Vilnius University
Faculty of Mathematics and Informatics

Vytautas Čyras

ARTIFICIAL INTELLIGENCE

Vilnius
2016

Contents

Preface

This course-book views artificial intelligence (AI) from the standpoint of programming. Fundamental concepts of classical AI are presented: problem solving by search, solver, planner, etc. A purpose is to understanding the spirit of a discipline of artificial intelligence.

The course-book presents primarily figures and text excerpts which are comprised in the extended Lithuanian edition[1].

The following themes are presented:

- History of artificial intelligence
- Philosophical questions
- The Turing test. Searle's "Chinese Room" experiment
- A system of artificial intelligence (according to Nils Nilsson): (1) a global data base, (2) a set of production rules, and (3) a control system
- Examples: the 8 queens puzzle, the knight's tour, path search in a labyrinth
- Problem spaces
- Backtracking
- Depth-first search and breadth-first search, Dijkstra's algorithm, A*
- The role of heuristics
- Forward chaining and backward chaining
- Knowledge-based reasoning, deduction, resolution technique
- Hill climbing
- Elements of expert systems architecture: facts, rules, and an inference engine
- Knowledge representation
- Structured representation, frames and objects
- Semantic networks
- Artificial intelligence and law

Acknowledgements. Sincere thanks to students who contributed to this course-book. Edgaras Abromaitis' work was essential in shaping the text and producing the figures.

[1] V. Čyras. Intelektualios sistemos. e-Book. ISBN 978-9955-33-561-0. <http://www.mif.vu.lt/~cyras/AI/konspektas-intelektualios-sistemos.pdf>.

1. Introduction

1.1.Historical Remarks

The term “artificial intelligence” was coined more than 50 years ago. The Dartmouth conference (1956) coined the term “artificial intelligence”. A conference dedicated to 50 years anniversary was celebrated in 2006; see http://www.ki2006.fb3.uni-bremen.de/50years.htm.

This section presents several sources of AI. They show the place of AI within the discipline of computer science. The sources can be viewed as precedents that prove the subject matter of AI.

Following is a classification of *computing*, provided by Association for Computer Machinery in US; see http://www.acm.org/class/1998/:

A. General Literature
B. Hardware
C. Computer Systems Organization
D. Software
E. Data
F. Theory of Computation
G. Mathematics of Computing
H. Information Systems
I. Computing Methodologies
J. Computer Applications
K. Computing Milieux

Artificial intelligence is comprised by the branch I. Computing Methodologies:

I.0 GENERAL
I.1 SYMBOLIC AND ALGEBRAIC MANIPULATION
I.2 **ARTIFICIAL INTELLIGENCE**
I.3 COMPUTER GRAPHICS
I.4 IMAGE PROCESSING AND COMPUTER VISION
I.5 PATTERN RECOGNITION
I.6 SIMULATION AND MODELING
I.7 DOCUMENT AND TEXT PROCESSING

Artificial intelligence – I.2 above – is classified further:

I.2 ARTIFICIAL INTELLIGENCE
- I.2.0 General
- I.2.1 Applications and Expert Systems
- I.2.2 Automatic Programming
- I.2.3 Deduction and Theorem Proving
- I.2.4 **Knowledge Representation Formalisms and Methods**
  - Frames and scripts
  - Modal logic
  - Predicate logic
  - Relation systems

Representation languages
Representations (procedural and rule-based)
Semantic networks
Temporal logic
I.2.5 Programming Languages and Software
I.2.6 Learning
I.2.7 Natural Language Processing
I.2.8 Problem Solving, Control Methods, and Search
I.2.9 Robotics
I.2.10 Vision and Scene Understanding
I.2.11 Distributed Artificial Intelligence

There are many definitions of artificial intelligence. They characterise the main features of AI. A definition by Judea Pearl, Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference, 14 (1988); see [MacCrimmon, Tillers 2002, p. 55]):

> The aim of artificial intelligence is to provide a computational model of intelligent behavior, most importantly, commonsense reasoning.

What means “artificially intelligent”? intelligent entities have the following abilities: 1) perceive, 2) understand, 3) predict, and 4) manipulate.

1.2.The Towers of Hanoi

See http://en.wikipedia.org/wiki/Towers_of_Hanoi.

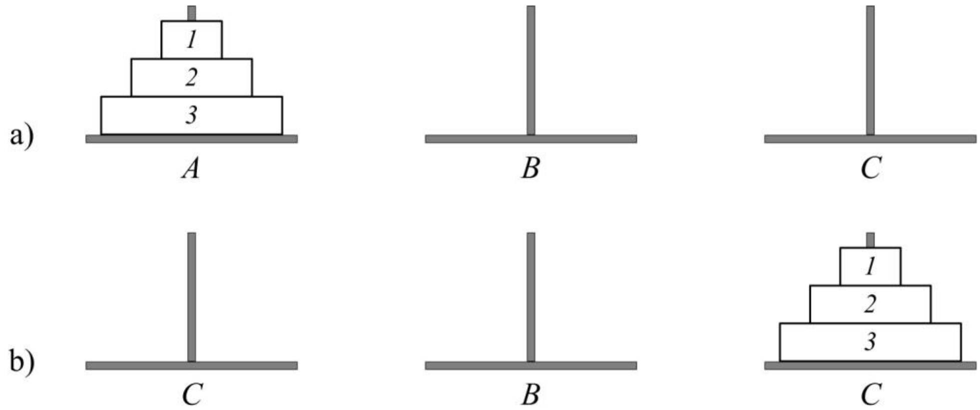


Fig. 1.1. a) Initial state A=(3,2,1), B=(), C=(). b) Terminal state A=(), B=(), C=(3,2,1)

Finding a sequence of moves is an intelligent task. Intelligence is hidden in the recursive algorithm below:

Part 1. Move *n–1* disks from *A* to *B*;

Part 2. Move disk *n* from *A* to *C*;

Part 3. Move *n–1* disks from *B* to *C*.

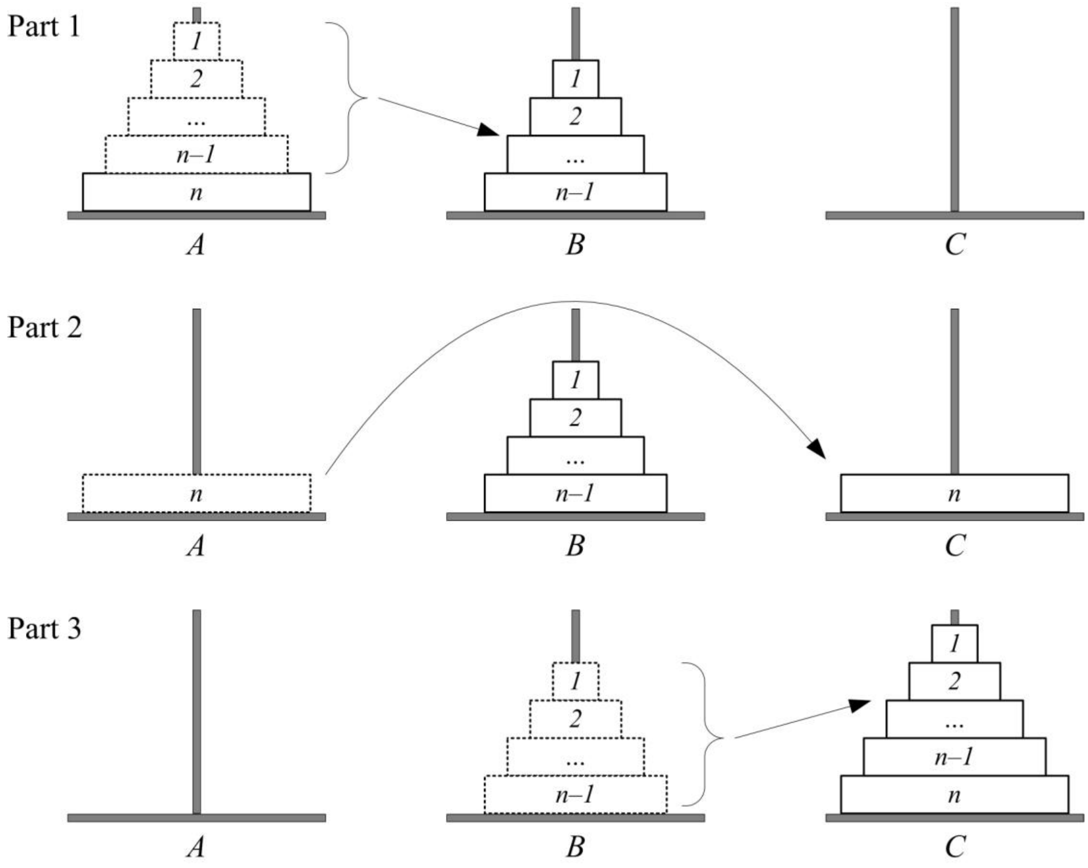


Fig. 1.2. Three parts of the recursive algorithm

The sequence of moves for *n=3*:

Initial state   *A*=(*3*,*2*,*1*), *B*=(),   *C*=()

1. Move disk *1* from *A* to *C*.   *A*=(*3*,*2*), *B*=(),   *C*=(*1*)
2. Move disk *2* from *A* to *B*.   *A*=(*3*),   *B*=(*2*),   *C*=(*1*)
3. Move disk *1* from *C* to *B*.   *A*=(*3*),   *B*=(*2*,*1*), *C*=()
4. Move disk *3* from *A* to *C*.   *A*=(),   *B*=(*2*,*1*), *C*=(*3*)
5. Move disk *1* from *B* to *A*.   *A*=(*1*),   *B*=(*2*),   *C*=(*3*)
6. Move disk *2* from *B* to *C*.   *A*=(*1*),   *B*=(),   *C*=(*3*,*2*)
7. Move disk *1* from *A* to *C*.   *A*=(),   *B*=(),   *C*=(*3*,*2*,*1*)

Following is the recursive procedure:

```
procedure ht(x, y, z: char; n: integer);
{x, y, z - rod names; n - number of disks.}
{x - from, y - intermediary, z - onto.}
begin
  if n > 0 then
  begin
    ht(x, z, y, n-1); {1. Move n-1 disks onto intermediary.}
    writeln(' Move from ', x,' to ', z);                  {2.}
    ht(y, x, z, n-1); {3. Move n-1 disks onto target.      }
  end
end
```

Invoking the above procedure with *n=3*:

```
ht(A, B, C, 3)
```

The number of moves is exponential – 2^{n-1}.

Further we discuss state transition in *state transition graphs*. Fig. 1.3 shows GRAPHSEARCH_DEPTH_FIRST search tree for the Hanoi tower problem *n=3*.

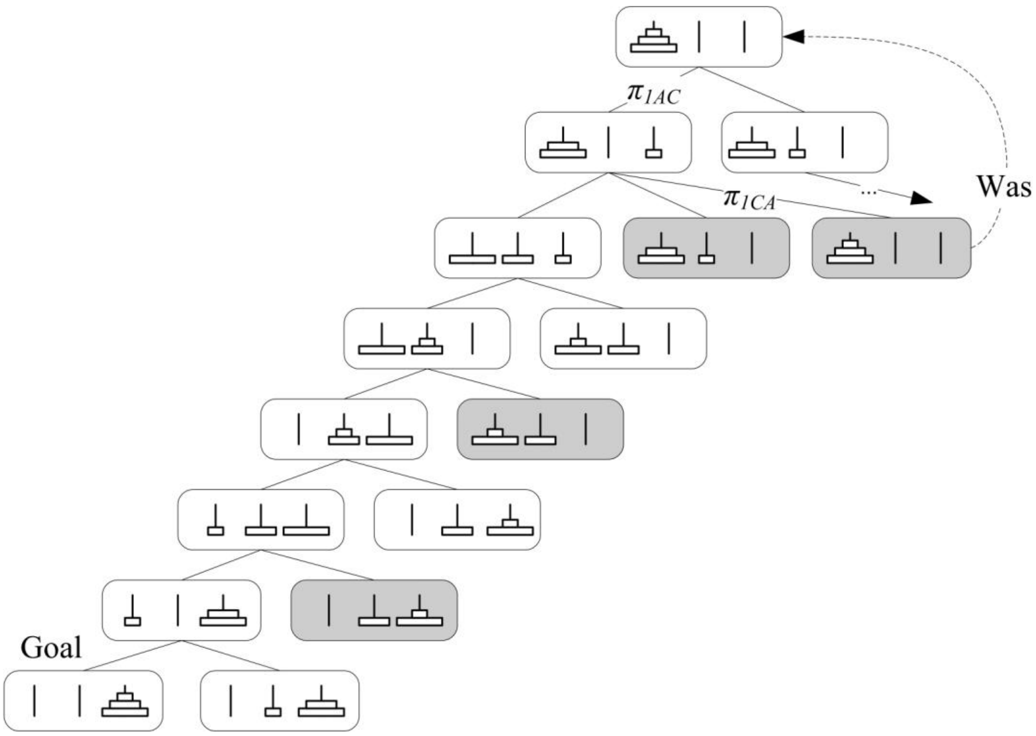


Fig. 1.3. A search tree of the algorithm GRAPHSEARCH_ DEPTH_FIRST for the Hanoi tower problem *n=3*. The grey nodes appear in the list OPEN or CLOSED

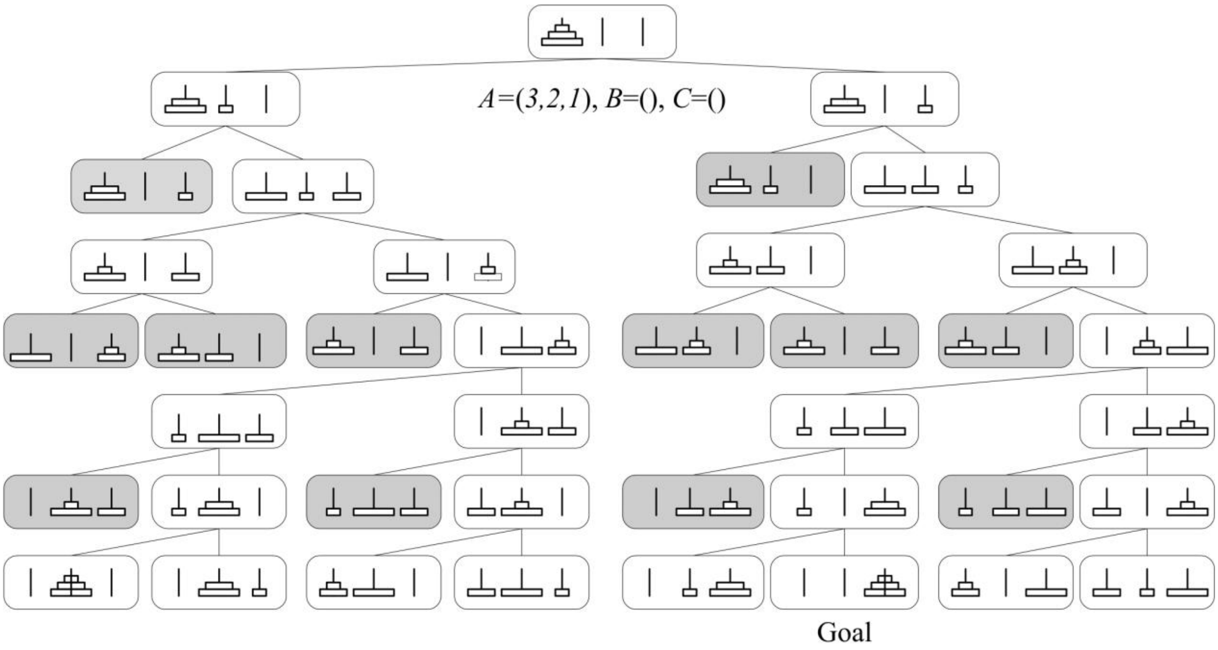


Fig. 1.4 . A search tree of the GRAPHSEARCH_BREADTH_FIRST algorithm for the Hanoi tower problem. The grey nodes appear in the list OPEN or CLOSED

Exercise 1. The iterative Hanoi tower solution

The iterative algorithm for the Hanoi tower problem is less well-known as the recursive one. First, we arrange the pegs in a circle, so that clockwise we have *A*, *B*, *C*, and then *A* again. Following this, assuming we never move the same disk twice in a row, there will always only be one disk that can be legally moved, and we transfer it to the first peg it can occupy, moving in a clockwise direction, if *n* is even (2, 4, 6,...), and counterclockwise, in *n* is odd (1, 3, 5,...). (See Brachman & Levesque (2004), *"Knowledge representation and reasoning"*, p. 133–134, Exercise 2)

1) Write an iterative program.

2) Or alternatively, write a collection of production rules that implement this procedure. Initially, the working memory will have elements (on peg: A disk: *i*), for each disk *i*, and an element (solve). When your rules stop firing, you working memory should contain (done) and (on peg: C disk: *i*), for each disk *i*.

Exercise 2. Monkey and banana problem

A monkey is in a room. Suspended from the ceiling is a bunch of bananas, beyond the monkey's reach. However, in the room there are also a chair and a stick. The ceiling is just the right height so that a monkey standing on a chair could knock the bananas down with the stick. The monkey knows how to move around, carry other things around, reach for the bananas, and wave a stick in the air. What is the best sequence of actions for the monkey? (https://en.wikipedia.org/wiki/Monkey_and_banana_problem)

1.3.Dynamic Contents of Artificial Intelligence

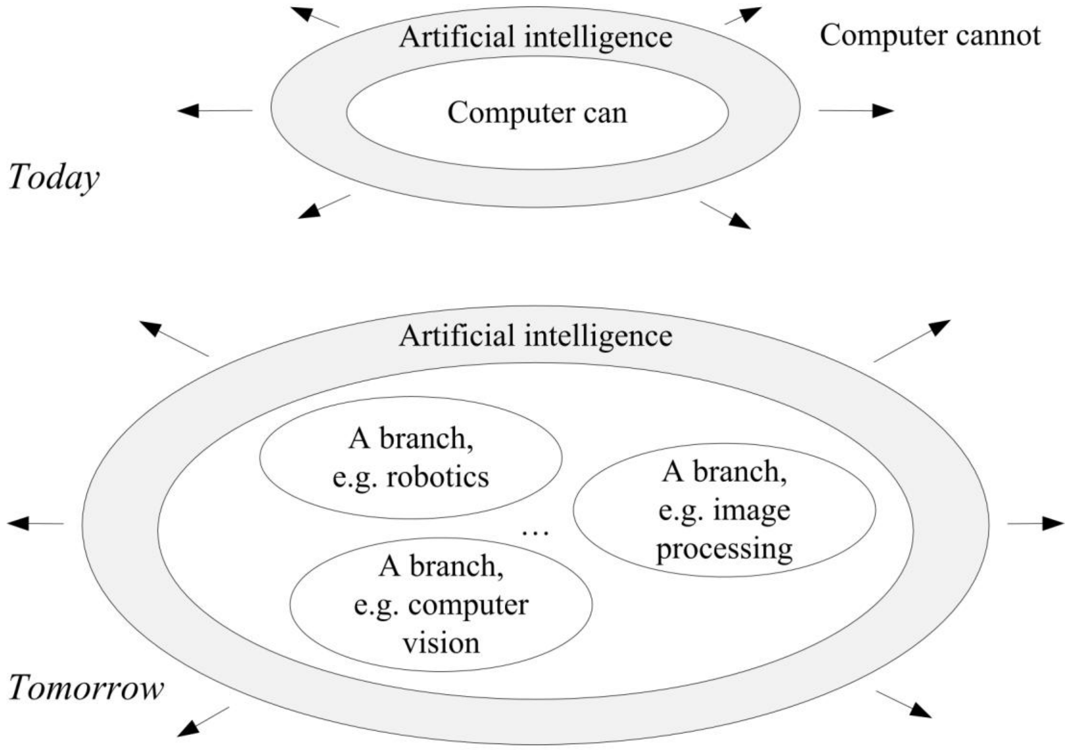


Fig. 1.5. The issues which are comprised by AI today, may be not comprised tomorrow. Thus new branches of AI are formed

1.4.The Methods of Problem Solving

A classification of tasks:

1. interpret (analysis)
   1.1. identify (recognise)
      1.1.1. monitor (audit, check)
      1.1.2. diagnose (debug)
   1.2. predict (simulate)
   1.3. control
2. construct (synthesis)
   2.1. specify (constrain)
   2.2. design
      2.2.1. plan (process)
      2.2.2. configure (structure)
   2.2. assemble (manufacture)
      2.3.1. modify (repair)

Following is a classification of problem solving methods:

1. classification)
   1.1. certain classification
      1.1.1. decision trees
      1.1.2. decision tables
   1.2. heuristic classification
   1.3. model-based classification
      1.3.1. set covering classification
      1.3.2. functional classification
   1.4. statistical classification
   1.5. case-based classification
2. construction)
   2.1. heuristic construction
      2.1.1. skeletal construction
      2.1.2. propose-and-revise strategy
      2.1.3. propose-and-exchange strategy
      2.1.4. least-commitment strategy
   2.2. model-based construction
   2.3. case-based construction
3. simulation
   3.1. one-step simulation
   3.2. multiple-step simulation
   3.2.1. numerical multiple-step simulation
   3.2.2. qualitative multiple-step simulation

What is a difference between a method and a principle? A method is applied in a clear situation. A principle is applied is a situation that needs not to be clear. A principle is applied in the case a method is not known in a given situation. This is similar to a difference between a legal norm and a principle.

Albertas Čaplinskas describes the terms “method” and “methodology” in software engineering:

Method = schema + procedures

Methodology = method + recommendations

2. Artificial Intelligence System as a Production System

This section follows [Nilsson 1982, Section 1.1]
DEFINITION 2.1. A *production system* is a triple:

1. A global database (GDB);
2. A production (rule) set $\{\pi_1, \pi_2, ..., \pi_m\}$;
3. A control system.

The basic production system algorithm for solving a problem such as the 8-puzzle, the knight's tour, the 8 queens puzzle, etc. can be written in nondeterministic form as follows:

```
procedure PRODUCTION
{1} DATA := initial GDB;
{2} until DATA will satisfies the termination condition, do
{3} begin
{4}     select  some rule, π, in the set of rules
                that can be applied to DATA
{5}     DATA := π(DATA)        {Result of applying π to DATA}
{6} end
```

The output is a sequence of productions $\langle \pi_{i1}, \pi_{i2},...,\pi_{in} \rangle$. This is a non-determinate procedure. PRODUCTION performs depth-first search. The procedure is treated as a *paradigm* of a control system. A meaning of the word "paradigm" can be found, for instance, in Oxford English Dictionary, http://dictionary.oed.com/.

The Knight's Tour problem (http://en.wikipedia.org/wiki/Knight%27s_tour) is examined to demonstrate problem solving with the PRODUCTION procedure. A knight's tour is a sequence of moves of a knight on a chessboard such that the knight visits every square only once. Suppose the knight starts from the square [*1*,*1*] (Fig. 2.1 a). Eight productions $\{\pi_1, \pi_2, \pi_3, \pi_4, \pi_5, \pi_6, \pi_7, \pi_8\}$ are shown in Fig. 2.1 b. Variations involve chessboards $N \times N$. The simplest case is N=5.

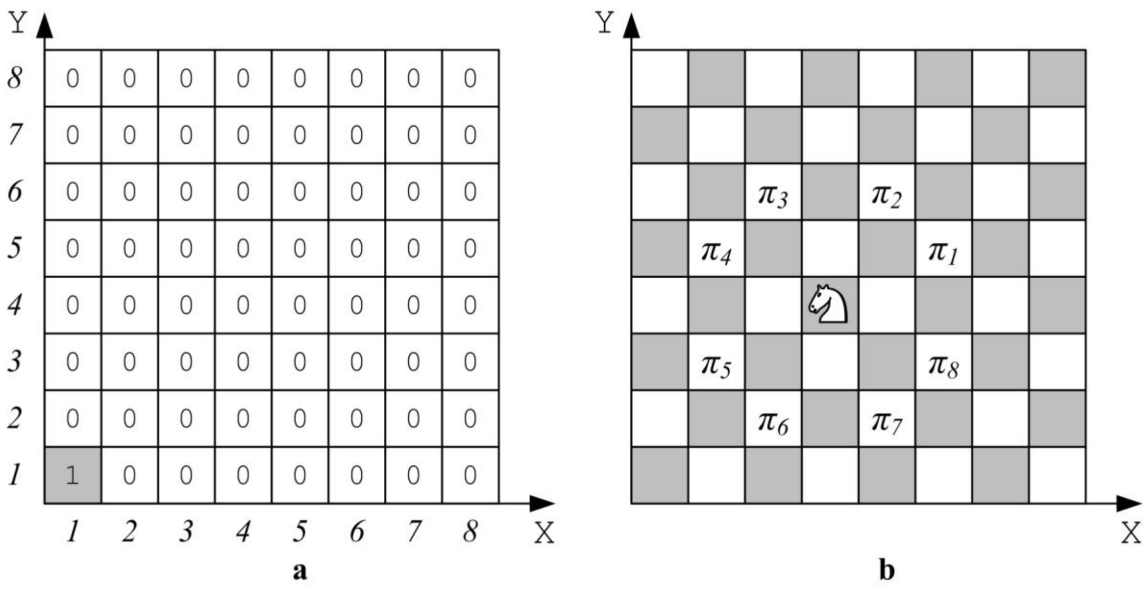


Fig. 2.1. The knight starts from the square [*1*,*1*] on a 8×8 chessboard

```
program KNIGHT;
const N     = 5;      {Length of the board}
      NN    = 25;     {Number of squares 5*5}
type  INDEX = 1..N;
var
   BOARD  : array[INDEX, INDEX] of integer;  {Global data base}
   CX, CY : array[1..8] of integer;          {8 production rules}
   I, J   : integer; YES: boolean;

procedure INITIALISE;
var   I, J : integer
begin
   {1) Production set is formed}
   CX[1] :=  2;    CY[1] :=  1;
   CX[2] :=  1;    CY[2] :=  2;
   CX[3] := -1;    CY[3] :=  2;
   CX[4] := -2;    CY[4] :=  1;
   CX[5] := -2;    CY[5] := -1;
   CX[6] := -1;    CY[6] := -2;
   CX[7] :=  1;    CY[7] := -2;
   CX[8] :=  2;    CY[8] := -1;

   {2) Initialise the global data base}
   for I := 1 to N do
      for J := 1 to N do
         BOARD[I, J] := 0;
end; {INITIALISE}

procedure TRY (L : integer; X, Y : INDEX; var YES : boolean);
{Input parameters: L – move number; X, Y – knight's last position.}
{Output parameter (i.e. the result): YES.}
var
   K     : integer; {Production number.}
   U, V  : integer; {New knight's position.}
begin
   K := 0;
   repeat  {Select each of 8 productions.}
      K := K + 1;
      U := X + CX[K]; V := Y + CY[K];
      {Check if the condition of the production is satisfied.}
      if (U >= 1) and (U <= N) and (V >= 1) and (V <= N)
      then {Within the board.}
         {Check if the square is empty.}
         if BOARD[U, V] = 0
         then
           begin
                {New position.}
                BOARD[U, V] := L;
                {Check if all squares are visited.}
                if L < NN
                then
                  begin {If not all are visited.}
                    TRY(L+1, U, V, YES);
                    {If no success,}
                    {then backtrack and free the position.}
                    if not YES then BOARD[U, V] := 0;
                  end
                else YES := true; {When L=NN.}
           end;
   until YES or (K = 8); {Either success or all productions were tried.}
end; {TRY}
```

```
begin {Main program)}
   {1. Initialise.}
   INITIALISE; YES := false;
   {2. Initial position [1,1].}
   BOARD [1, 1] := 1;
   {3. Make move no.2 from X=1 and Y=1 and obtain the answer YES.}
   TRY(2, 1, 1, YES);
   {4. If a solution found then print it.}
   if YES then
      for I := N downto 1 do
      begin
         for J := 1 to N do
            write(BOARD[I,J]);
         writeln;
      end;
   else writeln('Path does not exist.');
end.
```

The first solution is shown in Fig. 2.1 a. Note that several solutions exist. The first deadend is reached on square 21; see Fig. 2.1 b. The search tree is shown in Fig. 2.3.

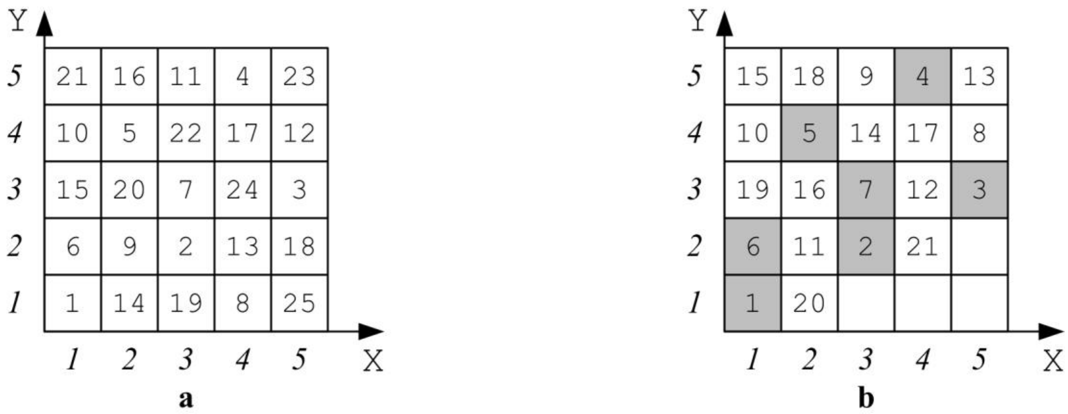


Fig. 2.2 . a) A solution on a 5×5 chessboard. b) Grey coloured are seven foremost moves that lead to success. Move 8 fails

What do the variables represent?

1. BOARD, X ir Y – the global data base;
2. CX and CY – the production set;
3. Procedure TRY – the control system.

A solution is a sequence of 24 moves $\langle \pi_{i1}, \pi_{i2}, \pi_{i3},..., \pi_{i24}\rangle$, where $\pi_{ij} \in \{\pi_1, \pi_2, \pi_3, \pi_4, \pi_5, \pi_6, \pi_7, \pi_8\}$, i.e. $ij \in \{1, 2, ..., 8\}$.

Two elements are distinguished in *knowledge-based system* architecture:

1. *reasoning*. This stands for a control system. The inference engine does not depend on a domain and can be purchased.
2. *knowledge base*. It is comprised of a global data base and a production set. It depends on the domain and can hardly be purchased. “God is in the detail”.

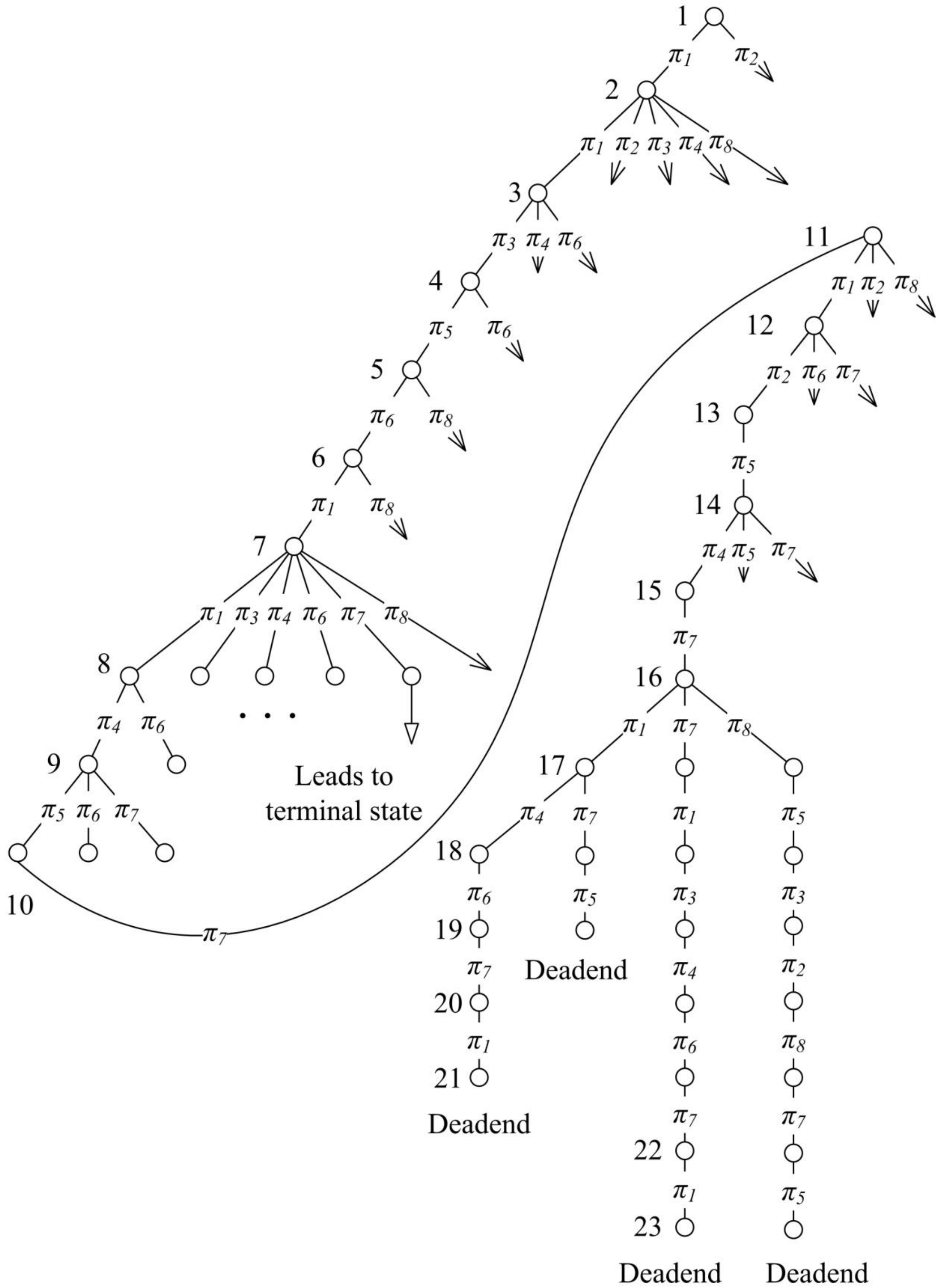


Fig. 2.5. Search tree which leads to the first deadend on a 5×5 board. The state is shown in Fig. 2.4

2.1.Testing

Modify your Knight's Tour solver to print the trace as below.

```
PART 1. Data
    1) Board 5x5.
    2) Initial position X=1, Y=1. L=1.

PART 2. Trace
       1).R1. U=3, V=2. L=2. Free. BOARD[3,2]:=2.
       2)..R1. U=5, V=3. L=3. Free. BOARD[5,3]:=3.
       3)...R1. U=7, V=4. L=4. Out.
       4)...R2. U=6, V=5. L=4. Out.
       5)...R3. U=4, V=5. L=4. Free. BOARD[4,5]:=4.
       6)....R1. U=6, V=6. L=5. Out.
       7)....R2. U=5, V=7. L=5. Out.
       8)....R3. U=3, V=7. L=5. Out.
       9)....R4. U=2, V=6. L=5. Out.
      10)....R5. U=2, V=4. L=5. Free. BOARD[2,4]:=5.
      11).....R1. U=4, V=5. L=6. Thread.
```

And so on until the deadend at L=21. Then backtrack one level, i.e. pop one dot.

```
     ...)....................R1. U=4, V=2. L=21. Free. BOARD[4,2]:=21.
     ...).....................R1. U=6, V=3. L=22. Out.
     ...).....................R2. U=5, V=4. L=22. Thread.
     ...).....................R3. U=3, V=4. L=22. Thread.
     ...).....................R4. U=2, V=3. L=22. Thread.
     ...).....................R5. U=2, V=1. L=22. Thread.
     ...).....................R6. U=3, V=0. L=22. Out.
     ...).....................R7. U=5, V=0. L=22. Out.
     ...).....................R8. U=6, V=1. L=22. Out. Backtrack.
     ...)....................R2. U=3, V=3. L=21. Thread.
     ...)....................R3. U=1, V=3. L=21. Thread.
     ...)....................R4. U=0, V=2. L=21. Out.
     ...)....................R5. U=0, V=0. L=21. Out.
     ...)....................R6. U=1, V=-1. L=21. Out.
     ...)....................R7. U=3, V=-1. L=21. Out.
     ...)....................R8. U=4, V=0. L=21. Out. Backtrack.
     ...)...................R8. U=3, V=2. L=20. Thread. Backtrack
     ...)..................R7. U=3, V=3. L=19. Thread.
     ...)..................R8. U=4, V=4. L=19. Thread. Backtrack.
     ...).................R5. U=2, V=3. L=18. Thread.
     ...).................R6. U=3, V=2. L=18. Thread.
     ...).................R7. U=5, V=2. L=18. Free. BOARD[5,2]:=18.
     ...)..................R1. U=7, V=3. L=19. Out.
```

And so on.

```
   70611)......................R1. U=5, V=5. L=23. Free. BOARD[5,5]:=23.
   70612).......................R1. U=7, V=6. L=24. Out.
   70613).......................R2. U=6, V=7. L=24. Out.
   70614).......................R3. U=4, V=7. L=24. Out.
   70615).......................R4. U=3, V=6. L=24. Out.
   70616).......................R5. U=3, V=4. L=24. Out.
   70617).......................R6. U=4, V=3. L=24. Free. BOARD[4,3]:=24.
   70618)........................R1. U=6, V=4. L=25. Out.
   70619)........................R2. U=5, V=5. L=25. Thread.
   70620)........................R3. U=3, V=5. L=25. Thread.
   70621)........................R4. U=2, V=4. L=25. Thread.
   70622)........................R5. U=2, V=2. L=25. Thread.
   70623)........................R6. U=3, V=1. L=25. Thread.
   70624)........................R7. U=5, V=1. L=25. Free. BOARD[5,1]:=25.
```

```
PART 3. Results
  1) Path is found.
  2) Path graphically

       Y, V
       ^
     5 | 21 16 11  4 23
     4 | 10  5 22 17 12
     3 | 15 20  7 24  3
     2 |  6  9  2 13 18
     1 |  1 14 19  8 25
       -----------------------> X, U

          1  2  3  4  5
```

3. Control with Backtracking and Procedure BACKTRACK

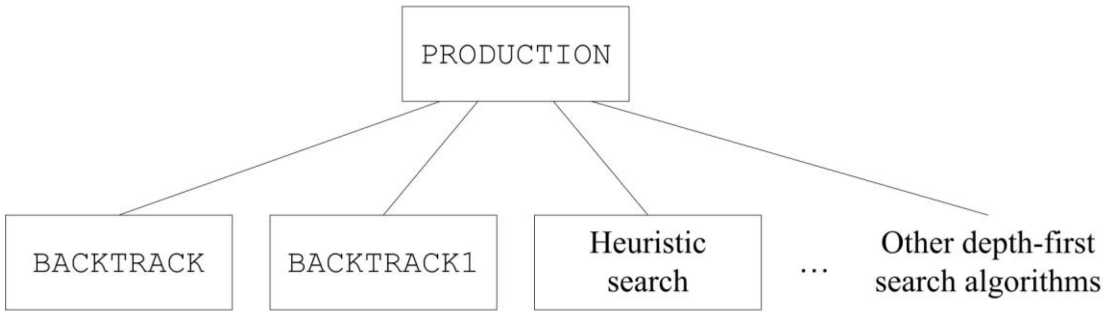


Fig. 3.1. The BACKTRACK procedure is an instance of PRODUCTION

BACKTRACK is demonstrated while solving the 8-queens problem:

1. Global data base – a chessboard.
2. A set of 64 productions $\{\pi_{i,j},\ i,j = 1, 2, ..., 8\}$, where $\pi_{i,j}$ denotes placing a queen into position $[i,j]$. We reduce this set to 8 productions $\{\pi_k,\ k = 1, 2, ..., 8\}$. Each queen i is placed in row i one by one, i=1, 2 and so on until 8. π_k denotes placing a queen into column k. The first queen is placed in row 1 and column *k2*, the second – in row 2 and column *k2* and so on.
3. Control system – procedure BACKTRACK.

An empty board stands for the initial state. A solution is $\langle \pi_{k1}, \pi_{k2}, \ldots, \pi_{k8} \rangle$, where $k1, k2, \ldots, k8 \in \{1, 2, \ldots, 8\}$.

Below we adapt from [Nilsson 1982, Section 2.1]. A general description of the backtracking control strategy was presented earlier as procedure PRODUCTION. Compared with graph-search control regimes, backtracking strategies are typically simpler to implement and require less storage.

A simple recursive procedure captures the essence of the operation of a production system under backtracking control. This procedure, which we call BACKTRACK, takes a single argument, Data, initially set equal to the global database of the production system. Upon successful termination, the procedure returns a list of rules, that, if applied in sequence to the initial database, produces a database satisfying the termination condition. If the procedure halts without finding such a list of rules, it returns FAIL. The BACKTRACK procedure is defined as follows:

```
recursive procedure BACKTRACK(DATA)
{DATA – a current state of the GDB. Returns a list of rules.}
    { 1}       if TERM(DATA) then return NIL; {TERM is a
                    predicate true for arguments that satisfy
                    the termination condition of the production
                    system. Upon successful termination, NIL,
                    the empty list is returned.}
    { 2}       if DEADEND(DATA) then return FAIL; {DEADEND is a
                    predicate true for arguments that are known
                    not to be on a path to a solution. In this
                    case, the symbol FAIL is returned.}
    { 3}       RULES := APPRULES(DATA); {APPRULES is a function
```

```
                    that computes the rules applicable to its
                    argument and orders them (either arbitrari-
                    ly or according to heuristic merit).}
{ 4}       LOOP: if NULL(RULES) then return FAIL;
                     {If there is no (more) rules to apply,
                     the procedure fails.}
{ 5}             R     := FIRST(RULES); {The best of the
                           applicable rules is selected.}
{ 6}             RULES := TAIL(RULES); {The list of
                     applicable rules is diminished by
                     removing the one just selected.}
{ 7}             RDATA := R(DATA); {Rule R is applied to
                                  produce a new database.}
{ 8}             PATH  := BACKTRACK(RDATA); {Recursive call
                                     on the new database.}
{ 9}             if PATH = FAIL then goto LOOP; {If the re-
                     cursive call fails, try another rule.}
{10}       return CONS(R, PATH); {Otherwise, pass the
                     successful list of rules up, by adding
                     R to the front of the list.}
end;
```

3.1. An Example of the DEADEND Predicate

Fig. 3.2. a) The terminal state in the 15-puzzle. b) Four moves – four productions $\langle \pi_1, \pi_2, \pi_3, \pi_4 \rangle$

Fig. 3.3. a) A sample initial state in the 15-puzzle. The number of permutations is 1, an odd number. This is a failure state. b) The number of permutations is 2, an even number. Success can be reached from this state.

