Artifical Intelligence

Lorant Csige PhD.

April 29, 2014

Acknowledgements

This work was supported by the European Union and the State of Hungary, co-financed by the European Social Fund in the framework of TÁMOP 4.2.4.A/2-11-1-2012-0001 'National Excellence Program'.

Contents

Sets

We begin with a brief survay of the symbolic notations employed in the course.

Definition

A set is a collection of distinct objects, without repetition, and without ordering. The elements of a set are referred to as its members. We sometimes write $a \in A$ to indicate that the element a is a member of the set A, and $a \notin A$ for a is not a member of A. Two sets are equal when they have the same members. An empty set, denoted $\emptyset$ or $\{\}$, is a set with no members.

Sets II.

A method defining particular sets is

- by an enumeration of the elements of the set; or
- by a description of some attribute or characteristic of the elements of the set; in this case, the set is said to be implicitly specified.

Examples

1. Set of marks in our university: $M \rightleftharpoons \{1, 2, 3, 4, 5\}$.
2. Set of positive integers less or equal than 5:

$$S \rightleftharpoons \{x | x \text{ is an integer and } 0 < x \text{ and } x \leq 5\}$$

Set A is said to be contained in the set B if every member of A is a member of B. We also say that A is a subset of B, denoted $A \subseteq B$. If $A \subseteq B$ and there is a member of B that is not in A then $A \subset B$ (A is a proper subset of B).

Sets III.

Definition

The set of all subsets of a set A is called the power set of A, denoted $P(A)$;

$$P(A) \rightleftharpoons \{X | X \subseteq A\}.$$

Note that if A has n members, $P(A)$ has 2^n members.

Examples

If $A = \{a, b, c\}$,

$$P(A) = \{\emptyset, \{a\}, \{b\}, \{c\}, \{a, b\}, \{a, c\}, \{b, c\}, \{a, b, c\}\}.$$

Sets IV.

Definition

Let A, B be nonempty sets, and let $A \times B$ (read A cross B) be defined as

$$A \times B \rightleftharpoons \{(a, b) | a \in A \text{ and } b \in B\};$$

that is, $A \times B$, the Cartesian product of A and B, is the set of all ordered pairs (a, b) such that the first element of the ordered pair, a, is from A and the second element of the pair, b, is from B.

Examples:

$$\{1, 2\} \times \{a, b, c\} = \{(1, a), (1, b), (1, c), (2, a), (2, b), (2, c)\}$$

The product of more factors is defined similarly:

$$A_1 \times A_2 \times \cdots \times A_n \rightleftharpoons \{(a_1, a_2, \ldots, a_n) | a_1 \in A_1 \text{ and } \ldots a_n \in A_n\}$$

Relations

Now a two-argument (or binary) relation, R, with domain A and range B, is any subset of $A \times B$. If $(a, b) \in R$, we often write aRb.
Examples Let the domain and the range $\{1, 2, 3\}$.
$R(<) \rightleftharpoons \{(1, 2), (1, 3), (2, 3)\}$ and $R(=) \rightleftharpoons \{(1, 1), (2, 2), (3, 3)\}$
Let R be a relation where the domain and the range are identical.

- $R \subseteq A \times A$ is reflexive if every element is related to itself.
- $R \subseteq A \times A$ is symmetric if, whenever (a_1, a_2) is in R, (a_2, a_1) is also in R.
- $R \subseteq A \times A$ is transitive if, whenever (a_1, a_2) is in R and (a_2, a_3) is also in R, (a_1, a_3) is in R also.

Examples $\{(a, d), (d, a), (a, a), (d, d)\}$ is a symmetric and transitive (but not reflexive) relation over $\{a, b, c, d\}$.

Relations II.

- "is greater than", "is equal to", or e.g. "divides" in arithmetic
- "is congruent to" in geometry
- **"is adjacent to" in graph theory**
- "is orthogonal to" in linear algebra

A binary relation is the special case $n = 2$ of an n-ary relation $R \subseteq A_1 \times \ldots \times A_n$, that is, a set of n-tuples where the jth component of each n-tuple is taken from the jth domain A_j of the relation.

Graphs

Definition

A directed graph is described as a pair (N, E) where N is a nonempty set of nodes and $E \subseteq N \times N$ is a set of edges. If $(n, m) \in E$, there is a directed edge from n to m.

Examples: $(\{a, b, c, d\}, \{(a, b), (b, c), (b, d), (c, a), (c, d)\})$ is a directed graph.

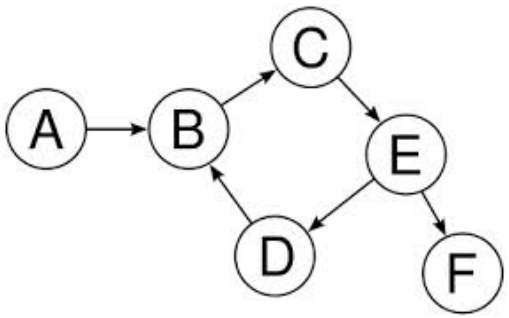

Figure : Visualization of a **directed** graph

Graphs II.

Definition

A finite **directed path** with origin n and endpoint m (path from n to m) is any finite sequence of edges $(n_1, m_1), (n_2, m_2), \ldots, (n_l, m_l)$ such that $n_1 = n$, $n_l = m$ and for all edges the endpoint of an edge is just the startpoint of the next one.

- The length of a path is the number of its edges.
- A cycle in a directed graph is a path (of length greater then 1) from a node to itself.
- A loop in a directed graph is some different paths from the same origin to the same endpoint.

Examples $(a, b), (b, c), (c, d)$ is a directed path of the previous graph. The length of the path is 3. $(a, b), (b, d)$ is an other path with the length 2. They form a loop. $(a, b), (b, c), (c, a)$ is a cycle.

First-order logic

Much of everyday and mathematical language can be symbolized by the first-order logic. In general, we use the following four types of symbols to construct an atom:

- variables: $x, y, z, \ldots$
- constants: These are names of objects such as John, Mary, and 3.
- function symbols: These are names of operations such as *father* and *plus*.
- predicate symbols: These are names of relations such as *Greater* and *Likes*.

Any function or predicate symbols takes a specified number of arguments. If a function symbol f (or a predicate symbol P) takes n arguments, f (or P) is called an n-place function (predicate) symbol.

First-order logic

Terms are defined recursively as follows:

1. A constant is a term.
2. A variable is a term.
3. If f is an n-place function symbol, and $t_1, \ldots, t_n$ are terms, then $f(t_1, \ldots, t_n)$ is a term.
4. All terms are generated by applying the above rules.

Examples
$plus(x, 3)$, $father(John)$, $father(father(John))$ are terms.

If P is an n-place predicate symbol, and $t_1, \ldots, t_n$ are terms, then $P(t_1, \ldots, t_n)$ is an atom.

Examples $Greater(x, 3)$, $Greater(plus(x, 3), 3)$, $Likes(John, Mary)$

First-order logic

Formulas are defined recursively as follows:

1. An atom is a formula.
2. If A and B formulas, then $\neg A, (A \wedge B), (A \vee B), (A \supset B)$ are formulas.
3. If A is formula, and x is a variable, then $\forall x A, \exists x A$ are formulas.
4. Formulas are generated by applying the above rules.

First-order logic

Examples

1. Every rational number is a real numbers.
2. There exists a number that is a prime.
3. For every number x, there exists a number y such that $x < y$.

Denote "x is a prime number" by $P(x)$, "x is a rational number" by $Q(x)$, "x is a real number" by $R(x)$, and "x is less than y" by $Less(x, y)$.

1. $\forall x(Q(x) \supset R(x))$
2. $\exists x P(x)$
3. $\forall x \exists y Less(x, y)$

Aim of AI

Many human mental activities such as writing programs, doing mathematics, engaging in commonsense reasoning, understanding language and even driving a car are said to demand intelligence. Recently, several computer systems have been built that can perform tasks such as these. We might say, that such systems possess some degree of artificial intelligence.
This lecture is about some of the most important, core AI ideas: problem-representations and search strategies.

