometric progression: $b_n = b_1 q^{n-1}$, ($b_1 \neq 0,\ q \neq 0,$).

4. Make a report on the research.

6.12. List of control questions on the task № 6

1. Name the types of repetitions that are implemented in the Prologue.
2. Define the term "recursive procedure".
3. Name the main advantages and disadvantages of recursion.
4. Where are the values of variables stored in recursion?
5. What is the difference between tail recursion and normal recursion?
6. Can I change the value of a variable in VisualProlog?

6.13. Task № 7. Lists and recursion

Lists

Processing lists, ie objects that contain any number of elements, is a powerful tool of Prolog. This lab explains what lists are and how to declare them, and provides examples of how you can use list processing in tasks. Two well-known predicates of the Prologue are defined – member and append (association) when considering procedural and recursive aspects of list processing. Also, the standard predicate Visual Prolog – findall is defined, which allows you to find and collect all solutions for one purpose. At the end of the laboratory work, compiled lists are considered, ie lists that contain combinations of elements of different types, and examples of grammatical analysis of lists.

In the Prologue, a list is an object that contains a finite number of other objects. Lists can be compared to arrays in other languages, but unlike arrays, lists do not need to be declared in advance. Of course, there are other ways to combine several objects into one. If the number of objects is known in advance, it is possible to make them arguments of one composite data structure. If the number of objects is not defined, you can use a recursive composite data

structure, such as a tree. But working with lists is usually easier because Visual Prolog provides a clearer entry for them.

The list containing the numbers 1, 2 and 3 is written as follows: [1, 2,3]

Each component of the list is called an element. To format a data structure of the list type, you need to separate the elements of the list with commas and mark them in square brackets.

Examples: [dog, cat, canary]

[“Valerie ann”, “Jennifer caitlin”, “benjamin thomas”]

Announcement of lists

To declare a domain for the list of integers, you must use a domain declaration, such as:

```
domains
integerlist = integer *
```

The symbol "*" means "list of something", so integer * means "List of Integers". It should be noted that the word "list" has no special meaning in Visual Prolog. The Zanzibar list can be called with the same success. The notation * (not the name) tells the compiler that this is a list.

List items can be any, including other lists. However, all its elements must belong to one domain. The domain declaration for the elements should look like this:

```
domains
    elementlist = elements*
    elements = ...
```

In this case, the elements have a single type (for example: integer, real or symbol) or are a set of different elements, denoted by different functors. You cannot mix standard types in a list in Visual Prolog. For example, the following declaration incorrectly identifies a list of elements that are integers and real numbers or identifiers:

```
elementlist = elements*
elements = integer; real; symbol       /* Wrong */
```

Щоб оголосити список, складений з цілих, дійсних і ідентифікаторів, треба визначити один тип, який включає всі три типи з функторами, які покажуть, до якого типу належить той або інший елемент. Наприклад:

```
elementlist = elements*
```

elements = i(integer); r(real); s(symbol)   % функтори тут i, r та s

Heads and tails

The list is a recursive composite object. It consists of two parts - the head, which is the first element, and the tail, which is a list that includes all subsequent elements. The tail of the list is always the list, the head of the list is always the element.

Example:

the head of the list [a, b, c] is a;

the tail of the list [a, b, c] is [b, c].

If the list is one-item, then:

the head of the list [c] is c;

the tail of the list [c] is [].

If you select the first item in the list enough times, you will definitely come to an empty list []. An empty list cannot be divided into head and tail. Conceptually, this means that the list has the structure of a tree, like other composite objects.

The structure of the tree [a, b, c, d] is shown in Fig. 6.43.

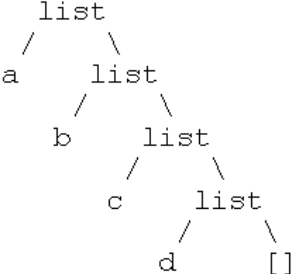


Fig. 6.43. Tree structure [a, b, c, d]

A one-element list, such as [a], is not the same as the element it contains, because [a] is actually a complex data structure, as shown in Fig. 6.44.

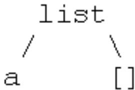


Fig. 6.44. Compound data structure

Working with lists

The Prologue has a way of clearly separating the head from the tail. Instead of separating the elements with commas, this can be done with a vertical line "|".

Example:

[a, b, c] is equivalent to [a | [b, c]] and, continuing the process,
[a | [b, c]] is equivalent to [a | [b | [c]]], which is equivalent to [a | [b | [c | []]]].

You can use both types of delimiters in the same list, provided that the vertical bar is the last delimiter. If desired, you can rewrite [a, b, c, d] as [a, b | [c, d]]. Table 6.7 shows other examples.

Table 6.7. Heads and tails of lists

List	Head	Tail
[“a“,“b“,“c“]	“a“	[“b“,“c“]
[“a“]	“a“	[ ] % empty list
[ ]	Not defined	Not defined
[[1, 2, 3], [2, 3, 4], [ ]]	[1, 2, 3]	[[2, 3, 4], [ ]]

Table 6.8 shows some examples of assignments in lists.

Table 6.8. Assignment in lists

<table>
<tr><th>List</th><th>List</th><th>Assigning variables</th></tr>
<tr><td>[X, Y, Z]</td><td>[Albert, eat ice cream]</td><td>X = Albert, Y = , eat, Z = ice cream</td></tr>
<tr><td>[7]</td><td>[X | Y]</td><td>X = 7, Y = [ ]</td></tr>
<tr><td>[1, 2, 3, 4]</td><td>[X, Y | Z]</td><td>X = 1, Y = 2, Z = [3, 4]</td></tr>
<tr><td>[1, 2]</td><td>[3 | X]</td><td>fail %</td></tr>
</table>

Using lists

The list is a recursive composite data structure, so you need algorithms to process it. The main way to process a list is to view and process each of its elements until the end is reached. An algorithm of this type usually requires two sentences. The first says what to do with a regular list (a list that can be divided into head and tail), the second – what to do with an empty list.

Print lists

If you want to print list items, do so as shown in Example 6.15.

Example 6.15.

```
domains
        list = integer*                    % or any other type
predicates
        write_a_list(list)
clauses
        write_a_list([]).     % if the list is empty - do nothing
        write_a_list([H|T]):-       assign H – head, T-хвіст, tail,
then...
                write(H), nl,
                write_a_list(T).
goal
        write_a_list([1, 2, 3]).
```

The following are two target statements write_a_list, described in plain language: Printing an empty list means doing nothing. Otherwise, to print a list means to print its head (which is a single element), then print its tail (list). At first glance, the target statement is as follows:

write_a_list ([1, 2, 3]).

It satisfies the second sentence at H = 1 and T = [2, 3]. The computer will print 1 and call recursively write_a_list:

write_a_list ([2, 3]).   % це write_a_list (T)

This recursive challenge satisfies the second sentence. This time H = 2 and T = [3], so that the computer prints 2 and again recursively calls write_a_iist with the target statement

write_a_list ([3]).

List [3] has only one element, it has a head 3 and a tail []. This means that the target statement again fits the second sentence with H = 3 and T = []. The program prints 3 and makes a recursive call:

write_a_list ([ ]).

Now it is clear that the first sentence is suitable for this target statement. The second sentence is not appropriate because [] cannot be divided into head and tail. Thus, if there were no first sentence, the target statement would be

unfeasible. But the first sentence is appropriate, and the target statement is executed without further action.

Counting list items

Consider how you can determine the number of items in a list. What is the length of the list? Here is a simple logical definition:

Length [] - 0.

The length of any other list is 1 plus the length of its tail.

To apply this in the Prologue requires two sentences (example 6.16).

Example 6.16.

```
domains
        list = integer*                         % or any other type
predicates
        length_of(list,integer)
clauses
        length_of([ ], 0).
        length_of([_|T],L):-
                length_of(T,TailLength),
                L = TailLength + 1.
```

Let's look at the second sentence first. Indeed, [_ | T] can be compared with any non-empty list, with the assignment of "T" tail list. The value of the head is not important, the main thing is that it is, and the computer can count it as one element.

Thus, the target statement length_of ([1, 2, 3], L). fits the second sentence at T = [2, 3]. The next step will be to calculate the length "T". When this is done (no matter how), TailLength will have a value of 2, and the computer will add 1 to it, and then assign an "L" to 3.

So how does a computer perform an intermediate step? This is the step in which the length [2, 3] is determined when executing the target statement length_of ([2, 3], TailLength). In other words, length_of calls itself recursively. This target statement fits the second sentence with the assignment:

- [3] from the target statement is assigned "T" in the sentence;

- TailLength from the target statement is assigned an "L" in the sentence.

Recall that TailLength in the target statement does not match TailLength in the sentence, because each recursive call in the rule has its own set of variables. Therefore, the objective statement is to find the length [3], ie 1, and then add 1 to the length [2, 3], ie up to 2, etc.

Thus, length_of calls itself recursively to get the length of the list [3]. Tail [3] - [], so that "T" will be assigned [], and the target statement will be to find the length [] and, adding 1 to it, get the length [3].

This time everything is simple. The target statement length_of ([], TailLength) satisfies the first sentence, which will assign 0 to the variable TailLength. Visual Prolog will add 1 to it and get the length [3], then return to the calling sentence. It in turn will add 1 again, will get length [2, 3] and will return to the sentence causing it. This initial sentence will again add 1 and get the length [1, 2, 3].

The following is an illustration of all the challenges:

```
length_of ([1,2,3], L1).
length_of ([2, 3], L2).
length_of ([3], L3).
length_of ([], 0).
L3 = 0 +1 = 1.
L2 = L3 +1 = 2.
L1 = L2 +1 = 3.
```

Exercise 6.1.

1. Write a predicate called sum_of, which works the same as length_of, except that it works with a list of numbers and summarizes them. For example, the target statement:

```
sum_of ([1,2,3,4], S).
```

must assign an "S" value of 10.

2. What will happen if you try to fulfill the following target statement:

```
sum_of (List, 10).
```

This targeted statement requires: "Make me a list to which you need to add 10". Can this be done in Visual Prolog? If not, why not? (Hint: You can't perform arithmetic operations on unrelated variables in Visual Prolog).

Tail recursion

length_of is not and cannot be tail recursion, because a recursive call is not the last step in its sentence. The problem with using length_of is that you can't calculate the length of the list until the tail length is calculated. But there is a workaround. To determine the length of the list, you need to use a predicate with three arguments:

- the first is the list itself, which the computer reduces with each call until the list is emptied in the same way as before;
- second – free parameter that will save the intermediate result (length);
- third – the counter, which starts from zero and increases by 1 for each call.

When the list becomes empty, the counter is unified with a free result. Consider an example (example 6.17).

Example 6.17.

```
domains
        list = integer*                         % or any other type
predicates
        length_of (list, integer, integer)
clauses
        length_of ([], Result, Result).
        length_of ([_ | T], Result, Counter):-
                NewCounter = Counter + 1,
                length_of (T, Result, NewCounter).
goal
        length_of ([1,7,3,2,5,1], L, 0),        % start with the counter
= 0
        write ("L =", L), nl.
```

This version of the predicate length_of is more complex and less logical than the previous one. It is shown only to show that tail recursive algorithms can be found for target statements that may require a different type of recursion.

Exercise 6.2.

1. Try both length_of versions for very large lists (such as 200 or 500 items). What will happen? How do the two versions of the long lists relate to speed?
2. Rewrite sum_of like the new version of length_of.

Sometimes you need to turn one list into another. This is done with the list element by element, replacing each element with the calculated values. The program in example 6.18 will add 1 to each element of the numerical list.

Example 6.18.

```
domains
        list = integer*
predicates
        add1(list,list)
clauses
        add1([], []).                   % boundary condition
        add1([Head|Tail],[Head1|Tail1]):-          %   separate   the
head of list
        Head1= Head+1,                  % add 1 to 1 item
```

```
        add1(Tail,Tail1).  % call an item from the rest of the
    list
```

Translating this into natural language, we get: to add 1 to all elements of an empty list, you need to create another empty list. To add 1 to all elements of any non-empty list, add 1 to the head and make the resulting element the head of the resulting list, then add 1 to each element of the tail of the list and make it the tail of the result.

Enter the program and run the Test Goal with the target statement add1 ([1,2,3,4], NewList). Test Goal will show the result:

```
NewList = [2, 3, 4, 5]
1 Solution
```

The add1 predicate performs the following operations:

1. Divides the list into Head and Tail.
2. Adds 1 to Head and assigns the result to Head1.
3. Recursively adds 1 to all Tail elements, assigns the result Tail1.
4. Combines Head1 and Tail1 and assigns the result to a new list.

This procedure is not a tail recursion, because a recursive call is not the last step. But what's important - Visual Prolog does it wrong, in it add1 is a tail recursion, because the steps are actually as follows:

1. Link the head and tail of the original list with Head and Tail.
2. Associate the head and tail of the result with Head1 and Tail1 (Head1 and Tail1 have not yet been determined).
3. Add 1 to Head and assign the result to Head1.
4. Recursively add 1 to all Tail elements, assigning the result Tail1.

When all is done, Head1 and Tail1 are already the head and tail of the result, and there is no separate operation to combine them. Thus, a recursive call is the last step.

Of course, it is not always necessary to replace every element. The following is example 6.19, in which the program views a list of numbers and makes a copy of it, discarding negative numbers.

Example 6.19.

```
domains
    list = integer*
predicates
    discard_negatives (list, list)
clauses
    discard_negatives ([], []).
    discard_negatives ([H | T], ProcessedTail):-
        H <0,
```

```
                !,
                discard_negatives (T, ProcessedTail).
        discard_negatives ([H | T], [H | ProcessedTail]):-
                discard_negatives (T, ProcessedTail).
```

For example, the target statement discard_negatives ([2, -45, 3, 468], X) will get X = [2, 3, 468].