4. The 8-Queens Puzzle

Below we follow [Nilsson 1982, Section 2.1]. Suppose the problem of placing 4 queens on a 4×4 chessboard so that none can capture any other. For our global database, we use a 4×4 array with marked cells corresponding to squares occupied by queens. The terminal condition, expressed by the predicate TERM, is satisfied for a database if and only if it has precisely 4 queen marks and the marks correspond to queens located so that they cannot capture each other.

There are many alternative formulations possible for the production rules. A useful one for our purposes involves the following rule schema, for $1 \le i, j \le 4$:

R_{ij} Precondition:
$i = 1$: There are no marks in the array.
$1 < i \le 4$: There is a queen mark in row $i - 1$ of the array.
Effect: Puts a queen mark in row i, column j of the array.

Thus, the first queen mark added to the array must be in row 1, the second must be in row 2, etc.

To use the BACKRACK procedure to solve the 4-queens problem, we have still to specify both the predicate DEADEND and an ordering relation for applicable rules. Suppose we arbitrarily say that R_{ij} is ahead of R_{ik} in the ordering only when $j < k$. The predicate DEADEND might be defined so that it is satisfied for databases where it is obvious that no solution is possible; for example, certainly no solution is possible for any database containing a pair of queen marks in mutually capturing positions. Altogether, the algorithm backtracks 22 times before finding a solution; even the very first rule applied must ultimately be taken back [Nilsson 1982, p. 57–58]. A search tree with 26 edges is shown in see Fig. 5.5.

To see an example of this multilevel backtracking phenomenon, consider using BACKTRACK to solve the 8-queens problem. In this problem, we must place 8 queens on an 8×8 board so that none of them can capture any others. [Nilsson 1982, p. 60]

Placing queens is encoded in the array `X`, where `X[I]=K` denotes placing a queen in row `I` and column `K`. A solution is shown in Fig. 4.1: `X[1]=1`, `X[2]=5`, `X[3]=8`, `X[4]=6`, `X[5]=3`, `X[6]=7`, `X[7]=2`, `X[8]=4`.

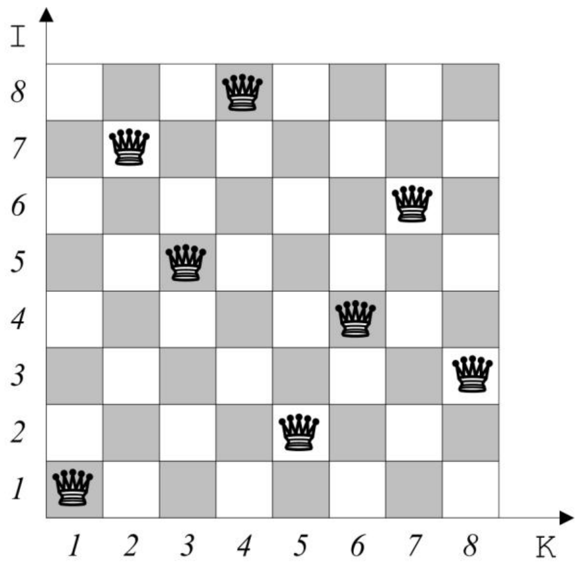


Fig. 4.1. A solution of the 8-queens puzzle

A *brute force* algorithm searches among $8! = 8{\cdot}7{\cdot}6{\cdot}\ldots{\cdot}2{\cdot}1 = 40320$ permutations. $N!$ permutations are in the case of $N{\times}N$ board. $N! = 2^{\alpha \cdot N}$, where α is a real number that depends on N. A solution is as follows:

$$\texttt{PATH}=\langle \pi_1, \pi_5, \pi_8, \pi_6, \pi_3, \pi_7, \pi_2, \pi_4 \rangle$$

Here the `I`-th element is $\pi_{X[I]}$, i.e. `PATH[I] =` $\pi_{X[I]}$, where `I` = 1, 2, …, 8.

The program below follows [Dagienė, Grigas, Augutis, 1986]; see also http://en.wikipedia.org/wiki/Eight_queens_puzzle.

In the case `I=2`, `K=3`, the column is `K=3`, going up diagonal is `K–I = 3–2 = 1`, and moving down diagonal diagonal is `I+K–1 = 2+3–1 = 4`.

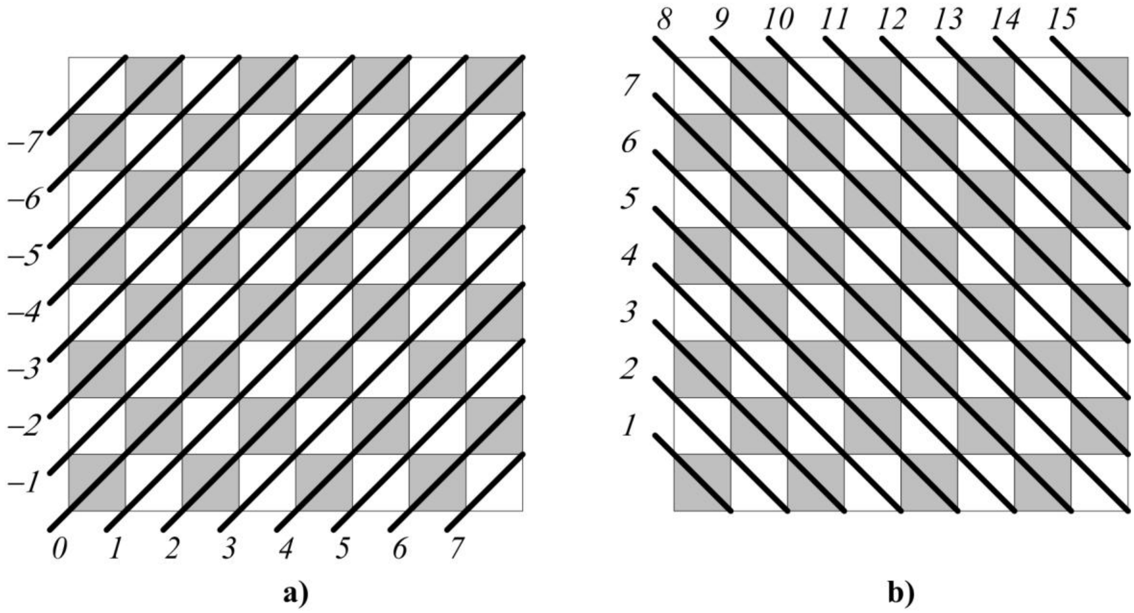


Fig. 4.2. a) Moving up diagonals are numbered from *–7* to *7*.
b) Moving down diagonals are numbered from *1* to *15*

```
program QUEENS;
const   N    =  8;
        NM1  =  7; {N-1}
        N2M1 = 15; {2*N-1}
var
   X      : array [1 .. N] of integer;
   VERT   : array [1 .. N] of boolean;      {Columns}
   UP    : array [-NM1 .. NM1] of boolean;  {Moving up diagonals}
   DOWN   : array [1..N2M1] of boolean;     {Moving down diagonals}
   YES    : boolean;
   I, J   : integer;
   NO_OF_TRIALS : longint;

procedure TRY (I : integer; var YES : boolean);
{Input I – trial's number. Output YES – succeed or no}
   var K : integer;
begin
   K := 0;
   repeat
      K := K + 1; NO_OF_TRIALS := NO_OF_TRIALS + 1;
      if VERT[K] and UP[K - I] and DOWN[I + K - 1]
      then {A column, moving up and moving down diagonals are free}
      begin
         X[I] := K;
         VERT[K] := false; UP[K - I] := false; DOWN[I + K - 1] := false;
         if I < N then
            begin
              TRY(I + 1, YES);
              if not YES
              then {Path further is not found}
                begin
                  VERT[K] := true; UP[K - I] := true;
                  DOWN[I + K - 1] := true; {Free the position}
                end;
           end
         else YES := true;
      end;
   until YES or (K = N);
end; {TRY}

begin {Main program}
   {1. Initialise}
   for J := 1    to N    do VERT[J] := true;
   for J := -NM1 to NM1  do UP[J]  := true;
   for J := 1    to N2M1 do DOWN[J] := true;
   YES := false; NO_OF_TRIALS := 0;
   {2. Invoke the procedure}
   TRY(1, YES);
   {3. Print the board}
   if YES then
     begin
       for I := N downto 1 do
       begin
         for J := 1 to N do
            if X[I] = J then write(1 : 3) else write(0 : 3);
         writeln;  {Write either number 1 or 0 in 3 positions.}
       end;
       writeln('The number of trials: ', NO_OF_TRIALS);
     end
   else writeln('No solutions exist.');
end.
```

5. Heuristic

A heuristic is a rule of thumb; see [Russell, Norvig, 2003, p. 94]. The term "heuristic" see Oxford English Dictionary http://www.oed.com/:

adjective

a. Serving to find out or discover…

c. Under an 'heuristic' programming procedure the computer searches through a number of possible solutions at each stage of the programme, it evaluates a 'good' solution for this stage and then proceeds to the next stage. Essentially heuristic programming is similar to the problem solving techniques by trial and error methods which we use in everyday life. (1964 T. W. McRae)

noun

b. A process that may solve a given problem, but offers no guarantees of doing so, is called a heuristic for that problem. *Ibid.* For conciseness, we will use 'heuristic' as a noun synonymous with 'heuristic process'. (1957 A. Newell et al.)

5.1.Heuristic Search in N-queens Problem

As an example, we examine the heuristic which is proposed in [Nilsson 1982, Section 2.1, p. 58] to solve the 4-queens problem. A solution exists; see Fig. 5.1. The problem can be generalised to N-queens on an N×N board. The previous section was devoted to obtaining a solution with the BACKTRACK procedure and involved no heuristics.

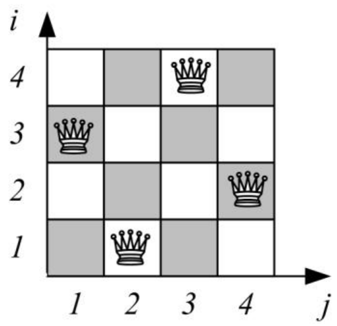


Fig. 5.1. A solution on a 4×4 board

A more efficient algorithm (with less backtracking) can be obtained if we use a more informed rule ordering. One simple, but useful ordering for this problem involves using the function *diag(i,j)*, defined to be the length of the longest diagonal passing through cell *(i,j)*. Let π_{ij} be ahead of π_{mn} in the ordering if $diag(i,j) < diag(m,n)$. (For equal values of *diag*, use the same orders as before.) Below we verify that this ordering scheme solves the 4-queens problem with only 2 backtracks. [Nilsson 1998, p. 58]

A corresponding search tree consists of 6 edges and is shown in Fig. 5.4. Let us examine the heuristics in more details. Hence, the heuristic is formalised

$$\pi_{i,j} < \pi_{i,k}, \text{ if } diag(i,j) < diag(i,k),$$

where *diag(i,j) = max(up(i,j), down(i,j))*. First,

$$diag(1,1)=max(4,1)=4$$

Similarly, other row 1 diagonals *diag(1,j), j=1,…,4* are calculated:

$$diag(1,1)=4;\ diag(1,2)=max(3,2)=3;\ diag(1,3)=max(2,3)=3;\ diag(1,4)=max(1,4)=4$$

Hence

$$diag(1,2) \le diag(1,3) \le diag(1,1) \le diag(1,4)$$

Row 1 productions are ordered:

$$\pi_{1,2} < \pi_{1,3} < \pi_{1,1} < \pi_{1,4}$$

Row 2 diagonals, t. y. *diag(2,j), j=1,…,4*:

$$diag(2,1)=max(3,1)=3;\ diag(2,2)=max(4,3)=4;\ diag(2,3)=max(3,4)=4;$$
$$diag(2,4)=max(2,3)=3$$

Row 2 productions are ordered:

$$\pi_{2,1} < \pi_{2,4} < \pi_{2,2} < \pi_{2,3}$$

Row 3 diagonals *diag(3,j), j=1,…,4*:

$$diag(3,1)=max(2,3)=3;\ diag(3,2)=max(3,4)=4;\ diag(3,3)=max(4,3)=4;$$
$$diag(3,4)=max(3,2)=3$$

Row 3 productions are ordered:

$$\pi_{3,1} < \pi_{3,4} < \pi_{3,2} < \pi_{3,3}$$

Row 4 diagonals *diag(4,j), j=1,…,4*:

$$diag(4,1)=max(1,4)=4;\ diag(4,2)=max(2,3)=3;\ diag(4,3)=max(3,2)=3;$$
$$diag(4,4)=max(4,1)=4$$

Row 4 productions are ordered:

$$\pi_{4,2} < \pi_{4,3} < \pi_{4,1} < \pi_{4,4}$$

To sum up, *diag(i,j)* values are shown in Fig. 5.2.

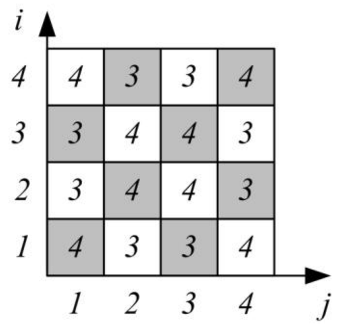


Fig. 5.2. *diag(i,j)* on a 4×4 board

Hence, the ordering of production rules is as follows:

$$\begin{array}{ll} \pi_{1,2} < \pi_{1,3} < \pi_{1,1} < \pi_{1,4} & \text{– row 1} \\ \pi_{2,1} < \pi_{2,4} < \pi_{2,2} < \pi_{2,3} & \text{– row 2} \\ \pi_{3,1} < \pi_{3,4} < \pi_{3,2} < \pi_{3,3} & \text{– row 3} \\ \pi_{4,2} < \pi_{4,3} < \pi_{4,1} < \pi_{4,4} & \text{– row 4} \end{array}$$

Table 5.3 . Ordering productions on a 4×4 board

A search tree is shown in Fig. 5.4.

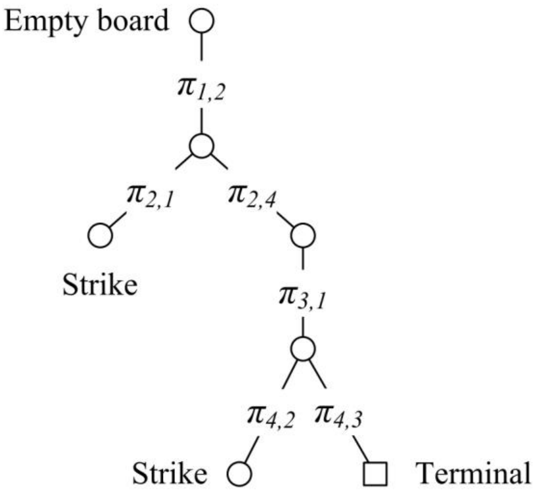


Fig. 5.4. Heuristic search produces a tree of 6 edges. Hence, only 2 backtracks

A search with no heuristic produces a tree shown in Fig. 5.5 ($\pi_{i,1} < \pi_{i,2} < \pi_{i,3} < \pi_{i,4}$).

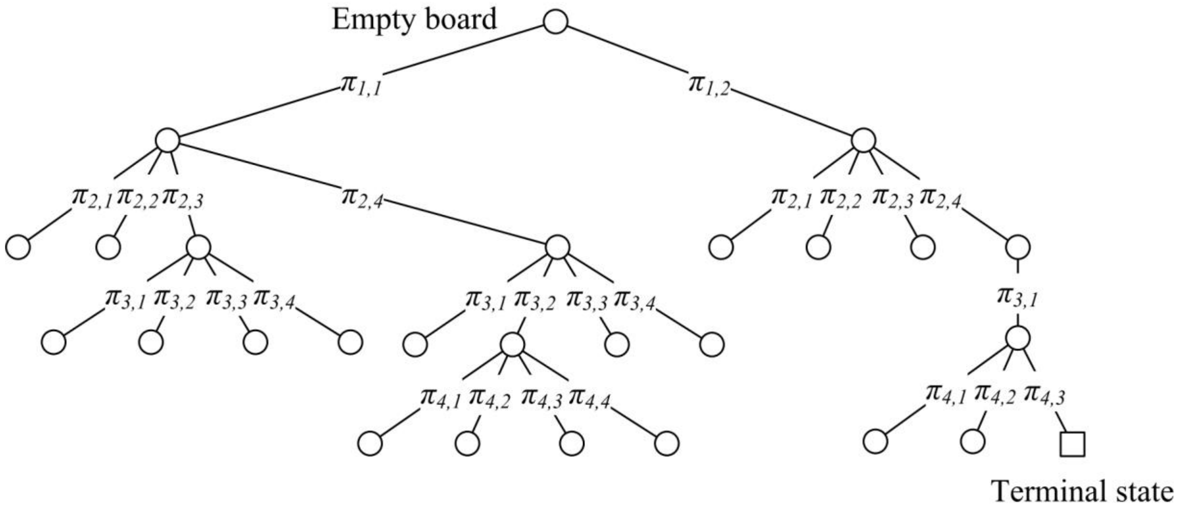


Fig. 5.5. Non-heuristic search on a 4×4 board. 26 edges is significantly more than 6 edges in the case of heuristic search

Heuristic search on a 8×8 board consists of 204 states. First 6 moves are shown in Fig. 5.6. Further moves require backtrack.

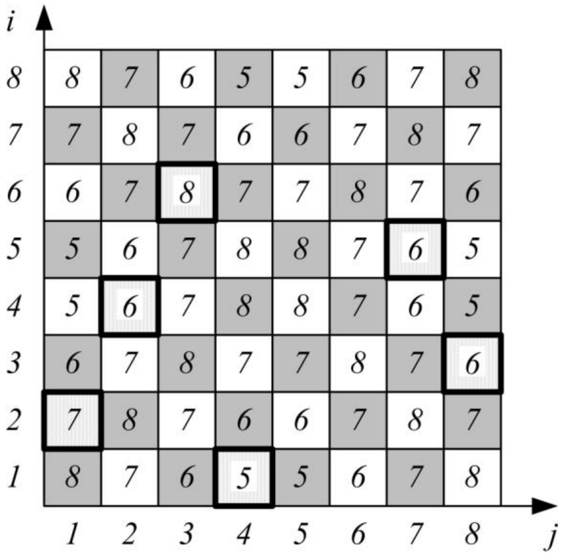


Fig. 5.6. Ordering productions on a 8×8 board

5.2.Knight's Move Heuristic in the N-queens Problem

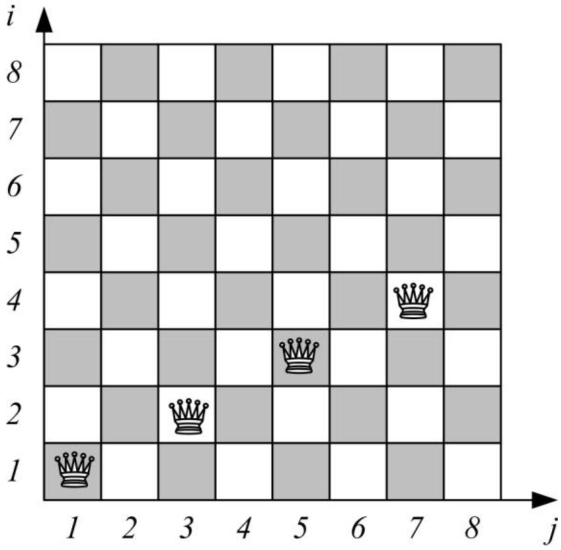


Fig. 5.7. Knight's move

$$\pi_1 = \begin{cases} \bullet\ [i=1, j=2] \text{ if } N \neq 6n+3 \\ \bullet\ [i=1, j=4] \text{ if } N = 6n+3 \end{cases}$$

$\pi_{\check{z}}$ =

- [$i+1,j+2$], if within the board. Here [i,j] is the last queen in the previous row
- [$i+1,j=1$], if [$i+1,j+2$] is out of the board

π_k = [$i,j+1$] in a cycle, i.e. [$i,j=1$] if $j+1 > N$ (out of the board);
In other words, [$i,(j \bmod N) + 1$], where [i,j] is the last attempt in this row.

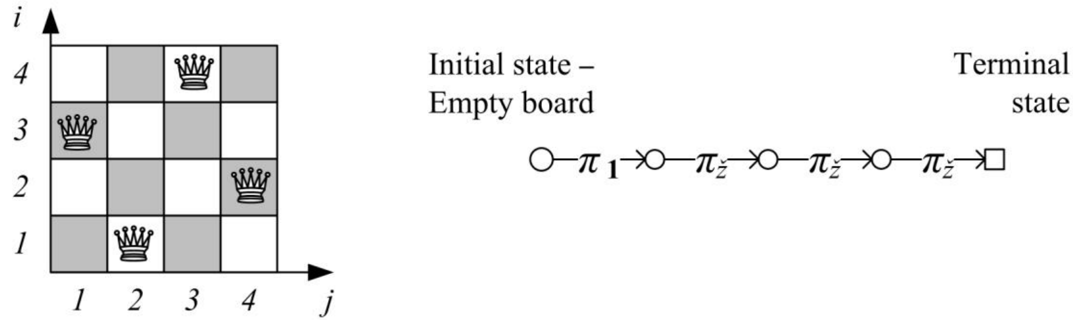


Fig. 5.8. Knight’s move on a 4×4 board

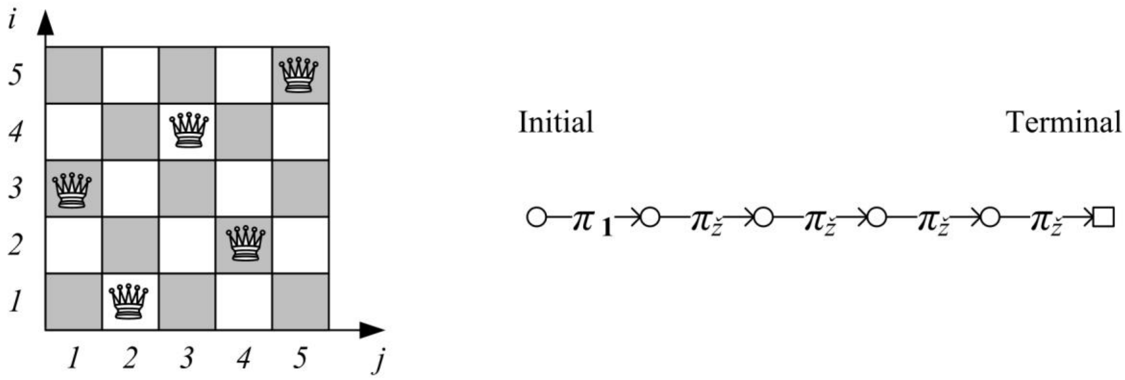


Fig. 5.9. Knight’s move on a 5×5 board

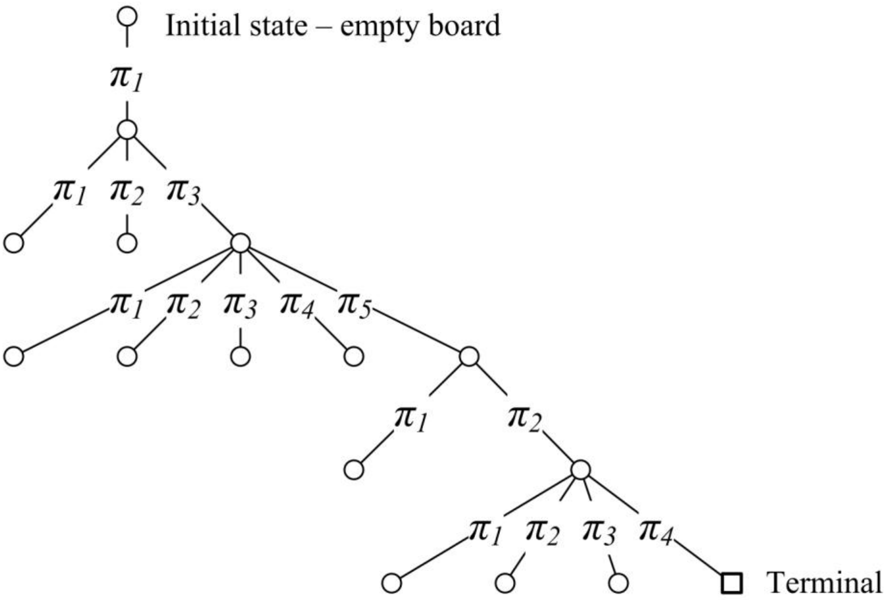


Fig. 5.10. Knight’s move on a 6×6 board

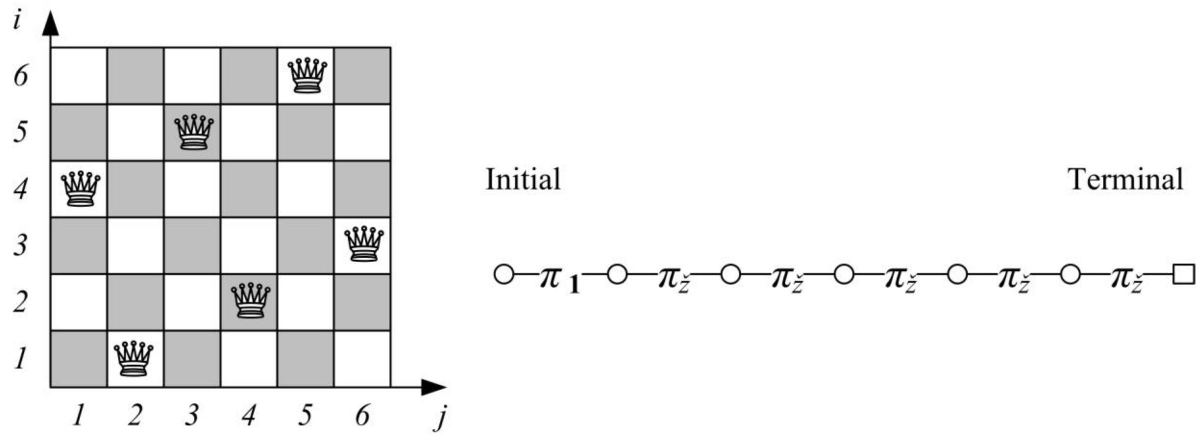


Fig. 5.11. Knight’s move on a 6×6 board

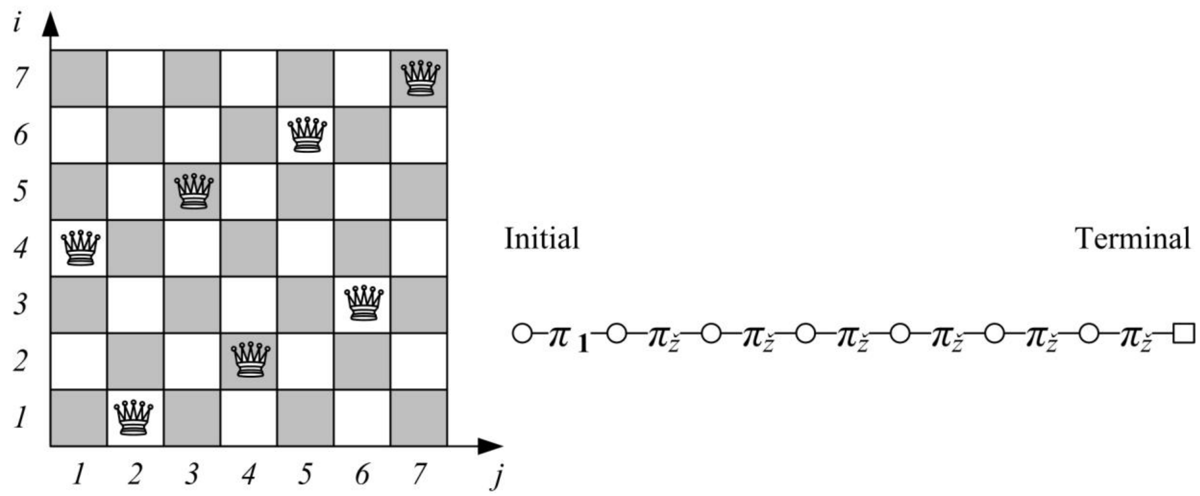


Fig. 5.12. Knight’s move on a 7×7 board

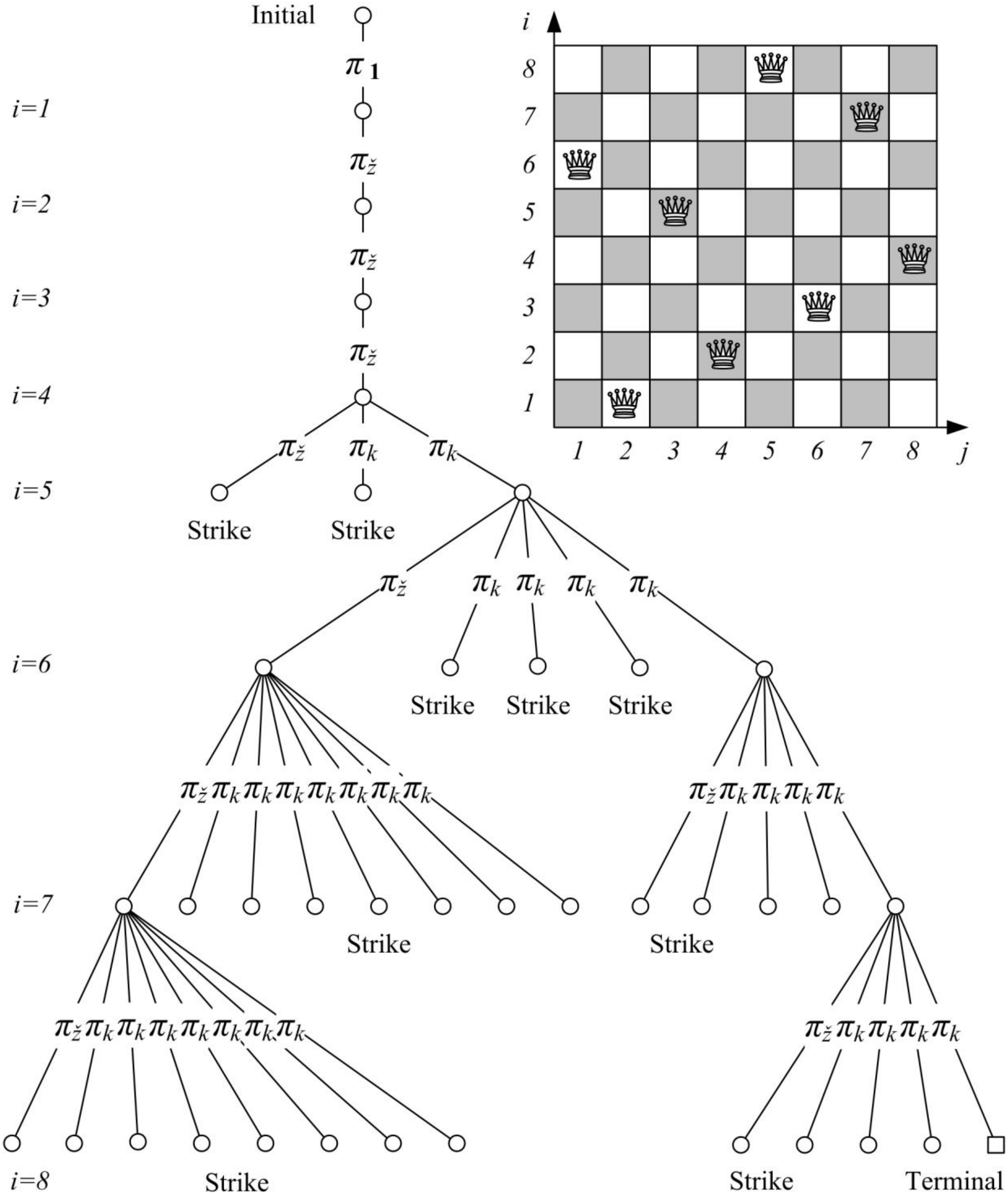


Fig. 5.13. Knight’s move on a 8×8 board. Search space consists of 38 states. This is significantly smaller number than 204 states in the case of shortest diagonal heuristics and 876 state in brute force search

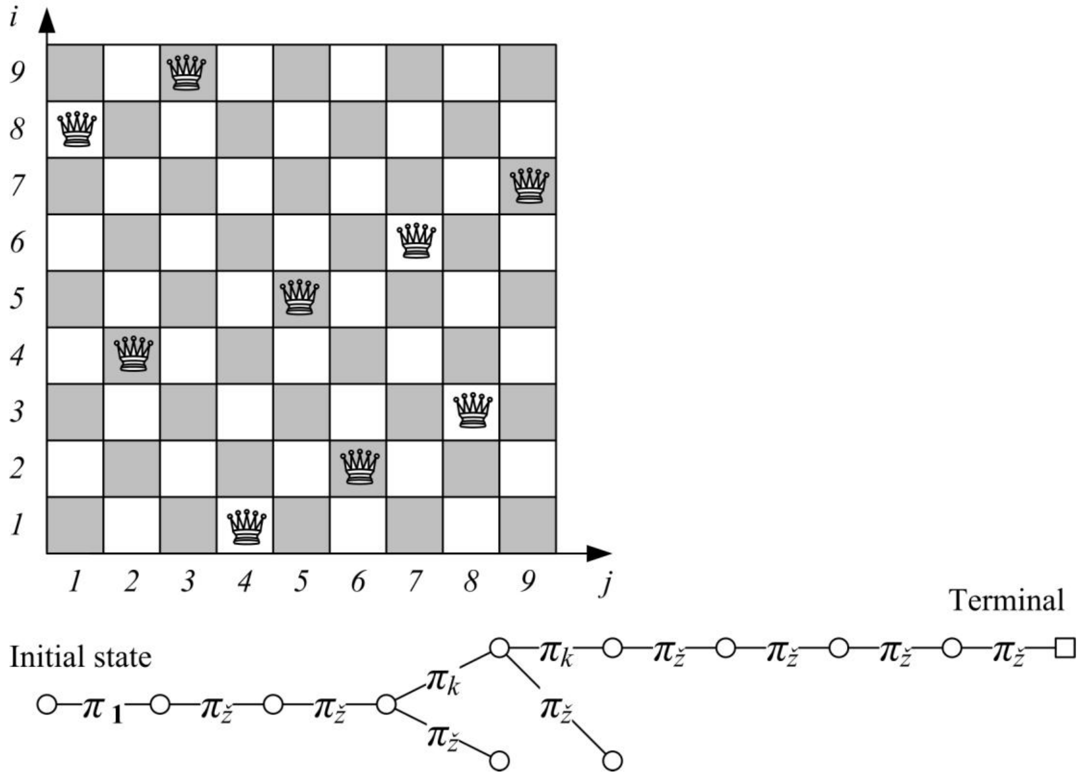


Fig. 5.14. Knight’s move on a 9×9 board

N	**Number of states in a search space**		
	Brute force	**Shortest diagonal**	**Knight’s move**
4	26	6	4
5	15	15	5
6	171	69	6
7	42	87	7
8	876	204	38
9	333	874	11
10	975	437	10
11	517	200	11
12	3066	297	12
13	1365	684	13
14	26495	1742	300
15	20280	487	17
16	160712	111	16
17	91222	294	17
18	743229	7130	18
19	48184	24714	19
20	3992510	918	372
21	179592	48222	23
22	38217905	40744	22
23	584591	10053	23
24	9878316	5723	24

25	1216775	2887	25
26	10339849	265187	2196
27	12263400	986476	29
28	84175966	2602283	28
29	44434525	1261296	29
30	1692888135	52601	30
31	Many	1850449	31
32	Many	2804692	2866
33	Many	2582396	35
34	Many	35784	34
35	Many	110473	35
36	Many	19605979	36
37	Many	135980	37
38	Many	642244758	30532
39	Many	193745	41
40	Many	4685041	40

Table 5.15. Number of states in a search space

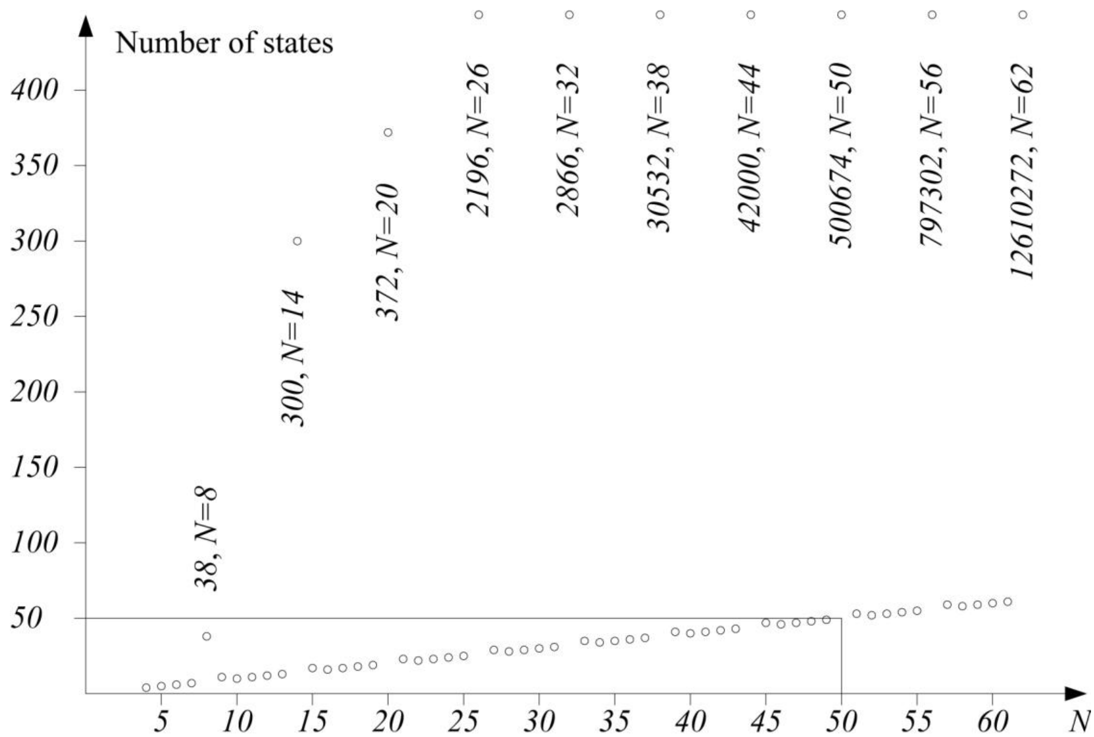


Fig. 5.16. Linear dependence – no backtrack. Exceptions: $N \in \{8, 14, 20, 26, 32, 38,$ etc.$\}$

6. Procedure BACKTRACK1 – a Loop-avoiding Algorithm

Below we follow [Nilsson 1982 Section 2.1, p. 58–61]. The BACKTRACK procedure may run into an infinite loop. This is demonstrated with a LABYRINTH depth-first search program in the next section. Therefore we need a slightly more complex algorithm to avoid cycles. All databases on a path back to the initial one must be checked to insure that none are revisited. In order to implement this backtracking strategy as a recursive procedure, the entire chain of databases must be an argument of the procedure. Again, practical implementations of AI backtracking production systems use various techniques to avoid the need for explicitly listing all of these databases in their entirety.

BACKTRACK1 takes a list of databases as its argument; when first called, this list contains the initial database as its single element. Upon successful termination, BACKTRACK1 returns a sequence of rules that can be applied to the initial database to produce one that satisfies the termination condition.

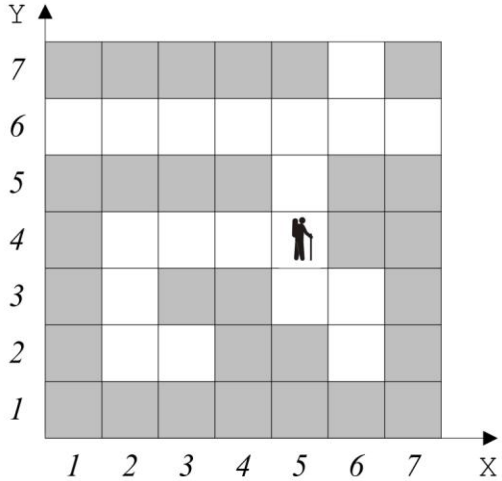
```
procedure BACKTRACK1(DATALIST); {Returns a list of rules.}
{ 1}  DATA := FIRST(DATALIST); {DATALIST - is a list of
              all databases on a path back to the initial one.
              DATA is the most recent one produced.}
{ 2}  if MEMBER(DATA, TAIL(DATALIST)) then return FAIL; {The
          procedure fails if it revisits an earlier database.}
{ 3}  if TERM(DATA) then return NIL;
{ 4}  if DEADEND(DATA) then return FAIL;
{ 5}  if LENGTH(DATALIST) > BOUND then return FAIL;
             {The procedure fails if too many rules have
              been applied. BOUND is a global variable
              specified before the procedure is first called.}
{ 6}  RULES := APPRULES(DATA);
{ 7}  LOOP:    if NULL(RULES) then return FAIL;
{ 8}    R     := FIRST(RULES);
{ 9}    RULES := TAIL(RULES);
{10}    RDATA := R(DATA);
{11}    RDATALIST := CONS(RDATA, DATALIST); {The list of data-
            bases visited so far is extended by adding RDATA.}
{12}    PATH := BACKTRACK1(RDATALIST);
{13}    if PATH = FAIL then goto LOOP;
{14}  return CONS(R, PATH);
end procedure;
```

The 8-puzzle example of backtracking that is examined in a further section uses BOUND = 7 and also checks to see if a tile configuration had been visited previously. Note that the recursive algorithm does not remember *all* databases that it visited previously. Backtracking involves “forgetting” all databases whose paths lead to failures. The algorithm remembers only those databases on the *current* path back to the initial one.