State space representation

- Let p be a problem and n be the number of relevant properties (flavours) connection with p. (e.g. object, position, size, temperature, colour, etc.) We found m properties.
- A set of values belongs to each property of p. We denote these sets by $\mathcal{H}_1, \mathcal{H}_2, \ldots, \mathcal{H}_m$, where $\mathcal{H}_1$ contains the possible values of the first property, and so on. (e.g. colour: black/white; temperature: $[-20°, 40°]$, etc.)

If the properties have the values $h_1, \ldots, h_m$ we say, that p is in the state $(h_1, \ldots, h_m)$. The state space is the set of all possible states of problem p.

The state space - Cartesian product

Let H_i $(i = 1, \ldots, m)$ be the set of possible values of the property i. Then the states of p are elements of the set of

$$H_1 \times \cdots \times H_m$$

.

We can have conditions, called *constraints*, which determine the possible states.

Therefore, the state space is a subset of the Cartesian product of $H_i (i = 1, \ldots, m)$ defined by the constraints.

$$\mathcal{A} = \{\, a \mid a \in H_1 \times \cdots \times H_m \text{and constraints}(a) \,\}$$

State space representation

- $\mathcal{S}$ is the state space, which is a nonempty subset of the set $\mathcal{H}_1 \times \mathcal{H}_2 \times \cdots \times \mathcal{H}_m$. We use so called constraints to determine the states from the set $\mathcal{H}_1 \times \mathcal{H}_2 \times \cdots \times \mathcal{H}_m$.
  $\mathcal{S} = \{(h_1, \ldots, h_m) | (h_1, \ldots, h_m) \in \mathcal{H}_1 \times \cdots \times \mathcal{H}_n, \mathit{consraint}(h_1, \ldots, h_m)\}$
  The elements of the state space are states.
- *start* is a state in $\mathcal{S}$, called start or initial state ($\mathit{start} \in \mathcal{S}$).
- $\mathcal{G}$ is a subset of $\mathcal{S}$, called set of goal states ($\mathcal{G} \subseteq \mathcal{S}$).
- $\mathcal{O}$ is a nonempty set of unary functions (operators) over $\mathcal{S}$, where $\mathrm{dom}(o) \subseteq \mathcal{S}$ and $\mathrm{rng}(o) \subseteq \mathcal{S}$ for all $o \in \mathcal{O}$. dom

$$(o) = \{s | s \in \mathcal{S}, \mathit{precondition}(s)\}$$

State space representation

Definition

A state-space representation of the problem p can be defined as the following tuple:

$$p = \langle \mathcal{S}, start, \mathcal{G}, \mathcal{O} \rangle$$

State space representation

Let $\langle \mathcal{S}, start, \mathcal{G}, \mathcal{O} \rangle$ be a state space representation and $s \in \mathcal{S}$, $s' \in \mathcal{S}$ be states.

- The state s' is directly accessible from s if there is an $o \in \mathcal{O}$ where $s \in \mathrm{dom}(o)$ and $o(s) = s'$. We denote direct accessibility by $s \underset{o}{\Rightarrow} s'$ or $s \Rightarrow s'$.
- The state s' is accessible from s if there is a list of states $s_0, s_1, \ldots, s_n$ and list of operators $o_1, \ldots, o_n$, where $s_0 = s$, $s_n = s'$ and $s_{k-1} \underset{o_k}{\Rightarrow} s_k$ for all $k \in \{1, 2, \ldots, n\}$. We denote accessibility by $s \overset{*}{\underset{o_1,\ldots,o_n}{\Rightarrow}} s'$ or $s \overset{*}{\Rightarrow} s'$.

Let $p = \langle \mathcal{S}, start, \mathcal{G}, \mathcal{O} \rangle$ be a problem. p is solvable in the state space representation if there is a state $g \in \mathcal{G}$ such that $start \overset{*}{\underset{o_1,\ldots,o_n}{\Rightarrow}} g$.
The sequence of operators $o_1, \ldots, o_n$ is called a solution of p.

The eight-puzzle

The eight-puzzle consists of eight numbered, movable tiles in a 3x3 frame. One cell of the frame is always empty. It is possible to move a neighbouring numbered tile into the empty cell (we could say to move the empty cell). Two configurations of tiles are given. Consider the problem of changing the initial configuration into the goal configuration. A solution to the problem is a sequence of moves.

The relevant properties in this word: the numbered tiles of the frame. We have nine cells of the frame, referred them by their row and column.

cell	$(1,1)$	$(1,2)$	$\ldots$	$(3,3)$
tile	$h_{1,1}$	$h_{1,2}$	$\ldots$	$h_{3,3}$

where $0 \leq l_{k,l} \leq 8$ for all $1 \leq k, l \leq 3$.

The eight-puzzle

Therefore $\mathcal{H}_{k,l} = \mathcal{H} = \{0, 1, \ldots, 8\}$ for all $1 \leq k, l \leq 3$. A state is a 3x3 array

$$\begin{pmatrix} h_{1,1} & h_{1,2} & h_{1,3} \\ h_{2,1} & h_{2,2} & h_{2,3} \\ h_{3,1} & h_{3,2} & h_{3,3} \end{pmatrix}$$

where the values are from $\mathcal{H}$, and every value from $\mathcal{H}$ appear exactly once:

$$\forall k \forall l \forall s \forall o (\neg(k = s) \vee \neg(l = o) \supset \neg(h_{k,l} = h_{s,o})).$$

So the number of states is $9! = 362880$. (This state space is small enough.) The start state and the goal state are:

$$start = \begin{pmatrix} 1 & 2 & 0 \\ 4 & 6 & 3 \\ 7 & 5 & 8 \end{pmatrix} \quad g = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 0 \end{pmatrix}$$

The eight-puzzle

The 8-puzzle has the following four moves: move empty space (blank) to the left, move blank up, move blank to the right, and move blank down. These moves are modeled by operators that operate on the states:

$$up : \begin{pmatrix} h_{1,1} & h_{1,2} & h_{1,3} \\ h_{2,1} & h_{2,2} & h_{2,3} \\ h_{3,1} & h_{3,2} & h_{3,3} \end{pmatrix} \mapsto \begin{pmatrix} h'_{1,1} & h'_{1,2} & h'_{1,3} \\ h'_{2,1} & h'_{2,2} & h'_{2,3} \\ h'_{3,1} & h'_{3,2} & h'_{3,3} \end{pmatrix}$$

The operator up is applicable if

$$\neg\exists k \exists l((h_{k,l} = 0) \wedge (k = 1)).$$

The result of applying if $h_{s,o} = 0$:

$$h'_{k,l} \rightleftharpoons \begin{cases} 0 & \text{if } k = s - 1, l = o \\ h_{s-1,o} & \text{if } k = s, l = o \\ h_{k,l} & \text{otherwise.} \end{cases}$$

Eight Queens Problem

In chess, a queen can move horizontally, vertically, or diagonally. A chess board has 8 rows and 8 columns. The Eight Queens problem asks how to place 8 queens on a chess board so that none of them can hit any other in one move.
(http://spaz.ca/aaron/SCS/queens/)
The relevant properties in this word: whether a cell of the chess board contains a queen or not, and if not, whether a queen on the board hits this cell or not. We have 8×8 cells on the board, referred them by their row and column.

cell	$(1,1)$	$(1,2)$	$\ldots$	$(8,8)$
tile	$h_{1,1}$	$h_{1,2}$	$\ldots$	$h_{8,8}$

where $l_{k,l} \in \{queen, empty, hit\}$ for all $1 \leq k, l \leq 8$.

