11. Artificial Intelligence

Prof. Dr.-Ing. habil. Wolfgang Oertel

Goals:

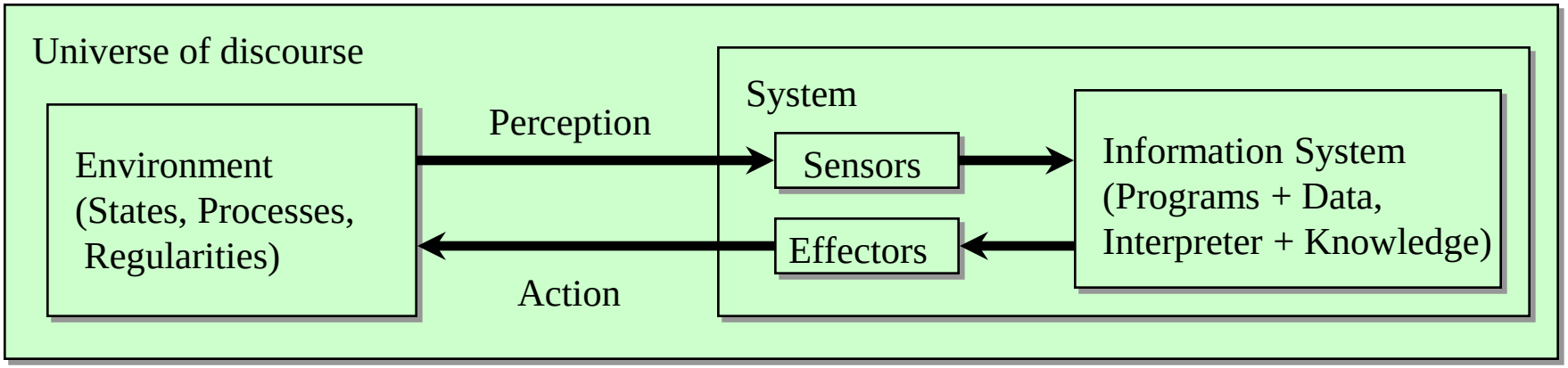
- Concepts of intelligent systems
- Overview of basic concepts

Contents:

• Intelligent System
• Knowledge Representation
• Problem Solving

Focus: Intelligent artificial systems

Intelligent Systems



Universe of discourse: Whole area relevant for the system behaviour
Precondition: States and processes exist in space and time according to regularities
Task: Control of actions depending on perceptions
Goal: correct behaviour, multi-dimensional performance criteria, survival

---

Intelligence levels:

- map-controlled (look-up table)
- reflex-controlled (fix program)
- state controlled (environment mappings)
- behaviour-controlled (adaptation by learning)
- knowledge-controlled (explicit regularities)
- goal-controlled (planning)
- use-controlled (optimisation)

Knowledge Representation

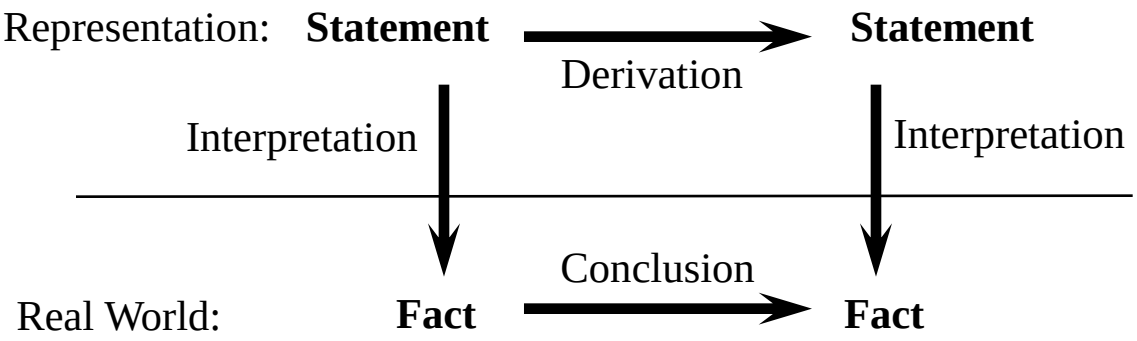
Knowledge Representation Agent: Agent getting knowledge about the real world that
- consists of the description of possible states, operations and dependencies,
- is used for derivation of state features and operation sequences
- can be changed and learned

Knowledge base: Set of statements as representations of facts of the real world

Inference method: Mechanism to derive new statements from existing statements

Knowledge representation language: Language for the specification of definitions, changes, and queries on the knowledge base
- Syntax (Set of statements)
- Semantics (Set of facts)

Representation of knowledge about the real world:



Example:

y = x + 1	y > x
4 = 3 + 1	4 > 3
5 = 4 + 1	5 > 4

Conclusion and derivation

Interpretation: Mapping from statements to facts

Truth: Each statement gets by interpretation a truth value: true / false (1 / 0)

- **general / not to fulfil**: statement that is in each interpretation true / false
- **to fulfil / to falsify**: sentence that is in at least one interpretation true / false