The backtracking strategies just described “fail back” one level at a time. If a level *n* recursive call of BACKTRACK fails, control returns to level *n–1* where another rule is tried. But sometimes the reason, or *blame*, for the failure at level *n* can be traced to rule choices made many levels above. In these cases it would be obviously futile to try another rule choice at level *n–1*; predictably, any such choice there would again lead to a failure. What is needed, then, is a way to jump several levels at a time, all the way back to one where a different rule choice will make a useful difference.

7. The Labyrinth Problem. Depth-first Search

This section demonstrates the BACKTRACK1 procedure on a labyrinth. The labyrinth is represented as a two-dimensional array. Wall cells are marked 1 and free ones 0. Agent position is marked 2; see Fig. 7.1. (See also http://en.wikipedia.org/wiki/Labyrinth; labyrinth variation maze see http://en.wikipedia.org/wiki/Maze_solving_algorithm.)



Y							
7	1	1	1	1	1	0	1
6	0	0	0	0	0	0	0
5	1	1	1	1	0	1	1
4	1	0	0	0	2	1	1
3	1	0	1	1	0	0	1
2	1	0	0	1	1	0	1
1	1	1	1	1	1	1	1
	1	2	3	4	5	6	7 X

Fig. 7.1. A sample labyrinth and its representation with a two-dimensional array LAB. Wall cells are marked 1 and free ones 0. Three exits are LAB[1,6], LAB[7,6] and LAB[6,7]. The agent starts from the position X=5, Y=4 marked 2, i.e. LAB[5,4] = 2

The agent can move in four directions $\{\pi_1, \pi_2, \pi_3, \pi_4\}$; see Fig. 7.2. A BACKTRACK1 program is presented below. It finds a path.

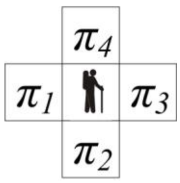


Fig. 7.2. Agent's four moves

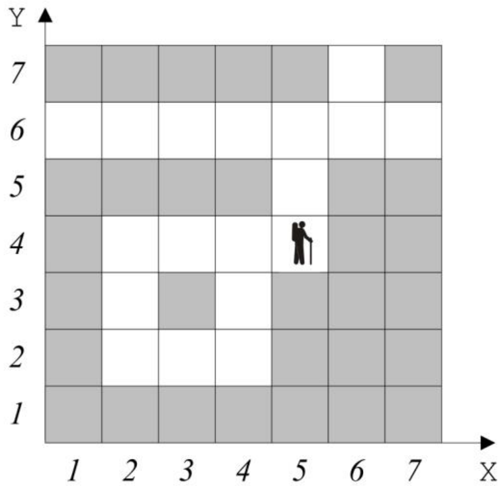


Fig. 7.3. A sample labyrinth which leads the agent into infinite cycle once BACKTRACK is used. Agent is in the position X=5, Y=4

```
program LABYRINTH; {BACKTRACK1, i.e. depth-first, no infinite cycle}
const   M = 7; N = 7;                          {Dimensions}
var LAB    : array[1..M, 1..N] of integer; {Labyrinth}
    CX, CY : array[1..4] of integer; {4 production - shifts in X and Y}
    L,         {Move's number. Starts from 2. Visited positions are marked}
    X, Y,      {Agent's initial position}
    I, J,      {Loop variables}
    TRIAL : integer; {Number of trials. To compare effectiveness}
    YES   : boolean; {true - success, false - failure}

procedure TRY(X, Y : integer; var YES : boolean);
  var   K,               {The number of a production rule}
        U, V : integer;  {Agent's new position}
begin {TRY}
{K1} if  (X = 1) or (X = M) or (Y = 1) or (Y = N)
     then YES := true {TERM(DATA} = true on the boarder}
     else
     begin K := 0;
{K2}   repeat K := K + 1; {Next rule. Loop over production rules}
{K3}     U := X + CX[K]; V := Y + CY[K]; {Agent's new position }
{K4}     if LAB[U, V] = 0                {If a cell is free}
         then
         begin TRIAL := TRIAL + 1;   {Number of trials}
{K5}       L := L + 1; LAB[U,V] := L;{Marking the cell}
{K6}       TRY(U, V, YES);           {Recursive call}
           if not YES             {If failure}
{K7}       then begin
{K8}              LAB[U,V] := -1; {then mark. (0 in case of BACKTRACK)}
                  L := L - 1;
                end;
         end;
       until YES or (K = 4);
     end;
end; {TRY}

begin {main program}
  {1. Reading the labyrinth}
  for J := N downto 1 do
  begin
     for I := 1 to M do read(LAB[I,J]);
     readln;
  end;
  {2. Reading agent's position}
  read(X, Y); L := 2;  LAB[X,Y] := L;
  {3. Forming four production rules}
  CX[1] := -1; CY[1] :=  0;  {Go West.      4   }
  CX[2] :=  0; CY[2] := -1;  {Go South.   1 * 3 }
  CX[3] :=  1; CY[3] :=  0;  {Go East.      2   }
  CX[4] :=  0; CY[4] :=  1;  {Go North.         }
  {4. Initialising variables}
  YES := false;  TRIAL := 0;
  {5. Invoking the BACKTRACK1 procedure}
  TRY(X, Y, YES);
  if YES
  then writeln('Path exists');          {Please also print the path found.}
  else writeln('Path does not exist');  {No paths exist.}
end.
```

Three variations:

V1) LAB[U,V] := –1. This is BACKTRACK_WITH_CLOSED.

V2) LAB[U,V] := 0 and LAB[U,V]:=L above. This is BACKTRACK1. Two routes around an island, but no cycles.

V3) LAB[U,V] := 0 and no LAB[U,V]:=L. This is a classical BACKTRACK. Infinite cycle.

Search tree is shown in Fig. 7.4. Note three variations V1, V2 and V3. Following is the ordering according to time efficiency, where < denotes ordering:

BACKTRACK < *BACKTRACK1* < BACKTRACK_WITH_CLOSED.

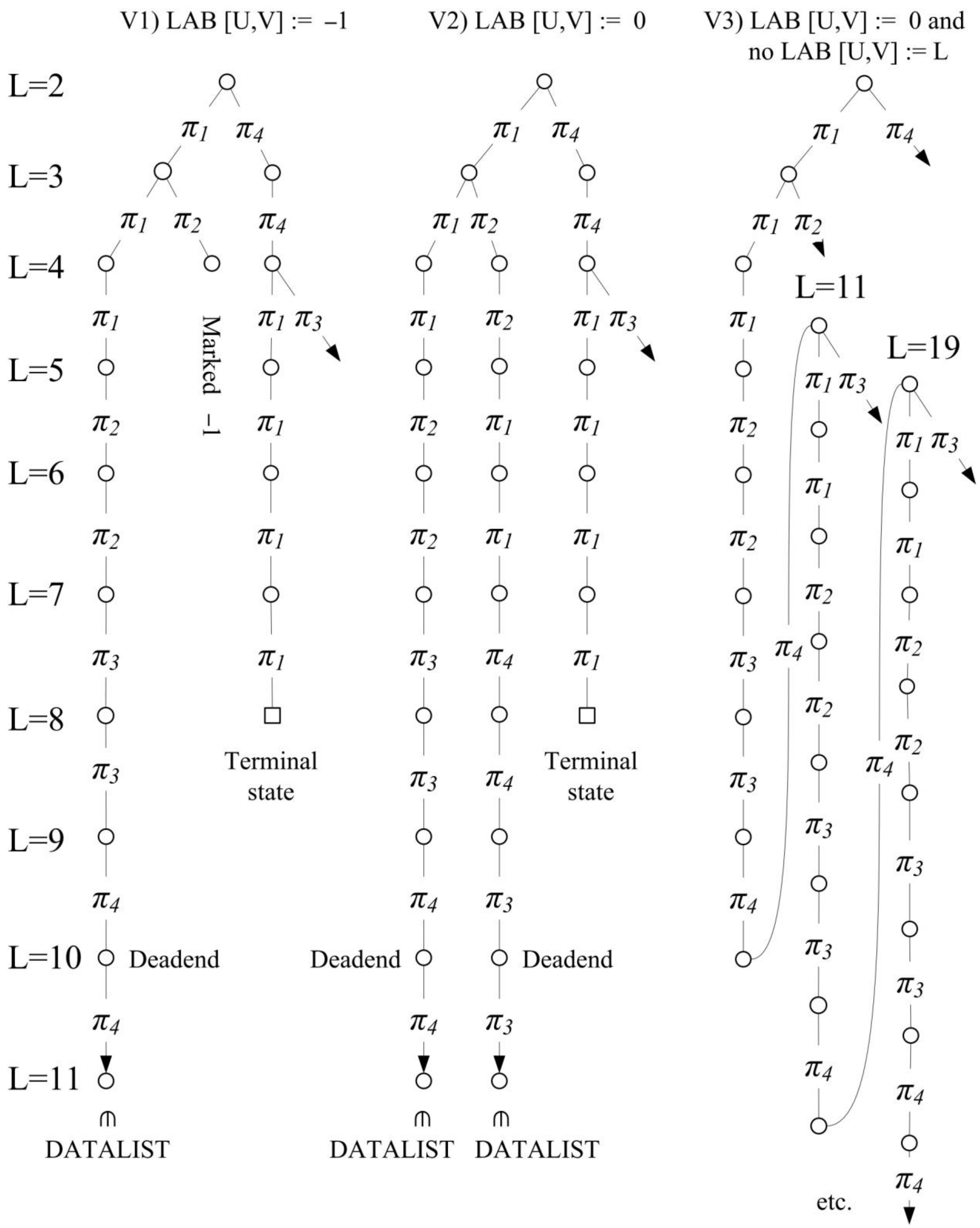


Fig. 7.4. Three variations of search tree in the labyrinth which is shown in Fig. 7.3. V1) LAB[U,V] := –1. BACKTRACK_WITH_CLOSED. V2) LAB[U,V] := 0 and LAB[U,V] := L. This is BACKTRACK1, i.e. BACKTRACK with DATALIST as a stack. An "island" is searched from two sides. No infinite cycles. V3) LAB[U,V] := 0 and no LAB[U,V] := L. This is classical BACKTRACK and runs into an infinite loop

7.1.Testing Labyrinth Depth-first

Modify your labyrinth depth-first solver to test it with the labyrinth in Fig. 7.3 and to print the trace as follows. The second test: build your labyrinth 20x15 with deadends (marked –1) and path length over 100 productions.

```
PART 1. Data
  1.1. Labyrinth

    Y, V
  7 |   1   1   1   1   1   0   1
  6 |   0   0   0   0   0   0   0
  5 |   1   1   1   1   0   1   1
  4 |   1   0   0   0   0   1   1
  3 |   1   0   1   0   1   1   1
  2 |   1   0   0   0   1   1   1
  1 |   1   1   1   1   1   1   1
    ------------------------------> X, U

        1   2   3   4   5   6   7

  1.2. Initial position X=5, Y=4. L=2.

PART 2. Trace
   1).R1. U=4, V=4. Free. L:=L+1=3. LAB[4,4]:=3.
   2)..R1. U=3, V=4. Free. L:=L+1=4. LAB[3,4]:=4.
   3)...R1. U=2, V=4. Free. L:=L+1=5. LAB[2,4]:=5.
   4)....R1. U=1, V=4. Wall.
   5)....R2. U=2, V=3. Free. L:=L+1=6. LAB[2,3]:=6.
   6).....R1. U=1, V=3. Wall.
   7).....R2. U=2, V=2. Free. L:=L+1=7. LAB[2,2]:=7.
   8)......R1. U=1, V=2. Wall.
   9)......R2. U=2, V=1. Wall.
  10)......R3. U=3, V=2. Free. L:=L+1=8. LAB[3,2]:=8.
  11).......R1. U=2, V=2. Thread.
  12).......R2. U=3, V=1. Wall.
  13).......R3. U=4, V=2. Free. L:=L+1=9. LAB[4,2]:=9.
  14)........R1. U=3, V=2. Wall.
  15)........R2. U=4, V=1. Wall.
  16)........R3. U=5, V=2. Wall.
  17)........R4. U=4, V=3. Free. L:=L+1=10. LAB[4,3]:=10.
  18).........R1. U=3, V=3. Wall.
  19).........R2. U=4, V=2. Thread.
  20).........R3. U=5, V=3. Wall.
  21).........R4. U=4, V=4. Thread.
     .........Backtrack from X=4, Y=3, L=10. LAB[4,3]:=-1. L:=L-1=9.
     ........Backtrack from X=4, Y=2, L=9. LAB[4,2]:=-1. L:=L-1=8.
  22).......R4. U=3, V=2. Thread.
     .......Backtrack from X=3, Y=2, L=8. LAB[3,2]:=-1. L:=L-1=7.
  23)......R4. U=3, V=3. Wall.
     ......Backtrack from X=2, Y=2, L=7. LAB[2,2]:=-1. L:=L-1=6.
  24).....R3. U=3, V=3. Wall.
  25).....R4. U=2, V=4. Thread.
     .....Backtrack from X=2, Y=3, L=6. LAB[2,3]:=-1. L:=L-1=5.
  26)....R3. U=3, V=4. Thread.
  27)....R4. U=2, V=4. Wall.
     ....Backtrack from X=2, Y=4, L=5. LAB[2,4]:=-1. L:=L-1=4.
  28)...R2. U=3, V=3. Wall.
  29)...R3. U=4, V=4. Thread.
  30)...R4. U=3, V=5. Wall.
```

```
     ...Backtrack from X=3, Y=4, L=4. LAB[3,4]:=-1. L:=L-1=3.
  31)..R2. U=4, V=3. Thread.
..32)  R3. U=5, V=4. -1.
  33)..R4. U=4, V=5. Wall.
     ..Backtrack from X=4, Y=4, L=3. LAB[4,4]:=-1. L:=L-1=2.
  34).R2. U=5, V=3. Wall.
  35) R3. U=6, V=4. Wall.
  36) R4. U=5, V=5. Free. L:=L+1=3. LAB[5,5]:=3.
  37)  R1. U=4, V=5. Wall.
..38)..R2. U=5, V=4. Thread.
  39)..R3. U=6, V=5. Wall.
  40)..R4. U=5, V=6. Free. L:=L+1=4. LAB[5,6]:=4.
  41)...R1. U=4, V=6. Free. L:=L+1=5. LAB[4,6]:=5.
  42)....R1. U=3, V=6. Free. L:=L+1=6. LAB[3,6]:=6.
  43)     R1. U=2, V=6. Free. L:=L+1=7. LAB[2,6]:=6.
  44)......R1. U=1, V=6. Free. L:=L+1=8. LAB[1,6]:=8. Terminal node.

PART 3. Results

  3.1. Path found.

  3.2. Path graphics

    Y, V
  7 |   1   1   1   1   1   0   1
  6 |   8   7   6   5   4   0   0
  5 |   1   1   1   1   3   1   1
  4 |   1  -1  -1  -1   2   1   1
  3 |   1  -1   1  -1   1   1   1
  2 |   1   0  -1  -1   1   1   1
  1 |   1   1   1   1   1   1   1
    -------------------------------> X, U

        1   2   3   4   5   6   7

  3.2. Path productions R4, R4, R1, R1, R1, R1.

  3.3. Path nodes [X=5,Y=4], [X=5,Y=5], [X=5,Y=6], [X=4,Y=6], [X=3,Y=6],
[X=2,Y=6], [X=1,Y=6].
```

8. Labyrinth Breadth-first Search

Suppose a labyrinth and four productions {$\pi_1, \pi_2, \pi_3, \pi_4$}; see Fig. 8.1. The breadth-first search tree is shown in Fig.8.2. The same tree in layers is shown in Fig. 8.3.

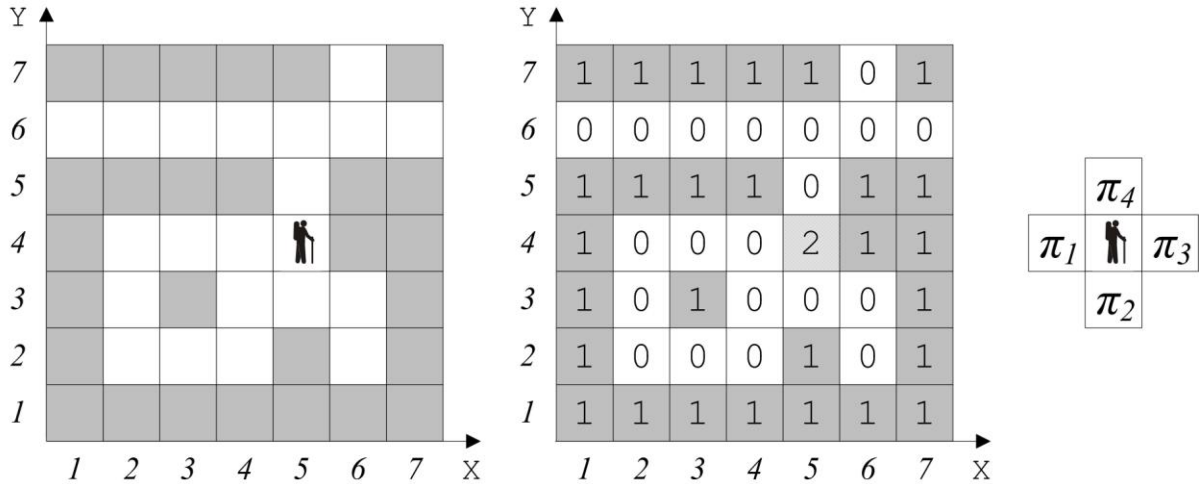


Fig. 8.1. A sample labyrinth and its representation to demonstrate breadth-first search. Agent starts from the position X=*5*, Y=*4*

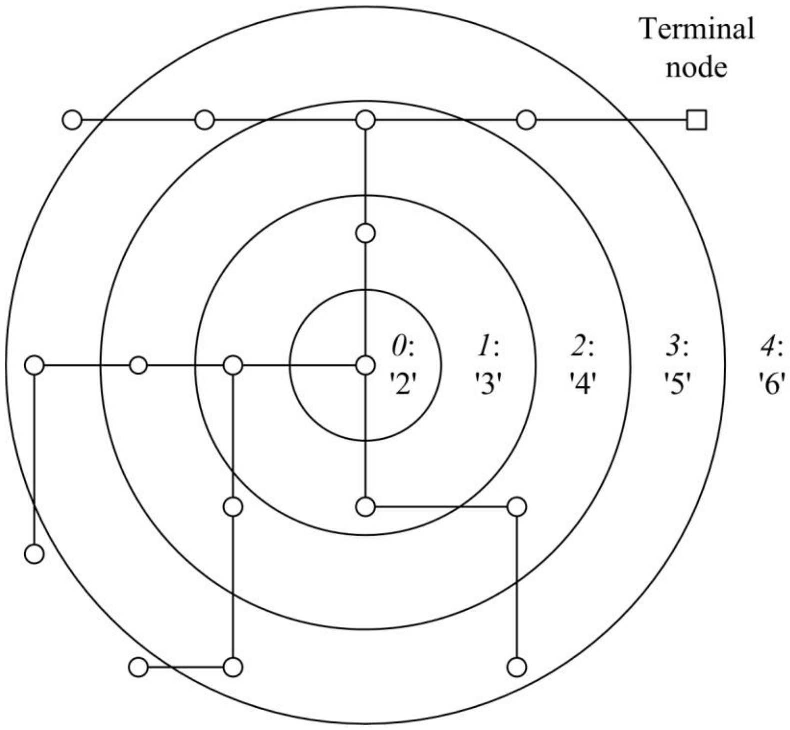


Fig. 8.2. Breadth-first search tree. The nodes of each consecutive layer ("wave") are shown in a separate ring

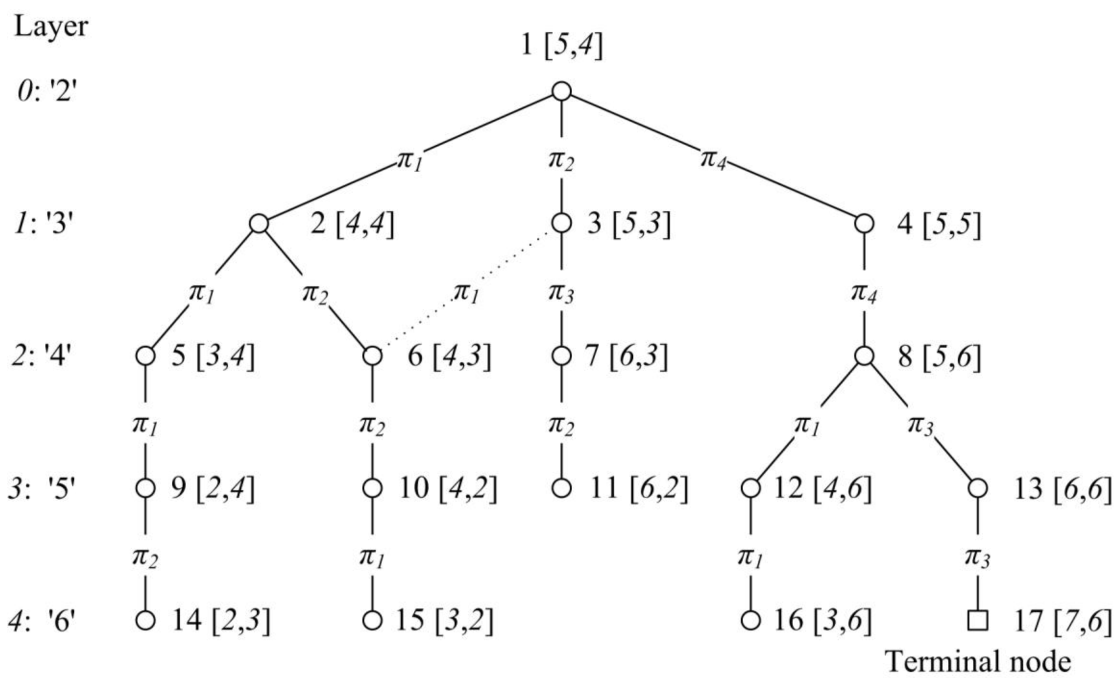


Fig. 8.3. A search tree with nodes sorted by layers (“waves”)

The global database when the procedure is completed is shown in Fig. 8.4 a. Array representation after the path is collected in order to print it is shown in Fig. 8.4 b.

Y \ X	1	2	3	4	5	6	7
7	1	1	1	1	1	0	1
6	0	0	6	5	4	5	6
5	1	1	1	1	3	1	1
4	1	5	4	3	2	1	1
3	1	6	1	4	3	4	1
2	1	0	6	5	1	5	1
1	1	1	1	1	1	1	1

Y \ X	1	2	3	4	5	6	7
7	1	1	1	1	1	0	1
6	0	0	0	0	4	5	6
5	1	1	1	1	3	1	1
4	1	0	0	0	2	1	1
3	1	0	1	0	0	0	1
2	1	0	0	0	1	0	1
1	1	1	1	1	1	1	1

Fig. 8.4. a) The final state of the global database – the procedure is completed. b) Array representation after the path is collected

```
program LABYRINTH_BREADTH_FIRST (input, output);
  const
    M = 7; N = 7; {The dimensions of the labyrinth.}
    MN = 49;      {The number of cells M*N.        }
  var
    LAB, LABCOPY : array[1..M, 1..N] of integer; {Labyrinth and its copy.}
    CX, CY : array[1..4] of integer;  {4 production rules.}
    FX, FY : array[1..MN] of integer; {The "front" to store opened nodes.}
    CLOSE,                            {The counter for a closed node.}
```

```
    NEWN,                              {The counter for an opened node.}
    K,                                 {The counter for a production.}
    X, Y,                              {The starting position of the agent.}
    U, V, I, J : integer;
    YES : boolean;

procedure BACK(U, V : integer); {Collect the path from the exit to start.}
    {INPUT: 1) U, V – the coordinates of the exit, and 2) global LABCOPY.
     OUTPUT: LAB.}
  var K, UU, VV : integer;
begin {BACK}
  LAB[U,V] := LABCOPY[U,V]; {The exit position is marked.}
  K := 0;
  repeat {The search within 4 productions. Search for cell LABCOPY[UU,VV]
          with the mark which is 1 less than LABCOPY[U,V].}
    K := K + 1; UU := U + CX[K]; VV := V + CY[K];
    if (1 <= UU) and (UU <= M) and (1 <= VV) and (VV <= N)
    then {Inside the boarders}
        if LABCOPY[UU,VV] = LABCOPY[U,V] – 1
        then
        begin
          LAB[UU,VV] := LABCOPY[UU,VV] ; {Marking a cell in LAB.}
          U := UU; V := VV; K := 0;     {Swapping the variables.}
        end;
  until LABCOPY[U, V] = 2;
end; {BACK}

begin {Main program}
  { 1. Reading the labyrinth.}
  for J := N downto 1 do
  begin
    for I := 1 to M do
    begin read(LAB[I,J]);
          LABCOPY[I,J]:= LAB[I,J];
    end
    readln;
  end;
  { 2. Reading the starting position of the agent.}
  read(X,Y); LABCOPY[X,Y]:=2;

  { 3. Initialising 4 production rules}
  CX[1] := -1; CY[1] :=  0;  {Go West.      2   }
  CX[2] :=  0; CY[2] := -1;  {Go South.   1 * 3 }
  CX[3] :=  1; CY[3] :=  0;  {Go East.      4   }
  CX[4] :=  0; CY[4] :=  1;  {Go North.         }

  { 4. Assigning initial values.}
  FX[1] := X;  FY[1] := Y;  CLOSE := 1;  NEWN := 1;  YES := false;
```

```
{ 5. Breadth-first search, i.e. the "wave" algorithm.}
  if (X = 1) or  (X = M) or  (Y = 1) or  (Y = N)
  then {If an exit is reached then finish.}
       begin YES := true; U := X; V := Y;
       end;
  if (X > 1) and (X < M) and (Y > 1) and (Y < N)
  then
  repeat {The loop through the nodes.}
    X := FX[CLOSE];  Y := FY[CLOSE]; {Coordinates of node to be closed.}
    K := 0;
    repeat {The loop trough 4 production rules.}
      K := K + 1;  U := X + CX[K];  V := Y + CY[K];
      if LABCOPY[U, V] = 0 {The cell is free.}
      then begin
             LABCOPY[U,V] := LABCOPY[X,Y] + 1; {New wave's number.}
             if (U = 1) or (U = M) or (V = 1) or (V = N) {Boarder.}
             then YES := true; {Success. Here BACK(U,V) could be called.}
             else begin {Placing a newly opened node into front's end.}
                    NEWN := NEWN + 1; FX[NEWN] := U; FY[NEWN] := V;
                  end;
           end;
    until (K = 4) or YES; {Each of 4 productions is checked or success.}
    CLOSE := CLOSE + 1;   {Next node will be closed.}
  until (CLOSE > NEWN) or YES;

  { 6. Printing the path found.}
  if YES
  then begin
         writeln('A path exists.');
         BACK(U,V); {Collecting the path.}
         { Here a procedure should be called to print the path.}
       end
  else writeln('A path does not exist.');
end.
```

A solution is $\langle \pi_4, \pi_4, \pi_3, \pi_3 \rangle$. During program execution, the coordinates of newly opened nodes are placed into the arrays `FX` and `FY`. This is shown in Fig. 8.5.

i	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**	**17**
Wave	'2'	'3'	'3'	'3'	'4'	'4'	'4'	'4'	'5'	'5'	'5'	'5'	'5'	'6'	'6'	'6'	'6'
FX	5	4	5	5	3	4	6	5	2	4	6	4	6	2	3	3	7
FY	4	4	3	5	4	3	3	6	4	2	2	6	6	3	2	6	6
CLOSE:=1	CLOSE=1	CLOSE=1	CLOSE=1	CLOSE=1	CLOSE=2	CLOSE=2	CLOSE=3	CLOSE=4	CLOSE=5	CLOSE=6	CLOSE=7	CLOSE=8	CLOSE=8	CLOSE=9	CLOSE=10	CLOSE=12	CLOSE=13
	NEWN:=2	NEWN:=3	NEWN:=4	NEWN:=5	NEWN:=6	NEWN:=7	NEWN:=8	NEWN:=9	NEWN:=10	NEWN:=11	NEWN:=12	NEWN:=13	NEWN:=14	NEWN:=15	NEWN:=16	NEWN:=17	NEWN:=18

Fig. 8.5. `FX`[*i*] and `FY`[*i*] while executing the program

Breadth-first search visits the sibling nodes before visiting the child nodes. Usually a queue is used in the search process. Depth-first search visits the child nodes before visiting the sibling nodes, that is, it traverses the depth of the tree before the breadth. Usually a stack is used in the search process; see http://en.wikipedia.org/wiki/Graph_search_algorithm.

8.1. Testing Labyrinth Breadth-first Search

Modify your labyrinth depth-first solver to test it with the labyrinth in Fig. 8.1 and to print the trace as follows. The second test: build your labyrinth 20x15 with deadends (marked –1) and path length over 100 productions.

```
PART 1. Data
  1.1. Labyrinth

    Y, V
  7 |   1   1   1   1   1   0   1
  6 |   0   0   0   0   0   0   0
  5 |   1   1   1   1   0   1   1
  4 |   1   0   0   0   2   1   1
  3 |   1   0   1   0   0   0   1
  2 |   1   0   0   0   1   0   1
  1 |   1   1   1   1   1   1   1
    ------------------------------> X, U

        1   2   3   4   5   6   7

  1.2. Initial position X=5, Y=4. L=2.

PART 2. Trace

WAVE 0, label L="2". Initial position X=5, Y=4, NEWN=1

WAVE 1, label L="3"
    Close CLOSE=1, X=5, Y=4.
        R1. X=4, Y=4. Free. NEWN=2.
        R2. X=5, Y=3. Free. NEWN=3.
        R3. X=6, Y=4. Wall.
        R4. X=5, Y=5. Free. NEWN=4.

WAVE 2, label L ="4"
    Close CLOSE=2, X=4, Y=4.
        R1. X=3, Y=4. Free. NEWN=5.
        R2. X=4, Y=3. Free. NEWN=6.
        R3. X=5, Y=4. CLOSED or OPEN.
        R4. X=4, Y=5. Wall.

    Close CLOSE=3, X=5, Y=3.
        R1. X=4, Y =3. CLOSED or OPEN.
        R2. X=5, Y =2. Wall.
        R3. X=6, Y =3. Free. NEWN=7.
        R4. X=5, Y =4. CLOSED or OPEN.

    Close CLOSE =4, X=5, Y=5.
        R1. X=4, Y=5. Wall.
        R2. X=5, Y=4. CLOSED or OPEN.
        R3. X=6, Y=5. Wall.
        R4. X=5, Y=6. Free. NEWN=8.

WAVE 3, label L="5"
    Close CLOSE=5, X=3, Y=4.
        R1. X=2, Y=4. Free. NEWN=9.
        R2. X=3, Y=3. Wall.
        R3. X=4, Y=4. CLOSED or OPEN.
        R4. X=3, Y=5. Wall.
```

```
    Close CLOSE=6, X=4, Y=3.
        R1. X=3, Y=3. Wall.
        R2. X=4, Y=2. Free. NEWN=10.
        R3. X=5, Y=3. CLOSED or OPEN.
        R4. X=4, Y=4. CLOSED or OPEN.

    Close CLOSE=7, X=6, Y=3.
        R1. X=5, Y=3. CLOSED or OPEN.
        R2. X=6, Y=2. Free. NEWN=11.
        R3. X=7, Y=3. Wall.
        R4. X=6, Y=4. Wall.

    Close CLOSE=8, X=5, Y=6.
        R1. X=4, Y=6. Free. NEWN=12.
        R2. X=5, Y=5. CLOSED or OPEN.
        R3. X=6, Y=6. Free. NEWN=13.
        R4. X=5, Y=7. Wall.

WAVE 4, label L="6"
    Close CLOSE=9, X=2, Y=4.
        R1. X=1, Y=4. Wall.
        R2. X=2, Y=3. Free. NEWN=14.
        R3. X=3, Y=4. CLOSED or OPEN.
        R4. X=2, Y=5. Wall.

    Close CLOSE=10, X=4, Y=2.
        R1. X=3, Y=2. Free. NEWN=15.
        R2. X=4, Y=1. Wall.
        R3. X=5, Y=2. Wall.
        R4. X=4, Y=3. CLOSED or OPEN.

    Close CLOSE=11, X=6, Y=2.
        R1. X=5, Y=2. Wall.
        R2. X=6, Y=1. Wall.
        R3. X=7, Y=2. Wall.
        R4. X=6, Y=3. CLOSED or OPEN.

    Close CLOSE=12, X=4, Y=6.
        R1. X=3, Y=6. Free. NEWN=16.
        R2. X=4, Y=5. Wall.
        R3. X=5, Y=6. CLOSED or OPEN.
        R4. X=4, Y=7. Wall.

    Close CLOSE=13, X=6, Y=6.
        R1. X=5, Y=6. CLOSED or OPEN.
        R2. X=6, Y=5. Wall.
        R3. X=7, Y=6. Free. NEWN=17. Terminal.

PART 3. Results

  3.1. Path found.
```

3.2. Grafics

```
  Y, V
7 |   1   1   1   1   1   0   1
6 |   0   0  -1  -1   4   5   6
5 |   1   1   1   1   3   1   1
4 |   1  -1  -1  -1   2   1   1
3 |   1  -1   1  -1  -1  -1   1
2 |   1   0  -1  -1   1  -1   1
1 |   1   1   1   1   1   1   1
  -------------------------------> X, U

      1   2   3   4   5   6   7
```

3.3. Rules R4, R4, R3, R3.

3.4. Nodes [X=5,Y=4], [X=5,Y=5], [X=5,Y=6], [X=6,Y=6], [X=7,Y=6].

9. Breadth-first Search in Graphs

INPUT: 1) a graph; 2) an initial node; 3) a terminal node.
OUTPUT: the shortest path from the initial node to the terminal node.

The algorithm operates with two lists, OPEN and CLOSED, which initially are empty.

1. Add the initial node to OPEN.
2. If OPEN is empty then there is no path. Return FAIL. This happens in the case of a non-connected graph; see Fig. 9.1.
3. Close the **first** node *n* from OPEN: move it from OPEN to CLOSED. If *n* is the terminal node, then collect the path and finish.
4. Take the set $S(n)$ of adjacent nodes to *n*. Add all the nodes from $S(n)$, which are neither in OPEN nor in CLOSED, to the **end** of OPEN. Formally,
$$\text{OPEN} := \text{OPEN} \cup S(n)/(\text{OPEN} \cup \text{CLOSED}) .$$
5. Go to 2.

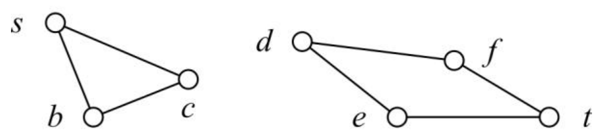


Fig. 9.1. A sample graph with no path from *s* to *t*

Following we show the search for a path from *a* to *e*, see Fig. 9.2.

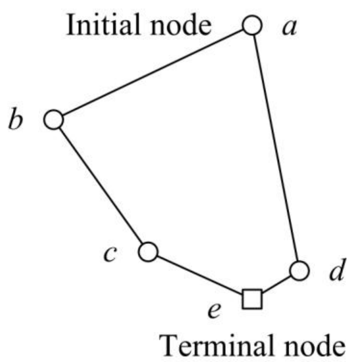


Fig. 9.2. A sample graph. The initial node *a* and the terminal one *e*

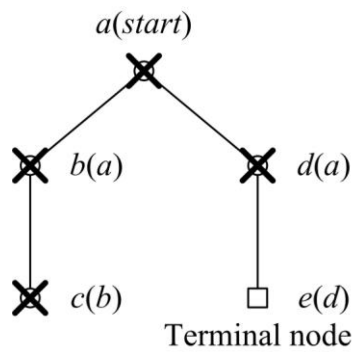


Fig. 9.3. A search tree of the search from *a* to *e*

The lists OPEN and CLOSED are shown in Table 9.4:

	OPEN	CLOSED	Comment
1	$a(start)$	$\varnothing$	Initial state
2	$b(a)$, $d(a)$	$a(start)$	$S(a) = \{b,d\}$. *a* is closed; *b* and *d* opened
3	$d(a)$, $c(b)$	$a(start)$, $b(a)$	$S(b) = \{a, c\}$ but $a \in$CLOSED
4	$c(b)$, $e(d)$	$a(start)$, $b(a)$, $d(a)$	$S(d) = \{a, e\}$ but $a \in$ CLOSED
5	$e(d)$	$a(start)$, $b(a)$, $d(a)$, $c(b)$	$S(c) = \{b, e\}$ but $b \in$ CLOSED and $e \in$ OPEN. Therefore they are not placed twice
6	$\varnothing$	$a(start)$, $b(a)$, $d(a)$, $c(b)$, $e(d)$	The terminal node *e* is being closed. Its

			children are not analysed

Table 9.4. The lists OPEN and CLOSED in every step of the algorithm. A path from *a* to *e* is searched in the graph which is shown in Fig. 9.2

Another example is more complicated. The graph, which is shown in Fig. 9.5, is searched for a path from *a* to *f*. A search tree is shown in Fig. 9.6.

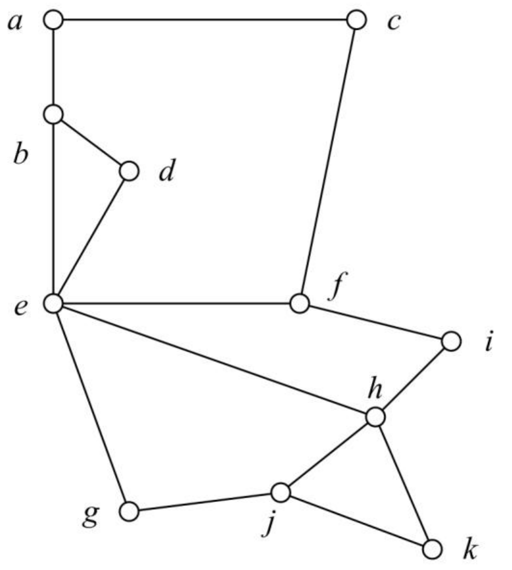


Fig. 9.5. A sample graph. Path search from *a* to *f*

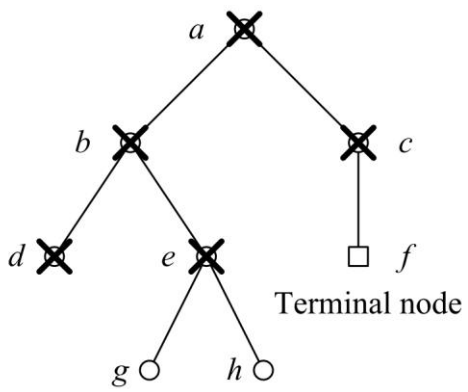


Fig. 9.6. A search tree from *a* to *f* in the graph shown in Fig. 9.5

	OPEN	CLOSED	Comment
1	*a*(*start*)	∅	
2	*b*(*a*), *c*(*a*)	*a*(*start*)	*S*(*a*) = {*b*,*c*}
3	*c*(*a*), *d*(*b*), *e*(*b*)	*a*(*start*), *b*(*a*)	*S*(*b*) = {*a*,*d*,*e*} but *a*∈CLOSED
4	*d*(*b*), *e*(*b*), *f*(*c*)	*a*(*start*), *b*(*a*), *c*(*a*)	*S*(*c*) = {*a*,*f*} but *a*∈CLOSED
5	*e*(*b*), *f*(*c*)	*a*(*start*), *b*(*a*), *c*(*a*), *d*(*b*)	*S*(*d*) = {*b*,*e*} but *b*∈CLOSED
6	*f*(*c*), *g*(*e*), *h*(*e*)	*a*(*start*), *b*(*a*), *c*(*a*), *d*(*b*), *e*(*b*)	*S*(*e*) = {*b*,*d*,*f*,*g*,*h*} but *b*,*d*∈CLOSED and *f*∈OPEN
7	*g*(*e*), *h*(*e*)	*a*(*start*), *b*(*a*), *c*(*a*), *d*(*b*), *e*(*b*), *f*(*c*)	The terminal node *f* is being closed. Its children are not analysed

Table 9.7. The lists OPEN and CLOSED in every step of the algorithm. Path search from *a* to *f* is searched in the graph within Fig. 9.5

One more graph is shown in Fig. 9.8. A set of nodes is $V = \{a,b,c,d,e,f,g,h,i,j,k,l,m,n,o\}$. A set of edges is:

$$E=\{(a,b),(a,c),(a,d),(b,e),(b,f),(c,f),(c,g),(d,h),(d,i),(f,j),(f,k),(g,l),(h,l),(h,m),(h,n),(i,n),(i,o)\}$$

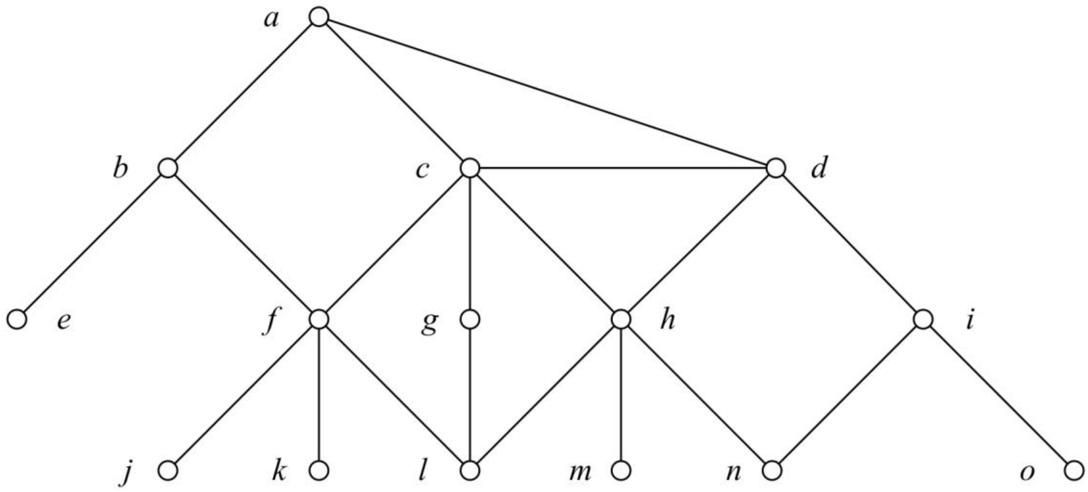


Fig. 9.8. Path search from *a* to *h*. A non-oriented graph $G=\langle V,E\rangle$, where $V = \{a,b,c,d,e,f,g,h,i,j,k,l,m,n,o\}$, $E = \{(a,b), (a,c), (a,d), (b,e), (b,f), (c,f), (c,g), (d,h), (d,i), (f,j), (f,k), (g,l), (h,l), (h,m), (h,n), (i,n), (i,o)\}$

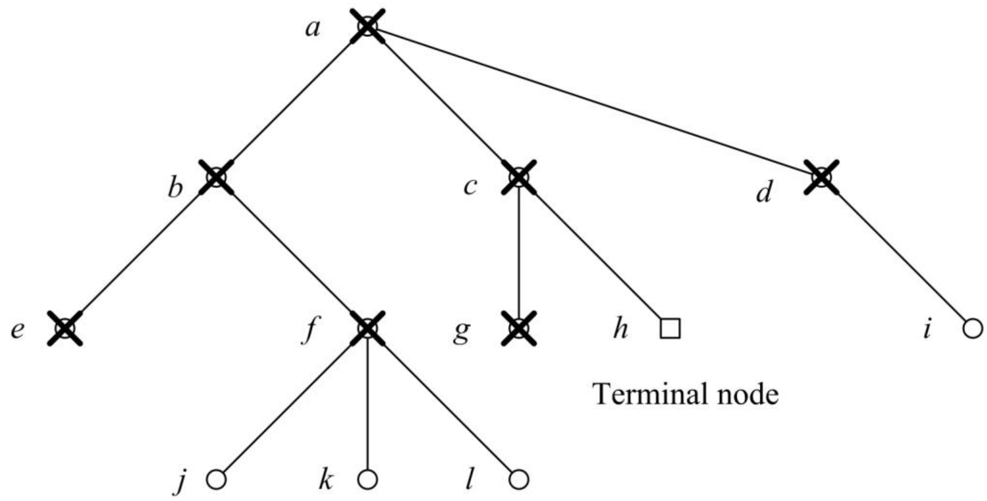


Fig. 9.9. Search tree from *a* to *h* in the graph, which is shown in Fig. 9.8

The OPEN and CLOSED lists are included in the “front” notation:

~~*a*~~, ~~*b*~~, ~~*c*~~, ~~*d*~~, ~~*e*~~, ~~*f*~~, ~~*g*~~, ~~*h*~~, *i*, *j*, *k*, *l*

10. Shortest Path Problem for Graphs with Edge Costs

Suppose graph edges have nonnegative costs. We provide an algorithm to search for the shortest path from an initial node to a terminal node. This is a classic algorithm and is presented in textbooks, see e.g. [Hunt 1978, 10.1.2]. This algorithm is a variation of Dijkstra's algorithm which is conceived by Dutch computer scientist Edsger Dijkstra in 1956; see http://en.wikipedia.org/wiki/Dijkstra%27s_algorithm.