Next is a predicate that copies list items, causing each item to appear twice:

```
doubletalk ([], []).
doubletalk ([H | T], [H, H | DoubledTail]):-
doubletalk (T, DoubledTail).
```

Belonging to the list

Suppose there is a list of names John, Leonard, Eric and Frank, and you need to use Visual Prolog to check if the specified name is in this list. In other words, you need to find out the relationship (affiliation) between two arguments – the name and the list of names. This is expressed by the following predicate:

```
member (name, namelist).        % «name» belongs to   «namelist»
```

In example 6.20, the first sentence checks the head of the list. If the head of the list matches the name you are looking for, you can conclude that Name belongs to the list. Since the tail of the list is not of interest, it is denoted by an anonymous variable. According to the first sentence, the target statement is member (john, [john, leonard, eric, frank]). will be performed.

Example 6.20

```
domains
        namelist = name*
        name = symbol
predicates
        member (name, namelist)
clauses
        member (Name, [Name | _]).
        member (Name, [_ | Tail]):-
                member (Name, Tail).
```

If the head of the list does not match Name, you need to check if Name can be found in the tail of the list. In plain language: Name belongs to the list if Name is the first item in the list, or Name belongs to the list if Name belongs to the tail.

The second member sentence, which expresses the relationship of affiliation, in Visual Prolog looks like this:

```
member (Name, [_ | Tail]):-
        member (Name, Tail).
```

Exercise 6.3.

1. For example 4.6, try the following goal statement using Test Goal:

```
member (susan, [ian, susan, john]).
```

2. Add statements to the domain and predicate sections so that you can use member to determine whether a number belongs to a numeric list. Try a few targeted statements, including the following:

```
member (X, [1,2, 3, 4]).
```

to test your new program.

3. Does the order of writing two sentences matter for the member predicate? Check how the program behaves if you swap two sentences. Will there be a difference in fulfilling the following target statement for both cases?

```
member (X, [1,2,3,4,5])
```

Merge lists

As the member predicate given in Example 4.6, it works in two ways. Consider his sentence again:

```
member (Name, [Name | _]).
        member (Name, [_ | Tail]):-
        member (Name, Tail).
```

These sentences can be viewed from two different points of view: declarative and procedural.

- from the declarative point of view, the sentences state the following: Name belongs to the list if the chapter coincides with Name; if not, Name belongs to the list, if it belongs to its tail;

- from a procedural point of view, these two sentences can be interpreted as follows: to find an element of the list, you need to find its head; otherwise you need to find the element in the tail.

These two points of view are correlated with the target statements:

member (2, [1,2,3,4]).

member (X, [1,2,3,4]).

As a result, the first target statement "asks" Visual Prolog to find out whether the statement is true, the second – to find all members of the list [1,2,3,4]. The predicate member is the same in both cases, but its behavior can be viewed from different points of view.

Recursion from a procedural point of view

The peculiarity of the Prologue is that often when you specify a sentence for a predicate from one point of view, they will be performed from another. To see this duality, create a predicate in the following example (example 4.7) to join one list to another. Define the predicate append with three arguments:

append (List1, List2, List3)

It combines List1 and List2 and creates List3. Once again we will use recursion (this time from the procedural point of view). If List1 is empty, List2 will be the result of merging List1 and List2. On the Prologue it will look like this:

append ([], List2, List2).

If List1 is not empty, you can combine List1 and List2 to form List3, making List1 head List3 head. (In the following statement, the variable H is used as the head for List1 and for List3.) The tail of List3 is L3, it consists of combining the remainder of List1 (ie L1) and the entire List2. That is:

```
append ([H | L1], List2, [H | L3]): -
append (L1, List2, L3).
```

The append predicate is executed as follows: until List1 is empty, the recursive sentence passes one element to List3. When List1 becomes empty, the first sentence unifies List2 with the resulting List3.

Exercise 6.4.

The **Append** predicate is defined in example 6.21. Download the program from example 6.21.

Example 6.21.

```
domains
    integerlist = integer*
```

```
predicates
        append(integerlist,integerlist,integerlist)
clauses
        append([],List,List).
        append([H|L1],List2,[H|L3]):-
                append(L1,List2,L3).
```

Run the following target statement:

```
append([1,2,3],[5,6],L)
```

And then try this:

```
append ([1,2], [3], L), append (L, L, LL).
```

Number of options for using predicates

Considering append from a declarative point of view, you have identified the relationship between the three lists. However, this relationship will be maintained even if List1 and List3 are known and List2 is not. It is also true if only List3 is known. For example, to determine which of the two lists can be combined to obtain a known list, you need to make a target statement of the following type:

```
append (L1, L2, [1,2,4]).
```

Visual Prolog will find the following solutions for this target statement:

```
L1 = [], L2 = [1, 2, 4]
L1 = [1], L2 = [2, 4]
L1 = [1, 2], L2 = [4]
L1 = [1, 2, 4], L2 = []
4 Solutions
```

You can also use append to determine which list can be attached to [3, 4] to get a list [1, 2, 3, 4]. Run the following target statement:

```
append (L1, [3,4], [1,2,3,4]).
```

Visual Prolog will find a solution:

```
L1 = [1, 2].
```

The append predicate defined the relationship between the input set and the output set so that the relationship can be applied in both directions.

The state of the arguments when the predicate is called is called the parameter stream. An argument that is assigned or assigned at the time of the call is called an input argument and is denoted by the letter "i"; and a free argument is the original argument, denoted by the letter "o".

The append predicate has the ability to work with different parameter streams, depending on what output you give it. However, not all predicates can be called with different parameter streams. If the sentence of the Prologue can be used with different streams of parameters, it is called a reversible sentence (Russian: "reversible sentence"). When writing your own sentences in Visual Prolog, keep in mind that reversible sentences have added benefits, and creating them adds "power" to predicates.

Find all solutions to the goal at once

The advantage of recursion is that, unlike a return search, it transmits information (through parameters) from one recursive call to the next. Therefore, a recursive procedure may store memory for intermediate results or counters as it is executed.

But there is one thing that a return search can do and recursion can't. This is the search for all alternative solutions in the target statement. It may turn out that all solutions are needed for the target statement, and they are needed all at once, as part of a single complex composite data structure. The built-in predicate findall uses target statements as one of its arguments and gathers all solutions for that target statement into a single list. The predicate findall has three arguments:

VarName (variable name) – determines the parameter to be compiled in the list;

myPredicate – determines the predicate from which to collect values;

ListParam – must specify the type to which the ListParam values belong. Example 6.22 shows a program that uses findall to print the arithmetic mean of a group of people.

Example 6.22.

```
domains
        name, address = string
        age = integer
        list = age*
predicates
        person (name, address, age)
        sumlist (list, age, integer)
clauses
        sumlist ([], 0,0).
        sumlist ([H | T], Sum, N):-
                sumlist (T, S1, N1),
```

```
            Sum = H + S1,
            N = 1 + N1.
        person ("Sherlock Holmes", "22B Baker Street", 42).
        person ("Pete Spiers","Apt. 22, 21st Street", 36).
        person ("Mary Darrow", "Suite 2, Omega Home", 51).
goal
        findall (Age, person (_, _, Age), L),
        sumlist (L, Sum, N),
        Ave = Sum / N,
        write ("Average =", Ave), nl.
```

The findall sentence in this program creates a list of L, which contains all the parameters that indicate the age derived from the predicate person. If you want to compile a list of all people who are 42 years old, you should fulfill the following sub-target statement:

```
findall (Who, person (Who, _, 42), List)
```

But this sub-goal requires the program to contain domain ads for the resulting list:

```
slist = string*
```

Compiled lists

The list of integer values can be declared simply:

```
integerlist = integer*
```

The same is true for a list of real numbers, a list of identifiers or a list of strings. It is often important to have a combination of elements of different types within one list:

```
[2,3,5.12, ["food", "goo"], "new"]      % incorrect in Visual Prolog
```

Composite lists are lists that use more than one type of item. Special declarations are required to work with lists that contain different types of items, because Visual Prolog requires that all items in the list belong to the same type. To create a list that could store different types of elements, you need to use functors in the Prologue, because a domain can contain more than one data type as arguments for functors.

An example of a domain ad for a list, which may contain characters, goals, strings, or lists, is as follows:

```
domains              % functors l, i, c and s
        Hist = l(list); i(integer); c(char); s(string)
        list = llist*
```

List

```
[2,9, ["food", "goo"], "new"]           % incorrect in Visual Prolog
```

should be presented in Visual Prolog as:

```
[i(2), i(9), l([s ("food"), s("goo")]), s("new")]       % correct
```

The following is an example (Example 6.23) that shows the merging of lists and the use of domain declarations in the typical case of lists.

Example 6.9.

```
domains
        llist = l(list); i(integer); c(char); s(string)
        list = llist*
predicates
        append (list, list, list)
clauses
        append ([], L, L).
        append ([X | L1], L2, [X | L3]):-
                append (L1, L2, L3).
goal
        append ([s (likes), l([s (bill), s (mary)])],[ s (bill), s (sue)],
Ans),
        write ("First list:", Ans, "\ n \ n"),
        append ([l([s ("This"), s ("is"), s ("a"), s ("list")]), s (bee)], [c
('c')], Ans2),
        write ("Second list:", Ans2, '\n', '\n').
```

Task

1. Run the programs shown in the examples (example 6.16 - example 6.23) in Visual Prolog 5.2.

2. Perform the tasks formulated in the exercises (exercise 6.1 - exercise 6.4) in Visual Prolog 5.2.

3. According to the variant of the task (tables 6.9 - 6.10), write three programs in the Prologue language. The option number is determined by the serial number of the student in the teacher's journal.

Table 6.9. Task options

Task number	Task number 1	Task number 2	Task number 3

1	1	7	17
2	2	8	18
3	3	9	19
4	4	10	20
5	5	11	21
6	1	12	22
7	2	13	17
8	3	14	18
9	4	15	19
10	5	16	20
11	1	8	21
12	2	9	22
13	3	10	17
14	4	11	18
15	5	12	19
16	1	13	20
17	2	14	21
18	3	15	22
19	4	16	17
20	5	7	18
21	1	9	22
22	2	10	21
23	3	11	17
24	4	12	20
25	5	13	17

Table 6.10. Task

Task number	Task
1	Display the entire list
2	Calculate the length of the list of integers
3	Calculate the length of the list of real numbers
4	Calculate the sum of the elements of the list of integers
5	Calculate the sum of the elements of the list of integers
6	Add the number x to each list item (list length n)
7	Display all elements of the list that satisfy the condition: x> 0
8	Display all elements of the list that satisfy the condition: x < 0
9	Display all items in the list that satisfy x>=0
10	Print all elements of the list that satisfy the condition: x <= 0

11	Print all elements of the list that satisfy the condition: x ≠ 0
12	Duplicate all list items that satisfy the condition: x> 0
13	Duplicate all elements of the list that satisfy the condition: x <0
14	Duplicate all list items that satisfy the condition: x> = 0
15	Duplicate all list items that satisfy the condition: x <= 0
16	Duplicate all elements of the list that satisfy the condition: x ≠ 0
17	Check the presence of a given item in the numerical list and, if found, determine its position in the list
18	Check the presence of a given item in the string list and, if found, determine its position in the list
19	Calculate the arithmetic mean of the elements of the list of real numbers
20	Calculate the arithmetic mean of the elements of the list of integers
21	Find the maximum item in the list
22	Find the minimum list item

6.14. List of control questions on the task № 7

1. Define the concept of a list.
2. What are the components of the list?
3. How is the structure of the data type list?
4. What symbol indicates that a list has been created?
5. What parts does the list consist of?
6. What is the way in Visual Prolog to separate the head from the tail?
7. Why is the length of any list?
8. What is the state of the arguments when calling a predicate?
9. Define the terms "input" and "output argument".
10. Which sentence is called reversible?
11. What can a return search do in contrast to recursion, and what built-in predicate is used to solve this problem?
12. Define the concept of "compiled list".
13. What are folded functors for?

6.15. Task № 8. Internal database of facts

Visual Prolog internal facts base This lab will look at how to declare internal fact databases and how you can change the content of your internal facts database. The internal fact database consists of facts that can be added and removed from the program on Visual Prolog during its execution. It is possible to declare predicates that describe the internal database in the facts section of the

program and apply these predicates in the same way as the predicates described in the predicates section.

In order to add new facts to the database, Visual Prolog uses the predicts assert, asserta, assertz, and the predicates retract and retractall are used to delete existing facts. You can change the contents of a fact database by first deleting a fact and then inserting a new version of that fact (or a completely different fact). The predicates consult / 1 and consult / 2 read the facts from the file and add them to the internal database, and save / 1 and save / 2 store the contents of the internal database of facts in the file.

Visual Prolog interprets database facts in the same way as regular predicates. The facts of the predicates of the internal database of facts are stored in a table that can be easily modified, while the usual predicates, to achieve maximum speed, are compiled into binary code.

Announcement of the internal database of facts

The keyword facts (synonymous with the obsolete word database) determines the beginning of the declaration of the facts section. The facts section consists of a sequence of predicate declarations that describe the corresponding internal facts base. During execution, you can use the asserta and assertz predicates to add facts (but not rules) to the fact base. Or, by calling the standard consult predicate, you can get the facts from a file on disk. An example of using the facts section is given below (example 6.24):

Example 6.24.

```
domains
        name, address = string
        age = integer
        gender = male; female
facts
        person (name, address, age, gender)
predicates
        male (name, address, age)
        female (name, address, age)
        child (name, age, gender)
clauses
        male (Name, Address, Age):-
        person (Name, Address, Age, male).
```

In this example, you can use the predicate person in the same way as other predicates (male, female, child). The only difference is that it is possible to add and remove facts for the person predicate while the program is running.

The following two restrictions on predicates announced in the facts section should be noted:

- it is allowed to add to the database only facts, but not rules;
- database facts cannot contain free variables.

Multiple fact sections are allowed, but you must explicitly specify the name of each fact section.

```
facts - mydatabase
        myFirstRelation (integer)
        mySecondRelation (real, string)
        myThirdRelation (string)
/ * etc .* /
```

The facts section description named mydatabase creates a fact database named mydatabase. If you do not name the internal fact database, it is given the default name dbasedom by default. The program can also contain local anonymous fact sections only if it consists of a single module that is not declared as part of the project. The Visual Development Environment (VDE) compiles the program file as a single module only when using the Test Goal utility. Otherwise, the anonymous section of facts must be declared global. To do this, put the keyword global in front of the keyword facts.