Eight Queens Problem

Therefore $\mathcal{H}_{k,l} = \mathcal{H} = \{queen, empty, hit\}$ for all $1 \leq k, l \leq 8$. A state is a 8x8 array

$$\begin{pmatrix} h_{1,1} & h_{1,2} & \cdot & h_{1,8} \\ h_{2,1} & h_{2,2} & \cdot & h_{2,8} \\ & & & \\ h_{8,1} & h_{8,2} & \cdot & h_{8,8} \end{pmatrix}$$

where the values are from $\mathcal{H}$, and:

$$\forall k \forall l (h_{k,l} = queen \supset \forall o((o \neq l) \supset (h_{k,o} = hit)) \wedge$$
$$\forall s((s \neq k) \supset (h_{s,l} = hit)) \wedge$$
$$\forall s \forall o((s \neq k) \wedge (o \neq l) \wedge (|s-o| = |k-l|) \wedge (s+o = k+l) \supset (h_{s,o} = hit)).$$

Eight Queens Problem

The start state is:

$$start = \begin{pmatrix} empty & empty & \dots & empty \\ empty & empty & \dots & empty \\ & & & \\ empty & empty & \dots & empty \end{pmatrix}$$

Let be $\delta(queen) = 1, \delta(empty) = 0, \delta(hit) = 0$. Then the goal condition is $g = \{h | \sum \delta(h_{k,l}) = 8\}$.
We can take a queen up the chess board onto the position (k, l), if this cell is empty and after taking, on this cell a queen will be. This queen will hits some other cells.

Eight Queens Problem

The move is modeled by an operator that operates on the states:

$$\textit{queen-up} : \left(\begin{pmatrix} h_{1,1} & h_{1,2} & \dots & h_{1,8} \\ h_{2,1} & h_{2,2} & \dots & h_{2,8} \\ & & & \\ h_{8,1} & h_{8,2} & \dots & h_{8,8} \end{pmatrix} ; (k, l) \right) \mapsto \begin{pmatrix} h'_{1,1} & h'_{1,2} & \dots & h'_{1,8} \\ h'_{2,1} & h'_{2,2} & \dots & h'_{2,8} \\ & & & \\ h'_{8,1} & h'_{8,2} & \dots & h'_{8,8} \end{pmatrix}$$

Eight Queens Problem

The operator $queen - up$ is applicable if $h_{k,l} = empty$. The result of applying:

$$h'_{s,o} \rightleftharpoons \begin{cases} queen & \text{if } s = k, o = l \\ hit & \text{if } (s + o = k + l) \vee (|s - o| = |k - l|) \\ h_{k,l} & \text{otherwise.} \end{cases}$$

(There is exactly one queen in each row.)

The state-space graph

Let $p = \langle \mathcal{S}, start, \mathcal{G}, \mathcal{O} \rangle$ be a problem. This tuple determines a directed graph. (A set + a binary relation determines a graph.)

- The states of state-space $\mathcal{S}$ are the nodes of this graph. If $s \in \mathcal{S}$ then n_s is a node.

$$N = \{n_s \mid s \in \mathcal{S}\}.$$

- n_{start} is the start node.
- The nodes prepared from goal states are goal nodes:

$$T = \{n_g \mid g \in \mathcal{G}\}.$$

  (They are also called termination nodes.)
- The directed edges:

$$E = \{(n_s, n_{s'}) \mid s, s' \in \mathcal{S} \text{ and } s \Rightarrow s'\}.$$

The state-space graph

The directed graph

$$\langle N, n_{start}, T, E \rangle$$

is the state-space graph of the problem p.
Obviously, an explicit specification of the state-space graph is impractical for large graphs and impossible for those having an infinite set of nodes. In the ordinary way, it is given implicitly (with the state-space representation).

Lemma

A graph represented problem p is solvable if and only if there is a directed path in the state-space graph from the start node into one of the goal nodes. The directed paths of this kind are called solutions of p.

Cost

Definition

Often it is conventient to assign costs to edges, to represent the cost of applying the corresponding operator. We use the notation $cost(n_k, n_l)$ to denote the cost of the directed edge (n_k, n_l). We assume that these costs are all greater than some arbitrarily small positive number, δ. The cost of a directed path $n_1, n_2, \ldots, n_l$ is

$$\sum_{k=1}^{l-1} cost(n_k, n_{k+1}).$$

In some problems, we want to find that path having minimal (optimal) cost between start node into any goal node.

The Travelling Salesman Problem

It is a well-known special problem with the following:

- Let A,B,C,D be four stations, salesman should visit all the stations once, A is the start and and the last station.
- Properties: many loops, all the nodes (states) have directed path (operator series) to a termination node (goal state)!
- No circles
- Representation graph has a root and meany leaves (goal nodes)
- Solutions (paths) have the same lengths (3), cost is important. Usually our goal is to find the minimal cost path.

The 4 Queens Problem

Put 4 Queens on the 4x4 board !

- States: 4x4 arrays, 1 operator
- With smart representation the number of edges and states can be reduced.
- Representation graph: root, goal nodes are leaves, no loops and circles.

Remarks

- Choice of cost function: route planning (distance vs. time)
- Travelling Salesman Problem: we have the same number of operators - we need a COST
- State Space Graph - State Space Representation are „equivalent”
- Complicated graphs: loops and circles (making trees), 9! number of nodes in 8 puzzle...
- Reducing the number of nodes and edges is important (e.g. 8 queens problem)
- Roots and leaves (NO in 8 puzzle, YES in Travelling Salesman)

Search systems in general

Search systems are programs that search solutions of problems with using a state space representation. The solution is a path of the state space graph. The first node of a solution is the start state, and the last node is one of the goal states.
The state space graph is not explicitly in the working memory (storage) of the search systems. The systems include only the state-space reprezentation of the problem. During the search, they build (make explicit) part of the implicitly specified state-space graph until neither a solution is discovered or search fails.

Search systems in general

Search systems consist of

- **a database**: This is a working memory containing the state(s) (part of the graph that has been constructed during the search) that have currently been reached, together with some additonal information about searching.
- **a set of production rules**: The production rules operate on the database. Each rule has a precondition that is either satisfied or not by the database. If the precondition is satisfied, the rule can be applied. Application of the rule changes the database.
  - production rules as operators of the state space representation
  - production rules as „technical operations" (moving back)
- **a control system**: The control system chooses which applicable rule should be applied and terminates searching when a termination condition on the database is satisfied.

A typical search system

```
 1: procedure SEARCHING(⟨S, start, G, O⟩)
 2:     database ← INITIALIZE(start)
 3:     while TRUE do
 4:         if THERE-IS-A-SOLUTION(database) then
 5:             break
 6:         end if
 7:         if CAN-NOT-CONTINUE(database) then
 8:             break
 9:         end if
10:         rule ← SELECT(database, rules)
11:         database ← APPLY(database, rule)
12:     end while
13:     if THERE-IS-A-SOLUTION(database) then
14:         SOLUTION-OUTPUT(database)
15:     else
16:         print „Searching without success”
17:     end if
18: end procedure
```

Properties of search systems

Classification of search systems:

1. Can the system modify the effect of an applied production rule later?
   - no: irrevocable control like trial and error method or hill-climbing method;
   - yes: tentative control like searching with backtracking or searching with tree.
2. Does the system use special knowledge (called heuristic information) about a problem to help reduce search?
   - no: blind or noninformed or exhaustive search;
   - yes: informed or heuristic search. (Heuristic: an estimation...)
3. The direction of searching may be
   - forward: from the start state into the goal states;
   - backward: from a goal state into the start state;
   - bidirectional: from both directions simultaneously.

Properties of search systems

The following criteria are used to evaluate search algorithms:

Completeness: Is the algorithm guaranteed to find solution, if one exists?

Optimality: Does it find an optimal solution, if multiple solutions exists?

Time complexity: How much time is required to find solution?

Space complexity: How much space is required to find solution?