INPUT: 1) a graph with edge costs; 2) an initial node *s*; 3) a terminal node.
OUTPUT: the shortest path from the initial node to the terminal node.

Initially the lists OPEN and CLOSED are empty.

1. Add the initial node to OPEN.
2. If OPEN is empty then there is no path. Return FAIL. This happens in the case of a non-connected graph.
3. Close the **first** node *n* from OPEN: move it from OPEN to CLOSED. Here *n* is the node with the shortest path (from the initial node). (OPEN is sorted in this way). If *n* is the terminal node then collect the path and return it.
4. Take the set $S(n)$ of nodes adjacent to *n*. For each n^* from $S(n)$, which does not appear in CLOSED, calculate its path cost and add it to OPEN. Formally, $\forall n^* \in S(n)$/CLOSED, OPEN :=OPEN $\cup$ $S(n)$/CLOSED, assign $pathcost(s,n^*) := pathcost(s,n) + edgecost(n,n^*)$, and sort OPEN. For each such n^* which appears twice (with an old *pathcost* and a new one) decide for the better one.
5. Go to 2.

Following we show the search for a path from *a* to *e*, see Fig. 10.1.

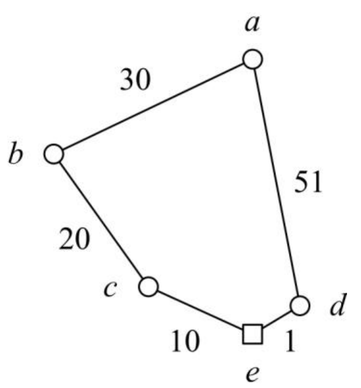


Fig. 10.1. A sample graph with nonnegative edge costs. Suppose an initial node *a* and a terminal node *e*

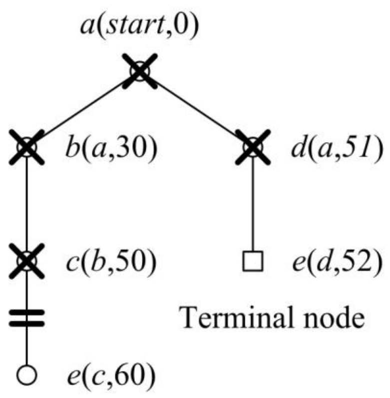


Fig. 10.2. The search tree for a path from *a* to *e* in the graph, which is shown in Fig. 10.1

Following we show OPEN and CLOSED in each execution step of the algorithm.

	OPEN	CLOSED	Comment
1	*a(start,0)*	$\varnothing$	
2	*b(a,30), d(a,51)*	*a(start,0)*	$S(a) = \{b,d\}$
3	*c(b,50), d(a,51)*	*a(start,0), b(a,30)*	$S(b) = \{a,c\}$ but $a \in$CLOSED

4	*d(a,51), e(c,60)*	*a(start,0), b(a,30), c(b,50)*	*S*(*c*) = {*b*,*e*} but *b*∈CLOSED
5	***e(d,52)***	*a(start,0), b(a,30), c(b,50), d(a,51)*	*S*(*d*) = {*a*,***e***} but *a*∈CLOSED and ***e***∈OPEN. New cost **e(d,52)** is better than the old one *e*(*c*,*60*). Take it
6	∅	*a(start,0), b(a,30), c(b,50), d(a,51),* ***e(d,52)***	The terminal node ***e*** **is closed**

Table 10.3. The lists OPEN and CLOSED in each step of the algorithm. A path from a to e is searched in the graph which is shown in Fig. 10.1. The path ⟨*a(start,0), d(a,51), e(d,52)*⟩

Suppose a more complicated graph which is shown in Fig. 10.4.

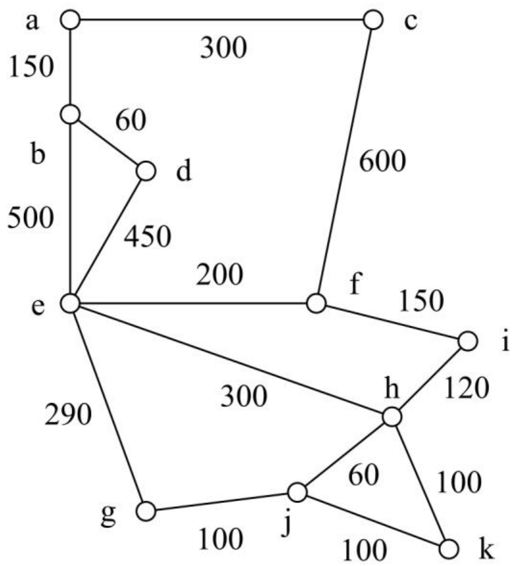


Fig. 10.4. A sample graph. A path is searched from *e* to *j*

The states of lists OPEN and CLOSED in each algorithm’s step.

	OPEN	CLOSED	Comment
1	*e(start,0)*	∅	
2	*f(e,200), g(e,290), h(e,300), d(e,450), b(e,500)*	*e(start,0)*	*S*(*e*) = {*b*,*d*,*f*,*g*,*h*}
3	*g(e,290), h(e,300), i(f,350), d(e,450), b(e,500), c(f,800)*	*e(start,0), f(e,200)*	*S*(*f*) = {*c*,*e*,*i*}, but *e*∈CLOSED
4	*h(e,300), i(f,350), j(g,390), d(e,450), b(e,500), c(f,800)*	*e(start,0), f(e,200), g(e,290)*	S(*g*) = {*e*,*j*}, but *e*∈CLOSED

5	*i(f,350)*, ***j(h,360)***, *k(h,400)*, *d(e,450)*, *b(e,500)*, *c(f,800)*	*e(start,0)*, *f(e,200)*, *g(e,290)*, *h(e,300)*	*S*(*h*) = {*e*,*i*,***j***,*k*}, but *e*∈CLOSED and *i*,***j***∈OPEN. New cost *i(h,420)* is worse than the old one *i(f,350)* and therefore the old one is chosen. New cost ***j(h,360)*** is better than the old one *j(g,390)* and therefore the new one is taken
6	*j(h,360)*, *k(h,400)*, *d(e,450)*, *b(e,500)*, *c(f,800)*	*e(start,0)*, *f(e,200)*, *g(e,290)*, *h(e,300)*, *i(f,350)*	*S*(*i*) = {*f*,*h*}, but *f*,*h*∈CLOSED
7	*k(h,400)*, *d(e,450)*, *b(e,500)*, *c(f,800)*	*e(start,0)*, *f(e,200)*, *g(e,290)*, *h(e,300)*, *i(f,350)*, *j(h,360)*	The terminal node *j* is closed. Its children are not analysed

Table 10.5. The lists OPEN and CLOSED in each step of the algorithm. A path from *a* to *e* is searched in the graph which is shown in Fig. 10.4. The found path is ⟨*e(start,0)*, *h(e,300)*, *j(h,360)*⟩

The corresponding search tree is shown in Fig. 10.6.

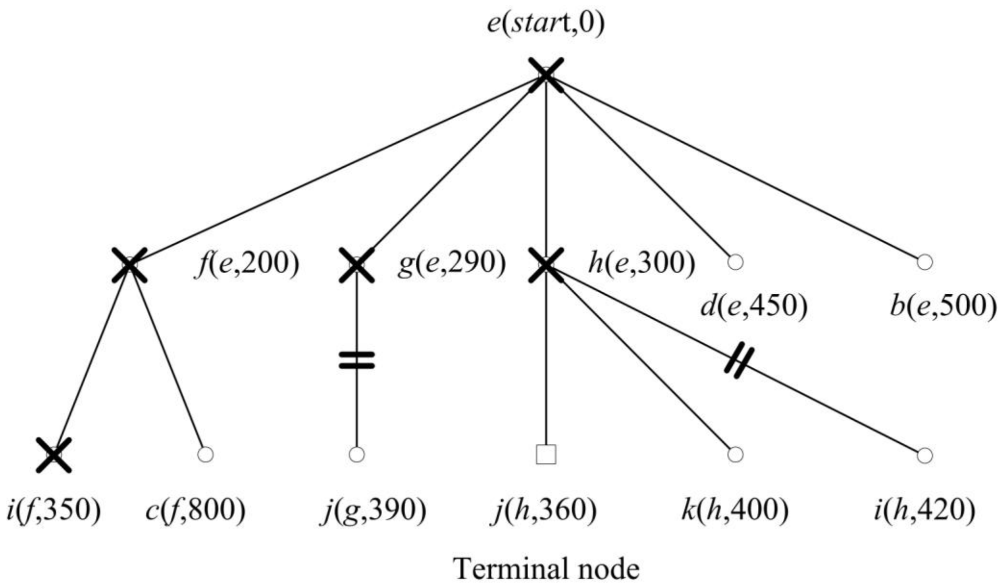


Fig. 10.6. The search tree for a path from *e* to *j* in the graph, which is shown in Fig. 10.4

11. Depth-first Search in Graphs with no Costs

A difference between depth-first search and breadth-first search is illustrated in a sample graph, which is shown in Fig. 11.1 a. Here *s* stands for the initial node and *t* for the terminal. Suppose that production rules – the edges – are sorted contra-clockwise (Fig. 11.1 b). Depth-first search produces the path ⟨*s*,*a*,*e*,*t*⟩ of three edges, whereas breadth-first search produces the path ⟨*s*,*t*⟩, i.e. one edge only.

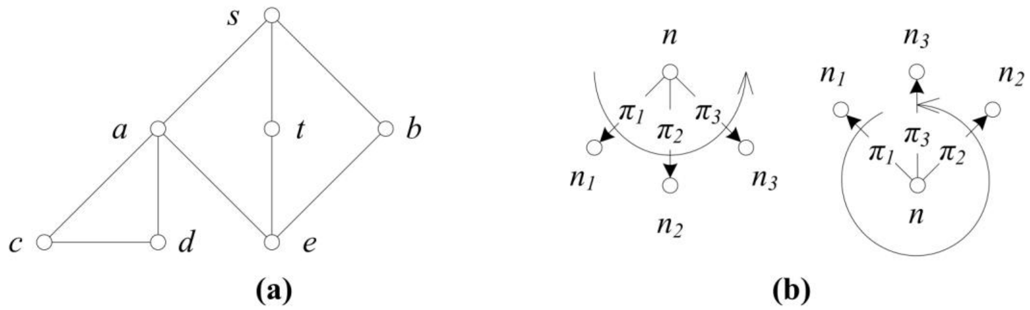


Fig. 11.1. (a) A sample graph. Suppose depth-first search with the initial node *s* and the terminal one *t*. (b) Contra-clockwise ordering of edges. “12 hours” edge is the last one – not the first

INPUT: 1) a graph; 2) an initial node; 3) a terminal node.
OUTPUT: The depth-first search path. It is not necessarily the shortest one.

1. Add the initial node to OPEN.
2. If OPEN is empty then there is no path. Return FAIL. This happens in the case of a non-connected graph; see Fig. 9.1.
3. Close the **first** node *n* from OPEN: move it from OPEN to CLOSED. If *n* is the terminal node, then collect the path and finish.
4. Take the set *S*(*n*) of adjacent nodes to *n*. Add all the nodes from *S*(*n*), which are not in OPEN, to the **beginning** of OPEN (i.e. according to “depth-first”). Formally, OPEN := *S*(*n*)/OPEN ∪ OPEN. Delete duplicated nodes (i.e. the old ones) from the end of OPEN.
5. Go to 2.

Table 11.3 below presents the states of the lists OPEN and CLOSED in each step of the algorithm. The corresponding search tree is shown in Fig. 11.3.

	OPEN	CLOSED	Comment
1	*s(start)*	∅	
2	*a(s), t(s), b(s)*	*s(start)*	*S*(*s*) = {*a*,*t*,*b*}
3	*c(a), d(a), e(a), t(s), b(s)*	*s(start), a(s)*	*S*(*a*) = {*c*,*d*,*e*,*s*}. But *s*∈CLOSED
4	***d(c)***, *e(a), t(s), b(s)*	*s(start), a(s), c(a)*	*S*(*c*) = {*d*,*a*}. But *a*∈CLOSED and ***d***∈OPEN. Therefore ***d(c)***, which is opened later, is placed into the OPEN’s head and *d(a)* is deleted the from OPEN’s tail
5	*e(a), t(s), b(s)*	*s(start), a(s), c(a), d(c)*	*S*(*d*) = {*c*,*a*}. But *c*,*a*∈CLOSED

6	***b(e)***, ***t(e)***	*s(start), a(s), c(a), d(c), e(a)*	$S(e) = \{a,b,t\}$. But $a \in$CLOSED and $b,t \in$OPEN. Therefore ***b(e)*** and ***t(e)***, which are opened later, are placed into the OPEN's head and *t(s)* and *b(s)* are deleted the from OPEN's tail
7	*t(e)*	*s(start), a(s), c(a), d(c), e(a), b(e)*	$S(b) = \{s,e\}$. But $s,e \in$CLOSED
8	∅	*s(start), a(s), c(a), d(c), e(a), b(e), t(e)*	The terminal node *t* is closed. The resulting path is collected backwards: *t* is reached from *e*, *e* from *a*, and *a* from *s*

Table 11.2. The lists OPEN and CLOSED in each step of the algorithm. A path from *s* to *t* is searched in the graph, which is shown in Fig. 11.1 a. The found path is $\langle s,a,e,t \rangle$

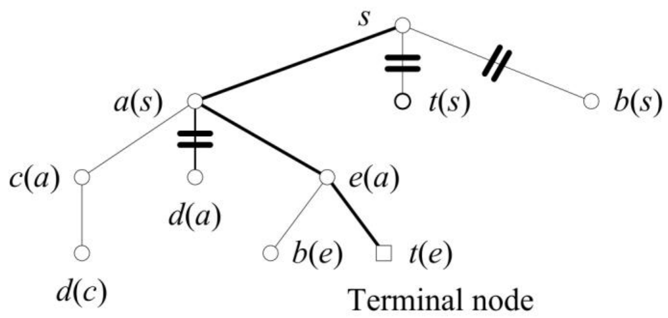


Fig. 11.3. The depth-first search tree for a path from *s* to *t* in the graph, which is shown in Fig. 11.1 a. The links of *d*, *b* and *t* are changed: ***d(c)*** replaces *d(a)*, ***b(e)*** replaces *b(s)* and ***t(e)*** replaces *t(s)*

Solver and planner. The depth-first search strategy is usually used by solvers and the breadth-first search by planners. Solver is shorthand for problem-solver and a solving agent. Planner – a planning agent. Their differences:

	Solver	Planner
1	Does not have a map	Has a map
2	Does not find the shortest path	Can find the shortest path
3	Depth-first search	Normally breadth-first search
4	Onetime usage of the path	Multiple usage of the path

12. The Prefix, Infix and Postfix order of Tree Traversal

Tree traversal refers to the problem of visiting all the nodes in a graph in a particular manner. Below we follow [Jensen, Wirth 1982] Section 12.1

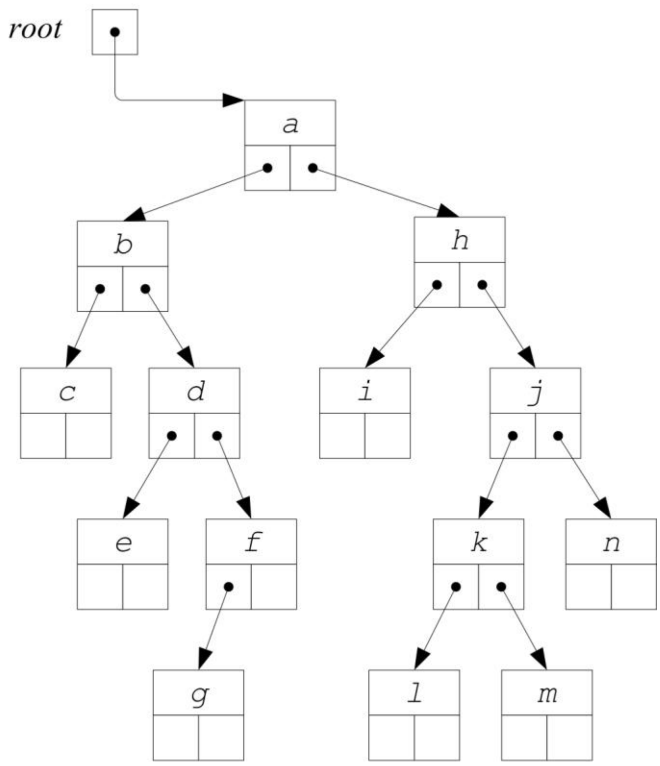


Fig. 12.1. A sample binary tree from [Jensen, Wirth 1982, p. 66]. It is encoded with the string `abc..de..fg...hi..jkl..m..n..`

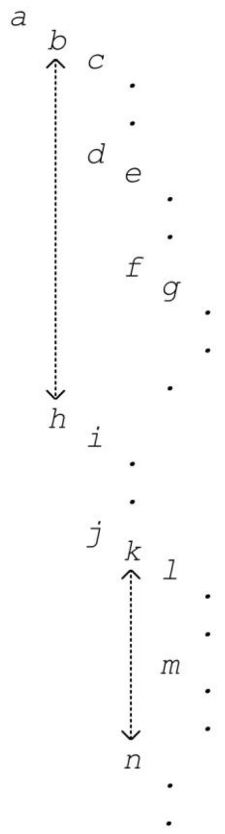


Fig. 12.2. A tree which is isomorphic to that in Fig. 12.1

The tree in Fig. 12.1 is formed reading the following string:

`abc..de..fg...hi..jkl..m..n..`

Following are the procedures to traverse a binary tree in prefix-, infix- and postfix order.

```
program traversal(input, output);
type ptr = ↑node;
     node = record
              info : char;
              llink, rlink : ptr;
            end;
var root : ptr;
    ch :   char;
```

```
procedure enter(var p:ptr);
begin
   read(ch);
   write(ch);
   if(ch ≠ '.') then
   begin
      new(p);
      p↑.info:=ch;
      enter(p↑.llink);
      enter(p↑.rlink);
   end;
   else p := nil;
end;

procedure preorder(p:ptr);
begin
   if p ≠ nil then
   begin
      write(p↑.info);
      preorder(p↑.llink);
      preorder(p↑.rlink);
   end;
end; { preorder }

procedure inorder(p:ptr);
begin
   if p ≠ nil then
   begin
      inorder(p↑.llink);
      write(p↑.info);
      inorder(p↑.rlink);
   end;
end; { inorder }

procedure postorder(p:ptr);
begin
   if p ≠ nil then
   begin
      postorder(p↑.llink);
      postorder(p↑.rlink);
      write(p↑.info);
   end;
end; { postorder }

begin
   write(' '); enter(root);     writeln;
   write(' '); preorder(root);  writeln;
   write(' '); inorder(root);   writeln;
   write(' '); postorder(root); writeln;
end.
```

The program prints four lines:

`abc..de..fg...hi..jkl..m..n..`	– the echo of input data
`abcdefghijklmn`	– preorder
`cbedgfaihlkmjn`	– inorder
`cegfdbilmknjha`	– postorder.

The depth-first traversal of the tree which is shown in Fig. 12.1 is as follows:

abcdefghijklmn

It coincides with the prefix order. The breadth-first traversal is as follows:

abhcdijefknglm

A sample arithmetic expression (see Fig. 12.4):

$$((A+B) * (C–D)+E) * F$$

The representation in the prefix notation is as follows:

$$*+*+AB–CDEF$$

The representation in the postfix notation is as follows:

$$AB+CD–*E+F*$$

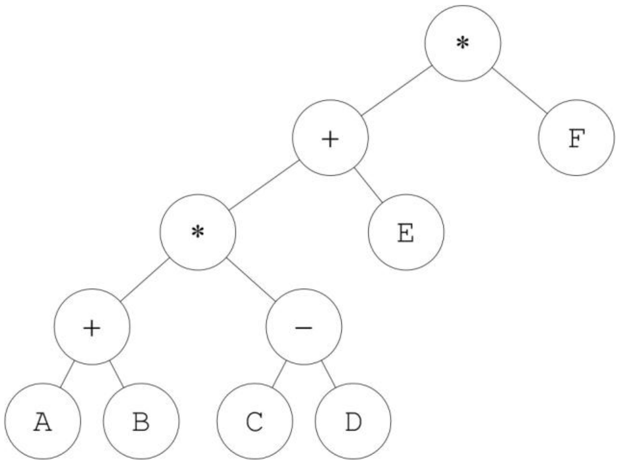


Fig. 12.3. A tree-representation of `(((A+B)*(C-D))+E)*F`

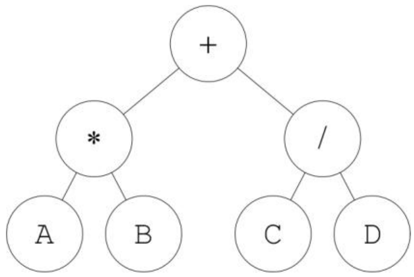


Fig. 12.4. The syntactic tree of `A*B+C/D`

Another example illustrates the compilation of expressions into stack machine’s code. For example, the expression A∗B+C/D (its syntactic tree is shown in Fig. 12.4) is compiled into the following code:

```
Load A
Load B
Multiply
Load C
Load D
Divide
Add
```

The change of stack during interpretation of the code is shown in Fig. 12.5.

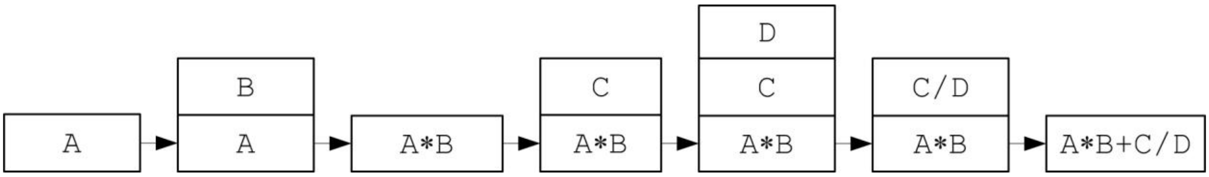


Fig. 12.5. Stack's change while calculating the expression `A*B+C/D`. An arrow shows stack's transition from time *t* to *t*+1

Another example deals with the expression ((A+B) ∗ (C–D)+E) ∗ F. It's syntactic tree is shown in Fig. 12.3. The expression is compiled into the following code:

```
Load A
Load B
Add
Load C
Load D
Subtract
Multiply
Load E
Add
Load F
Multiply
```

Stack's change is shown in Fig. 12.6.

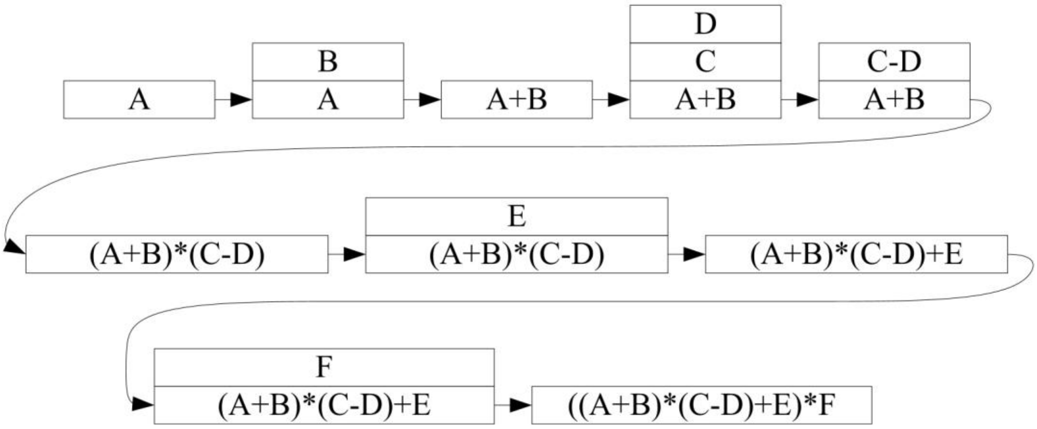


Fig. 12.6. Stack's change while calculating the expression `((A+B)*(C-D)+E) * F`

13. GRAPHSEARCH Algorithm

GRAPHSEARCH is a generalisation of depth-first search, breadth-first search and heuristic search. See also [Nilsson 1982] Section 2.2.2.

INPUT: 1) a graph with nonnegative edge costs; otherwise the cost is 1; 2) an initial node; 3) a terminal node (several terminal nodes can also exist and they are defined by a predicate).

OUTPUT: a path from the initial node to a terminal node.

Initially the lists OPEN and CLOSED are empty.

1. Add the initial node to OPEN and create the search tree T which consists of one node, the initial node.
2. If OPEN is empty then there is no path. Return FAIL.
3. Close the first node n from OPEN: move it from OPEN to CLOSED.
4. If n is the terminal node, then collect the path and finish.
5. Take the set $S(n)$ of all nodes adjacent to n. Add each $n^* \in S(n)$, which is not present neither in OPEN nor in CLOSED, to OPEN and T. Link n^* to n with the notation $n^*(n)$. Formally OPEN := OPEN $\cup$ $S(n)$/(OPEN$\cup$CLOSED)
6. For each $n^* \in S(n)$, which was present in OPEN (in the old OPEN – to the right in the assignment above), i.e. $n^* \in (S(n) \cap$OPEN), decide to change or not to change the link. Do not analyse the rest n^*, i.e. $n^* \in (S(n) \cap$CLOSED) because they are closed.
7. Sort OPEN according to a certain scheme, principle, or heuristic, for instance:
   a) cost
   b) **“depth-first”**. Place $S(n)$/CLOSED to the **beginning** of OPEN and delete duplicated nodes n^* from the end of OPEN. Hence, OPEN is sorted as a **stack**, i.e. LIFO (Last In First Out).
   c) **“breadth-first”**. Place $S(n)$/CLOSED to the **end** of OPEN an delete duplicated nodes n^* from the beginning of OPEN. Hence, OPEN is sorted as a **queue**, i.e. FIFO (First In First Out).
8. Go to 2.

14. Differences between BACKTRACK1 and GRAPHSEARCH-DEPTH-FIRST

14.1. Searching a Graph

Below is the graph which was explored in Fig. 11.1. Suppose the initial node *s* and the terminal one *t*.

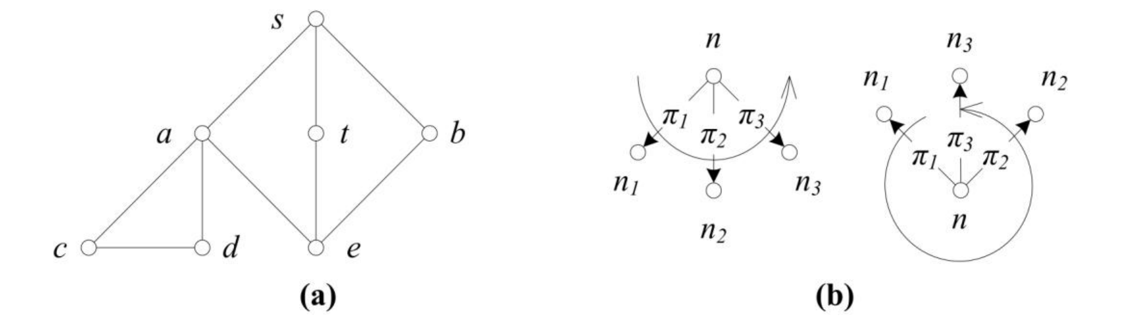


Fig. 14.1 (a) A sample graph. (b) Contra-clockwise ordering of edges. “12 hours” edge is the last one

BACKTRACK1 produces a search tree which is shown in Fig. 14.2 a.

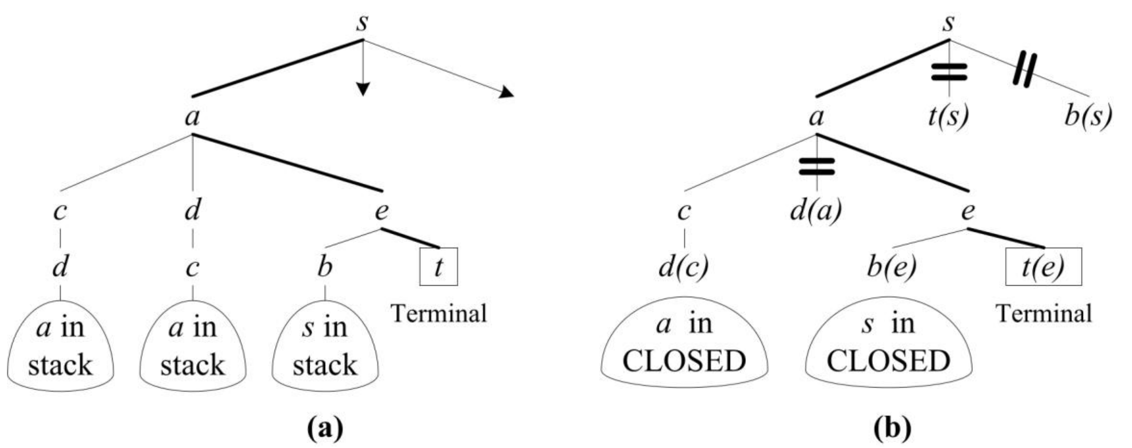


Fig. 14.2. Search trees: (a) BACKTRACK1, (b) GRAPHSEARCH-DEPTH-FIRST. The path is $\langle s,a,e,t \rangle$ is found in both cases

GRAPHSEARCH-DEPTH-FIRST produces a tree which is shown in Fig. 14.2b. It is smaller than the tree in Fig. 14. This is due to cutting branches. GRAPHSEARCH-DEPTH-FIRST produces OPEN and CLOSED which are shown in the front notation:

{*s(start)* {*a(s) t(s) b(s)*} {*c(a) d(a) e(a)*} ***d(c)*** {***b(e) t(e)***}

Three situations to cut branches:

Situation	Opened earlier	Opened later. This is **chosen**
1	d(a)	**d(c)**
2	b(s)	**b(e)**
3	t(s)	**t(e)**

14.2. Example of Labyrinth Search with Loops

Suppose a more complicated example. A labyrinth is shown in Fig. 14.3 a. The node s stands for the initial node. There are two terminal nodes – exits *y* and *t*.

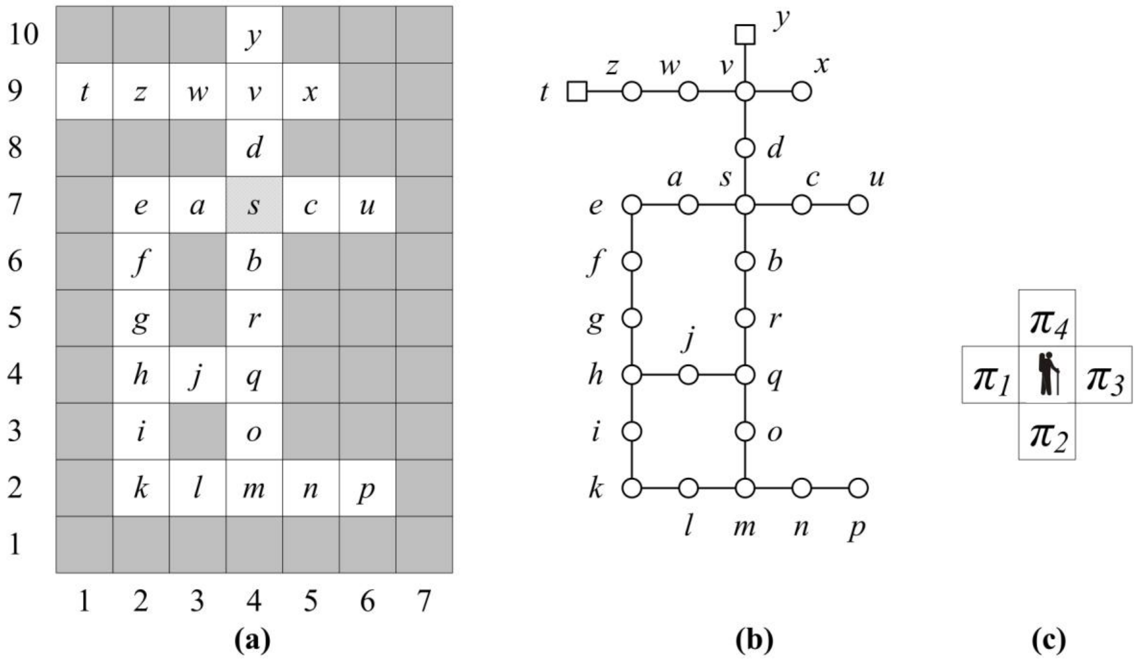


Fig. 14.3. a) A sample labyrinth. The initial node is *s*. There are two exits *y* and *t*. b) A representation in graph notation. c) Edge ordering

Both algorithms find the same path ⟨*s*,*d*,*v*,*w*,*z*,*t*⟩.

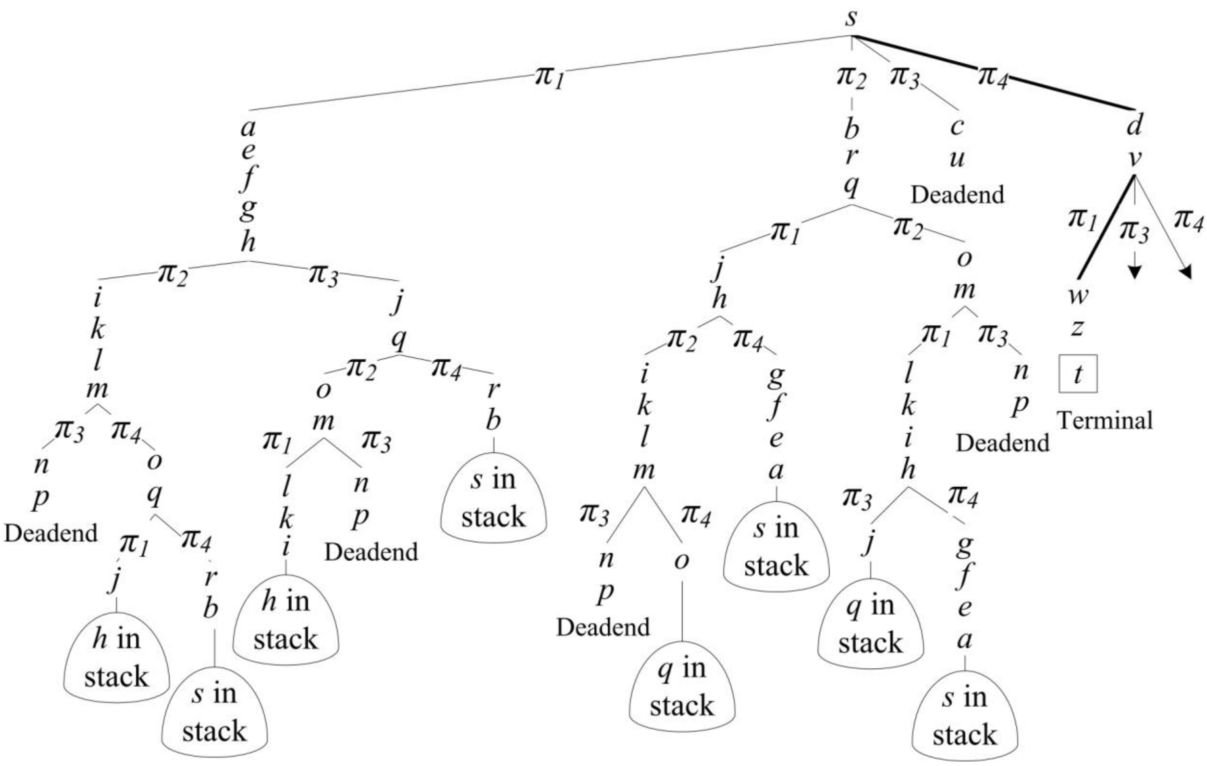


Fig. 14.4. A search tree produced by BACKTRACK1. It is implicit. The found path is ⟨*s*,*d*,*v*,*w*,*z*,*t*⟩

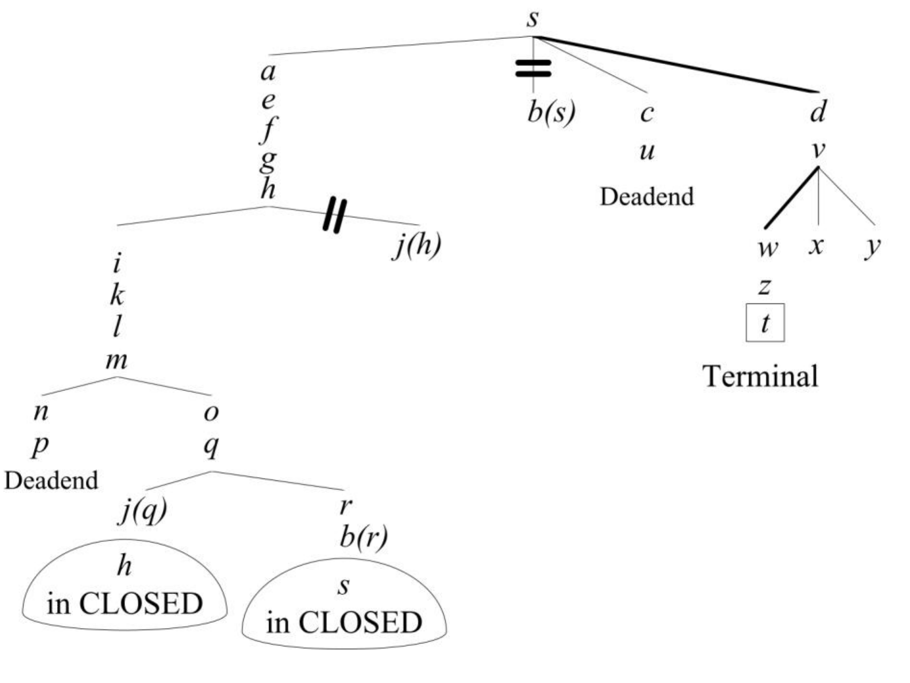


Fig. 14.5 A search tree produced by GRAPHSEARCH-DEPTH-FIRST. It is explicit due to OPEN and CLOSED lists. The path is ⟨*s*,*d*,v,*w*,*z*,*t*⟩

Two cut situations are as follows:

Situation	Opened earlier	Opened later. This is **chosen**
1	*j(h)*	***j(q)***
2	*b(s)*	***b(r)***

14.3. A Counterexample: A Labyrinth where BACKTRACK1 Performs Better

Suppose a labyrinth which is shown in Fig. 14.7.

