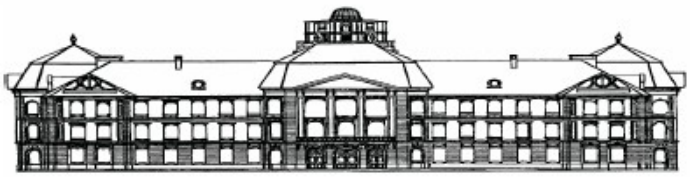


Eszterházy Károly Collage

Institute of Mathematics and Informatics

Artificial Intelligence and its Teaching

Lecture Notes by

Dr. Gergely Kovásznai

and

Dr. Gábor Kusper

NEW SZÉCHENYI PLAN

National Development Agency
www.ujszechelyiterv.gov.hu
06 40 638 638

IMPROVING HUNGARY

The programme is financed by the European Union and the European Regional Development Fund

Table of Contents

1 Introduction

Surely everyone have thought about what artificial intelligence is? In most cases, the answer from a mathematically educated colleague comes in an instant: It depends on what the definition is? If artificial intelligence is when the computer beats us in chess, then we are very close to attain artificial intelligence. If the definition is to drive a land rover through a desert from point A to point B, then we are again on the right track to execute artificial intelligence. However, if our expectation is that the computer should understand what we say, then we are far away from it.

This lecture note uses artificial intelligence in the first sense. We will bring out such „clever" algorithms, that can be used to solve the so called graph searching problems. The problems that can be rewritten into a graph search – such as chess – can be solved by the computer.

Alas, the computer will not become clever in the ordinary meaning of the word if we implement these algorithms, at best, it will be able to systematically examine a graph in search of a solution. So our computer remains as thick as two short planks, but we exploit the no more than two good qualities that a computer has, which are:

- The computer can do algebraic operations (addition, subtraction, etc.) very fast.
- It does these correctly.

So we exploit the fact that such problems that are to difficult for a human to see through – like the solution of the Rubik Cube – are represented in graphs, which are relatively small compared to the capabilities of a computer, so quickly and correctly applying the steps dictated by the graph search algorithms will result in a fast-solved Cube and due to the correctness, we can be sure that the solution is right.

At the same time, we can easily find a problem that's graph representation is so huge, that even the fastest computers are unable to quickly find a solution in the enormous graph. This is where the main point of our note comes in: the human creativity required by the artificial intelligence. To represent a problem in a way that it's graph would keep small. This is the task that should be started developing in high school. This requires the expansion of the following skills:

- Model creation by the abstraction of the reality
- System approach

It would be worthwhile to add algorithmic thinking to the list above, which is required to think over and execute the algorithms published in this note. We will talk about this in a subsequent chapter.

The solution of a problem is the following in the case of applying artificial intelligence:

- We model the real problem.
- We solve the modelled problem.
- With the help of the solution found in the model, we solve the real problem.

All steps are helped by different branches of science. At the first step, the help comes from the sciences that describe reality: physics, chemistry, etc. The second step uses an abstract idea system, where mathematics and logic helps to work on the abstract objects. At last, the engineering sciences, informatics helps to plant the model's solution into reality.

This is all nice, but why can't we solve the existing problem in reality at once? Why do we need modelling? The answer is simple. Searching can be quite difficult and expensive in reality. If the well-know 8 Queens Problem should be played with 1-ton iron queens, we would also need a massive hoisting crane, and the searching would take a few days and a few hundreds of diesel oil till

we find a solution. It is easier and cheaper to search for a solution in an abstract space. That is why we need modelling.

What guarantees that the solution found in the abstract space will work in reality? So, what guarantees that a house built this way will not collapse? This is a difficult question. For the answer, let's see the different steps in detail.

Modelling the existing problem:

- We magnify the parts of the problem that are important for the solution and neglect the ones that are not.
- We have to count and measure the important parts.
- We need to identify the possible „operators" that can be used to change reality.

Modelling the existing problem is called state space representation in artificial intelligence. We have a separate chapter on this topic. We are dealing with this question in connection with the „will-the-house-collapse" - issue. Unfortunately, a house can be ruined at this point, because if we neglect an important issue, like the depth of the wall, the house may collapse. How does this problem, finding the important parts in a text, appear in secondary school? Fortunately, it's usually a maths exercise, which rarely contains unnecessary informations. The writer of the exercise usually takes it the other way round and we need to find some additional information which is hidden in the text.

It is also important to know that measuring reality is always disturbed with errors. With the tools of Numeric mathematics, the addition of the the initial errors can be given, so the solution's error content can also be given.

The third step, the identification of the „operators", is the most important in the artificial intelligence's aspect. The operator is a thing, that changes the part of reality that is important for us, namely, it takes from one well-describable state into another. Regarding artificial intelligence, it's an operator, when we move in chess, but it may not if we chop down a tree unless the number of the trees is not an important detail in the solution of the problem.

We will see that our model, also know as state space can be given with

- the initial state,
- the set of end states,
- the possible states and
- the operators (including the pre and post condition of the operators).

We need to go through the following steps to solve the modelled problem:

- Chose a framework that can solve the problem.
- Set the model in the framework.
- The framework solves the problem.

Choosing the framework that is able to solve our model means choosing the algorithm that can solve the modelled problem. This doesn't mean that we have to implement this algorithm. For example, the Prolog interpreter uses backtrack search. We only need to implement, which is the second step, the rules that describe the model in Prolog. Unfortunately, this step is influenced by the fact, that we either took transformational- (that creates a state from another state) or problem reduction (that creates more states from another state) operators in the state space representation. So we can take the definition of the operators to be the next step after choosing the framework. The frameworks may differ from each other in many ways, the possible groupings are:

- Algorithms, that surly find the solution in a limited, non-circle graph.
- Algorithms, that surly find the solution in a limited graph.
- Algorithms, that give an optimal solution according to some point of view.

If we have the adequate framework, our last task is to implement the model in the framework. This is usually means setting the initial state, the end condition and the operators (with pre- and postconditions). We only need to push the button, and the framework will solve the problem if it is able to do it. Now, assume that we have got a solution. First of all, we need to know what do we mean under 'solution'. Solution is a sequence of steps (operator applications), that leads from the initial state into an end state. So, if the initial state is that we have enough material to build a house and the end state is that a house had been built according to the design, then the solution is a sequence of steps about how to build the house.

There is only one question left: will the house collapse? The answer is definitely 'NO', if we haven't done any mistake at the previous step, which was creating the model, and will not do at the next step, which is replanting the abstract model into reality. The warranty for this is the fact that the algorithms introduced in the notes are correct, namely by logical methods it can be proven that if they result in a solution, that is a correct solution inside the model. Of course, we can mess up the implementation of the model (by giving an incorrect end condition, for example), but if we manage to evade this tumbler, we can trust our solution in the same extent as we can trust in logics.

The last step is to solve the real problem with the solution that we found in the model. We have no other task than executing the steps of the model's solution in reality. Here, we can face that a step, that was quite simple in the model (like move the queen to the A1 field) is difficult if not impossible in reality. If we found that the step is impossible, than our model is incorrect. If we don't trust in the solution given by the model, then it worth trying it in small. If we haven't messed up neither of the steps, then the house will stand, which is guaranteed by the correctness of the algorithms and the fact that logic is based on reality!

2 The History of Artificial Intelligence

Studying the intelligence is one of the most ancient scientific discipline. Philosopher have been trying to understand for more than 2000 years what mechanism we use to sense, learn, remember, and think. From the 2000 years old philosophical tradition the theory of reasoning and learning have developed, along with the view that the mind is created by the functioning of some physical system. Among others, these philosophical theories made the formal theory of logic, probability, decision-making, and calculation develop from mathematics.

The scientific analysis of skills in connection with intelligence was turned into real theory and practice with the appearance of computers in the 1950s. Many thought that these „electrical masterminds" have infinite potencies regarding executing intelligence. „Faster than Einstein" - became a typical newspaper article. In the meantime, modelling intelligent thinking and behaving with computers proved much more difficult than many have thought at the beginning.

Figure 1. The early optimism of the 1950s: „The smallest electronic mind of the world" :)

The *Artificial Intelligence* (AI) deals with the ultimate challenge: How can a (either biological or electronic) mind sense, understand, foretell, and manipulate a world that is much larger and more complex than itself? And what if we would like to construct something with such capabilities?

AI is one of the newest field of science. Formally it was created in 1956, when its name was created, although some researches had already been going on for 5 years. AI's history can be broken down into three major periods.

2.1 Early Enthusiasm, Great Expectations (Till the end of the 1960s)

In a way, the early years of AI were full of successes. If we consider the primitive computers and programming tools of that age, and the fact, that even a few years before, computers were only though to be capable of doing arithmetical tasks, it was astonishing to think that the computer is – even if far from it – capable of doing clever things.

In this era, the researchers drew up ambitious plans (world champion chess software, universal translator machine) and the main direction of research was to write up general problem solving methods. Allen Newell and Herbert Simon created a general problem solving application (*General*

Program Solver, GPS), which may have been the first software to imitate the protocols of human-like problem solving.

This was the era when the first theorem provers came into existence. One of these was Herbert Gelernter's Geometry Theorem Prover, which proved theorems based on explicitly represented axioms.

Arthur Samuel wrote an application that played Draughts and whose game power level reached the level of the competitors. Samuel endowed his software with the ability of learning. The application played as a starter level player, but it became a strong opponent after playing a few days with itself, eventually becoming a worthy opponent on strong human race. Samuel managed to confute the fact that a computer is only capable of doing what it was told to do, as his application quickly learnt to play better than Samuel himself.

In 1958, John McCarthy created the *Lisp* programming language, which outgrew into the primary language of AI programming. Lisp is the second oldest programming language still in use today.

2.2 Disillusionment and the knowledge-based systems (till the end of the 1980s)

The general-purpose softwares of the early period of AI were only able to solve simple tasks effectively and failed miserably when they should be used in a wider range or on more difficult tasks. One of the sources of difficulty was that early softwares had very few or no knowledge about the problems they handled, and achieved successes by simple syntactic manipulations. There is a typical story in connection with the early computer translations. After the Sputnik's launch in 1957, the translations of Russian scientific articles were hasted. At the beginning, it was thought that simple syntactic transformations based on the English and Russian grammar and word substitution will be enough to define the precise meaning of a sentence. According to the anecdote, when the famous „The spirit is willing, but the flesh is weak" sentence was re-translated, it gave the following text: „The vodka is strong, but the meat is rotten." This clearly showed the experienced difficulties, and the fact that general knowledge about a topic is necessary to resolve the ambiguities.

The other difficulty was that many problems that were tried to solve by the AI were untreatable. The early AI softwares were trying step sequences based on the basic facts about the problem that should be solved, experimented with different step combinations till they found a solution. The early softwares were usable because the worlds they handled contained only a few objects. In computational complexity theory, before defining NP-completeness (Steven Cook, 1971; Richard Karp, 1972), it was thought that using these softwares for more complex problems is just matter of faster hardware and more memory. This was confuted in theory by the results in connection with NP-completeness. In the early era, AI was unable to beat the „combinatorial boom" – combinatorial explosion and the outcome was the stopping of AI research in many places.

From the end of the 1960s, developing the so-called expert systems were emphasised. These systems had (rule-based) knowledge base about the field they handled, on which an inference engine is executing deductive steps. In this period, serious accomplishments were born in the theory of resolution theorem proving (J. A. Robinson, 1965), mapping out knowledge representation techniques, and on the field of heuristic search and methods for handling uncertainty. The first expert systems were born on the field of medical diagnostics. The MYCIN system, for example, with its 450 rules, reached the effectiveness of human experts, and put up a significantly better show than novice physicians.

At the beginning of the 1970s, Prolog, the logical programming language were born, which was built on the computerized realization of a version of the resolution calculus. Prolog is a remarkably

prevalent tool in developing expert systems (on medical, judiciary, and other scopes), but natural language parsers were implemented in this language, too. Some of the great achievements of this era is linked to the natural language parsers of which many were used as database-interfaces.

2.3 AI BECOMES INDUSTRY (SINCE 1980)

The first successful expert system, called R1, helped to configure computer systems, and by 1986, it made a 40 million dollar yearly saving for the developer company, DEC. In 1988, DEC's AI-group already put on 40 expert systems and was working on even more.

In 1981, the Japanese announced the „fifth generation computer" project – a 10-year plan to build an intelligent computer system that uses the Prolog language as a machine code. Answering the Japanese challenge, the USA and the leading countries of Europe also started long-term projects with similar goals. This period brought the brake-through, when the AI stepped out of the laboratories and the pragmatic usage of AI has begun. On many fields (medical diagnostics, chemistry, geology, industrial process control, robotics, etc.) expert systems were used and these were used through a natural language interface. All in all, by 1988, the yearly income of the AI industry increased to 2 billion dollars.

Besides expert systems, new and long-forgotten technologies have appeared. A big class of these techniques includes statistical AI-methods, whose research got a boost in the early years of the 1980's from the (re)discovery of neural networks. The hidden Markov-models, which are used in speech- and handwriting-recognition, also fall into this category. There had been a mild revolution on the fields of robotics, machine vision, and learning.

Today, AI-technologies are very versatile: they mostly appear in the industry, but they also gain ground in everyday services. They are becoming part of our everyday life.

3 PROBLEM REPRESENTATION

3.1 STATE-SPACE REPRESENTATION

The first question is, how to represent a problem that should be solved on computer. After developing the details of a representation technology, we can create algorithms that work on these kind of representations. In the followings, we will learn the state-space representation, which is a quite universal representation technology. Furthermore, many problem solving algorithms are known in connection with state-space representation, which we will be review deeply in the 3rd chapter.

To represent a problem, we need to find a limited number of features and parameters (colour, weight, size, price, position, etc.) in connection with the problem that we think to be useful during the solving. For example, if these parameters are described with the values $h_1, \ldots, h_n$ (colour: black/white/red; temperature: [-20C°, 40C°]; etc.), then we say that the problem's world is in thestate identified by the vector $(h_1, \ldots, h_n)$. If we denote the set which consists of values adopted by the $i.$ parameter with H_i, then the states of the problem's world are elements of the set $H_1 \times H_2 \times \ldots \times H_n$.

As we've determined the possible states of the problem's word this way, we have to give a special state that specifies the initial values of the parameters in connection with the problem's world. This is called the *initial state*.

During the problem-solving, starting from the initial state, we will change the different states of the problem's world again and again, till we reach an adequate state called the *goal state*. We can even define several goal states.

Now we only need to specify which states can be changed and what states will these changes call forth. The functions that describe the state-changes are called *operator*s. Naturally, an operator can't be applied to each and every state, so the domain of the operators (as functions) is given with the help of the so-called *preconditions*.

Definition 1. A *state-space representation* is a tuple $\langle A, k, C, O \rangle$, where:

(1) A : is the set of states, $A \neq \emptyset$,
(2) $k \in A$: is the initial state,
(3) $C \subseteq A$: is the set of goal states,
(4) O : is the set of the operators, $O \neq \emptyset$.
Every $o \in O$ operator is a function $o : Dom(o) \rightarrow A$, where

$$Dom(o) = \{a \mid a \notin C \ \wedge \ precondition_o(a)\} \subseteq A$$

The set C can be defined in two ways:

- By enumeration (in an explicit way): $C = \{c_1, \ldots, c_m\}$
- By formalizing a goal condition (in an implicit way): $C = \{a \mid goal\ condition(a)\}$

The conditions $precondition_o(a)$ and $goal\ condition(a)$ can be specified as *logical formulas*. Each formulas' parameter is a state a , and the precondition of the operator also has the applicable operator o .

Henceforth, we need to define what we mean the solution of a state-space represented problem – as that is the thing we want to create an algorithm for. The concept of a problem's solution can be described through the following definitions:

Definition 2. Let $\langle A, k, C, O \rangle$ be a state-space representation, and $a, a' \in A$ are two states.

a' is *directly accessible* from a if there is an operator $o \in O$ where $precondition_o(a)$ holds and $o(a) = a'$.

Notation: $a \underset{o}{\rightarrow} a'$.

Definition 3. Let $\langle A, k, C, O \rangle$ be a state-space representation, and $a, a' \in A$ are two states.

a' is *accessible* from a if there is a $a_1, a_2, \ldots, a_n \in A$ state sequence where

- $a_1 = a$,
- $a_n = a'$,
- $\forall\, i \in \{1,2,\ldots,n-1\}$: $a_i \underset{o_i}{\rightarrow} a_{i+1}$ (any $o_i \in O$ operator)

Notation: $a \underset{o_1, o_2, \ldots, o_{n-1}}{\rightarrow} a'$

Definition 4. The problem $\langle A, k, C, O \rangle$ is solvable if $k \underset{o_1, \ldots, o_n}{\rightarrow} c$ for any goal state $c \in C$. In this case, the operator sequence $o_1, \ldots, o_n$ is referred as a *solution* to the problem.

Some problems may have more than one solution. In such cases, it can be interesting to compare the solutions by their costs - and select the less costly (the cheapest) solution. We have the option to assign a cost to the application of an operator to the state a, and denote it as $cost_o(a)$ (assuming that o is applicable to a, that is, $precondition_o(a)$ holds), which is a positive integer.

Definition 5. Let $k \underset{o_1, \ldots, o_n}{\rightarrow} c$ in the case of a $\langle A, k, C, O \rangle$ problem for any $c \in C$. The cost of the solution $o_1, \ldots, o_n$ is:

$$\sum_{i=1}^{n} cost(o_i, a_i) .$$

Namely, the cost of a solution is the cost of *all the operator applications* in the solution. In the case of many problems, the cost of operator applications is uniform, that is $cost(o, a) = 1$ for any operator o and state a. In this case, the cost of the solution is implicitly *the number of applied operators.*

3.2 STATE-SPACE GRAPH

The best tool to demonstrate the state-space representation of a problem is the *state-space graph*.

Definition 6. Let $\langle A, k, C, O \rangle$ be the state-space representation of a problem. The problem's state-space *graph* is the graph[1] $\langle A, E \rangle$, where $(a, a') \in E$ and (a, a') is labelled with o if and only if $a \underset{o}{\rightarrow} a'$.

Therefore, the vertices of the state-space graph are the states themselves, and we draw an edge between two vertices if and only if one vertex (as a state) is directly accessible from another vertex (as a state). We label the edges with the operator that allows the direct accessibility.

It can be easily seen that a *solution* of a problem is nothing else but a path that leads from a vertex k (aka the *initial vertex*) to some vertex $c \in C$ (aka the *goal vertex*). Precisely, the solution is the sequence of labels (operators) of the edges that formulate this path.

In Chapter 4, we will get to know a handful of problem solving algorithms. It can be said, in general, that all of them explore the state-space graph of the given task in different degree and look for the path that represents the solution in the graph.

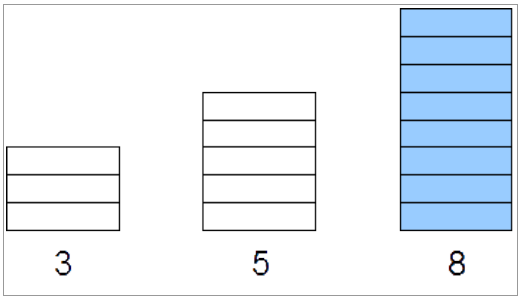
1 As usual: is the set of the graph's vertices, is the set of the graph's edges.

3.3 EXAMPLES

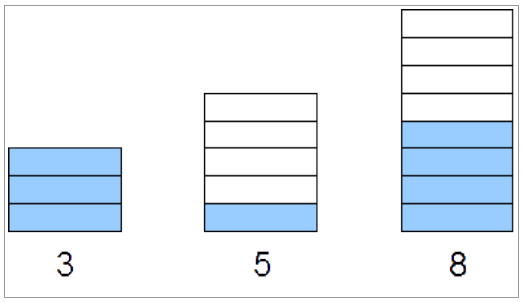
In this chapter, we introduce the possible state-space representations of several noteworthy problems.

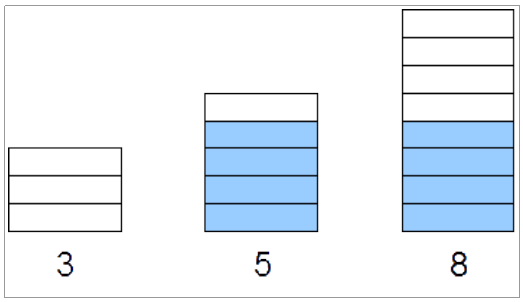
3.3.1 3 JUGS

We have 3 jugs of capacities 3, 5, and 8 litres, respectively. There is no scale on the jugs, so it's only their capacities that we certainly know. Initially, the 8-litre jug is full of water, the other two are empty:



We can pour water from one jug to another, and the goal is to have exactly 4 litres of water in any of the jugs. The amount of water in the other two jugs at the end is irrelevant. Here are two of the possible goal states:





Since there is no scale on the jugs and we don't have any other tools that would help, we can pour water from jug A to jug B in two different ways:

- We pour all the water from jug A to jug B.
- We fill up jug B (and it's possible that some water will remain in jug A).

Give a number to each jug: let the smallest be 1, the middle one 2, and the largest one 3! Generalize the task to jugs of any capacity: introduce a vector with 3 components (as a constant object out of the state-space), in which we store the capacities of the jugs:

$$max = (3,5,8)$$

- **Set of states:** In the states, we store the amount of water in the jugs. Let the state be an tuple, in which the i[th] part tells about the jug denoted by i how many litres of water it is containing.
  So, the set of states is defined as follows:
  $$A = \{(a_1, a_2, a_3) \mid 0 \le a_i \le max_i\}$$
  where every a_i is an integer.
- **Initial state:** at first, jug 3 is full all, the other ones are empty. So, the initial state is:

$$k=(0,0,max_3)$$

- **Set of goal states:** We have several goal states, so we define the set of goal states with help of a goal condition:

$$C=\{(a_1,a_2,a_3)\in A \mid \exists i \;\; a_i=4\}$$

- **Set of operators:** Our operators realize the pouring from one jug (denoted by i) to another one (denoted by j). We can also specify that the source jug (i) and the goal jug (j) can't be the same. Our operators are defined as follows:

$$O=\{pour_{i,j} \mid i,j\in\{1,2,3\} \;\wedge\; i\neq j\}$$

- **Precondition of the operators:** Let's define when an operator $pour_{i,j}$ can be applied to a state (a_1,a_2,a_3) ! It's practical to specify the following conditions:
  - Jug i is not empty.
  - Jug j is not filled.

  So, the precondition of the operator $pour_{i,j}$ to the state (a_1,a_2,a_3) is:

$$a_i\neq 0 \;\wedge\; a_j\neq max_j$$

- **Function of applying:** Define what state (a'_1,a'_2,a'_3) does the operator $pour_{i,j}$ create from the state (a_1,a_2,a_3) ! The question is how many litres of water can we pour from jug i to jug j. Since at most

$$max_j-a_j$$

litres of water can be poured to jug j, we can calculate the exact amount to be poured by calculating

$$min(a_i \;,\; max_j-a_j)$$

Denote this amount with T. Consequently:
$pour_{i,j}(a_1,a_2,a_3)=(a'_1,a'_2,a'_3)$, where

$$a'_m=\begin{cases} a_i-T & \text{, if } m=i \\ a_j+T & \text{, if } m=j \\ a_m & \text{, otherwise} \end{cases} \qquad \text{where } m\in\{1,2,3\}$$

STATE-SPACE GRAPH

The state-space graph of the aforementioned state-space representation can be seen in Figure 2.

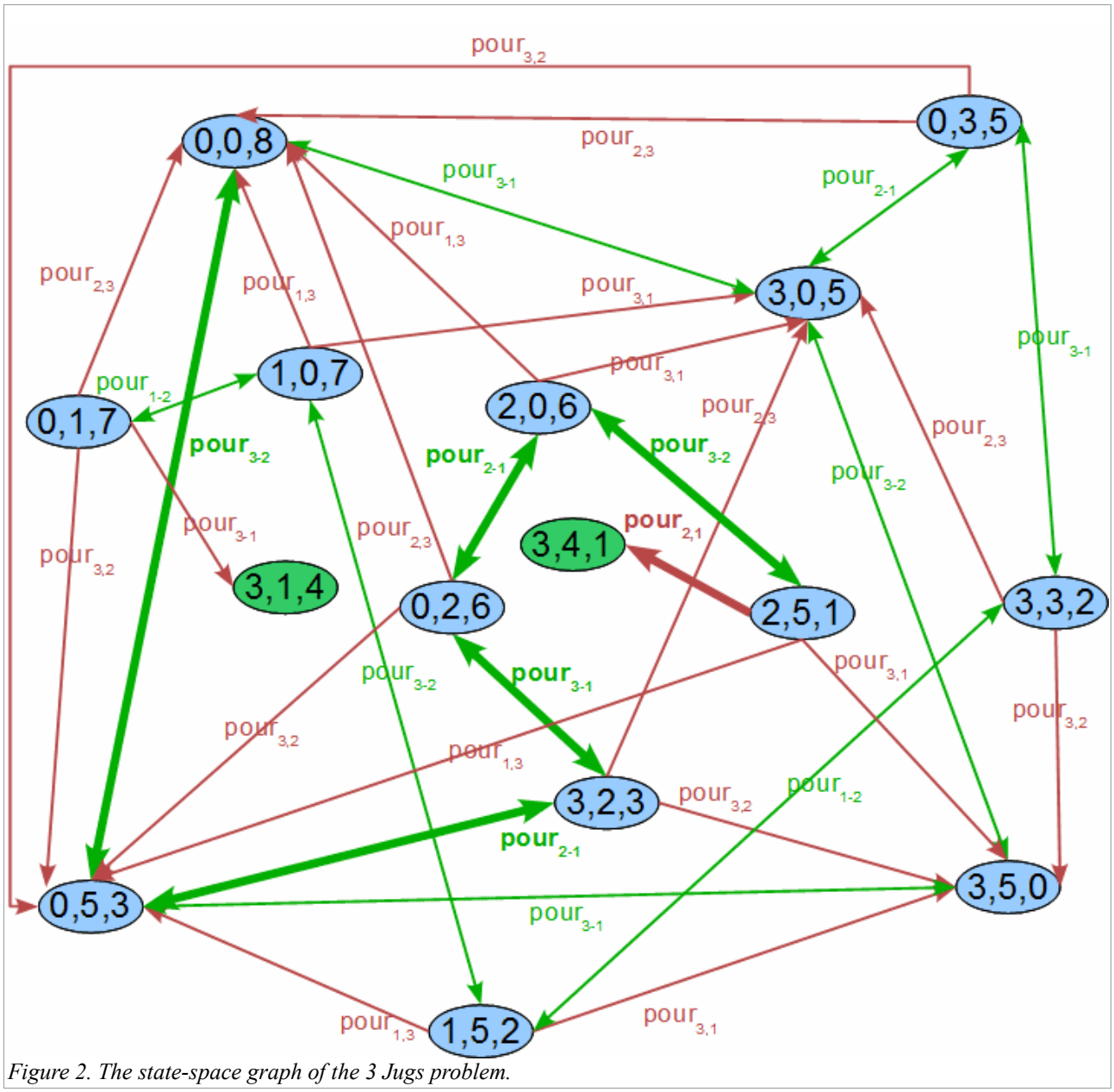


Figure 2. The state-space graph of the 3 Jugs problem.

In the graph, the red lines depict unidirectional edges while the green ones are bidirectional edges. Naturally, bidirectional edges should be represented as two unidirectional edges, but due to lack of space, let us use bidirectional edges. It can be seen that the labels of the bidirectional edges are given in the form of $pour_{i,\,j1-j2}$, which is different from the form of $pour_{i,\,j}$ as it was given in the state-space representation. The reason for this is that one $pour_{i,\,j1-j2}$ label encodes two operators at the same time: the operators $pour_{i,\,j1}$ and $pour_{i,\,j2}$.

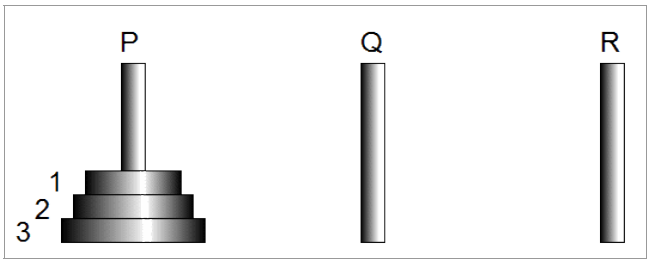
The green vertices represent the *goal states*. The bold edges represent one of the *solutions,* which is the following operator sequence:

$$pour_{3,2}\ ,\ pour_{2,1}\ ,\ pour_{1,3}\ ,\ pour_{2,1}\ ,\ pour_{3,2}\ ,\ pour_{2,1}$$

Notice that the problem has several solutions. It can also be noticed that the state-space graph contains cycles , that makes it even more difficult to find a solution.

3.3.2 Towers of Hanoi

There are 3 discs with different diameters. We can slide these discs onto 3 perpendicular rods. It's important that if there is a disc under another one then it must be bigger in diameter. We denote the rods with „P", „Q", and „R", respectively. The discs are denoted by „1", „2", and „3", respectively, in ascending order of diameter. The initial state of discs can be seen in the figure below:



We can slide a disc onto another rod if the disc

(5) is on the top of its current rod, and
(6) the discs on the goal rod will be in ascending order by size after the replacing.

Our goal is to move all the discs to rod R.

We create the state-space representation of the problem, as follows:

- **Set of states:** In the states, we store the information about the currently positions (i.e., rods) of the discs. So, a state is a vector (a_1, a_2, a_3) where a_i is the position of disc i (i.e., either P, Q, or R). Namely:

$$A = \{(a_1, a_2, a_3) \mid a_i \in \{P, Q, R\}\}$$

- **Initial state:** Initially, all the discs are on rod P, i. e.:

$$k = (P, P, P)$$

- **Set of goal states:** The goal is to move all the three discs to rod R. So, in this problem, we have only one goal state, namely:

$$C = \{(R, R, R)\}$$

- **Set of operators:** Each operator includes two pieces of information:
  - which disc to move
  - to which rod?

  Namely:

$$O = \{move_{which, where} \mid which \in \{1,2,3\} \ , \ where \in \{P, Q, R\}\}$$

- **Precondition of the operators:** Take an operator $move_{which, where}$! Let's examine when we can apply it to a state (a_1, a_2, a_3)! We need to formalize the following two conditions:

(1) The disc $which$ is on the top of the rod a_{which}.