The names of the predicates of the fact database must be unique in the module (source file), the same predicate names cannot be used in two different sections of facts. Similarly, you cannot use the same predicate names in the facts and predicates sections. However, the names of the predicates defined in the local facts sections are local to the module where they are declared and do not conflict with the local names of the predicates or facts declared in other modules.

Use of internal fact bases

Because Visual Prolog presents a relational database as a collection of facts, you can use it as a powerful query language for internal fact databases. The Visual Prolog unification algorithm automatically selects facts with the correct values for known arguments and assigns values to unknown arguments until its return search algorithm returns all the solutions for a given query.

Access to the internal evidence base

Predicates belonging to the internal database of facts are available in the same way as other predicates. The only visible difference is that the announcements of such predicates are located in the facts section instead of the predicates section.

Example 6.25.

```
domains
        name = string
```

```
        sex = char
facts
        person (name, sex)
clauses
        person ("Helen", 'F').
        person ("Maggie", 'F').
        person ("Suzanne", 'F').
        person ("Per", 'M').
```

In Example 6.25, you can call person with the name person ('F') to find all women, or person ('Maggie', 'F') to verify that a woman named "Maggie" exists in your database. By their nature, predicates in the facts section are always indeterminate. Since facts can be added at any time during program execution, the compiler should always keep in mind that there is a possibility to find alternative solutions during the return search. If there is a predicate in the facts section for which there will never be more than one fact, you can declare it by writing the determ keyword before declaring the fact predicate (or the single keyword if the predicate must always have one and only one fact):

```
facts
        determ daylight_saving (integer)
```

Note that when you try to add a new fact to a deterministic fact base predicate that already has a fact, Visual Prolog will give an error**.**

Update the internal facts base

Facts for fact base predicates can be determined at compile time in the clauses section, as shown in Example 5.2. During execution, facts can be added and removed using the predicates described below. It should be noted that the facts identified at compile time in the clauses section can also be deleted, they are no different from the facts added at runtime.

Standard Visual Prolog predicates for working with facts: assert, asserta, assertz, retract, retractall, consult and save – can have one or two arguments. The optional second argument is the name of the internal evidence base. The notation "/ 1" and "/ 2" after each predicate name indicates the required number of arguments for this version of the predicate. Comments (such as "/ * (i) * /" and "/ * (o, i) * /") show the stream (s) of parameters for this predicate.

Entering facts during the program

At runtime, facts can be added to the internal fact database using predicates: assert, asserta and assertz, or by downloading facts from a file using consult. There are three predicates for adding one fact at runtime:

asserta (<the fact>) % (i)
asserta (<the fact>, facts_sectionName) % (i, i)
assertz (<the fact>) % (i)
assertz (<the fact>, facts_sectionName) % (i, i)
assert (<the fact>) % (i)
assert (<the fact>, facts_sectionName) % (i, i)

The **asserta** predicate inserts a new fact into the fact database before the available facts for that predicate, and the assertz inserts the facts after the available facts for that predicate. Using the assert predicate gives a result similar to using assertz.

Because fact base predicate names are unique within the program (for global fact sections) or module (for local fact sections), asserta and assertz predicates always know which fact database to add the fact to. However, an optional second argument can be used to ensure that the required fact database is used to verify the type.

The first predicate of example 6.26 will insert the fact about "Suzanne", described by the predicate person, after all the facts of person, currently stored in memory. The second is the fact of "Michael" before all the available facts of the predicate person. The third is the fact of "John" after all the other likes in the likesDatabase, and the fourth is to insert the fact of "Shannon" in the same facts before all the other likes.

Example 6.26.

```
assertz (person ("Suzanne", "New Haven", 35)).
asserta (person ("Michael", "New York", 26)).
assertz (likes ("John", "money"), likesDatabase).
asserta (likes ("Shannon", "hard work"), likesDatabase).
```

After calling these predicates, the fact base will look as if you have started working with the following facts:

% **Internal database of facts** – dbasedom
    person ("Michael", "New York", 26).
% ... **Other facts** person ...
    person ("Suzanne", "New Haven", 35).
% **Internal database of facts** – likesDatabase
    likes ("Shannon", "hard work").
% ... **Other facts** likes ...
    likes ("John", "money").

Beware of accidentally written code that states the same fact twice. Internal fact databases do not provide for any uniqueness, so the same fact can appear in the internal fact database many times. However, it is very easy to write the assert version with uniqueness check:

```
facts - people
        person (string, string)
predicates
        uassert (people)
clauses
        uassert (person (Name, Address)):-
        person (Name, Address),
        !,
        % OR
        assert (person (Name, Address)).
```

Read facts from a file

The consult predicate reads the facts described in the facts section from the fileName file and inserts them into the program at the end of the corresponding fact database. The consult predicate has one or two arguments:

```
consult (fileName)% (i)
consult (fileName, databaseName)     % (i, i)
```

However, unlike assertz, if you call consult with only one argument (without the fact base name), only the facts described in the untitled section (by default dbasedom) will be read. If you call consult with two arguments (file name and fact base name), only the facts from the specified fact base will be checked. If the file contains anything other than the facts of the specified database, the consult predicate will return an error when it reaches this line.

It should be noted that the consult predicate reads one fact at a time. If the file contains ten facts and there is a syntax error in the seventh fact, the consult will enter the first six facts in the fact database and then issue an error message.

It should be noted that the consult predicate can only read files in the format created by save. Files must not contain:

- uppercase characters, except those contained inside double-quoted lines;
- gaps, except those contained inside double-quoted rows;
- comments;
- blank lines;

Symbol identifiers (symbol) without double quotes.

Delete facts during program execution

The retract predicate unifies the facts and removes them from the internal facts base. It has the following format:

```
retract (<the fact>)          % (i)
retract (<the fact>, databaseName)          % (i, i)
```

The retract predicate removes the first fact from your database, which coincides with <the fact>, by associating free <the fact> variables during program execution. Deleting facts from the internal facts base is equivalent to the process of accessing them with the side effect of deleting unified facts. retract is indeterminate if the predicate predicate to be removed by retract has not been declared deterministic. When searching with a return, the retract predicate deletes all unified facts as long as they exist, after which it no longer finds the required facts and fails.

Example 6.27.
Assume that the program has the following sections facts.

```
facts
        person (string, string, integer)
facts - likesDatabase
        likes (string, string)
        dislikes (string, string)
clauses
        person ("Fred", "Capitola", 35).
        person ("Fred", "Omaha", 37).
        person ("Michael", "Brooklyn", 26).
        likes ("John", "money").
        likes ("Jane", "money").
        likes ("Chris", "chocolate").
        likes ("John", "broccoli").
        dislikes ("Fred", "broccoli").
        dislikes ("Michael", "beer").
```

With the following facts, Visual Prolog sections, you can set the following goals:

```
retract (person ("Fred ",_,_)),    % 1
retract (likes (_, "broccoli")),          % 2
retract (likes (_, "money"), likesDatabase),          % 3
retract (person ("Fred ",_,_), likesDatabase).          % 4
```

The first goal is to remove the first person is Fred fact from the dbasedom fact database. The second target will remove the first fact that matches likes (X,

"broccoli") from the likesDatabase. For both purposes, Visual Prolog knows from which database to delete, because the names of the predicates of the fact database are unique: the predicate person is only in the unnamed database of facts, a likes – only in the database likesDatabase. The third and fourth goals show how the second argument can be used for type checking. The third goal is successfully implemented by removing the first fact that matches likes (_, "money") from likesDatabase, and the fourth goal will give an error because there is no (and cannot be) fact person in the likesDatabase fact database. The following goal illustrates how you can get values from a retract predicate:

```
retract (person (Name,Home, Age)),
write (Name, ",", Age), nl,
fail.
```

When a fact base name is specified as the second retract argument, you may not specify the name of the fact base predicate from which the facts are removed. In this case, retract will search for and delete all facts in the specified database. Example:

```
goal
        retract (X, mydatabase),
        write (X),
        fail.
```

Delete multiple facts at once

The retractall predicate removes from the fact base all facts that match the <the fact> pattern. The predicate retractall has the following format:

```
retractall (<the fact>)
retractall (<the fact>, databaseName)
```

The retractall acts similarly to the action given as follows

```
retractall (X):- retract (X), fail.
retractall (_).
```

but much faster.

Obviously, the retractall predicate always completes successfully. From retractall initial values cannot be received. This means that, as with not, you need to use an underscore for free variables. As with the assert and retract predicates, you can use a second argument to check the type.

And, as in the case of the retract predicate, if the retractall call uses an underscore, you can delete all the facts from the specified facts section.

The following goal removes all facts about men from the person facts database: retractall (person (_,_,_, male)).

The following goal removes all facts from mydatabase:

retractall (_, mydatabase).

Preservation of the database of facts during the program

The save predicate stores facts from the specified facts database in the file. This predicate has one or two arguments:

```
save (fileName)% (i)
save (fileName, databaseName)% (i, i)
```

If you call the save predicate with only one argument (without the fact base name), the file from the fileName will save the facts from the dbasedom fact database, which is used by default. When you call the save predicate with two arguments (file name and fact base name), the specified file will store the facts from the facts section of the fact database named databaseName..

Keywords that determine the properties of facts

You can use the following optional keywords in the facts section ad:

```
facts [- <databasename>]
[nocopy] [{nondeterm | determ | single}]
dbPredicate ['('[ Domain [ArgumentName ]]*')']
```

Optional keywords nondeterm, determ and single declare the determinism mode of the declared dbPredicate fact base predicate. Only one of them can be used. If the determinism mode of the predicate of the fact base is not explicitly specified, then the default value is nondeterm. Note that the nondeterm mode for fact base predicates is always set by default, regardless of the Default Predicate Mode check box in the Compiler Options VDE dialog box.

- nondeterm – specifies that the fact base can contain any number of facts for the dbPredicate fact base predicate. This is the default mode.

- determ – specifies that the fact database can contain no more than one fact for the dbPredicate fact database predicate.

- single – specifies that the fact database always contains one and only one fact for the dbPredicate fact database predicate.

- nocopy – usually when the fact base predicate is called to associate a variable with a string or compound object, the called data is copied from the

heap to the global Visual Prolog stack (GStack). nosoru declares that the data will not be copied, and the variables will refer directly to the fact data stored in the "heap". This can greatly increase efficiency, but if a copy has not been made, after deleting the fact, the variable will indicate some "garbage". Therefore, this approach should be used with caution.

- global – determines that the fact base is global. It should be noted that reliable programming techniques require the use of global facts. Instead, you can use global predicates that work with local facts.

Facts declared with the keyword nondeterm

The nondeterm keyword defines the default mode for facts (fact base predicates) declared in the facts section. If none of the words determ or single are used to declare the facts, the compiler uses nondeterm mode. Usually, by their nature, predicates of factual bases are indeterminate. Because facts can be added at any time during program execution, the compiler must keep in mind that alternative solutions may be found when searching with returns.

Facts announced with the keyword determ

The determ keyword specifies that the fact base can contain no more than one fact for the fact base predicate declared with that keyword. Therefore, if the program tries to establish the second such fact in the fact database, Prolog will generate an error. Therefore, the programmer should use special care with deterministic facts.

Declaring a fact deterministic allows the compiler to generate more efficient code, and when calling such predicates you will not be warned about a possible non-deterministic call. This is useful for flags, counters and other similar objects.

Particular attention should be paid to the fact that when deleting a fact that is declared determ, the call of non-deterministic predicates retract / 1 and retract / 2 will be deterministic. Therefore, if it is known that at any time the fact base contains no more than one fact counter, you can write:

```
facts
        determ counter (integer CounterValue)
goal
        ...
        retract (counter (CurrentCount)),
                % Пролог не встановить точку відкату Count =
        CurrentCount + 1,
        assert (counter (Count)),
```

замість

```
facts
        counter (integer CounterValue)
```

```
predicates
        determ retract_d (dbasedom)
clauses
        retract_d (X): - retract (X),!.          %          deterministic
predicate
goal
        ...
        retract_d (counter (CurrentCount)),
                % Пролог не встановить точку відкату Count =
        CurrentCount + 1,
        asserta (counter (Count)),
```

Facts declared with the keyword single

The single keyword specifies that the fact database always contains one and only one fact for the fact base predicate declared with the single keyword. Therefore, single facts must already be known when the program calls the target. Therefore, they must be initialized in the clauses sections of the program source code. Example:

```
facts - properties
        single numberWindows s (integer)
clauses
        numberWindows_s (0).
```

Single facts cannot be deleted. If you try to delete a single fact, the compiler will generate an error. In most cases, the compiler can detect an attempt to delete a single fact at compile time.

Since one instance of a single fact always exists, a single fact call never fails if it is called with free arguments. For example, the following call: numberWindows_s (Mum), never fails if Mum is a free variable. Therefore, it is convenient to use single facts in predicates declared with the procedure determinism type.

The predicates assert, asserta, assertz and consult, applied to the fact of single, act similarly to the pair of predicates retract and assert. Namely, the predicates assert (consult) change the existing instance of the fact to the specified new one. Using the single keyword before declaring a fact allows the compiler to make optimized code to access and modify the single fact. For example, for assert predicates applied to a single fact, the compiler generates code that works more efficiently than a pair of retract and assert predicates applied to a deterministic fact (and even more so than a pair of retract and assert predicates when used with normal (nondeterministic) fact).