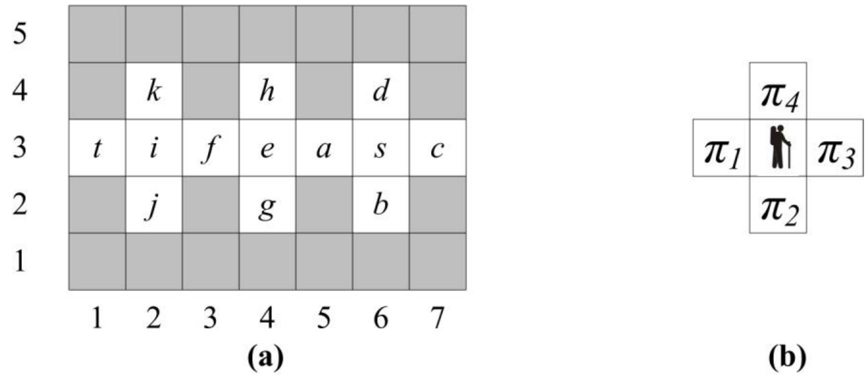


Fig. 14.7 a) A labyrinth for a search from *s* to an exit. There are two exits: *t* and *c*. b) Ordering four production rules

BACKTRACK1 finds an exit straightforwardly without search: $\langle s,a,e,f,i,t\rangle$. To compare two algorithms suppose cost criteria. BACKTRACK1 costs:

- 1 – for placing a production into RULES list when using APPRULES function; see Section 2. In other words, this is the cost to look into an edge.
- 100 – for placing a node into the stack.

Suppose GRAPHSEARCH-DEPTH-FIRST costs:

- 50 – for opening a node. In other words, for placing into the OPEN list.

Both algorithms find the same path. However, BACKTRACK1 costs less; see Fig. 14.8 a and Fig. 14.8 b.

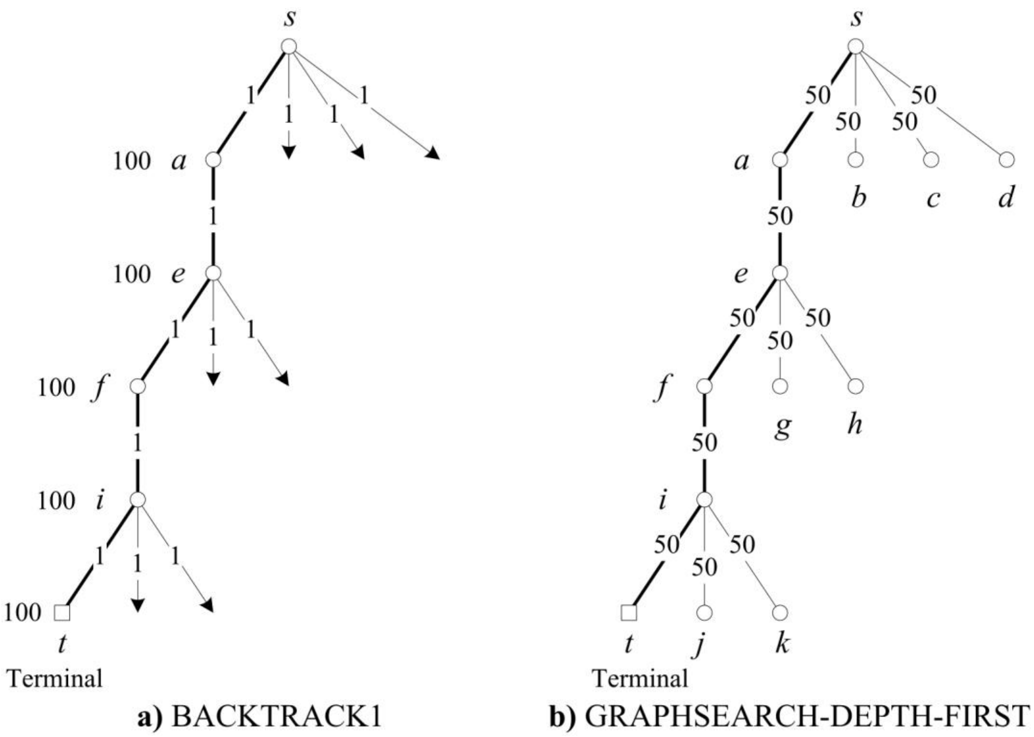


Fig. 14.8 a) BACKTRACK1 search tree costs $5 \cdot 100 + 12 \cdot 1 = 512$. b) GRAPHSEARCH-DEPTH-FIRST search tree costs more: $12 \cdot 50 = 600$

15. Hill-climbing Strategy

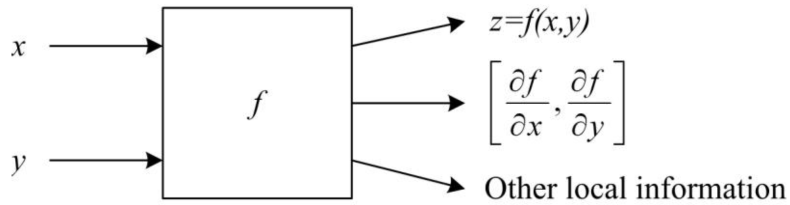


Fig. 15.1. A black-box that produces local information about a function *f* in a point *(x,y)*

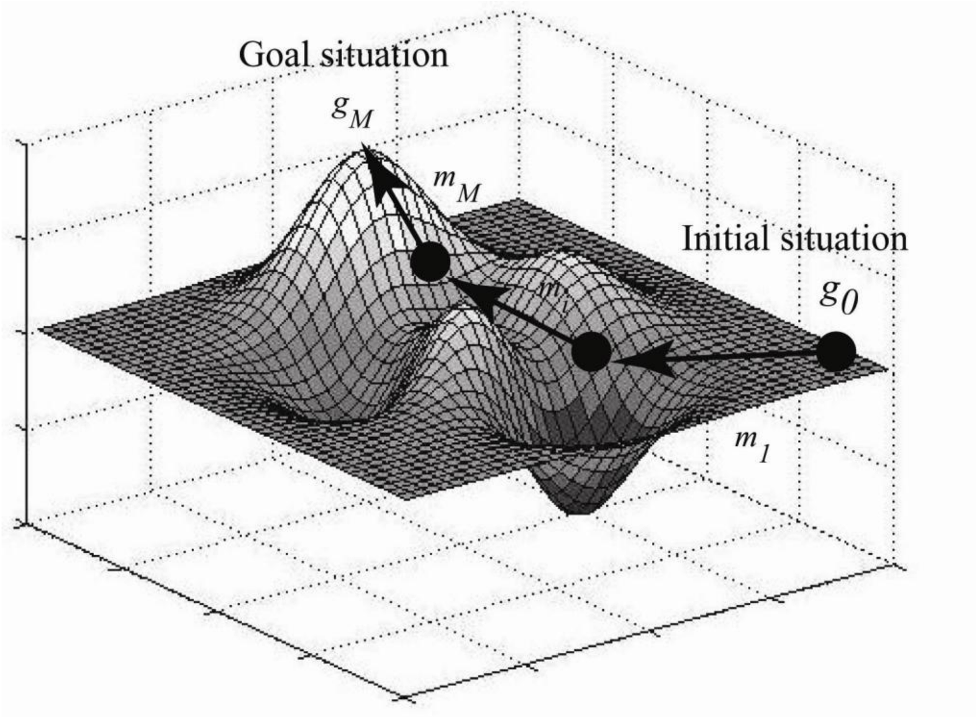


Fig. 15.2. Suppose maximisation of a function *z=f(x,y)*. Hill-climbing is modelled with a sequence $\langle[x_0,y_0], [x_1,y_1], [x_2,y_2], \ldots\rangle$ in the direction of the steepest gradient. The agent is blind and does not see the landscape from a bird's flight view

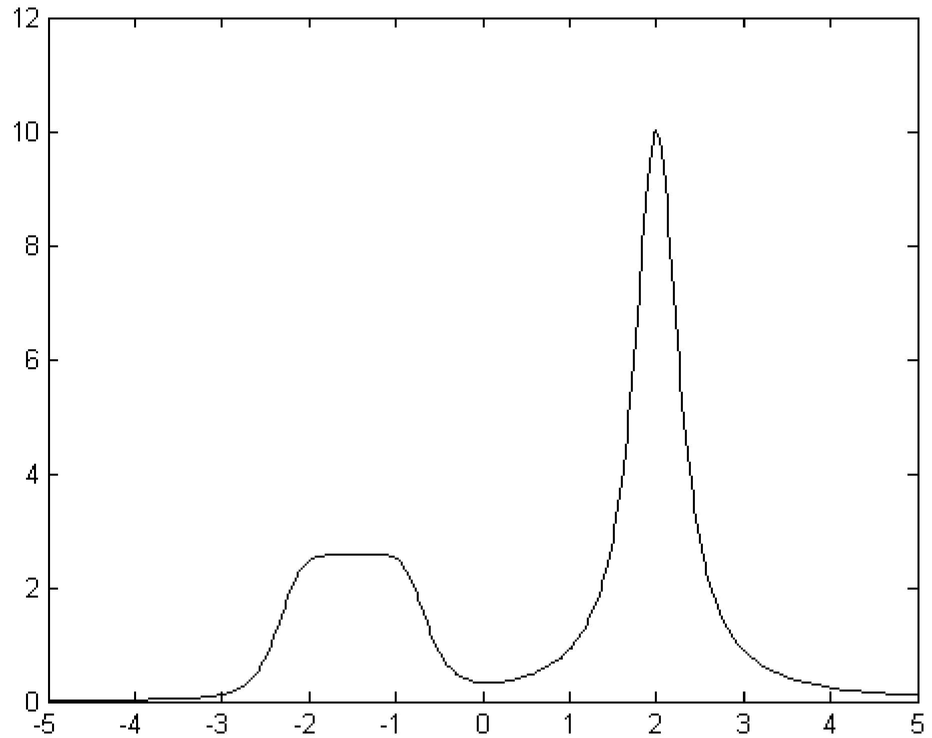


Fig. 15.3. Hill-climbing in 2-dimensional space is modelled with a sequence $\langle x_0, x_1, x_2, \ldots\rangle$ in the direction of positive derivative $f'(x_i)$

Two control strategies are distinguished in artificial intelligence: irrevocable and tentative. Hill-climbing is an example of the irrevocable strategy. Dangers: local maximum (minimum) and plateau.

Hill climbing is demonstrated with the 8-puzzle; see [Nilsson 1982 ch. 2.4.1]. Here the evaluation function is $f(n) = -w(n)$, where $w(n)$ is the number of misplaced tiles (comparing with the terminal state) in that database associated with node n.

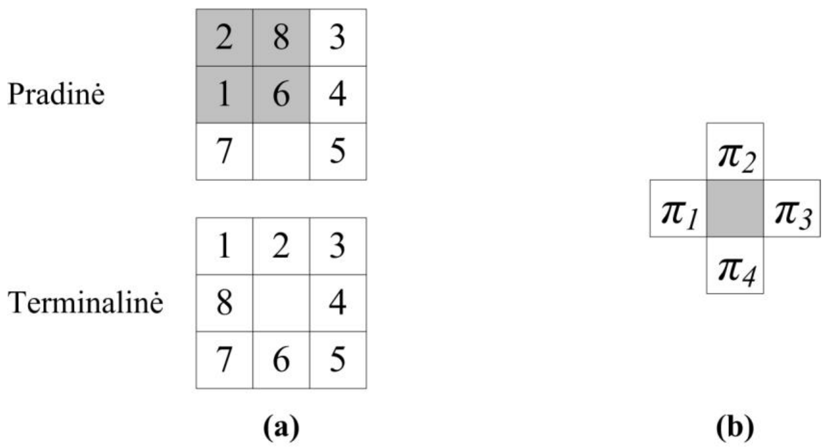


Fig. 15.4. a) A sample initial state s and the terminal state t in the 8-puzzle. $f(s) = -4$, $f(t) = 0$. The task is to maximize f. b) Four moves – a production set $\langle \pi_1, \pi_2, \pi_3, \pi_4\rangle$

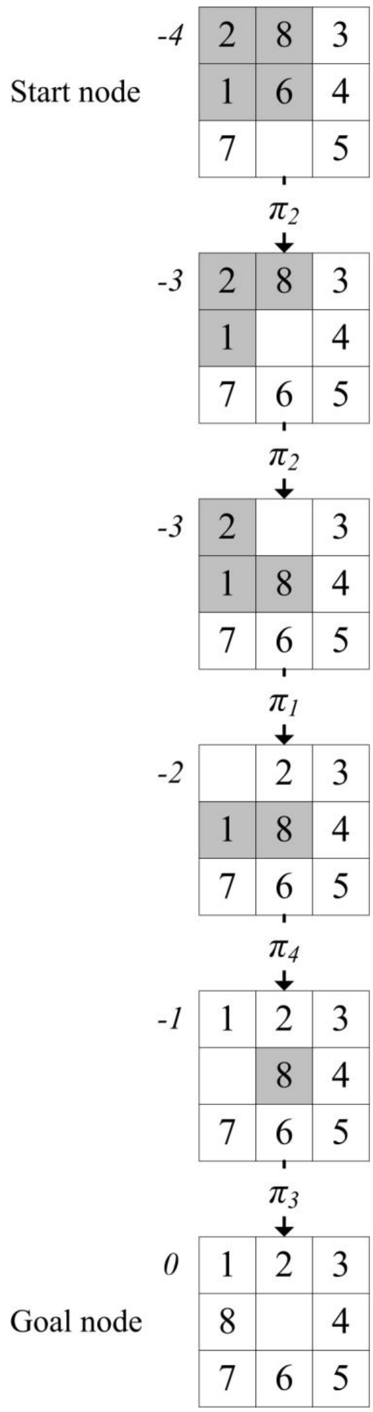


Fig. 15.5. Maximising $f(n) = -w(n)$, where $w(n)$ counts the number of misplaced tiles comparing with the goal node

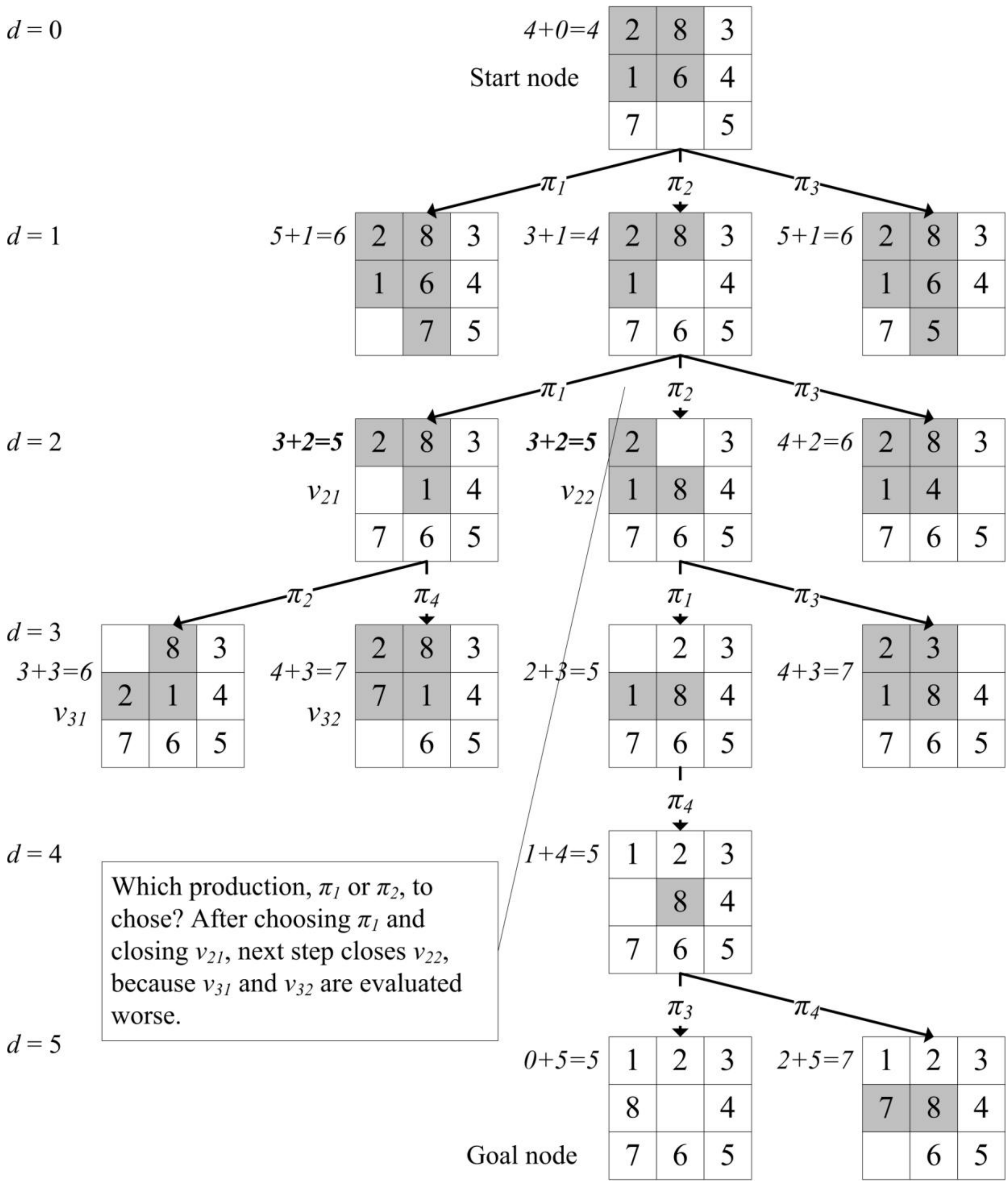


Fig. 15.6. Minimisation of *f(n) = w(n)+d(n)*, where *w(n)* is the number of pieces in wrong places and *d(n)* is the depth of the node

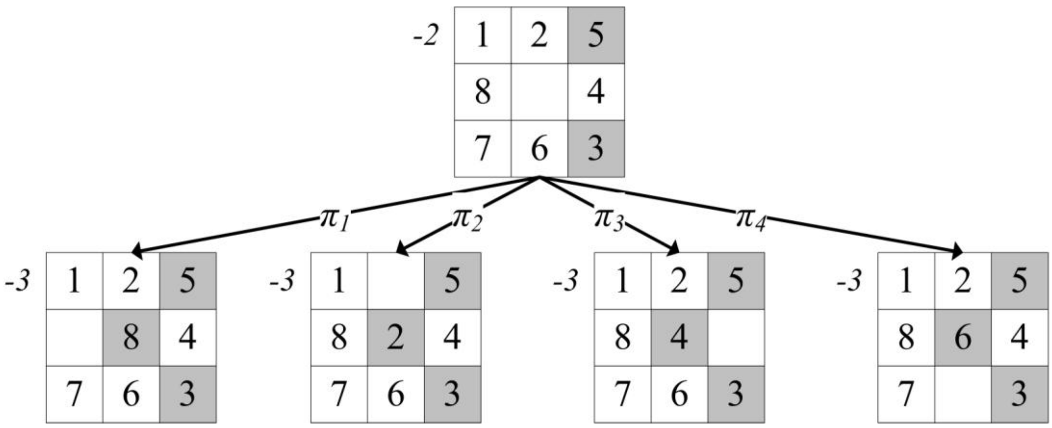


Fig. 15.7. The start state is in a local maximum. Each production leads to a worse state

16. Manhattan Distance

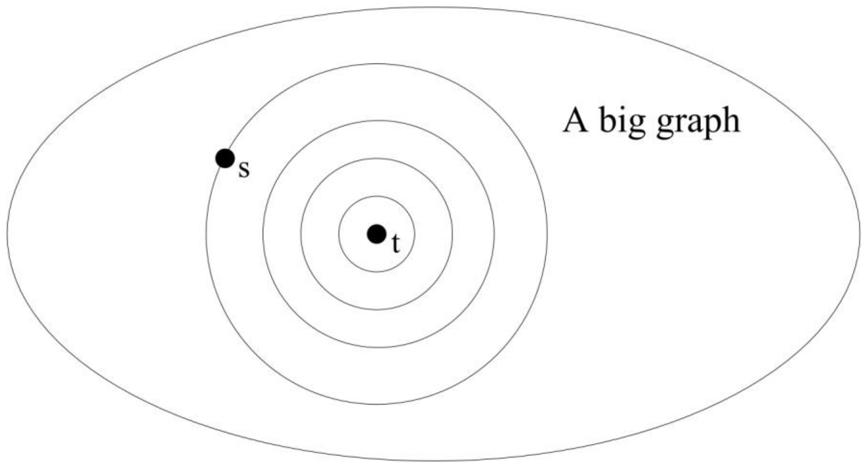


Fig. 16.1. Breadth-first search in a very big graph with s standing for the initial node and t for the terminal node

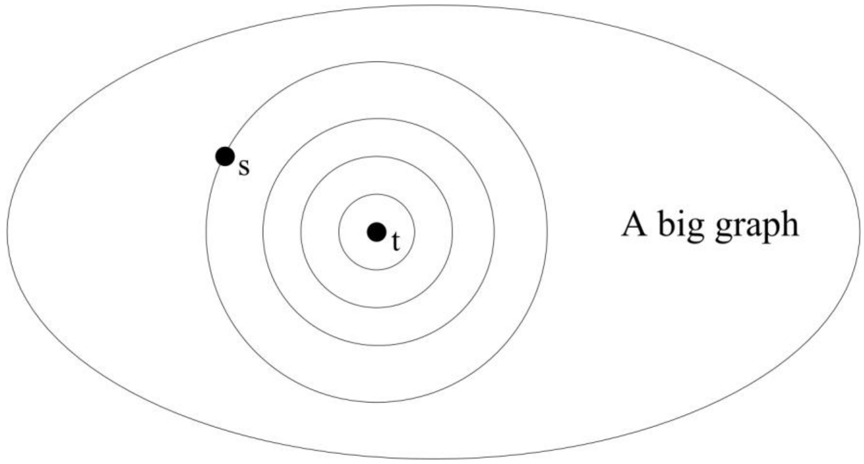


Fig. 16.2. Breadth-first search in opposite direction – backwards from t to s

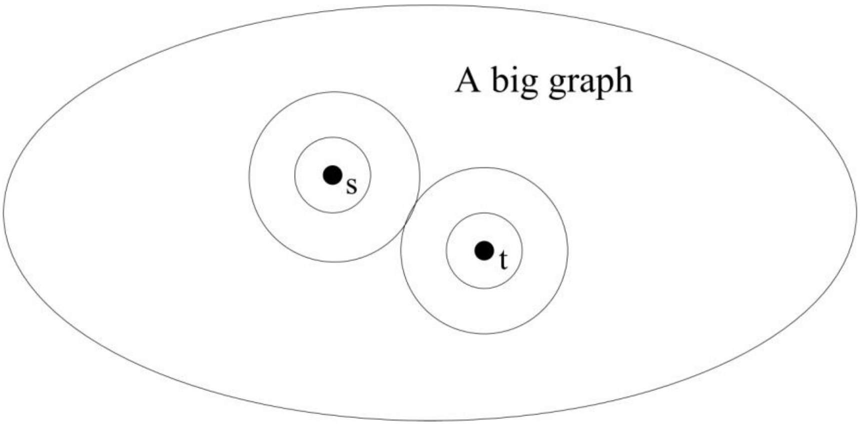


Fig. 16.3. Two concurrent breadth-first searches: from s to t and from t to s. Stop when the fronts meet. This is better than one previous search (either from s to t and from t to s)

Let us denote N the length of the path. The number of nodes in the front is $O(N^{\alpha})$ during breadth-first search from s to t. The same is in the opposite direction from t to s.

$$O(N^{\alpha}) > O((N/2)^{\alpha}) + O((N/2)^{\alpha})$$

In 2-dimensional space $\alpha = 2$ and two concurrent fronts take less memory:

$$N^2 > (N/2)^2 + (N/2)^2 = N^2/4 + N^2/4 = N^2/2$$

In 3-dimensional space $\alpha = 3$ two concurrent fronts take much less memory:

$$N^3 > (N/2)^3 + (N/2)^3 = N^3/8 + N^3/8 = N^3/4$$

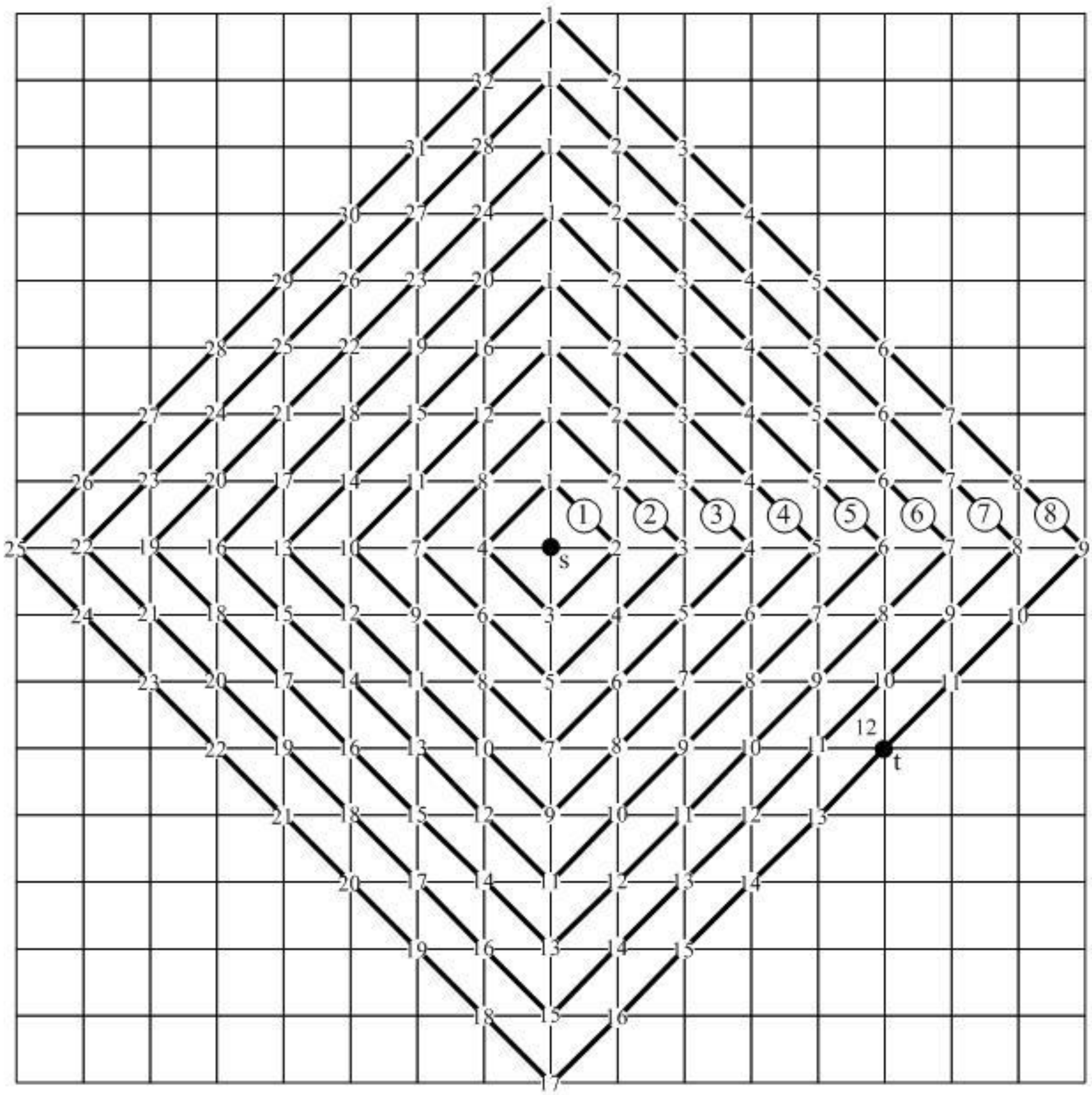


Fig. 16.4. A sample Manhatan-style graph. Searching a path from s to t

The number of opened nodes is listed in Table 16.5.

Wave number	Number of nodes in the front
0	1 + 0
1	4
2	8
3	12
4	16
5	20
6	24
7	28
8	32

Table 16.5. The number of opened nodes in the graph which is shown in Fig. 16.4. Total 145 nodes. The path length is 5 + 3 = 8 edges

$$0 + 4 + 8 + \ldots + 4N = 4(1 + 2 + 3 + \ldots + N) = 4(N(N + 1)/2) = 2(N^2 + N) = O(N^2)$$

The *N*-th wave capture the surface which is equal to

$$2(N^2 + N) + 1$$

Now suppose two concurrent waves; see Fig 16.6.

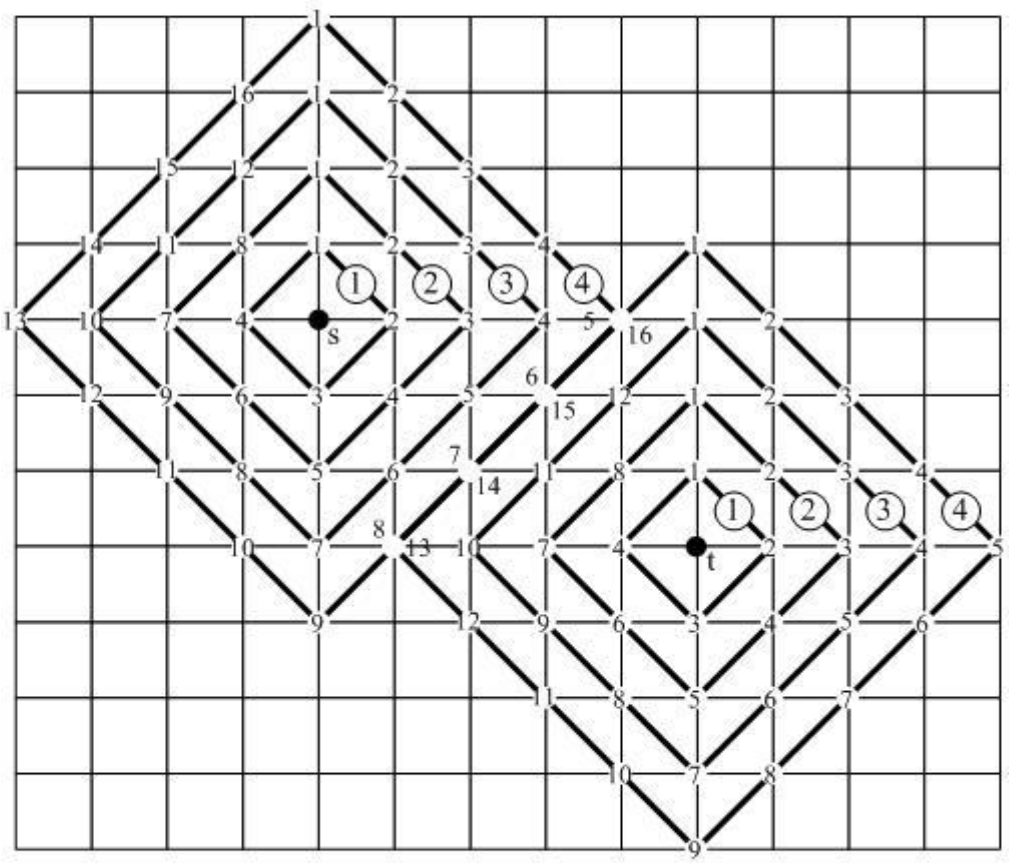


Fig. 16.6. A sample Manhatan-style graph. Two concurrent breadth-first searches: from s to t and from t to s

The number of opened nodes is listed in Table 16.7.

Wave number	Number of nodes in the front
0	1 + 1 + 0 + 0
1	4 + 4
2	8 + 8
3	12 + 12
4	16 + 16

Table 16.7. The number of opened nodes in the graph which is shown in Fig. 16.6. Total 82 nodes

17. A* Search Algorithm

A* (say A star) is the GRAPHSEARCH algorithm with the following evaluation function *f* of the current node *n*:

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

where

- *g(n)* is the shortest path from the starting node *s* to *n*
- *h(n)* is an admissible heuristic estimate of the distance from *n* to the goal *t*.

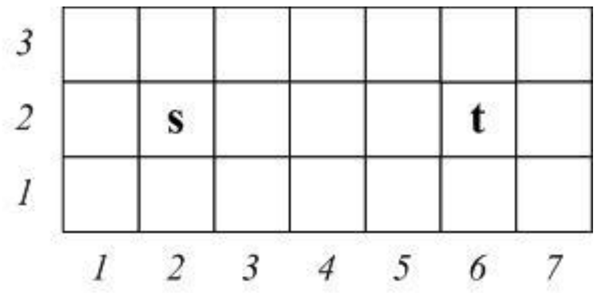
Details see in [Hart, Nilsson, Raphael 1968], [Nilsson 1982] and numerous textbooks e.g. [Russell, Norvig 2003], [Luger 2005] and Wikipedia http://en.wikipedia.org/wiki/A*_search_algorithm. The algorithm employs the lists OPEN and CLOSED.

17.1. *Manhattan Distance in the Tile World*

Further we present an example where the Manhattan distance stands for *h(n)*.

DEFINITION (Manhattan distance) Manhattan norm $d(n_1,n_2)$, i.e. distance between two points $n_1=(x_1,y_1)$ and $n_2=(x_2,y_2)$ is:

$$d(n_1,n_2)=|x_2-x_1|+|y_2-y_1|$$



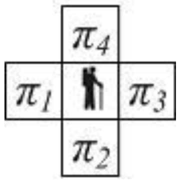


Fig. 17.1. The agent searches a path fron the initial node s to the terminal node t. For moves (productions) are go-West, go-South, go-East and go-North

The front (i.e. OPEN and CLOSED) is shown in Fig. 17.2.

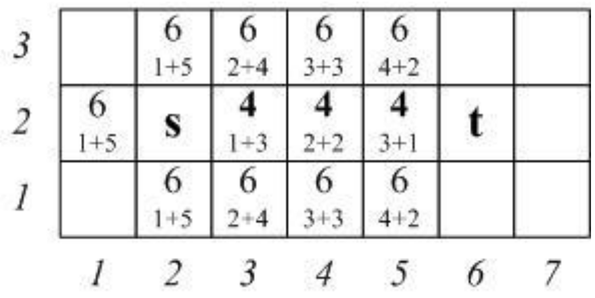
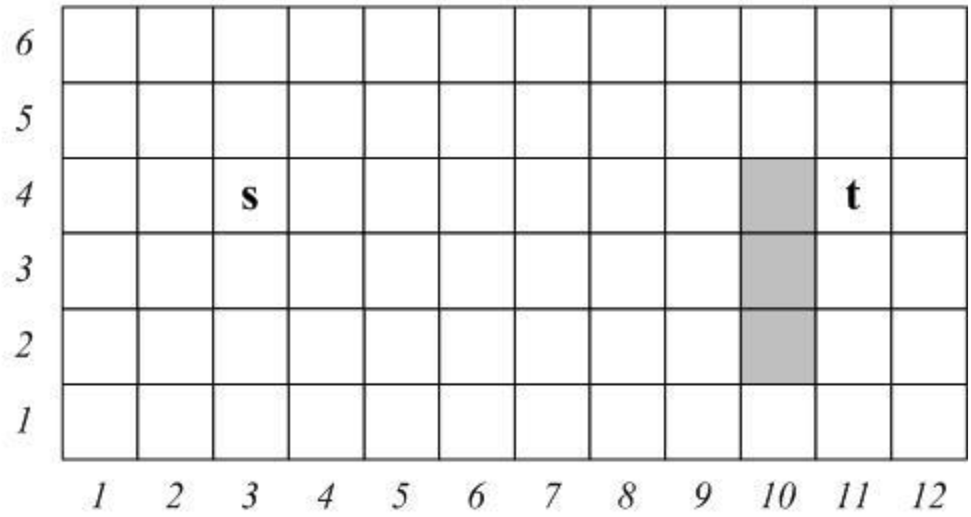


Fig. 17.2. Nodes in the front (OPEN ∪ CLOSED) in A* search from s to t

A more complicated example is shown in Fig. 17.3.



17.3 pav. The agent travels from s to t. Grey positions denote obstacles

Nodes in the front (OPEN ∪ CLOSED) are shown in Fig. 17.4.

6			12 2+10	12 3+9	12 4+8	12 5+7	12 6+6	12 7+5	12 8+4	12 9+3		
5		12 2+10	10 1+9	10 2+8	10 3+7	10 4+6	10 5+5	10 6+4	**10** 7+3	**10** 8+2	**10** 9+1	
4	12 2+10	10 1+9	**s**	**8** 1+7	**8** 2+6	**8** 3+5	**8** 4+4	**8** 5+3	**8** 6+2		**t**	
3		12 2+10	10 1+9	10 2+8	10 3+7	10 4+6	10 5+5	10 6+4	10 7+3			
2			12 2+10	12 3+9	12 4+8	12 5+7	12 6+6	12 7+5	12 8+4			
1												
	1	2	3	4	5	6	7	8	9	10	11	12

Fig. 17.4. The front nodes in A* path search from s to t. The found path is in bold

A one more example is shown in Fig. 17.5.

8				14 3+11	14 4+10	14 5+9	14 6+8	14 7+7	14 8+6	14 9+5	14 10+4	14 11+3	
7			14 3+11	12 2+10	12 3+9	12 4+8	12 5+7	12 6+6	12 7+5	**12** 8+4	**12** 9+3	**12** 10+2	14 11+3
6		14 3+11	12 2+10	10 1+9	10 2+8	10 3+7	10 4+6	10 5+5	10 6+4	**10** 7+3		**12** 11+1	
5	14 3+11	12 2+10	10 1+9	**s**	**8** 1+7	**8** 2+6	**8** 3+5	**8** 4+4	**8** 5+3	**8** 6+2		**t**	
4		14 3+11	12 2+10	10 1+9	10 2+8	10 3+7	10 4+6	10 5+5	10 6+4	10 7+3			
3			14 3+11	12 2+10	12 3+9	12 4+8	12 5+7	12 6+6	12 7+5	12 8+4			
2				14 3+11	14 4+10	14 5+9	14 6+8	14 7+7	14 8+6	14 9+5			
1													
	1	2	3	4	5	6	7	8	9	10	11	12	13

Fig. 17.5. The front nodes in A* path search from s to t. The path is in bold

The fron is elipse-shaped. This is more effective because the elipse-shaped area contains less nodes than than a circle-shaped area.

Algoritmas A* is optimal when the heuristic function *h(n)* is optimistic; see [Russell, Norvig 2003]. A heuristic function is called optimistic in case the evaluation is smaller than an actual path.

17.2. An Example from Russell & Norvig 2003

Further A* explanation follows [Russell, Norvig 2003] Chapter 4 p. 94-98. A path is searched from A to B in the graph which is shown in Fig. 17.6.

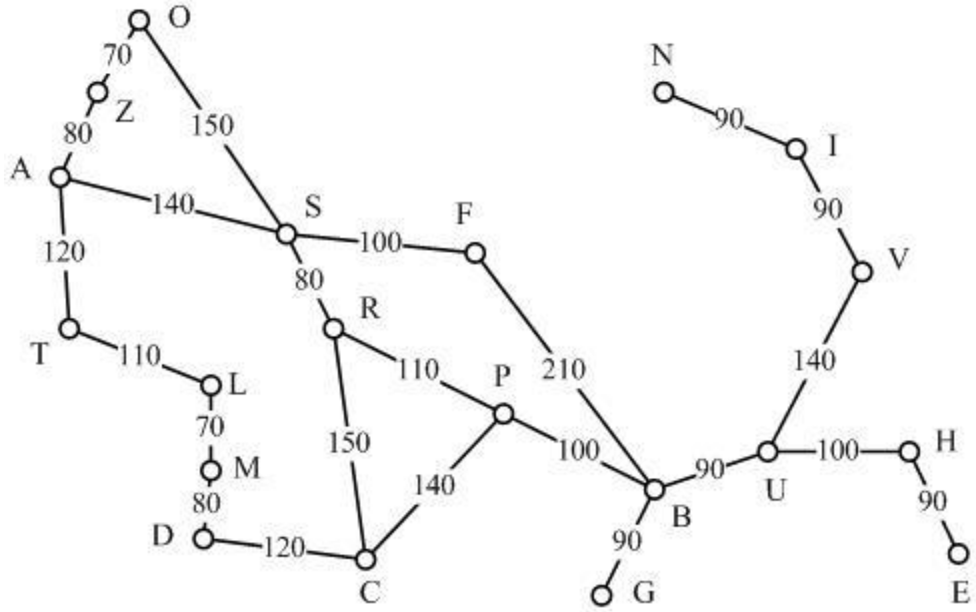


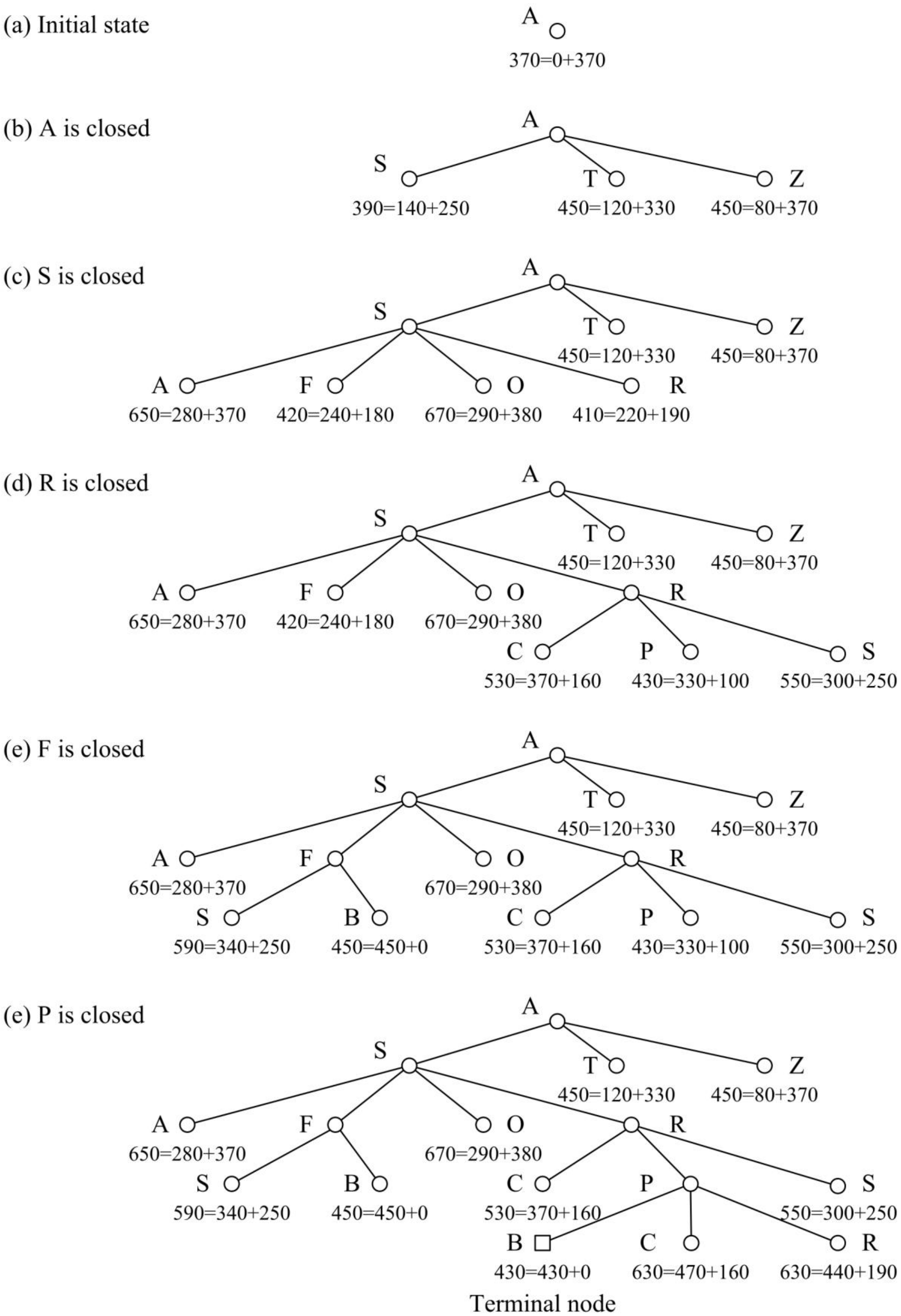
Fig. 17.6 pav. The graph shows transport routes in Romania and is adapted from [Russell, Norvig 2003] 3.2 pav. p. 63. A path is searched from A to B

Heuristic function *h(n)* is listed in Table 17.7. This heuristic evaluation presents straight distance from *n* to B.