(2) The disc $which$ is getting moved to the top of the rod $where$.

What we need to formalize as a logical formula is that each disc that is smaller than disc $which$ (if such one does exist) is not on either rod a_{which} or rod $where$.

It's worth to extend the aforementioned condition with another one, namely that we don't want to move a disc to the same rod from which we are removing the disc. This condition is not obligatory, but can speed up the search (it will eliminate trivial cycles in the state-space graph).

Thus, the *precondition of the operators is:*

$$a_{which} \neq where \wedge \forall\, i \left(i < which \;\Rightarrow\; a_i \neq a_{which} \wedge a_i \neq where \right)$$

- **Function of applying:** Take any operator $move_{which,where}$! If the precondition of the operator holds to a state (a_1, a_2, a_3), then we can apply it to this state. We have to formalize that how the resulting state (a'_1, a'_2, a'_3) will look like.

  We have to formalize that the disc *which* will be moved to the rod *where*, while the other discs will stay where they currently are. Thus:

  $move_{which,where}(a_1, a_2, a_3) = (a'_1, a'_2, a'_3)$, where

  $$a'_i = \begin{cases} where & \text{, if } i = which \\ a_i & \text{, otherwise} \end{cases} \qquad \text{where } i \in \{1,2,3\}$$

  Important note: we have to define *all* of the components of the new state, not just the one that changes!

STATE-SPACE GRAPH

The state-space graph of the aforementioned state-space representation can be seen in Figure 3.

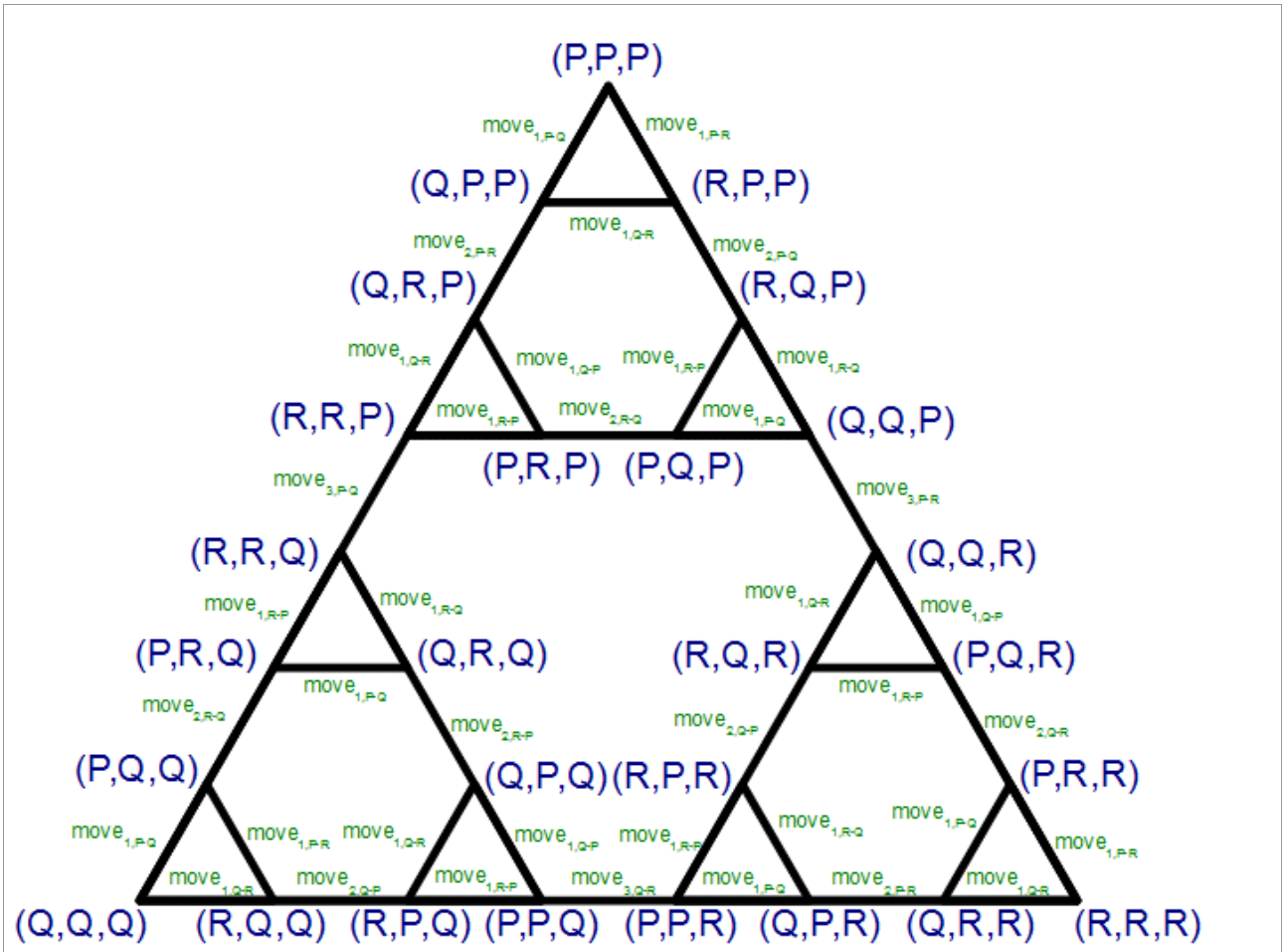


Figure 3. The state-space graph of the Towers of Hanoi problem.

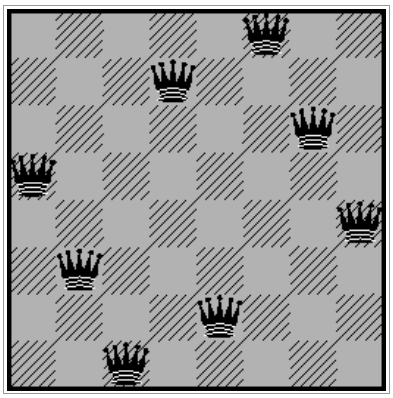
Naturally, all of the edges in the graph are bidirectional, and their labels can be interpreted as in the previous chapter: a label $move_{i,\,j1-j2}$ refers to both the operators $move_{i,\,j1}$ and $move_{i,\,j2}$.

As it can be clearly seen in the figure, the *optimal* (shortest) *solution* of the problem is given by the

rightmost side of the large triangle, namely, the optimal solution consists of 7 steps (operators).

3.3.3 8 QUEENS

Place 8 queens on a chessboard in a way that no two of them attack each other. One possible solution:



Generalize the task to a $N \times N$ $(N \geq 1)$ chessboard, on which we need to place N queens. N is given as a constant out of the state-space.

The basic idea of the state-space representation is the following: since we will place exactly one queen to each row of the board, we can solve the task by placing the queens on the board *row by row*. So, we place one queen to the 1st row, then another one to the 2nd row in a way that they can't attack each other. In this way, in step ith we place a queen to row i while checking that it does not attack any of the previously placed $i-1$ queens.

- **Set of states:** In the states we store the positions of the placed queens within a row! Let's have a N-component vector in a state, in which component i tells us to which column in row i a queen has been previously placed. If we haven't placed a queen in the given row, then the vector should contain 0 there.
  In the state we also store the row in which the next queen will be placed.
  So:
  $$A = \{(a_1, a_2, \ldots, a_N, s) \mid 0 \leq a_i \leq N, 1 \leq s \leq N+1\}$$
  As one of the possible value of s, $N+1$ is a non-existent row index, which is only permitted for testing the terminating conditions.
- **Initial state:** Initially, the board is empty. Thus, the initial state is:
  $$k = (0,0,\ldots,0,1)$$
- **Set of goal states:**
  We have several goal states. If the value of s is a non-existent row index, then we have found a solution. So, the set of goal states is:
  $$C = \{(a_1, \ldots, a_N, N+1) \in A\}$$
- **Set of operators:**
  Our operators will describe the placing of a queen to row s. The operators are expecting only one input data: the column index where we want to place the queen in row s. The set of our operators is:
  $$O = \{place_i \mid 1 \leq i \leq 8\}$$
- **Precondition of the operators:** Formalize the precondition of applying an operator

$place_i$ to a state $(a_1,\ldots,a_8,s)$! It can be applied if the queen we are about to place is

- not in the same row as any queens we have placed before. So, we need to examine if the value of i was in the state before the s[th] component. i. e.,

$$\text{for all } \; m<s: \; a_m \neq i$$

- not attacking any previously placed queens diagonally. Diagonal attacks are the easiest to examine if we take the absolute value of the difference of the row indices of two queens, and then compare it to the absolute value of the difference of the column indices of the two queens. If these values are equal then the two queens are attacking each other. The row index of the queen we are about to place is s, while the column-index is i. So:

$$\text{for all } \; m<s \; : \; |s-m| \neq |i-a_m|$$

Thus, the precondition of the operator $place_i$ to the state $(a_1,\ldots,a_8,s)$ is:

$$\forall \; m\left(m<s \;\Rightarrow\; a_m \neq i \;\wedge\; |s-m| \neq |i-a_m|\right),$$
$$\text{where } \; 1 \leqslant m \leqslant 8$$

Function of applying: Let's specify the state $(a'_1,\ldots,a'_8,s')$ which the operator $place_i$ will create from a state $(a_1,\ldots,a_8,s)$! In the new state, as compared to the original state, we only need to make the following modifications:

- write i to the s[th] component of the state, and
- increment the value of s by one.

Thus:

$place_i(a_1,\ldots,a_8,s)=(a'_1,\ldots,a'_8,s')$, where

$$a'_m = \begin{cases} i & \text{, if } \; m=s \\ a_m & \text{, otherwise} \end{cases} \quad \text{, where } \; m \in \{1,2,3\}$$
$$s' = s+1$$

STATE-SPACE GRAPH

The state-space graph of the aforementioned state-space representation for the case $N=4$ case can be seen in Figure 4. In this case, the problem has 2 solutions.

Notice that every solution of the problem is N long for sure. It is also important to note that there is no cycle in the state-space graph, that is, by carefully choosing the elements of the state-space representation, we have managed to exclude cycles from the graph, which will be quite advantageous when we are searching solutions.

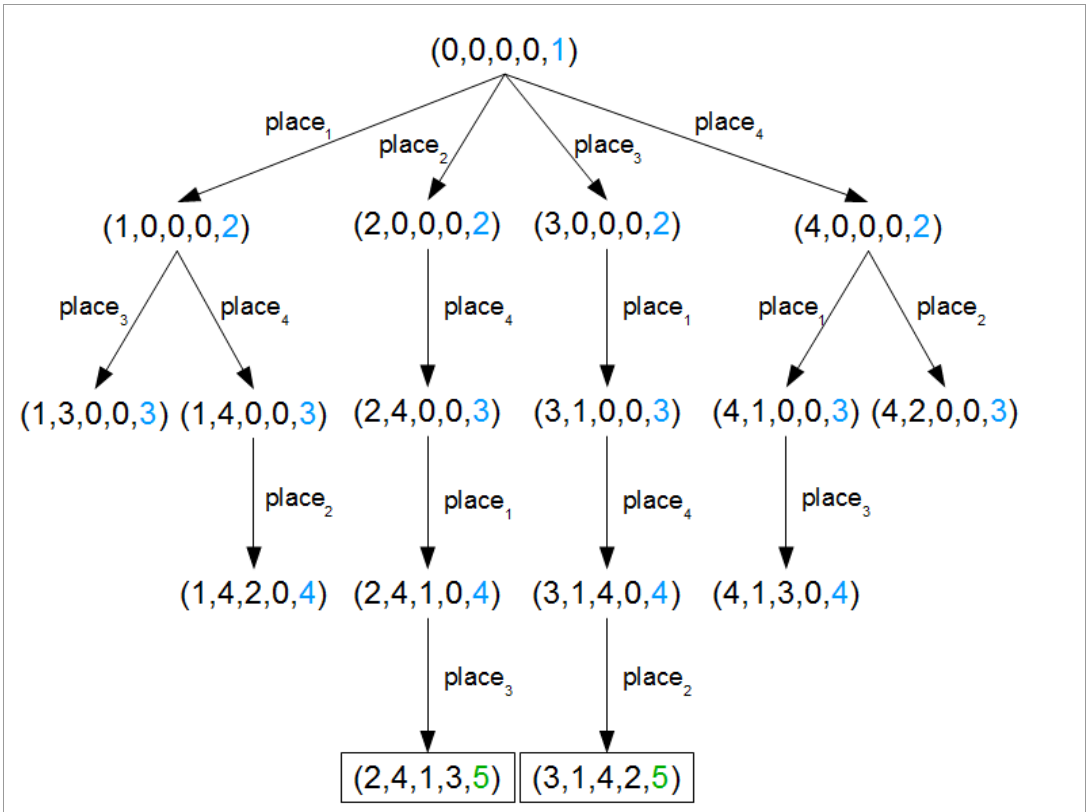


Figure 4. The state-space graph of the 4 Queens problem.

4 Problem-solving methods

Problem-solving methods are assembled from the following components:

- **Database:** stored part of the state-space graph.
  As the state-space graph may contain circles (and loops), in the database we store the graph in an *evened tree* form (see below).
- **Operations:** tools to modify the database.
  We usually differentiate two kinds of operations:
  - operations originated from operators
  - technical operations
- **Controller:** it controls the search in the following steps:
  (1) initializing the database,
  (2) selecting the part of the database that should be modified,
  (3) selecting and executing an operation,
  (4) examining the terminating conditions:
    - positive terminating: we have found a solution,
    - negative terminating: we appoint that there is no solution.

  The controller usually executes the steps between (1) and (4) iteratively.

Unfolding the state-space graph into a tree

Let's see the graph on Figure 5. The graph contains cycles, one such trivial cycle is the edge from s to s or the path s, c, b, s and the path c, d, b, s, c. We can eliminate the cycles from the graph by *duplicating* the appropriate vertices. It can be seen on Figure 6, for example, we eliminated the edge from s to s by inserting s to everywhere as the child of s. The s, c, b, s cycle appears on the figure as the rightmost branch. Of course, this method may result in an infinite tree, so I only give a finite part of it on the figure.

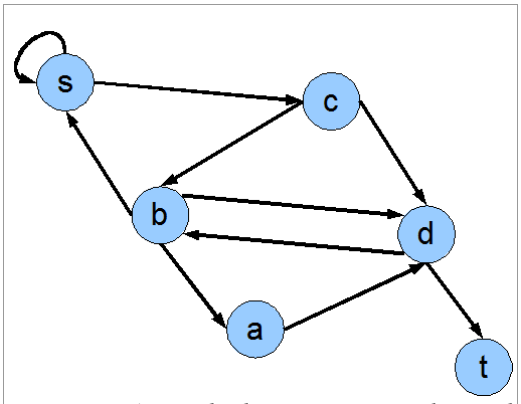


Figure 5. A graph that contains cycles and multiple paths.

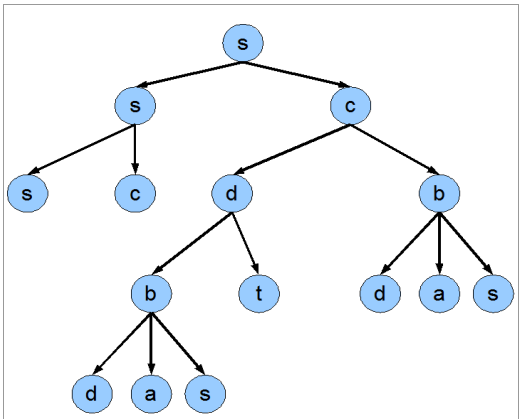


Figure 6. The unfolded tree version.

After unfolding, we need to filter the duplications on the tree branches if we want the solution seeking to terminate after a limited number of steps. That's we will be using different cycle detection techniques (see below) in the controller.

Although, they do not endanger the finiteness of the search, but the *multiple paths* in the state-space graph do increase the number of vertices stored in the database. On Figure 5, for example, the c, d and the c, b, d paths are multiple, as we can use either of them to get from c to d. The c, d and c, b, a, d paths are also multiple in a less trivial way. On Figure 6, we can clearly see what

multiple paths become in the tree we got: the vertices get duplicated, although, not on the same branches (like in the case of cycles), but on different branches. For example, the vertex *d* appears three times on the figure, which is due to the previously mentioned two multiple paths. Note that the existence of multiple paths not only results in the duplication of one or two vertices, but the duplication of some subtrees: the subtree starting at *b* and ending at the vertices appears twice on the figure.

As I already mentioned, the loops do not endanger the finiteness of the search. But it is worth to use some kind of cycle detection technique in the controller if it holds out a promise of sparing many vertices, as we reduce the size of the database on a large scale and we spare the drive. Moreover, this last thing entails the reduction of the runtime.

THE FEATURES OF PROBLEM-SOLVING METHODS

In the following chapter we will get to know different problem-solving methods. These will differentiate from each other in the composition of their databases, in their operations and the functioning of their controllers. These differences will result in problem-solving methods with different features and we will examine the following of these features in the case of every such method:

- **Completeness:** Will the problem-solving method stop after a finite number of steps on every state-space representation, will it's solution be correct or does a solution even exist for the problem? More clearly:
  - If there is a solution, what state-space graph do we need for a solution?
  - If there is no solution, what state-space graph will the problem-solving method need to recognize this?

  We will mostly differentiate the state-space graphs by their finiteness. A graph is considered finite if *it does not contain a circle*.
- **Optimality:** If a problem has more than one solution, does the problem-solving method produce the solution with the lowest cost?

THE CLASSIFICATION OF PROBLEM-SOLVING METHODS

The problem-solving methods are classified by the following aspects:

- *Is the operation retractable?*
  (1) **Non-modifiable problem-solving methods:** The effects of the operations cannot be undone. This means that during a search, we may get into a dead end from which we can't go back to a previous state. The advantage of such searchers is the *simple and small database*.
  (2) **Modifiable problem/solving methods:** The effects of the operations can be undone. This means that we can't get into a dead end during the search. The cost of this is the more *complex database.*
- *How does the controller choose from the database?*
  (1) **Systematic problem-solving methods:** Randomly or by some general guideline (e.g. up to down, left to right). Universal problem-solving methods, but due to their blind, systematic search strategy they are ineffective and result in a huge database.
  (2) **Heuristic problem-solving methods:** By using some guessing, which is done on the basis of knowledge about the topic by the controller. The point of heuristics is to reduce the size of the database so the problem-solving method will become effective. On the other hand, the quality of heuristics is based on the actual problem, there is no such thing as „universal heuristics”.

4.1 NON-MODIFIABLE PROBLEM-SOLVING METHODS

The significance of non-modifiable problem-solving methods is smaller, due to their features they can be rarely used, only in the case of certain problems. Their vital advantage is their *simplicity*. They are only used in problems where the task is not to find a solution (as a sequence of operators), but to decide if there is a solution for the problem and if there is one, than to create a (any kind of) goal state.

The general layout of non-modifiable sproblem-solving methods:

- **Database:** consists of only one state (*the current state*).
- **Operations:** *operators* that were given in the state-space representation.
- **Controller:** The controller is trying to execute an operator on the initial state and will overwrite the initial state in the database with the resulting state. It will try to execute an operator on the new state and it will again overwrite this state. Executing this cycle continues till the current state happens to be a goal state. In detail:

(1) *Initiating*: Place the initial state in the database.

(2) *Iteration*:

(a) *Testing*: If the current state (marked with) is a goal state then the search stops. A solution exists.

(b) Is there an operator that can be executed on ?

- If there isn't, then the search stops. We haven't found a solution.
- If there is, then mark it with . Let $o(a)$ (be the current state.

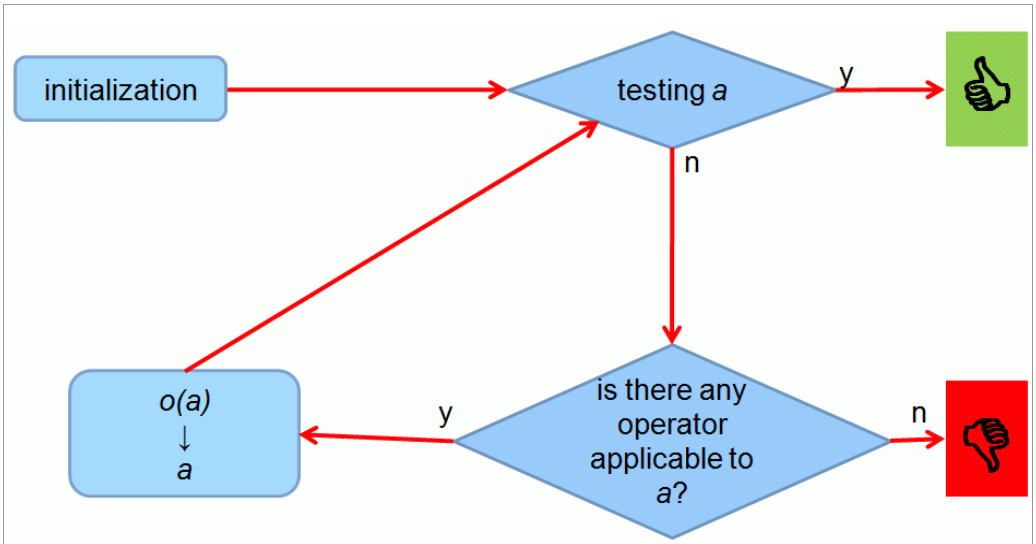


Figure 7. The flowchart of a non-modifiable problem-solving method.

The features of non-modifiable problem-solving methods:

- **Completeness:**
  - Even if there is a solution, finding it is not guaranteed.
  - If there is no solution, it will recognize it in the case of a finite state-space graph.
- **Optimality:** generating the optimal goal state (the goal state that can be reached by the optimal solution) is not guaranteed.

The certain non-modifiable problem-solving methods differ in the way they choose their operator o for the state a. Wemention two solutions:

(1) **Trial and Error method:** o *i*s chosen randomly.

(2) **Hill climbing method:** We choose the operator that we guess to lead us the closest to any of the goal states.

The magnitude of non-modifiable problem-solving methods is that they can be restarted. If the algorithm reaches a dead end, that is, there is no operator we can use for the current state, then we simply restart the algorithm (RESTART). In the same time, we replenish the task to exclude this dead end (which can be most easily done by replenishing the precondition of the operator leading to the dead end). We set the number of restarts in advance. It's foreseeable that by increasing the number of restarts, the chance for the algorithm to find a solution also increases – provided that there is a solution. If the number of restarts approaches infinity, than the probability of finding a solution approaches 1.

The non-modifiable problem-solving algorithms that use restarts are called restart algorithms.

The non-modifiable problem-solving methods are often pictured with a ball thrown into a terrain with mountains and valleys, where the ball is always rolling down, but bouncing a bit before stopping at the local minimum. According to this, our heuristics chooses the operator that brings to a smaller state in some aspect (rolling down), but if there is no such option, than it will randomly select an operator (bouncing) till it is discovered that the ball will roll back to the same place. This will be the local minimum.

In this example, restart means that after finding a local minimum, we throw the ball back again to a random place.

In the restart method, we accept the smallest local minimum we have found as the approached global minimum. This approach will be more accurate if the number of restarts is greater.

The non-modifiable algorithms with restart have great significance in solving the SAT problem. The so-called „random walk" SAT solving algorithms use these methods.

4.1.1 The Trial and Error method

As it was mentioned above, in the case of the trial and error method, we apply a random operator on the current vertex.

- **Completeness:**
  - Even if there is a solution, finding it is not guaranteed.
  - If there is no solution, it will recognize it in the case of a finite state-space graph.
- **Optimality:** generating the optimal goal state is not guaranteed.

The (only) advantage of the random selection is: the infinite loop is nearly impossible.

Idea:

- If we get into a dead end, *restart*.
- In order to exclude getting into that dead end, note that vertex (augment the database).

4.1.2 The trial and error method with restart

- **Database:** the current vertex, the noted dead ends, the number of restarts and the number of maximum restarts.
- **Controller:**

(1) *Initiating:* The initial vertex is the current vertex, the list of noted dead ends is empty, the number of restarts is 0.

(2) *Iteration:* Execute a randomly selected applicable operator on the current vertex. Examine the new state we got whether it is in the list of known dead ends. If yes, then jump back to

the beginning of the iteration. If no, then let the vertex we got be the current vertex.

(a) *Testing:* If the current vertex is a terminal vertex, then the solution can be backtracked from the data written on the screen.

(b) If there is no applicable operator for the current vertex, so the current vertex is a dead end:

- If we haven't reached the number of maximum restarts, the we put the found dead end into the database, increase the number of restarts by one, let the initial vertex be the current vertex, and jump to the beginning of the iteration.
- If the number of maximum restarts have been reached, then write that we have found no solution.

The features of the algorithm:

- **Completeness:**
  - Even if there is a solution, finding it is not guaranteed.
  - The greater the number of maximum restarts is, the better the chances are to find the solution. If the number of restarts approaches infinity, then the chance of finding a solution approaches 1.
  - If there is no solution, it will recognize it.
- **Optimality:** generating the optimal goal state is not guaranteed.

The trial and error algorithm has theoretical significance. The method with restart is called „random walk". The satisfiability of conjunctive normal forms can be most practically examined with this algorithm.

4.1.3 The hill climbing method

The hill climbing method is a *heuristic* problem-solving method. Because the distance between a state and goal state is guessed through a so-called *heuristics*. The heuristics is nothing else but a function on the set of states (A) which tells what approximately the path cost is between a state and the goal state. So:

Definition 7. The heuristics given for the $\langle A, k, C, O \rangle$ state-space representation is a $h: A \to \mathbb{N}$ function, that $h(c)=0 \quad \forall\, c \in C$.

The hill climbing method uses the applicable operator o for the state a where $h(o(a))$ is *minimal*.

Let's see how the hill climbing method works in the case of the Towers of Hanoi! First, give a possible heuristics for this problem! For example, let the heuristics be the sum of the distance of the discs from rod R. So:

$$h(a_1, a_2, a_3) = \sum_{i=1}^{3} (R - a_i)$$

where $R-P=2$, $R-Q=1$, and $R-R=0$. Note that for the goal state (R, R, R) $h(R, R, R)=0$ holds.

(P,P,P)

Initially, the initial state (P, P, P) is in the database. We can apply either the operator $move_{1,Q}$ or $move_{1,R}$. The first one will result in the state (Q, P, P) with heuristics 5, and the later one will result in (R, P, P) with 4. So (R, P, P) will be the current state. Similarly, we will insert (R, Q, P) into the database in the next step.

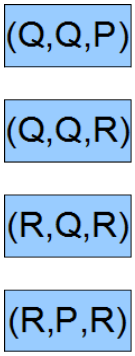
(R,P,P)

(R,Q,P)

Next, we have to choose between two states: we insert either (R, P, P) or (Q, Q, P) into the database. The peculiarity of this situation is that the two states have equal heuristics, and the hill climbing method doesn't say a thing about how to choose between states having the same heuristics. So in this case, we choose randomly between the two states. Note that, if we chose (R, P, P), then we would get back to the previous state, from where we again get to (R, Q, P), where we again

step to (R, P, P), and so on till the end of times. If we choose (Q, Q, P) now, then the search can go on a hopefully not infinite branch.

Going on this way, we meet a similar situation in the state (R, Q, R), as we can step to the states (Q, Q, R) and (R, P, R) with equal heuristics. We would again run into infinite execution with the first one.



We have to say that we need to be quite lucky with this heuristics for hill climbing method even to stop. Maybe, a more sophisticated heuristics might ensure this, but there is no warranty for the existence of such heuristics. All in all, we have to see that without storing the past of the search, it's nearly impossible to complete the task and evade the dead ends.

Note that the Towers of Hanoi is a typical problem for which applying a non-modifiable problem-solving method is pointless. The (only one) goal state is known. In this problem, the goal is to create one given solution and for this a non-modifiable method is inconvenient by its nature.

4.1.4 Hill climbing method with restart

The hill climbing method with restart is the same as the hill climbing method with the addition that we allow a set number of restarts. We restart the hill climbing method if it gets into a dead end. If it reaches the maximum number of restarts and gets into a dead end, then the algorithm stops because it haven't found a solution.

It is important for the algorithm to learn from any dead end, so it can't run into the same dead end twice. Without this, the heuristics would lead the hill climbing method into the same dead end after a restart, except if the heuristics has a random part. The learning can happen in many ways. The easiest method is to change the state-space representation in a way that we delete the current state from the set of states if we run into a dead end. Another solution is to expand the database with the list of forbidden states.

It is worth to use this method if

1. either it learns, that is, it notes the explored dead ends,
2. or the heuristics is not deterministic.

4.2 Backtrack search

One kind of the modifiable problem-solving methods is the backtrack search, which has several variations. The basic idea of backtrack search is to not only store the current vertex in the database, but all the vertices that we used to get to the current vertex. This means that we will store a larger part of the state-space graph in the database: *the path from the initial vertex to the current vertex*.

The great advantage of backtrack search is that the search *can't run* into a dead end. If there is no further step forward in the graph from the current vertex, then we step back to the parent vertex of the current one and try to another direction from there. This special step – called the back-step – gave the name of this method.

It is logical that in the database, besides the stored vertex's state, we also need to store the directions we tried to step to. Namely, in every vertex we have to *register those operators* that we haven't tried to apply for the state stored in the vertex. Whenever we've applied an operator to the state, we delete it from the registration stored in the vertex.

4.2.1 Basic backtrack

- **Database:** *the path* from the initial vertex to the current vertex, in the state-space graph.
- **Operations:**
  - *Operators*: they are given in the state-space representation.
  - *Back-step*: a technical operation which means the deleting of the lowest vertex of the path stored in the database.
- **Controller:** Initializes the database, executes an operation on the current vertex, tests the goal condition, and decides if it stops the search or re-examines the current vertex. The controller's action in detail:

(1) *Initialization:* Places the initial vertex as the only vertex in the database.
Initial vertex = initial state + all the operators are registered.

(2) *Iteration:*

(a) If the database is empty, the search stops. We haven't found a solution.

(b) We select the vertex that is at the bottom of the path (the vertex that was inserted at latest into the database) stored in the database; we will call this the *current vertex*. Let us denote the state stored in the current vertex with a!

(c) *Testing*: If a is a goal state then the search stops. The solution we've found is *the database itself* (as a path).

(d) Examines if there is an operator that we *haven't tried to apply* to a. Namely, is there any more operators registered in the current vertex?