Initializing single facts for some domains (for which there are no default values) is non-trivial. The following information may be useful:

- binary data domains can be initialized by assigning them a specific binary value. Example:

```
global domains
        font = binary
facts - properties
        single my_font (font)
clauses
        my_font ($ [00])
```

- another important special case is the initialization of single facts that contain the standard domain ref. The ref domain is a domain for reference numbers in external Visual Prolog databases, but it is also used in many specific domains declared in packages provided by Visual Prolog. For example, the main domain VPI window is declared as follows:

```
domains
        window = ref
```

It should be noted that to initialize the values of the domain ref, you can use unsigned numbers with the previous tilde character "(~)". For example, you can write:

```
facts
        single mywin (WINDOW)
clauses
        mywin (~ 0).
```

Examples of using the internal facts base

The following is a simple example (Example 6.28) of how to use an internal factual database to write a classification expert system. An important advantage of using the fact base in this example is that it is possible to add knowledge (and delete them) during the program.

Example 6.28.

```
domains
        thing = string
        conds = cond*
        cond  = string
facts - knowledgeBase
        is_a(thing,thing,conds)
        type_of(thing,thing,conds)
        false(cond)
```

```
predicates
        run(thing)
        ask(conds)
        update

clauses
        run(Item):-
                is_a(X,Item,List),
                ask(List),
                type_of(ANS,X,List2),
                ask(List2),
                write("The ", Item," you need is (a/an) ", Ans),nl.

        run(_):-
                write("This program does not have enough "),
                write("data to draw any conclusions."),
                nl.
        ask([]).
        ask([H|T]):-
                not(false(H)),
                write("Does this thing help you to "),
                write(H," (enter y/n)"),
                readchar(Ans), nl, Ans='y',
                ask(T).
        ask([H|_]):-
                assertz(false(H)), fail.
        is_a("language","tool",["communicate"]).
        is_a("hammer","tool",["build a house","fix a fender","crack a
nut"]).
        is_a("sewing_machine","tool",["make          clothing","repair
sails"]).
        is_a("plow","tool",["prepare fields","farm"]).
        type_of("english","language",["communicate with people"]).
        type_of("prolog","language",["communicate        with        a
computer"]).
        update:-
                retractall(type_of("prolog","language",
                        ["communicate with a computer"])),
                asserta(type_of("Visual Prolog","language",
                        ["communicate with a personal computer"])),
                asserta(type_of("prolog","language",
                        ["communicate with a mainframe computer"])).
```

The following facts could be entered using the predicate asserta or assertz, or read from the file using the predicate consult. However, in this example, they are located in the clauses section.

```
is_a (language, tool ["communicate"]).
is_a (hammer, tool, ["build a house", "fix a fender", "crack a nut"]).
is_a (sewing_machine, tool, ["make clothing", "repair sails"]).
is_a (plow, tool, ["prepare fields", "farm"]).
type_of (english, language, ["communicate with people"]).
type_of (prolog, language, ["communicate with a computer"]).
```

Enter as a goal:

```
goal
    run (tool).
```

Now enter the following goal:

```
update,
run (tool).
```

The update predicate is included in the source code of the program, deletes the fact type_of (prolog, language, ["communicate with a computer"]). from the internal knowledge base KnowBase and adds two new facts to the program:

```
type_of (prolog, language, ["communicate with a mainframe computer"])
type_of ("Visual Prolog", language, ["communicate with a personal computer"])
```

By calling the save/2 predicate with the names of the text file and the fact base as its arguments, you can save the entire knowledgeBase fact database in a text file. For example, after calling save ("mydata.dba", knowledgeBase) the mydata.dba file will be similar to the clauses section of a regular Visual Prolog program, and each fact will be written on a separate line.

Using the predicate consult, you can read the facts from this file into memory: consult ("mydata.dba", knowledgeBase). You can manipulate facts that describe the predicates of fact bases (facts described in the facts sections) as if they were terms. When declaring the fact base, Visual Prolog generates an internal domain corresponding to the facts from the facts section (example 6.29).

Example 6.29.

facts - dba1 % dba1 **domain for these predicates**

person (name, telno)

```
city (cno, cname)
```

Upon receiving the following declarations, the Visual Prolog compiler will generate the appropriate dba1 domain:

```
domains
        dba1 = person (name, telno); city (cno, cname)
```

This dba1 domain can be used just like any other domain. For example, you can use the standard readterm predicate to create a my_consuit predicate similar to the standard consult predicate.

Task

1. Implement the programs given in the examples (examples 6.27 - 6.28) in Visual Prolog 5.2.
2. According to the variant of the task (tables 6.11 - 6.12), using the program given in example 6.29, write two programs in the Prologue language. The option number is determined by the serial number of the student in the teacher's journal.

Example 6.29.

```
domains
        brand = string
        model = string
        year = integer
        color = string
        owner = string
facts
        car (brand, model, year, color, owner)

clauses
        car ("Skoda", "Fabia", 2008, "Silver", "Alexander").
        car ("Skoda", "Octava", 2005, "Green", "Georgy").
        car ("Skoda", "Octava", 2010, "White", "Denis").
        car ("Skoda", "Octava", 2006, "Blue", "Ivan").
        car ("Chevrolet", "Aveo", 2008, "Brown", "Oleg").
        car ("Chevrolet", "Captiva", 2007, "Brown", "Alexander").
        car ("Chevrolet", "Lacetti", 2007, "Brown", "Gennady").
        car ("Chevrolet", "Lacetti", 2010, "Red", "Vitaly").
        car ("Hyundai", "Sonata", 2006, "Red", "Nikita").
        car ("Hyundai", "Getz", 2006, "White", "Olga").
        car ("Hyundai", "Sonata", 2011, "Red", "Pavel").
        car ("Hyundai", "Elantra", 2008, "Red", "Oksana").
```

```
car ("Mazda", "323", 1994, "White", "Sergey").
car ("Mazda", "3", 2004, "Red", "Olga").
car ("Mazda", "6", 2011, "Red", "Pavel").
car ("Mitsubishi", "Lancer", 2004, "Gold", "Natalia").
car ("Mitsubishi", "Colt", 2005, "Blue", "Elena").
car ("Mitsubishi", "Lancer X", 2011, "Silver", "Nikita").
car ("Mitsubishi", "Lancer", 2008, "Silver", "Victor").
car ("Opel", "Vectra", 2007, "White", "Natalia").
car ("Opel", "Vectra", 2003, "Silver", "Elena").
car ("Opel", "Astra", 2011, "Silver", "Olga").
car ("Opel", "Kadet", 1991, "Red", "Gennady").
car ("Opel", "Kadet", 1993, "Black", "Nadezda").
car ("Opel", "Omega", 2004, "Silver", "Victor").
car ("Peugeot", "406", 1999, "Silver", "Natalia").
car ("Peugeot", "206", 2000, "Silver", "Elena").
car ("Peugeot", "307", 2008, "Silver", "Olga").
car ("Peugeot", "406", 1999, "Red", "Gennady").
car ("Peugeot", "206", 2001, "Black", "Nadezda").
car ("Peugeot", "605", 2001, "Silver", "Victor").
```

Table 6.11. Task options

№ option	Task number 1	Task number 2
1	1	12
2	2	13
3	3	14
4	4	15
5	5	16
6	6	17
7	7	18
8	8	19
9	9	20
10	10	21
11	11	22
12	12	23
13	13	24
14	14	25
15	15	1

16	16	2
17	17	3
18	18	4
19	19	5
20	20	6
21	21	7
22	22	8
23	23	9
24	24	10
25	25	11

Table 6.12. Task

Task number	Task
1	Remove from the "facts base" all cars that have the color of the body "Red" and which are cars manufactured by "Hyundai". After deletion, print the data from the "fact database".
2	Delete from the "facts base" the first fact that satisfies the conditions: the brand of the car "Opel", the year of manufacture of the car 2007. After deletion, remove the data from the "facts base".
3	Remove from the "fact base" all cars that have the color of the body "Brown" and manufactured in 2007. After deletion, print the data from the "fact database".
4	Remove the first satisfying fact from the "fact base": year of manufacture 2007, owner's name - "Alexander". After deletion, print the data from the "fact database".
5	Remove from the "fact base" all cars that have a body color "Silver". After deletion, print the data from the "fact database".
6	Remove from the "facts base" the first fact that satisfies the conditions: the car model "Octava", body color "White". After deletion, print the data from the "fact database".
7	Remove from the "fact base" all cars manufactured in 2004. After deletion, print the data from the "fact database".
8	Remove from the "facts base" the first fact that satisfies the conditions: the car brand "Peugeot", the owner of the car "Natalia". After deletion, print the data from the "fact database".
9	Remove from the "fact sheet" all cars manufactured by

	Peugeot, manufactured in 2001. After deletion, print the data from the "fact database".
10	Remove from the "facts base" the first fact that satisfies the conditions: the car brand "Skoda", the owner of the car "Denis". After deletion, print the data from the "fact database".
11	Remove from the "facts base" all cars that have the color of the body "Silver" and made in 2008. After deletion, print the data from the "fact database".
12	Delete from the "facts database" the first fact that satisfies the conditions: the car brand "Chevrolet", year of manufacture 2007. After deletion, remove the data from the "facts database".
13	Remove from the "fact base" all cars manufactured in 2008. After deletion, print the data from the "fact database".
14	Remove from the "facts base" the first fact that satisfies the conditions: the car brand "Opel", the car model "Kadet". After deletion, print the data from the "fact database".
15	Remove from the "facts base" all the fact that satisfies the conditions: the car brand "Opel", the car model "Vectra". After deletion, print the data from the "fact database".
16	Remove from the "facts base" the first fact that satisfies the conditions: the car brand "Peugeot", the car model "206". After deletion, print the data from the "fact database".
17	Remove from the "facts base" all the fact that satisfies the conditions: the car brand "Mitsubishi", body color "Silver". After deletion, print the data from the "fact database".
18	Remove from the "facts base" the first fact that satisfies the conditions: the brand of the car "Hyundai", the color of the body "Red". After deletion, print the data from the "fact database".
19	Remove from the "facts base" all the fact that satisfies the conditions: the car brand "Chevrolet", the color of the body "Brown". After deletion, print the data from the "fact database".
20	Delete from the "facts base" the first fact that satisfies the conditions: the car brand "Peugeot", year of manufacture 1999. After deletion, remove the data from the "facts base".
21	Remove from the "facts base" all the fact that satisfies the conditions: the brand of the car "Mazda", the color of the body "Red". After deletion, print the data from the "fact database".

22	Remove from the "facts base" the first fact that satisfies the conditions: the brand of the car "Mitsubishi", the color of the body "Silver". After deletion, print the data from the "fact database".
23	Remove from the "facts base" all the fact that satisfies the conditions of the car brand "Hyundai", the color of the body "Red". After deletion, print the data from the "fact database".
24	Remove from the "facts base" the first fact that satisfies the conditions: the car brand "Skoda", model "Octava". After deletion, print the data from the "fact database".
25	Remove from the "facts base" all cars belonging to "Elena". After deletion, print the data from the "fact database".
26	Add the following fact to the "fact base": car ("Alfa Romeo", "156", 2005, "Green", "Vitaly"). The new fact must take precedence over the existing facts. After entering the fact, save the "fact base" in a file named [Artist's name].
27	Add the following fact to the "fact base": car ("Audi", "80", 1995, "Red", "Vladimir"). The new fact must be placed after the existing facts. After entering the fact, save the "fact base" in a file named [Artist's name].
28	Add the following fact to the "fact base": car ("BMW", "X5", 2011, "Black", "Dmitry"). The new fact must take precedence over the existing facts. After entering the fact, save the "fact base" in a file named [Artist's name].
29	Add the following fact to the "fact base": emperor ("Chrysler", "Saratoga", 2000, "Gold", "Yuriy"). The new fact must be placed after the existing facts. After entering the fact, save the "fact base" in a file named [Artist's name].
30	Add the following fact to the "fact base": car ("Citroen", "Berlingo", 2006, "White", "Ignat"). The new fact must take precedence over the existing facts. After entering the fact, save the "fact base" in a file named [Artist's name].
31	Add the following fact to the "fact base": emperor ("Chery", "Amulet", 2009, "Blue", "Nikita"). The new fact must be placed after the existing facts. After entering the fact, save the "fact base" in a file named [Artist's name].
32	Add the following fact to the "fact base": car ("Daewoo", "Espero", 1997, "Brown", "Elena"). The new fact must take precedence over the existing facts. After entering the fact,

	save the "fact base" in a file named [Artist's name].
33	Add the following fact to the "fact base": emperor ("Daihatsu", "Rocky", 2006, "White", "Hope"). The new fact must be placed after the existing facts. After entering the fact, save the "fact base" in a file named [Artist's name].
34	Add the following fact to the "fact base": emperor ("Dodge", "Spirit", 1994, "Black", "Gennady"). The new fact must take precedence over the existing facts. After entering the fact, save the "fact base" in a file named [Artist's name].
35	Add the following fact to the "fact base": car ("Fiat", "Punto", 2009, "Gray", "Olga"). The new fact must be placed after the existing facts. After entering the fact, save the "fact base" in a file named [Artist's name].
36	Add the following fact to the "fact base": car ("Ford", "Fiesta", 2007, "Silver", "Alexander"). The new fact must take precedence over the existing facts. After entering the fact, save the "fact base" in a file named [Artist's name].
37	Add the following fact to the "fact base": car ("Honda", "Civic", 2006, "Black", "Pavel"). The new fact must be placed after the existing facts. After entering the fact, save the "fact base" in a file named [Artist's name].
38	Add the following fact to the "fact base": car ("Infiniti", "QX 56", 2008, "White", "Oksana"). The new fact must take precedence over the existing facts. After entering the fact, save the "fact base" in a file named [Artist's name].
39	Add the following fact to the "fact base": car ("Jeep", "Grand Cherokee", 2005, "Black", "Denis"). The new fact must be placed after the existing facts. After entering the fact, save the "fact base" in a file named [Artist's name].
40	Add the following fact to the "fact base": car ("KIA", "Rio", 2009, "Blue", "Hope"). The new fact must take precedence over the existing facts. After entering the fact, save the "fact base" in a file named [Artist's name].
41	Add the following fact to the "fact base": car ("Lancia", "Theme", 2007, "White", "Oleg"). The new fact must be placed after the existing facts. After entering the fact, save the "fact base" in a file named [Artist's name].
42	Add the following fact to the "fact base": car ("Land Rover", "Freelander", 2010, "Black", "Ivan"). The new fact must take precedence over the existing facts. After entering the fact, save the "fact base" in a file named [Artist's

	name].
43	Add the following fact to the "fact base": car ("Lexus", "LX-470", 2008, "Black", "Natalia"). The new fact must be placed after the existing facts. After entering the fact, save the "fact base" in a file named [Artist's name].
44	Add the following fact to the "fact base": car ("Mercedes-Benz", "B-class", 2005, "Silver", "Yuriy"). The new fact must take precedence over the existing facts. After entering the fact, save the "fact base" in a file named [Artist's name].
45	Add the following fact to the "fact base": emperor ("Pontiac", "Firebird", 2005, "Gold", "Paul"). The new fact must be placed after the existing facts. After entering the fact, save the "fact base" in a file named [Artist's name].
46	Add the following fact to the "fact base": car ("Porsche", "911", 2004, "Gray", "Sergey"). The new fact must take precedence over the existing facts. After entering the fact, save the "fact base" in a file named [Artist's name].
47	Add the following fact to the "fact base": car ("Renault", "Clio", 2009, "White", "Victor"). The new fact must be placed after the existing facts. After entering the fact, save the "fact base" in a file named [Artist's name].
48	Add the following fact to the "fact base": tsar ("Saab", "9000", 2007, "Black", "Nikita"). The new fact must take precedence over the existing facts. After entering the fact, save the "fact base" in a file named [Artist's name].
49	Add the following fact to the "fact base": car ("Seat", "Toledo", 2011, "Silver", "Georgy"). The new fact must be placed after the existing facts. After entering the fact, save the "fact base" in a file named [Artist's name].
50	Add the following fact to the "fact base": car ("Toyota", "Avensis", 2011, "White", "Vitaly"). The new fact must take precedence over the existing facts. After entering the fact, save the "fact base" in a file named [Artist's name].