Irrevocable search

Let $p = \langle \mathcal{S}, start, \mathcal{G}, \mathcal{O} \rangle$ be a state space representation of the problem p. An irrevocable search system has

- a database with only one state (node), so called actual state (node),
- production rules based on operators and
- the following control:

Algorithm of irrevocable search

```
 1: procedure GENERAL-PROCEDURE(⟨S, start, G, O⟩)
 2:     actual ← start
 3:     while TRUE do
 4:         if actual ∈ G then
 5:             break
 6:         end if
 7:         O′ ← {o | o ∈ O ∧ PRECONDITION(actual, o)}
 8:         if O′ ≠ ∅ then
 9:             operator ← SELECT(O′)
10:             actual ← APPLAY(actual, operator)
11:         else
12:             break
13:         end if
14:     end while
15:     if actual ∈ G then
16:         print actual
17:     else
18:         print „Searching without success”
19:     end if
20: end procedure
```

Algorithm of irrevocable search

In the course of searching, it may be different manner to select the next rule (operator) from the set of applicable rules (operators).

- Selecting at random: try and error method;
- Selecting according with heuristic: hill-climbing method.

Hill Climbing

Let $f : \mathcal{S} \to \{0, 1, 2, \ldots\}$ be the so called heuristic function. $f(s)$ evaluates the lenght of path from node n_s into the goal.

$$f(s) \rightleftharpoons \begin{cases} 0, & \text{if } s \in \mathcal{G} \\ \infty, & \text{if } \neg\exists g(g \in \mathcal{G} \wedge s \stackrel{*}{\Rightarrow} g.). \end{cases}$$

The control strategy uses f to select a rule. It selects the applicable rule that products a database (new actual state) having the largest decrease in the value of f. (Among directly accessible states, the new actual state will be nearest to a goal.) If none of the applicable rules permits an decrease in the value of f, a rule is selected that does not increase the value. If there are no such rules, the process halts.

The algortihm of Hill Climbing method

```
1:  procedure HILL-CLIMBING(⟨S, start, G, O⟩, f)
2:      actual ← start
3:      while TRUE do
4:          if actual ∈ G then
5:              break
6:          end if
7:          O' ← {o | o ∈ O ∧
                        ∧ PRECONDITION(actual, o) ∧ f(APPLY(actual, o)) ≤ f(actual)}
8:          if O' ≠ ∅ then
9:              operator ← SELECT({o | o ∈ O' ∧
                        ∧ ∀o'(o' ∈ O' ⊃ f(APPLY(actual, o)) ≤ f(APPLY(actual, o'))))})
10:             actual ← APPLY(actual, operator)
11:         else
12:             break
13:         end if
14:     end while
15:     if actual ∈ G then
16:         print actual
17:     else
18:         print „Searching without success"
19:     end if
20: end procedure
```

An example

Applying hill-climbing to the eight-puzzle we might use, as a heuristic function, the number of tiles "out of place", as compared to the goal state. $\begin{pmatrix} 1 & 2 & 0 \\ 4 & 6 & 3 \\ 7 & 5 & 8 \end{pmatrix}_4 \rightarrow \begin{pmatrix} 1 & 2 & 3 \\ 4 & 6 & 0 \\ 7 & 5 & 8 \end{pmatrix}_3 \rightarrow \begin{pmatrix} 1 & 2 & 3 \\ 4 & 0 & 6 \\ 7 & 5 & 8 \end{pmatrix}_2$
$\rightarrow \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 0 & 8 \end{pmatrix}_1 \rightarrow \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 0 \end{pmatrix}_0$

Evaluating of irrecovable methods

Completeness: They are not complete methods. At the same time, if the state space graph is finite and it does not contains circles and there is a solution (a path) from each states into a goal in it, then these methods terminate by finding a goal state. We can not get any solution, only a goal state.

Space complexity: The database is very small (only a state).

The basis version

Let be $p = \langle \mathcal{S}, start, \mathcal{G}, \mathcal{O} \rangle$ a state-space reprezentation.

- The database of a backtracking search system contains a directed path of the state-space graph. This path is from the start node to a node called actual node. The memory stores the nodes and the edges of this path and some other information about searching. The data structure is called "search node record". Each search node contains
  - a state (a node of the state-space graph) $s \in \mathcal{S}$;
  - a pointer to the parent search node record (the state s is the result of applying an operator o to the state of the parent search node);
  - an operator o that has produced s;
  - the set of operators, that have already been applied to s (or that have not been applied to s yet).

The basis version

- The production rules of a backtracking search system are
  - based on operators:
    A rule based on the operator o is applicable if it has not been applied on the state in actual search node yet. The effect of the rule is the following. We apply the operator to the state of actual search node. A new state comes into being. We produce a new actual search node from the state, store the applied operator and the pointer of the old actual search node, finally we mark that we have not applied any operators to this state yet. In the old search state, we have to add the operator o to the set of already applied operators also (or we have to delete o from the set of applicable operators).
  - and the so called backtracking:
    The backtracking rule deletes the actual search node and its parent will be actual again. If the database contained only one search node, after backtracking it will be empty.
- The control algorithms are the next pages.

The algorithm of the basis version

1: **procedure** BACKTRACK-1($\langle \mathcal{S}, \text{start}, \mathcal{G}, \mathcal{O} \rangle$)
2: STATE[actual-search-node] $\leftarrow$ start
3: PARENT[actual-search-node] $\leftarrow$ NIL
4: OPERATOR[actual-search-node] $\leftarrow *$
5: APPLIED[actual-search-node] $\leftarrow \emptyset$
6: **while** TRUE **do**
7: **if** actual-search-node $=$ NIL **then**
8: **break**
9: **end if**
10: **if** STATE[actual-search-node] $\in \mathcal{G}$ **then**
11: **break**
12: **end if**
13: $\mathcal{O}' \leftarrow \{o \mid o \in \mathcal{O} \wedge$ PRECONDITION(STATE[actual-search-node], $o) \wedge$
$\wedge\, o \notin$ APPLIED[actual-search-node]$\}$

The algorithm of the basis version

```
14:         if O' ≠ ∅ then
15:             operator ← SELECT(O')
16:             APPLIED[actual-search-node] ←
                            APPLIED[actual-search-node] ∪ {operator}
17:             STATE[new] ← APPLY(STATE[actual-search-node], operator)
18:             PARENT[new] ← actual-search-node
19:             OPERATOR[new] ← operator
20:             APPLIED[new] ← ∅
21:             actual-search-node ← new
22:         else
23:             actual-search-node ← PARENT[actual-search-node]
24:         end if
25:     end while
26:     if actual-search-node ≠ NIL then
27:         SOLUTION-OUT(actual-search-node)
28:     else
29:         print „There is not any solution."
30:     end if
31: end procedure
```

The algorithm of the basis version

```
1:  procedure BACKTRACK-2(⟨S, start, G, O⟩)
2:      STATE[actual-search-node] ← start
3:      PARENT[actual-search-node] ← NIL
4:      OPERATOR[actual-search-node] ← ∗
5:      APPLICABLE[actual-search-node] ←
                              {o | o ∈ O ∧ PRECONDITION(STATE[actual-search-node], o)}
6:      while TRUE do
7:          if actual-search-node = NIL then
8:              break
9:          end if
10:         if STATE[actual-search-node] ∈ G then
11:             break
12:         end if
```

The algorithm of the basis version

```
13:         if APPLICABLE[actual-search-node] ≠ ∅ then
14:             Operator ← SELECT(APPLICABLE[actual-search-node])
15:             APPLICABLE[actual-search-node] ←
                            APPLICABLE[actual-search-node] \ {operator}
16:             STATE[new] ← APPLY(STATE[actual-search-node], operator)
17:             PARENT[new] ← actual-search-node
18:             OPERATOR[new] ← operator
19:             APPLICABLE[new] ←
                            {o | o ∈ O ∧ PRECONDITION(STATE[new], o)}
20:             actual-search-node ← new
21:         else
22:             actual-search-node ← PARENT[actual-search-node]
23:         end if
24:     end while
25:     if actual-search-node ≠ NIL then
26:         SOLUTION-OUT(actual-search-node)
27:     else
28:         print „There is not any solution."
29:     end if
30: end procedure
```

Properties of backtracking

Evaluating of the basis versions of backtacking:

Completeness: If the representation grapf is a finite graph without circles, the backtracking algorithms are complet.