- If there is, denote it with o! Delete o from the current vertex. Test o's precondition on a. If it holds then apply o to a and insert the resulted state $o(a)$ at the bottom of the path stored in the database. In the new vertex, besides $o(a)$, register *all* the operators.
- If there isn't, then the controller *steps back*.

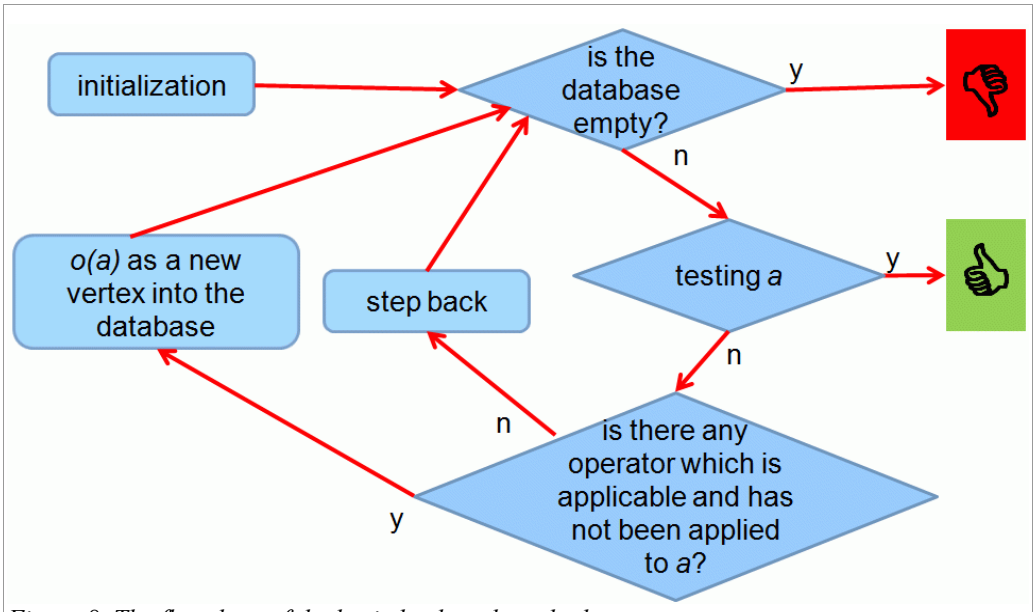


Figure 8. The flowchart of the basic backtrack method.

The backtrack search we have got has the following features:

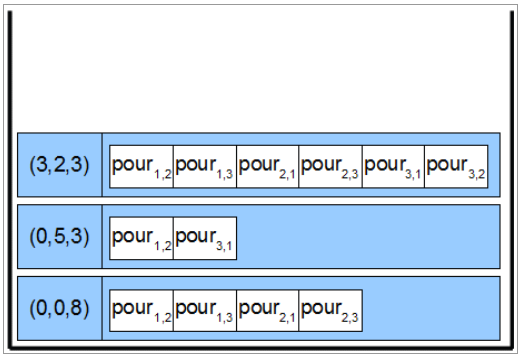
- **Completeness:**
  - If there is a solution, it will find it in a finite state-space graph.
  - If there is no solution, it will recognize it in the case of a finite state-space graph.

- **Optimality:** generating the optimal goal state is not guaranteed.

IMPLEMENTATION QUESTIONS

- **What data structure do we use for the database?**
  Stack.
  The operations of the method can be suited as the following stack operations:
  - applying an operator ⇐ PUSH
  - back-step ⇐ POP
- **In what form do we register the operators in the vertices?**

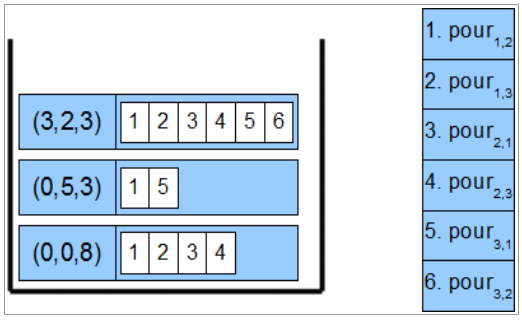
(1) We store a *list of operators* in each vertex.



In this stack, 3 vertices of the 3-mugs-problem can be seen. We have tried to apply the operators $pour_{3,1}$ and $pour_{3,2}$ to the initial vertex (at the bottom). We have got the 2nd vertex by applying $pour_{3,2}$, and we only have the operators $pour_{1,2}$ and $pour_{3,1}$ left. By applying $pour_{2,1}$, we have got the 3rd (uppermost, current) vertex, to which we haven't tried to apply any operator.

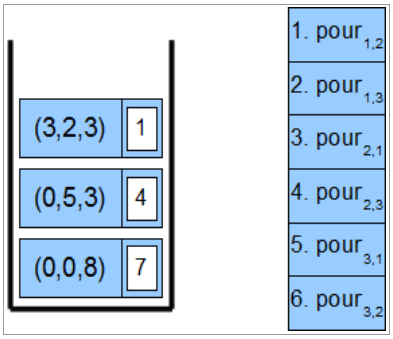
Idea: when inserting a new state, don't store all the operators in the new vertex's list of operators, only those that can be applied to the given state. We can save some storage this way, but it can happen that we needlessly test some operators' precondition on some states (as we may find the solution before their turn).

(2) We store the operators outside the vertices in a constant array (or list). In the vertices, we only store the operator indices, namely the position (index) of the operator in the above mentioned array. The advantage of this method is that we store the operators themselves ain one place (there is no redundancy).



In the case of the 3-mugs-problem, the (constant) array of the operators consists of 6 elements. In the vertices, we store the indices (or references) of the not-yet-used operators.

(3) We can further develop the previous solution by that we apply the operators on every state *in the order of their position they occur in the array of operators*. With this, we win the following: it's completely enough to store one operator index in the vertices (instead of the aforementioned list of operators). This operator index will refer to the operator that we will *apply next to the state stored in the vertex*. In this way, we know that the operators on the left of the operator index in the array of operators have already been applied to the state, while the ones on the right haven't been applied yet.



We haven't applied any operator to the current vertex, so let its operator index be 1, noting that next time we will try to apply the operator with index 1.

To the 2nd vertex, we tried to apply the operators in order, where $pour_{2,1}$ was the last one (the 3rd operator). Next time we will try the 4th one.

We have tried to apply all the operators to the initial vertex, so its operator index refers to a non-existent operator.

The precondition of the back-step can be very easily defined: if the operator index stored in the current vertex is greater than the size of the operators' array, then we step back.

EXAMPLE:

In case of the Towers of Hanoi problem, the basic backtrack search will get into an infinite loop, as sooner or later the search will stuck in one of the cycles of the state-space graph. The number of operator executions depends on the order of the operators in the operators' array.

On Figure 9, we show a few steps of the search. In the upper left part of the figure, the operators' array can be seen. We represent the stack used by the algorithm step by step, and we also show the travelled path in the state-space graph (see Figure 3).

As it can be seen, we will step back and forth between the states (R, P, P) and the (Q, P, P) while filling up the stack. This happens because we have assigned a kind of priority to the operators, and the algorithm is strictly using the operator with the highest priority.

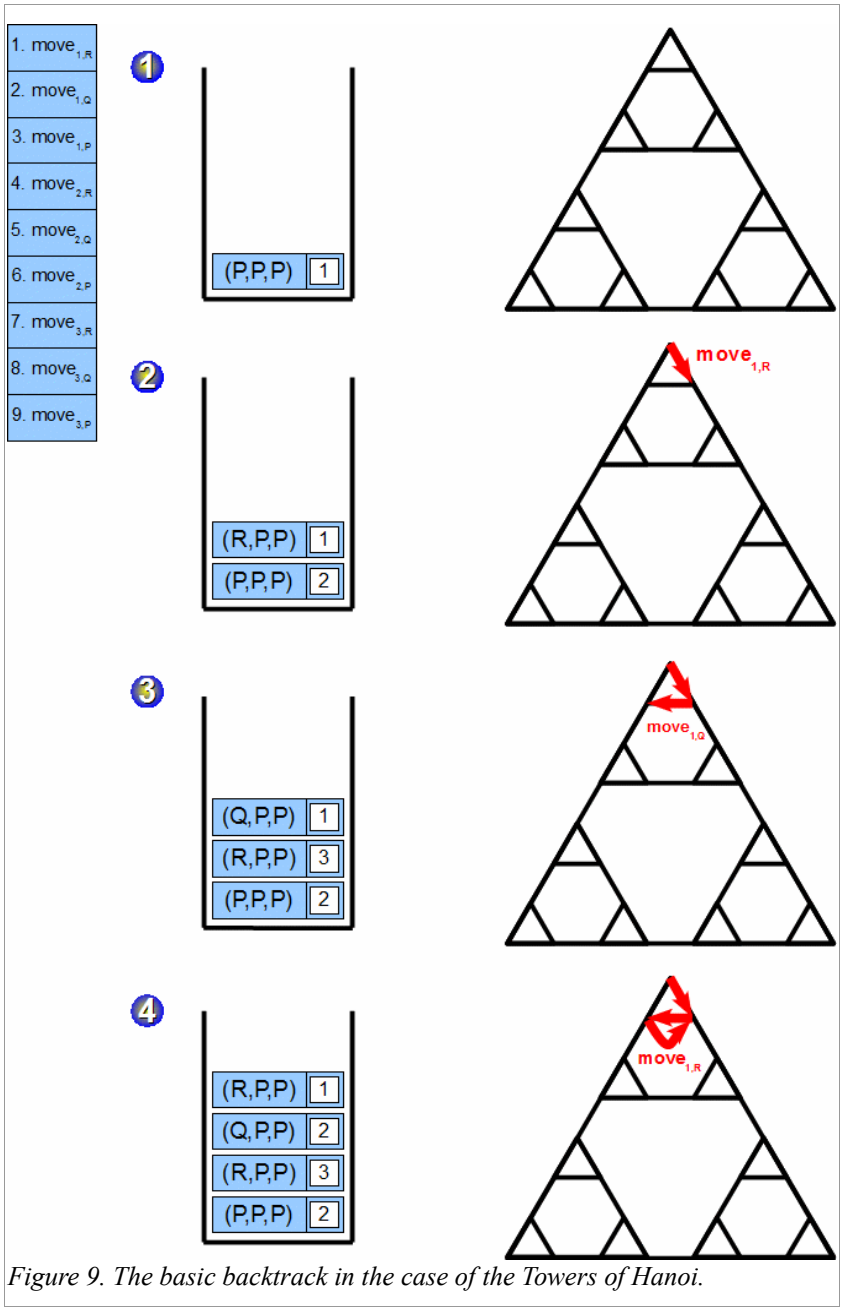


Figure 9. The basic backtrack in the case of the Towers of Hanoi.

4.2.2 Backtrack with depth limit

One of the opportunities for improving the basic backtrack method is the expansion of the algorithm's completeness. Namely, we try to expand the number of state-space graphs that the algorithm can handle.

The basic backtrack method only guarantees stopping after a limited amount of steps in case of a finite state-space graph. The cycles in the graph endanger the finite execution, so we have to eliminate them in some way. We get to know two solutions for this: we will have the backtrack method combined with *cycle detection* in the next chapter, and a more simple solution in this chapter, that does not eliminate the cycles entirely but allows to walk along them only a limited number of times.

We achieve this with a simple solution: *maximizing the size of the database*. In a state-space graph it means that we traverse it only within a previously given (finite) depth. In implementation, it means that we specify the maximum size of the stack ain advance. If the database gets „full” in this sense

during the search, then the algorithm *steps back*.

So let us specify an integer $limit > 0$. We expand the back-step precondition in the algorithm: if the size of the database has reached the $limit$, then we take a *back-step*.

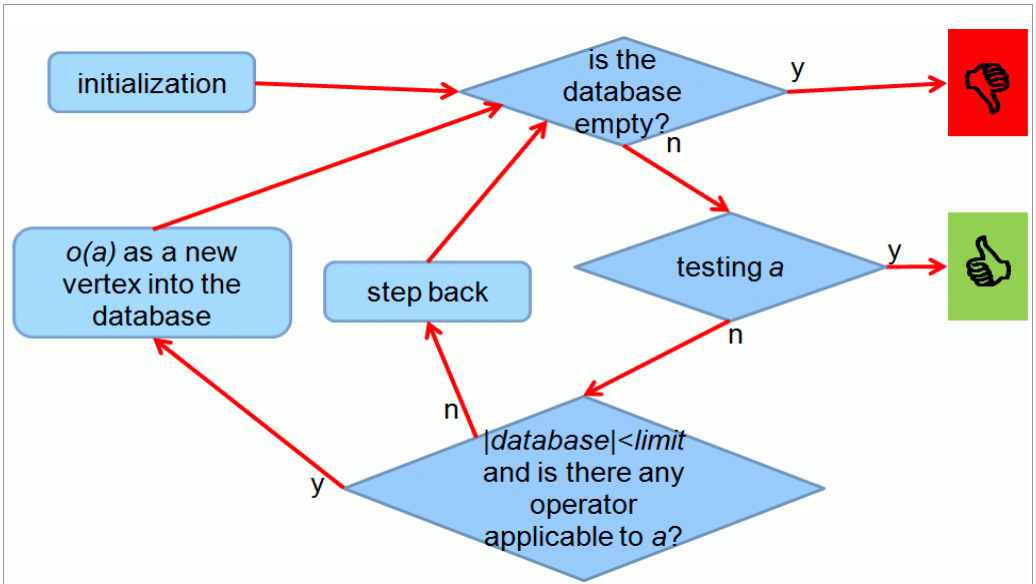


Figure 10. The flowchart of the backtrack method with depth limit.

The resulting backtrack method's features are the following:

- **Completeness:**
  - If there is a solution and the value of the $limit$ is not smaller than the length of the optimal solution, then the algorithm finds a solution in any state-space graph.
    But if we chose the $limit$ too small, then the algorithm doesn't find any solution even if there is one for the problem. In this sense, backtrack with depth limit does not guarantee a solution.
  - If there is no solution, then it will recognize this fact in the case of any state-space graph.
- **Optimality:** generating the optimal solution is *not guaranteed*.

EXAMPLE

On figure 11, we have set a depth limit, which is 7. The depth limit is indicated by the red line at the top of the stack on the figure. Let us continue the search from the point where we finished it on Figure 11, namely the constantly duplicating of the states (R, P, P) and (Q, P, P). At the 7^{th} step of the search, the stack's size reaches the depth limit, a back-step happens, which means deleting the vertex on the top of the stack and applying the next applicable operator to the vertex below it. As it is clearly visible, the search gets out of the infinite execution, but it can also be seen that it will take tons of back-steps to head forth the goal vertex.

Note that if we set the depth limit lower than 7, then the algorithm would not find a solution!

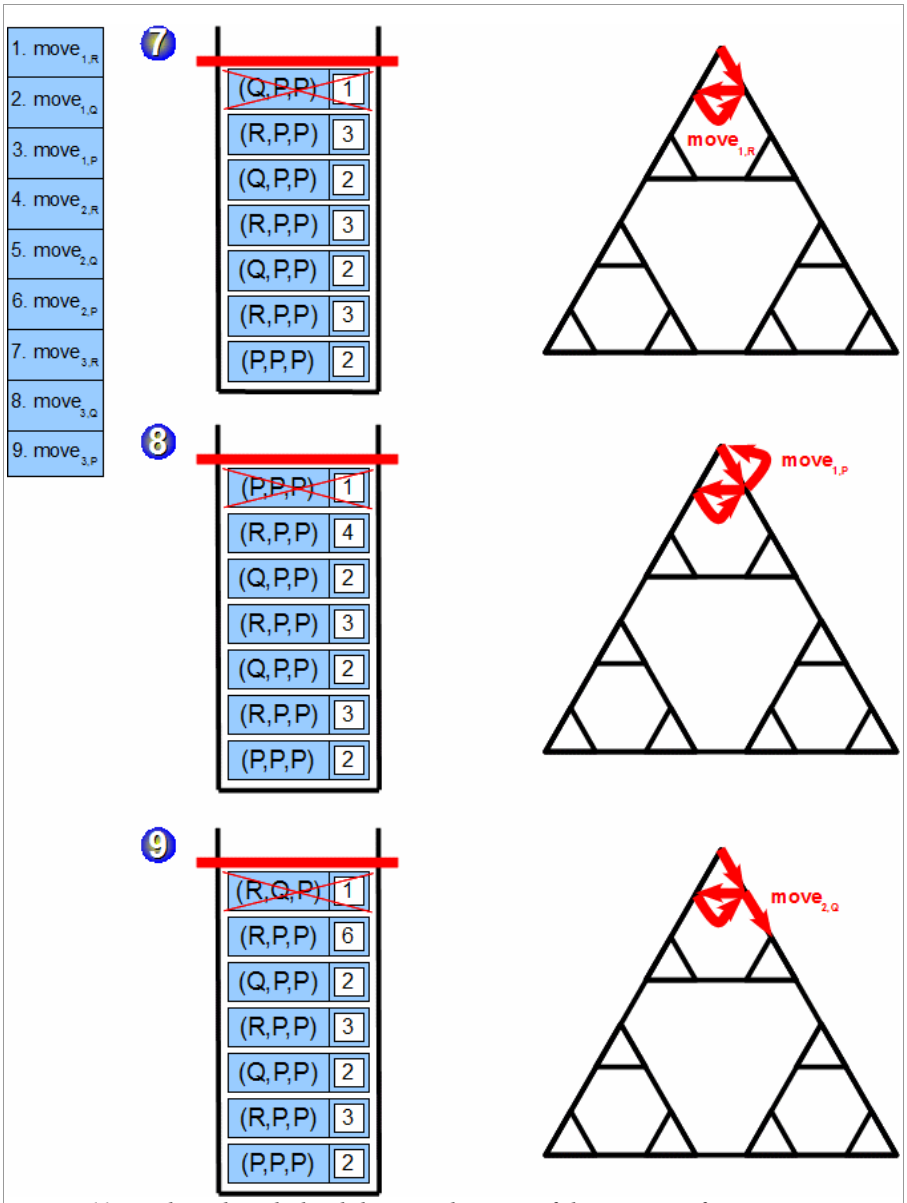


Figure 11. Backtrack with depth limit in the case of the Towers of Hanoi.

4.2.3 Backtrack with cycle detection

For ensuring that the search is finite, another method is the complete elimination of the cycles in the state-space graphe. It can be achieved by introducing an additional test: a vertex can only be inserted as a new one into the database if it hasn't been the part of it yet. That is, any kind of duplication is to be eliminated in the database.

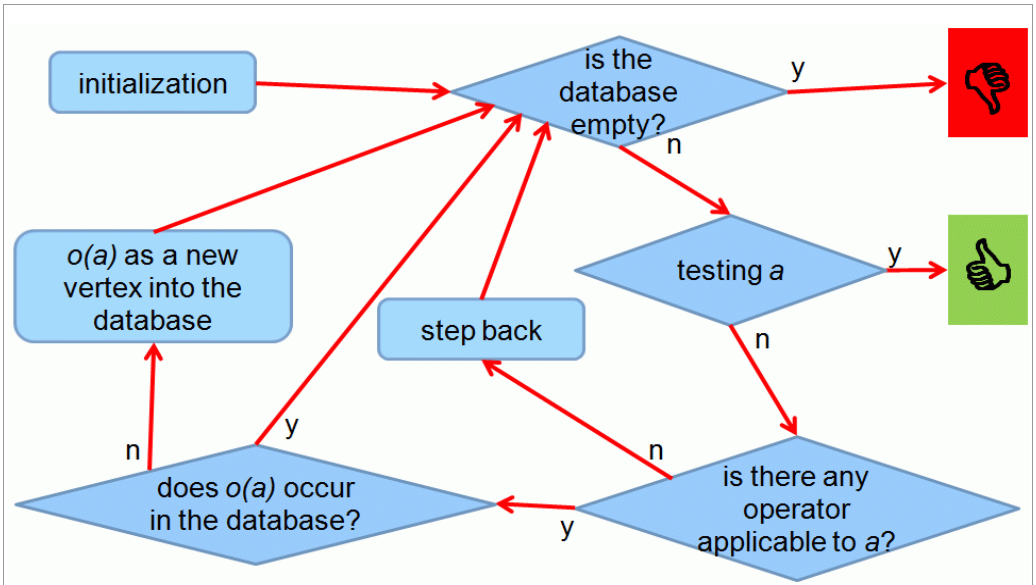


Figure 12. The flowchart of the backtrack algorithm with cycle detection.

The resulting problem-solving method has the best features regarding completeness:

- **Completeness:**
  - If there is a solution, the algorithmfinds it in *any* state-space graph.
  - If there is no solution, the algorithm realizes this fact in the case of *any* state-space graph.
- **Optimality:** finding the optimal solution is *not guaranteed*.

The price of all these great features is a highly expensive additional test. It's important to only use this cycle detection test in our algorithm if we are sure that there is a cycle in the state-space graph. It is quite an expensive amusement to scan through the database any time a new vertex is getting inserted!

EXAMPLE

In Figure 13, we can follow a few steps of the backtrack algorithm with cycle detection, starting from the last but one configuration in Figure 9. Among the move attempts from the state (Q, P, P), there are operators $through_{1,R}$ and $through_{1,P}$, but the states (R, P, P) and (P, P, P) created by them are already in the stack, so we don't put them in again. This is how we reach the operator $through_{2,R}$, which creates the state (Q, R, P), that is not part of the stack yet.

This is the spirit the search is going on, eliminating the duplications in the stack completely. It's worthwhile to note that although the algorithm cleverly eliminates cycles, it reaches the goal vertex in a quite dumb way, so the solution it finds will be far from optimal.

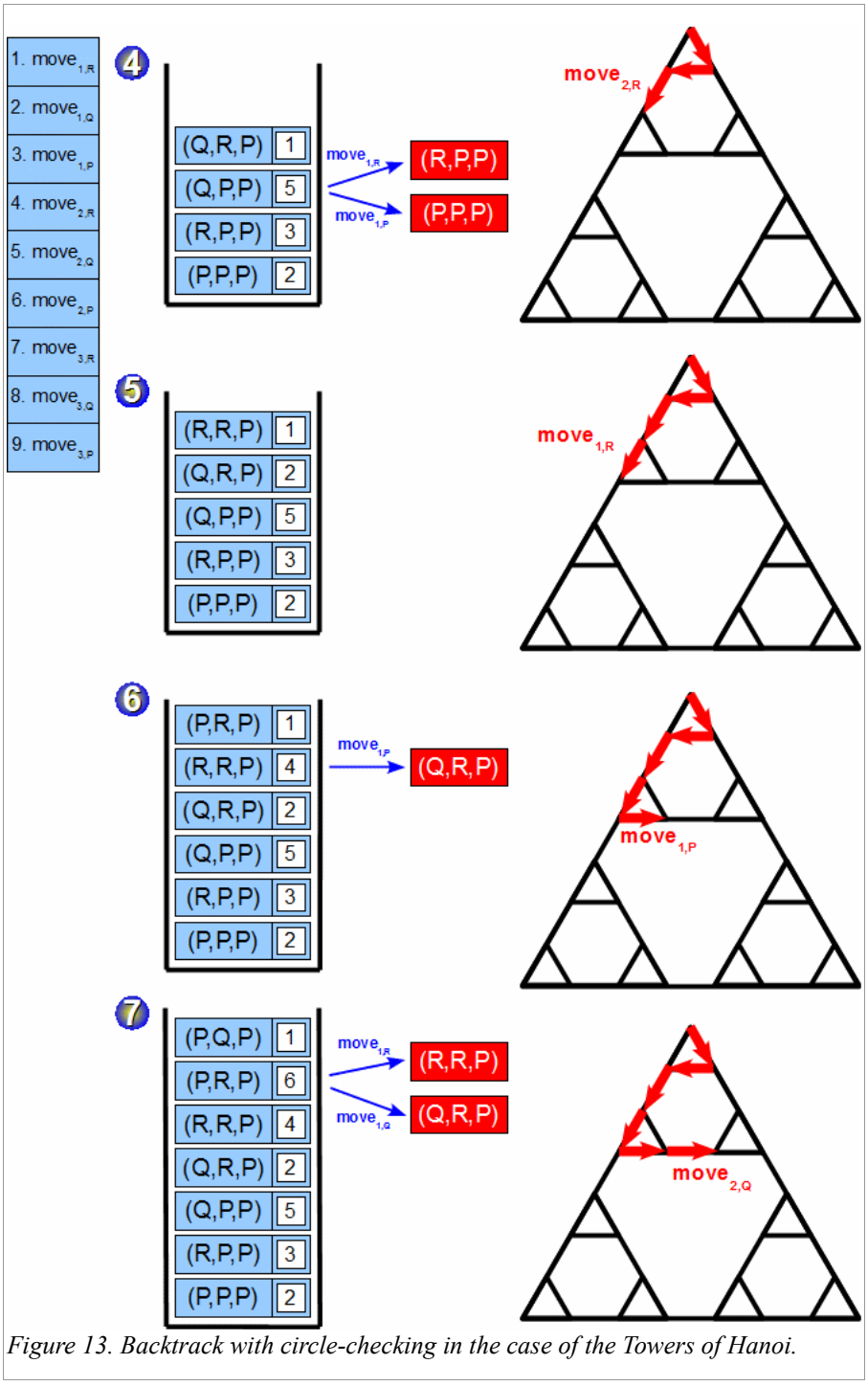


Figure 13. Backtrack with circle-checking in the case of the Towers of Hanoi.

4.2.4 The Branch and Bound algorithm

We can try to improve the backtrack algorithm shown in the last chapter for the following reason: to guarantee the finding of an *optimal solution*! An idea for such an improvement arises quite naturally: the backtrack algorithm should perform minimum selection in the universe of possible solutions.

So, the algorithm will not terminate when it finds a solution, but will make a copy of the database (the vector called *solution* will be used for this purpose), then *steps back* and continues the search. It follows that the search will only end if the *database gets empty*. As we don't want to traverse the whole state-space graph for this, we are going to use a version of the *backtrack algorithm with*

depth limit in which the value of the *limit dynamically changes*. Whenever finding a solution (and storing it in the vector *solution*), the *limit* 's new value will be the length of the currently found solution! So, we won't traverse the state-space graph more deeply than the length of the last solution.

It's obvious that the resulting backtrack algorithm – which is called the B*ranch and Bound algorithm* – finds an optimal solution (if the given problem has a solution).

At the beginning of the search, the variables *solution* and *limit* must be initialized. The *solution* is an empty vector at the beginning, and the *limit* is such a great number that is by all means greater than the length of the optimal solution. The programmers in most cases set the value of *limit* to the largest representable integer value. The Branch and Bound algorithm is usually combined with cycle detection.

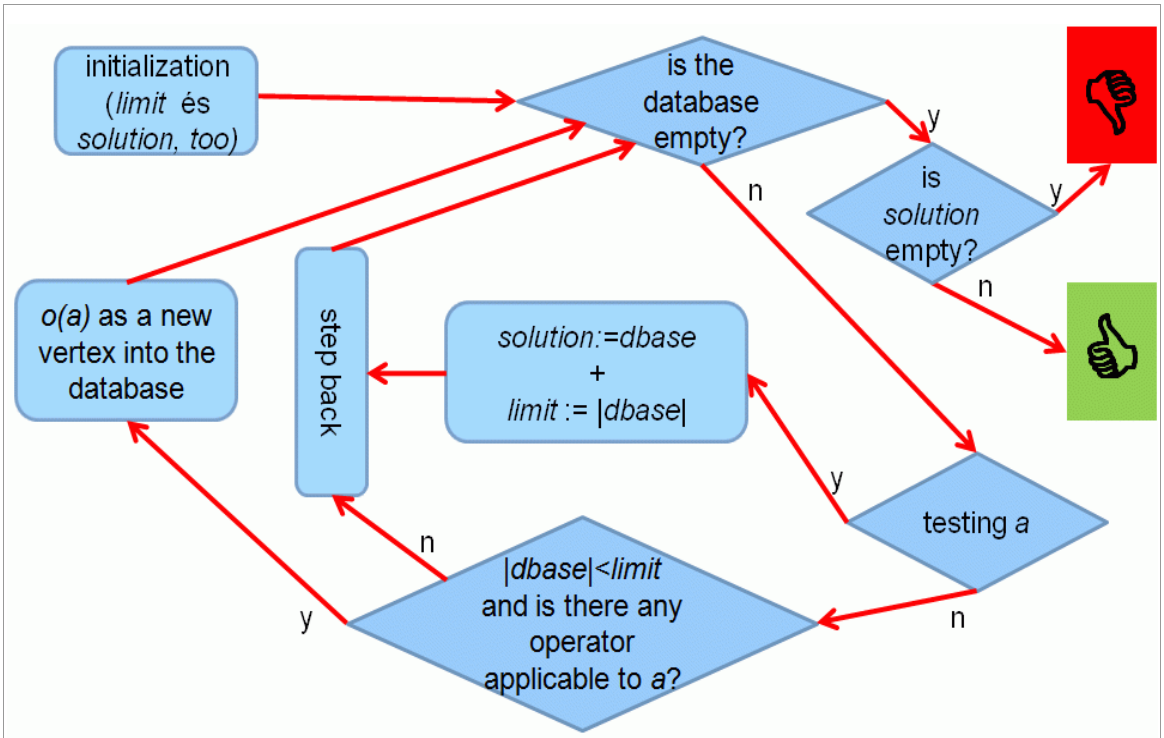


Figure 14. The flowchart of the Branch and Bound algorithm.

The features of the Branch and Bound algorithm:

- **Completeness:**
  - If there is a solution, the algorithm finds it in *any* state-space graph.
  - If there is no solution, the algorithm realizes this fact in the case of *any* state-space graph.
- **Optimality:** finding the optimal solution is *guaranteed*.

4.3 TREE SEARCH METHODS

Another large class of the modifiable problem-solving algorithms is the class of the tree search methods. The basic difference compared to the backtrack algorithms is that we not only store one branch of the state-space graph in the database, but a more complex part of it, in the form of a tree. As a matter of fact, the search is done simultaneously on several branches at the same time, so we probably get a solution sooner, which is maybe an optimal solution, too. An obvious disadvantage of this method is its higher space complexity.