6.16. List of control questions on the task № 8

1. What predicates are used to add and remove facts?
2. Where are the predicates of the internal database of facts stored?
3. What does the facts section consist of?
4. Name the restrictions imposed on predicates announced in the facts section.

5. What is the name of the internal facts base?
6. What are the keywords determ and single used for?
7. What are the predicates assert, asserta and assertz used for, and what is the difference between them?
8. What are the predicate consult used for?
9. What are the restrictions on the files that consult can work with?
10. What are the predicates retract and retractall used for, and what is the difference between them?
11. What is the predicate save used for?
12. List the keywords that define the properties of facts.

Recommended books

1. Walter Enders. Applied Econometric Time Series. Wiley, 2-е издание, 2004,
460 P.
2. Philip H. Franses & Dick van Dijk. Nonlinear Time Series Models in Empirical Finance. Cambridge University Press, 2000, 296 P.
3. Daniel Pẽna, George C. Tiao & Ruey S. Tsay (редакторы). A Course in Time
Series Analysis. Wiley, 2001, 496 P.
4. Ruey S. Tsay. Analysis of Financial Time Series. Wiley, 2005, 640 P.
5. Enders, W. & P. Chung (1995). Instructor’s Manual to Accompany Applied Econometric Time-Series. John Wiley & Sons: New York.: Chung, P., W. Enders, L. Shao & J. Yuan (2004).
6. Tolvi, J. (2003). Analysis of Financial Time Series by R. S. Tsay. The Statistician 52, 128–129 P.
7. Eric J. Larson The Myth of Artificial Intelligence : Why Computers Can't Think the Way We Do. - Cambridge, Mass., United States. – 2021. – 320 P.
8. Charles Petzold. - The Annotated Turing : A Guided Tour Through Alan Turing's Historic Paper on Computability and the Turing Machine. – Auflage. – 2008. – 384 P.
9. Стьюарт Рассел, Питер Норвиг. – искусственный интеллект. Современный подход. Том 1. Решение проблем. Знания и рассуждения. – Диалектика. – 2021. – 706 С.
10. Ian Barkin, Jochen Wirtz , Pascal Bornet : Intelligent Automation : Learn how to harness Artificial Intelligence to boost business & make our world more human. - Independently Published. – 2020. – 432 P.

11. Marco Iansiti , Karim R. Lakhani : Competing in the Age of AI : Strategy and Leadership When Algorithms and Networks Run the World. - Harvard Business Review Pres. – 2020. – 288 P.
12. Nick Bostrom: Superintelligence : Paths, Dangers, Strategies. –- Oxford, United Kingdom. – 2016.- 432 P.
13. Tom Taulli : The Robotic Process Automation Handbook : A Guide to Implementing RPA Systems. - Berkley, United States. – 2020. – 344 P.
14. Thomas M. Siebel : Digital Transformation : Survive and Thrive in an Era of Mass Extinction; .- Rosetta Books. - 2019.- 256 P.
15. Kate Crawford : Atlas of AI.- Yale University Press.- 2021. – 336 P.
16. Panos Louridas ; Algorithms - MIT Press Ltd . – 2020. – 424 P.

Additions

Additions A. Determining relationships based on facts

Prolog (PROgramming in LOgic) is the most popular logic programming language.

The language is based on the theory of calculation of first-order predicates and methods of proving theorems.

The main method of calculation is resolution (proof procedure). Procedurality of the Prologue is expressed in determining the order of actions to achieve the goal (how to get).

Supports declarative (descriptive) programming style – programs are designed in terms of accurately defining the problem situation without writing a program in the form of a sequence of instructions for performing the algorithm.

Prolog is a programming language for symbolic, non-numerical calculations. It is designed to solve problems related to objects and relationships between objects.

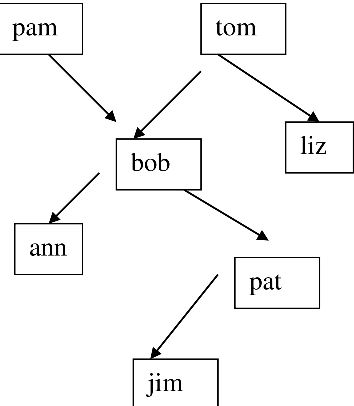


Fig.A.1. Family tree

The figure shows an example – Family Tree. The fact that Tom is one of Bob's parents can be written in Prolog as follows: parents (tom, bob).

In this example, the relationship name is parents, tom, bob - relationship parameters (names are written in lower case). The entire relationship tree, which is depicted in the diagram in Prolog, is defined by the following program:

parents (pam, bob).
parents (tom, bob).
parents (tom, liz).
parents (bob, ann).
parents (bob, pat).
parents (pat, jim).

Each of the 6 sentences that make up the program reveals one fact about the parent relationship. For example, the fact parents (tom, bob) is a specific instance of a relationship, such an instance of a relationship is called a relationship. In general, the ratio is the set of all its instances.

1. We can ask if Bob is one of Pat's parents. ? – parents(bob,pat). Маємо відповідь:

Yes

2. ? – parents(liz,pat).

No

3. ? – parents(tom,ben).

No

Question: Who is the father of liz?

? – parents(X,liz).

X= tom

Who are Bob's children?

? – parents(bob, X).

There should be 2 answers: first there will be an answer – X-ann then to get the second answer enter a semicolon and Prolog will get the next answer X-pat. If after that to demand the additional decision the system will answer No.

You can ask a general question:

? – parents(X,Y).

After that Prolog will be looking for all the pairs one by one

X – pam
Y – bob
X – tom
Y – bob
…….

To stop outputting solutions, press Enter instead;

This program can be asked a more difficult question: who are the parents of Jim's grandparents. The ratio of grandparents is not defined, so this query is divided into 2 stages:

1 - who is one of Jim's parents, for example, W.

2 - who is one of the parents of U.

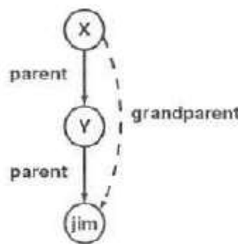


Fig.A.2 The grandparent relationship, expressed as a composition of two parent relationships

In this scheme, the grandparents relationship is expressed as a composition of two parents relationships.

This complex query is written in Prolog as follows:

? - parents (Y, jim), parents (X, Y).

Reply:

X - bob

Y - path

This complex query can be read as follows: find the following X and Y that meet the following 2 requirements: parents (Y, jim) and parents (X, Y).

The result will be the same if you change the query as follows:

? - parents (X, Y), parents (Y, jim).

Similarly, you can ask who are Tom's grandchildren:

? - parents (tom, X),? - parents (X, Y).

The answer will be as follows:

X - bob

Y - ann

X - bob

Y - path

Question: Do Ann I Rat have common parents?

The task is divided into 2 stages:

1 - who is one of the parents Ann (X)

2 - whether X is one of the parents of Pat

The corresponding question in Prolog is as follows:

? - parents (X, ann), parents (X, pat).

Reply:

X – bob

This example illustrates the following concepts:

- in Prolog, you can define relationships such as parents by specifying objects that satisfy that relationship;
- the user can ask the system about the relationships that are defined in the program;
- prolog consists of sentences, each of which ends with a period;
- relationship parameters can be defined as objects or constants (such as tom and ann), as well as general objects (such as X and Y). Objects of the first type are called atoms, objects of the second type are called variables);
- questions to the system consist of one or more goals. The sequence of goals is such as parent (X, ann), patent! X, pat) means the conjunction of goals: X is one of Anne's parents and X is one of Pat's parents;
- the word “goal” is used for such questions because the Prolog system perceives questions as goals that need to be achieved;
- the answer to the question can be positive or negative, depending on whether the goal has been achieved;
- if there are several answers to the question, Prolog know all of them.

Exercises

1.1. Provided the parent relation is defined as described in this section (see fig.1), indicate what answer the Prolog system will give to the following questions?

a)? – parent (jim, X).
b)? – parent (X, jim).
c)? – parent (pam, X), parent (X, p a t).
r)? – parent (pam, X), parent (X, Y), parentt Y, jim).

1.2. State the following relationship questions in Prolog parent.

a) Who is Pat's parent?
b) Does Liz have a baby?
c) Who is Pat's grandfather or grandmother?

Valuation of business on the basis of rules

Adoptionally, the butt stock is equipped with the same methods. Introduced information about the person and the life to become, as they will take part in the parents. Dodamo onset facts in the program (here female - woman, feminine - female, male - male. Masculine - male, sex - to become).

female (pam).
male (tom).
female (liz).
female (ann).
male (jim).

Here is the introduction of male and female. Quality of business є unary (one-off). Binarny is used for the type of parents, but there are links between

pairs of objects. Unary vidnoshennya vikoristovuyutsya for the nakedness of the simple authorities of the objects, as the stench can be mother or not mother.

Information that is announced in 2 unary reports can be presented in binary reports:

sex (pam, feminine).

sex (tom, masculine).

sex (bob, masculine).

As a next addition, we will add to the program the relationship offspring (offspring) – (son or daughter), the opposite of the relationship parents. This relationship can be defined as follows. offspring (liz, tom) we have parents (tom, liz).

But the relation offspring can be defined differently, using the relation parents: Y is the son or daughter of X, if X is the father of Y.

If parents (tom (X), liz (Y)) true -> offspring (liz (Y), tom (X)).

offspring (Y, X): - parents (X, Y). - it reads as follows: For all X and Y if X is the father of Y, then Y is the son or daughter of X.

Such sentences are called rules.

The difference between facts and rules: facts are logical conclusions that always matter truth. Rules take on the value of truth when certain conditions are met. It is believed that the rules consist of 2 parts: the condition - the right part of the rule, the conclusion - the left part of the rule.

In the expression offspring (Y, X): - parents (X, Y) the left part is called the head of the sentence, and the right part - the body of the sentence.

Concretization of variables

Let's ask the program the question: is Liz the daughter of Tom?

To apply a rule to Liz and Tom objects, you must replace Liz with Y, and X must replace Tom with this rule (this rule can be applied to any X and Y object).

offspring (liz, torn): - parent (tom, l and z).

Let's introduce some more information about family relationships.

For all X and Y, X is the mother of Y, if X is one of the parents of Y and X is a woman.

This statement can be written as follows:

Mother (X, Y): - parent (X, Y), female (X).

A comma between the conditions indicates the conjunction of these conditions, which means that both conditions must be true.

Relationships such as parent, offspring and mother can be illustrated by the following diagrams. Graph nodes refer to the parameters of the relationship. The lines between the nodes correspond to binary (double ratio). Unary relations are denoted in the form

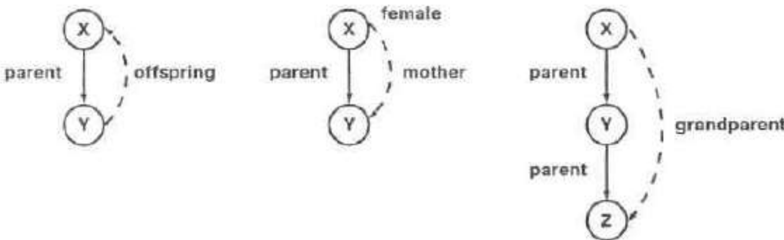


Fig. A.3. Relationship scheme granparent, offspring и mother

marks on the corresponding objects with the name of the relationship. In the form of arcs denote relations that are determined on the basis of other relations.

According to this scheme, the grandparent relation can be written as follows.

grandparent (X, Z): - parent (X, Y), parent (Y, Z).

Consider the scheme of relations sister

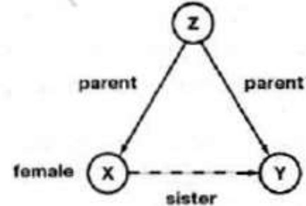


Fig. A.4. Determining the relationship sister

For any X and Y, X is the sister of Y if:

1) X and Y have a common parent and
2) X — woman.

The graph presented in Prolog can be represented as follows:

sister (X, Y): - parent (Z, X), parentf Z, Y), female (X).

Here X and have a common relative Z. This can be paraphrased as follows:

If Z1 is a relative of X and Z2 is a relative of B then Z1 = Z2.

Now you can ask the system questions:

? – sister (arm, pat).
The answer is yes.