Model: Interpretation in which a statement is true

Conclusion: semantic relation between interpretations of statements F and G:
all models of F are models of G: $F \models G$

Derivation: syntactic relation between statements F and G:
G can be derived from F by application of en inference method: $F \vdash G$

Inference method: defines elementary rules for derivation and the control of their application:

- **correct**: if $F \vdash G$ then $F \models G$
- **complete**: if $F \models G$ then $F \vdash G$
- **monotone**: if $F1 \models G$ then $(F1 \text{ and } F2) \models G$
- **local**: if $F \models G$, then F is a real part of the knowledgebase

Proof: Recording of the application of derivation rules in en inference process

Logics: formal system that defines syntax and semantics of a language by a calculus (with axioms and rules) derivations between statements of the language

⟶ **Theory**

Examples: sentence, predicate, temporal, epistemological, fuzzy, ontological modal logics

Predicate Logic:

Formula set and derivation operations:

Formula = AtomicFormula | ComplexFormula ;
AtomicFormula = **Predicate** , '(' , Terms , ')' ;
ComplexFormula = '~' , Formula | Formula, Junctor, Formula | Quantor, Variable, Formula |
'(' , Formula , ')' ;
Terms = Term | Term , ',' , Terms ; **Term** = **Konstant** | **Variable** | **Function** , '(' , Terms , ')' ;
Junctor = '$\wedge$' | '$\vee$' | '$\rightarrow$' | '$\leftrightarrow$' ; **Quantor** = '$\forall$' | '$\exists$' ;

Separation: $p , p \rightarrow q \vdash q$
Cut: $p \vee q , \sim q \vee r \vdash p \vee r$
All quantor removal: $\forall x p \vdash p[x / t]$
Exist quantor introduction: $p \vdash \exists x p[g / x]$

Semantics: - Term: object of a domain
- Constant: directly specified object
- Variable: any unspecified object
- Function: unique mapping between objects
- Predicate: relation between objects

Examples:

$Bird(Eagle)$

$\forall x\, Bird(x) \rightarrow Fly(x)$

$\vdash Fly(Eagle)$

$Mark(Anton, Physics, 1)$ $Mark(Karl, Chemistry, 2)$
$Mark(Anton, Music, 3)$ $Mark(Karl, Music, 4)$

$\forall x(\exists f(Mark(x,f,1) \vee Mark(x,f,2)) \wedge$
$\sim \exists g Mark(x,g,n) \wedge > (n,3)) \rightarrow Specialist(x) \vdash Specialist(Anton)$

Prolog - Clause Logic:

Features:
- Program consists of a set of **horn clauses** of PK1
- Query is a formula **conjunctive normal form**
- Special **not operator** considers failing as negation
- Set of **built-in Predicates** are proven by execution
- Use of a special syntax
- Inference by **backward chaining** with **depth-first search**
- Derivation rule is **generalised modus ponens** inclusive **unification**
- Search order in clause and clause set **sequential from left to right**

Systems: IF-PROLOG, QUINTUS-PROLOG

Examples:

```
member (X , [X | L]) .
member (X , [Y | L]) :- member (X , L) .
```

```
?- member ( 3 , [1 , 2 , 3 , 4]) .
?- member ( Z , [1 , 2 , 3 , 4]) .
```

```
faculty (0 , 1) :- !.
faculty (N , Z) :- U is N - 1 ,
                   faculty (U , V) , Z is V * N .
```

```
?- faculty (7 , X) .
?- faculty (X , 24) .
```

```
ancestor (X , Y) :- parents ( X , Y).
ancestor (X , Y) :- parents (X , Z) , ancestor (Z , Y) .
parents (X , Y) :- father (X , Y) .
parents (X , Y) : - mother (X , Y) .
mother (X , Y) :- child (Y , X) .
father (X , Y) :- mother (X , Z), married (Z , Y) .
married (anna , anton).
married (anna , martha).
```

```
?- ancestor (martha , X) .
?- ancestor (X , anton) .
?- ancestor (X ,Y) .
?- father (X ,Y) .
```

Production Rules:

Features:
- A **fact base** contains a set of atomic formulas p1, p2, … without variables.
- A **rule base** contains a set of implications of the form (p1, p2, ...) → (a1, a2, ...) with conditions on the left side and actions on the right side.
- The **conditions** are conjunctive and are related to the facts in the fact base.
- The **actions** are carried out sequentially by adding or deleting of facts in the fact base.
- By **matching** rules are determined with conditions are fulfilled in the fact base.
- By conflict **resolution** one rule is selected from a set of rules.
- An **execution** is carried out by application of the actions of a selected rule.
- Inference means a cycle with the three steps matching, resolution and execution until either the an end condition is reached or no more rules are applicable.