In the next chapter, we give the description of a general tree search problem-solving method That

method is not a concrete algorithm, but includes the common components of the later described other algorithms – the Breadth-first method, the Depth-first method, the Uniform-cost method, the Best-first method, and the A algorithm.

4.3.1 General tree search

- **Database:** It is a part of the state-space graph unfolded into a tree. The vertices of the tree stored in the database are divided into two classes:
  - *Closed vertices:* previously expanded vertices (c.f. operations).
  - *Open vertices:* not yet expanded vertices.
- **Operation: expansion**.
  Only open vertices can be expanded. Expanding a vertex means the following: *all the applicable operators* are applied to the vertex. Practically speaking, all the children vertices (in the state-space graph) of the given vertex are created.
- **Controller:** Initializes the database, expands the chosen open vertex, tests the goal condition, and, according to this, decides either top with the search or to expand another vertex. In details:
  (1) *Initialization: Inserts* the initial vertex as the only one into the database as an open vertex.
  (2) *Iteration:*
  (a) If there is no open vertex in the database, the search stops. It hasn't found a solution.
  (b) It selects one of the open vertices, which is going to be the *current vertex*. Let us denote this vertex v !
  (c) If v is a goal vertex, the search stops. The found solution is: the path from the initial vertex to v in the database.
  (d) Expands v, inserts the created new states as children of v and open vertices into the database, then reclassifies v as a closed vertex.

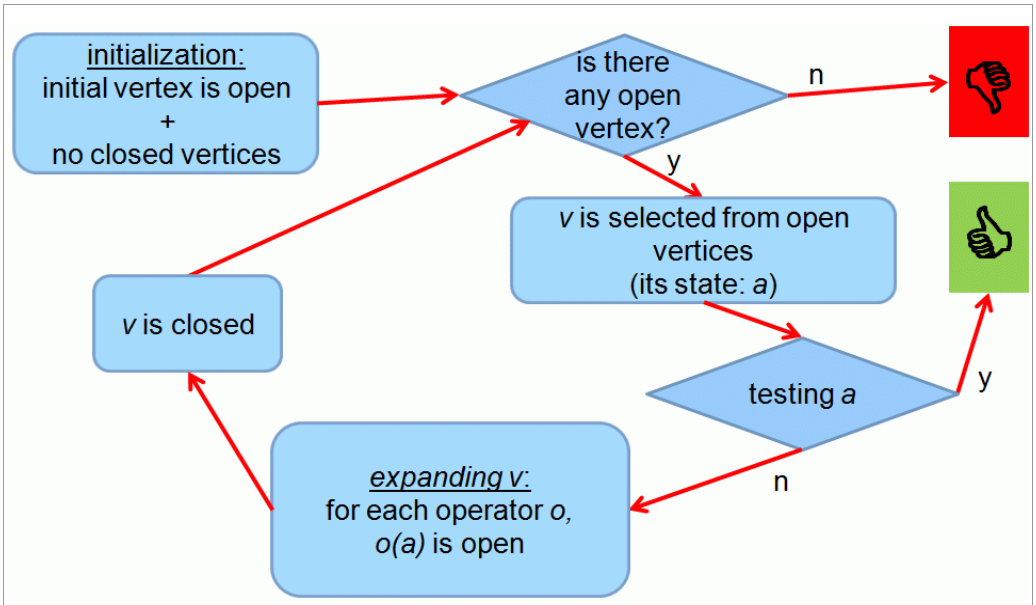


Figure 15. The flowchart of tree search methods.

There are only three not completely fixed elements in the aforementioned general tree search method:

- at (2)(a): If there are several open vertices in the database, which one should be chosen for expansion?

- at (2)(c): The goal condition is tested on the current vertex. Thus, after a vertex has been inserted into the database, we have to wait with the testing of the goal condition until the given vertex gets selected for expansion. In case of some problem-solving methods, the testing can be performed sooner in order to haste the search, which means that we test the goal condition immediately (before inserting into the database) for the new states that were created in (2)(d).
- at (2)(d): Should we use any kind of cycle detection technique, and if we do, which one? Namely, if any state that comes into being due to expansion occurs in the database, shall we insert it again?

Two sorts of cycle detection techniques can be used:

- We scan the whole database looking for the created state. In this case, we are not only checking for cycles but also for multiple paths, since there won't be two identical states in the database. Fully excluding The entire elimination of duplications obviously speeds up the search, but scanning the whole database all the time is quite a costly procedure.
- Only the branch that leads to the current vertex is scanned. In this case, we only check for cycle and don't exclude multiple paths from the database. This is a less costly procedure than the previous one, but the database may contain duplications.

Detecting cycles or multiple paths in our algorithm depends on the nature of the solvable problem. If the problem's state-space graph doesn't contain any cycles, then, naturally, cycle detection is not needed. Otherwise, a cycle detection must be included by all means. If the graph contains only a few multiple paths, then it's sufficient to use the less costly procedure, but if it contains many, and hence, duplications substantially slow the search, then it's expedient to choose the full scan (since this cycle detecting procedure also checks for multiple paths).

It is matter of by arrangement *which open vertex is selected for expansion* at (2)(a). *This is the point* where the concrete (not general) tree search methods *differ* from each other. In the next chapters, we introduce the most important such problem-solving methods, and we examine which one is worth to use for what kind of problems.

IMPLEMENTATION QUESTIONS

- **What data structure should be used to realize the vertices of the database?**
  Every vertex should store *pointers pointing to the children* of the vertex. This requires the use of a *vertex list* in every vertex. It's more economic to store a pointer *pointing to the vertex's parent*, since a vertex can have several children, but only one parent (except for the root vertex, which does not have any parent).
- **How should the open and closed vertices be registered?**
  One option is to store the information about being open or closed in the vertex itself. This solution implies that we always have to scan for the open vertex we want to expand in the tree. On the one hand, this is a quite costly method, on the other hand, if we stored, according to the previous point, the parent pointers, then traversing the tree top-down is impossible.
  So, it would be practical to store the vertices in one more list. We would use a list for the open vertices, and another list for the closed ones. The vertex to be expanded would be taken from the list of open vertices (at the beginning of the list), and after expansion, it would be moved into the list of closed vertices. The new vertices created during the expansion would be inserted into the list of open vertices.
- **How to apply cycle detection?**
  If we only perform *the scanning on the current branch*, then this can be easily realized, because of the parent pointers.
  If we chose *the scanning of the whole database* (i.e., we are also looking for multiple paths), then our task is more difficult, since we have to traverse the whole tree, which is impossible due to the parent pointers. But if we store the vertices in the above-mentioned two

lists, then this kind of cycle detection can easily be performed: we need to scan both the list of the open vertices and the list of the closed ones.

4.3.2 Systematic tree search

On page 21, we can see a classification of problem-solving methods that differentiate systematic methods and heuristic methods. In this chapter, we get to know the systematic versions of tree search methods.

4.3.2.1 Breadth-first search

The breadth-first method always expands an open vertex with the *smallest depth* (if there are more than one such vertices, it selects randomly). By this method, the different levels (vertices with the same depth) of the tree are created breadth-wise, i.e., only after a level has fully been created we go on with the next level. This is where the method's name comes from.

For the exact description, we assign a so-called *depth number* to each of the vertices stored in the database. The breadth-first method selects the vertex with the smallest depth number to expand.

Definition 8. The depth number of a vertex in a search tree is defined the following way:

- $g(s)=0$, where s is the initial vertex.
- $g(m)=g(n)+1$, where m is a child of the vertex n.

It can be easily seen that, if the breadth-first method finds a solution, it is the *optimal solution*. This, of course, has a cost: all the levels of the tree must be generated in breadth-wise, which means a lot of vertices in case of certain problems. In practice, it causes difficulties when the problem to be solved has long solutions, since finding them can be extremely time-consuming.

- **Completeness:**
  - If there is a solution, the method finds it in *any* state-space graph.
  - If there is no solution, the method realizes this fact in the case of a finite state-space graph.
- **Optimality:** finding the optimal solution is *guaranteed*.
- **Testing:** can be performed sooner.

Although the breadth-first method finds a solution in finite steps without any cycle detection techniques (assuming that there is a solution), in case of certain problems one of the extra tests from Chapter 4.3.1 should be added. Naturally, this is worth only in connection with problems with frequent cycles (and multiple paths) in their state-space graphs, since we can substantially lower the number of vertices that are inserted into the database. Not to speak of the fact that we can also realize in finite steps if there is no solution.

Implementation questions

- **How to select the open vertex with the smallest depth number?**
  One option is to store the vertex's depth number in itself, and, before every expansion, to look for the vertex with the smallest such number in the list of open vertices.
  Another opportunity for this is to store the open vertices ordered by their depth number in a list. The cheapest way of guaranteeing ordering is *to insert each new vertex at the correct position* (by its depth number) into the already-sorted list.
  It's easy to notice that, in this way, new vertices will always get to end of the list. So, the list of open vertices functions as a data structure where the elements *get in at the end and leave at the front* (when they are being expanded). I.e., the most simple data structure to store open vertices in is a *queue*.

EXAMPLE

In Figure 16, a search tree generated by the breadth-first method can be seen, for a few steps, in the case of the 3-mugs problem. It's interesting to follow on Figure 2 how this search tree look like according to the state-space graph. In the search tree, we illustrate the open vertices with ellipses, and the closed vertices with rectangles. The search tree given here has been built by the breadth-first method without either cycle or multiple path detection. The red edges would be eliminated by a cycle detection technique, which filters *the duplications on branches*. The yellow edges are the ones (besides the red ones) that would be eliminated by a complete detection of cycles and multiple paths (namely, by scanning over the database). It can be seen that such a cycle detection technique reduces the size of the database radically, or at least it does in the case of the 3-mugs problem.

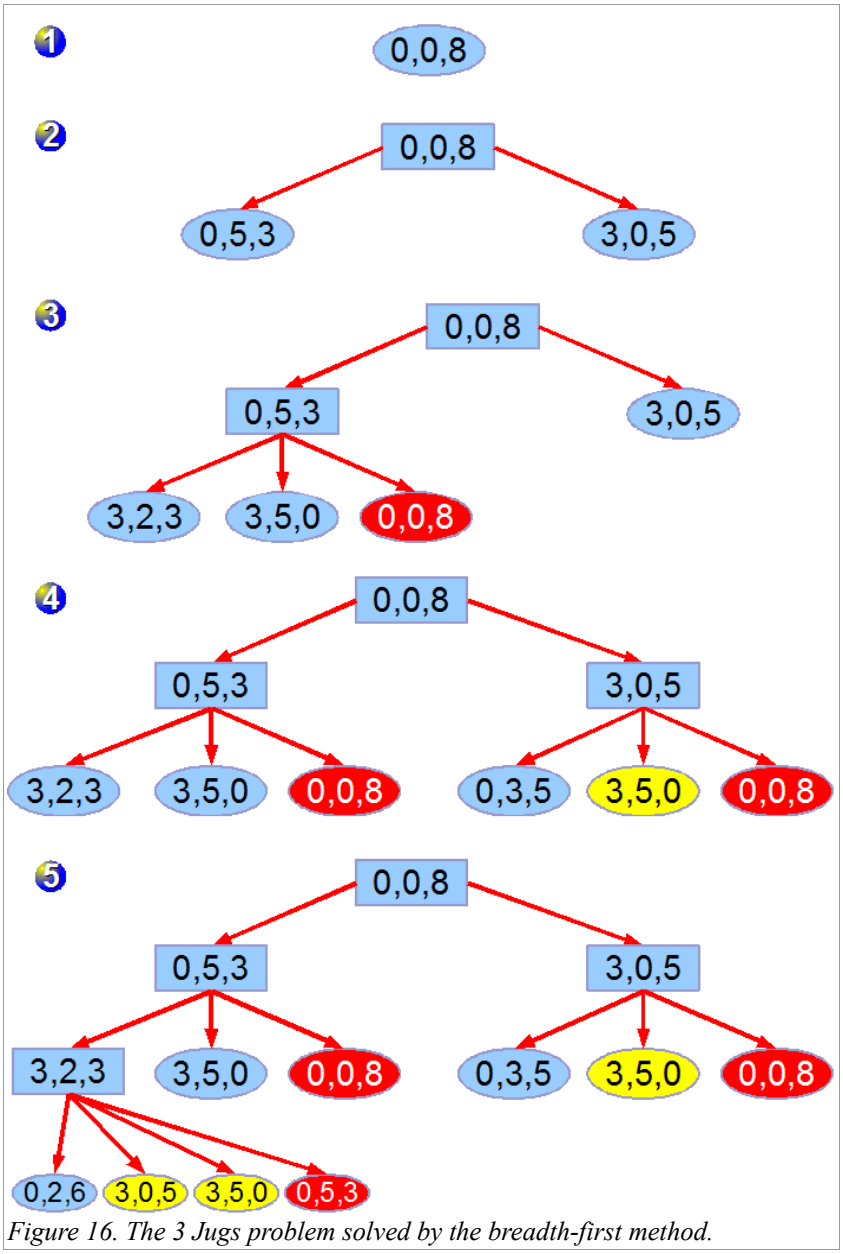


Figure 16. The 3 Jugs problem solved by the breadth-first method.

4.3.2.2 DEPTH-FIRST SEARCH

The depth-first search method expands an open *vertex with the highest depth number*(if there are more than one such vertices, it selects randomly). The result of this method is that we don't need to generate the tree's levels breadth-wise, thus, we may quickly find goal vertices even if they are deep in the tree. This quickness, of course, has a price: there's *no guarantee that the found solution is*

optimal.

The depth-first method, compared to the breadth-first method, usually find a solution sooner, but it also depends on the state-space graph. If the graph contains many goal vertices, or they are deep in the tree, then the depth-first method is the better choice; if the goal vertices are rare and their depths is small, then the breadth-first method is better. If our goal is to find an optimal solution, depth-first search is (generally) out of the question.

- **Completeness:**
  - If there is a solution, the method finds it in *a finite* state-space graph.
  - If there is no solution, the method realizes this fact in the case of a finite state-space graph.
- **Optimality:** finding the optimal solution is *not guaranteed*.
- **Testing:** can be performed sooner.

The similarity between the depth-first method and the backtrack method is striking. Both explore the state-space graph „in depth", both are complete if the graph is finite (if we don't use a cycle detection technique, of course) and none of them is for finding the optimal solution. The question is: what extra service does the depth searcher offer compared to the backtrack method? What do we win by not just storing the path from the initial vertex to the current vertex in the database, but by storing all the previously generated vertices? The answer is simple: we can perform a *multiple path detection*. So, by combining the depth-first method with the cycle and multiple path detection techniques detailed in Chapter 4.3.1 (namely by scanning the database before inserting a new vertex) the method "runs into" a state only once during the search. Such a search is impossible with the backtrack method.

IMPLEMENTATION QUESTIONS

- **How to select the open vertex with the highest depths number?**
  While in the case of breadth-first search, open vertices are stored in a queue, in the case of depth-first search it's practical to store them in a *stack*.

EXAMPLE

In Figure 17, we introduce the search tree of the 3 jugs problem generated by the depth-first method. Since the state-space graph of this problem is not finite (there are cycles in it), using some kind of a cycle detection technique is a must. The vertices excluded by cycle detection are red, while the vertices excluded by multiple path detection are yellow.

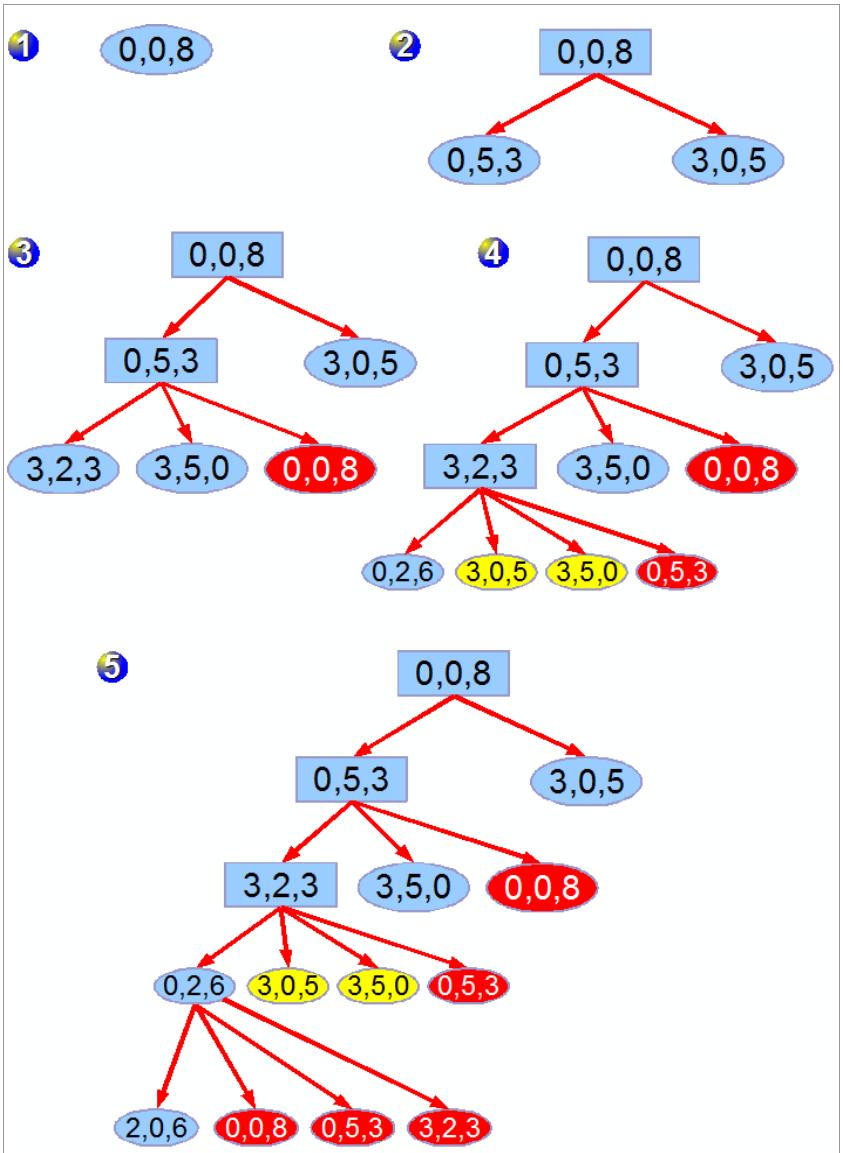


Figure 17. The 3 Jugs problem solved by the depth-first method.

4.3.2.3 Uniform-cost search

The uniform-cost search is used for problems whose operators has some kind of *cost* assigned to. Let's recall the concepts and notations from Page 11: the cost of an operator o applied to a state a is denoted by $cost_o(a)$, and the cost of a solution equals to the sum of the costs of all the operators contained by the solution.

In case of the *breadth-first* and the *depth-first* methods, we did not assign a cost to operators. Naturally, not all problems are such, that is why we need the uniform-cost method. To describe the search strategy of the uniform-cost method, we have to introduce the following concept (like the depth number defined on Page 37):

Definition 9. The cost of a vertex in the search tree is defined as follows:

- $g(s)=0$, where s is the initial vertex.
- $g(m)=g(n)+cost_o(n)$, where we have got m by applying the operator o operator to n.

The *uniform-cost method* always extends the open vertex with the *lowest cost*.

Notice that the *depth number* used by the breadth-first and depth-first methods is actually a special vertex cost, in the case when $cost_o(n)=1$ for every operator o and every state n. So, the

following fact can be appointed: *the breadth-first method is a special uniform-cost method, where the cost of every operator is one unit*.

The properties of the uniform-cost method:

- **Completeness:**
  - If there is a solution, the method finds it in *any* state-space graph.
  - If there is no solution, the method realizes this fact in the case of a finite state-space graph.
- **Optimality:** finding the optimal solution is *guaranteed*.
- **Testing:** can't be performed sooner, since such a modification endangers the finding of an optimal solution.

In case of the breadth-first and depth-first methods, *cycle detection* was simple (if a vertex has already got into our database, we haven't put it in again), but with the uniform-cost method, it gets more complicated. Examine the situation when the vertex m is created as the child of the vertex n, and we have to insert it into the database! Assume that m is already in the database! In this case, there are two possibilities according to the relation between the cost of m (denoted by $g(m)$) and the cost of the new vertex (which is actually $g(n)+cost_o(n)$):

- $g(m) \leq g(n)+cost_o(n)$: The database contains the new vertex with a more optimal path (the cost is higher). In this case, the new vertex is *not added* to the database.
- $g(m) > g(n)+cost_o(n)$: We managed to create m on a more optimal path. Then change the vertex m stored in the database to the new vertex!

This last operation can be easily done: set m tot be chained to n, and update the cost assigned to m (its g-value) to $g(n)+cost_o(n)$.

A problem arises with such a chaining when m already has children in the tree, since, in this case, the g-values of the vertices occurring in the subtree starting from m must be updated (since all of their g-values originate in $g(m)$). In other words, we have a problem when m is closed. This is called *the problem of closed vertices*. Several solutions exist for the problem of closed vertices, but, fortunately, we don't need any in case of the uniform-cost method, since the problem of the closed vertices *can't occur* with the uniform-cost method. This is guaranteed by the following statement:

Statement 1. If a vertex m is closed in the database of the uniform-cost method, then $g(m)$ is optimal.

Proof. If m is closed, then we expand it sooner than the currently expanded vertex n. So,

$$g(m) \leq g(n)$$
$$\Downarrow$$
$$g(m) < g(n)+cost_o(n)$$

Namely, any newly explored path to m is more costly than the one in the database.

IMPLEMENTATION QUESTIONS

- **How to select the open vertex with the lowest cost?**
  Unfortunately, in case of the uniform-cost method, neither a stack nor a queue can be used to store open vertices. The only solution is to register the costs within the vertices, and to sort the list of open vertices by cost. As we wrote earlier, the sorting of the database should be guaranteed by insertion sort.

4.3.3 HEURISTIC TREE SEARCH

The uniform-cost method and the breadth-first method (which is a special uniform-cost method) have very advantageous features, including the most important one, namely the ability to guarantee

optimal solution. But this has a heavy cost, which is the long execution time. This is caused by the large database. The search tree stored in the database can consist of a lot of vertices. For example, if a vertex in the tree has d children at most, and the length of the optimal solution is n, then the search tree consists of

$$1+d+d^2+d^3+\ldots+d^n = \frac{d^{n+1}-1}{d-1}$$

vertices at most. Namely, the number of the vertices in the search tree is exponential in the length of the solution.

The uniform-cost method is a so-called systematic method, that is it uses some „blind" search strategy, and this causes the necessity of generating so many vertices. To avoid this, we try to enhance our searchers with some kind of foresight, to wire our own human intuition up in the method. The tool for this purpose is the so-called *heuristics*. Heuristics is actually an *estimation*, that tells is which child of a vertex should we select to go on with the search. Notice that it is really an estimation, since the subtree starting from the child has not been yet generated, so we can't be sure if we are going to the right way (towards a goal vertex) in the tree

Anyway, it would be the best for us to have such a heuristics, that could exactly tell us in every vertex which way (which child) leads us to the closest goal vertex. Such a heuristics is called *perfect heuristics* and can generate

$$1+d+d+\ldots+d = 1+n\cdot d$$

vertices at most, so in the case of such a heuristics, the number of vertices is only *linear* in the length of the solution. Of course, perfect heuristics *does not exist*, since if it existed then we could know the solution in advance, so why bother with searching for it?!

The definition of heuristics is quite simple, as we have already introduced it in Chapter 4.1.3, in connection with the hill climbing method. The point is that the heuristics as a function assigns a natural number to every state. With this number we estimate the total cost of the operators that we need to use to get from a given state to a goal state. If we think of a search tree: the heuristic value of a vertex approximately gives *the cost of a path to one of the goal vertices*. In a state-space graph, where the edges have the same cost (like in the case of breadth-first search), the heuristic value of a vertex estimates *the number of edges* that leads into one of the goal vertices. Anyhow, by using heuristics, we hope to reduce the size of the search tree.

4.3.3.1 Best-first search

The best-first method is a tree search method which selects the open vertex with the lowest heuristic value to be expanded. The advantage of the resulting method is its simplicity, but, in the same time, it loses on important features as compared to the uniform-cost (and the breadth-first) method.

- **Completeness:**
  - If there is a solution, the method finds it in *any* state-space graph.
    Exception: if the heuristic value of every state is the same (1 unit).
  - If there's no solution, the method recognizes this fact in the case of a *finite* state-space graph.
- **Optimality:** finding the optimal solution is *not guaranteed*.
- **Testing:** can be performed sooner, since we can't expect generating an optimal solution.

Implementation questions

- **How to select the open vertex with the lowest heuristic value?**

Obviously, the list of open vertices have to be sorted by the heuristic values of the vertices.

4.3.3.2 The A algorithm

In connection with the best-first method, we could see that using heuristics ruined all the good features of the uniform-cost (and the breadth-first) method. However, we need to use heuristics to solve the efficiency problems of the uniform-cost method. The uniform-cost method isn't effective since it only considers the „past" during the search. The best-first method, on the other hand, only „gazes into the future", not learning from the past of the search, hence it doesn't always walks on the logical way.

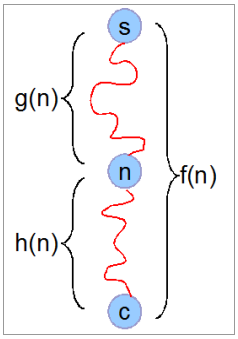
The idea: Combine the uniform-cost method with the best-first method! The resulting method is called the *A algorithm*. To describe the search strategy of the A algorithm, we need to introduce the following concept:

Definition 10. The *total cost* of a vertex n is denoted by $f(n)$, and defined as follows:

$$f(n)=g(n)+h(n).$$

The A algorithm always selects the open vertex with the lowest total cost (f-value) to expand.

The f-value of a vertex n is actually the estimated cost of a path leading from the initial vertex to a goal vertex via n (namely a solution via n).



Notice that *the uniform-cost method is a special A algorithm where the heuristic value is zero for every vertex*. This, of course, means that the breadth-first method is a special A algorithm, too, since the costs of the operators are equal and the heuristics is constant zero.

But there is a very sensitive point in the A algorithm: *the cycle detection*. The same can be told as with the uniform-cost method (c.f. Chapter 4.3.2.3): If the vertex we would like to insert is already in the database, and now it is created with a lower cost, we have to replace the vertex in the database with the new one (update its parent and g-value)! However, *the problem of closed vertices* could not occur in the case of the uniform-cost method, but *it can happen* in the A algorithm. Namely, it can happen that the vertex we want to replace (m) is closed, so it has descendants in the database, which means that we have to update their g-values, too. The possible solutions for this problems are:

(1) Traverse the subtree starting from m in the database, and update the g-values of the vertices in it! Since we are only using parent pointers due to implementation questions, such a traverse is *not possible!*

(2) Delegate the updating of the values to the A algorithm! We can force this by reclassifying m as an open vertex. This means that every descendant of m (all the vertices of the subtree starting from m) will be regenerated once again with a lower g-value than their current one, so their g-values will be updated by the A algorithm, too.

(3) Avoid the occurrence of the problem of closed vertices! This problem doesn't appear at all in the uniform-cost method, so – as the uniform-cost method is a special A algorithm – the question is *what kind of heuristics is needed to completely avoid the problem of closed vertices*?

Option (3) will be examined in Chapter 4.3.3.4. For now, we use option (2)! So we allow the algorithm to reclassify closed vertices to open vertices in certain cases. However, this brings forth the danger of a never-stopping algorithm, since we don't know how many times (or even infinitely) a vertex will be reclassified. Fortunately, the following statement can be proved:

Statement 2. Any vertex of the search tree will be finitely reclassified to open.

Proof. According to the definition (c.f. Page 11), we know that the cost of every operator is positive. Denote the lowest such cost with δ! It can be easily seen that during reclassifying a vertex to open, its g-value decreases at least with δ. It's also obvious that every vertex's g-value has a lower limit: the optimal cost of getting to the vertex (from the initial vertex). All of these facts imply the truth of the statement.

Afterwards, let's examine the features of the A algorithm! By the previous statement, the following facts can be told about the completeness of the A algorithm:

Completeness:

- If there is a solution, the method finds it in *any* state-space graph.
- If there is no solution, the method recognizes this fact in the case of a finite state-space graph.

But what about the *optimality* of the generated solution? We wonder if the A algorithm really guarantees the generation of an optimal solution? A counterexample in Figure 18 shows that it does not. On the left side of the figure, a state-space graph can be seen, in which we put the heuristic values of the vertices aside them: for example, 5 for the initial vertex s, or 0 for the goal vertex c (absolutely correctly, since all goal vertices should have a heuristic value of 0). We put the cost of the edges (as operators) on the edges. In the figure, the optimal solution is depicted bold.

On the right side of the figure, we followed the operation of the A algorithm in this state-space graph step by step. We put their g-values and h-values (cost and heuristics) aside every vertex. In the 2nd step, it can be clearly seen that the method expands the open vertex a, since its f-value is $2+3=5$, which is lower than the other vertex's f-value of $3+4=7$.

The search finishes in the 3rd step, as the method selects the vertex c with its f-value of $6+0=6$, and it is a goal vertex. Notice that, by this way, the method has found a solution with a cost of 6, which is not an optimal solution!