Space Complexity: The database is very small (only a path of the state-space graph).

Circles

If we want to search in a graph with circles

- Backtracking with circle checking:
  If a state-space problem has a solution, it has a solution without circle also. So the control selects the backtrack, when the actual node is already on the actual path.
- Backtracking with boundary:
  We introduce a bound for the actual path. The control selects the backtrack, when the length of actual path access to the bound. During the search, the circles are circulated only finitely times.

Circles

[Completeness:]

- If the state-space graph is a finite graph, then the backtracking with circle checking terminates searching after finitelly many steps. If there are solutions, it discovers one of them, otherwise the database will be empty. (The discovered solution does not contains any circles.)
- The backtracking with boundary terminates searching after finitely many steps. If there are nonlonger solutions than the bound, it discovers one of them, otherwise the database will be empty: there may be only longer solutions in the state-space graph than the bound.

[Time Complexity:] The backtracking with circle checking is time-consuming (especially in the case of long circles).
[Space Complexity:] The database of backtracking with boundary contains no more search node than the bound. The discovered solution does not definitely circle free.

Algorithm with circle checking

```
1:  procedure BACKTRACK-WITH-CIRCLE-CHECKING(⟨S, start, G, O⟩)
2:      STATE[actual-search-node] ← start
3:      PARENT[actual-search-node] ← NIL
4:      OPERATOR[actual-search-node] ← *
5:      APPLICABLE[actual-search-node] ←
                          {o | o ∈ O ∧ PRECONDITION(STATE[actual-search-node], o)}
6:      while TRUE do
7:          if actual-search-node = NIL then
8:              break
9:          end if
10:         if STATE[actual-search-node] ∈ G then
11:             break
12:         end if
13:         if IS-ON-THE-ACTUAL-PATH(STATE[actual-search-node]) then
14:             actual-search-node ← PARENT[actual-search-node]
15:         end if
```

Algorithm with circle checking

```
16:         if APPLICABLE[actual-search-node] ≠ ∅ then
17:             operator ← SELECT(APPLICABLE[actual-search-node])
18:             APPLICABLE[actual-search-node] ←
                        APPLICABLE[actual-search-node] \ {operator}
19:             STATE[new] ← APPLY(STATE[actual-search-node], operator)
20:             PARENT[new] ← actual-search-node
21:             OPERATOR[new] ← operator
22:             APPLICABLE[new] ←
                        {o | o ∈ 𝒪 ∧ PRECONDITION(STATE[new], o)}
23:             actual-search-node ← new
24:         else
25:             actual-search-node ← PARENT[actual-search-node]
26:         end if
27:     end while
28:     if actual-search-node ≠ NIL then
29:         SOLUTION-OUT(actual-search-node)
30:     else
31:         print „There is not any solution."
32:     end if
33: end procedure
```

The algorithm with boundary

```
1:  procedure BACKTRACK-WITH-BOUNDARY(⟨S, start, G, O⟩, bound)
2:      STATE[actual-search-node] ← start
3:      PARENT[actual-search-node] ← NIL
4:      DEPTH[actual-search-node] ← 0
5:      OPERATOR[actual-search-node] ← ∗
6:      APPLICABLE[actual-search-node] ←
                              {o | o ∈ O ∧ PRECONDITION(STATE[actual-search-node], o)}
7:      while TRUE do
8:          if actual-search-node = NIL then
9:              break
10:         end if
11:         if STATE[actual-search-node] ∈ G then
12:             break
13:         end if
14:         if DEPTH[actual-search-node] = bound then
15:             actual-search-node ← PARENT[actual-search-node]
16:         end if
```

The algorithm with boundary

```
17:         if APPLICABLE[actual-search-node] ≠ ∅ then
18:             operator ← SELECT(APPLICABLE[actual-search-node])
19:             APPLICABLE[actual-search-node] ←
                            APPLICABLE[actual-search-node] \ {operator}
20:             STATE[new] ← APPLY(STATE[actual-search-node], operator)
21:             PARENT[new] ← actual-search-node
22:             DEPTH[new] ← DEPTH[actual-search-node] + 1
23:             OPERATOR[new] ← operator
24:             APPLICABLE[new] ←
                            {o | o ∈ 𝒪 ∧ PRECONDITION(STATE[new], o)}
25:             actual-search-node ← new
26:         else
27:             actual-search-node ← PARENT[actual-search-node]
28:         end if
29:     end while
30:     if actual-search-node ≠ NIL then
31:         SOLUTION-OUT(actual-search-node)
32:     else
33:         print „Searching without success."
34:     end if
35: end procedure
```

Branch and bound

The branch and bound algorithm is suited for searching an optimal solution of state-space graph.

- We search a solution with backtracking that is not lengthier than a starting bound.
- If we find such a solution, we store it, the new bound is the length of this solution and the searching is continued.

Completeness: The branch and bound algorithm terminates searching after finitelly many steps. If there are nonlonger solutions than the starting bound, it discovers the optimal solution, otherwise the database will be empty. In the latter case, there may be only longer solutions in the state-space graph than the bound. (Either there is not any solution, or the bound is too small.)

Space Complexity: The database of the branch and bound algorithm contains no more search node than two

The algorithm

```
 1: procedure BRANCH-AND-BOUND(⟨S, start, G, O⟩, bound)
 2:     there-has-already-been-a-solution ← FALSE
 3:     STATE[actual-search-node] ← start
 4:     PARENT[actual-search-node] ← NIL
 5:     DEPTH[actual-search-node] ← 0
 6:     OPERATOR[actual-search-node] ← *
 7:     APPLICABLE[actual-search-node] ←
                              {o | o ∈ O ∧ PRECONDITION(STATE[actual-search-node], o)}
 8:     while TRUE do
 9:         if actual-search-node = NIL then
10:             break
11:         end if
12:         if STATE[actual-search-node] ∈ C then
13:             there-has-already-been-a-solution ← TRUE
14:             SOLUTION-IN(actual-search-node)
15:             bound ← DEPTH[actual-search-node]
16:         end if
17:         if DEPTH[actual-search-node] = bound then
18:             actual-search-node ← PARENT[actual-search-node]
19:         end if
```

The algorithm

```
20:         if APPLICABLE[actual-search-node] ≠ ∅ then
21:             operator ← SELECT(APPLICABLE[actual-search-node])
22:             APPLICABLE[actual-search-node] ←
                                APPLICABLE[actual-search-node] \ {operator}
23:             STATE[new] ← APPLY(STATE[actual-search-node], operator)
24:             PARENT[new] ← actual-search-node
25:             DEPTH[new] ← DEPTH[actual-search-node] + 1
26:             OPERATOR[new] ← operator
27:             APPLICABLE[new] ←
                                {o | o ∈ 𝒪 ∧ PRECONDITION(STATE[new], o)}
28:             actual-search-node ← new
29:         else
30:             actual-search-node ← PARENT[actual-search-node]
31:         end if
32:     end while
33:     if there-has-already-been-a-solution then
34:         SOLUTION-OUT
35:     else
36:         print „Searching without success"
37:     end if
38: end procedure
```

Searching with tree: an introduction

Let $p = \langle \mathcal{S}, \text{start}, \mathcal{G}, \mathcal{O} \rangle$ be a state-space reprezentation. In a search system with tree,

- the database contains an explicit subtree of the state-space graph. The name of this subtree is search tree. The nodes of the search tree have already been produced by the search system. The memory stores the nodes and the edges of this tree and some other information about searching. The data structure is called "search node record". Each search node contains:
  - a state (a node of the state-space graph) $s \in \mathcal{S}$;
  - a pointer to the parent search node record (the state s is the result of applying an operator o to the state of the parent search node);
  - an operator o that has produced s;
  - a *status*:
    - it is **closed**, if we tried to produce the successor nodes of s;
    - otherwise it is **open**.