Systems: PLANNER, OPS5, LOOPS, KEE

Example:

{Vehicle(1,Empty), Vehicle(2,Empty), Vehicle(3,Empty), Station(1,Free), Station(2,Free)}
{(Vehicle(x,Empty), Station(y,Free)) → (~Station(y,Free), Station(y,Occupied,x)),
(Station(y, Occupied,x)) → (~Station(y, Occupied,x), ~ Vehicle(x,Empty),
Vehicle(x,Full), Station(y,Free))}

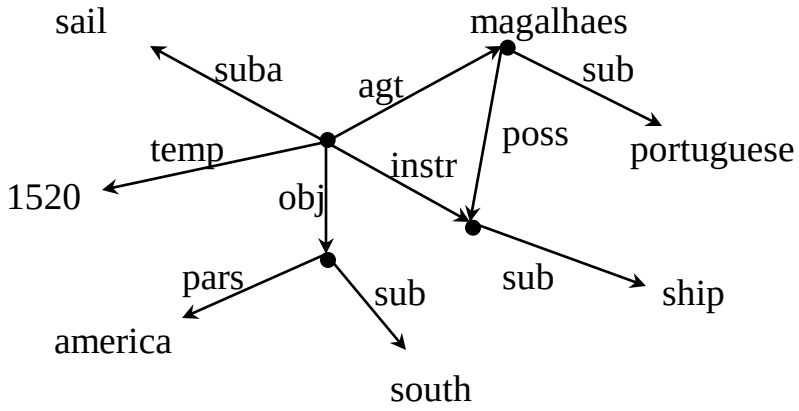
Deductive Inference

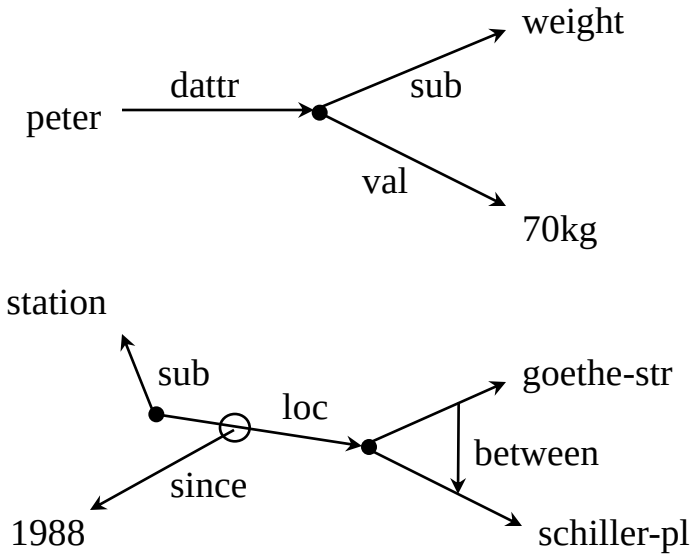
Semantic Network:

Features:
- **Network** with nodes and arcs.
- **Nodes** represent objects, **arcs** represent relation between them.
- Nodes and arcs belong to **predefined semantic categories**.
- The semantics of categories is defined by the inheritance of features
- Typical object categories are classes, instances, sets, actions, actors, times, locations, …
- Typical relation categories are classification, generalisation, aggregation, values, …
- Inference means **navigation** through the network using arcs and inheritance.
- The network si defined as **graph or hypergraph**.

Systems: KL-ONE

Examples:





Frames:

Features:
- **Scheme** describing an entity by a set of **attribute-value pairs**.
- A scheme is a stereotypic **description pattern** occurring in several situations
- Schemes are organised in a **hierarchy**.
- In the hierarchy **inheritance** rules are defined.
- Attributes are **slots** that can be filled by fillers (values).

Systems: KRL, FRL, FLAVORS, SMALLTALK

Examples:

```
[trip       unit basic <self>
            <agent (a traveller)>
            <goalpoint (or (a city) (an recoveryplce))>
            <kind (xor flight ship train)>
            <begin (a date)>  <duration (a timeinterval)>
            <costs (a price)>  <tripnumber (a string)>]
[person     unit basic <self>
            <name (a string)>  <firstname (a string)>
            <age (an integer)>  <residence (an city)>]
[traveller  unit spec <self (a person)>
            <agecategory {(xor child adult)
              (using (the age from person thisone)
              selectfrom (which < 16) child
              otherwise adult)}>
            <tripnumber (a string)>
            <startpoint {(a trainstation)
              (in (the trainstation from city (the residence from person thisone))); default}>]
```

```
[g1  unit indiv
   <self {(a person with name = ”meyer”  age = 40
                firstname = ”otto”  residence = dresden)
           (a traveller with tripnumber = ”r22”
                startpoint = leipzig)
           (a customer with account = ”12343242”)}>]
[g2  unit indiv
   <self (a trip with agent = g1  goalpoint = rostock
                kind = train  price = 500 euro
                begin = (1 8 1988)  duration = 14
                tripnumber = ”r22”)>]
[berlin unit indiv
   <self (a city with trainstation=
                (items lichtenberg, ostbahnhof,
                  schoenefeld))>]
```