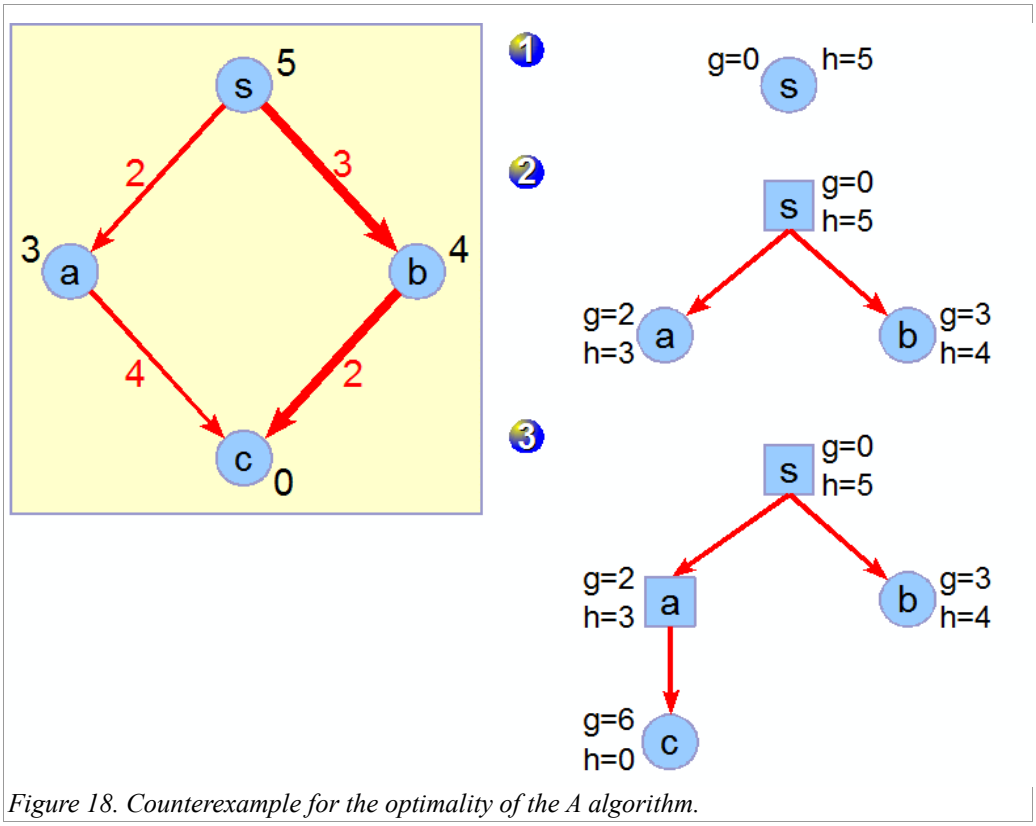


Figure 18. Counterexample for the optimality of the A algorithm.

So the following facts can be told about the optimality (and testing) of the A algorithm:

- **Optimality:** generating the optimal solution is *not guaranteed*.
- **Testing:** can be performed sooner, since the optimal solution is not guaranteed.

IMPLEMENTATION QUESTIONS

- **How to select the vertex with the lowest total cost?**
  Similarly to the uniform-cost method, the *costs are stored* within the vertices, and their f-values originate in these values. The list of open vertices is sorted by f-value.
- **How to handle the problem of closed vertices?**

  It's worth to select the 2nd one from the options described on Page 43. When inserting a vertex into the database with a lowest cost than the existing one, it's worth deleting the old (closed) vertex, and when it's done, the new (open) vertex can be inserted.

EXAMPLE

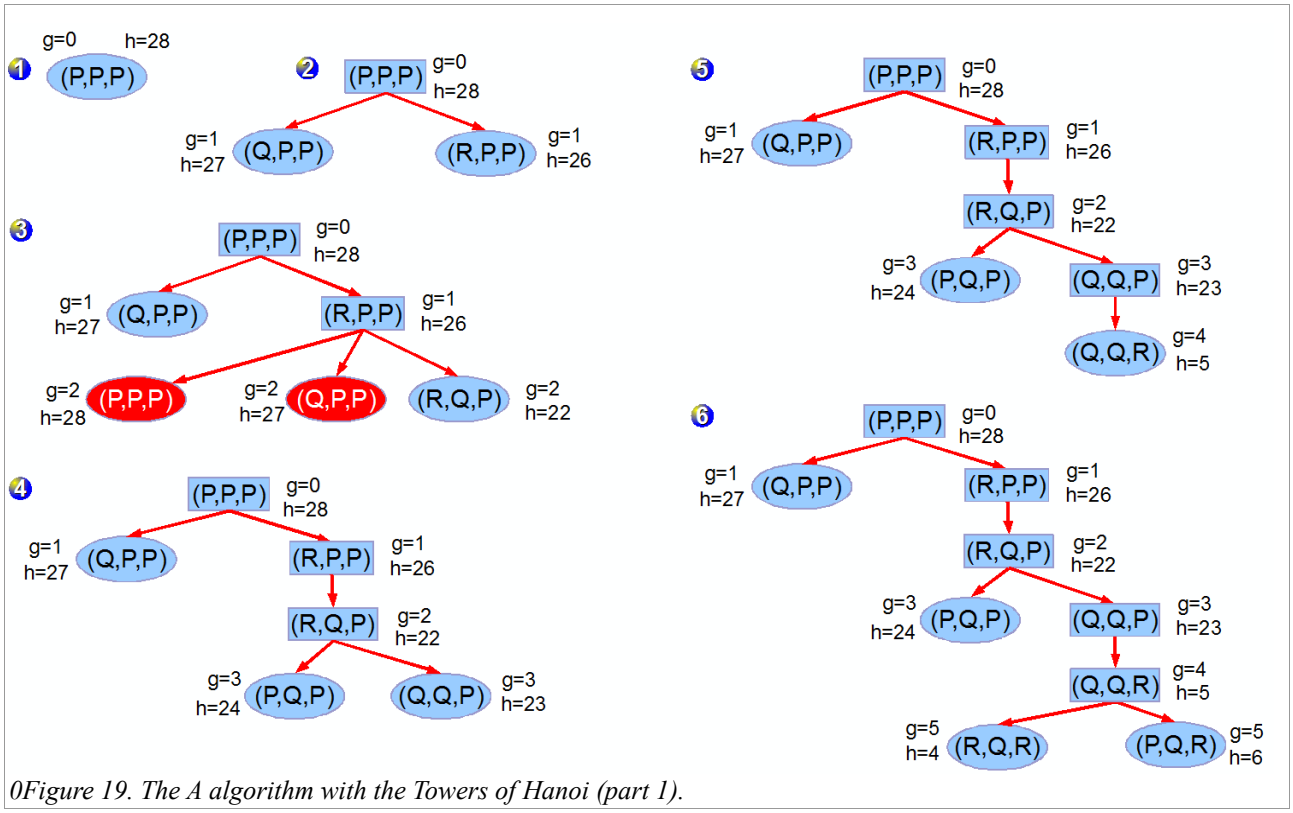
In the Figures 19 and 20, a search tree generated by the A algorithm for the Towers of Hanoi problem can be seen, step by step, until finding a solution. We have decided to use the following heuristics:

$$h(a_1, a_2, a_3) = 28 - index(a_1) - 4 \cdot index(a_2) - 9 \cdot index(a_3)$$

where

$$index(a_i) = \begin{cases} 0 & \text{,if } a_i = P \\ 1 & \text{,if } a_i = Q \\ 2 & \text{,if } a_i = R \end{cases}$$

So, we assign an ordinal number to the rods: 0 to P, 1 to Q, and 2 to R. It can be seen that the bigger discs are weighted more in the heuristics, as the goal is to move the bigger discs to the rod R. It can also be seen that the subtraction from 28 happens to achieve the heuristic value of 0 in the goal state (R, R, R). Let us note that, in case of the Towers of Hanoi state-space representation, it would have been more practical to mark the discs with numeric values (0,1, and 2) from the beginning.



0Figure 19. The A algorithm with the Towers of Hanoi (part 1).

In the 3rd step, we indicated two such vertices which the A algorithm does not add to the database by default, since their f-values are not lower than the f-value of the ones with the same content already in the database. During the further steps, we do not show the other such vertices that are excluded for the same reason.

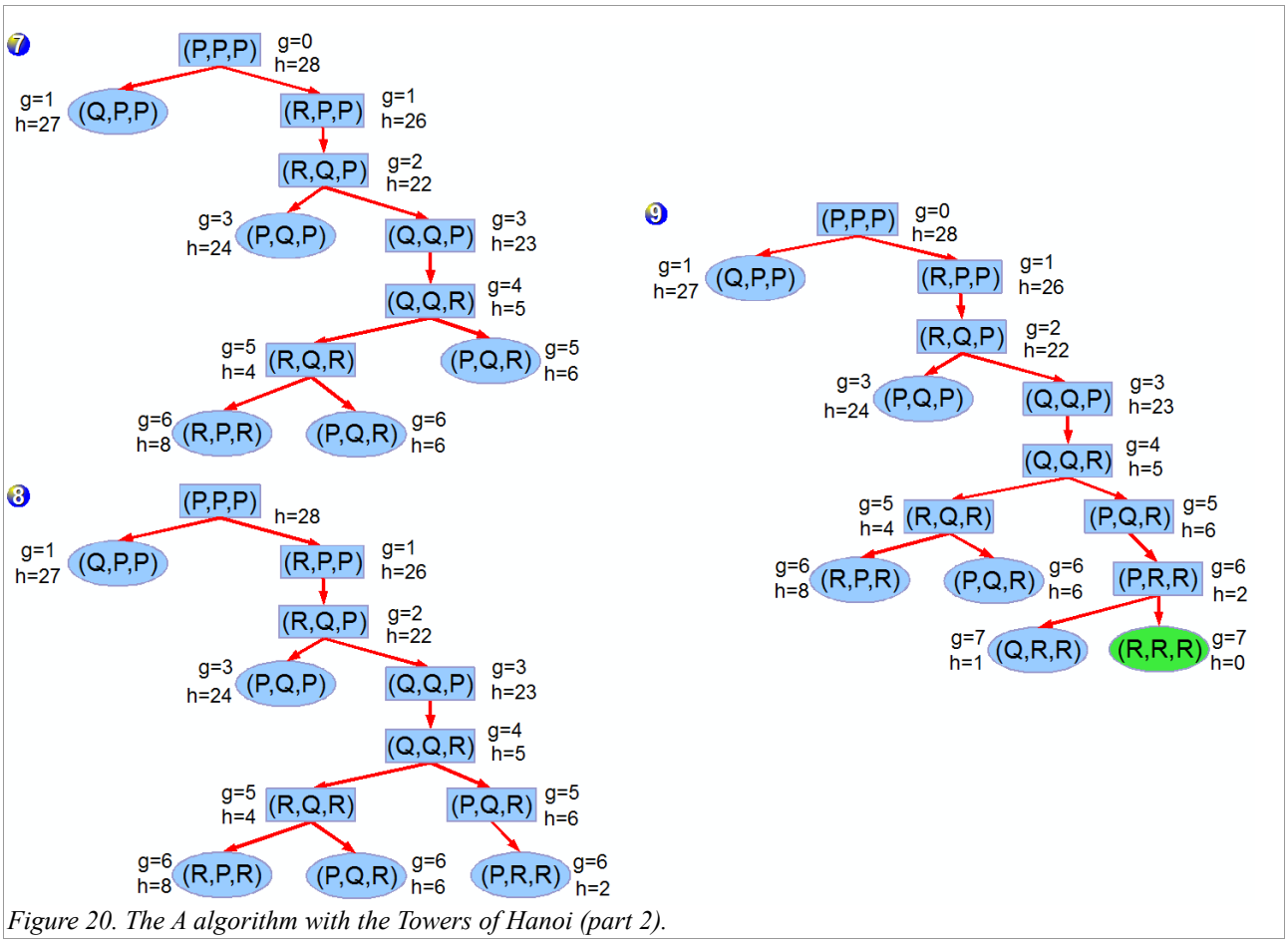


Figure 20. The A algorithm with the Towers of Hanoi (part 2).

The following conclusions can be drawn:

- The method founds an optimal solution. The question is if it happened by accident, or the chosen heuristics forced the method to do so? C.f. the next chapter!
- The problem of closed vertices did not occur during the search. Namely, there wasn't any case when a newly created vertex was found in the database as a closed vertex with a higher f-value.
- The use of heuristics made the search very insinuating. We went for a non-optimal direction only once during the search: in the 6th step, when the vertex (R, Q, R) was being expanded. But one step later, the method returned to the optimal path.

4.3.3.3 The A* algorithm

The A* algorithm is a variant of the A algorithm that *guarantees the generation of an optimal solution*. We surely know that the uniform-cost method is such a method, and it is obvious that this advantageous feature is due to the constant zero heuristics. The question is: what kind of (but not a constant zero) heuristics could guarantee the generation of an optimal solution?

To describe this accurately, we have to introduce the following notations for every vertex n of the search tree:

- $h^*(n)$: the *optimal* cost of getting to a goal vertex from n .
- $g^*(n)$: the *optimal* cost of getting to n from the initial vertex.
- $f^*(n)=g^*(n)+h^*(n)$: Consequently, the optimal cost of getting to a goal vertex from the initial vertex via n .

Definition 11. A heuristics h is an *admissible heuristics* if the following condition holds for every state a :

$$h(a) \leq h^*(a) .$$

The *A* algorithm* is an A algorithm with an *admissible heuristics.* In the followings, we prove that the A* algorithm guarantees the generation of an optimal solution. For this, we will also use two lemmas.

Lemma 1. If there is a solution, the following condition holds for the A* algorithm in any given time: *The optimal solution has an element among the open vertices*.

Proof. This is a proof by complete induction. Denote the vertices occurs in the *optimal solution* as follows:

$$(s=)\, n_1 \;,\; n_2 \;,\; \ldots \;,\; n_r (=c)$$

- Before the 1st expansion, the initial vertex s is open.
- **Basis:** Assume that before an expansion, the optimal solution has an element among the open vertices, and the one with the smallest index of these is n_i.
- **Inductive step:** Let's see the situation before the next expansion! There are two options:
  - If we haven't expanded n_i right now, then n_i remains open.
  - If we have expanded n_i right now, then n_{i+1} has been inserted to the database as open.

Lemma 2. For every vertex n that was expanded by the A* algorithm:

$$f(n) \leq f^*(s) .$$

Proof: According to the previous lemma, the optimal solution always has an element among the open vertices. Denote the first such vertex with n_i. Since n_i is stored in the database on an optimal path:

$$g(n_i) = g^*(n_i)$$

Denote the vertex that is selected for expansion with n ! Since the A* algorithm always selects the vertex with the lowest f-value for expansion, we know that $f(n)$ is not higher than the f-value of any open vertices, so in case of $f(n_i)$:

$$f(n) \leq f(n_i)$$

Consequently,

$$f(n) \leq f(n_i) \;=\; \underbrace{g(n_i)}_{=g^*(n_i)} + \underbrace{h(n_i)}_{\leq h^*(n_i)} \;\leq\; g^*(n_i) + h^*(n_i) \;=\; f^*(n_i) \;=\; f^*(s)$$

The fact $h(n_i) \leq h^*(n_i)$ originates in the *admissible* heuristics. The fact $f^*(n_i) = f^*(s)$ arises from the knowledge, that n_i is an element of the optimal solution, so the optimal cost of the solutions going via n_i actually the cost of the optimal solution, namely $f^*(s)$.

Theorem 1. The *A* algorithm* guarantees the generation of an optimal solution.

Proof: At the moment of the solution being generated, the algorithm selects the goal vertex c to be expanded. According to the previous lemma:

$$f(c) \leq f^*(s)$$

By considering the fact that c is a *goal vertex*:

$$f(c) = g(c) + \underbrace{h(c)}_{=0} \;=\; g(c) \;\leq\; f^*(s)$$

Thus, in the database, the cost of the path from the initial vertex to c (denoted by $g(c)$) is not greater than the cost of the optimal solution. Obviously, $g(c)$ can't be less than the cost of the optimal solution, thus:

$$g(c)=f^*(s)\,.$$

Let us summarize the features of the A* algorithm!

- **Completeness:**
  - If there is a solution, the method finds it in any state-space graph.
  - If there is no solution, the method recognizes this fact in the case of a finite state-space graph.
- **Optimality:** generating the optimal solution is *guaranteed*.
- **Testing:** can't be performed sooner.

4.3.3.4 The monotone A algorithm

In the previous chapter we examined, what heuristics the A algorithm needs to guarantee the generation of an optimal solution. Now, we examine what heuristics we need to avoid *the problem of closed vertices*. The uniform-cost method (as a special A algorithm) avoids this problem with its constant zero heuristics, but what can we say in general about the heuristics of the A algorithm?

Definition 12. A heuristics h is a monotone heuristics, if the following condition holds for every state a: if we get the state a' by applying an operator o to a, then

$$h(a)\ \le\ h(a')+cost_o(a)\,.$$

The A algorithm with monotone heuristics is called the *monotone A algorithm,* and it can be proved that the problem of closed vertices does not occur with this algorithm. This is the consequence of the following theorem:

Theorem 2. For any vertex n that was selected by the A algorithm for expanding:

$$g(n)=g^*(n)\,.$$

Proof: This is an indirect proof.

Assumption: $g(n)>g^*(n)$. Namely, we assume that the algorithm hasn't found the optimal path from the initial vertex to n.

Demote the vertices *on the optimal path* from the inital vertex to n as follows:

$$(s=)n_1\ ,\ n_2\ ,\ \dots\ ,\ n_r(=n)$$

Let n_i be *the first open vertex* among these vertices. The following fact can be noted:

$$f(n_i)=g(n_i)+h(n_i)\ =\ g^*(n_i)+h(n_i)$$

Here we exploited that n_i is already on the optimal path in the database; namely, $g(n_i)=g^*(n_i)$. Let us continue with the aforementioned formula!

$$f(n_i)=g(n_i)+h(n_i)\ =\ g^*(n_i)+h(n_i)\ \le$$
$$\le\ g^*(n_i)+h(n_{i+1})+cost_{o_i}(n_i)\ =\ g^*(n_{i+1})+h(n_{i+1})$$

On the one hand, we exploited that the heuristics is monotone, thus, $h(n_i)\le h(n_{i+1})+cost_{o_i}(n_i)$. On the other hand, we exploited the fact $g^*(n_{i+1})=g^*(n_i)+cost_{o_i}(n_i)$. Let us continue with the aforementioned inequality in a similar way!

$$f(n_i)=g(n_i)+h(n_i)\ =\ g^*(n_i)+h(n_i)\ \le$$
$$\le\ g^*(n_i)+h(n_{i+1})+cost_{o_i}(n_i)\ =\ g^*(n_{i+1})+h(n_{i+1})\ \le$$
$$\le\ g^*(n_{i+1})+h(n_{i+2})+cost_{o_{i+1}}(n_{i+1})\ =\ g^*(n_{i+2})+h(n_{i+2})\ \le$$
$$\vdots$$
$$\le\ g^*(n_r)+h(n_r)$$

So, where are we now?

$$f(n_i) \leq g^*(n)+h(n) < g(n)+h(n)=f(n)$$

Here we used our indirect assumption, namely: $g^*(n)<g(n)$. Note that this leads to a *contradiction*, since we have got that the f-value of n_i is lower than the f-value of n, thus, should have expanded n_i instead of n!

The consequence of the previous two theorems is the following: in the monotone A algorithm a cycle detection technique that is looking for the new vertex *only among the open vertices* is absolutely sufficient. Accordingly, the following facts can be told about the completeness of the monotone A algorithm:

Completeness:

- If there is a solution, the method finds it in any state-space graph.
- If there is no solution, the method recognizes this fact in the case of a finite state-space graph.

When summarizing the features of the monotone A algorithm, another main question is whether the algorithm provides an *optimal solution*? Fortunately, it can be seen that *the monotone A algorithm is a special A* algorithm*, so the answer to the previous question is „yes". We prove this fact with the help of the next theorem.

Theorem 3. If a heuristics is monotone then it's admissible, too.

Proof: It must be proved for any vertex n that the following fact holds in the case of the monotone heuristics h:

$$h(n) \leq h^*(n)$$

There are two options. The first one is that no goal vertices can be reached from n. In this case $h^*(n)$ equals to ∞ by definition, thus, the relation we want to prove obviously holds.

The other option is that a goal vertex can be reached from n. Take the optimal path among the paths leading from n to any goal vertex, and denote its vertices as follows:

$$(n=)n_1 \ , \ n_2 \ , \ \dots \ , \ n_r(=c)$$

By the monotonicity of h, the following relation can be drawn:

$$\begin{aligned} h(n_1) &\leq h(n_2)+cost_{o_1}(n_1) \\ h(n_2) &\leq h(n_3)+cost_{o_2}(n_2) \\ &\vdots \\ h(n_{r-1}) &\leq h(n_r)+cost_{o_{r-1}}(n_{r-1}) \end{aligned}$$

Sum the left and right sides of these inequalities! We get the following fact:

$$\sum_{i=1}^{r-1} h(n_i) \leq \sum_{i=2}^{r} h(n_i) + \sum_{i=1}^{r-1} cost_{o_i}(n_i)$$

If we subtract the first sum of the right side from both sides:

$$h(n_1)-h(n_r) \leq \sum_{i=1}^{r-1} cost_{o_i}(n_i) = h^*(n)$$

Here, we exploited that $h^*(n)$ is actually the cost of the path (which is optimal!) leading to c (namely to n_r). Moreover, we even know that $h(n_r)=0$, since n_r is a goal vertex. Consequently:

$$h(n) \leq h^*(n)$$

Thus, the following facts can be told about the optimality and testing of the monotone A algorithm:

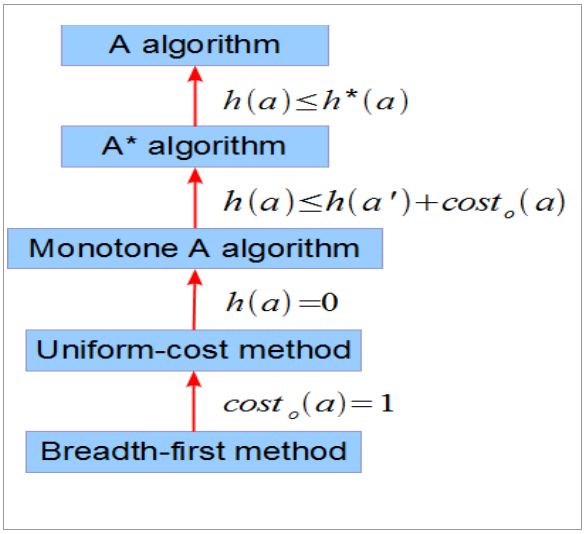
- **Optimality:** generating the optimal solution is *guaranteed*.
- **Testing:** can't be performed sooner.

4.3.3.5 The connection among the different variants of the A algorithm

We have got to know the following facts:

(1) The *breadth-first method* is a uniform-cost method, where the cost of the operators are one unit, equally.
(2) The *uniform-cost method* is an A algorithm, where the heuristics is constant zero.
(3) The *A* algorithm* is an A algorithm, where the heuristics is admissible. The uniform-cost method is an A* algorithm, too, since the constant zero heuristics is also admissible.
(4) The *monotone Aalgorithm* is an A algorithm, where the heuristics is monotone. The *uniform-cost method* is a monotone A algorithm, too, since the constant zero heuristics is monotone.

According to these facts, the appropriate relations among the different variants of the A algorithm can be seen in Figure 21. As going top-down in the figure, we get more and more specialized algorithms and more and more narrow algorithm classes.



5 2-PLAYER GAMES

Games – sometimes to a frightening amount – amuses the human intellect since the beginning of civilization. Probably it comes from the aspect that a game can be understood as an idealized world in which the opposing players compete with each other and face it, competition is present at every field of life.

The first successes of artificial intelligence researches can be connected to games. Naturally, those games are challenging to research, in which the players (either human or machine) have verifiable influence on the outcome of the game. These games are called *strategy games*, like chess, draughts or poker. There were gaming machines before the appearance of computer science, such as the Spanish Quevedo's chess machine, which was specialized on the endgame of chess (the machine had king and rook, while the human player had king) and was able to give a checkmate in any situation (if the machine had the first step). The Nimitron, which was built in 1940 and was able to win the Nim game any time, had similar qualities.

All these initial attempts remained isolated till the middle of the 1940s, when the first programmable computers were developed. Neumann and Morgenstern's „Theory of Games and Economic Behavior", which is a basic work in this topic, came out in 1944 and gives an exhaustive analysis of game strategies. Even in this book, the minimax algorithm plays and articular role and it will become one of the basic algorithm in computer game programs.

The computer program, that was able to play through a full game of chess was created by Alan Turing in 1951. Actually, Turing's program never run on a computer, it was tested with hand simulation against a quite weak human opponent who defeated the program. The chess is a game with huge state-space, this is the cause of the difficulties of creating a chess program. As time has passed, the minimax algorithm was improved (in order to reduce the state-space) along certain aspects – all of these, the *alpha-beta purning*, which was worked out by McCarthy in 1956, deserve more attention.

When we try to transplant a game to computer, we have to give the following information in some way:

- the possible states of the game
- the number of players
- the regular moves
- the initial state
- when will the game end and who will win (and how much)
- what information do the players have during the game
- if randomness has any part in the game

According to the above criterion, we can categorize the games as follows:

(1) By the number of players:
   - *2-player games*
   - *3-player games*
   - etc.

(2) By the role of randomness
   - *Deterministic games:* randomness plays no role
   - *Stochastic games:* randomness plays role

(3) By finiteness
   - *Finite games:* all players have finite moves and the game will end after finite moves.

(4) By the amount of information:
   - *Full information games:* The players have all the information that arise during the

game. Card games are good counterexamples, as these have masked cards.

(5) By profit and loss:

- *Zero-sum games*: the total of the players' profit and loss is zero.

In the following, we will deal with 2-player, finite, deterministic, full information, zero-sum games.

5.1 State-space representation

We can use the state-space representation to represent games. The state-space representation of a game is formally the same as in Chapter 3.1. There are only a few differences in the content:

(1) The forms of *states* are (a, p), where a is a state of the game and p is the player who is next. The players will be marked with `A` and `B`, so $p \in \{\mathtt{A}, \mathtt{B}\}$.

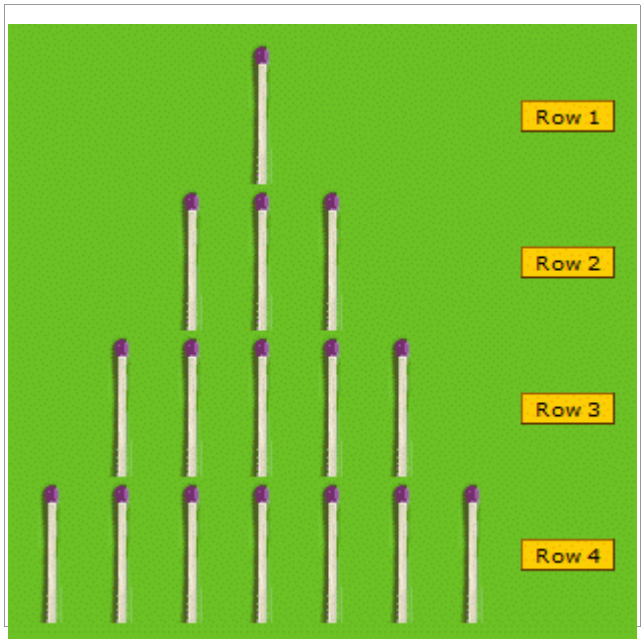
(2) In every goal state, it must be given exactly that who wins or loses in that given state or the game is a drawn.

(3) Every operator o (invariably) has a precondition of the operator and a function of applying. The function of applying shows if we apply o to a state (a, p), than what state (a', p') will we get. So it is important to define p', which is the player who will move after player p!

5.2 Examples

5.2.1 Nim

There are match-sticks in n pieces of mounds. The player with the next move can take any amount of matches (at least one and no more than the number of matches in the mound) from a chosen mound. The players take turns. Whoever takes the last match-stick loses the game.



- **Set of states:** In the states, we store the number of matches in a mound. Let the state be an n-tuple, where $n > 0$ is a constant out of the state-space, denoting the number of the mounds. A number $max > 0$ must be also given in advance, which maximizes the number of

matches in the mound.
Naturally, the state must store the mark of the player with the next move.
So the set of states is the following:

$$A=\{(a_1,a_2,\ldots,a_n,p) \mid \forall\, i \;\; 0\le a_i\le max \;\;,\;\; p\in\{\mathtt{A},\mathtt{B}\}\}$$

where every a_i is an integer.

- **Initial state:** The initial state can be any state, so the only clause is the following:

$$k\in A$$

- **Set of goal states:** The game ends when all of the mounds are empty. The player who takes the last match loses the game, so the player with the next move wins. Namely:

$$C=\{(0,\ldots,0,p)\}\subseteq A \;\;,\;\; p \text{ wins}$$

- **Set of operators:** The operators remove some matches from the mound. So the set of our operators is:

$$O=\{remove_{mound,pcs} \mid 1\le mound\le n \;\;,\;\; 1\le pcs\le max\}$$

- **Precondition of the operators:** Formalize when an operator $remove_{mound,pcs}$ can be applied to a state $(a_1,\ldots,a_n,p)$! We need to formalize the following conditions:
  - The $mound$ th mound is not empty.
  - The value of pcs is actually the number of matches the $mound$ th mound has.

  So, the precondition of the operator $remove_{mound,pcs}$ to the state $(a_1,\ldots,a_n,p)$ is:

$$a_{mound}>0 \;\wedge\; pcs\le a_{mound}$$

- **Function of applying:** Let's formalize what state $(a'_1,\ldots,a'_n,p')$ the operator $remove_{mound,pcs}$ generates from the state $(a_1,\ldots,a_n,p)$! Namely:
  $remove_{mound,pcs}(a_1,\ldots,a_n,p)=(a'_1,\ldots,a'_n,p')$ where

$$a'_i=\begin{cases} a_i-pcs & ,\text{if } i=mound \\ a_i & ,\text{otherwise}\end{cases} \qquad \text{where } 1\le i\le n$$

$$p'=\begin{cases}\mathtt{A} & ,\text{if } p=\mathtt{B}\\ \mathtt{B} & ,\text{if } p=\mathtt{A}\end{cases}$$

NOTES

It's more worthwhile to denote the players with numeric values, where the most common solution is using the values 1 and -1. On one hand, this is quite useful in the function of applying of the operator, e.g., the previous definition of p' can be as easily described as:

$$p'=-p$$

On the other hand, this notation of the players will come handy in the *negamax algorithm* (Chapter 5.5).

5.2.2 TIC-TAC-TOE

We know this game as a 3x3 Gomoku (Five-in-a-row). On the 3x3 game board, the players put their own marks in turns. The winner is who puts 3 of his or her marks in a row, in a column, or in a diagonal line. The outcome of the game can be a drawn if the game board is full but no one has 3 marks abreast.

- **Set of states:** In the states we need to store the whole 3x3 game boards, and of course the mark of the player with the next move. One cell of the 3x3 matrix is 0 if none of the players has put a mark there.
  So the set of states is defined as follows:

$$A=\{(T_{(3\times3)},p)\ \mid\ \forall i,j\ \ T_{i,j}\in\{0,\mathtt{A},\mathtt{B}\}\ \ ,\ \ p\in\{\mathtt{A},\mathtt{B}\}\}\ .$$

- **Initial state:** In the initial state the game board is completely empty, and player $\mathtt{A}$ is going to put his or her mark first!

$$k=\left(\begin{pmatrix}0&0&0\\0&0&0\\0&0&0\end{pmatrix},\mathtt{A}\right)$$

- **Set of goal states:** The game may end because of two reasons:
  - One of the players has 3 marks abreast.
  - The game board is full.