Searching with tree: an introduction

- the production rule is the expanding. The expanding increases the search tree through an open search node. First we apply all of the applicable operators for the state of this open search node. So we get some states.
  - If an arising state is a new state in the search tree (there is not any search node in it, that contains this state), then we produce a new open search node from it.
  - If an arising state is already in one search node record of the search tree, we work depending of the search strategy.

  Finally we remove the expanded node from open nodes, and put it into the set of the closed nodes.
- The **control** selects a node from the set of open nodes.
  - If the state of the selected open node satisfies the goal conditions, we can contruct a solution along the pointers to the parents.
  - There is not any solution in the state space graph if the set of open nodes will be empty.

Searching with tree: the algorithm

```
 1: procedure SEARCHING-WITH-TREE(⟨S, start, G, O⟩)
 2:     STATE[search-node] ← start
 3:     PARENT[search-node] ← NIL
 4:     OPERATOR[search-node] ← ∗
 5:     open ← {search-node}; closed ← ∅
 6:     while TRUE do
 7:         if open = ∅ then
 8:             break
 9:         end if
10:         search-node ← SELECT(open)
11:         if STATE[search-node] ∈ G then
12:             break
13:         end if
14:         EXPAND(search-node, open, closed)
15:     end while
16:     if open ≠ ∅ then
17:         SOLUTION-OUT(search-node)
18:     else
19:         print „There is not any solution."
20:     end if
21: end procedure
```

Types of searching with tree

In the case of the searching in the same state space graph there are significant differences

1. in the way of selecting. We can select
   - with exhaustive method
     - according to the depth of the searching nodes: breadth-first and depth-first searches;
     - according to the cost: optimal searching;
   - with heuristic method
     - best-first algorithm;
     - **A** algorithm;
2. what is happening if the control discovers a new path to a node of the search tree;
3. when is the test of goal condition.

The Breadth-first and depth-first searches

1. We count the depth of the nodes:

$$\text{depth}(m) \rightleftharpoons \begin{cases} 0 & \text{if } m = \textit{start} \\ \text{depth}(n) + 1 & (n, m) \in E. \end{cases}$$

   - The control of breadth-first search expands the least deapth open node.
   - The control of depth-first search expands the deepest open node.

2. If the control discovers a new path to a node of the search tree, the control leaves behind it.
3. We can test a state in the moment producing.

Properties of Breadt-first and Depth-first

Evaluating of the breadth-first search:

Completeness: If there are solutions in the state-space graph, the breadth-first searching discovers one of them after finitelly many steps, otherwise the set of open nodes will be empty.

Optimality: If there are solutions in the state-space graph, the breadth-first searching discovers the shortest solution.

Space Complexity: The database is great. If the state space graph is a tree, every node has exactly d children, and the length of shortest solution is l, the number of the search nodes of the search tree is:

$$1 + d + d^2 + d^3 + \ldots + d^{l+1} - d \approx O(d^{l+1}).$$

Evaluating of the depth-first search:

Completeness: If there are solutions in a finit state-space graph, the depth-first searching discovers one of them after

The Breadth-first algorithm

```
1:  procedure EXPAND(⟨S, start, G, O⟩, search-node, open, closed)
2:      for all o ∈ O do
3:          if PRECONDITION(STATE[search-node], o) then
4:              state ← APPLY(STATE[search-node],o)
5:              op ← FIND(open, state)
6:              cl ← FIND(closed, state)
7:              if op = NIL and cl = NIL then
8:                  STATE[new-search-node] ← state
9:                  PARENT[new-search-node] ← search-node
10:                 OPERATOR[new-search-node] ← o
11:                 DEPTH[new-search-node] ← DEPTH[search-node] + 1
12:                 open ← open ∪ {new-search-node}
13:             end if
14:         end if
15:     end for
16:     open ← open \ {search-node}  closed ← closed ∪ {search-node}
17: end procedure
```

The Breadth-first algorithm

```
 1: procedure BREADTH-FIRST-SEARCH(⟨S, start, G, O⟩)
 2:     STATE[new-search-node] ← start
 3:     PARENT[new-search-node] ← NIL
 4:     OPERATOR[new-search-node] ← *
 5:     DEPTH[new-search-node] ← 0
 6:     open ← {new-search-node}  closed ← ∅
 7:     while TRUE do
 8:         if open = ∅ then
 9:             break
10:         end if
11:         search-node ← SELECT({no | no ∈ open ∧
                                   ∧ ∀no′(no′ ∈ open ⊃ DEPTH[no] ≤ DEPTH[no′])})
12:         if STATE[search-node] ∈ G then
13:             break
14:         end if
15:         EXPAND(⟨S, start, G, O⟩, search-node, open, closed)
16:     end while
17:     if open ≠ ∅ then
18:         SOLUTION-OUT(search-node)
19:     else
20:         print „There is not any solution."
21:     end if
22: end procedure
```

The Depth-first algorithm

```
1:  procedure EXPAND(⟨S, start, G, O⟩, search-node, open, closed)
2:      for all o ∈ O do
3:          if PRECONDITION(STATE[search-node], o) then
4:              state ← APPLY(STATE[search-node],o)
5:              op ← FIND(open, state)
6:              cl ← FIND(closed, state)
7:              if op = NIL and cl = NIL then
8:                  STATE[new-search-node] ← state
9:                  PARENT[new-search-node] ← search-node
10:                 OPERATOR[new-search-node] ← o
11:                 DEPTH[new-search-node] ← DEPTH[search-node] + 1
12:                 open ← open ∪ {new-search-node}
13:             end if
14:         end if
15:     end for
16:     open ← open \ {search-node}  closed ← closed ∪ {search-node}
17: end procedure
```

The Depth-first algorithm

```
 1: procedure DEPTH-FIRST-SEARCH(⟨S, start, G, O⟩)
 2:     STATE[new-search-node] ← start
 3:     PARENT[new-search-node] ← NIL
 4:     OPERATOR[new-search-node] ← *
 5:     DEPTH[new-search-node] ← 0
 6:     open ← {new-search-node}  closed ← ∅
 7:     while TRUE do
 8:         if open = ∅ then
 9:             break
10:         end if
11:         search-node ← Select({no | no ∈ open ∧
                                   ∧ ∀no′(no′ ∈ open ⊃ DEPTH[no] ≥ DEPTH[no′])})
12:         if STATE[search-node] ∈ G then
13:             break
14:         end if
15:         EXPAND(⟨S, start, G, O⟩, search-node, open, closed)
16:     end while
17:     if open ≠ ∅ then
18:         SOLUTION-OUT(search-node)
19:     else
20:         print „There is not any solution."
21:     end if
22: end procedure
```

The Optimal Search

1. We count the cost of path to the nodes:

$$\text{cost-of-path}(m) \rightleftharpoons \begin{cases} 0 & \text{if } m = start \\ \text{cost-of-path}(n) + cost(n, m) & (n, m) \in E. \end{cases}$$

   The control of optimal search expands the least cost of path open node.
2. If the control discovers a new path to a node of the search tree, the control makes a choice from two pathes. If the new path has less cost, so that

$$\text{cost-of-path}(n) + \text{cost}(n, m) < \text{cost-of-path}(m),$$

   the search node will be updated with the new parent, operator and cost of path. We can not find a new path with less cost to any cosed node.
3. We can not test a state in the moment producing.

Properties of the Optimal search

Evaluating of the optimal searching:

Completeness: If there are solutions in the state-space graph, the optimal searching discovers one of them after finitelly many steps, otherwise the set of open nodes will be empty.