Miestas	**Atstumas**	**Miestas**	**Atstumas**
A	370	M	240
B	0	N	230
C	160	O	380
D	240	P	100
E	160	R	190
F	180	S	250
G	80	T	330
H	150	U	80
I	230	V	200
L	240	Z	370

Table 17.7 lentelė. Heuristic functio *h(n)*. This heuristic evaluation presents straight distance from *n* to B

Search trees in each step of A* are shown in Fig. 17.8.



17.8 pav. Search trees in each step of A*. Nodes mark *f(n)=g(n)+h(n)*, where *h(n)* presents straight distance from *n* to B

18. Forward Chaining and Backward Chaining with Rules

In the beginning we follow [Waterman 1989]. Suppose a rule system:

$$\begin{aligned} \pi_1&: \quad F, B \rightarrow Z \\ \pi_2&: \quad C, D \rightarrow F \\ \pi_3&: \quad A \quad \rightarrow D \end{aligned} \tag{18.1}$$

Each rule is of the form rule_name: antecedent → consequent. In the case consequent consists of several propositional variables, the rule is replaced by several rules. For example, π_j: A, B, C → J, K is replaced with

$$\begin{aligned} \pi_{j1}&: \quad A, B, C \rightarrow J \\ \pi_{j2}&: \quad A, B, C \rightarrow K \end{aligned}$$

Global data base (GDB) consists of facts, for example, {A,B,C}. A goal is a new fact (propositional variable), for example, Z.

The problem is to find a sequence of rules which produces the goal from the facts. Two algorithms – forward chaining and backward chaining – are discussed further.

Both algorithms are implemented in Web services; see the server at http://juliuschainingexample.appspot.com/. The client see at http://juliuschainingwsclient.herokuapp.com/. The client requires to copy-paste the JSON format result, which is produced by the server.

18.1. Forward Chaining

It starts from facts and proceed to the goal.

Suppose a rule system (18.1). Suppose an initial state of GDB which consists of three facts {A,B,C}; see Fig. 18.1.

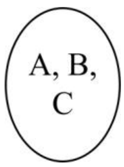


Fig. 18.1. The initial state of GDB is {A,B,C}

Iteration 1. Rule π_3 is applied; see Fig. 18.2.

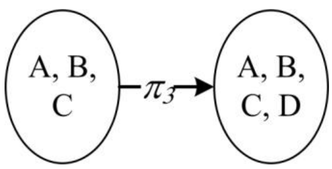


Fig. 18.2. Rule π_3 is applied. A new GDB state is {A,B,C,D}

Proceed with next iteration.

Iteration 2. Rule π_2 is applied; see Fig. 18.3.

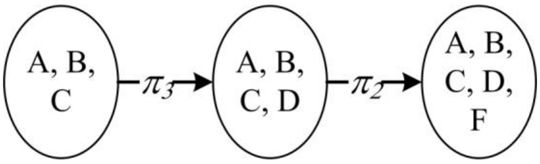


Fig. 18.3. Rule π_2 is applied. A new GDB state is {A,B,C,D,F}

Proceed with next iteration.
Iteration 3. Rule π_1 is applied; see Fig. 18.4.

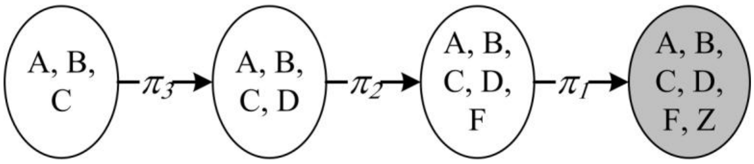


Fig. 18.4. Rule π_1 is applied. A new GDB state is {A,B,C,D,F,Z}. This is a terminal state because it contains the goal Z

The following result – a sequence of rules – is obtained. It is called a *path* (or a plan):

$$\langle \pi_3, \pi_2, \pi_1 \rangle \tag{18.2}$$

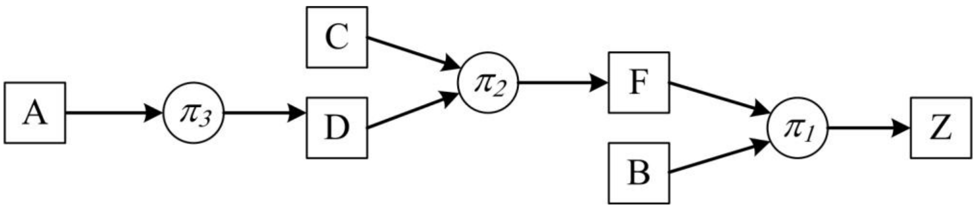


Fig. 18.5. A semantic network shows a derivation of the object (fact) Z from objects (facts) {A,B,C} in the rule system (18.1). The rule sequence $\langle \pi_3, \pi_2, \pi_1 \rangle$ stands for the result

18.2. Backward Chaining

Backward chaining starts from the goal and proceeds towards the facts. This is in contrast to forward chaining. Suppose (18.1) as an example.
Iteration 1. Rule π_1 is selected in (18.1). See Fig 18.6.

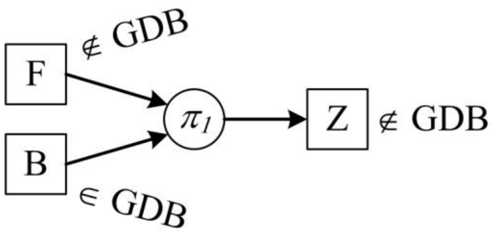


Fig. 18.6. The semantic network of the rule π_1. It shows the derivation of Z backwards. The input object B belongs to {A,B,C}, but F does not. Therefore F is taken as a new goal

Proceed with next iteration.
Iteration 2. Rule π_2 is selected in (18.1). See Fig. 18.7.

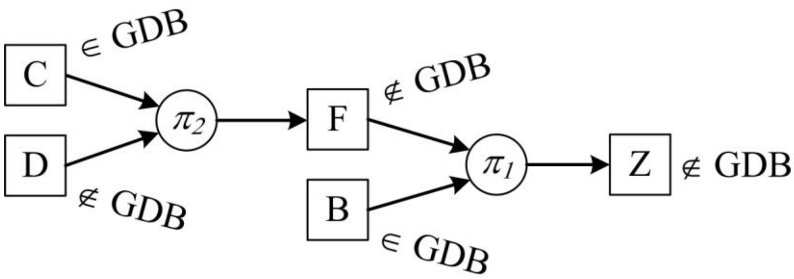


Fig. 18.7. The semantic network of the rule π_2. It shows the derivation of F backwards. The input object C belongs to {A,B,C}, but D does not. Therefore D is a new goal

Proceed with next iteration.
Iteration 3. Rule π_3 is selected in (18.1). See Fig 18.8.

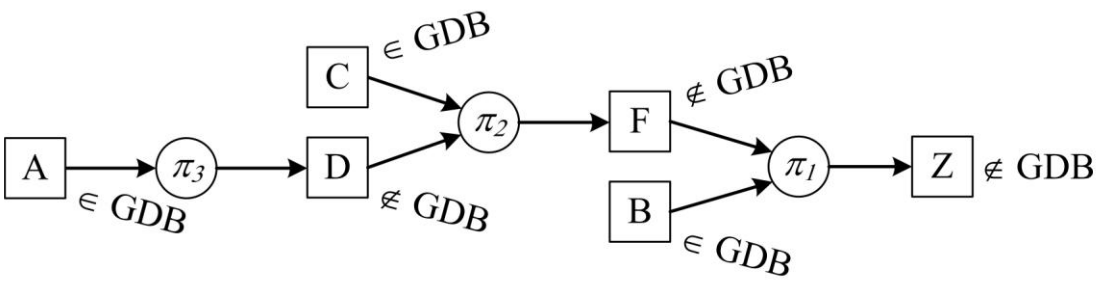


Fig. 18.8. A semantic network shows the derivation of Z from {A,B,C}. The rule sequence $\langle \pi_3, \pi_2, \pi_1 \rangle$ stands for result

18.3. Program Synthesis

Suppose procedures (18.3). Their semantics in terms of input-output is represented in (18.1). The whole object list {A,B,C,D,E,F,Z} stands for the alphabet.

```
procedure P1;
  Z := f1(B,F)

procedure P2;
  F := f2(C,D)

procedure P3;
  D := f3(A)
```
(18.3)

The semantics of procedures P1, P2, P3 is represented with π_1, π_2 and π_3. This is graphically represented in Fig. 18.9.

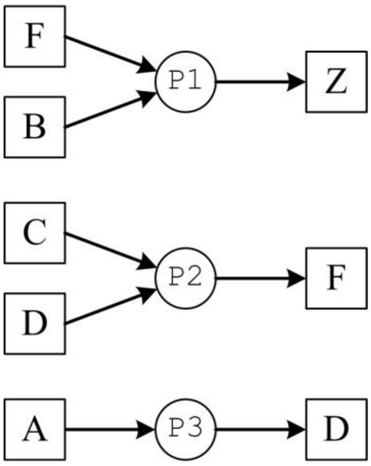


Fig. 18.9. A graphical representation of P1, P2 and P3 semantics

The sequence $\langle \pi_3, \pi_2, \pi_1 \rangle$ (18.2) corresponds to the following *synthesized* program:

```
call P3;
call P2;
call P1
```
(18.4)

The semantics of this program is shown in Fig. 18.10.

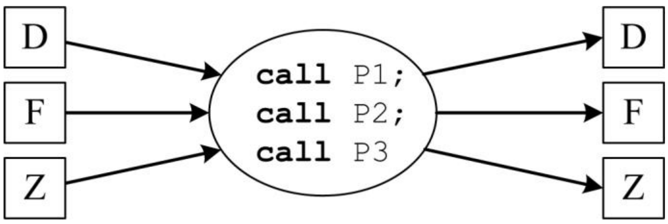


Fig. 18.10. Semantic network representation of the synthesized program

This program can be executed many times. However, it synthesized once. Each execution can read new values of objects A, B, C:

```
for J := 1 to 900 do  { Repeat }
begin
  readln (A, B, C);   { 1) Read A, B ir C              }

  call P3;            { 2) Invoke synthesized program }
  call P2;
  call P1;

  writeln(Z);         { 3) Print Z                     }
end
```

18.4. Redundant Rules in Forward Chaining

Suppose rules (18.1) and add two redundant rules π' ir π'':

$$\begin{array}{lll} \pi': & A & \to L \\ \pi'': & L & \to K \\ \pi_1: & F, B & \to Z \\ \pi_2: & C, D & \to F \\ \pi_3: & A & \to D \end{array} \tag{18.5}$$

Facts are {A,B,C} and goal Z. Forward chaining is shown in Fig. 18.11.

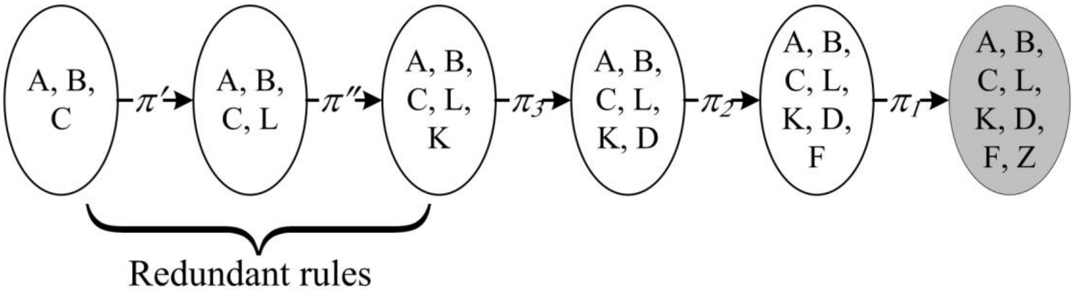


Fig. 18.11. Forward chaining produces $\langle \pi', \pi'', \pi_3, \pi_2, \pi_1 \rangle$. It contains redundant rules π' ir π'' that derive redundant objects L and K

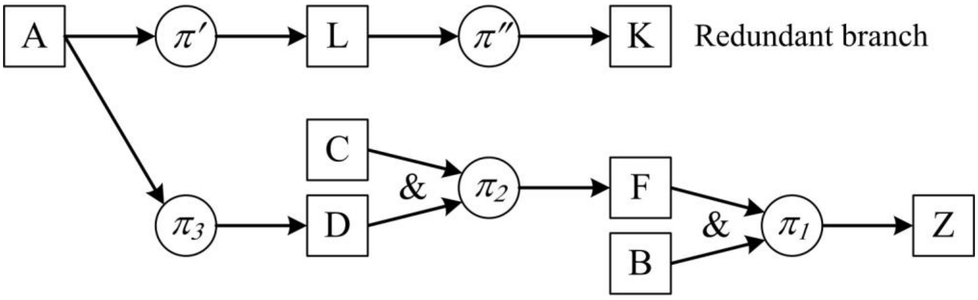


Fig. 18.12. Redundant rules π' ir π'' derive redundant objects L and K

18.5. Redundant Rules in Backward Chaining

Suppose rules (18.1) and add two redundant different rules π' ir π'':

$$
\begin{array}{ll}
\pi': & K \quad \rightarrow M \\
\pi'': & M, N \rightarrow Z \\
\pi_1: & F, B \rightarrow Z \\
\pi_2: & C, D \rightarrow F \\
\pi_3: & A \quad \rightarrow D
\end{array}
\tag{18.6}
$$

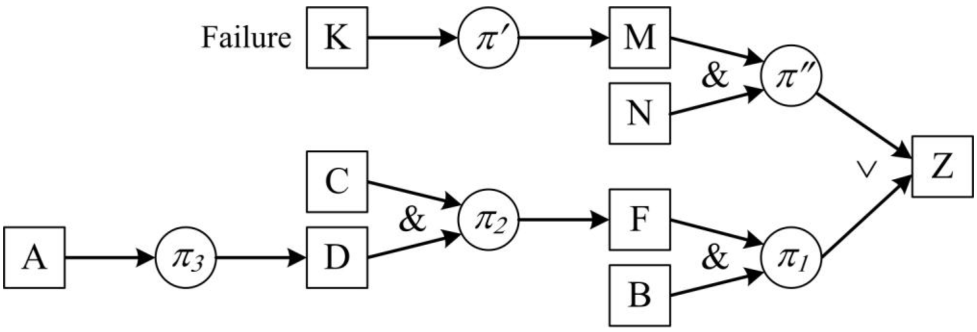


Fig. 18.13. Two redundant rules π' and π'' in backward chaining. Their goals N and K cannot be derived. Therefore these rules are not included in the resulting path $\langle \pi_3, \pi_2, \pi_1 \rangle$

18.6. Complexity of Forward Chaining

Theorem. Forward chaining in a rule system that consists of *N* rules of (18.1) pattern pavidalą is yra $O(N^2)$.

Proof. The worst case to search among *N* rules consists of *N* iterations. Maximum sum of iterations is

$$N + N–1 + N–2 + \ldots + 1 = N \cdot (N–1)/2 = O(N^2)$$

18.7. Testing Forward Chaining

1. **Initial fact in right hand side**
Rules see in the table.
Facts A, B, C.
Goal Z.
Path R1, R2, R7, R4, R6, R5.

Input file:

R1:	A	→ L
R2:	L	→ K
R3:	D	→ A
R4:	D	→ M
R5:	F, B	→ Z
R6:	C, D	→ F
R7	A	→ D

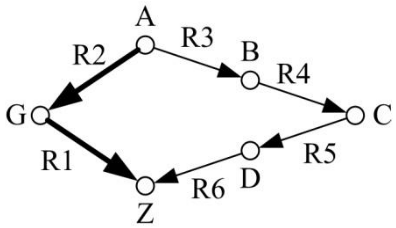
```
Student First name Last name. University, study program.
Test 1. Initial fact in right hand side
1) Rules
L A          // R1: A -> L. Comments.
K L          // R2: L -> K
A D          // R3: D -> A
M D          // R4: D -> M
Z F B        // R5: F, B -> Z
F C D        // R6: C, D -> F
D A          // R7: A -> D

2) Facts
A B C

3) Goal
Z
```

The trace:

```
PART 1. Data

  1) Rules
     R1: A -> L
     R2: L -> K
     R3: D -> M
     R4: D -> M
     R5: F, B -> Z
     R6: C, D -> Z
     R7: A -> D

  2) Facts
     A, B, C

  3) Goal
     Z

PART 2. Trace

  ITERATION 1
    R1:A->L apply. Raise flag1. Facts A, B, C, L.

  ITERATION 2
    R1:A->L skip, because flag1 raised.
    R2:L->K apply. Raise flag1. Facts A, B, C, L, K.

  ITERATION 3
    R1:A->L skip, because flag1 raised.
    R2:L->K skip, because flag1 raised.
    R3:D->A not applied, because of lacking D.
    R4:D->M netaikome, nes trūsta D.
    R5:F,B->Z netaikome, nes trūksta F.
    R6:C,D->F netaikome, nes trūksta D.
    R7:A->D taikome. Pakeliame flag1. Faktai A, B, C, L, K, D.

  ITERATION 4
    R1:A->L skip, because flag1 raised.
    R2:L->K skip, because flag1 raised.
    R3:D->A not applied, because RHS in facts. Raise flag2.
    R4:D->M apply. Raise flag1. Facts A, B, C, L, K, D, M.

  ITERATION 5
    R1:A->L skip, because flag1 raised.
```

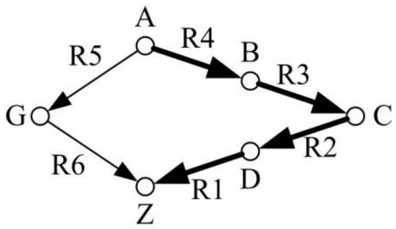
```
    R2:L->K skip, because flag1 raised.
    R3:D->A skip, because flag2 raised.
    R4:D->M skip, because flag1 raised.
    R5:F,B->Z not applied, because of lacking F.
    R6:C,D->F apply. Raise flag1. Facts A, B, C, L, K, D, F.

  ITERATION 6
    R1:A->L skip, because flag1 raised.
    R2:L->K skip, because flag1 raised.
    R3:D->A skip, because flag2 raised.
    R4:D->M skip, because flag1 raised.
    R5:F,B->Z apply. Raise flag1. Facts A, B, C, L, K, D, F, Z.
    Goal achieved.

PART 3. Results
  1) Goal Z achieved.
  2) Path R1, R2, R7, R4, R6, R5.
```

3. **Čyras vs. Negnevitsky; Čyras wins**
Rules see in the table.
Fact A.
Goal Z.
Path R2, R1.



R1:	G → Z
R2:	A → G
R3:	A → B
R4:	B → C
R5:	C → D
R6:	D → Z

3. **Čyras vs. Negnevitsky; Negnevitsky laimi,** bet su atliekama taisykle.
Rules see in the table.
Fact A.
Goal Z.
Path R4, R3, R2, R1.



R1:	D → Z
R2:	C → D
R3:	B → C
R4:	A → B
R5:	A → G
R6:	G → Z

4. **Goal in facts**

```
PART 3. Results
  Goal A in facts. Empty path.
```

5. **No path**
Rules R1: A → B; R2: C → Z. Fact A. Goal Z.

6. **Negnevitsky's example**
Take an example with 5 rules from the section about forward chaining by Negnevitsky (2011).

18.8. Testing Backward Chaining

1. **Failed branch**
Rules R1: C, D → Z; R2: T → C; R3: T → Z. Fact T. Goal Z. Path R3. Note that the path is not R2, R3. Facts in the end are T and Z and not T and C, Z. This happens due to backtracking and failed branch from Z to C.

```
PART 2. Trace

 1).Goal Z. Find R1:C,D->Z. New goals C, D.
 2)..Goal C. Find R2:T->C. New goals T.
 3)...Goal T. Fact (initial), as facts are T. Back, OK.
 4)..Goal C. Fact (presently obtained). Facts T and C.
 5)..Goal D. No rules. Back, FAIL.
 6).Goal Z. Find R3:T->Z. New goals T.
 7)..Goal T. Fact (initial), as facts are T. Back, OK.
 8).Goal Z. Fact (presently obtained). Facts T and Z. Back, OK.

PART 3. Results
  1) Goal Z derived.
  2) Path R3.
```

2. **Nine rules D, C**

Rules see in the table.

Fact T.

Goal Z.

Path R6, R5, R4, R3, R1.

R1:	D, C → Z
R2:	C → D
R3:	B → C
R4:	A → B
R5:	D → A
R6:	T → D
R7	G → A
R8	H → B
R9	J → C

```
PART 2. Trace

 1).Goal Z. Find R1:D,C->Z. New goals D, C.
 2)..Goal D. Find R2:C->D. New goals C.
 3)...Goal C. Find R3:B->C. New goals B.
 4)....Goal B. Find R4:A->B. New goals A.
 5).....Goal A. Find R5:D->A. New goals D.
 6)......Goal D. Loop. Back, FAIL.
 7).....Goal A. Find R7:G->A. New goals G.
 8)......Goal G. No rules. Back, FAIL.
 9).....Goal A. No more rules. Back, FAIL.
10)....Goal B. Find R8:H->B. New goals H.
11).....Goal H. No rules. Back, FAIL.
12)....Goal B. No more rules. Back, FAIL.
13)...Goal C. Find R9:J->C. New goals J.
14)....Goal J. No rules. Back, FAIL.
15)...Goal C. No more rules. Back.
16)..Goal D. Find R6:T->D. New goals T.
17)...Goal T. Fact (initial), as facts T. Back, OK.
18)..Goal D. Fact (presently obtained). Facts T and D.
19)..Goal C. Find R3:B->C. New goals B.
20)...Goal B. Find R4:A->B. New goals A.
21)....Goal A. Find R5:D->A. New goals D.
22).....Goal D. Fact (earlier obtained), as facts T and D. Back, OK.
23)....Goal A. Fact (presently obtained). Facts T and D, A. Back, OK.
24)...Goal B. Fact (presently obtained). Facts T and D, A, B. Back, OK.
25)..Goal C. Fact (presently obtained). Facts T and D, A, B, C. Back, OK.
26).Goal Z. Fact (presently obtained). Facts T and D, A, B, C, Z. Back, OK.

PART 3. Results
  1) Goal Z derived.
  2) Path R6, R5, R4, R3, R1.
```

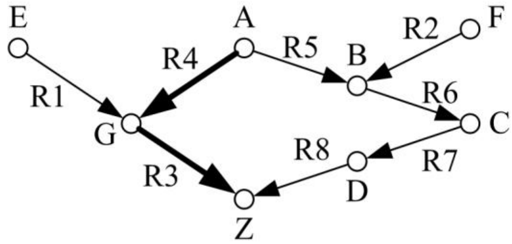
3. **Nine rules C, D**

See above but R1: C, D → Z. Same path.

4. **A loop and a failed branch**

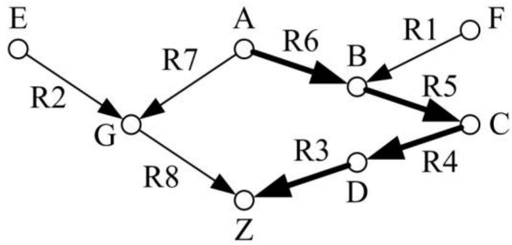
Rules R1: A → Z; R2: B → A; R3: A, C → B; R4: T → B; R5: T → C. Fact T. Goal Z. Path R4, R2, R1.

5. **Graph and a short path**
Rules in the table.
Fact A.
Goal Z.
Path R4, R3.



R1:	E → G
R2:	F → B
R3:	G → Z
R4:	A → G
R5:	A → B
R6:	B → C
R7:	C → D
R8:	D → Z

6. **Graph and a long path**
Rules in the table.
Fact A.
Goal Z.
Path R6, R5, R4, R3.



R1:	F → B
R2:	E → G
R3:	D → Z
R4:	C → D
R5:	B → C
R6:	A → B
R7:	A → G
R8:	G → Z

7. **Three alternative rules**
Rules R1: A → Z; R2: B → Z; R3: C → Z. Fact C. Goal Z. Path R3.

8. **Three alternatives and an unachievable goal**
Rules R1: A, D → Z; R2: B, D → Z; R3: C, D → Z; R4: C, E → Y. Facts C, D. Goal Y. No path.

9. **Negnevitsky's example**
Take an example with 5 rules from the section about backward chaining by Negnevitsky (2011).

19. Resolution

The resolution method is introduced with an example from [Thayse et al. 1990, p. 165]. Suppose a statement “A professor can examine students of a distinct faculty”. Suppose two facts:

1. Žakas is a professor at the faculty of informatics;
2. Mari is a student at the faculty of mathematics.

The task is to prove that Žakas is allowed to examine Mari.

Two facts are represented with predicates:

Fact F1: `Prof(Info,Žakas)` (19.1)

Fact F2: `Stud(Mat,Mari)` (19.2)

The statement “A professor can examine students of a distinct faculty” is represented with the following rule:

$$\texttt{Prof(x,y)} \ \& \ \texttt{Stud(z,w)} \ \& \ \neg\texttt{Lygu(x,z)} \Rightarrow \texttt{Egz(y,w)} \tag{19.3}$$

Implication $F \Rightarrow G$ can be replaced by conjunction and negation $F \vee G$. This replacement can be treated as a term rewriting rule:

$$\frac{F \Rightarrow G}{\neg F \vee G} \quad \text{Implication elimination} \tag{19.4}$$

The implication elimination rule above is sound. The tables of both formulae $F \Rightarrow G \equiv \neg F \vee G$ coincide:

F	G	¬F	¬F ∨ G	F ⇒ G
t	t	f	t	t
t	f	f	f	f
f	t	t	t	t
f	f	t	t	t

Implication elimination can be generalised as follows:

$$\frac{F_1 \ \& \ F_2 \ \& \ \ldots \ \& \ F_n \Rightarrow G}{\neg F_1 \vee \neg F_2 \vee \ldots \vee \neg F_n \vee G} \tag{19.5}$$

Following is the rule of double negation elimination. It is sound because $\neg\neg F \equiv F$:

$$\frac{\neg\neg F}{F} \quad \text{Double negation elimination} \tag{19.6}$$

We rewrite rule (19.3) using (19.5) and obtain

$$\neg\texttt{Prof(x,y)} \vee \neg\texttt{Stud(z,w)} \vee \neg\neg\texttt{Lygu(x,z)} \vee \texttt{Egz(y,w)}$$

Double negation is eliminated:

$$\neg\texttt{Prof(x,y)} \vee \neg\texttt{Stud(z,w)} \vee \texttt{Lygu(x,z)} \vee \texttt{Egz(y,w)} \tag{19.7}$$

In order to the resolution rule we start with the *modus ponens* rule.:„1) Suppose F. 2) Suppose if F, then G. 3) Therefore G“.

$$\frac{\begin{array}{l} F \\ F \Rightarrow G \end{array}}{G} \qquad \begin{array}{l} \text{1) Small premise} \\ \text{2) Great premise} \\ \text{3) Conclusion} \end{array} \qquad \textit{modus ponens}$$

Another representation of the *modus ponens* rule:

$$\frac{F, \quad F \Rightarrow G}{G} \qquad \textit{modus ponens}$$

An example of *modus ponens* application:

$$\frac{\begin{array}{l} \text{1) } \mathrm{human(Socrates)} \\ \text{2) } \forall x\ \mathrm{human}(x) \Rightarrow \mathrm{mortal}(x) \end{array}}{\text{3) } \mathrm{mortal(Socrates)}} \quad \{\mathrm{Socrates}/x\}$$

Resolution rule is represented as follows:

$$\frac{F, \quad \neg F \vee G}{G}\ \sigma \qquad \textbf{Resolution rule} \text{ (simple form)} \qquad (19.8)$$

Formulae of form $\forall x_1 \forall x_2 \ldots \forall x_n\ G(x_1, x_2, \ldots, x_n)$ are considered; see [Norgėla 2007, p. 26, 92, 158–163], [Nilsson 1998, p. 253–268] etc.

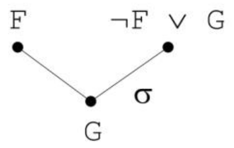


Fig. 19.1. Graphuical representation of resolution rule

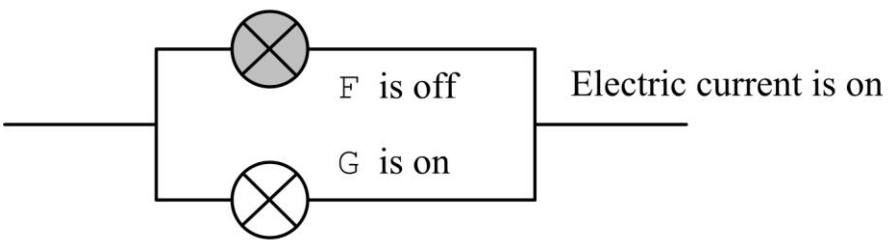


Fig. 19.2. Parallel circuit of two bulbs is modelled with disjunction $F \vee G$. The statement „Bulb F is off“ (i.e. the bulb is broken) is represented $\neg F$. A conclusion of thes two statements is that the bulb G is on

Another form of representing resolution rule:

$$\frac{\neg F, \quad F \lor G}{F} \tag{19.9}$$

A general case:

$$\frac{F \lor G, \quad \neg F \lor H}{G \lor H} \quad \{A_1/x_1,\ A_2/x_2, \ldots A_n/x_n\} \qquad \textbf{Resolution rule (general)} \tag{19.10}$$

Variables x_i, if any, are replaced with A_i; see Fig. 9.3. The substitution σ is of the form $\{A_1/x_1,\ A_2/x_2, \ldots A_n/x_n\}$.

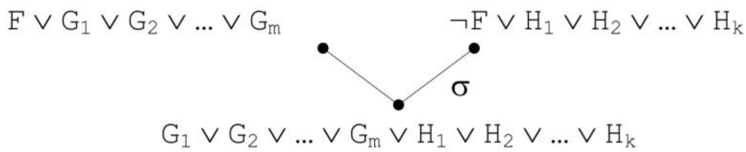


Fig. 19.3. Graphical representation of resolution rule. m, $k \geq 0$

19.1. Inference Example

In order to prove `Egz(Žakas,Mari)` start from the opposite:

$$\neg \mathtt{Egz(Žakas,Mari)} \tag{19.11}$$

Take (19.11) and rule (19.7) and apply resolution rule (19.9). In order to reduce the negative disjunct ¬`Egz(Žakas,Mari)` with the positive one `Egz(y,w)`, the substitution is `{Žakas/y, Mari/w}`.

$$\frac{\neg\mathtt{Egz(Žakas,Mari)}, \quad \neg\mathtt{Prof(x,Žakas)}\lor\neg\mathtt{Stud(z,Mari)}\lor\mathtt{Lygu(x,z)}\lor\mathtt{Egz(Žakas,Mari)}}{\neg\mathtt{Prof(x,Žakas)}\lor\neg\mathtt{Stud(z,Mari)}\lor\mathtt{Lygu(x,z)}}$$

Further tale the conclusion above and fact F1 (19.1) and apply resolution rule (19.8). The substitution is `{Info/x}`.

$$\frac{\neg\mathtt{Prof(Info,Žakas)}\lor\neg\mathtt{Stud(z,Mari)}\lor\mathtt{Lygu(Info,z)}, \quad \mathtt{Prof(Info,Žakas)}}{\neg\mathtt{Stud(z,Mari)}\lor\mathtt{Lygu(Info,z)}}$$

Further tale the conclusion above and fact F2 (19.2) and apply resolution rule (19.9). The substitution is `{Mat/z}`.

$$\frac{\neg\mathtt{Stud(Mat,Mari)}\lor\mathtt{Lygu(Info,Mat)}, \quad \mathtt{Stud(Mat,Mari)}}{\mathtt{Lygu(Info,Mat)}}$$

Faculties of informatics and mathematics are distinct objects, formally, `Lygu(Info,Mat) = false`. Therefore the conclusion above is `false`. A contradiction is obtained. Thus goal negation leads to contradiction. Therefore the goal Egz(Žakas,Mari) is true. The inference tree is shown in Fig. 19.4.

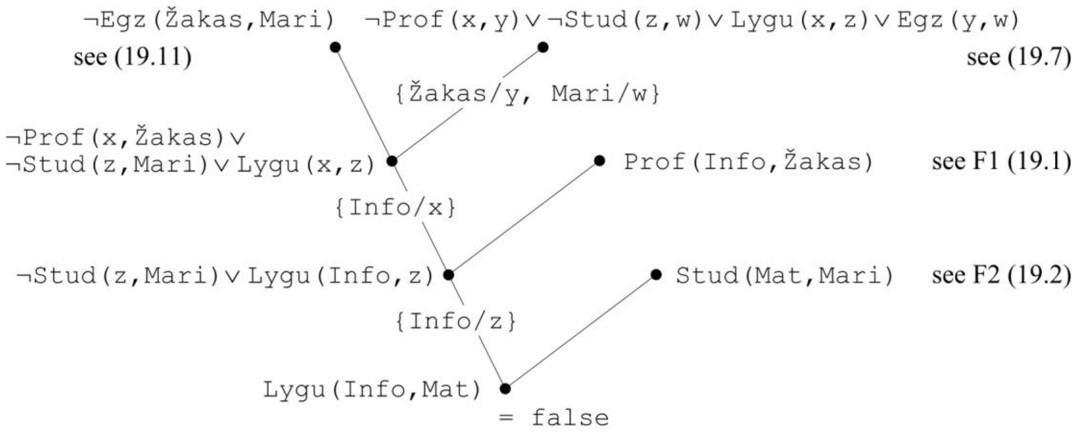


Fig. 19.4. Inference tree of statement Egz(Žakas,Mari). Statement's negation leads to contradiction

End of the proof backward.

The proof forward – from the facts to the goal. The data base consists of facts F1 (19.1) and F2 (19.2) and rule R1 (19.7). Need to prove Egz(Žakas,Mari).

Take fact F1 Prof(Info,Žakas) and search a rule to match. Take rule (19.7) and apply resolution rule (19.8). The substitution is {Info/x, Žakas/y}:

Prof(Info,Žakas), ¬Prof(Info,Žakas)∨¬Stud(z,w)∨Lygu(Info,z)∨Egz(Žakas,w)

¬Stud(z,w)∨Lygu(Info,z)∨Egz(Žakas,w)

Then take fact F2 Stud(Mat,Mari) and match with the conclusion above. Apply resolution rule (19.8) with the substitution {Mat/z, Mari/w}:

Stud(Mat,Mari), ¬Stud(Mat,Mari)∨Lygu(Info,Mat)∨Egz(Žakas,Mari)

Lygu(Info,Mat)∨Egz(Žakas,Mari)

As Lygu(Info,Mat) = false and false∨H≡ H, therefore Egz(Žakas,Mari). Q.e.d. The inference tree is shown in Fig. 19.5.

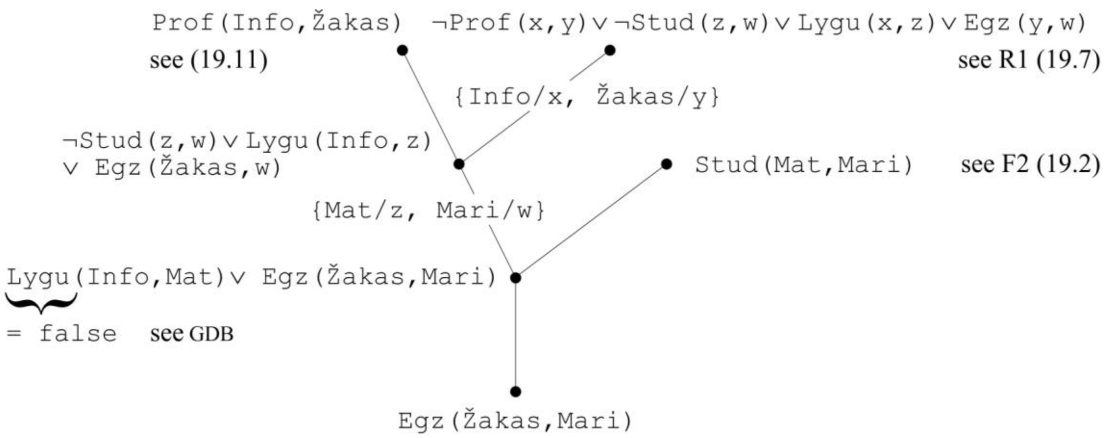


Fig. 19.5. Inference tree to prove `Egz(Žakas,Mari)`. This is a direct forward proof – from the facts to the goal

End of the direct (forward) proof.

19.2. Example with Three Rules

Suppose the rules (18.1). Implication elimination leads to normal disjunctive form below.

	Rule	Formula	Implication eliminated
π_1:	F, B → Z	F&B ⇒ Z	¬F∨¬B∨Z
π_2:	C, D → F	C&D ⇒ F	¬C∨¬D∨F
π_3:	A → D	A ⇒ D	¬A∨D

Suppose the facts:

Fact F1: A
Fact F2: B
Fact F3: C

In other words, `A=true`, `B=true` ir `C=true`. Proof Z.

Proof backward. Start from the goal's negation ¬Z.

Search for a rule to match. The rule π_1 is found. Apply resolution rule (19.9):

$$\frac{\neg Z, \qquad \neg F \vee \neg B \vee Z}{\neg F \vee \neg B}$$

For the conclusion ¬F ∨ ¬B above search another rule to match. Find π_2. Then apply resolution rule (19.10):

$$\frac{\neg F \vee \neg B, \qquad \neg C \vee \neg D \vee F}{\neg B \vee \neg C \vee \neg D}$$

For the conclusion ¬B∨¬C∨¬D above search another rule to match. Find π_3. Then apply resolution rule (19.10):

$$\frac{\neg B\vee\neg C\vee\neg D, \qquad \neg A\vee D}{\neg B\vee\neg C\vee\neg A}$$

The global data base contains `A`, `B` and `C`, i.e `A=true`, `B=true` and `C=true`, and therefore `false` is derived. A contradiction is obtained. This means that the goal is true.

`false` can be derived fro the conclusion $\neg B\vee\neg C\vee\neg A$ above. Apply resolution rule (19.8) three times. This is shown below.

Take fact `F1`, i.e. `A`:

$$\frac{A, \qquad \neg B\vee\neg C\vee\neg A}{\neg B\vee\neg C}$$

Take fact `F2`, i.e. `B`:

$$\frac{B \qquad \neg B\vee\neg C}{\neg C}$$

Take fact `F3`, i.e. `C`:

$$\frac{C, \qquad \neg C}{\varnothing}$$

Empty clause means contradiction ($C\&\neg C$ = `false`) Hence, goal's negation $\neg Z$ leads to contradiction. Therefore the goal is true. The proof tree is shown in Fig. 19.6.

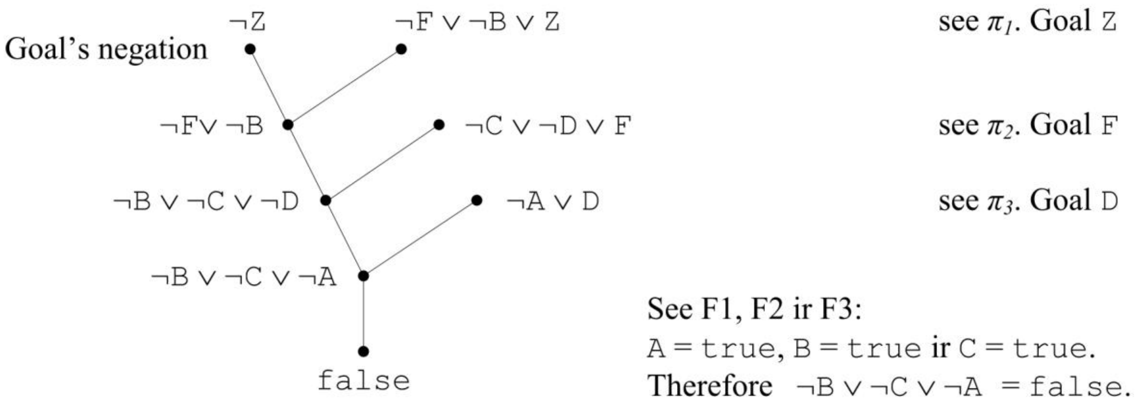


Fig. 19.6. The proof tree of `Z`. Inference is backward: from goal negation to contradiction

End of proof backward.

Intelligence is in a sequence π_1, π_2, π_3 to match. The plan obtained is $\langle\pi_3, \pi_2, \pi_1\rangle$.