  Namely: $C=C_1\cup C_2$, where

$$C_1=\left\{(T,p)\ \middle|\ \begin{array}{c}\exists i\ \ T_{i,1}=T_{i,2}=T_{i,3}=p'\\ \vee\\ \exists i\ \ T_{1,i}=T_{2,i}=T_{3,i}=p'\\ \vee\\ T_{1,1}=T_{2,2}=T_{3,3}=p'\\ \vee\\ T_{1,3}=T_{2,2}=T_{3,1}=p'\end{array}\right\}\ ,p'\ \text{ wins}$$

  (see the definition of p' in the operators' function of applying)

  and

$$C_2=\{(T,p)\notin C_1\ \mid\ \neg\exists i,j\ \ T_{i,j}=0\}\ \text{, drawn}$$

  In the set C_1 , we specify the winning goal states. As it was the opponent of p last time, it's sufficient to look for triplets of the marks of p' on the board. In the condition, that consists of four rows, we specify (in order) the triplets in one row, in one column, in the main diagonal , and in the side-diagonal.

  In the set C_2 , we give the drawn goal states. The game ends in a drawn if the board is full (and none of the players has 3 marks abreast).

- **Set of operators:** Our operators put the mark of the current player into one cell of the board. So, the set of our operators is defined as follows:

$$O=\{put_{x,y}\ \mid\ 1\le x,y\le3\}$$

- **Precondition of the operators:** Let's formalize when can we apply an operator $put_{x,y}$ to a state (T,p) ! Naturally, when the cell (x,y) of T is empty.
  So, the precondition of the operator $put_{x,y}$ to a state (T,p) is:

$$T_{x,y}=0$$

- **Function of applying:** We have to formalize what state (T',p') the operator $put_{x,y}$ generates from a state (T,p) ! So:

  $put_{x,y}(T,p)=(T',p')$ where

$$T'_{i,j}=\begin{cases}p & \text{, if } i=x\ \wedge\ j=y\\ T_{i,j} & \text{, otherwise}\end{cases}\qquad \text{where } 1\le i,j\le3$$

$$p'=\begin{cases}\mathtt{A} & \text{, if } p=\mathtt{B}\\ \mathtt{B} & \text{, if } p=\mathtt{A}\end{cases}$$

5.3 GAME TREE AND STRATEGY

According to the state-space representation of the 2-player games, using the method given in

Chapter 3.2, we can create a *state-space graph*. As we told it earlier, we only deal with *finite games*, so this graph must be finite, too. This also indicates that the graph may not contain cycles. On the other hand, cycles are usually eliminated by the game's rules. By unfolding the state-space graph into a tree, we get the so-called *game tree.*[2] One *play* of the game is represented by one branch of the tree.

For a game program to be able to decide the next move of a player, it needs a *strategy*. The strategy is actually a kind of instruction: which operator should the computer apply to a given state? It would be an interesting question whether does any of the players have any *winning strategy,* in the case of a given game? Namely, such a strategy in which the operators are applied following the instructions will always lead to the given player's victory (no matter what moves the opponent takes).

For this, we need to investigate how a given strategy (in connection with either player $\mathtt{A}$ or $\mathtt{B}$) appears in the game tree. If we want to represent the possible strategies of the player $p \in \{\mathtt{A}, \mathtt{B}\}$, then we transform the game tree into an *AND/OR-graph* in the following way: with an arc we conjoin all the edges that start from a vertex in which *the opponent of* p moves. In Figure 23, we can see a AND/OR-graph from the point of view of player $\mathtt{A}$. We have conjoined the edges starting from a vertex in which player $\mathtt{B}$ moves. We call the resulting conjoined edges an *AND-edge group,* which represent the fact that player p has no influence on the opponent's moves, thus, the strategy of p must be prepared for any of the opponent's move. The edges highlighted with red in the figure could be the moves of a strategy of $\mathtt{A}$, since they unambiguously define the moves of $\mathtt{A}$ during the game.

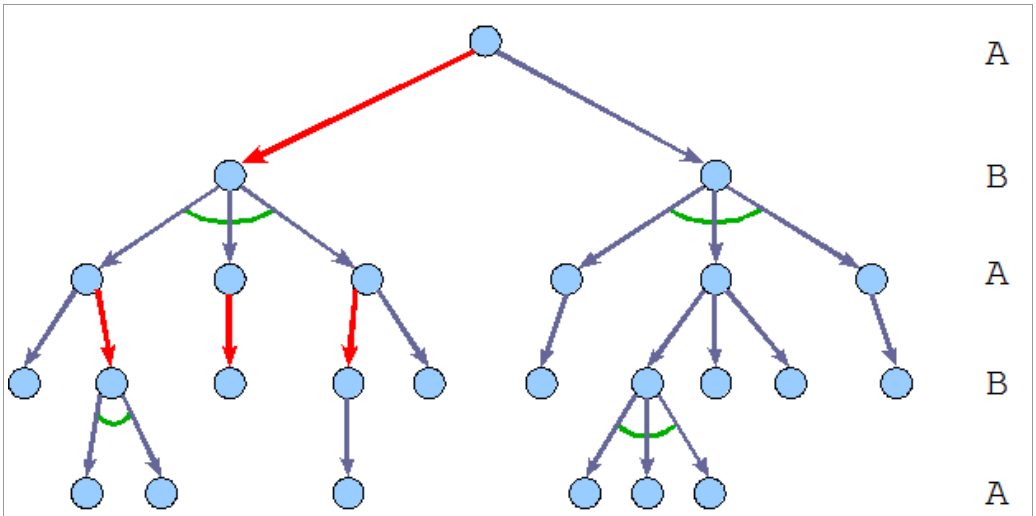


Figure 23. The game tree as a AND/OR-graph.

Now, let us formalize, in an exact way, what we mean by and AND/OR-graph and a strategy!

Definition 13. An AND/OR-graph is a ordered pair $\langle V, E \rangle$ where

- $V \neq \emptyset$ is the set of vertices, and
- E is the set of the hyperedges, where

$$E \subseteq \{(n, M) \in V \times P(V) \mid M \neq \emptyset\}$$

An $(n, M) \in E$ hyperedge can be of two kinds:

- an *OR-edge*, if $|M| = 1$.

2 Unfolding into a tree actually means the elimination of the multiple paths. So, the individual copies of the vertices (and the subtrees starting from them) should be inserted into the tree.

- *an AND-edge group*, if $|M|>1$.

The *hyperedge* is the generalization of the edge concept as we know. A hyperedge can be drawn from one vertex to any number of vertices; that's why we define M as the set of vertices[3]. The usual edge concept is equivalent to the concept of an OR-edge, since it is a special hyperedge that leads from one vertex to exactly one vertex.

We call the generalization of the corresponding path concept *hyperpath*: a hyperpath leading from vertex n to the set M of vertices is a sequence of the hyperedges between n and M.

After all of the above, it is easy to formalize the *strategy* of player p : a hyperpath leading from the initial vertex to a set $M \subseteq C$ of vertices (namely, each element of the M is a goal vertex) in an AND/OR-graph from the point of view of p . In Figure 24 and Figure 25, a possible strategy for player `A` and player `B` can be seen, respectively, framed in the tree.

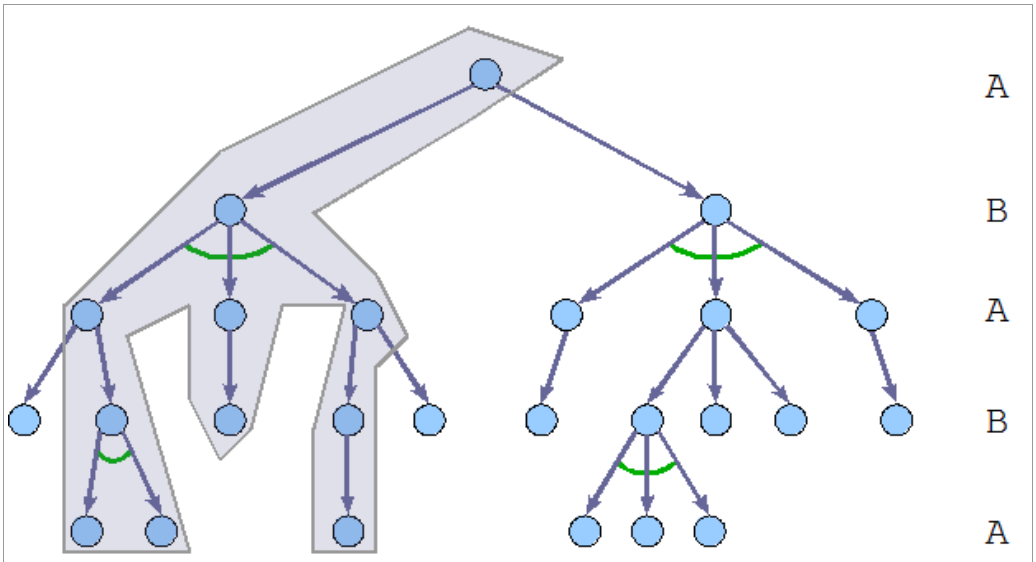
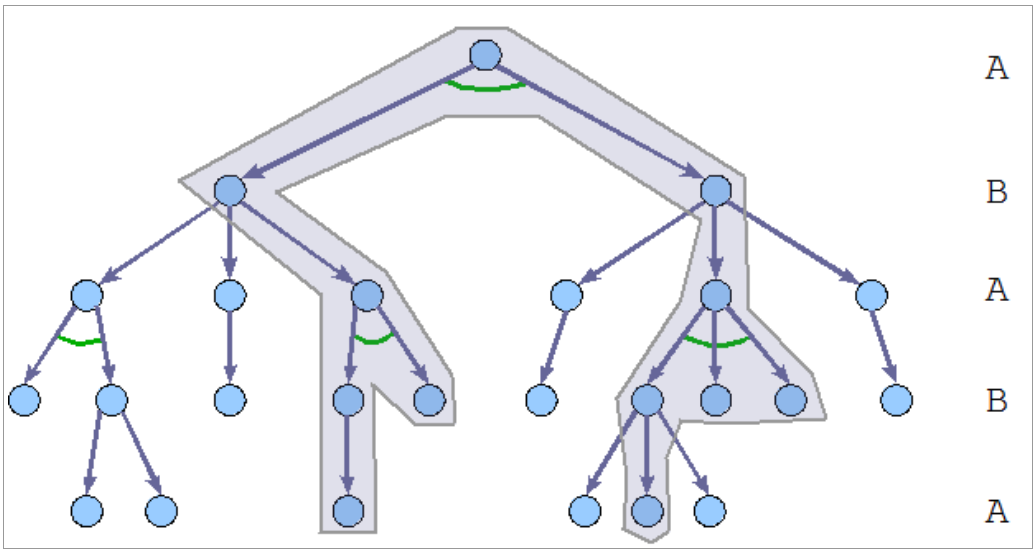


Figure 24. A possible strategy for player `A`.

3 $P(V)$ denotes the powerset of the set V (namely, the set of all the subsets of V).



5.3.1 Winning strategy

One of the seducing goals in connection with a game is to look for winning strategies. Does one such thing exist? If it does, then does it for which player?

First, we need to clarify what we call a *winning strategy* of player p : such a strategy of p which leads (as a hyperpath) to such goal states in which p wins.

Theorem 4. In every game (examined by us) that cannot be a drawn, one of the players has a *winning strategy.*

Proof: We generate the game tree, and then we label its leaves with `A` and `B`, depending on who wins in the given vertex.

After this, we label the vertices in the tree bottom-up, in the following way: if p moves in the given vertex, then

- it gets the label p if it has a child labelled with p ;
- it gets the label of the opponent, if it doesn't have a child labelled with p (so, every children of the vertex is labelled with the opponent's mark).

After labelling all the vertices, the label of the initial vertex (i.e., the root) shows the player which has a winning strategy.

In games which can end in a drawn, a *non-losing strategy* can be guaranteed for one of the players.

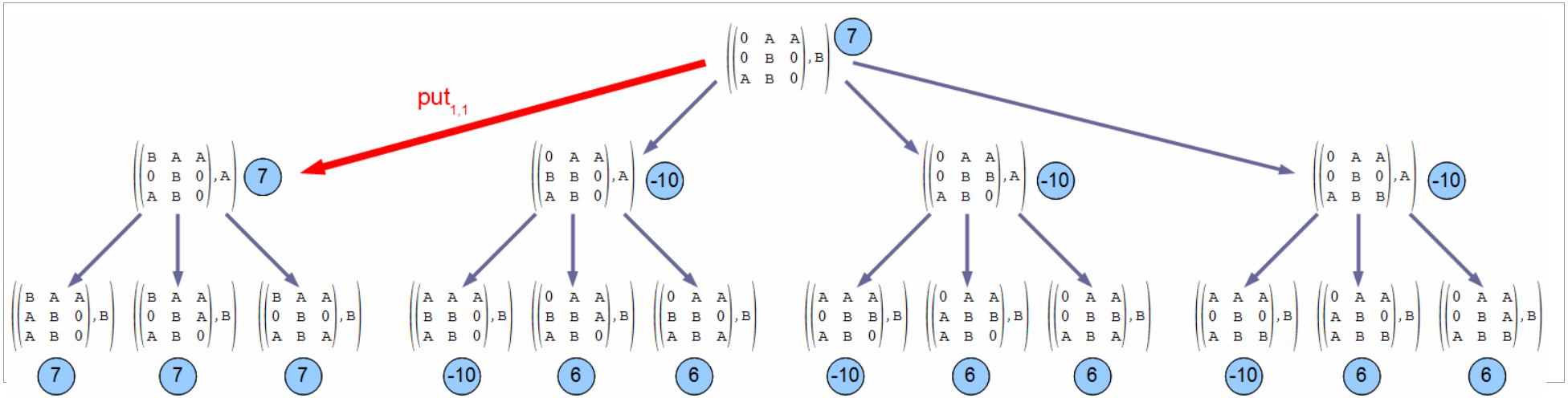
5.4 The Minimax algorithm

Since a game tree can be extremely huge, it is an irrational wish for our game program to wholly generate it (and look for a winning strategy in it). The most we can do is that before every move of the computer player, we generate *a part* of the game tree. This tree has the following properties:

- its root is the current state of the game, and
- its depths is limited by value *maxDepths* , that has been given in advance.

Generating the tree this way is the same as the common „trick” applied in reality, when the players think ahead for a few moves, playing all the possible move combinations in their heads. The value *maxDepths* specifies for how many moves the program should think ahead. It's worth asking for the value of *maxDepths* as a *difficulty level* at the start of our game program.

In Figure 26, we built up an appropriate subtree for the Tic-tac-toe game, starting from the state $\left(\begin{pmatrix} 0 & \mathtt{A} & \mathtt{A} \\ 0 & \mathtt{B} & 0 \\ \mathtt{A} & \mathtt{B} & 0 \end{pmatrix}, \mathtt{B}\right)$, and for $maxDepths = 2$.



The leaves of the tree built in this way can be of two sorts. Some of them are *goal vertices,* and we have clear concepts about their usefulness: if the given player wins in the leaf, then it's a „very good" goal state for him; if he loses, then it's a „very bad" one.
The other leaves are *not goal vertices*; we haven't expanded them because of the depth limit (most of the leaves in Figure 26 are such leaves). That's why we are trying to define the „goodness" of leaves with some kind of estimation; so, we need a heuristics.

HEURISTICS

How to specify a chosen heuristics? The definition of heuristics introduced in Page 24 must be reconsidered in connection with 2-player games:

- Heuristic values must be *arbitrary integer* values. The better it is for the given player, the greater (positive) the value is, and the worse it is for the player, the lower (negative) the value is.
- In case of a goal state, the heuristic value should be 0 only if the goal state results in a drawn. Memo: in Page 24, we assigned 0 to every goal states. In the case of 2-player games, if the given player *wins* in the goal state, the heuristic value should be a very great value, while if the player *loses,* it should be a very small value.
- Since there are two players, we need two heuristics. $h_{\mathtt{A}}$ is the heuristics for player $\mathtt{A}$, while $h_{\mathtt{B}}$ is the heuristics for player $\mathtt{B}$.

In a given game, we define the heuristics. For example, in the case of Tic-tac-toe, let the heuristics h_p (where $p \in \{\mathtt{A}, \mathtt{B}\}$) be defined as follows:

(1) 10 , if we are in a goal state, where p wins.
(2) -10 , if we are in a goal state, where the opponent of p wins.
(3) 0 , if the goal state is a drawn.
(4) The number of all rows, columns, and diagonals that p „blocks" (his mark occurs there).

THE FUNCTIONING OF THE MINIMAX ALGORITHM

Denote the player in the root of the tree (who has the current move) with $aktP$! Let us assign a *weight* to all of the vertices (a, p) of the tree, by the following way:

(1) If (a, p) is a leaf, then let its weight be $h_{aktP}(a, p)$.

(2) If (a, p) is not a leaf, it means that it has some children. The weights assigned of its children are used to calculate the weight of their parent. This is done as follows:

- If $p = aktP$, then the *maximum* of the children's weights is assigned to (a, p) .
- If $p \neq aktP$, then the *minimum* of the children's weights is assigned to (a, p) .

The explanation of calculating maximum/minimum is obvious: $aktP$ is always about to move towards the most advantageous state, that is, the state with the highest heuristic value. The opponent of $aktP$ is about to do quite the opposite.

The values besides the vertices in Figure 26 are the weights of the vertices, according to the heuristics of player $aktP = \mathrm{B}$, who takes his move in the root.

After assigning a weight to each vertex in the tree, comes the step why we have arranged all these before. We determine *which operator* is worth applying in the root of the tree (that is, to the current state of the game). Namely: the game program should use (or suggest, in the case of the human player's move) the operator that (as an edge) leads *from the root of the tree to the child with the highest weight*. In Figure 26, the operator $put_{1,1}$ is being applied (or advised) by the game program.

5.5 THE NEGAMAX ALGORITHM

The difficulty of using the Minimax algorithm is that *two kinds of heuristics must be defined for a game*. It's easy to see that there is a strong relation between the two heuristics. This connection can be described with simple words, as follows: the better a state of the game is for one player, the worse it is for the other player. So, the heuristics for one player can be simply the *opposite* (negated) of the heuristics for the other player.

Consequently, only one heuristic is absolutely sufficient. Let the heuristic value of the state (a, p) be calculate by the *heuristics for* p .

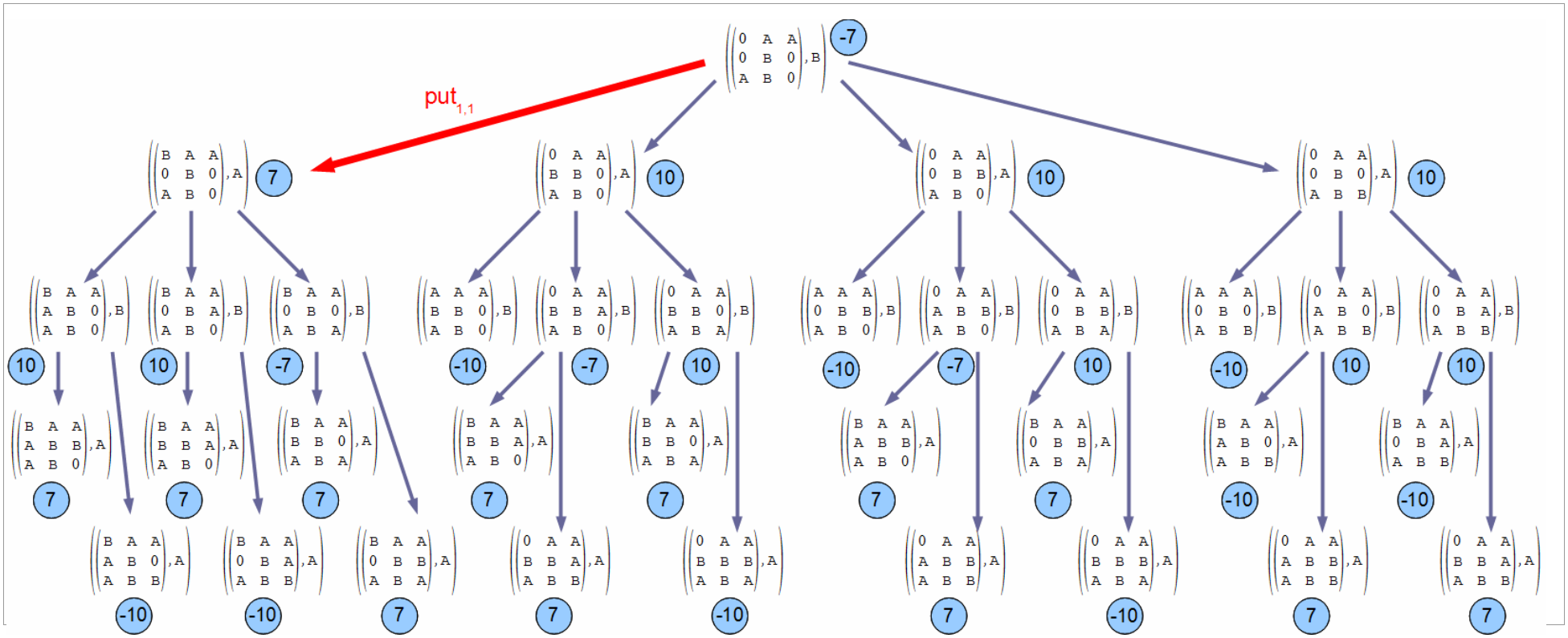
Starting on this track, we can make the minimax algorithm simpler at two points:

(1) When assigning heuristic values to the leaf vertices of the generated tree, we don't need to take into account which player moves in the root of the tree.

(2) When calculating the weight of a non-leaf vertex (a, p), we calculate *maximum,* in all cases. The trick is to find a way to transform the weight of the child (a', p') of a vertex (a, p), as follows:

- If $p \neq p'$, then we use the negated value of the weight of (a', p') to determine the weight of (a, p).
- If $p = p'$, than we use the (unchanged) weight of (a', p') to determine the weight of (a, p).

In Figure 27, it can be seen the tree of the Tic-tac-toe game, generated by the Negamax algorithm, this time for $maxDepth = 3$. It is worth contrasting with Figure 26. In these figures, it can also be seen that player `B` , who is the currently moving, makes a clever move as he or she is controlling the game to the only one direction where he or she has any chance of winning. So, we've chosen the heuristics quite well.



5.6 THE ALPHA-BETA PRUNING

If we choose a high value for *maxDepth* , then the generated tree (and also, the time-demand of the minimax algorithm) will be very large. So, it is worth optimizing the algorithm, based on the observation that it is often unnecessary to generate certain parts of the tree. Eliminating these parts is the main goal of the *alpha-beta pruning*.

With the alpha-beta pruning, we can *interrupt the expanding* of a certain vertex. We do this in cases when it turns out before completing the expansion that the weight of the vertex being currently expanded will have no effect on the weight of it's parent. Let's see two possible situations:

(1) **Alpha cut-off**: We are calculating minimum in the current vertex, and maximum in it's parent. We interrupt the expansing of the current vertex, if the minimum value calculated in the vertex becomes less then the maximum value calculated in it's parent. We can see such a situation in Figure 28: the vertex being expanded bears the caption ≤ 4, since we have already created a child with a weight of 4, hence, the weight of the current vertex will be not greater than 4. For similar reasons, its parent's weight will be not less than 9, so we don't need to generate all the other children of the current vertex, since the weight of the current vertex is irrelevant from this point on.

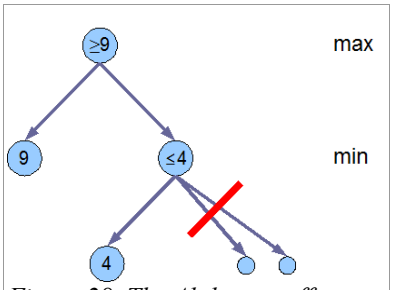


Figure 28. The Alpha cut-off.

(2) **Beta cut-off**: It is the same as the alpha cut-off, but in this case, we calculate maximum in the vertex being currently expanded, and calculate minimum in it's parent. We can see such a situation in Figure 29.

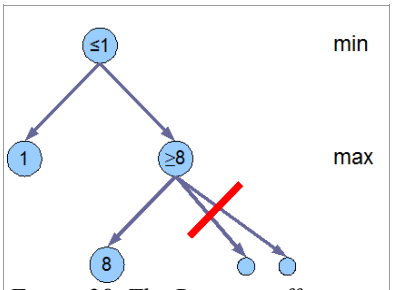
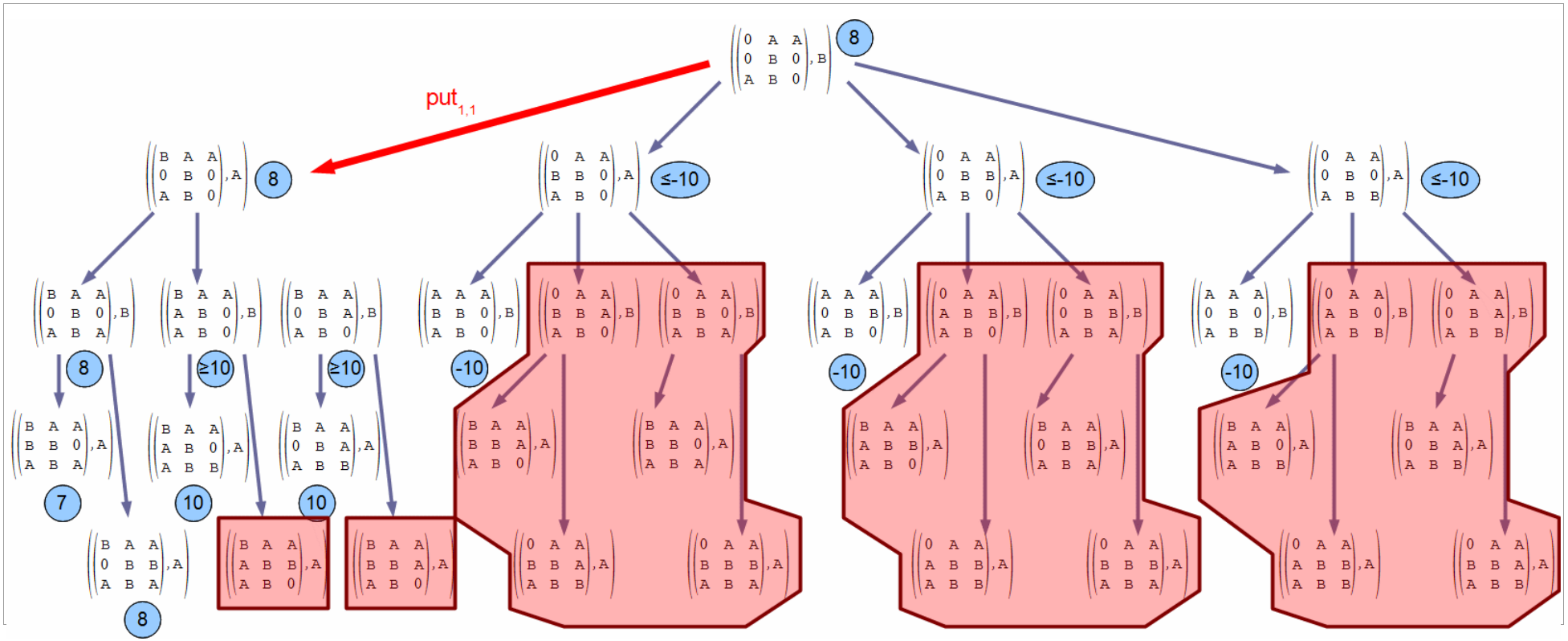


Figure 29. The Beta cut-off.

In Figure 30, a tree generated by the Minimax algorithm for Tic-tac-toe can be seen, for $maxDepth = 3$. The parts of the tree that the algorithm doesn't need to create due to the Alpha-beta pruning are in a red frame. Count them, and it will turn out that we need to generate 20 vertices less, and only need to generate 15 vertices. We managed to cut 57% of the tree.


put1,1
8
≤-10
≤-10
≤-10
8
≥10
≥10
-10
-10
-10
7
10
10
8

6 USING ARTIFICIAL INTELLIGENCE IN EDUCATION

The main topic of artificial intelligence consists of the solution searcher algorithms, that search for the path from the starting vertex to the goal vertex in a graph. It is difficult to represent these algorithms on a (white)board, as the state of a vertex is constantly changing. For example, in case of the depth-first search, a vertex can be unexplored, open or closed. As there is no space on the board to draw the graph many times, we can only represent what we want by re-drawing the status of the vertex in the same graph. By this way, the audience easily loses the time-line sequence. We are looking for a solution for this problem among others.

One suggested solution is to use animated figures, where the algorithm and the graph can be seen. On the right side of the figure, the execution of the algorithm can be seen step-by-step. The actual line is coloured. If any command changes the state of the vertex, than the graph's appropriate vertex changes on the left of the figure. Usually it turns to another colour. Of course, every colour's meaning is listed.

According to our experiences, the use of this aid helps and deepens the understanding of the searcher algorithms.

We noticed that the students are afraid of the searcher algorithm as they find them to complicated, that exceed their abilities. So it's very important to make these algorithms tangible and bring them close. One great method for this is the visualization of the algorithm, that's why our digital notes includes several animations.

6.1 THE PROBLEM

The problem has two levels, one that is psychological and another that is technical.

When realizing the searcher algorithms, the students first face the problem that they need to use their programming knowledge in a complex way. First, they need to create a state-space representation along with the attached data structures (usually with the help of a State and Vertex class), than they need to select the appropriate algorithm which is recursive in many cases.

As the length of the solution is not determinable beforehand, it's storing requires a dynamic data structure, which is a usually a list (or a stack, or maybe a queue). So to summarize, they need to routinely use:

- the classes,
- the recursion and
- the dynamic data structures.

These requirements scare off many students of the subject before they could see it's beauty. If the student feels that a subject is to complicated, it exceeds his or her abilities than he or she secludes himself of understanding it. This is the psychological problem.