Optimality: If there are solutions in the state-space graph, the optimal searching discovers the optimal solution.

Optimal search algorithm

```
1:  procedure EXPAND(⟨S, start, G, O⟩, cost, search-node, open, closed)
2:      for all o ∈ O do
3:          if PRECONDITION(STATE[search-node], o) then
4:              state ← APPLY(STATE[search-node],o)
5:              op ← FIND(open, state)
6:              z ← FIND(closed, state)
7:              if op = NIL and cl = NIL then
8:                  STATE[new-search-node] ← state
9:                  PARENT[new-search-node] ← search-node
10:                 OPERATOR[new-search-node] ← o
11:                 COST-OF-PATH[new-search-node] ←
                                    COST-OF-PATH[search-node] + cost(o, STATE[search-node])
12:                 open ← open ∪ {new-search-node}
13:             else if op ≠ NIL then
14:                 new-cost-of-path ←
                                    COST-OF-PATH[search-node] + cost(o, STATE[search-node])
15:                 if new-cost-of-path < COST-OF-PATH[op] then
16:                     PARENT[op] ← search-node
17:                     OPERATOR[op] ← o
18:                     COST-OF-PATH[op] ← new-cost-of-path
19:                 end if
20:             end if
21:         end if
22:     end for
23:     open ← open \ {search-node}
24:     closed ← closed ∪ {search-node}
25: end procedure
```

Optimal search algorithm

```
 1: procedure OPTIMAL-SEARCH(⟨S, start, G, O⟩, cost)
 2:     STATE[new-search-node] ← start
 3:     PARENT[new-search-node] ← NIL
 4:     OPERATOR[new-search-node] ← ∗
 5:     COST-OF-PATH[new-search-node] ← 0
 6:     open ← {new-search-node}
 7:     closed ← ∅
 8:     while TRUE do
 9:         if open = ∅ then
10:             break
11:         end if
12:         search-node ← SELECT({no | no ∈ open ∧ ∀no′(no′ ∈ open ⊃ COST-OF-PATH[no] ≤
    COST-OF-PATH[no′])})
13:         if STATE[search-node] ∈ G then
14:             break
15:         end if
16:         EXPAND(⟨S, start, G, O⟩, cost, search-node, open, closed)
17:     end while
18:     if open ≠ ∅ then
19:         SOLUTION-OUT(search-node)
20:     else
21:         print „There is not any solution."
22:     end if
23: end procedure
```

The Best-first algorithm

1. With our knowledge, we estimate the cost of the remainder path to a goal node. A so called heuristic function counts this estimated cost. The control of the best-first search expands the least heuristic open node.
2. If the control discovers a new path to a node of the search tree, the control leaves behind it.
3. We can test a state in the moment producing.

The Best-first algorithm

```
1:  procedure EXPAND(⟨S, start, G, O⟩, h, search-node, open, closed)
2:      for all o ∈ O do
3:          if PRECONDITION(STATE[search-node], o) then
4:              state ← APPLY(STATE[search-node],o)
5:              op ← FIND(open, state) cl ← FIND(closed, state)
6:              if op = NIL and cl = NIL then
7:                  STATE[new-search-node] ← state
8:                  PARENT[new-search-node] ← search-node
9:                  OPERATOR[new-search-node] ← o
10:                 HEURISTIC[new-search-node] ← h(state)
11:                 open ← open ∪ {new-search-node}
12:             end if
13:         end if
14:     end for
15:     open ← open \ {search-node} closed ← closed ∪ {search-node}
16: end procedure
```

The Best-firt algorithm

```
 1: procedure BEST-FIRST(⟨S, start, G, O⟩, h)
 2:     STATE[new-search-node] ← start
 3:     PARENT[new-search-node] ← NIL
 4:     OPERATOR[new-search-node] ← ∗
 5:     HEURISTIC[new-search-node] ← h(start)
 6:     open ← {new-search-node} closed ← ∅
 7:     while TRUE do
 8:         if open = ∅ then
 9:             break
10:         end if
11:         search-node ← SELECT({no | no ∈ open ∧
                                  ∧ ∀no′(no′ ∈ open ⊃ HEURISTIC[no] ≤ HEURISTIC[no′])})
12:         if STATE[search-node] ∈ G then
13:             break
14:         end if
15:         EXPAND(⟨S, start, G, O⟩, h, search-node, open, closed)
16:     end while
17:     if open ≠ ∅ then
18:         SOLUTION-OUT(search-node)
19:     else
20:         print „There is not any solution."
21:     end if
22: end procedure
```

Properties of the Best-first algorithm

Evaluating of the best-first searching:

Completeness: If there are solutions in a finite state-space graph, the best-first searching discovers one of them after finitelly many steps, otherwise the set of open nodes will be empty.

Space Complexity: The database is not too great, if we have a good heuristic. If the state space graph is a tree, every node has exactly d children, and the length of shortest solution is l, by perfect heuristic, the number of the search nodes of the search tree is:

$$1 + d + d + \ldots + d = O(l \cdot d).$$

The A algorithm

1. Let us define the evaluation function totalcost-of-path(n) so that its value at any node n estimates the sum of the cost of a minimal cost path from the start node s to the node n (cost-of-path(n)) plus the cost of a minimal cost path from node n to a goal node (heuristics(n)):

$$\text{totalcost-of-path}(n) = \text{cost-of-path}(n) + \text{heuristics}(n).$$

   That is totalcost-of-path(n) is an estimate of the cost of a minimal cost path constrained to go through node n. The control of algorithm A expands the least totalcost of path open node.
2. If the control discovers a new path to a node of the search tree, the control makes a choice from two pathes. If the new path has less cost, so that

$$\text{cost-of-path}(n) + \text{cost}(n, m) < \text{cost-of-path}(m),$$

the search node will be updated with the new parent operator

Properties of the A algorithm

Evaluating of the algorithm A:

Completeness: If there are solutions in the state-space graph, the algorithm A discovers one of them after finitelly many steps, otherwise the set of open nodes will be empty.

Optimality: When algorithm A uses a function h that $h(a) \leq h^*(a)$ for all $a \in \mathcal{A}$, where $h^*(a)$ is the cost of the minimal cost path from a to a goal state, we call it the algorithm A^*(read A-star). If there are solutions in the state-space graph, the algorithm A^* discovers the optimal solution.

The A* algorithm

```
 1: procedure EXPAND(⟨S, start, G, O⟩, cost, h, search-node, open, closed)
 2:     for all o ∈ O do
 3:         if PRECONDITION(STATE[search-node], o) then
 4:             state ← APPLY(STATE[search-node],o)
 5:             op ← FIND(open, state)
 6:             cl ← FIND(closed, state)
 7:             if op = NIL and cl = NIL then
 8:                 STATE[new-search-node] ← state
 9:                 PARENT[new-search-node] ← search-node
10:                 OPERATOR[new-search-node] ← o
11:                 COST-OF-PATH [new-search-node] ←
                                    COST-OF-PATH[search-node] + cost(o, STATE[search-node])
12:                 HEURISTICS[new-search-node] ← h(state)
13:                 open ← open ∪ {new-search-node}
14:             else
15:                 cost-of-new-path ←
                                    COST-OF-PATH [search-node] + cost(o, STATE[search-node])
```

The A* algorithm

```
16:                 if op ≠ NIL then
17:                     if cost-of-new-path < COST-OF-PATH [op] then
18:                         PARENT[op] ← search-node
19:                         OPERATOR[op] ← o
20:                         COST-OF-PATH [op] ← cost-of-new-path
21:                     end if
22:                 else
23:                     if cost-of-new-path < COST-OF-PATH [cl] then
24:                         PARENT[cl] ← search-node
25:                         OPERATOR[cl] ← o
26:                         COST-OF-PATH [cl] ← cost-of-new-path
27:                         closed ← closed \ {cl}
28:                         open ← open ∪ {cl}
29:                     end if
30:                 end if
31:             end if
32:         end if
33:     end for
34:     open ← open \ {search-node}
35:     closed ← closed ∪ {search-node}
36: end procedure
```