Proof forward (direct inference). Let us start from facts.

Take fact F1, i.e. `A`. Sear rule to match. Find π_3, i.e. $\neg A \vee D$. Apply resolution rule (19.8):

$$\frac{A, \qquad \neg A\vee D}{D}$$

Search rule to match. Find π_2, i.e. $\neg C \vee \neg D \vee F$. Apply resolution rule (19.8):

$$\frac{D, \qquad \neg C \vee \neg D \vee F}{\neg C \vee F}$$

Take fact F3, i.e. C. Apply the resolution rule (19.8):

$$\frac{C \qquad \neg C \vee F}{F}$$

Search a rule to match. Find π_1, t. y. $\neg F \vee \neg B \vee Z$. Apply resolution rule (19.9):

$$\frac{F, \qquad \neg F \vee \neg B \vee Z}{\neg B \vee Z}$$

Take fact F2, i.e. B. Apply resolution rule (19.9):

$$\frac{B, \qquad \neg B \vee Z}{Z}$$

The goal Z is obtained in the conclusion above. Q.e.d. Proof tree is shown in Fig. 19.7.

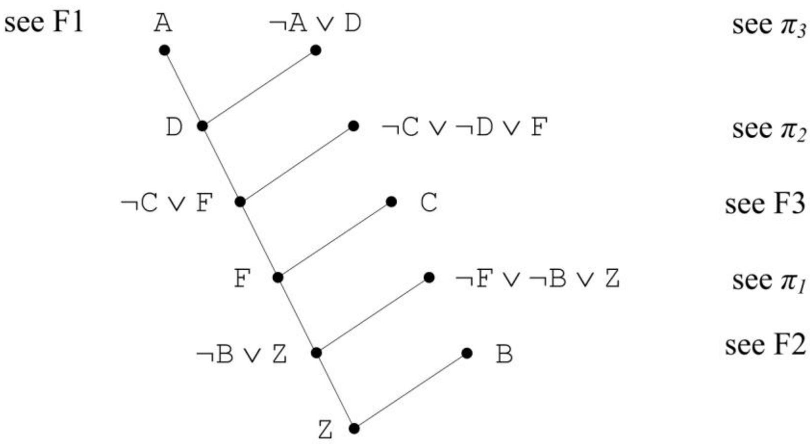


Fig. 19.7. Proof tree of Z. The proof is direct forward: from facts to the goal

End of proof forward.

19.3. Using Resolution to Prove Theorem

This section follows [Nilsson 1998, 16.5, 260–261].

(1) $\forall$ x,y Package(x) & Package(y) & Inroom(x,27) & Inroom(y,28) $\Rightarrow$ Smaller(x,y)

Shortly:

(2) $\neg P(x) \vee \neg P(y) \vee \neg I(x,27) \vee \neg I(y,28) \vee S(x,y)$

Robot knows that A is in room 27 or 28 (but does not know in which one), formally, I(A,27) ∨ I(A,28). Robot knows that B is in room 27, I(B,27). Robot knows that B is not smaller than A, ¬S(B,A). Hence:

(3) P(A). Fact.

(4) P(B). Fact.

(5) I(A,27) ∨ I(A,28). Robot knows.

(6) I(B,27). Robot knows.

(7) ¬S(B,A). Robot knows.

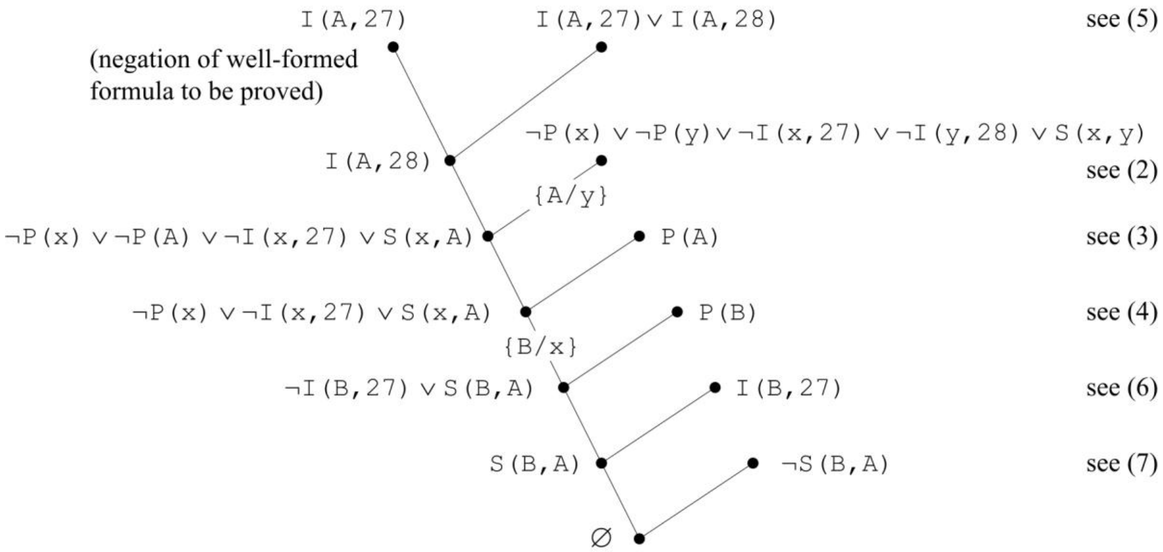


Fig. 19.8. Proof tree; adapted from [Nilsson 1998, p. 261]

A possible world is shown in Fig. 19.9.

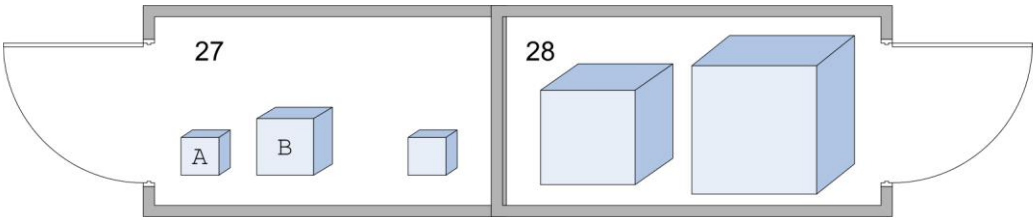


Fig. 19.9. A possible world the robot knows about

20. Expert Systems

See e.g. Brian Sawyer and Dennis Foster (1986). A general architecture of an expert system is shown in Fig. 20.1.

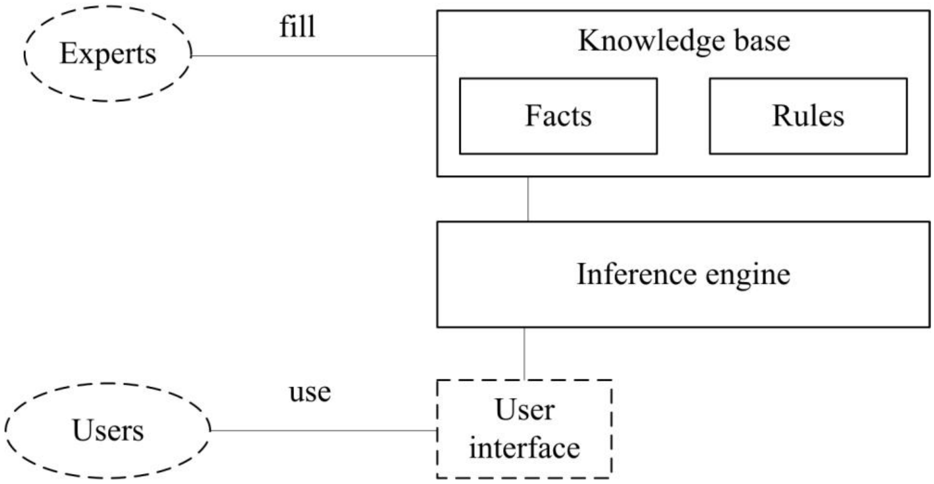


Fig. 20.1. A general architecture of an expert system

Following is an example from [Sawyer, Foster 1986]. Facts are shown in Fig. 20.2.

1) amžius: 1. 25_ir_mažiau 2. **25-55** 3. 55_ir_daugiau	2) lytis: 1. vyras 2. **moteris**	3) svoris: 1. **55_ir_mažiau** 2. 55-85 3. 85_ir_daugiau
4) sudėjimas: 1. **smulkus** 2. stambus	5) cholesterolis: 1. **vidutinis** 2. daug	6) druska: 1. **norma** 2. daug
7) rūko: 1. taip 2. **ne**	8) charakteris: 1. **agresyvus** 2. švelnus	9) alkoholis: 1. nevartoja 2. **vidutiniškai** 3. daug

Fig. 20.2. Facts in an expert system

Several rules of total about 100 are shown below:

Pr.1	Jeigu santykinis svoris normalus, tai bendras įvertinimas „taip“.
Pr.4	Jeigu širdies sustojimo rizika yra vidutinė, tai širdies pavojus žemas.
Pr.8	Jeigu amžius 25-55 metai ir lytis moteris, tai bendra gyvenimo trukmė 67 metai.
Pr.12	Jeigu svoris ne didesnis nei 55 kilogramai, sudėjimas smulkus ir lytis moteris, tai santykinis svoris yra normalus.
Pr.27	Jeigu cholesterolio suvartojama vidutiniškai, tai širdies sustojimo rizika vidutinė.
Pr.29	Jeigu druskos suvartojimas yra normos ribose, tai kraujo spaudimas vidutinis.

Pr.37	Jeigu bendras įvertinimas yra „taip“, širdies pavojus yra žemas, kraujo spaudimas yra vidutinis ir nerūko, tai perspektyva gera.
Pr.43	Jeigu charakteris agresyvus, tai asmenybės tipas A.
Pr.45	Jeigu asmenybės tipas A, tai rizikos lygis yra aukštas.
Pr.47	Jeigu alkoholio vartoja vidutiniškai, tai papildomas faktorius yra geras.
Pr.52	Jeigu perspektyva yra gera, rizikos lygis yra aukštas ir papildomas faktorius yra geras, tai faktorius yra nulis.
Pr.64	Jeigu bendra gyvenimo trukmė yra 67 metai ir faktorius yra nulis, tai gyvenimo trukmė yra 67 metai.

Fig. 20.3. Rules in an expert system

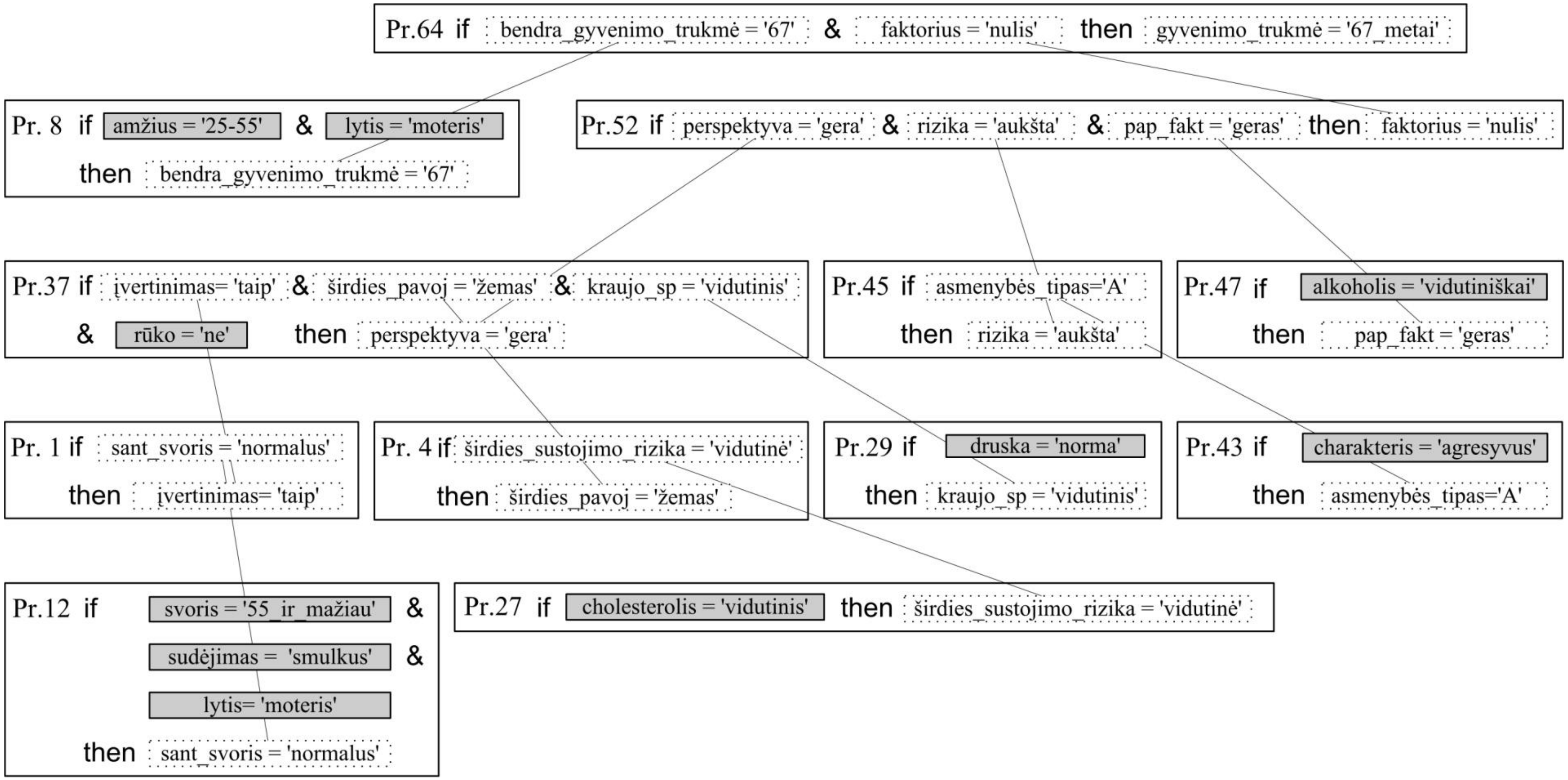


Fig. 20.4. A samplederivation tree in an expert system. Facts are in grey rectangles

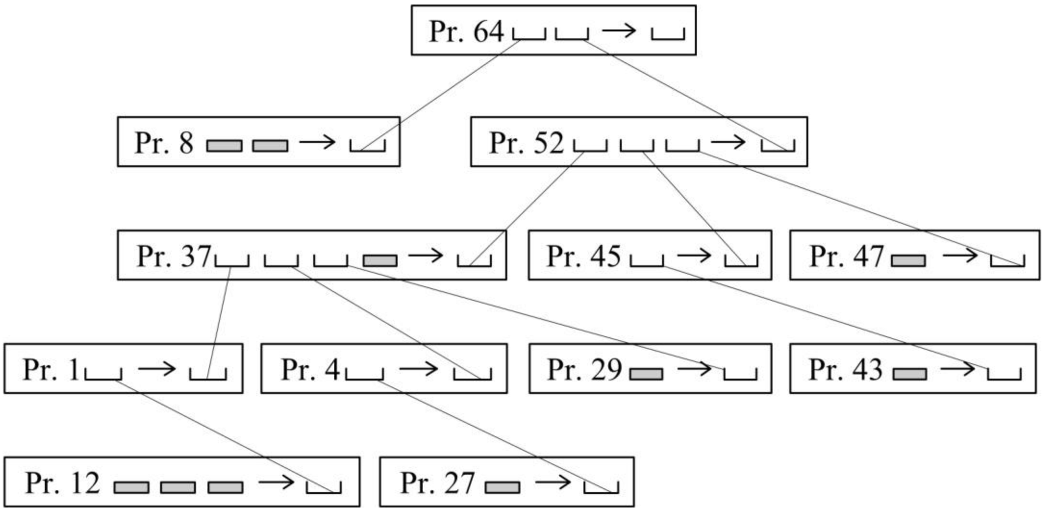


Fig. 20.5. A simplified proof tree

A rule

$$o_1=r_1\ ,\ o_2=r_2\ ,\ \dots\ ,\ o_n=r_n \rightarrow o_{n+1}=r_{n+1}$$

means

$$\textit{if } o_1=r_1\ \&\ o_2=r_2\ \&\ \dots\ \&\ o_n=r_n \textit{ then } o_{n+1}=r_{n+1}$$

21. Internet Shopping

This section expands [Russell, Norvig 2003, p. 344–348].

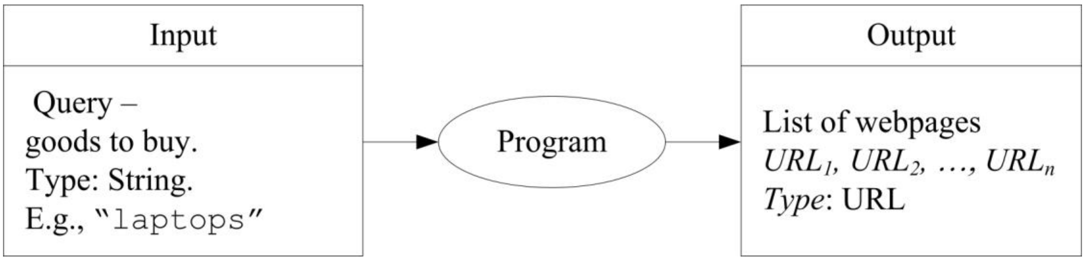
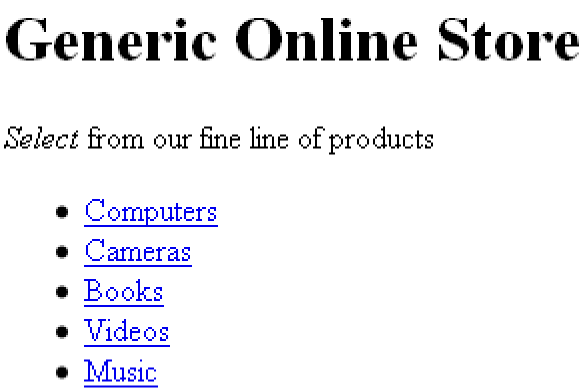


Fig. 21.1. Input-output description of the internet shopping program

The program is treated as a computer agent (Fig. 21.2); see [Russell, Norvig 2003].

Fig. 21.2. Input-output description of computer agent

A typical internet shop is shown in Fig. 21.3.


Generic Online Store

Select from our fine line of products

- Computers
- Cameras
- Books
- Videos
- Music

```
<html>
<body>
<h1>Generic Online Store</h1>
<i>Select</i> from our fine line of products
<ul>
<li><a href="http://www.gen-store.com/compu">Computers</a>
<li><a href="http://www.gen-store.com/camer">Cameras</a>
<li><a href="http://www.gen-store.com/books">Books</a>
<li><a href="http://www.gen-store.com/video">Videos</a>
<li><a href="http://www.gen-store.com/music">Music</a>
</ul>
</body>
</html>
```

Fig. 21.3. A web page which is perceived an a internet. We build under address http://www.gen-store.com ir pavadinsime GenStore

We use top-down refinement to formulate requirements.

Step 1 etapas. Predicate **RelevantOffer**

```
RelevantOffer(page,url,query) ⇔
     Relevant(page,url,query) & Offer(page)
```

Types: `page` – HTML text, `url` – URL, `query` – String.

Step 2. Predicate **Offer**

Tags „a“ or „form“ should contain *buy* or *price*:

```
Offer(page)              ⇔    ( InTag('a',str,page) ∨
                                 InTag('form',str,page) )
                             & ( In('buy',str) ∨ In('price',str) )
InTag(tag,str,page)  ⇔  In('<'+tag+str+'</'+tag+'>', page)
In(sub,str)              ⇔  ∃i str[i:i+Length(sub)-1] = sub
```

For example,

```
In('KAD', 'ABRAKADABRA') = true
In('laptop', 'ABRAKADABRA') = false
'ABC' + 'DEFG' = 'ABCDEFG'
```

Step 3. Predicate **OnlineStores**(store)
For example,

```
Amazon ∊ OnlineStores
          & Homepage(Amazon, 'http://www.amazon.com')
Ebay ∊ OnlineStores
          & Homepage(Ebay, 'http://www.ebay.com')
GenStore ∊ OnlineStores
          & Homepage(GenStore, 'http://www.gen-store.com')
```

Step 4. Predicate **Relevant**

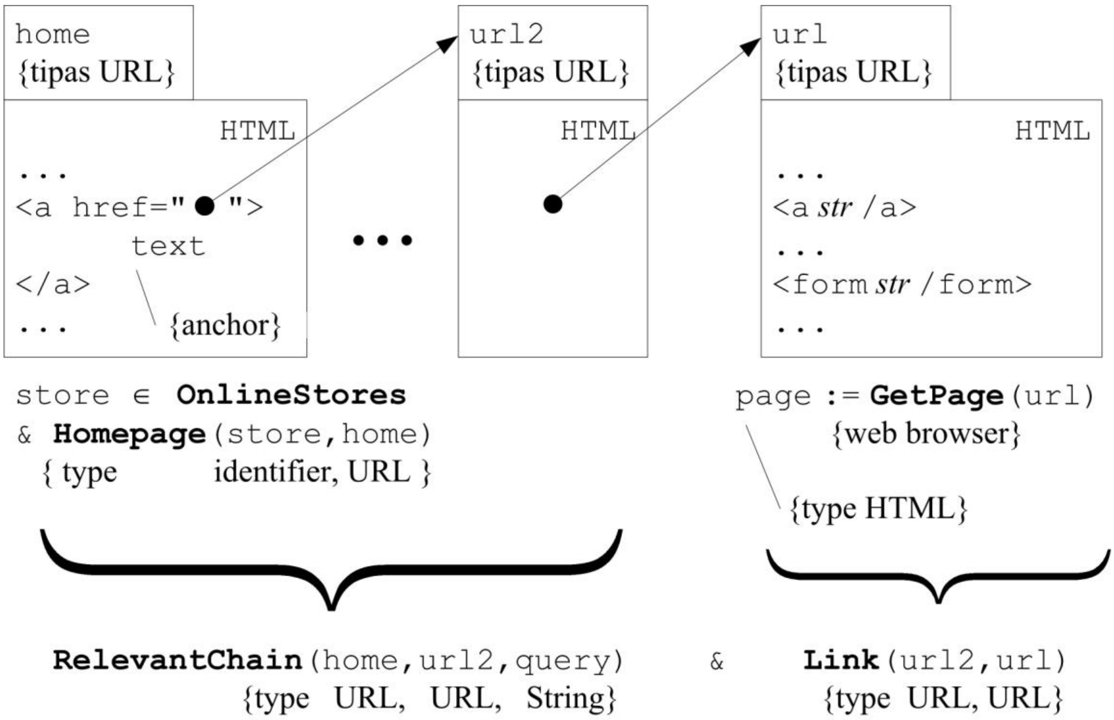


Fig. 21.4. Internet shopping specification. For example, `home = 'http://www.gen-store.com'` and `store = GenStore`

Predicate **Relevant** is decomposed further:

```
Relevant(page,url,query) ⇔
 ∃ store ∃ home ( store∊OnlineStores & Homepage(store,home)
 & ∃url2 RelevantChain(home,url2,query) & Link(url2,url)
 & page = GetPage(url) )
```
(21.1)

Step 5. Predicate **RelevantChain**

```
RelevantChain(start,end,query) ⇔ (start = end)
   ∨ ( ∃u ∃text LinkText(start,u,text)
       & RelevantCategoryName(query,text)
       & RelevantChain(u,end,query) )
```

(21.2)

Types: `start` – URL, `end` – URL, `query` – String.

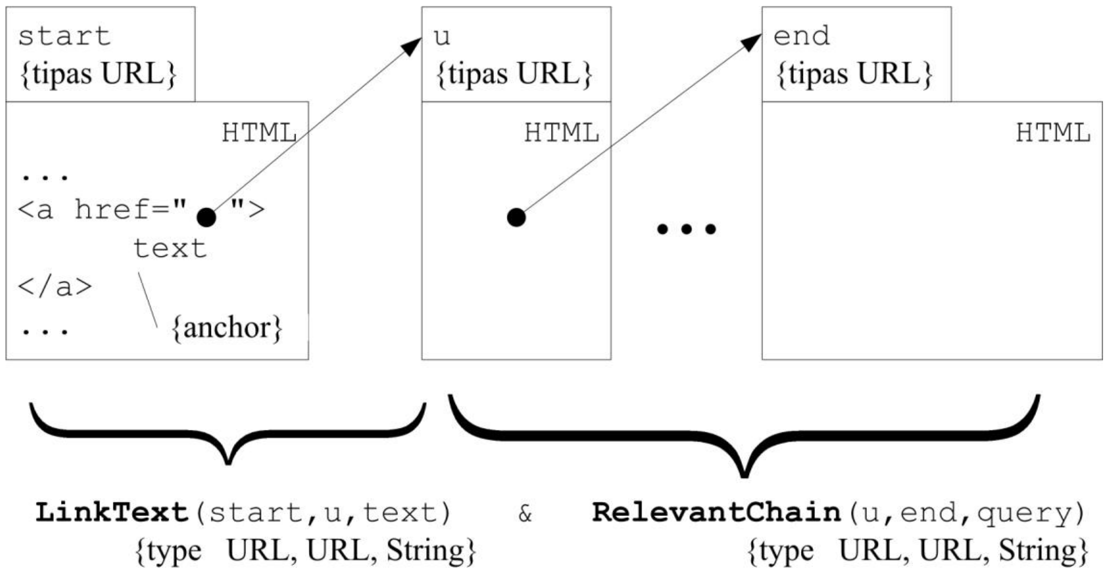


Fig. 21.5. Predicate **RelevantChain**(start,end,query)

For example, forward chaining specification is as follows:

```
ForwardChaining(s,t) ⇔ Link(s,u) & ForwardChaining(u,t)
```

Following is backward chaining specification:

```
BackwardChaining(s,t) ⇔ BackwardChaining(s,v) & Link(v,t)
```

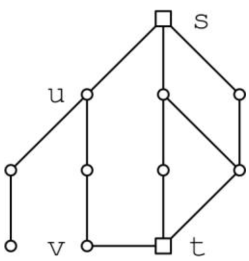


Fig. 21.6. Forward chaining and backward chaining algorithms can be used to search in a graph

Step 6. Assigning words to categories

Recall **RelevantChain**(start,end,query) (21.2):

LinkText(start,u,text) & **RelevantChain**(u,end,query)

Here, for example, text obtains value 'Computers', and query – 'I need laptops'.

Is-a categorisation is shown in Fig. 21.7.

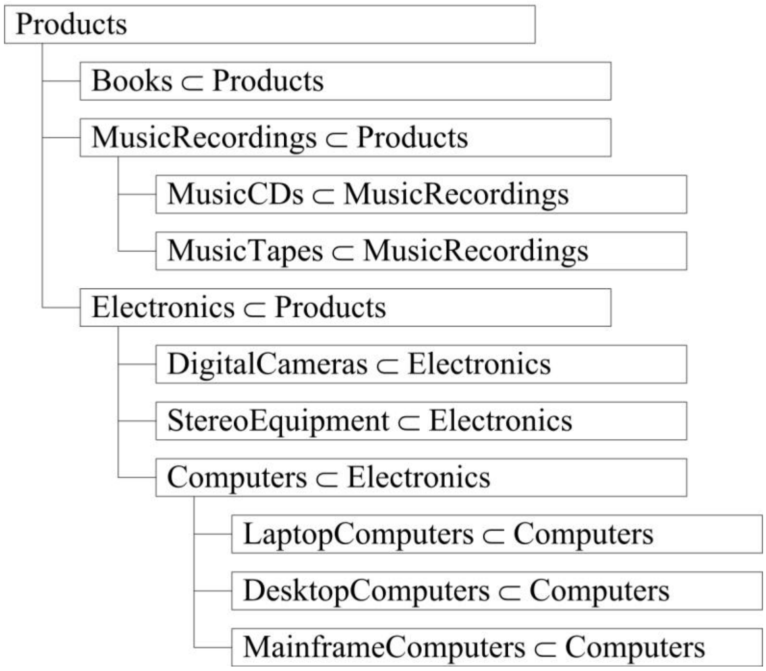


Fig. 21.7. Product hierarchy

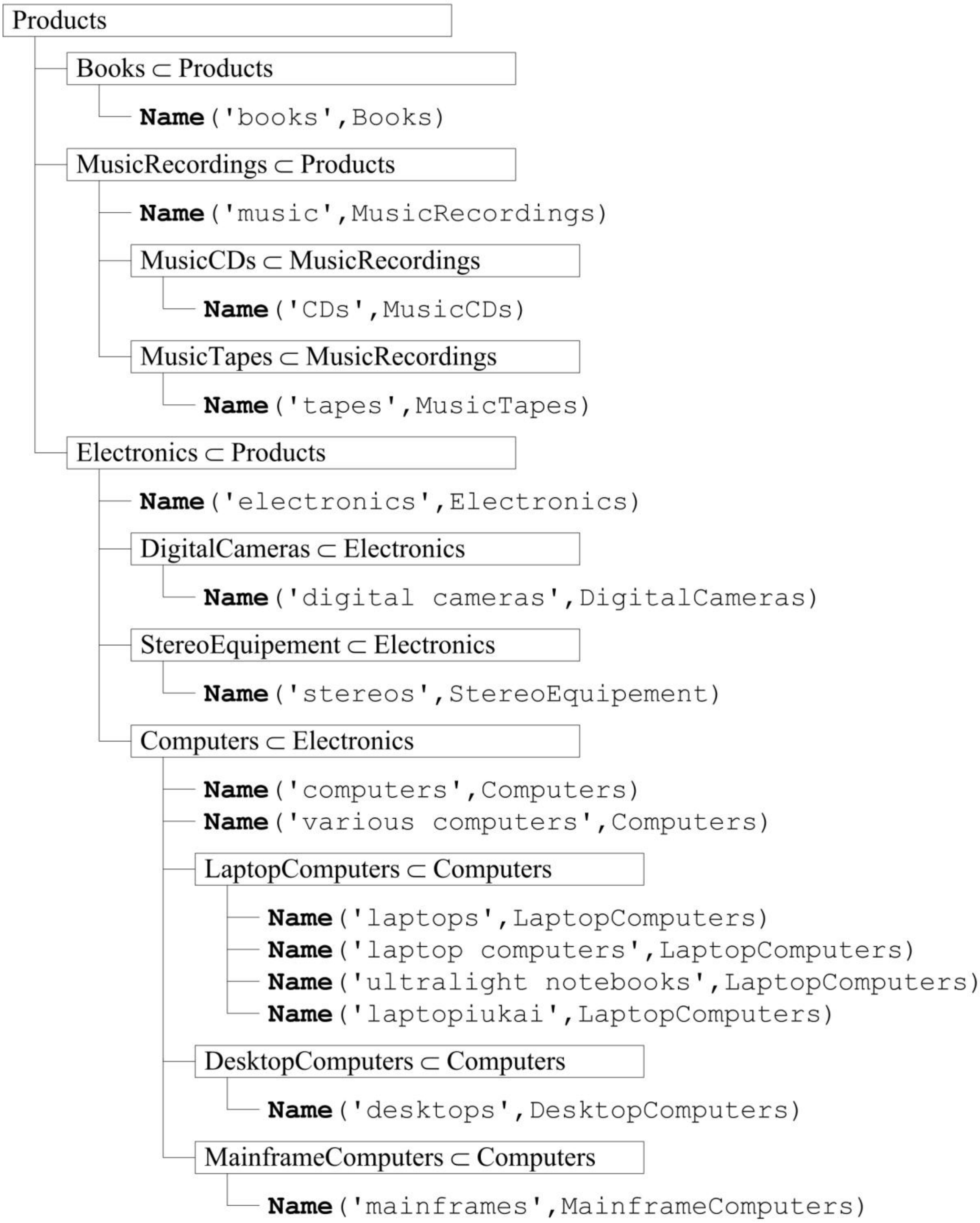


Fig. 21.8. Assigning words to categories

Step 7. Predicate **`RelevantCategoryName`**`(query,text)`

`RelevantCategoryName``(query,text)` ⇔
  `∃c1 ∃c2 Name(query,c1) & Name(text,c2) & (c1=c2 ∨ c1⊂c2 ∨ c1⊃c2)`

The case `c1⊂c2` is shown in Fig. 21.9 a and the case `c1⊃c2` in Fig. 21.9 b.

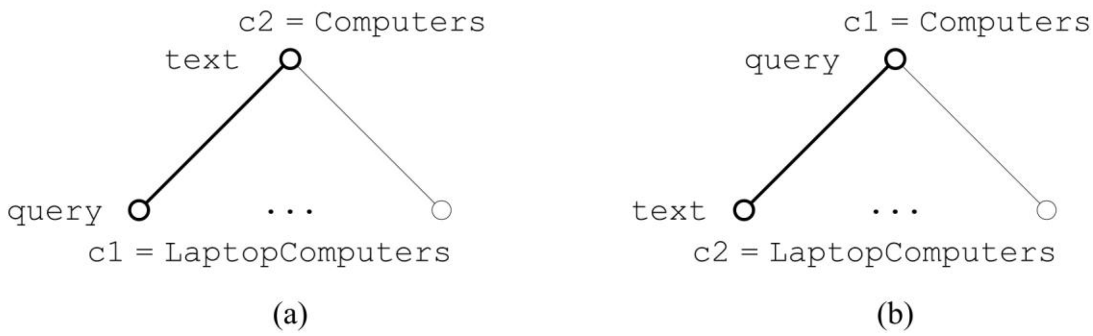


Fig. 21.9. (a) The case c1⊂c2. (b) The case c1⊃c2

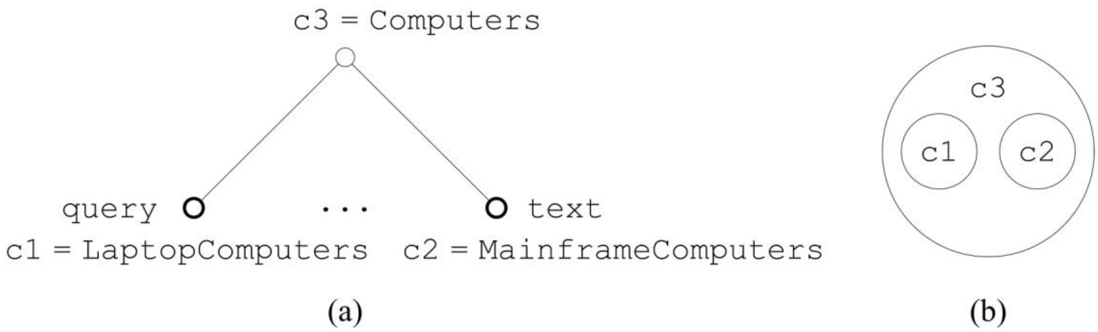


Fig. 21.10. (a) Predicate is not satisfied when none of relations c1=c2, c1⊂c2 and c1⊃c2 is satisfied. Categories c1 are c2 not comparable

22. The Turing Test

See textbooks, e.g. [Luger 2005] and Wikipedia http://en.wikipedia.org/wiki/Turing_test too.

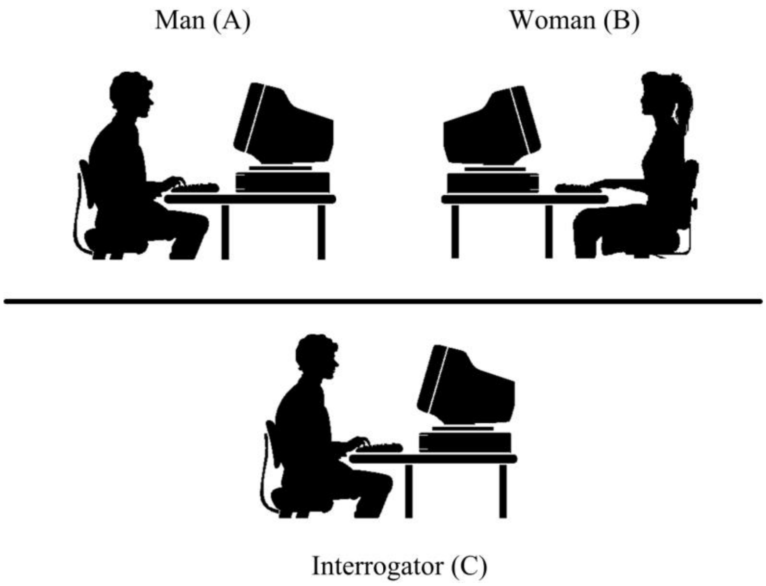


Fig. 22.1. Imitation game

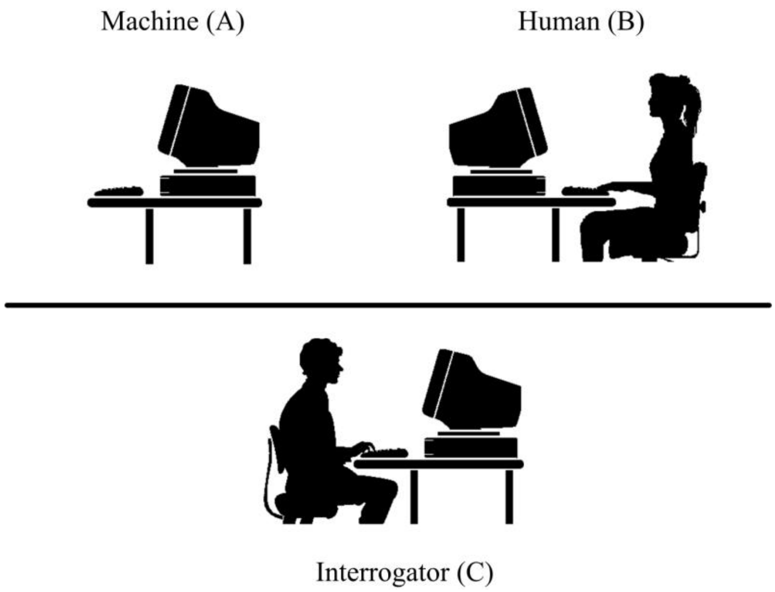


Fig. 22.2. The Turing test

See also John Searle's Chinese Room argument http://en.wikipedia.org/wiki/Chinese_room).

See also Weizenbaum's program Eliza http://en.wikipedia.org/wiki/ELIZA. A chat bot example see e.g. http://nlp-addiction.com/eliza.

23. Intension, Extension and Ontology

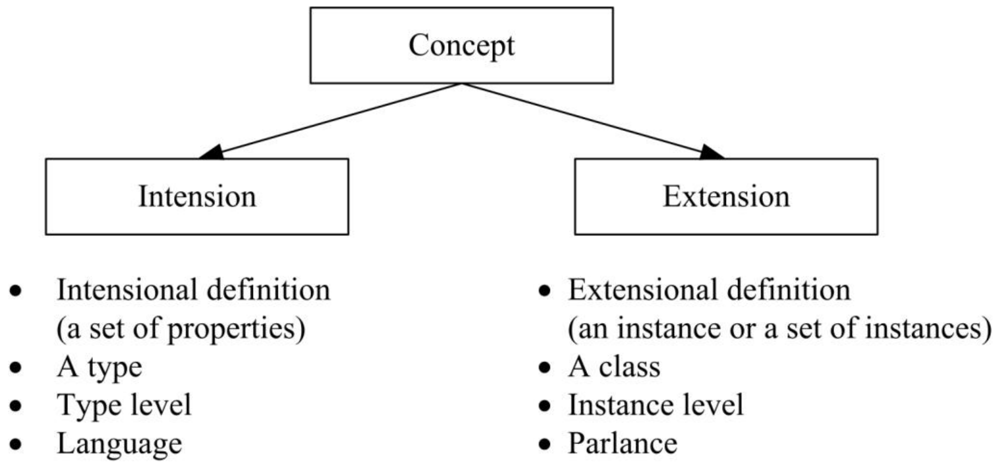


Fig. 23.1. The notion of a concept is a triad

See intensional definition http://en.wikipedia.org/wiki/Intensional_definition. For example, an intensional definition of bachelor is ‘unmarried man’. Being an unmarried man is an essential property of something referred to as a bachelor. It is a necessary condition: one cannot be a bachelor without being an unmarried man. It is also a sufficient condition: any unmarried man is a bachelor [Cook 2009].

This is the opposite approach to the extensional definition, which defines by listing everything that falls under that definition — an extensional definition of bachelor would be a listing of all the unmarried men in the world [Cook 2009].

As becomes clear, intensional definitions are best used when something has a clearly defined set of properties, and it works well for sets that are too large to list in an extensional definition. It is impossible to give an extensional definition for an infinite set, but an intensional one can often be stated concisely — there is an infinite number of even numbers, impossible to list, but they can be defined by saying that even numbers are integer multiples of two.

Definition by genus and difference, in which something is defined by first stating the broad category it belongs to and then distinguished by specific properties, is a type of intensional definition. As the name might suggest, this is the type of definition used in Linnaean taxonomy to categorize living things, but is by no means restricted to biology. Suppose we define a miniskirt as “a skirt with a hemline above the knee.” We’ve assigned it to a genus, or larger class of items: it is a type of skirt. Then, we've described the differentia, the specific properties that make it its own sub-type: it has a hemline above the knee.

Intensional definition also applies to rules or sets of axioms that generate all members of the set being defined. For example, an intensional definition of “square number” can be “any number that can be expressed as some integer multiplied by itself.” The rule—“take an integer and multiply it by itself”—always generates members of the set of square numbers, no matter which integer one chooses, and for any square number, there is an integer that was multiplied by itself to get it.

Similarly, an intensional definition of a game, such as chess, would be the rules of the game; any game played by those rules must be a game of chess, and any game properly called a game of chess must have been played by those rules.

23.1. Signs

Fig. 23.2. Graphical signs

In semiotics, a *signifier* refers to a *significant*, i.e. a *meaning* (Fig. 23.3). Semiotika yra mokslas apie ženklus ir jų aiškinimą (interpretavimą).