The other problem is rather technical. Representing the main solution searcher algorithms on the board is uneasy, as the states of the vertices are changing with time. For example, in case of the depth-first search, a vertex can be unexplored, open or closed. As there is no space on the board to draw the graph many times, we can only represent what we want by re-drawing the status of the vertex in the same graph. By this way, the audience easily loses the time-line sequence. We find a solution to both problems if we call for the aid of multimedia.

Solution suggestion

Using multimedia in education is not a new thing. It's a widely accepted view that the materials thought in any subject is more easily learnt if we use multimedia aids during the teaching, such as:

- **colourful figures,**
- **videos, animations,**
- **interactive programs.**

The cause of this phenomenon is that multimedia bring closer and makes tangible anything that was said before in theory.

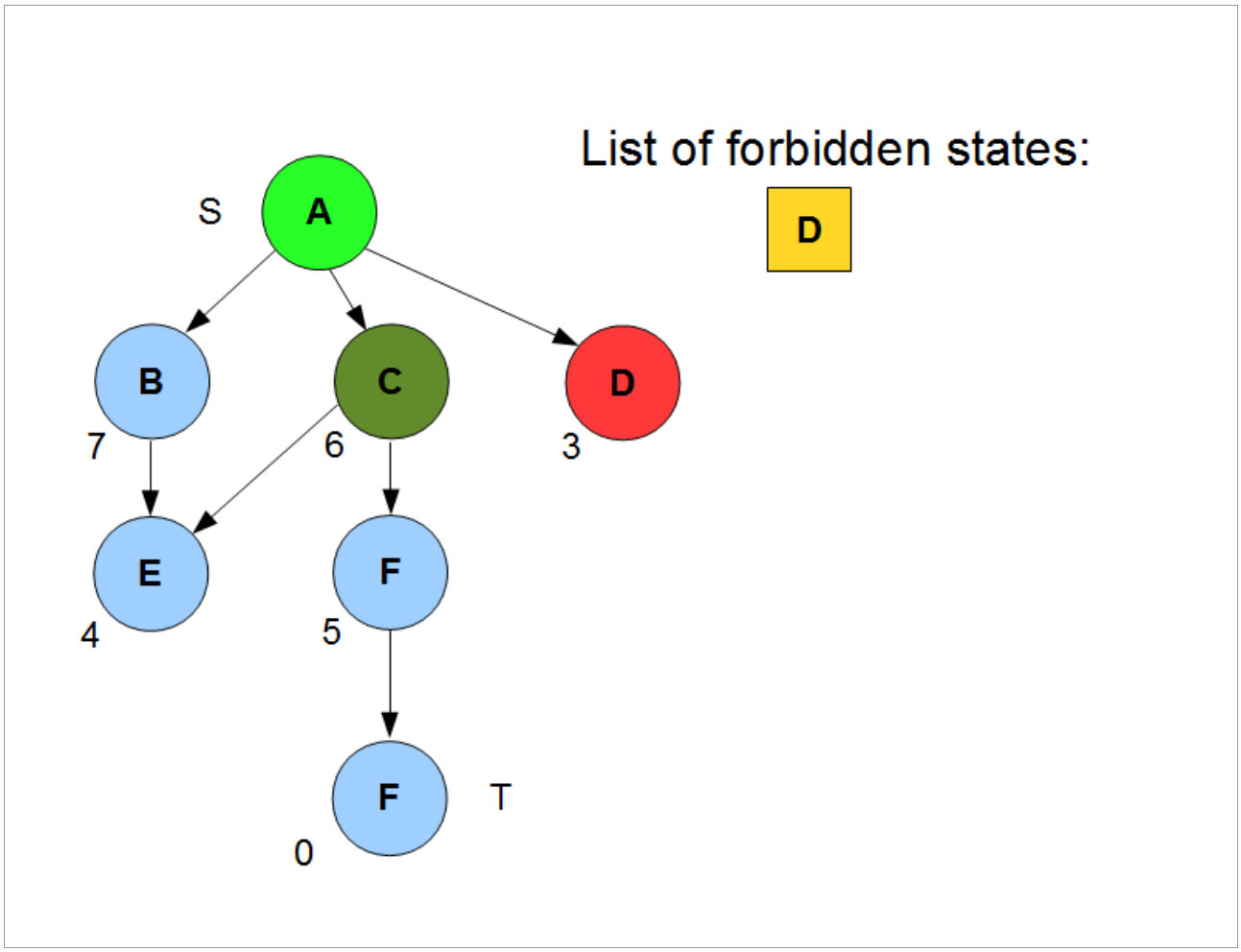
At the same time, using multimedia has drawbacks too. It's usage needs such tools that are not always given in a lecture room, they may break down, their starting takes time and such problems.

The practical part of teaching artificial intelligence usually takes place in a computer laboratory, so the conditions of multimedia are usually given.

The representable information changes in the cases of the different solution searcher algorithms, so it's required to define for each of them what information should appear in the animation, as part of the graph or completing it and what kind of data structure is used as a database to store the graph. In the following, we will discuss the differences between the representation of the different searcher types.

6.1.1 Non-modifiable searchers

We use the non-modifiable searchers in such special cases when our goal is not to produce a sequence of operators that lead to the solution, but we want to know if there is a solution for the problem, and if the answer is yes, than we have to produce this goal state. One of the most well-know and most customizable type is the Mountaineer method. This is a heuristics searcher, which means that the algorithm selects on operator for a given state by an estimation given by us. This is very fortunate in the cases when we are able to give an effective heuristics. In an opposing case, we can use the more simple Trials and Errors method which selects operators randomly. In the notes, we demonstrate the build-up of the animation on the Mountaineer with restart method. In this version we record the list of states that we couldn't use to continue the search and we have to give the maximum number of members of this list. When we are unable to continue the search, the current vertex is inserted into the list of forbidden vertices and the search restarts.



On the figure above we can see the searcher in a state when we have inserted the D vertex to the list of forbidden vertices after a restart. The searcher's current, actual vertex is C. The next step will be done from this point and will select the operator leading to vertex E. We know this because the heuristics of the state in the E vertex is smaller. The heuristics of the different states are marked with the number by the vertex.

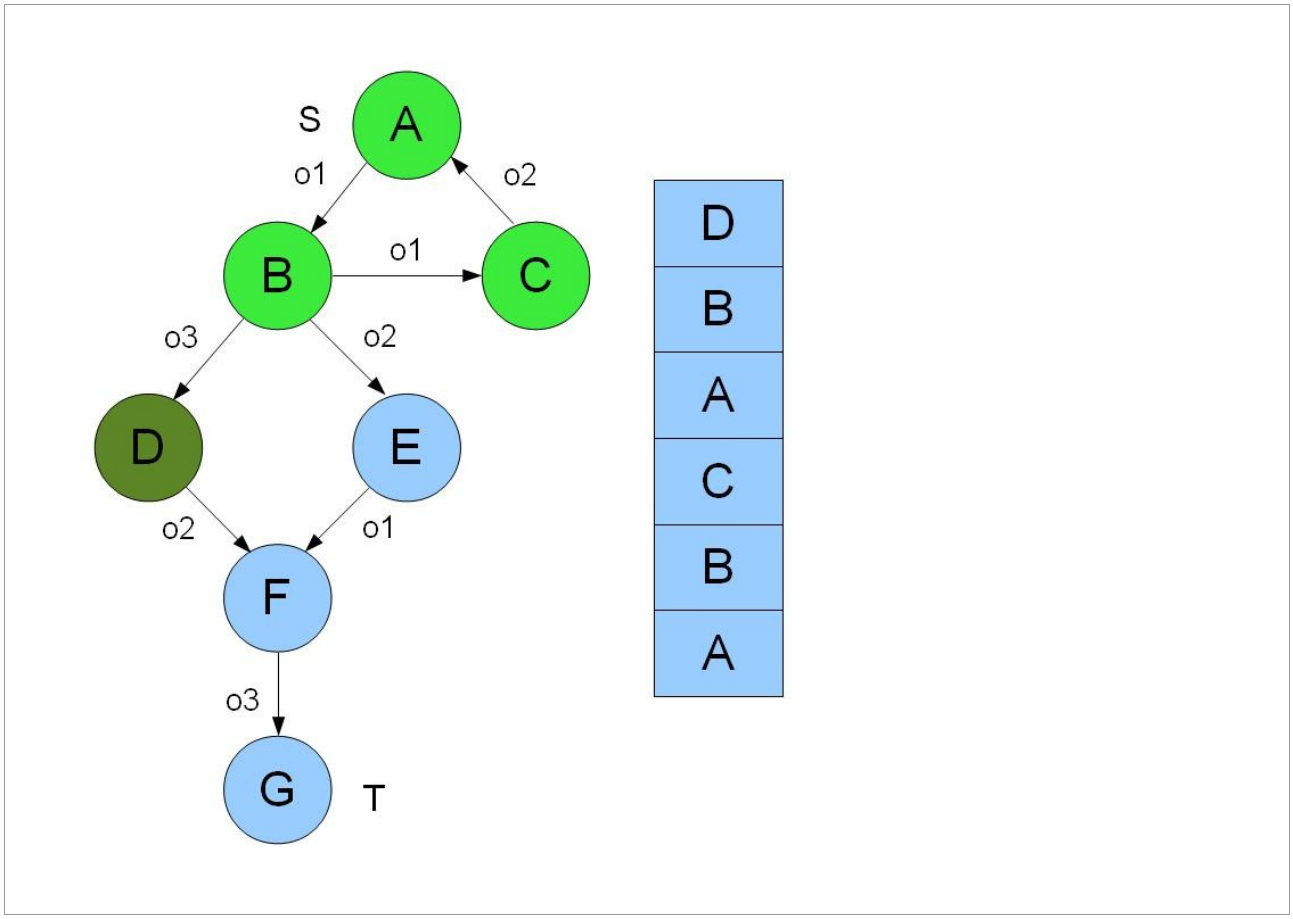
Legend:

Notions	Markings
Not yet visited vertex	blue
Current vertex	Deep green
Visited vertex	green
Forbidden vertex/state	red
Heuristics	Number by the vertex
Starting vertex	There's an S by the vertex
Terminal vertex	There's a T by the vertex
List of forbidden states	In the yellow list under the caption

We do not note the number of restarts separately, if it is necessary, we portray it when restarting.

6.1.2 Backtrack searchers

One type of the modifiable searchers is the backtrack searcher, where we expend our database outside the current vertex with the list vertices leading to the actual vertex. By this, we have the option to continue the search to another direction if the searcher runs into a dead end. This backstep gives the name of the searcher. If we find a terminal vertex, by the expanded database we have the option to give the sequence of operators that we use on the initial state to get the goal state. On the figure, you can see the backtrack searcher with length-limit. In this version, we work with a database of which size is given beforehand. We can use this if we can estimate the step number of the optimal solution well.



On the figure above we can see the searcher in a state when the database that is used to store the vertices gets full. The database on the figure gets filled with elements from the bottom to the top. This also represents well the operation of the stack data structure, that is usually used for realizing the database in case of the backtrack searchers. We can easily read from the figure that as the current vertex is the D and it is located under B, we will step back to vertex B in the next step.

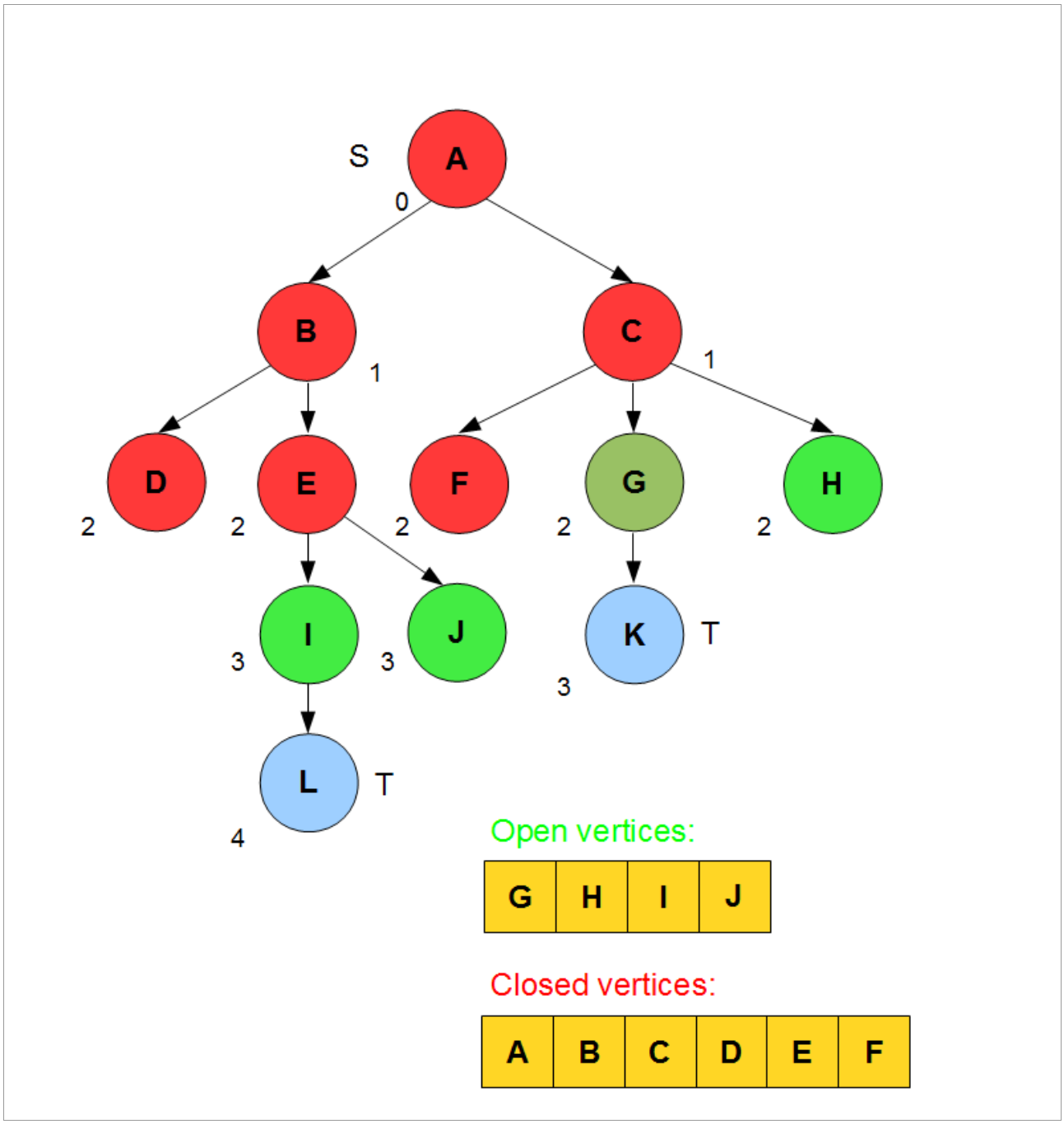
Legend:

Notions	Markings
Not yet visited vertex	blue
Current vertex	Deep green
Visited vertex	green
Operator, that we use to step forth	Number by the arrow

Starting vertex	There's an S by the vertex
Terminal vertex	There's a T by the vertex
The stack representing the database	The column right to the graph

6.1.3 Tree Search Methods

These make up the other great group of the modifiable solution searchers. By further extending the database we try to find the solution faster and if it's possible to find the solution with the least steps. While we stored one path in the case of the backtrack searchers, we store more paths at the same time in case of the tree search methods, that can be represented as a tree. We do the search on different paths in turns. The different algorithms differ in the method they use to select how to continue the search. Going forth in the tree is extending (or extension), by this we differentiate between open and closed vertices. The closed vertices are the ones we already extended, the others are considered open vertices. Another new notion is the depth number. A vertex's depth number is given by the steps we need to reach it from the starting vertex.



On the figure above we can see the depth-first method which selects the vertex with the lowest depth number for extending. If more vertices have the same depth, it selects the next vertex for extending randomly or by agreement, like from left to right. The searcher on the figure is currently at extending the G vertex. In the next step, the G vertex will be closed and the K vertex will appear in the list of open vertices and it's colour will change to green too.

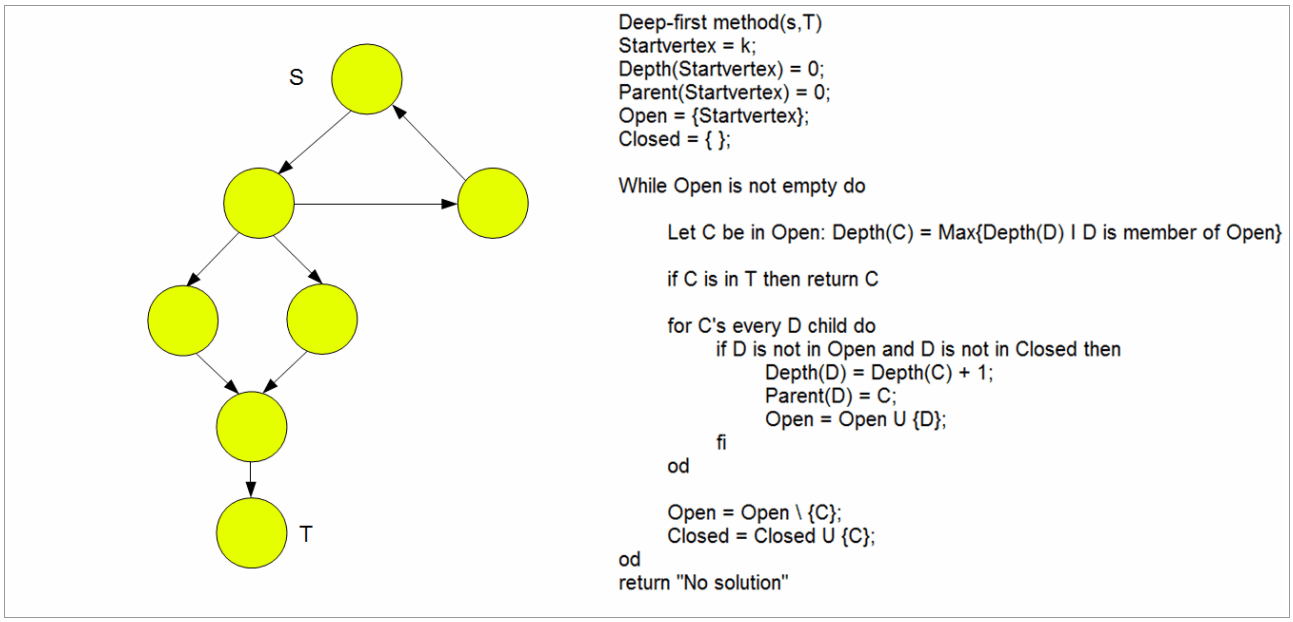
Legend:

Notions	Markings
Not yet visited vertex	blue
Vertex under extension	Deep green
Open vertex	green

Closed vertex	red
Depth number	Number by the vertex
Starting vertex	There's an S by the vertex
Terminal vertex	There's a T by the vertex
List of open/closed vertices	The list under the given caption

6.1.4 Depth-First Method

We show the depth-first method in short here.



It is visible that the figure has two main parts. On the left part of the animation we can see a graph in which we are searching for the solution. On the right side, we can examine the algorithm of the depth-first method itself. On the right side of the figure we can see as the algorithm is executed step by step. The actual line is red. If any command changes the state of a vertex, then on the left side of the figure, the appropriate vertex of the graph changes. The yellow vertices mark the not yet visited vertices, a the blacks are the closed ones and the red ones are the open vertices. The starting vertex is the uppermost vertex of the graph, the goal (or terminal) vertex is at the bottom. The numbers in the vertices show the depth number of the vertices.

The important fact, that the formation of the graph on the picture or in the animation is not entirely random also have to be mentioned, indeed, it was built that way willingly. It's goal is for the figure to show special cases that are usually problematic with some algorithm of artificial intelligence. Such as the diamond and the loop.

To understand the animation, we summarize the depth-first method in short. The depth-first method extends the vertex with the highest depth number (the number in the vertex) of the open vertices (the red ones). If there are more such vertices, it selects randomly. We put a ring around the vertex that is selected for extension. As the result of the extension, the not yet visited (yellow) children of the vertex will be open (red) and the extended vertex will be closed (black). The search ends if we are out of open vertices (than we have no solution), or if the last visited vertex is a goal vertex (in this case we have a solution).

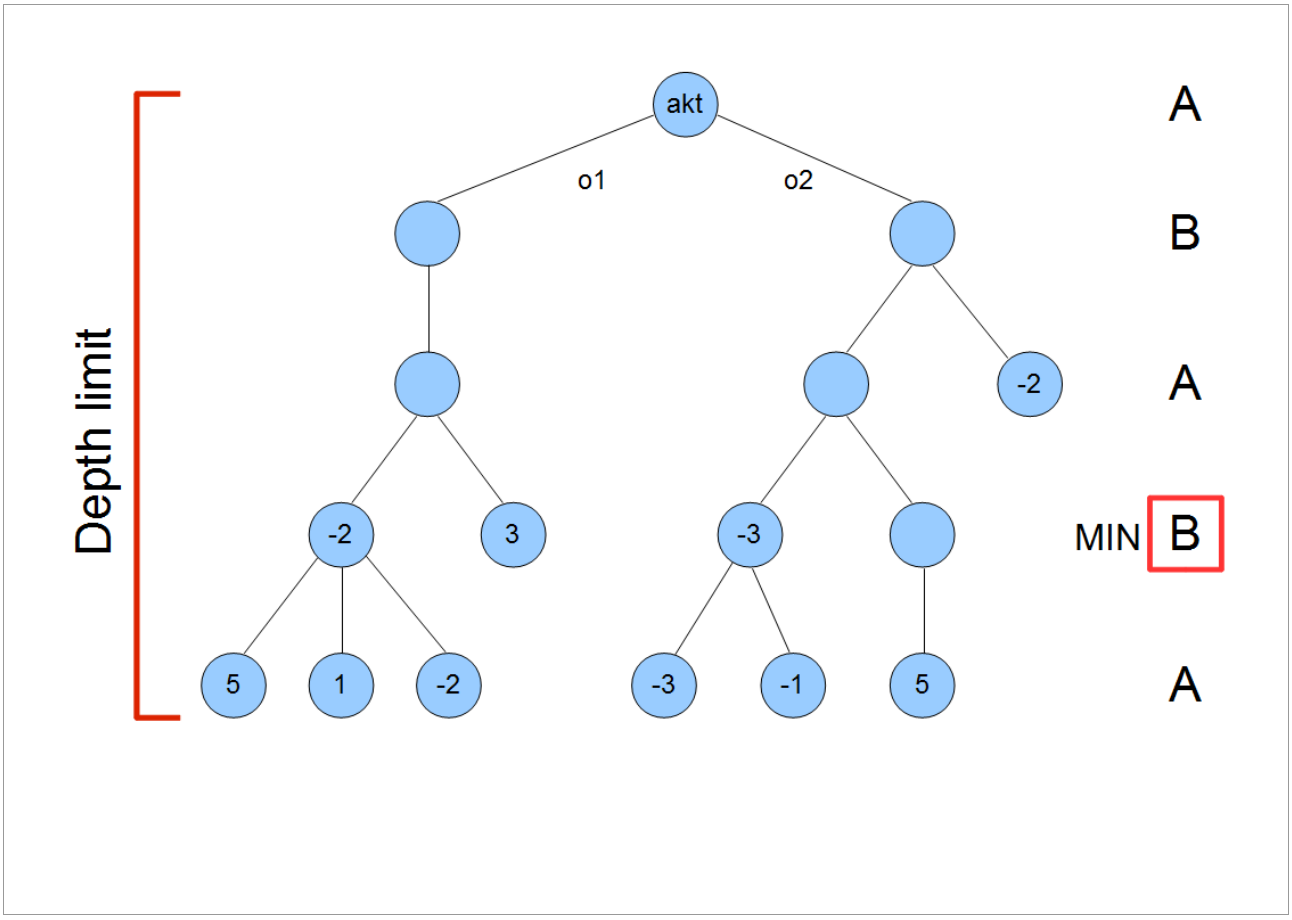
According to the above explanation, it's visible that such abstract notions, like open vertices can be

specified: red vertices. The table below pairs the notions and the markings of the animation:

Notions	Markings
Vertices not yet visited	yellow
open vertices	red
closed vertices	black
Depth number	the number in the vertex
vertex selected for extension	ringed vertex
extension/extending	the yellow off-springs turn red, the ringed ones turn black

6.1.5 2-Player game programs

2-Player games, like the chess, the draughts or the tic-tac-toe can be represented with a game tree. We can create this by using the game's state-space graph. In case of the 2-player games, each branch of the tree represents a different play of the game. To decide in the certain states that where to step next, namely which operator to use, we need to know which player can step in the turn. According to this, we can built up the And/Or graph, which helps with the path selection.



We can see the Minimax algorithm on the figure above. The algorithm works by generating the game tree from the current state for a given depth. When we are done with this, in the leaves of the tree, namely in the finally generated vertices of the branch, we try to give an estimation about how

good the given state is for us. The children elements' minimum or maximum is inserted to the vertex, heading for the current vertex up in the tree, depending on who has the next step, our opponent or we. That what do we need to select and when is noted on the right side of the animation, by the frame.

Legend:

Notions	Markings
Current vertex	Vertex with the "akt" caption
Selectable operators	Caption by the arrows
Depth limit	On the left ledge
Player	On the right, marked with captions "A" and "B"
Minimum/maximum selection	On the right, by the frame

6.2 Advantages and disadvantages

According to the experiences, with the use of the markings of the animation, the explanation is much more intuitive. The algorithm becomes more touchable and visible, so less will feel that it doesn't worth paying attention, as the subject is too difficult. Hence, we have found a solution for the psychological problem.

It is called psychological difference reduction when a problem that is too difficult to solve is traced back to more simple problems that can be solved. This is the explanation why the use of animations reduces the reluctance of the difficult subject – as it is easier to understand the animation.

The animation also solves a technical problem, as we don't need to use a board to draw a graph. Everyone can see the animation on his/her own computer or it can be displayed with a projector, and if the student was unable to follow it, it will restart automatically.

It is a disadvantage that we need a computer or a projector at least to see the animation. Without electricity, these devices don't work. To download the animation, internet connection is required.

The other disadvantageous phenomenon is that the audience may recall the concepts with their markings during the feedbacks. This is a greater problem, but if somebody reached the point to understand the algorithm, he or she doesn't need to put so much more energy to use the proper concepts. Moreover, during the feedbacks, the teacher can help in a mediate way with the markings, if the student gets stuck in his/her train of thoughts.

7 SUMMARY

As a summary, let's see a short heuristics, what can be done if we start to solve a problem with the techniques learnt here, but our program doesn't find the solution in time.

In our notes, we introduce the most well-known variations of the graph searcher algorithms, we showed it how can these be used to solve problems that are difficult in a way and a human can't solve them for the first time. At the same time, these algorithms are closer to systematic trials and errors, except for a few trivial optimizations, like the loop-checking.

Systematic trials and errors, is that all? Yes, that's all! We have to try many possibilities, and have to cover all the possible cases. This is systematicy. Of course, another question is how fast do we try these possibilities.

Till a certain level, it makes no difference what order do we try the operators, as the state-space is quite small and our fast computer can scan through fast. But we can easily run into a problem, which has a very large state-space.

In such cases, we can still try to be tricky by optimizing the operators or the algorithms themselves, for example by implementing the more effective relative of backtrack, the backjumping. It's even more effective if we use a lazy data structure. This means that we postpone everything we can of the algorithm's steps till we can. Such variants of the algorithms are called lazy algorithms and the data structures that support them are called lazy data structures.

This helps, but it's not the answer of artificial intelligence for the problem. The artificial intelligence's answer is the correct choice of the problem-representation. A problem can be represented with a very large state-space and with a quite small too.

How do we know that if the state-space is small or large? Generally, the less operators we have the smaller the state-space is, so we have to strive to have as few operators as possible.

Let's see a simple problem: In case of the 3 monks and 3 cannibals problem, one operator can be that when the boat brings some monks and cannibals to the other side of the river and a different operator can be when we sit in and get out. In the second case, the state-space will be much larger.

Unfortunately, after a difficulty level, not even creating a small state-space will help, as the solution of the problem will still takes many hours. This is when the second super-weapon comes into view: the use of heuristics.

What did we say? The graph search is only systematic tries and errors? Heuristics tell which operator worth trying. A good heuristics can hasten up the search by many degrees.

In the notes, we haven't dealt much with heuristics, only in case of the Minimax algorithm. This is by no means an accident. If the kind reader gets to write artificial intelligence for games that tells what the computer's next step should be, he will have an easy time except for writing the heuristics. This will be the hardest part and it's up to the heuristics how „smart” the game will be.

If the second super-weapon doesn't help either, than we still have a chance. We abandon systematicy and trust in tries and errors. Yes, the last super-weapon is the mountaineer – method, the mountaineer – method with restart. Have you ever thought?

On one hand, the mountaineer – method uses heuristics, on the other hand, it is really simple and in this manner it is quick. This algorithm is unable to realize if there is no solution, but in many cases, this is not important, the only thing that counts is to find the solution fast if it exists. If the heuristics is good, we have a good chance for this. If we don't find the solution fast, we give up the search after a given time.

We use the mountaineer – method with restart to solve one of the most significant artificial intelligence problem, the SAT problem. The version we use there is called Random Walk. We analyse the SAT problem is details in the Calculation Theories notes.

All kind readers can see that this short train of thoughts has just scratched the surface of artificial intelligence, but at the same time all that is written here gives a strong base for further studies.

8 Example programs

In this chapter we review some very cohesive classes. The classes materialize two algorithms, the backtrack searcher and the depth-first method. These classes embody the experience of a few years, we have tried and tested them many times, so they probably work. However, at some places the source must be altered if someone wants to use it for another graph search problem. Fortunately, this only needs an own State class, which have to originate from the AbstractState class. Other than this, neither the Vertex nor the BackTrack (and other) classes need rewriting, only the main program.

In the next part of the chapter, we review the different classes emphasizing what needs rewriting and what can stay in it's current form.

8.1 The AbstractState class

The AbstractState class is the ancestor of every State class. It is the same as the state-space representation. It orders that each and every State class have to contain a IsState function, that matches the State predicate, a IsGoalState function, that matches the goalstate predicate. The set of operators is given in a bit oakwardly way for the first sight. There is no such set, but we know how many member would it have. This is given by the NumberOfOps function. This must be extended by the children too. We don't need to specify the operators to, but we know that we can access them by the so-called super-operator. This is what the SuperOperator(int i) method is for.