So the system can be asked: s i s t e r (X, pat).
The prologue will give an unexpected answer:
X – ann
X – pat.

So pat is her sister. After all, the rules on sisters did not state that they should be the same. To correct this, it is necessary to specify that X and Y must be different. In the Prolog system there is a relation different and it is satisfied if X and Y are not equal. An improved rule for the sister relationship might look like this:

Sister (X, Y): -parent (Z, X), parent (Z, Y), female (X), different (X, Y).

Conclusions*:*

1. Programs in the Prologue can be supplemented by introducing new sentences
2. Sentences in the Prologue belong to 3 types: facts, rules, questions
3. With the help of facts you can enter into the program information that is always true
4. With the help of rules you can enter information into the program that is true depending on the specified condition
5. By asking questions, the user can find out what information is true
6. Sentences Prologue consists of head and body. A body is a comma-separated list of goals. Comos are considered signs of conjunction.
7. Facts are sentences that have a head and an empty body. Questions have only the body. The rules have a head and a non-empty body.
8. In the process of calculation, changes can be replaced by other objects. In this case, the variable becomes specified.

Additions B. Perceptroni

The perceptron is one of three methods for solving tasks for creating images of objects on a model of a hypothetical mechanism of a robot and a human brain. The structure of the model is postulated for a long time (the postulate is firmness, pretense, which, when prompted by a scientific theory, accept without proof, such as an axiom). With this approach, the level of biological knowledge, or hypotheses about biological mechanisms and a change of mind, on which the models of these mechanisms are based. It is necessary to note that the perceptron at the dawn of his own judgment was seen only as a heuristic model of the mechanism of the brain. Since then, the stench has

become the main scheme in the inducement of piecewise-linear models to develop images.

Heuristics is a science, how I create creativity, methods, how to become victorious in the face of the new and in the new. Heuristic methods (another name is Heuristics) allow you to customize the process of resolving tasks. Significant interest in the connection with the possibility of broadcasting a number of tasks (developing objects, bringing theorems, etc.), in which people cannot date an accurate algorithm for scanning with additional technical data. By the method of Heuristics motivating models for the process of resolving certain new tasks.

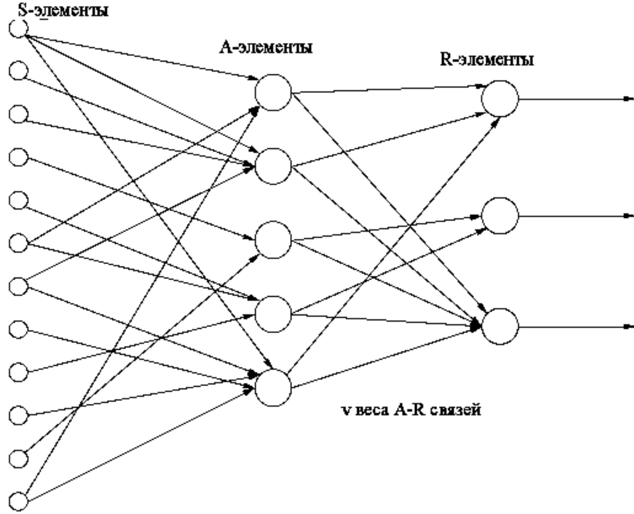


Fig. B.1. Perceptroni

In the most simple view, the perceptron (fig. B.1) is stored in a number of sensitive (sensory) elements (S-elements), which receive input signals. S-elements in the same rank are tied with a combination of associative elements (A-elements), types of which appear from zero only, if there is a large number of S-elements, for one A-elements are connected with reacting elements (R-elements) links, performance efficiency (v) of some changes and changes in the process of creation. It is important to combine R-elements to become the reaction of the system, in order to make the object fit, so that it is developed to a singing image. If only two images are recognized, then only one R-element will appear in the perceptron, which in two reactions is positive and negative. If there are more than two images, then a specific R-element is installed for the skin image, and the output of such an element is a linear combination of A-elements:

$$R_j = \Theta_j + \sum_{i=1}^{n} v_{ij} x_i \ , \qquad \text{(B.1)}$$

where $\boldsymbol{R}_j$ – reaction *j* element R; x_i reaction *I* element A; v_{ij} – link from the i-th A-element to the j-th R-element; Θj - threshold of the j-th R-element.

Similarly, write down the ryvnyannya of the i-th A-element:

$$x_i = \Theta_i + \sum_{k=1}^{S} y_k \ . \qquad \text{(B.2)}$$

Here the signal yk can be continuous, but most often it takes only two values: 0 or 1. Signals from S-elements are fed to the inputs of A-elements with constant weights equal to one, but each A-element is associated only with a group by chance selected S-elements.

Suppose you need to teach the perceptron to distinguish between two images V1 and V2. We assume that in the perceptron there are two R-elements, one of which is intended for the image V1, and the other - the image V2. The perceptron will be trained correctly if the output R1 exceeds R2, when the recognized object belongs to the image V1, and vice versa. The division of objects into two images can be performed using only one R-element. Then the objects of the image V1 must correspond to the positive reaction of the R-element, and the objects of the image V2 - the negative.

The perceptron is trained by presenting a learning sequence of images of objects belonging to images V1 and V2. In the process of learning the weights of vi A-elements change. In particular, if an error-correction reinforcement system is used, the correctness of the decision made by the perceptron is taken into account first of all. If the solution is correct, then the weights of the bonds of all triggered A-elements leading to the R-element that gave the correct solution increase, and the weights of the failed A-elements remain unchanged. You can leave the weight of the triggered A-elements unchanged, but reduce the weight of the failed ones. After the learning process, the perceptron itself, without a teacher, begins to classify new objects.

If the perceptron operates according to the described scheme and it allows only connections from binary S-elements to A-elements and from A-elements to a single R-element, then such a perceptron is called an elementary a-perceptron. Usually the classification C (W) is set by the teacher. The perceptron must develop in the process of learning the classification conceived by the teacher.

With regard to perceptrons, several sound theorems have been formulated and proved, two of which determine the basic properties of the perceptron below.

Theorem 1. The class of elementary a-perceptrons for which there is a solution for any conceived classification is not empty.

This theorem states that for any classification of an educational sequence it is possible to choose such a set (from an infinite set) of A-elements in which the intended division of an educational sequence by means of a linear decisive rule will be carried out.

Theorem 2. If for some classification C (W) a solution exists, then in the process of learning a-perceptron with error correction, which begins with an arbitrary initial state, this solution will be reached within a finite period of time.

The meaning of this theorem is that if in relation to the intended classification we can find a set of A-elements in which there is a solution, then within this set it will be achieved in a finite period of time.

Usually discuss the properties of an infinite perceptron, ie a perceptron with an infinite number of A-elements with different connections with S-elements (a complete set of A-elements). In such perceptrons the solution always exists, and once it exists, it is achievable in a-perceptrons with error correction.

The structure of the induced perceptron is generalized. A very interesting area of research is multilayer perceptrons and perceptrons with cross-links, but the theory of these systems has not yet been developed.

Neural networks

Neural network model with back propagation. A biological neuron is modeled as a device that has several inputs (dendrites) and one output (axon). Each input is associated with a certain weighting factor (w), which characterizes the bandwidth of the channel and evaluates the degree of influence of the signal from this input on the signal at the output. Depending on the specific implementation, the signals processed by the neuron may be analog or digital (1 or 0). In the body of the neuron there is a weighted summation of the input excitations, and then this value is an argument of the activation function of the neuron, one of the possible variants of which is presented in Fig. B.3.

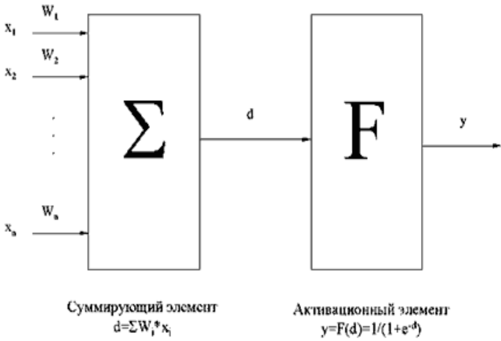


Fig. B.3. Artificial neuron

Being connected in a certain way, neurons form a neural network. The work of the network is divided into learning and adaptation. Training means the process of adaptation of the network to the proposed reference samples by modifying (according to certain algorithms) the weights of connections between neurons. It should be noted that this process is the result of the algorithm of the network, rather than pre-embedded human knowledge, as is often the case in artificial intelligence systems.

Among the various structures of neural networks (NM), one of the most well-known is the multilayer structure, in which each neuron of an arbitrary layer is associated with all axons of neurons of the previous layer or, for the first layer, with all NM inputs. Such NM are called fully connected. When there is only one layer in the network, the algorithm of its training with the teacher is quite obvious, because the correct initial states of the neurons of the single layer are known, and the adjustment of synaptic connections goes in a direction that minimizes the error at the network output. According to this principle, an algorithm for learning a single-layer perceptron is built, for example. In turn, in multilayer networks, the optimal initial values of neurons of all layers, except the latter, are usually unknown, and two or more layered perceptron can no longer be taught, guided only by the magnitude of errors at the outputs of the NM. One solution to this problem is to develop sets of output signals that correspond to the inputs for each layer of NM, which is usually a very time consuming operation and not always feasible. The second option is a dynamic adjustment of the weights of synapses (connections), during which the weakest connections are usually selected and changed by a small amount in one direction or another, and only those changes are preserved that caused a decrease in the output error. the entire network. Obviously, this method of "ticking", despite its simplicity, requires cumbersome routine calculations. And finally, the third, more acceptable option - the propagation of error signals from the outputs of the

NM to its inputs, in the direction opposite to the direct propagation of signals in normal operation. This NM learning algorithm is called the backpropagation procedure. It will be considered further.

According to the least squares method, the minimized objective function of the NM error is the value:

$$E(w) = \frac{1}{2}\sum_{j,p}(y_{j,p}^{(N)} - d_{j,p})^2 \ , \tag{B.3}$$

where $y_{j,p}^{(N)}$ – the real initial state of the neuron j of the output layer N of the neural network when applied to its inputs of the p-th image; djp is the ideal (desired) initial state of this neuron.

The summation is performed for all neurons of the source layer and for all images processed by the network. Minimization is carried out by the method of gradient descent, which means adjusting the weights as follows:

$$\Delta w_{ij}^{(n)} = -\eta \cdot \frac{\partial E}{\partial w_{ij}} \ , \tag{B.4}$$

where w_{ij} – the weight of the synaptic connection connecting the i-th neuron of the layer n –1 with the j-th neuron of the layer n, η is the coefficient of learning speed, $0 < \eta < 1$.

$$\frac{\partial E}{\partial w_{ij}} = \frac{\partial E}{\partial y_j} \cdot \frac{dy_j}{\partial s_j} \cdot \frac{\partial s_j}{\partial w_{ij}} \ , \tag{B.5}$$

where y_j still means the output of neuron *j*, and s_j means the weighted sum of its input signals, ie the argument of the activation function. Since the factor d_{yj} / d_sj is a derivative of this function by its argument, it follows that the derivative of the activation function must be defined along the entire abscissa. In this regard, the single jump function and other activation functions with inhomogeneities are not suitable for the considered NM. They use such smooth functions as a hyperbolic tangent or a classic sigmoid with an exponent. In the case of a hyperbolic tangent:

$$\frac{dy}{ds} = 1 - s^2. \tag{B.6}$$

And the third factor $\frac{\partial s_j}{\partial w_{ij}}$ is obviously equal to the output of the neuron of the previous layer $y_i^{(n-1)}$.

As for the first factor in (A.7), it is easily decomposed as follows:

$$\frac{\partial E}{\partial y_j} = \sum_k \frac{\partial E}{\partial y_k} \cdot \frac{dy_k}{ds_k} \cdot \frac{\partial s_k}{\partial y_j} = \sum_k \frac{\partial E}{\partial y_k} \cdot \frac{dy_k}{ds_k} \cdot w_{jk}^{(n+1)} \qquad \text{(B.7)}$$

The summation of k is performed among the neurons of the layer n + 1. Introducing a new variable

$$\delta_j^{(n)} = \frac{\partial E}{\partial y_j} \cdot \frac{dy_j}{ds_j}, \qquad \text{(B.8)}$$

we obtain a recursive formula for calculating quantities $d_j^{(n)}$ ayer n of the values older layer *n+1*.

$$\delta_j^{(n)} = \left[\sum_k \delta_k^{(n+1)} \cdot w_{jk}^{(n+1)} \right] \cdot \frac{dy_j}{ds_j}, \qquad \text{(B.9)}$$

For the source layer

$$\delta_i^{(N)} = (y_i^{(N)} - d_i) \cdot \frac{dy_j}{ds_j}, \qquad \text{(B.10)}$$

Now we can write (B.4) in the open form:

$$\Delta w_{ij}^{(n)} = -\eta \cdot \delta_j^{(n)} \cdot y_i^{(n-1)}. \qquad \text{(B.11)}$$

Sometimes, to give the weight correction process some inertia, which smooths out the sharp jumps when moving on the surface of the objective function, (B.11) is supplemented by the value of the change in weight on the previous iteration

$$\Delta w_{ij}^{(n)}(t) = -\eta \cdot (\mu \cdot \Delta w_{ij}^{(n)}(t-1) + (1-\mu) \cdot \delta_j^{(n)} \cdot y_i^{(n-1)}). \qquad \text{(B.12)}$$

where μ – inertia coefficient, t is the number of the current iteration.

Thus, the complete algorithm for learning NM using the backpropagation procedure is built as follows:

1. Apply to the network inputs one of the possible images and in the normal operation of the NM, when the signals propagate from the inputs to the outputs, calculate the values of the latter. You have to remember that

$$s_j^{(n)} = \sum_{i=0}^{M} y_i^{(n-1)} \cdot w_{ij}^{(n)}, \tag{B.13}$$

where M – the number of neurons in the layer n – 1 з урахуванням нейрона з постійним вихідним станом n+1, that sets the offset; $y_i^{(n-1)} = x_{ij}^{(n)}$ – i-th input of the neuron j layer n.

$$y_j^{(n)} = f(s_j^{(n)}), \tag{B.14}$$

where $f()$ – sigmoid.

$$y_q^{(0)} = I_q, \tag{B.15}$$

where I_q – q-th component of the vector of the input image.