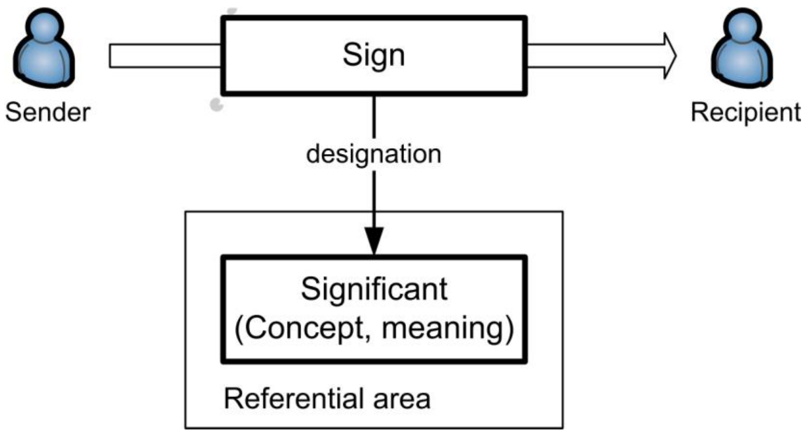


Fig. 23.3. A sign (i.e. a signifier) refers to its significant

See the problem of universals, http://en.wikipedia.org/wiki/Problem_of_universals. Example of universals: redness, cupness, homo sapiens.

Three viewpoints (Fig. 23.4):

- *universalia ante rem* (Platon)
- *universalia in re* (Aristotle)
- *universalia post rem* (nominalism, Latin *nomen* – name)

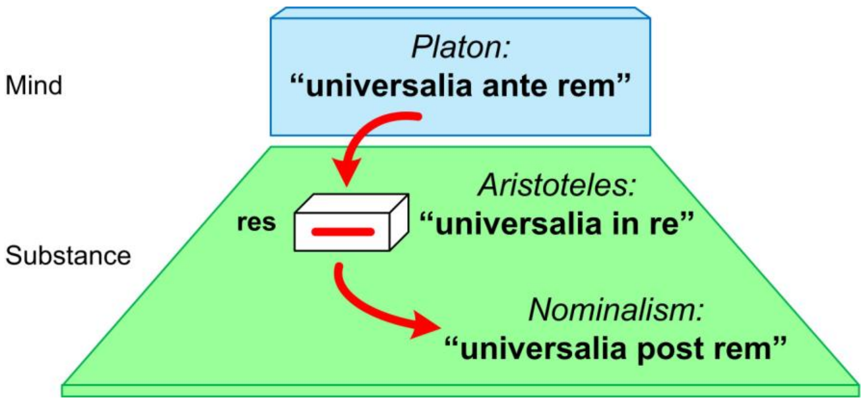


Fig. 23.4. Different viewpoints in philosophy

23.2. What is a Conceptualization?

This section follws [Guarino et al. 2009]. An ontology is a formal, explicit specification of a shared conceptualization [ibid., p. 3].

Definition 23.1 (Extensional relational structure $S = (D, \mathbf{R})$ **)** An extensional relational structure is a tuple $(D, \mathbf{R})$, where

- *D* is a set called the universe of discourse
- **R** is a set of relations on *D*. [Guarino et al. 2009, definition 2.1] □

Every element of **R** is an *extensional relation*, i.e., a mathematical relation, a subset of a Cartesian product.

Example 23.2. Let us consider human resources management in a large software company with 50.000 people, each one identified by a number (e.g., the social security number, or a similar code) preceded by letter *I*. Let us assume that our universe of discource *D* contains all these people, and that we are only interested in relations involving people. Our **R** will contain some unary relations, such as *Person*, *Manager*, and *Researcher* as well as the binary relations reports-to and cooperates-with.[2] [Guarino et al. 2009, example 2.1]

Consider an information system of the company. Let us assume that *D* contains 4 elements: $D = \{I1, I2, I3, I4\}$ (Fig. 23.5).

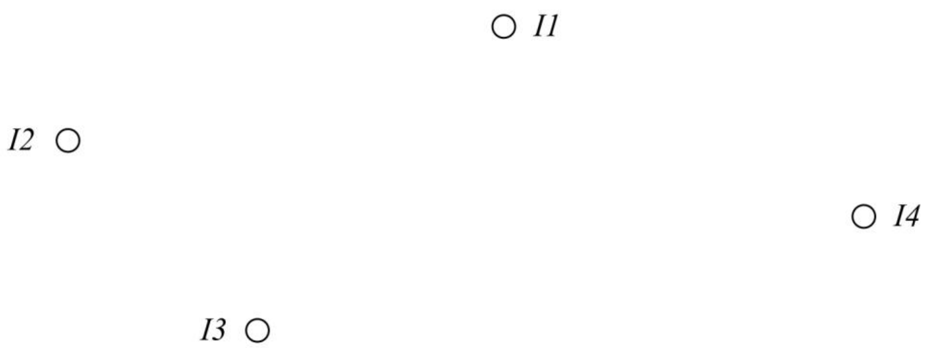


Fig. 23.5. The universe of discourse $D = \{I1, I2, I3, I4\}$

[2] The name of a person could also be assigned via relations, e.g., *firstname*(*I4*, '*Daniel*') and *lastname*(*I4*, '*Oberle*').

Let us assume 5 relations (5 tables):

$$\mathbf{R} = \{ \{ (I1), (I2), (I3), (I4) \}, \\ \{ (I1) \} \\ \{ (I2), (I3) \} \\ \{ (I2, I1), (I3, I1) \} \\ \{ (I2, I3) \} \}$$

Let us provide names (i.e. textual symbols) to the relations:

R = { *Person*, *Manager*, *Researcher*, *reports-to*, *cooperates-with* }, where

Person = { (*I1*), (*I2*), (*I3*), (*I4*) } – unary relation
Manager = { (*I1*) } – unary relation
Researcher = { (*I2*), (*I3*) } – unary relation
reports-to = { (*I2*, *I1*), (*I3*, *I1*) } – binary relation
cooperates-with = { (*I2*, *I3*) } } – binary relation

These 5 relations can be represented graphically by 5 tables (Fig. 23.6):

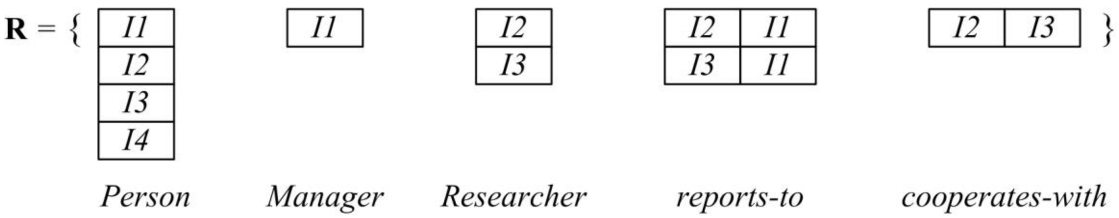


Fig. 23.6. 5 Representing 5 relations can graphically by 5 tables

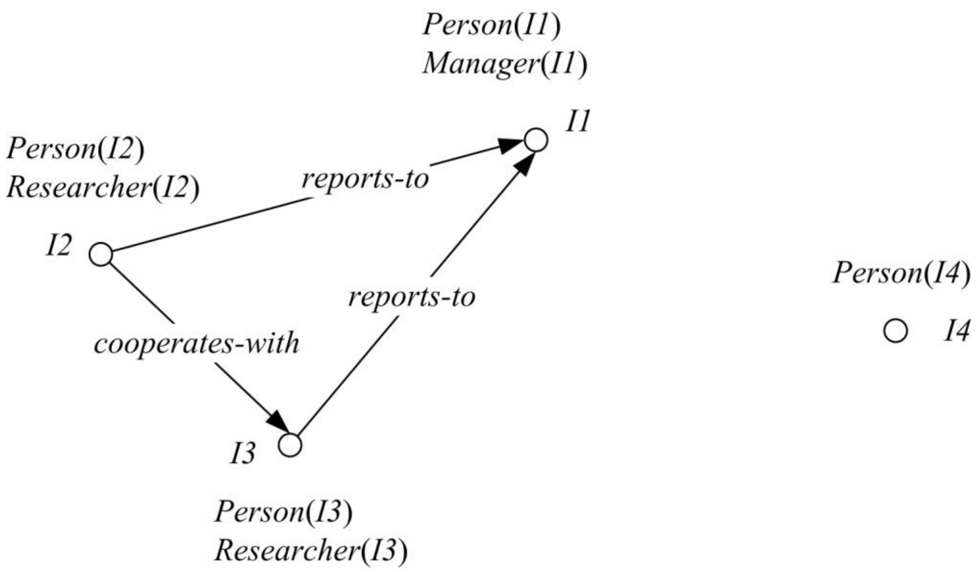


Fig. 23.7. An extensional relational structure from Example 23.2. It corresponds to a specific state of the world. This is a tiny part of a specific world with persons, managers, researchers, and their relationships in the running example of human resources in a large software company. Adapted from [Guarino et al. 2009, Fig. 1]

Example 23.3 [Guarino et al. 2009, example 2.2] Let us consider the following alteration of Example 23.2. An extensional relational structure (*D'*, **R'**) is supplemented with the edge (*I1*, *I4*):

- *D'* = *D*
- **R'** = { *Person*, *Manager*, *Researcher*, *reports-to'*, *cooperates-with* }, where *reports-to'* = *reports-to* ∪ {(*I1*, *I4*)}

Hence (*D'*,**R'**) ≠ (*D*,**R**), although *reports-to'* is supplemented with one line (Fig. 36.7).

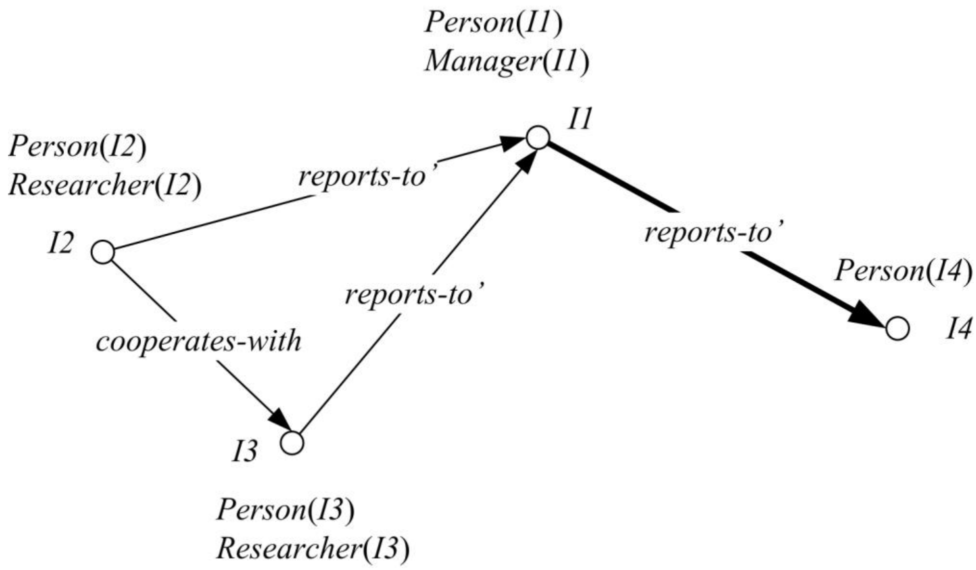


Fig. 23.8. The extensional relational structure (*D'*, **R'**) in Example 23.6. The relation *reports-to'* is supplemented with the edge (*I1*, *I4*). This corresponds to another state of the world

Definition 23.4 (World *W*). A world is a totally ordered set of world states:

$$W = \{w_1, w_2, w_3 \ldots\} \qquad \Box$$

A contrast of total ordering can be illustrated with a partially ordered set. Consider a set $W = \{w_1, w_2, w_3, w_4\}$ with relations $\{w_1<w_2, w_1<w_3, w_2<w_4, w_3<w_4\}$ (Fig. 23.9). There is no relation between w_2 and w_3 – neither $w_2<w_3$ nor $w_2>w_3$ – which are called *non-comparable*.

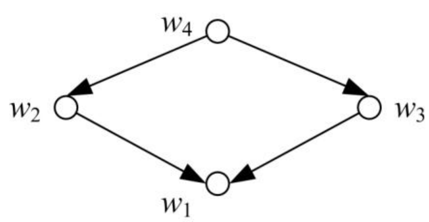


Fig. 23.9. A partially ordered set $W = \{w_1, w_2, w_3, w_4\}$

Definition 23.5 (Intensional relation, or conceptual relation $\rho^{(n)}$). An intensional relation (or conceptual relation) $\rho^{(n)}$ of arity n on (D, W) is a total function $W \rightarrow powerset(D^n)$ from the set W into the set of all n-ary extensional relations on D. In other words, $\rho^{(n)}$: $w_i \rightarrow \rho^{(n)}(w_i)$, i.e. $\rho^{(n)}$ maps a world's state w_i into $\rho^{(n)}(w_i)$, a subset of the Cartesian product D^n (w_i) –

a set of tuples from D^n, in other words, an extensional relational structure. [Guarino et al. 2009, definition 2.3] □

Here *powerset*(A) denotes a set of subsets of A. Also denoted 2^A. For instance, suppose $A = \{a, b\}$. Then *powerset*(A) = $2^A = \{ \{\}, \{a\}, \{b\}, \{a, b\} \}$.

Definition 23.6 (Intensional relational structure C = (D, W, $\mathcal{R}$), **or conceptualization)** An intensional relational structure (or a conceptualization) is a triple **C** = (D, W, $\mathcal{R}$) with

- D a universe of discourse
- W a world, i.e. a set of world's states
- $\mathcal{R}$ a set of intensional relations on the domain space (D, W)
  [Guarino et al. 2009, definition 2.3] □

Example 23.7 [Guarino et al. 2009, example 2.3] Comming back to Examples 23.2 and 23.3, we can see them as describing two different world's states compatible with the following intensional relational structure **C** = (D, W, $\mathcal{R}$) (Fig. 23.10):

- $D = \{I1, I2, I3, I4\}$
- $W = \{w_1, w_2, w_3 \ldots\}$
- $\mathcal{R} = \{Person^{(1)}, Manager^{(1)}, Researcher^{(1)}, reports\text{-}to^{(2)}, cooperates\text{-}with^{(2)}\}$
- $\forall i=1,2,3\ldots \quad Person^{(1)}(w_i) = \{(I1), (I2), (I3), (I4)\}$
- $\forall i=1,2,3\ldots \quad Manager^{(1)}(w_i) = \{(I1)\}$
- $\forall i=1,2,3\ldots \quad Researcher^{(1)}(w_i) = \{(I2), (I3)\}$
- $\forall i=1,2,3\ldots \; reports\text{-}to^{(2)}(w_{2i-1}) = \{(I2, I1), (I3, I1)\}$ – 2 edges in odd states
- $\forall\, i=1,2,3\ldots \; reports\text{-}to^{(2)}(w_{2i}) = \{(I2, I1), (I3, I1), (I1, I4)\}$ – 3 edges in even states
- $\forall\, i=1,2,3\ldots \; cooperates\text{-}with^{(2)}(w_i) = \{(I2, I3)\}$

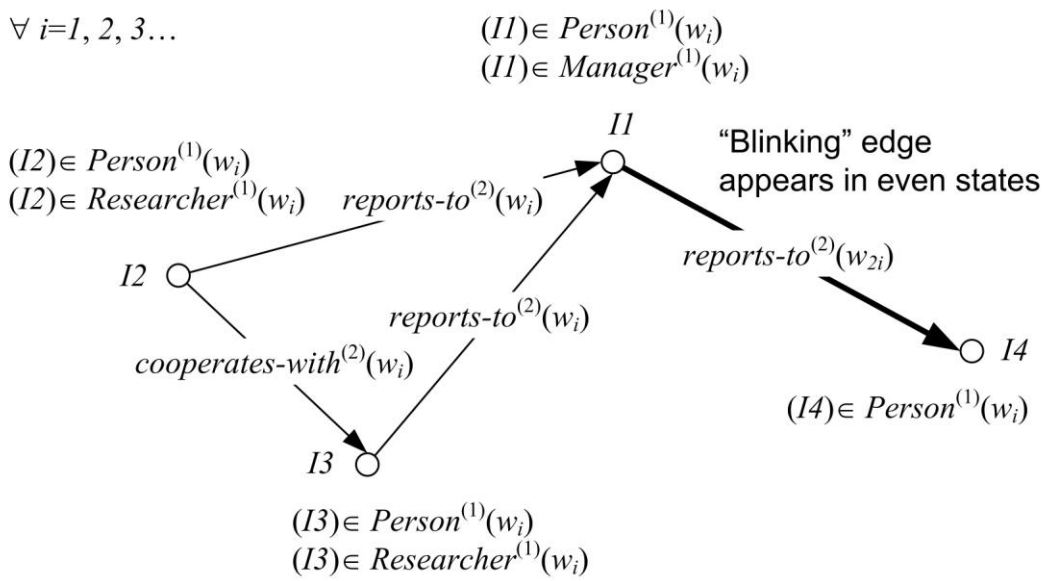


Fig. 23.10. The intensional relational structure in Example 23.7. The intensional relations *Person*$^{(1)}$, *Manager*$^{(1)}$, *Researcher*$^{(1)}$ and *cooperates-with*$^{(2)}$ do not change, but *reports-to*$^{(2)}$ changes – the edge (*I1*, *I4*) blinks

23.3. What is a Proper Formal, Explicit Specification?

In practical applications, as well as in human communication, we need to use a language to refer to the elements of a conceptualization: for instance, to express the fact that *I2* cooperates with *I3*, we have to introduce a specific symbol (formally, a predicate symbol, say '*cooperates-with*', which, in the user's intention, is intended to represent a certain conceptual relation. We say in this case that our language (let us call it **L**) *commits* to a conceptualization. Suppose now that **L** is a first-order logical language, whose non-logical symbols (i.e., its vocabulary) is V. The vocabulary is divided into two sets: 1) *constant symbols*, e.g, '*I1*', '*I2*', '*I3*', '*I4*', and 2) *predicate symbols*, e.g., '*Person*', '*Manager*', '*Researcher*', '*reports-to*', '*cooperates-with*':

$$\mathbf{V} = \textit{constant symbols} \cup \textit{predicate symbols}$$

A first-order language has the following difference from higher-order languages: predicate arguments can be formed of variables and expressions, but not of predicate names. In other words, a table field cannot be a table's name. Examples of correct first-order sentences are '*reports-to*'('*I2*', '*I1*') = '*true*' and $\forall x,y$ '*reports-to*'$(x,y) \Rightarrow$ '*Person*'(x) & '*Person*'(y), but not '*reports-to*'('*Person*', '*Person*').

Definition 23.8 (Extensional first-order structure $M = (S, I)$ **).** Let us assume

- **L** a first-order logical language with vocabulary **V**
- $S = (D, \mathbf{R})$ an extensional relational structure

An **extensional first-order structure** (also called model for **L**) is a tuple $M = (S, I)$, where I (called **extensional interpretation function**) is a total function I: $\mathbf{V} \rightarrow D \cup \mathbf{R}$ that maps each vocabulary symbol of **V** to either an element of D or an extensional relation belonging to the set **R**. In other words, I maps constant symbols to D and predicate symbols to **R**, where $S = (D, \mathbf{R})$. [Guarino et al. 2009, definition 3.1] □

Definition 23.9 (Intensional first-order structure K= (**C**, $\mathcal{I}$), also called **ontological commitment).** Let us assume

- **L** a first-order logical language with vocabulary **V**
- **C** = (D, W, $\mathcal{R}$) an intensional relational structure (conceptualization)

An **intensional first-order structure** (also called ontological commitment) for **L** is a tuple **K**= (**C**, $\mathcal{I}$), where kur $\mathcal{I}$ (called **intensional interpretation function**) is a total function $\mathcal{I}$: $\mathbf{V} \to D \cup \mathcal{R}$, that maps each vocabulary symbol of **V** to either an element of D or an intensional relation belonging to the set $\mathcal{R}$. In other words, $\mathcal{I}$ maps constant symbols to D and predicate symbols to $\mathcal{R}$, where **C** = (D, W, $\mathcal{R}$). [Guarino et al. 2009, definition 3.2] □

Example 23.10 [Guarino et al. 2009, example 3.1] Let us come back to the extensional relational structure in our Example 23.2. An intensional first-order structure is as follows:

V = {'*I1*', '*I2*', '*I3*', '*I4*'} ∪
{'*Person*', '*Manager*', '*Researcher*', '*reports-to*', '*cooperates-with*'}

$\mathcal{I}$: '*I1*' → *I1*, '*I2*' → *I2*, '*I3*' → *I3*, '*I4*' → *I4*,
'*Person*' → $Person^{(1)}$, '*Manager*' → $Manager^{(1)}$, 'Researcher' → $\text{Researcher}^{(1)}$,
'*reports-to*' → $reports\text{-}to^{(2)}$, '*cooperates-with*'→ $\text{cooperates-with}^{(2)}$

Definition 23.11 (Intended models $\mathbf{I_K}$(L)). Let us assume

- **C** = (D, W, $\mathcal{R}$) an intensional relational structure (conceptualization)
- **L** a first-order logical language with vocabulary **V**
- **K**= (**C**, $\mathcal{I}$) an intensional first-order structure (ontological commitment)

A model $M = (S, I)$, with $S = (D, \mathbf{R})$, is called an **intended model** of **L** according to **K** iff

1. For all constant symbols $c \in \mathbf{V}$ we have $I(c) = \mathcal{I}(c)$
2. There exists a world $w \in W$ such that, for each predicate symbol $v \in \mathbf{V}$ there exists an intensional relation $\rho^{(n)} \in \mathcal{R}$ such that $\mathcal{I}(v) = \rho^{(n)}$ and $I(v) = \rho^{(n)}(w)$

The set $\mathbf{I_K}(\mathbf{L})$ of all models of **L** that are compatible with **K** is called the **set of intended models** of **L** according to **K**. (See Guarino et al. 2009, definition 3.3) □

In Example 23.7, for instance, we have for w_1 (Fig. 23.11):

I('*Person*') = { (*I1*), (*I2*), (*I3*), (*I4*) } = $Person^{(1)}(w_1)$
I('*reports-to*') = { (*I2*, *I1*), (*I3*, *I1*) } = $reports\text{-}to^{(2)}(w_1)$
etc.

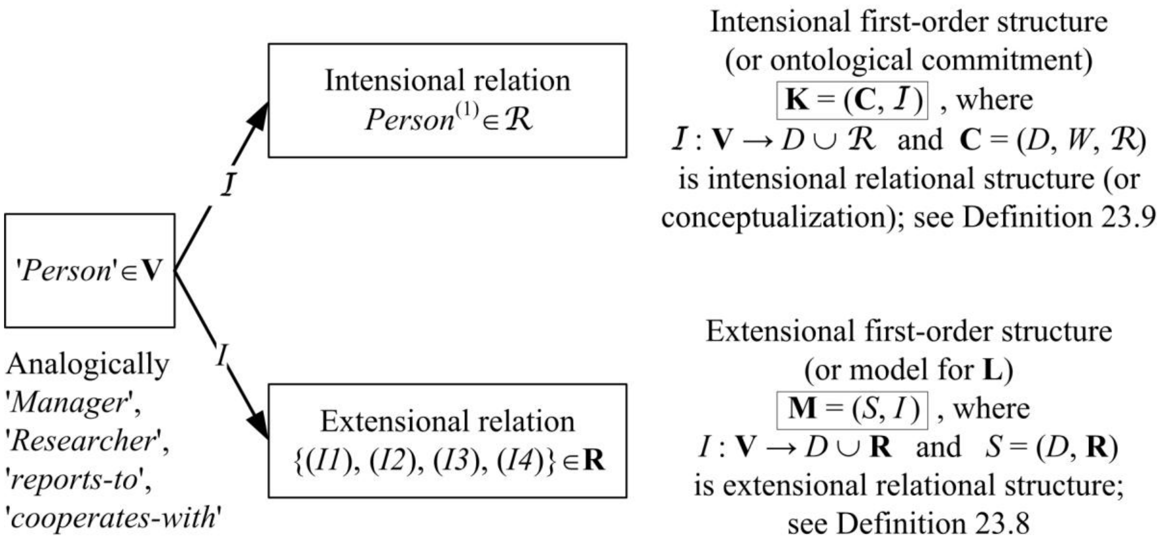


Fig. 23.11. The predicate symbol '*Person*' has both an extensional interpretation (through the usual notion of model, or extensional first-order structure) and an intensional interpretation (through the notion of ontological commitment, or intensional first-order structure). Adapted from [Guarino et al. 2009, Fig. 3]

Definition 23.12 (Ontology $\mathbf{O_K}$). Let us assume

- $\mathbf{C} = (D, W, \mathcal{R})$ an intensional relational structure (conceptualization)
- $\mathbf{L}$ a first-order logical language with vocabulary $\mathbf{V}$
- $\mathbf{K} = (\mathbf{C}, \mathcal{I})$ an intensional first-order structure (ontological commitment)

An ontology $\mathbf{O_K}$ for $\mathbf{C}$ with vocabulary $\mathbf{V}$ and ontological commitment $\mathbf{K}$ is a logical theory consisting of a set of formulas of $\mathbf{L}$, designed so that the set of its models approximates as well as possible the set of intended models $\mathbf{I_K}(\mathbf{L})$ of $\mathbf{L}$ according to $\mathbf{K}$.
[Guarino et al. 2009, definition 3.4] □

Example 23.13. In the following we build an ontology $\mathbf{O_K}$ consisting of a set of logical formulae. Through O_1 to O_5 we specify our human resources domain (see Examples 23.2 and 23.7) with increasing precision. [Guarino et al. 2009, example 3.2]

- Taxonomic information. We start our formalization by specifying that the concepts *Researcher* and *Manager* are sub-concepts of *Person*:

  $$O_1 = \{ \text{'}Researcher\text{'}(x) \Rightarrow \text{'}Person\text{'}(x), \quad \text{'}Manager\text{'}(x) \Rightarrow \text{'}Person\text{'}(x) \}$$

- Domains and ranges. We continue by adding formulae to O_1 which specify the domains and ranges of the binary relations:

  $$O_2 = O_1 \cup \{ \text{'}cooperates\text{-}with\text{'}(x, y) \Rightarrow \text{'}Person\text{'}(x) \ \& \ \text{'}Person\text{'}(y) \quad \text{'}reports\text{-}to\text{'}(x, y) \Rightarrow \text{'}Person\text{'}(x) \ \& \ \text{'}Person\text{'}(y) \}$$

- Symmetry. *cooperates-with* can be considered a symmetric relation:

  $$O_3 = O_2 \cup \{ \text{'}cooperates\text{-}with\text{'}(x,y) \Leftrightarrow \text{'}cooperates\text{-}with\text{'}(y, x) \}$$

- Transitivity. Although arguable, we specify *reports-to* as a transitive relation:

$O_4 = O_3 \cup \{$ '*reports-to*'$(x, z) \Leftarrow$ '*reports-to*'(x, y) & '*reports-to*'(y, z) $\}$

- Disjointness. There is no *Person* who is both a *Researcher* and a *Manager*:

  $O_5 = O_4 \cup \{$ '*Manager*'$(x) \Rightarrow \neg$'*Researcher*'(x) $\}$

The more formulae an ontology contains, the more constraints are concerned. Therefore, $\mathbf{I_K}(\mathbf{L})$ contains less intended models (Fig. 23.12). In other words, if $D' \subset D$ and $\mathcal{R}' \subset \mathcal{R}$ then $\mathbf{M}_{D'}(\mathbf{L}) \supset \mathbf{M}_D(\mathbf{L})$.

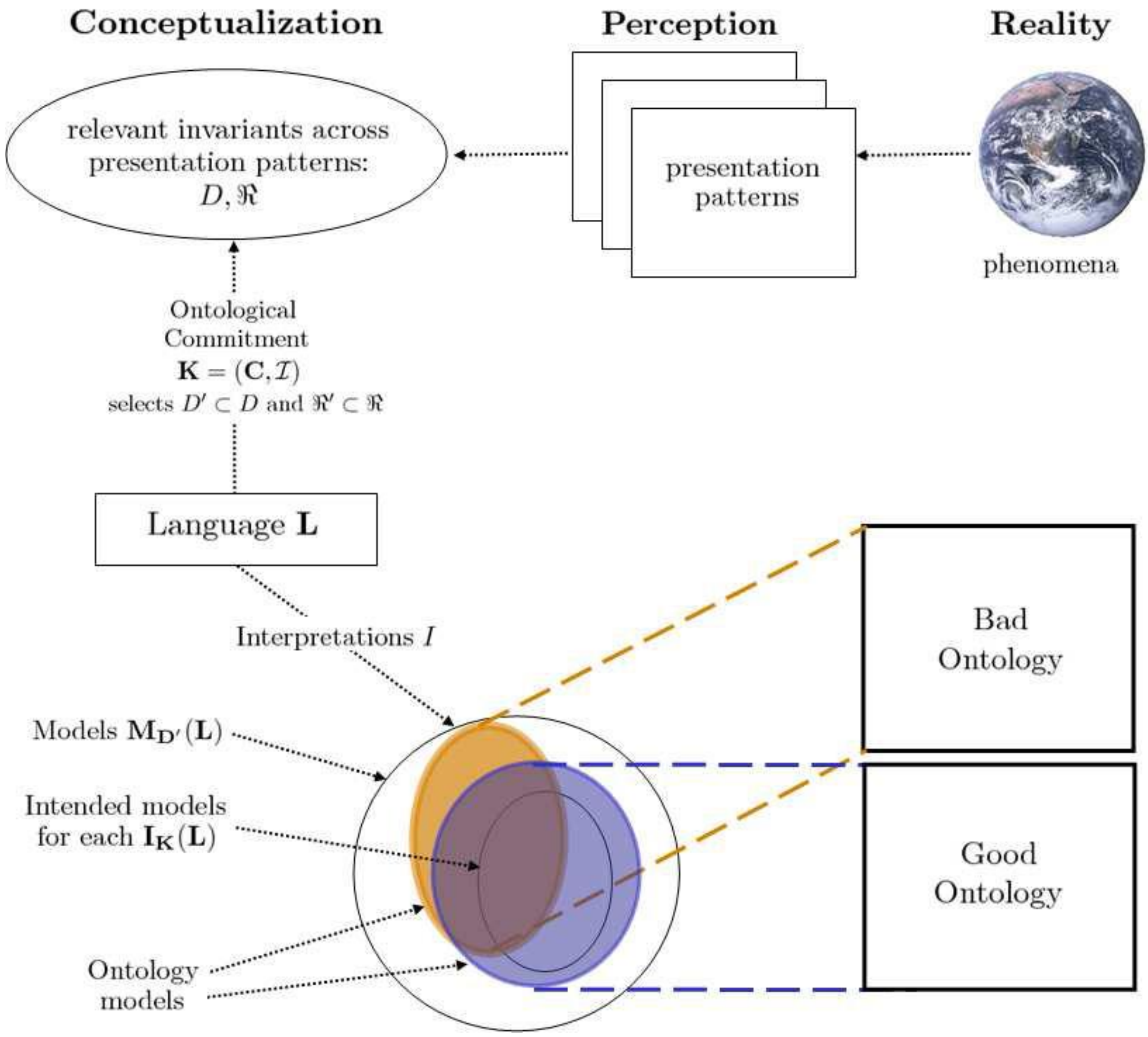


Fig. 23.12. The relationships between the phenomena occurring in reality, their perception (at different times), their abstract conceptualization, the language used to talk about such conceptualization, its intended models, and an [Guarino et al. 2009, Fig. 2]

"At one extreme, we have rather informal approaches for the language **L** that may allow the definitions of terms only, with little or no specification of the meaning of the term. At the other end of the spectrum, we have formal approaches, i.e., logical languages that allow specifying rigorously formalized logical theories… Within this rightmost category one typically encounters the trade-off between expressiveness and efficiency when choosing the language **L**. On the one end, we find higher-order logic, full first-order logic, or modal logic. They are very expressive, but do often not allow for sound and complete reasoning and if they do, reasoning sometimes remains untractable. At the other end, we find less stringent subsets of first-order logic, which typically feature decidable and more efficient reasoners. They can be split in two major paradigms. First, languages from the family of *description logics (DL)*… e.g., OWL-DL", see http://www.w3.org/2001/sw/wiki/OWL for the Web Ontology Language (OWL). "The second major paradigm comes from the tradition of *logic programming (LP)*… with one prominent representor being F-Logic" [Guarino et al. 2009, p. 12–13].

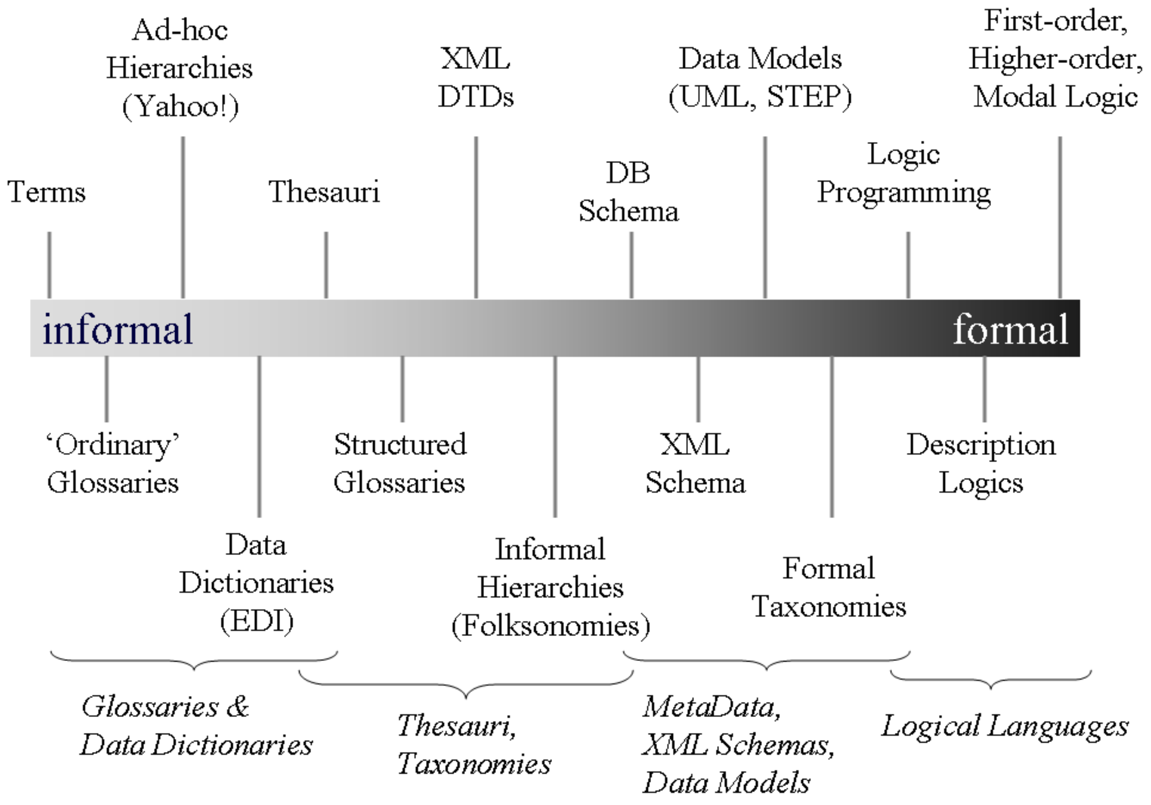
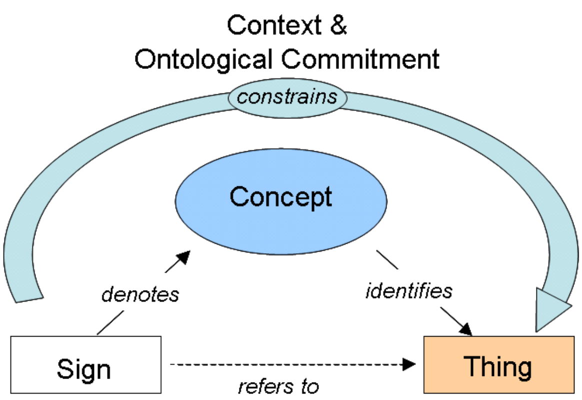


Fig. 23.13. Different approaches to the language **L**. Typically, logical languages are eligible for the formal, explicit specification, and, thus, ontologies [Guarino et al. 2009, Fig. 4]

Reference and meaning is shown in Fig. 23.14.



23.14 pav. Semiotic triangle [Guarino et al. 2009, Fig. 6]

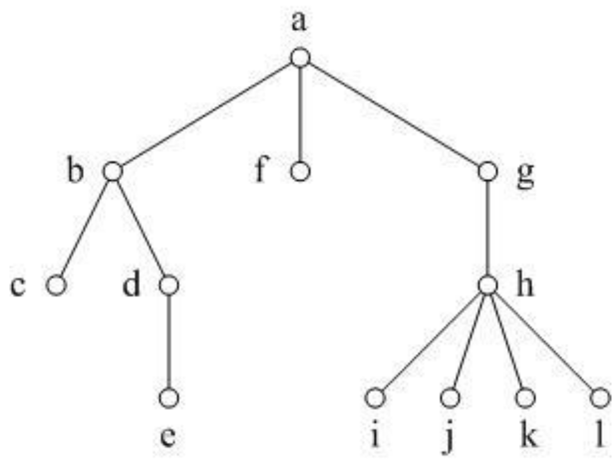
24. Examination Questions

1. Artificial intelligence production system, a formalisation, PRODUCTION algorithm, examples. The concept of artificial intelligence (according to your reading).

2. BACKTRACK and BACKTRACK1 search algorithms. The concept of backtracking. Is an infinite loop possible? Examples. The concept of heuristic.

3. Labyrinth puzzle. Depth-first search and breadth-first search.

4. Prefix and postfix traversal of a tree. Binary trees and general trees. Write procedures for general trees: 1) enter a general tree, 2) prefix order traversal, and 3) postfix order traversal. For example, the string “a(b(cd(e))fg(h(ijkl))).” Represents the graph:



5. Depth-first search and breadth-first search in trees. Shortest path search algorithm for non-weighted graphs (edges have equal costs 1).

6. Shortest path search algorithm for weighted graphs (edges have non-negative costs).

7. Depth-first search for graphs. The concepts of solver and planner.

8. Procedure GRAPHSEARCH. Uniform search, heuristic search. A difference between BACKTRACK1 and GRAPHSEARH-DEPTH-FIRST. A counterexample.

9. A* search algorithm.

10. Forward chaining and backward chaining over rules. Semantic graphs. Program synthesis. The complexity of inference.

11. The resolution inference rule. Proof examples. Forward-chaining and backward-chaining strategies in resolution proof.

12. Expert systems as artificial intelligence systems. An architecture. An example.

13. Internet shopping specification (according to Russell and Norvig).

14. The Turing test and the philosophy of artificial intelligence (according to your reading).

15. The infeasibility of achieving several goals. The punishment problem as an example.

25. References

Main list

1. [Brachman, Levesque 2004] Ronald Brachman, Hector Levesque. *Knowledge representation and reasoning*. The Morgan Kaufmann Series in Artificial Intelligence, 2004. 381 p. VU MIF: 004.8/Br-04.
2. [Luger 2005] George Luger. *Artificial intelligence: structures and strategies for complex problem solving* (fifth ed.). Addison-Wesley, 2005. 928 p. http://www.cs.unm.edu/~luger/. VU MIF: 004.8/Lu-59.
3. [Negnevitsky 2011] Michael Negnevitsky. *Artificial Intelligence: A guide to Intelligent Systems* (3rd ed.). Addison-Wesley, 2011. VU MIF: 004.8/Ne-44.
4. [Russell, Norvig 2010] Stuart Russell, Peter Norvig. *Artificial intelligence: a modern approach* (third edition). Prentice Hall, 2010. http://aima.cs.berkeley.edu. VU MIF: 004.8/Ru-151.

Supplementary list

5. [Antoniou et al. 2012] G. Antoniou, P. Groth, F. van Harmelen, R. Hoekstra. *A semantic web primer* (third ed.). The MIT Press, 270 p. VU MIF 004.8/An-154.
6. [Nilsson 2010] Nils Nilsson. *The quest for artificial intelligence: a history of ideas and achievements*. Cambridge University Press. VU MIF: 004.8/Ni-133.
7. [Nilsson 1998] Nils Nilsson. *Artificial intelligence: a new synthesis*. Morgan Kaufmann Publishers, 1998. 513 p. VU MIF: 004.8/Ni-133.
8. [Sowa 2000] John F. Sowa. *Knowledge representation: logical philosophical, and computational foundations*. Brooks/Cole, a division of Thomson Learning. 2000. 573 p.

Other references

9. [Bench-Capon, Prakken 2006] T. J. M. Bench-Capon, H. Prakken (2006) Justifying actions by accruing arguments. In: *Computational Models of Argument – Proceedings of COMMA 2006*, IOS Press, 2006, pp. 247–258. IOS Press. http://www.cs.uu.nl/groups/IS/archive/henry/action.pdf, slides http://www.cs.uu.nl/deon2006/tbc+hp.pdf.
10. [Guarino et al. 2009] N. Guarino, D. Oberle, S. Staab. What is an Ontology? In: *Handbook on Ontologies*, S. Staab, R. Studer (eds.) pp. 1–17, Springer, 2009. http://iaoa.org/isc2012/docs/Guarino2009_What_is_an_Ontology.pdf.
11. [MacCrimmon, Tillers 2002] The dynamics of judicial proof: computation, logic, and common sense, Marilyn MacCrimmon, Peter Tillers (ed.). Heidelberg; New York: Physica-Verlag, 2002. 494 p. (Studies in fuzziness and soft computing; Vol. 94).
12. [Nilsson 1998] Nils Nilsson. *Artificial Intelligence: A New Synthesis*. Morgan Kaufmann Publishers, 1998. 513 p. VU MIF 004.8/Ni-133.
13. [Sawyer, Foster 1986] B. Sawyer, D. Foster. Programming expert systems in Pascal. John-Wiley, 1986.
14. [Thayse et al. 1990] A. Thayse, P. Gribomont, G. Louis et al. *Approche logique de l'intelligence artificielle*. 1988.
15. [Turing 1950] A. M. Turing. Computing machinery and intelligence. *Mind*, 1950. Vol. 59. Nr. 236, pp. 433–460. http://mind.oxfordjournals.org/content/LIX/236/433, http://www.loebner.net/Prizef/TuringArticle.html.