Problem Solving

Problem-solving agent: Agent which determines a sequence of actions in its information system before carrying them out the real environment

Working step:
- **Formulation** of the problem
- **Search** of a solution (in the information system)
- **Execution** of the solution (in the real world)

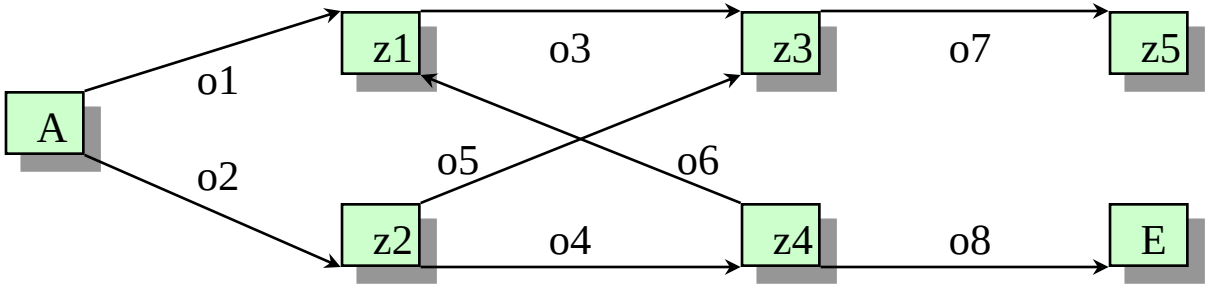
Reasons:
- Action can be carried out easier and faster within the information system.
- False Actions can be cancelled easier within the information system
- This means a simulation of actions before the real execution.

Problem types:
States: known, unknown, fixed, infinite, discrete, continuous
Operations: known, unknown, deterministic, stochastic, reversible
Environment: accessible, not accessible

Formal Problem:

Problem:		
delivered:	**- State**	Z = {z1,z2,...}
	- Operations	O = {o1,o2,...}, oi: Z→Z
	- Start state	A in Z
	- End state	E in Z
	- Cost function	K(P) = Σ K(oi), oi in P
requested:	**- Solution**	L = P(A,E)

State space: possible states connected by possible operations



K(o1) = 8
K(o2) = 2
K(o3) = 2
K(o4) = 1
else = 1

Path: P(z1,zn) direct or indirect connection between two states by a sequence of operations: (z1, o1, z2, o2, ..., zn)

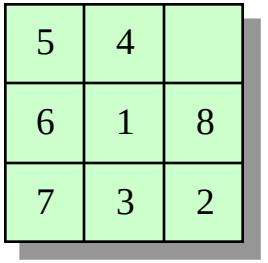
Form of problem components: - explicit (elements of a sets)
- implicit (functions or predicates)

Examples of problems:

8-Puzzle:

State: Arrangement of 8 numbered parts
in a 3x3 area

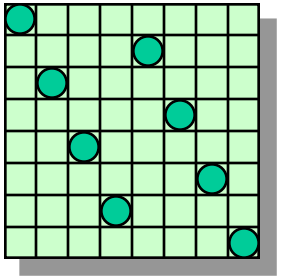
Operation: Motion of the empty field
left, right, up, down



8-Queens-Problem:

State: Arrangement of 0 to 8 queens in save positions
on a chess board

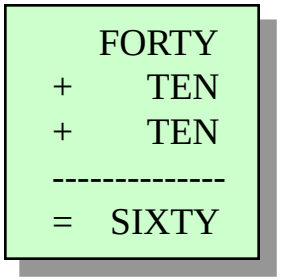
Operation: Positioning of one queen
on a free field



Crypto-Arithmetic:

State: Arithmetic task with letters, where same letters
have been replaced by digits

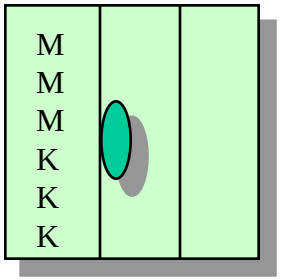
Operation: Replacement o each occurrence o on letter
by one digit



Missionary and Cannibal Problem:

State: Number missionaries, cannibals, and boats
on one side of the river

Operation: 1 or 2 persons change the side of the river
using the boat



Real Problems: Connection, Travelling, Navigation, Construction, Manufacturing, Proofs

General Search

Search: Find a path from the start state to the end state
Search tree: Tree structure in the search space with the start state as root and connection to all following states that can be reached directly by operations

Complexity:

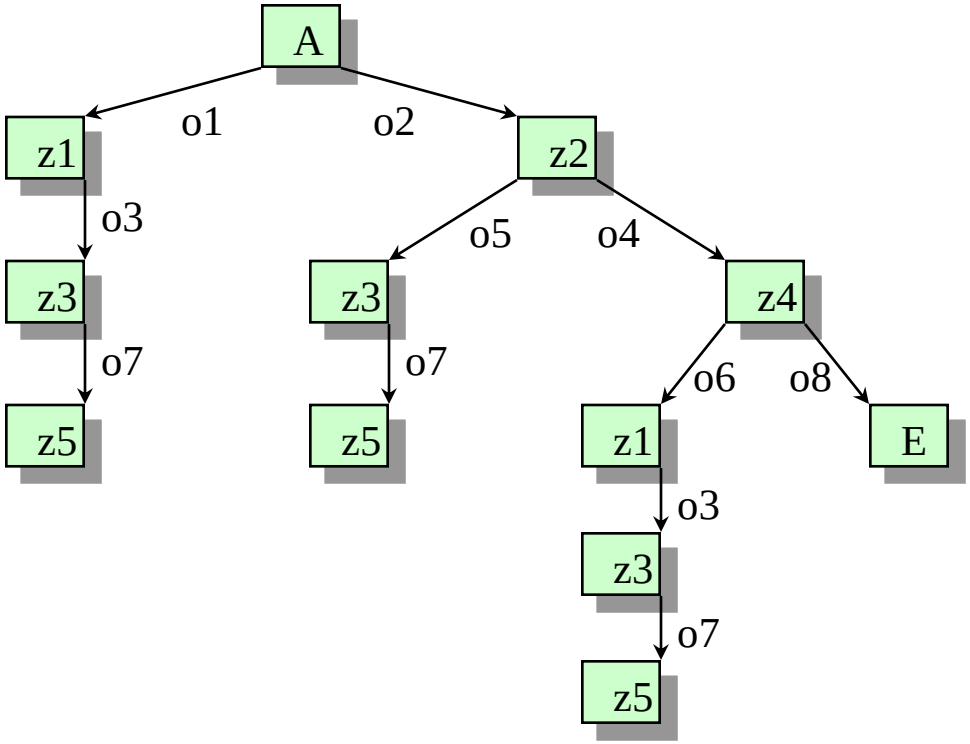
Breadth b: Number of successors of a node
Depth d: Distance between root and leaf

Data structures:

Problem:
(Start node, Operations,
End node, Cost function)
Node:
(State, Path, Depth, Costs)

Search algorithm:

Store start state in node set
Repeat: If node set is empty then stop.
Take one node from node set.
If node is end node; then finish.
Expand nodes with operations.
Add new nodes to node set.



Search Strategies

Strategy: Determining in which sequence the nodes are expanded

Evaluation:
- Completeness (one or all solutions)
- Time complexity (time for solution)
- Space complexity (memory for solution)
- Optimality (best solution)

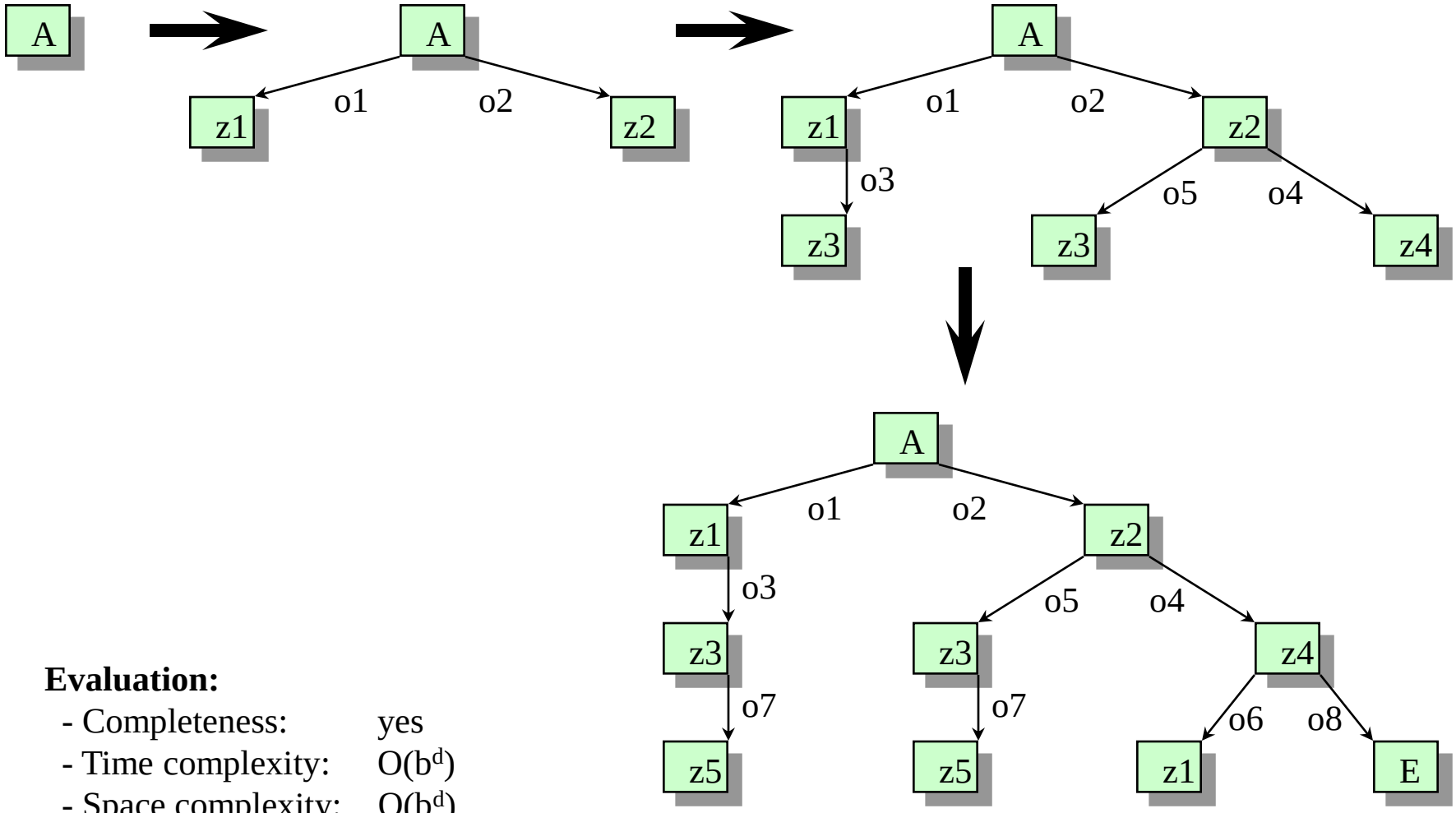
Space and time requirements: (Breadth = 10, 1000 Nodes/Second, 100 Byte/Node)