The other methods don't need overwriting. The Clone method probably won't need rewriting in any children class. It would only need it if the inner fields of State would be arrays. In such case, the arrays need cloning too. We call this deep cloning.

The Equals method need rewriting if we want to use backtrack with loop-checking or depth-first method with loop-checking (with depth-first method, the loop-checking is the default). If we overwrite the Equals method than we should overwrite the GetHashCode method too. This is not obligatory, but the compiler will give ugly warnings instead.

8.1.1 Source code

```
/// <summary>
/// The ancestor of every state class.
/// </summary>
abstract class AbstractState : ICloneable
{
   // Checks if the inner state is a state.
   // If yes, then the result is true, else it's false.
   public abstract bool IsState();
   // Checks if the inner state is a goal state.
   // If yes, then the result is true, else it's false.
   public abstract bool IsGoalState();
   // Gives the number of the basic operators.
   public abstract int NumberOfOps();
   // The super operator is used to access all the operators.
   // Is true if the i th basic operator is executable for the inner state.
   // It must be called from a "for" cycle from 0 to the number of basic operators. For example:
   // for (int i = 0; i < state.GetNumberOfOps(); i++)
   // {
   //     AbstractState clone=(AbstractState)state.Clone();
```

```
    //      if (clone.SuperOperator(i))
    //      {
    //          Console.WriteLine("The {1}th for the {0} state" +
    //              "operator is executable", state, i);
    //      }
    // }
    public abstract bool SuperOperator(int i);
    // Clones. We need it because the effects of some operators must be withdrawn.
    // The most simple is to clone the state. That is what we call the operator for.
    // If there is a problem, we return to the original state.
    // If there is no problem, it will the clone be the state of which we continue the search.
     // This executes shallow cloning. If the cloning is swallow enough, than it does not need overwriting in the child class.
    // If deep cloning is required, than it must be overwritten.
    public virtual object Clone() { return MemberwiseClone(); }
    // Only needs overwriting if we use backtrack with memory or depth searcher.
    // Else, this may remain the basic implementation.
    // By programming technique, this is a hook method, that we can overwrite without violating the OCP.
    public override bool Equals(Object a) { return false; }
    // If the two instances are equal, than their hash codes are equal too.
    // So, if the Equals method is overwritten, this should be overwritten too.
    public override int GetHashCode() { return base.GetHashCode(); }
}
```

8.2 How to create my own operators?

We talk about the BlindState class in this chapter. This class has no function, only show in general how to create basic operators and operators with parameters and how to connect these in the super operator. We call basic operators the operators with no parameters or with fix parameters. These must be listed in a switch construct in the super operator. With an operator with parameters, more basic operators can be given with different values of the parameters.

From the example, it is visible that every operator has a true/false returning value. It is true, if it is executable on the current inner state of the State instance, else it is false. By programming technique, the operators can be done in many ways. Maybe the one selected here follows most the encapsulation theory of the OOP, which says that the inner state of the instance can only be modified by the methods of the instance. We know of artificial intelligence that these methods are called operators.

As the basic operators need to be accessed through the super operator, their visibility level can be private. Only the super operator needs to be public.

Every operator has a precondition. They also have a postcondition, but that is the same as the IsState predicate. A new operator have to start with calling it's precondition. The precondition must get the same parameters as the operator itself. If the precondition is false, than the operator is by no means executable. If it's true, than the state transitions must be executed. Than we have to check if we are out of the state-space with the IsState predicate. If this is true, so we've remained inside the state-space, then we are ready, the operator can be executed. Else it can't be executed and the state transition must be withdrawn too.

8.2.1 Source code

```
/// <summary>
/// The BlindState is only here for demonstration.
/// It shows how to write the operators and connect them to the super operator.
/// </summary>
abstract class BlindState : AbstractState
```

```
{
    // Here we need to give the fields that contain the inner state.
    // The operators execute an inner state transition.

    // First the basic operators must be written.
    // Every operator has a precondition.
    // Every operator's postcondition is the same as it's SateE predicate.
    // The operator returns a true, if it's executable and a false if it is not executable.
    // An operator is executable if it's true for the inner state,
    // for the preconditions and for the postcondition after state transition.
    // This is the first basic operator.
    private bool op1()
    {
        // If the precondition is false, the operator is not executable.
        if (!preOp1()) return false;
        // state transition
        //
        // TODO: We need to complete the code here!
        //
        // Checking the postcondition, if it is true, the operator is executable.
        if (IsState()) return true;
        // Else the inner state transition must be withdrawn,
        //
        // TODO: We need to complete the code here!
        //
        // and have to return that the operator is not executable.
        return false;
    }
    // The precondition of the first basic operator. The name of the precondition is usually: pre+name of the operator.
    // It helps understanding the code, but we can calmly differ from it.
    private bool preOp1()
    {
        // If the precondition is true, it returns as true.
        return true;
    }
    // Another operator.
    private bool op2()
    {
        if (!preOp2()) return false;
        // State transation:
        // TODO: We need to complete the code here!
        if (IsState()) return true;
        // Else the inner state transition must be withdrawn
        // TODO: We need to complete the code here!
        return false;
    }
    private bool preOp2()
    {
        // If the precondition is true, it returns as true.
        return true;
    }
    // Let's see what is the situation if the operator has parameters.
    // This time one operator equals more basic operators.
    private bool op3(int i)
    {
        // The precondition must be invited with the same parameters as the operator itself.
        if (!preOp3(i)) return false;
        // State transition:
        // TODO: We need to complete the code here!
        if (IsState()) return true;
        // Else the inner state transition must be withdrawn
        // TODO: We need to complete the code here!
        return false;
```

```
    }
    // The precondition's parameter list is the same as the operator's parameter list.
    private bool preOp3(int i)
    {
        // If the precondition is true than it return as true. The precondition depends on the parameters.
        return true;
    }
    // This is the super operator. The basic operators can be called through this one.
    // With parameter i, we say which operator we want to invite.
    // Usually we invite it from a "for" cycle, which runs from 0 to NumberOfOPs.
    public override bool SuperOperator(int i)
    {
        switch (i)
        {
            // We need to list all of the basic operators here.
            // We have to insert a new operator here.
            case 0: return op1();
            case 1: return op2();
            // The operators with parameters equal to more basic operators, so we have more lines for them here.
            // The number of lines depends on the task.
            case 3: return op3(0);
            case 4: return op3(1);
            case 5: return op3(2);
            default: return false;
        }
    }
    // Returns the number of basic operators.
    public override int NumberOfOps()
    {
        // we have to return the number of the last case.
        // If we extend the number of operators, this number must be extended too.
        return 5;
    }
}
```

8.3 A State class example: HungryCavalryState

In this chapter we can see a well written state-space representation. We can see that every State class must originate from the AbstractState class. We can see that how the different methods should be realized. As to solve one's own task, only a personal State class must be written, it worth starting from this example or the one in the next chapter.

8.3.1 Source code

```
/// <summary>
/// This is the state-space representation of the "hungry cavalry" problem.
/// The cavalry must get from the place of the station, the upper left corner,
/// to the cantina, which is in the bottom right corner.
/// We represent the table with a (N+4)*(N+4) matrix.
/// The outer 2 lines are margins, the inside part is the table.
/// With the use of the margins, it much more easy to write the IsState predicate.
/// The meaning of 0 is empty. The meaning of 1 is: here is the horse.
/// With a 3*3 table, the initial state is the following:
///  0,0,0,0,0,0,0
///  0,0,0,0,0,0,0
///  0,0,1,0,0,0,0
///  0,0,0,0,0,0,0
///  0,0,0,0,0,0,0
///  0,0,0,0,0,0,0
```

```
///  0,0,0,0,0,0,0
/// It is visible from the above representation that it's enough to store the position of the horse,
/// as it is the horse only that is on the table. So the initial state is (beginning from the upper left corner):
///  x = 2
///  y = 2
/// The goal state (I'm going to the bottom right corner):
///  x = N+1
///  y = N+1
/// Operators:
///  The 8 possible horse steps.
/// </summary>
class HungryCavalryState : AbstractState
{
   // By default it runs on a 3*3 chess board.
   static int N = 3;
   // The fields describing the inner state.
   int x, y;
   // Sets the inner state for the initial state.
   public HungryCavalryState()
   {
      x = 2; // We start from the upper left corner, which on the (2,2) coordinate
      y = 2; // due to the margin.
   }
   // Sets the inner state for the initial state.
   // We can also set the size of the board here.
   public HungryCavalryState(int n)
   {
      x = 2;
      y = 2;
      N = n;
   }
   public override bool IsGoalState()
   {
      // The bottom right corner is on (N+1,N+1) due to the margin.
      return x == N + 1 && y == N + 1;
   }
   public override bool IsState()
   {
      // the horse is not on the margin
      return x >= 2 && y >= 2 && x <= N + 1 && y <= N + 1;
   }
   private bool preHorseStep(int x, int y)
   {
      // if it's a correct horse step, if not then return false
      if (!(x * y == 2 || x * y == -2)) return false;
      return true;
   }
   /* This is my operator, returns true if it's executable,
    * else it returns false.
    * Parameters:
    * x: x direction move
    * y: y direction move
    * The precondition checks if the move is a horse step.
    * The postcondition checks if we remained on the board.
    * Example:
    * HorseStep(1,-2) meaning: 2 up, 1 right.
    */
   private bool HorseStep(int x, int y)
   {
      if (!preHorseStep(x, y)) return false;
      // If the precondition is true, we execute the
      // state transation.
      this.x += x;
```

```
        this.y += y;
        // The postcondition is always equals to the IsState.
        if (IsState()) return true;
        // If the postcondition is not true, than it must be withdrawn.
        this.x -= x;
        this.y -= y;
        return false;
    }
    public override bool SuperOperator(int i)
    {
        switch (i)
        {   // We list the 8 possible horse steps here
            case 0: return HorseStep(1, 2);
            case 1: return HorseStep(1, -2);
            case 2: return HorseStep(-1, 2);
            case 3: return HorseStep(-1, -2);
            case 4: return HorseStep(2, 1);
            case 5: return HorseStep(2, -1);
            case 6: return HorseStep(-2, 1);
            case 7: return HorseStep(-2, -1);
            default: return false;
        }
    }
    public override int NumberOfOps() { return 8; }
    // when writing it, we substract the width of the margin from x and y.
    public override string ToString() { return (x-2) + " : " + (y-2); }
    public override bool Equals(Object a)
    {
        HungryCavalryState aa = (HungryCavalryState)a;
        return aa.x == x && aa.y == y;
    }
    // If two példánys are equal, than their hasítókóds are equal too.
    public override int GetHashCode()
    {
        return x.GetHashCode() + y.GetHashCode();
    }
}
```

8.4 Another State class example

In this chapter we can see the state-space representation of the well-known 3 monks 3 cannibals problem. As all State classes, this has to originate from the AbstractState class too. As to solve one's own task, only a personal State class must be written, it worth starting from this example or the one in the previous chapter.

8.4.1 The example source code of the 3 monks and 3 cannibals

```
/// <summary>
/// The state-space representation of the "3 monks, 3 cannibals" problem.
/// Or rather it's generalization to any number of monks and cannibals.
/// Problem: there are 3 monks and 3 cannibals on the left bank of the river.
/// All of them must be transported to the other side of the river.
/// For this, a 2-man boat is available.
/// One man is enough to get to the other side, more than two people can't fit in.
/// If someone goes to the other side, he must get out, can't stay in the boat.
/// The problem is, if there are more cannibals than monks on either side of the river,
/// then the cannibals eat the monks.
/// Initial state:
///  3 monks on the left side.
```

```
///  3 cannibals on the left side.
///  The boat is on the left side.
///  0 monks on the right side.
///  0 cannibals on the right side.
/// This state is described with the following organized quintet:
///  (3,3,'L',0,0)
/// The goal state:
///  (0,0,'R',3,3)
/// Operator:
///  Op(int m, int c):
///  m monks go to the other side and
///  c cannibals go to the other side.
/// Possible parameters:
///  Op(1,0): 1 monks goes to the other.
///  Op(2,0): 2 monks goes to the other side.
///  Op(1,1): 1 monk and 1 cannibal goes to the other side.
///  Op(0,1): 1 cannibal goes to the other side.
///  Op(0,2): 2 cannibal goes to the other side.
/// </summary>
class MonksAndCannibalsState : AbstractState
{
   int m;   // the number of monks all together
   int c;    // the number of cannibals all together
   int ml;  // the number of monks on the left side
   int cl;   // the number of cannibals on the left side
   char b;  // Where the boat is? It's value is either 'L' or 'R'.
   int mr;  // the number of monks on the right side
   int cr;   // the number of cannibals on the right side
   public MonksAndCannibalsState(int m, int c) // sets the initial state
   {
      this.m = m;
      this.c = c;
      ml = m;
      cl = c;
      b = 'L';
      mr = 0;
      cr = 0;
   }
   public override bool IsState()
   {   // true, if the monks are safe
      return ((ml >= cl) || (ml == 0)) &&
         ((mr >= cr) || (mr == 0));
   }
   public override bool IsGoalState()
   {
      // true, if everyone got to the other side
      return mr == m && cr == c;
   }
   private bool preOp(int m, int c)
   {
      // The boat can carry 2 people, at least 1 is needed for paddling.
      // At the end, it's needles to check it as the super operator invites it according to this.
      if (m + c > 2 || m + c < 0 || m < 0 || c < 0) return false;
      // m monks and c cannibals are only transportable
      // if the boat's side has at least that many.
      if (b == 'L')
         return ml >= m && cl >= c;
      else
         return mr >= m && cr >= c;
   }
   // Transports m monks and c cannibals to the other side.
   private bool op(int m, int c)
   {
```

```
        if (!preOp(m, c)) return false;
        MonksAndCannibalsState mentes = (MonksAndCannibalsState)Clone();
        if (b == 'L')
        {
            ml -= m;
            cl -= c;
            b = 'R';
            mr += m;
            cr += c;
        }
        else
        {
            ml += m;
            cl += c;
            b = 'L';
            mr -= m;
            cr -= c;
        }
        if (IsState()) return true;
        ml = without.ml;
        cl = without.cl;
        b = without.b;
        mr = without.mr;
        cr = without.cr;
        return false;
    }
    public override int NumberOfOps() { return 5; }
    public override bool SuperOperator(int i)
    {
        switch (i)
        {
            case 0: return op(0, 1);
            case 1: return op(0, 2);
            case 2: return op(1, 1);
            case 3: return op(1, 0);
            case 4: return op(2, 0);
            default: return false;
        }
    }
    public override string ToString()
    {
        return ml + "," + cl + "," + b + "," + mr + "," + cr;
    }
    public override bool Equals(Object a)
    {
        MonksAndCannibalsState aa = (MonksAndCannibalsState)a;
        // mr and cr are countable, so no need for checking
        return aa.ml == ml && aa.cl == cl && aa.b == b;
    }
    // If two instances are equal, than their hash codes are equal too.
    public override int GetHashCode()
    {
        return ml.GetHashCode() + cl.GetHashCode() + b.GetHashCode();
    }
}
```

8.5 The Vertex class

In this chapter, we get to know the vertex class. Fortunately, we don't need to change this if we want to solve one's one task with backtrack or with depth-first method. If we want to implement some

other algorithm, like the uniform-cost method, instead of an algorithm based on depth, then we need to modify it so the state will contain the the value of the cost function.

In this version, the vertex contains a state, it's depth and it's parent. It supports extending and with the help of the super operators, we can use the operators one by one.

As the vertex knows it's parent, for recording the path it's enough to store the last vertex of the path. This means that for the solution, it's enough to return the terminal vertex where the solution leads too.

8.5.1 Source code

```
/// <summary>
/// The vertex contains a state, the vertex's depth and the vertex's parent.
/// So the vertex represents a whole path till the starting vertex.
/// </summary>
class Vertex
{
   // The vertex contains a state, it's depth and it's parent
   AbstractState state;
   int depths;
   Vertex parent; // Going upwards on the parents we get to the starting vertex.
   // Constructor:
   // Sets the inner state on the starting vertex.
   // It's the inviter's responsibility to invite it with the initial state.
   // The depth of the starting vertex is 0 and it has no parent.
   public Vertex(AbstractState initialState)
   {
      state = initialState;
      depth = 0;
      parent = null;
   }
   // A new child creates a vertex.
   // We need to invite an applicable operator, only after that it is ready.
   public Vertex(Vertex parent)
   {
      state = (AbstractState)parent.state.Clone();
      depth = parent.depth + 1;
      this.parent = parent;
   }
   public Vertex GetParent() { return parent; }
   public int GetDepth() { return depth; }
   public bool TerminalVertexE() { return state.IsGoalState(); }
   public int NumberOfOps() { return state.NumberOfOps(); }
   public bool SuperOperator(int i) { return state.SuperOperator(i); }
   public override bool Equals(Object obj)
   {
      Vertex cs = (Vertex)obj;
      return state.Equals(cs.state);
   }
   public override int GetHashCode() { return state.GetHashCode(); }
   public override String ToString() { return state.ToString(); }
   // Executes all the executable operators.
   // Returns the so created new vertices.
   public List<Vertex> Extending()
   {
      List<Vertex> newVertices = new List<Vertex>();
      for (int i = 0; i < NumberOfOps(); i++)
      {
         // We create a new child vertex.
         Vertex newVertex = new Vertex(this);
```

```
            // We try the i th basic operator? Is it executable?
            if (newVertex.SuperOperator(i))
            {
                // If yes, we add it to the new ones.
                newVertices.Add(newVertex);
            }
        }
        return newVertices;
    }
}
```

8.6 The GraphSearch class

In this chapter we introduce the common concepts of the graph searcher algorithms. This class probably won't need modification, not even in the case when a new graph searcher algorithm has been written. Al graph searches must originate from this class.

It has one abstract method, the Search. This is what needs to be realized in children classes. In case of the backtrack, it will be the backtrack itself, in case of the depth-first method, it will be the depths search. The Search returns with a vertex. This must be the terminal vertex, if a solution exists, in other case, it must be null. The returning null value indicates that there is no solution.

The solution can be written with the WriteSolution method. This is only a small aid, its use is not required. If we decide on using it, there is an example in the main program for its call.

Every graph searcher starts searching in the starting vertex, so we prescribe a constructor too. This must be invited by every children class.

8.6.1 Source code

```
/// <summary>
/// The ancestor of all graph searchers.
/// The graph searchers only need to realize the Search method.
/// This will return a terminal vertex, if there is a solution or a null if there isn't.
/// From the terminal vertex, the solution can be made by going up on the parent references.
/// </summary>
abstract class GraphSearch
{
    private Vertex startingVertex; // The starting vertex vertex.
    // Every graphsearxher starts searching in the starting vertex.
    public GraphSearch(Vertex startingVertex)
    {
        this. startingVertex = startingVertex;
    }
    // It's better if the starting vertex is private, but the children classes can request it.
    protected Vertex GetStartingVertex () { return startingVertex; }
    /// If there is a solution, namely there is a path in the state-space graph,
    /// that leads from the starting vertex to the terminal vertex,
    /// it returns a solution, else it's null.
    /// The solution is given as a terminal vertex.
    /// From this terminal vertex, the solution can be made by going up on the parent references in reverse order.
    public abstract Vertex Search();
    /// <summary>
    /// Writes the solution based on a terminal vertex.
    /// It assumes that by going up on the references of the parent vertex we will get to the starting vertex.
    /// It reverses the order of the vertices to write the solution correctly.
    /// If a vertex is null, it writes that there is no solution.
    /// </summary>
    /// <param name="oneTerminalVertex">
```

```
    /// The terminal vertex representing the solution or null.
    /// </param>
    public void writingSolution(Vertex oneTerminalVertex)
    {
        if ( oneTerminalVertex == null)
        {
            Console.WriteLine("No solution");
            return;
        }
        // The order of the vertices must be reversed.
        Stack<Vertex> solution = new Stack<Vertex>();
        Vertex curVertex = oneTerminalVertex;
        while (curVertex != null)
        {
            solution.Push(curVertex);
            curVertex = curVertex.GetParent();
        }
        // It has been reversed, can be written.
        foreach(Vertex cur in solution) Console.WriteLine(cur);
    }
}
```

8.7 The backtrack class

In this chapter we introduce a recursive solution of the backtrack graph search. Different constructors can be used to create a backtrack with depth limit, with memory or the combination of these.

It worth trying to rewrite the below realization to a loop based solution. It is a great practice, and if it doesn't work, the original is still there to create the hand-in program. In this, we don't need to even touch this code. It is a well-tested code.

8.7.1 Source code

```
/// <summary>
/// Class that realizes the the backtrack graph searcher algorithm.
/// It contains the three basic backtrack algorithms in one. These are:
/// - the basic backtrack
/// - the backtrack with depth limit
/// - the backtrack with memory
/// The backtrack with branch-limit hasn't been worked out here.
/// </summary>
class BackTrack : GraphSearch
{
    int limit; // If it's not null, than the searcher has depth limit.
    bool memory; //if it's true, the searcher has memory.
    public BackTrack(Vertex startingVertex, int limit, bool memory) : base(startingVertex)
    {
        this.limit = limit;
        this.memory = memory;
    }
    // there is neither a depth limit, nor a memory
    public BackTrack(Vertex startingVertex) : this(startingVertex, 0, false) { }
    // searcher with depth limit
    public BackTrack(Vertex startingVertex, int limit) : this(startingVertex, limit, false) { }
    // searcher with memory
    public BackTrack(Vertex startingVertex, bool memory) : this(startingVertex, 0, memory) { }
    // The search starts from the starting vertex.
    // Returns a terminal vertex. We can get to this terminal vertex from the starting vertex.
```

```
  // If there is no such vertex, it returns a null value.
  public override Vertex Search() { return Search(GetStartingVertex()); }
  // The recursive realization of the searcher algorithm.
  // As it is recursive, the backstep is the "return null".
  private Vertex Search(Vertex curVertex)
  {
    int depth = curVertex.GetDepth();
    // checking the depth limit
    if (limit > 0 && depth >= limit) return null;
    // the use of memory to remove circles
    Vertex curParent = null;
    if (memory) curParent = curVertex.GetParent();
    while (curParent != null)
    {
      // We check if we have been in this state previously. If yes, then backstep.
      if (curVertex.Equals(curParent)) return null;
      // Going back on the parent trail.
      curParent = curParent.GetParent();
    }
    if (curVertex.TerminalVertexE())
    {
      // We have the solution, the terminal vertex must be returned.
      return curVertex;
    }
    // We invite the basic operators here through the super operator.
    // If any of them can be applied, we create a new vertex,
    // and invite myself recursively.
    for (int i = 0; i < curVertex.NumberOfOperators(); i++)
    {
      // We create the new child vertex.
      // This will be done only if we apply an applicable operator on it.
      Vertex newVertex = new Vertex(curVertex);
      // We try the i^th basic operator. Is it applicable?
      if (newVertex.SuperOperator(i))
      {
        // If yes, then we invite myself recursively to the new vertex.
        // If it returns with a value other than null, then a solution has been found.
        // If it returns with a null value, we have to try the next basic operator.
        Vertex terminal = Search(newVertex);
        if (terminal != null)
        {
          // We return the terminal vertex that represents the solution.
          return terminal;
        }
        // The operator should be withdrawn on the else-branch.
        // This is required if there was no cloning in the creation of the new child.
        // As we cloned, this part is empty.
      }
    }
    // If we tried all the operators and none of them has given a solution then backstep.
    // After the backstep, we try to use the next basic operator one level higher.
    return null;
  }
}
```

8.8 The DepthFirstMethod class

In this chapter, we introduce the depth-first search in two versions. The first version uses loop-checking, as the original depth searcher uses it too. The other version doesn't use it, so it is much faster. The second version should only be used if there are no loops in the state-space graph, else the

searcher may enter an infinite cycle.

Both versions assume that the starting vertex is not terminal. If this cannot be assumed, than we need to test the vertex that we selected for expanding if it is a terminal vertex or not. Otherwise, it's enough to test the new vertices, as it is done in the code below.

It is a well-known fact that the depth-first search should be done with a stack, while the breadth-first search should be done with a queue. According to this, we store the open vertices in a stack. Due to this, on the top of the stack, it will be the open vertex with the greatest depth. This greatly hastens the search, as we don't need to look for the vertex with the greatest depth.

We advice that the reader would write the breadth-first search based on the code below. This shouldn't cause any trouble. It is a bit more complicated task to write the uniform-cost method based on these information. To achieve this, the Vertex class must be completed with the cost, and instead of the stack, a sorted list should be used to find the open vertex with the lowest cost. With the uniform-cost method, we also need to pay attention to the fact that more paths may lead to one vertex, but we only need to store the one with the lowest cost. The A-algorithm, where there is a heuristic too, is even more exciting.

8.8.1 Source code

```
/// <summary>
/// A graph search class that realizes the depth search algorithm.
/// This realization of the depth search assumes that the starting vertex is not terminal.
/// The open vertices are stored in a stack.
/// </summary>
class DepthFirstMethod : GraphSearch
{
   // In the depth search, it worth storing the open vertices in a stack,
   // as in this way, the vertex with the greatest depth will be on the top of the stack.
   // So we don't need to search for the open vertex with the greatest depth.
   Stack<Vertex> Open; // Set of the open vertices.
   List<Vertex> Closed; // Set of the closed vertices.
   bool circleChecking; // If it is false, it may enter into an infinite cycle.
   public DepthFirstMethod(Vertex startVertex, bool circleChecking) :
      base (startingVertex)
   {
      Open = new Stack<Vertex>();
      Open.Push(startingVertex);    // at the beginning, only the starting vertex is open
      Closed = new List<Vertex>();  // at the beginning, the set of closed vertices is empty
      this.circleChecking = circleChecking;
   }
   // The default value of circle checking is true.
   public DepthFirstMethod(Vertex startingVertex) : this(startingVertex, true) { }
   // The is the point where the search for the solution starts.
   public override Vertex Search()
   {
      // If we don't need the circle checking, the algorithm will be much faster.
      if (circleChecking) return SearchingTerminalVertex();
      return SearchingTerminalVertexFast();
   }
   private Vertex SearchingTerminalVertex()
   {
      // Till the set of open vertices is not empty.
      while (Open.Count != 0)
      {
         // This is the open vertex with the greatest depth.
         Vertex C = Open.Pop();
         // We expand this one.
         List<Vertex> newVertices = C.Expending();
```

```
            foreach (Vertex D in newVertices)
            {
               // I'm ready if we have found the terminal vertex.
               if (D.TerminalVertexE()) return D;
               // We only pick up the vertices to the list of open vertices
               // that haven't been included neither in the set of open, nor in the set of closed vertices.
               // The Contains invites the Equals method that was written in the Vertex class.
               if (!Closed.Contains(D) && !Open.Contains(D)) Open.Push(D);
            }
            // We made the expanded vertex to be a closed vertex.
            Closed.Add(C);
         }
         return null;
      }
      // This should only be used if we are sure that the state-space graph does not contain a circle!
      // Else it will probably cause an infinite cycle.
      private Vertex SearchingTerminalVertexFast()
      {
         while (Open.Count != 0)
         {
            Vertex C = Open.Pop();
            List<Vertex> newVertices = C.Expanding();
            foreach (Vertex D in newVertices)
            {
               if (D.TerminalVertexE()) return D;
               // If there is no circle, it is needless to check if D had been among the open or closed vertices.
               Open.Push(D);
            }
            // if there is no circle, it is needless to made C closed.
         }
         return null;
      }
   }
```

8.9 The Main Program

In this chapter we can see how the different parts should be merged. It's very important for the starting vertex to get the initial state. We have to give the starting vertex to the constructor of the graph searcher. These are all the responsibilities of the main program.

It worth checking that we can invite different searchers on the same starting vertex and they will end in slightly different results. Another worthy thing is playing with the constants. Note that a depth limit of 10 is enough for the example of Hungry Cavalryman, but it is not enough for the 3 monks, 3 cannibals. It's also interesting to check that what size of a task will make the problem difficult to solve. It may be salient to, that the depth-first method may run out of memory with greater problems, where the backtrack has plenty of memory. It also worth noting that reordering the sequence of operators in the super operator influences the run time greatly. A brave man may rewrite the super operator to use heuristics.

It is visible that it doesn't matter which graph searcher algorithm we use, the Searching method must be invited. The writeSolution method should be invited on the vertex that was returned by that method.

8.9.1 Source code

```
class Program
{
   static void Main(string[] args)
```

```
    {
        Vertex startingVertex;
        GraphSearch search;
        Console.WriteLine("We solve the Hungry Cavalryman problem on a 4x4 board.");
        startingVertex = new Vertex(new HungryCavalryState(4));

        Console.WriteLine("The searcher is a backtrack searcher with memory and depth limit of 10.");
        search = new BackTrack(startingVertex, 10, true);
        search.writeSolution(search.Searching());

        Console.WriteLine("The searcher is a depth searcher with circle checking.");
        search = new DepthFirstMethod(startingVertex, true);
        search.writeSolution(search.Searching());

        Console.WriteLine("We solve the 3 monks, 3 cannibals problem.");
        startingVertex = new Vertex(new MonksAndCannibalsState(3,3));

        Console.WriteLine("The searcher is a backtrack searcher with memory and depth limit of 15.");
        search = new BackTrack(startingVertex, 15, true);
        search.writeSolution(search.Searching());

        Console.WriteLine("The searcher is a depth searcher with circle checking.");
        search = new DepthFirstMethod(startingVertex, true);
        search.writeSolution(search.Searching());

        Console.ReadLine();
    }
}
```

BIBLIOGRAPHY

(1) Stuart J. Russell, Peter Norvig: *Mesterséges intelligencia modern megközelítésben*
Panem – Prentice Hall, 2000 (*Hungarian title*)

(2) Pásztorné Varga Katalin, Várterész Magda: *A matematikai logika alkalmazásszemléletű tárgyalása*
Panem, 2003

(3) Dr. Várterész Magda: *Mesterséges intelligencia*
http://www.inf.unideb.hu/~varteres/mi/tartalom.htm

(4) Dr. Kovásznai Gergely: *Logikai programozás*
http://aries.ektf.hu/~kovasz/logprog/logikai_programozas.pdf

(5) CLIPS
http://www.ghg.net/clips/CLIPS.html

(6) SWI-Prolog
http://www.swi-prolog.org