2. To calculate $d^{(N)}$ for the source layer by the formula (B.10). Calculate by the formula (B.11) or (B12) weight changes $\Delta w^{(N)}$ layer N.

3. Розрахувати за формулами (B.9) and (B.11) (or (B.9) and (B.12)) in accordance $d^{(n)}$ and $\Delta w^{(n)}$ for all other layers, $n = N-1, \ldots 1$.

4. Adjust all weights in NM

$$w_{ij}^{(n)}(t) = w_{ij}^{(n)}(t-1) + \Delta w_{ij}^{(n)}(t). \tag{B.16}$$

If the network error is significant, go to step 1. Otherwise, end.

The networks in step 1 are alternately randomly presented with all the training images, so that the network, figuratively speaking, does not forget some as you remember others. The algorithm is illustrated in Fig. B.4. From expression (B.11) it follows that when the initial value goes to zero, the effectiveness of training is significantly reduced. With binary input vectors, on average, half of the weights will not be adjusted, so the range of possible values of neuron outputs [0,1] is desirable to shift within [-0.5, +0.5], which is achieved

by simple modifications of logistics functions. For example, a sigmoid with an exponent will turn into:

$$f(x) = -0.5 + \frac{1}{1 + e^{-a \cdot x}} \tag{B.17}$$

The considered NM has several "bottlenecks". First, in the process of learning there may be a situation where large positive or negative values of weights will shift the operating point on the sigmoids of many neurons in the saturation region. Small values of the derivative of the logistics function will lead in accordance with (B9) and (B10) to stop learning, which paralyzes NM. Second, the application of the gradient descent method does not guarantee that a global rather than a local minimum of the objective function will be found. This problem is associated with another, namely - with the choice of the speed of learning. Proof of the convergence of learning in the process of reverse propagation is based on derivatives, ie weight gain and, accordingly, the speed of learning must be infinitesimal, but in this case, learning will be unacceptably slow. On the other hand, too much weight correction can lead to permanent instability in the learning process. Therefore, the quality of η is usually chosen as a number less than 1, but not very small, for example, 0.1, and it may gradually decrease during training. In addition, to avoid accidental hits in local minima, sometimes, after the values of the weights are stabilized, η briefly greatly increased to begin the gradient descent from a new point. If repeating this procedure several times brings the algorithm to the same state of the NM, we can more or less confidently say that a global maximum has been found, and not some other. There is another method of excluding local minima, and at the same time paralysis of NM, which consists in the use of stochastic NM, but it is better to talk about them separately.

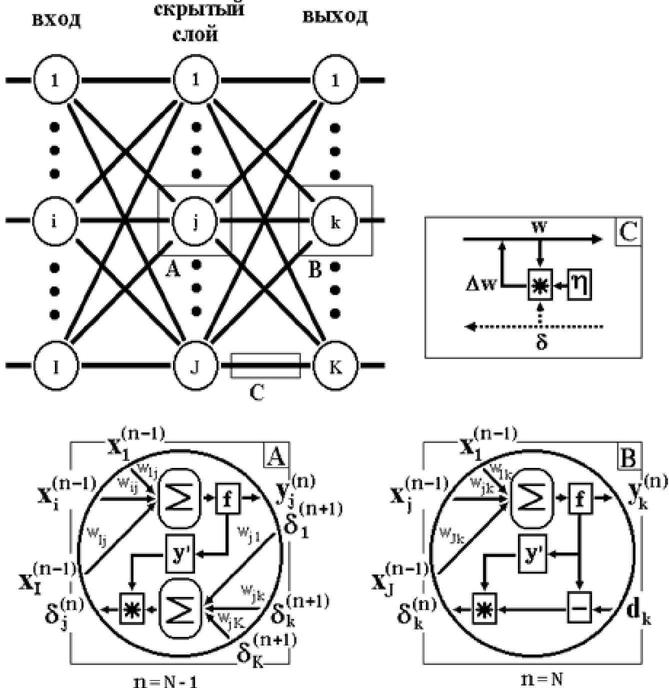


Fig. B.4. The work of the algorithm

Neural networks: learning without a teacher

The main feature that makes learning without a teacher attractive is his "independence". The learning process, as in the case of learning with a teacher, is to adjust the weights of the synapses. Some algorithms change the structure of the network, ie the number of neurons and their relationships, but such transformations are better called a broader term - self-organization, and in this chapter they will not be considered. Obviously, the adjustment of synapses can be carried out only on the basis of information available in the neuron, ie its state and already existing weights. Based on this reasoning and, more importantly, by analogy with the known principles of self-organization of nerve cells, Hebb's learning algorithms are constructed.

Hebb's signaling method is to change the weights according to the following

rule: $$w_{ij}(t) = w_{ij}(t-1) + \alpha \cdot y_i^{(n-1)} \cdot y_j^{(n)}, \qquad (B.18)$$

where $y_i^{(n-1)}$ is the initial value of the neuron i layer (n - 1), $y_j^{(n)}$ - the initial value of the neuron j layer n; $w_{ij}(t)$ and $w_{ij}(t-1)$ - the weight of the synapse connecting these neurons at iterations t and t - 1, respectively; α - coefficient of learning speed. Hereinafter, n means an arbitrary layer of the network. When learning this method, the connections between excited neurons are strengthened.

There is also a differential method of teaching Hebb.

$$w_{ij}(t) = w_{ij}(t-1) + \alpha \cdot \left[y_i^{(n-1)}(t) - y_i^{(n-1)}(t-1)\right] \cdot \left[y_j^{(n)}(t) - y_j^{(n)}(t-1)\right], \tag{B.19}$$

where $y_i^{(n-1)}(t)$ and $y_i^{(n-1)}(t-1)$ – the initial value of the neuron i layer n - 1, respectively, on iterations of t and t - 1; $y_j^{(n)}(t)$ and $y_j^{(n)}(t-1)$ – the same for neuron j layer n. As can be seen from formula (17), the most intensively studied synapses connecting neurons, the outputs of which have changed most dynamically in the direction of increase.

The complete learning algorithm using the above formulas will look like this:

1. At the initialization stage, all weights are assigned small random values.

2. The input image is fed to the inputs of the network, and the excitation signals are propagated in all layers according to the principles of classical directforward (feedforward) networks, ie for each neuron is calculated weighted sum of its inputs, which then applies the activation (transmitting) function of the neuron. it turns out its initial value $y_i^{(n)}$ $i = 0,\ldots,M_i - 1$,, where M_i is the number of neurons in layer i; $n = 0,\ldots,N-1$, and N is the number of layers in the network. 3. Based on the obtained initial values of neurons according to formula (B.18) or (B.19) there is a change in weights.

4. Cycle from step 2 until the original network values are stabilized with the specified accuracy. The use of this new method of determining the completion of learning, different from that used for the reverse propagation network, due to the fact that the adjustable synapse values are not actually limited.

In the second step of the cycle, all the images from the input set are alternately presented. It should be noted that the type of responses to each class of input images is not known in advance and will be an arbitrary combination of states of the neurons of the output layer due to the random distribution of weights at the stage of initialization. However, the network is able to generalize similar images, assigning them to one class. Testing a trained network allows you to determine the topology of classes in the source layer. To bring the

feedback of the trained network to a convenient form, you can supplement the network with one layer, which, for example, according to the algorithm of learning a single-layer perceptron must be forced to reflect the initial reactions of the network in the necessary images.

Another algorithm for learning without a teacher - Kohonen's algorithm - involves adjusting synapses based on their values from the previous iteration.

$$w_{ij}(t) = w_{ij}(t-1) + \alpha \cdot \left[y_i^{(n-1)}(t) - w_{ij}(t-1) \right] \tag{B.20}$$

From the above formula it is seen that the training is reduced to minimizing the difference between the input signals of the neuron coming from the outputs of the neurons of the previous layer $y_i^{(n)}$, and the weights of its synapses

The complete learning algorithm has approximately the same structure as in Hebb's methods, but in step 3 a neuron is selected from the whole layer, the synapse values of which are as similar as possible to the input image, and the weight adjustment according to formula (B.20) is performed only for it. This so-called accreditation may be accompanied by inhibition of all other neurons in the layer and the introduction of the selected neuron in saturation. The choice of such a neuron can be made, for example, by calculating the scalar product of the vector of weights with a vector of input values. The maximum product is given by the winning neuron. Another option is to calculate the distance between these vectors in p-dimensional space, where p is the size of the vectors.

$$D_j = \sqrt{\sum_{i=0}^{p-1} (y_i^{(n-1)} - w_{ij})^2}\,, \tag{B.21}$$

where j is the index of the neuron in the layer n, i is the summation index of the neurons of the layer (n - 1), w_{ij} is the weight of the synapse connecting the neurons; the outputs of the neurons of the layer (n - 1) are the input values for the layer n. It is not necessary to take the root in formula (B.21), as only the relative estimation of different D_j is important.

In this case, "wins" the neuron with the shortest distance. Sometimes, neurons that very often receive accreditation are forcibly excluded from consideration in order to "equalize the rights" of all neurons in the layer. The simplest version of this algorithm is to inhibit the newly won neuron. When using Kohonen algorithm training, there is a practice of normalization of input images, as well as – at the stage of initialization – and normalization of the initial values of weights.

$$x_i = x_i \,/\, \sqrt{\sum_{j=0}^{n-1} x_j^2}\,, \tag{B.22}$$

where x_i is the i-th component of the vector of the input image or the vector of weights, and n is its dimension.

This reduces the duration of the learning process.

Initialization of weights by random values can lead to different classes, which correspond to densely distributed input images, merge or, conversely, split into additional subclasses in the case of close images of the same class. To avoid this situation, the convex combination method is used. Its essence is that the input normalized images can be transformed:

$$x_i = a(t) \cdot x_i + (1 - a(t)) \cdot \frac{1}{\sqrt{n}}, \tag{B.23}$$

where x_i is the i-th component of the input image, n is the total number of its components, a (t) is the coefficient that changes in the learning process from zero to one, resulting in first the network inputs are almost identical images, and over time they are increasingly converging on the weekend.

The weights are set at the initialization step equal to the value:

$$w_o = \frac{1}{\sqrt{n}}, \tag{B.24}$$

where n is the dimension of the weight vector for the neurons of the initialized layer.

Based on the above method, neural networks of a special type are built - the so-called self-organizing features – self-organizing feature maps (this established translation from English, in my opinion, is not very successful, because it is not about changing the network structure, but only about adjusting synapses). For them, after selecting from the layer n of the neuron j with the minimum distance D_j (A.21) teaches by formula (A.20) not only this neuron, but also its neighbors located in the vicinity of R. The value of R in the first iterations is very large, so all neurons learn , but over time it decreases to zero. Thus, the closer the end of training, the more accurately determined the group of neurons corresponding to each class of images.

Hopfield and Hemming neural networks

Among the various configurations of artificial neural networks (NM), there are those in the classification of which according to the principle of learning, strictly speaking, neither learning with a teacher nor learning without a teacher is suitable. In such networks, the synapse weights are calculated only once before the network begins to operate on the basis of information about the data being processed, and all network training is reduced to this calculation. On

the one hand, the presentation of a priori information can be seen as a teacher's help, but on the other - the network actually just remembers the samples before the input of real data, and can not change their behavior, so talk about the reverse link connection with the "world" (teacher) is not necessary. Among the networks with a similar logic, the most well-known are the Hopfield network and the Hemming network, which are commonly used to organize associative memory. Then we will talk about them.

The block diagram of the Hopfield network is shown in Fig. B.5. It consists of a single layer of neurons, the number of which is both the number of inputs and outputs of the network. Each neuron is connected by synapses to all other neurons, and has one input synapse through which the signal is input. Output signals are usually formed on axons.

The problem solved by this network as associative memory is usually formulated as follows. There is a set of binary signals (images, sound digits, other data describing objects or process characteristics) that are considered exemplary. The network must be able to select ("remember" from partial information) the corresponding sample (if any) from any arbitrary non-ideal signal submitted to its input or "give a conclusion" that the input data do not correspond to any of the samples. In the General case, any signal can be described by the vector $X = \{x_i: i = 0, ..., n - 1\}$, n is the number of neurons in the network and the dimension of the input and output vectors. Each element x_i is either +1 or -1. Denote the vector describing the k-th sample by X_k, and its components, respectively: x_i^k, $k = 0, ..., m - 1$, m is the number of samples. When the network recognizes (or "remembers") any sample based on the data presented to it, its outputs will contain it, ie $Y = X^k$, where Y is the vector of the original values of the network: $Y = \{y_i: i = 0, ..., n - 1\}$. Otherwise, the original vector will not match any of the samples. If, for example, the signals are some images, then, displaying graphically the data from the network output, you can see a picture that completely coincides with one of the models (in case of success) or "free improvisation" of the network (in case of failure).

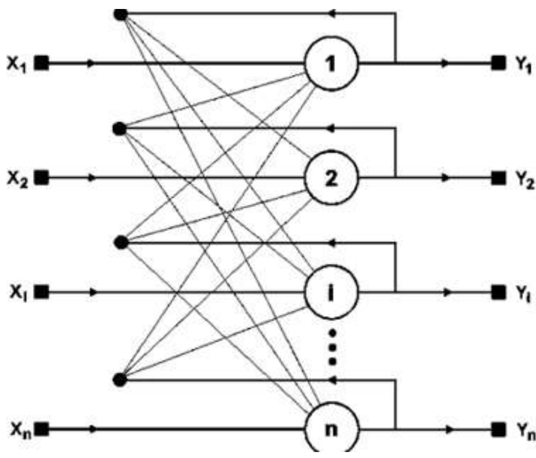


Fig. B.5. Block diagram of the Hopfield network

At the stage of network initialization, the synapse weights are set as

follows: $$w_{ij} = \begin{cases} \sum_{k=0}^{m-1} x_i^k x_j^k, & i \neq j \\ 0, & i = j \end{cases} \quad \text{(B.25)}$$

where i and j – indices, respectively, of presynaptic and postsynaptic neurons; x_i^k and x_j^k – i-th та j-th vector elements of the k-th sample.