The A* algorithm

```
 1: procedure A-ALGORITHM(⟨S, start, G, O⟩, cost, h)
 2:     STATE[new-search-node] ← start
 3:     PARENT[new-search-node] ← NIL
 4:     OPERATOR[new-search-node] ← ∗
 5:     COST-OF-PATH [new-search-node] ← 0
 6:     HEURISTICS[new-search-node] ← h(start)
 7:     open ← {new-search-node}
 8:     closed ← ∅
 9:     while TRUE do
10:         if open = ∅ then
11:             break
12:         end if
13:         search-node ← SELECT({no | no ∈ open ∧ ∀no′(no′ ∈ open ⊃
                                                    ⊃ (COST-OF-PATH [no] + HEURISTICS[no]) ≤
                                                    ≤ (COST-OF-PATH [no′] + HEURISTICS[no′]))})
14:         if STATE[search-node] ∈ G then
15:             break
16:         end if
17:         EXPAND(⟨S, start, G, O⟩, cost, h, search-node, open, closed)
18:     end while
19:     if open ≠ ∅ then
20:         SOLUTION-OUT(search-node)
21:     else
22:         print „There is not any solution."
23:     end if
24: end procedure
```

The A* algorithm

Definition

Let P and Q be two A^* algorithm! We say, that P is better informed, than Q, if for all nodes n, except the terminal nodes,

$$\text{heuristics}_P(n) > \text{heuristics}_Q(n)$$

, where heuristics_P and heuristics_Q the heuristic functions of the P and Q algorithms. (In other words: the P algorithm is estimating the cost of the remaining path from the bottom in every node more precesily!)

Theorem

If P is better informed A^* algorithm than Q, then all the nodes expanded by P is expanded by Q as well.

The monotone A algorithm

Definition

We say that a h heuristic function satisfies the condition of monotone constraints (so it is a monotone heuristic), if for all $(n, m) \in E$ edge

$$h(n) - h(m) \leq \text{koltseg}(n, m).$$

Theorem

If a heuristic function satisfies the condition of monotone constraints, then

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

for all $n \in N$.

The monotone A algorithm

Definition

We call an A algorithm monotone, if the heuristic function is monotone.

Theorem

If a monotone A algorithm is selecting an open node n for expanding, then an optimal path has already been found to n, so $\text{cost}(n) = \text{cost}^*(n)$.

Strategy games

To describe a game we have to give

- the possible positions of the game,
- the number of players,
- the way of movements (etc. in parallel or after each),
- how much information a player has during the game,
- chance, fortune plays a role (and where) or not
- what is the start and end position of the game,
- how much the players win or lose and when.

Classification of games

classification:

- number of players: pl. *two-players game*;
- the game is made of steps or continuous: *discrete games*;
- games are enden after finately many steps or not: *finate games*;
- information the playsers have: *fully-informed games*;
- what about the fortune: *deterministic games*;
- sum of wins and losses of the players: *0sum games*.

Game theory: an introduction

The games that we consider are two-person, perfect-information games. These are played by two players who move in turn. They each know completely what both players have done and can do. We are intererested in those games where either one of the two players wins and the other loses or where the result is a draw. Example games:

- tic-tac-toe (http://boulter.com/ttt/index.cgi),
- chess (http://en.wikipedia.org/wiki/Chess),
- go (http://en.wikipedia.org/wiki/Go_(game)),
- checkers (http://www.darkfish.com/checkers/Checkers.html) and
- nim (http://www.archimedes-lab.org/game_nim/nim.html).

We are not going to consider here any games whose results are determined even partially by chance.

State space representation of the game

Suppose we call our two players A and B. Let P be a set of the positions of the game. Let us suppose that it is player A turn to play first at the position $p_0 \in P$. We know the legal moves of the game: $\{m \mid m : P \to P\}$. The game is over at some end positions and suppose now that one player wins the other loses. A state-space representation of the game can be defined as the following tuple:

$$\langle \mathcal{S}, start, \mathcal{G}, \mathcal{O} \rangle,$$

where

- $\mathcal{S} = \{(p, \mathtt{I}) \mid p \in P, \mathtt{I} \in \{\mathtt{A}, \mathtt{B}\}, \text{ it is } \mathtt{I} \text{ turn } \}$,
- $start = (p_0, \mathtt{A})$
- $\mathcal{G} = \{(p, \mathtt{I}) \mid p \text{ is an end position, } \mathtt{I} \text{ loses (wins) } \}$
- $\mathcal{O} = \{o_m \mid o_m(p, \mathtt{I}) = (m(p), \mathtt{J}), \mathtt{I}, \mathtt{J} \in \{\mathtt{A}, \mathtt{B}\}, \mathtt{I} \neq \mathtt{J}\}$

The state-space graph of a game is called game graph or tree. In a game tree at the positions of the even levels it is A turn, and at the positions of the odd levels it is B turn.

The MINIMAX algorithm

Our goal in searching a game tree might be to find a good next move at a position. Input:

- $\langle \mathcal{S}, \mathit{start}, \mathcal{G}, \mathcal{O} \rangle$,
- a state $(p, \mathtt{I})$,
- a heuristics $h : \mathcal{S} \to R$ (it estimates the worth of the states),
- and a depth.

The MINIMAX algorithm

The main steps of the algorithm:

1. We build a subtree of the state-space tree from the state $(p, \mathtt{I})$ until level depth.
2. We estimate the leaves of the subtree with the help of heuristics: $\mathrm{worth(q)} = h(q)$.
3. We continue to evaluation, level by level, until the the successors of the start state are assigned worth values: if the children of the state q are $q_1, \ldots, q_k$, then

$$\mathrm{worth}(q) \rightleftharpoons \begin{cases} \max\{\mathrm{worth}(q_1), \ldots, \mathrm{worth}(q_k)\}, & \text{if the level of } q \text{ is} \\ \min\{\mathrm{worth}(q_1), \ldots, \mathrm{worth}(q_k)\}, & \text{if the level of } q \text{ is} \end{cases}$$

Recommendation: at the position p, the player $\mathtt{I}$ should to choose the move to the best child state.

The MINIMAX algorithm

```
1:  function MINIMAX-STEP(⟨S, start, G, O⟩, state, depth, h)
2:      max ← −∞
3:      operator ← NIL
4:      for all o ∈ O do
5:          if PRECONDITION(state, o) then
6:              new-state ← APPLY(state, o)
7:              v ← MINIMAX-VALUE(⟨S, start, G, O⟩, new-state, depth − 1, h)
8:              if v > max then
9:                  max ← v
10:                 operator ← o
11:             end if
12:         end if
13:     end for
14:     return operator
15: end function
```

The MINIMAX algorithm

```
1:  function MINIMAX-VALUE(⟨S, start, G, O⟩, state, depth, h)
2:      if state ∈ G or depth = 0 then
3:          return h(state)
4:      else if PLAYER[state] = J then
5:          max ← −∞
6:          for all o ∈ O do
7:              if PRECONDITION(state, o) then
8:                  new-state ← APPLY(state, o)
9:                  v ← MINIMAX-VALUE(⟨S, start, G, O⟩,
                                                  new-state, depth −
    1, h)
10:                 if v > max then
11:                     max ← v
12:                 end if
13:             end if
14:         end for
15:         return max
```

The MINIMAX algorithm

```
16:     else
17:         min ← ∞
18:         for all o ∈ 𝒪 do
19:             if PRECONDITION(state, o) then
20:                 new-state ← APPLY(state, o)
21:                 v ← MINIMAX-VALUE(⟨𝒮, start, 𝒢, 𝒪⟩,
                                                    new-state, depth −
    1, h)
22:                 if v < min then
23:                     min ← v
24:                 end if
25:             end if
26:         end for
27:         return min
28:     end if
29: end function
```