The algorithm of network operation is as follows (p – iteration number):

An unknown signal is applied to the network inputs. In fact, its introduction is carried out by direct setting of axon values:

$$y_i(0) = x_i,\ i = 0,...,n-1, \quad \text{(B.26)}$$

therefore, the designation on the circuit of the network of input synapses in the explicit form is purely conditional. Zero in parentheses to the right of yi means zero iteration in the network cycle.

A new state of neurons is calculated

$$s_j(p+1) = \sum_{i=0}^{n-1} w_{ij} y_i(p),\ \ j = 0,...,n-1, \quad \text{(B.27)}$$

and new axon values

$$y_j(p+1) = f[s_j(p+1)] \tag{B.28}$$

where f is the activation function in the form of a jump, shown in Fig. B.6. a.

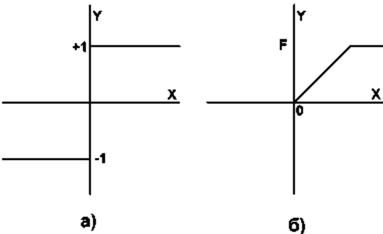


Fig. B.6. Activation function

Check whether the original values of the axons have changed since the last iteration. If so – go to step 2, otherwise (if the outputs have stabilized) – the end. The source vector is the sample that best combines with the input data

As mentioned above, sometimes the network cannot recognize and outputs a non-existent image. This is due to a limited network capability issue. For the Hopfield network, the number of images to be stored m should not exceed approximately 0.15n. In addition, if the two images A and B are very similar, they may cause cross-associations in the network, ie the presentation of vector A to the inputs of the network will lead to the appearance of vector B at its outputs and vice versa.

When it is not necessary for the network to explicitly issue a sample, that is, it is sufficient, say, to obtain a sample number, the associative memory is successfully implemented by the Hamming network. This network is characterized, compared to the Hopfield network, lower memory costs and computational volume, which becomes apparent from its structure (Fig. B.7).

The network consists of two layers. The first and second layers have m neurons, where m is the number of samples. The neurons of the first layer have n synapses connected to the inputs of the network (forming a fictitious zero layer). The neurons of the second layer are interconnected by inhibitory (negative feedback) synaptic connections. A single positive feedback synapse for each neuron is connected to its own axons.

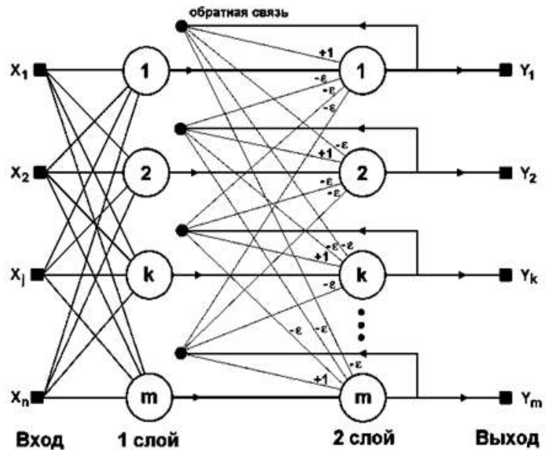


Fig. Fig. B.7. Block diagram of the Hamming network.

The idea of the network is to find the distance of Hamming from the tested image to all samples. The Hamming distance is the number of bits that differ in two binary vectors. The network must select a sample with a minimum Hamming distance to an unknown input signal, which will activate only one network output corresponding to the specified sample.

At the stage of initialization, the weights of the first layer and the threshold of the activation function are assigned the following values:

$$w_{ij} = \frac{x_i^k}{2}, \; i = 0,\ldots,n-1, \; k = 0,\ldots,m-1, \quad \text{(B.29)}$$

$$T_k = n/2, \; k = 0,\ldots,m-1 \quad \text{(B.30)}$$

where is the i-th element of the k-th sample

The weighting factors of the inhibitory synapses in the second layer are taken to be equal to some value 0 < *e* <1 / *m*. The synapse of a neuron associated with its own axon has a weight of +1.

The algorithm of operation of the Hamming network is as follows:

1. An unknown vector X = {x_i: *i* = 0, ..., *n* - 1} is fed to the network inputs, based on which the states of neurons of the first layer are calculated (the superscript in parentheses indicates the layer number

$$y_j^{(1)} = s_j^{(1)} = \sum_{i=0}^{n-1} w_{ij} x_i + T_j, \; j = 0,\ldots,m-1. \quad \text{(B.31)}$$

After that, the obtained values initialize the values of the axons of the second layer:

$$y_j^{(2)} = y_j^{(1)},\ \ j = 0,...,m-1. \tag{B.32}$$

Обчислити нові стани нейронів другого шару:

$$s_j^{(2)}(p+1) = y_j(p) - \varepsilon \sum_{k=0}^{m-1} y_k^{(2)}(p),\ k \neq j,\ \ j = 0,...,m-1 \tag{B.33}$$

and the value of their axons:

$$y_j^{(2)}(p+1) = f[s_j^{(2)}(p+1)],\ \ j = 0,...,m-1 \tag{B.34}$$

The activation function f has the form of a threshold (Fig. B.6.b), and the value of F must be large enough that any possible values of the argument do not lead to saturation.

4. Check whether the outputs of the neurons of the second layer have changed during the last iteration. If so – go to step 2. Otherwise – the end.

The evaluation of the algorithm shows that the role of the first layer is quite conditional: using once in step 1 the values of its weights, the network no longer accesses it, so the first layer can be excluded from the network (replaced by a matrix of weights).

Addition C. Method of group consideration of arguments

Least squares method

Before beginning the consideration of MGCA, it would be useful to mention or learn for the first time the method of least squares – the most common method of adjusting linearly dependent parameters.

For example, consider the MNC for three arguments. Let the function $T = T(U, V, W)$ be given by the table, ie from experience the numbers U_i, V_i, W_i, T_i ($i = 1, ... n$) are known. We will look for the relationship between this data in the form:

$$T(U,V,W) = aU + bV + cW \tag{C.1}$$

where a, b, c are unknown parameters.

Select the values of these parameters so that was the smallest sum of the squares of the deviations of the known data Ti and theoretical $T_i = aU_i + bV_i + cW_i$, ie the sum:

$$\sigma = \sum_{i=1}^{n} (T_i - aU_i - bV_i - cW_i)^2 \rightarrow \min \qquad (C.2)$$

The value of σ is a function of three variables a, b, c. A necessary and sufficient condition for the existence of a minimum of this function is the equality of zero partial derivatives of the function σ for all variables, is:

$$\frac{\partial \sigma}{\partial a} = 0, \ \frac{\partial \sigma}{\partial b} = 0, \ \frac{\partial \sigma}{\partial c} = 0 \qquad (C.3)$$

Since:

$$\begin{aligned} \frac{\partial \sigma}{\partial a} &= -2\sum_{i=1}^{n} (T_i - aU_i - bV_i - cW_i)U_i \\ \frac{\partial \sigma}{\partial b} &= -2\sum_{i=1}^{n} (T_i - aU_i - bV_i - cW_i)V_i \\ \frac{\partial \sigma}{\partial a} &= -2\sum_{i=1}^{n} (T_i - aU_i - bV_i - cW_i)W_i \end{aligned} \qquad (C.4)$$

then the system for finding a, b, c will look like:

$$\begin{aligned} a\sum_{i=1}^{n} U_i^2 + b\sum_{i=1}^{n} U_i V_i + c\sum_{i=1}^{n} U_i W_i &= \sum_{i=1}^{n} T_i U_i \\ a\sum_{i=1}^{n} U_i V_i + b\sum_{i=1}^{n} V_i^2 + c\sum_{i=1}^{n} V_i W_i &= \sum_{i=1}^{n} T_i V_i \\ a\sum_{i=1}^{n} U_i W_i + b\sum_{i=1}^{n} W_i V_i + c\sum_{i=1}^{n} W_i^2 &= \sum_{i=1}^{n} T_i W_i \end{aligned} \qquad (C.5)$$

this system is solved by any standard method of solving systems of linear equations (Gauss, Jordan, Seidel, Cramer).

Consider some practical examples of finding approximate functions:

1. $y = ax^2 + bx + g$.

The problem of selection of coefficients a, b, g is reduced to the solution of the general problem at:

$T = y$, $U = x^2$, $V = x$, $W = 1$, $a = a$, $b = b$, $g = c$.

2. $f(x, y) = a \sin(x) + b \cos(y) + g / x$.

The problem of selection of coefficients a, b, g is reduced to the solution of the general problem at:

$T = f$, $U = \sin(x)$, $V = \cos(y)$, $W = 1 / x$, $a = a$, $b = b$, $g = c$.

If we extend MNCs to the case with m parameters,

$$\sigma = \sum_{i=1}^{n}\left(T_i - \sum_{\nu=1}^{m} u_{i\nu}c_\nu\right)^2 \to \min \tag{C.6}$$

then by reasoning similar to the above, we obtain the following system of linear equations:

$$\begin{cases} c_1\bar{u}_1\bar{u}_1 + c_2\bar{u}_1\bar{u}_2 + \ldots + c_m\bar{u}_1\bar{u}_m = \bar{T}\bar{u}_1 \\ c_1\bar{u}_2\bar{u}_1 + c_2\bar{u}_2\bar{u}_2 + \ldots + c_m\bar{u}_2\bar{u}_m = \bar{T}\bar{u}_2 \\ \qquad\qquad\ldots \\ c_1\bar{u}_m\bar{u}_1 + c_2\bar{u}_m\bar{u}_2 + \ldots + c_m\bar{u}_m\bar{u}_m = \bar{T}\bar{u}_m \end{cases} \tag{C.7}$$

where, $\bar{T} = \{T_i\}_{i=1}^{m}$, $\bar{u}_\nu = \{u_{i\nu}\}_{i=1}^{n}$.

Borrowing algorithms for processing information in nature is one of the main ideas of cybernetics. The "selection hypothesis" states that the algorithm of mass selection of plants or animals is the optimal algorithm for processing information in complex problems. At mass selection some quantity of seeds is sown. As a result of pollination, complex hereditary combinations are formed. Breeders choose some of the plants in which the properties they are interested in are most pronounced (heuristic criterion). The seeds of these plants are collected and re-sown to form new, even more complex combinations. After a few generations, the selection stops and its result is optimal. If you excessively continue the selection, then there will be "incuht" - degeneration of plants. There is an optimal number of generations and the optimal number of seeds selected in each of them.

MGCA algorithms reproduce the scheme of mass selection shown in Fig. 1. They have generators of combinations that are complicated from row to row, and threshold self-selection of the best of them. The so-called "complete" description of the object

$$j = f(x_1, x_2, x_3, \ldots, x_m),$$

where f is some elementary function, such as a power polynomial, is replaced by several series of "partial" descriptions:

1st row of selection: $y_1 = f(x_1x_2)$, $y_2 = f(x_1x_3)$, ..., $y_s = f(x_{m-1}x_m)$,

2 st row of selection: $z_1 = f(y_1y_2)$, $z_2 = f(y_1y_3)$, ..., $z_p = f(y_{s-1}y_s)$,

where $s = c^2$, $p = c_s^2$, etc.

The input arguments and intermediate variables are combined in pairs, and the complexity of the combinations on each row of information processing increases (as in mass selection) until a single model of optimal complexity is obtained. Each partial description is a function of only two arguments. Therefore, its coefficients are easy to determine according to the training sequence with a small number of interpolation nodes. Excluding intermediate variables (if possible), you can get an "analogue" of the full description. Mathematics does not prohibit both of these operations. For example, ten interpolation nodes can be obtained by estimating the coefficients of polynomials of the hundredth degree, etc.

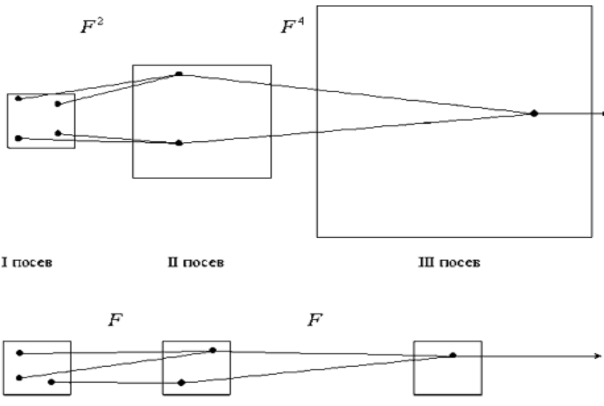


Fig. C.1. General scheme of construction of MGCA algorithms (selection of the blackest tulip) for the experimental field, which has the possibility of expansion (equivalent to full search), and at a constant field size (equivalent of selection while maintaining the freedom of choice of solutions F = const)

Only a number of the most regular variables are skipped from row to row of selection. The degree of regularity is estimated by the magnitude of the standard error (mean for all variables selected in each generation, or for one of the most accurate change) on a single test sequence. Sometimes a correlation coefficient is used as an indicator of regularity.

Rows of selection are increased as long as the regularity increases. As soon as the minimum error is reached, the selection should be stopped to avoid "incuht". It is practically recommended to stop the selection even a little before reaching the full minimum, as soon as the error begins to fall too slowly. This leads to simpler and more reliable equations.

Наукове видання

SCHERBAN V.Y., DEMKIVSKIY E.O., DEMKIVSKA T.I.,
SHRAMCHENKO B.L., REZANOVA V.G.

Methods and systems of artificial intelligence

Підписано до друку 22.06.2022 р.
Формат 60x84/16. Папір офсетний.
Ум. друк. арк. 10,81
Наклад 200 прим.

Видано ТОВ "Фастбінд Україна"
Свідоцтво про внесення суб'єкта видавничої справи до
державного реєстру видавців, виготівників
і розповсюджувачів видавничої продукції
ДК 6324 від 31.07.2018 р.