Depth	Node number	Time	Space
0	1	1 Milliseconds	100 Byte
2	111	100 Milliseconds	11 Kilobyte
4	11 Thousand	11 Seconds	1 Megabyte
6	1 Million	18 Minutes	111 Megabyte
8	100 Million	31 Hours	11 Gigabyte
10	10 Billion	128 Days	1 Terabyte
12	1000 Billion	35 Years	111 Terabyte

Direction:	- forward	- backward	- bidirectional
Expansion:	- breadth-first	- depth-first	
	- uniform costs	- restricted depth	- Iterative deepening
Information:	- blind / uniform	- informed / heuristic	
Memory:	- tree	- graph	

Breadth-First Search:

Expand always the node on the highest level of the tree

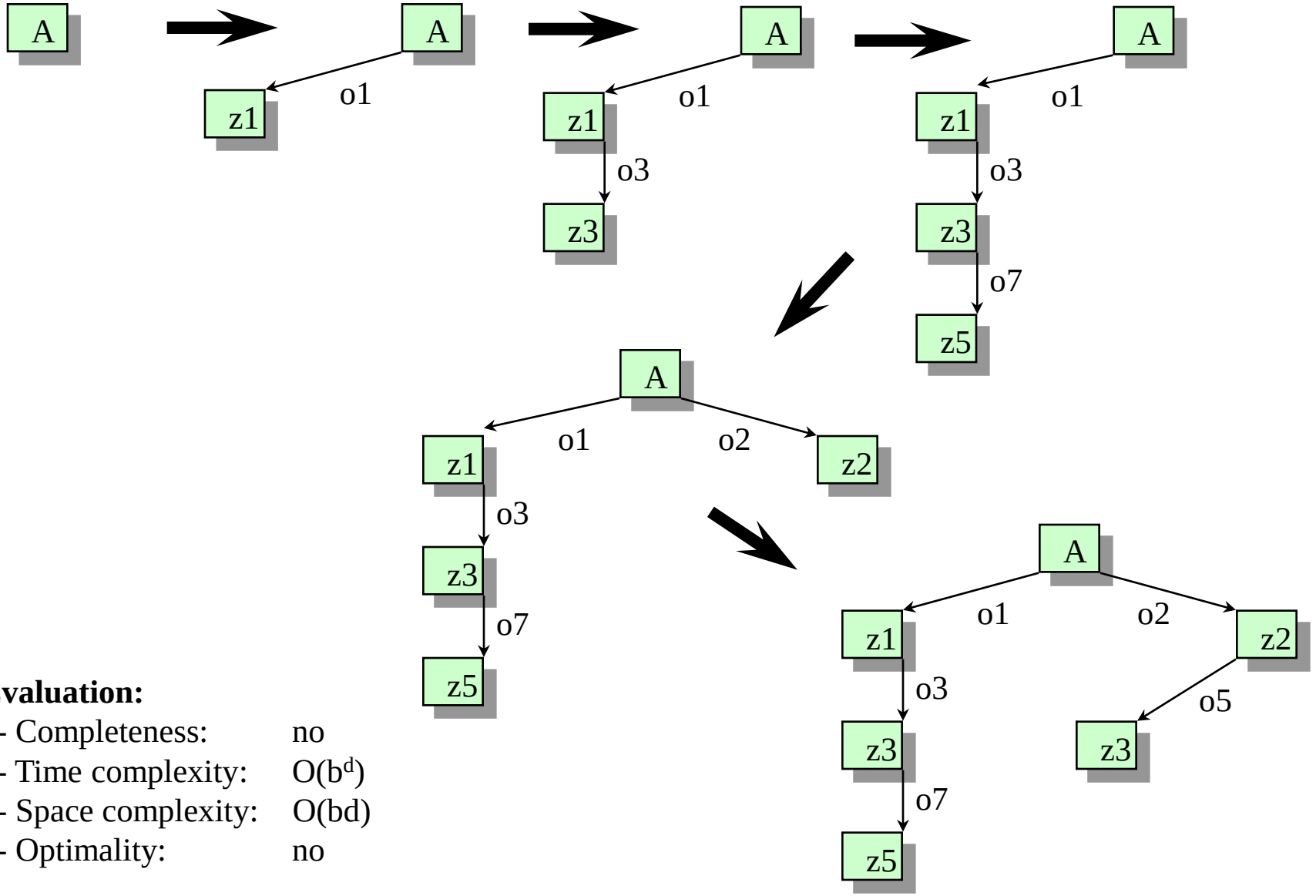


Evaluation:

- Completeness: yes
- Time complexity: $O(b^d)$
- Space complexity: $O(b^d)$
- Optimality: yes

Depth-First Search:

Expand always the node on the lowest level of the tree

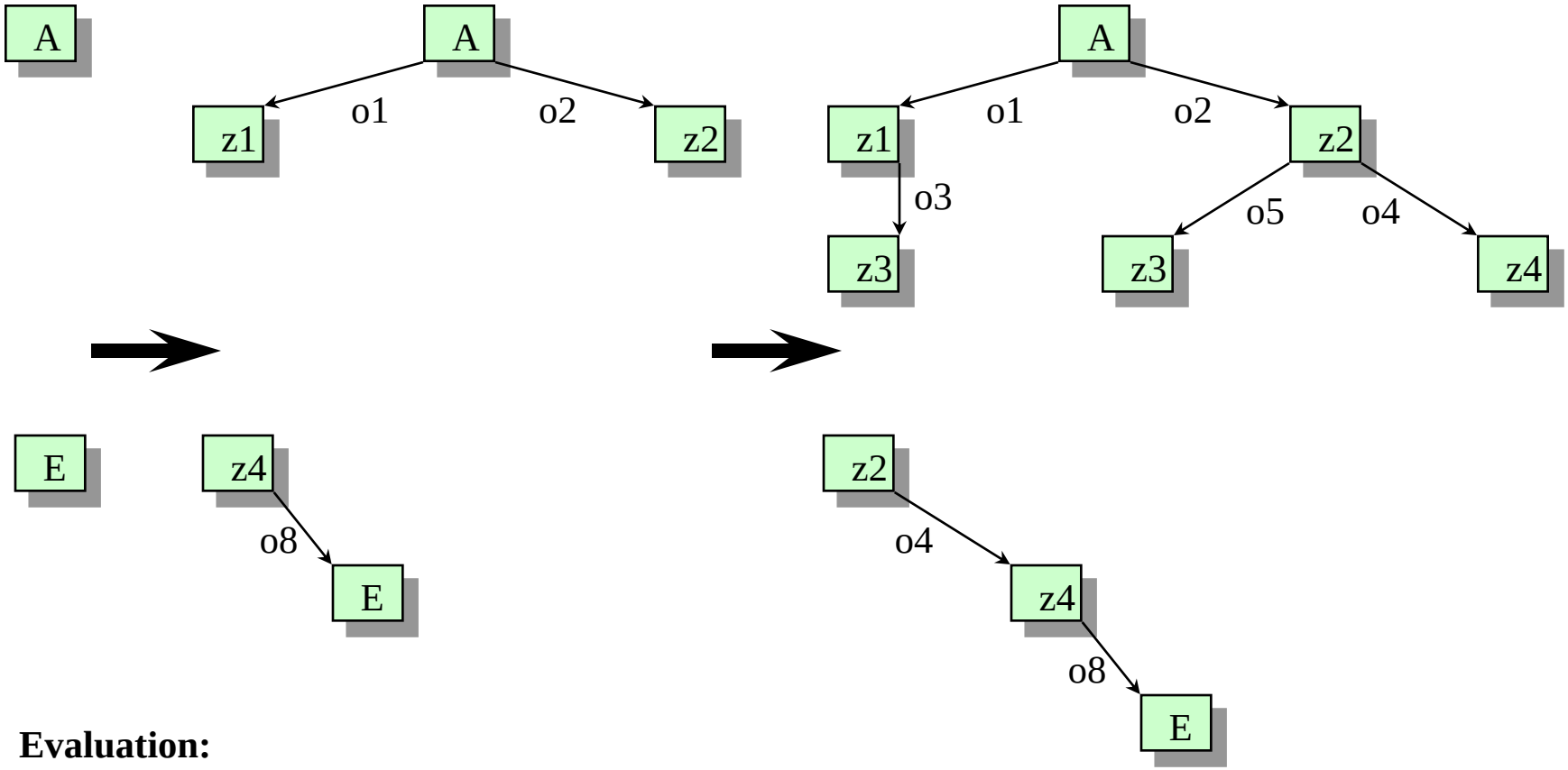


Evaluation:

- Completeness: no
- Time complexity: $O(b^d)$
- Space complexity: $O(bd)$
- Optimality: no

Bidirectional Search:

Simultaneous expand nodes forward from the start and backward from the end



Evaluation:

- Completeness: yes
- Time complexity: $O(b^{d/2})$
- Space complexity: $O(b^{d/2})$
- Optimality: yes

Further Search Strategies:

Uniform Cost Search:

Expand the nodes with the minimal costs
(special breadth-first search)

Evaluation:
- Completeness: yes
- Time complexity: $O(b^d)$
- Space complexity: $O(b^d)$
- Optimality: yes

Restricted Depth Search:

Expand the nodes to a restricted depth
(special depth-first search)

Evaluation:
- Completeness: $l>d$
- Time complexity: $O(b^l)$
- Space complexity: $O(bl)$
- Optimality: nein

Iterative Depth Search:

Depth-first search with stepwise increasing the depth
(special bidirectional search)

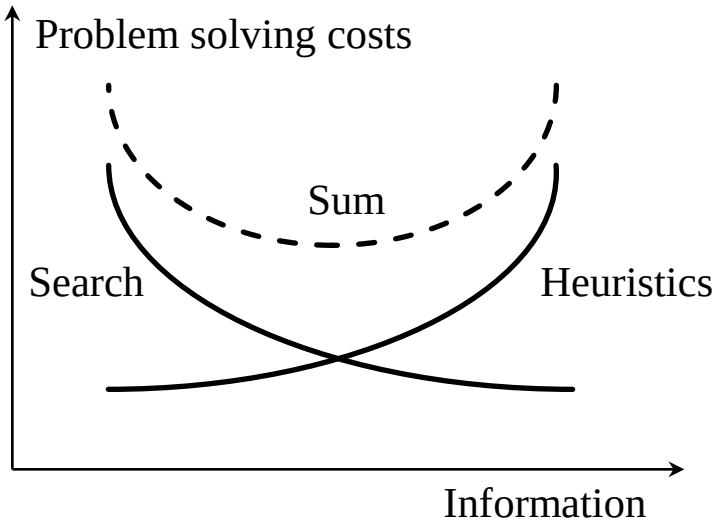
Evaluation:
- Completeness: yes
- Time complexity: $O(b^d)$
- Space complexity: $O(bd)$
- Optimality: yes

Heuristic Search

Heuristic Search: Use of knowledge about the domain to influence the search process
(informed search)
→ Controlling the selection of the nodes to be expanded

Problem relaxation: Weakening, Removing of restrictions of a problem
(The costs for the solution of a relaxed problem
are a good heuristic measure for the original problem)

Optimal Search Strategy:



Best-First-Search

Features:

- Ordering of the nodes according to the height of the assessed costs
- Evaluation function: delivers costs of a node
- Implementation with in a general search algorithm

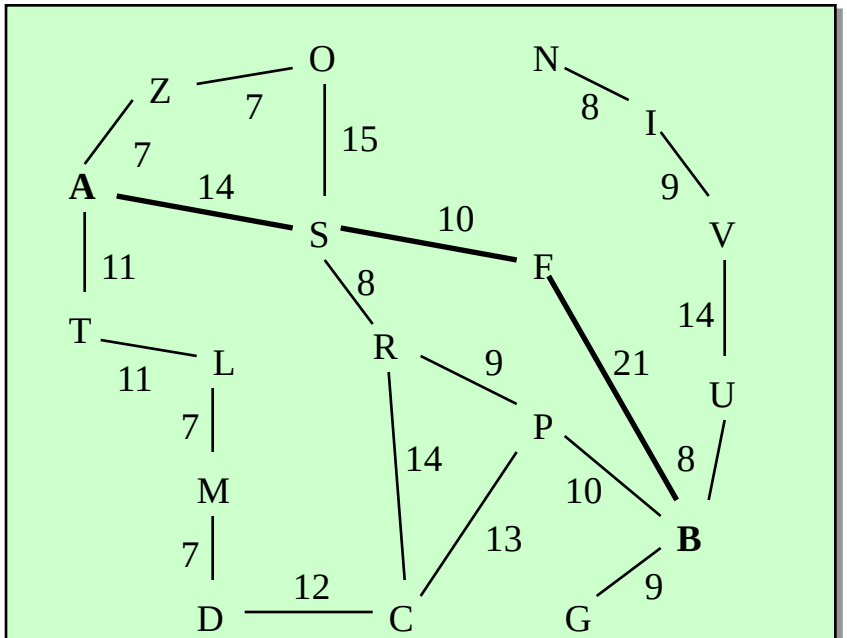
Minimisation of the expected costs (goal-oriented search):

- Expected costs h to reach the goal E starting from a node n

$$h(n) \geq 0, \quad h(E) = 0$$

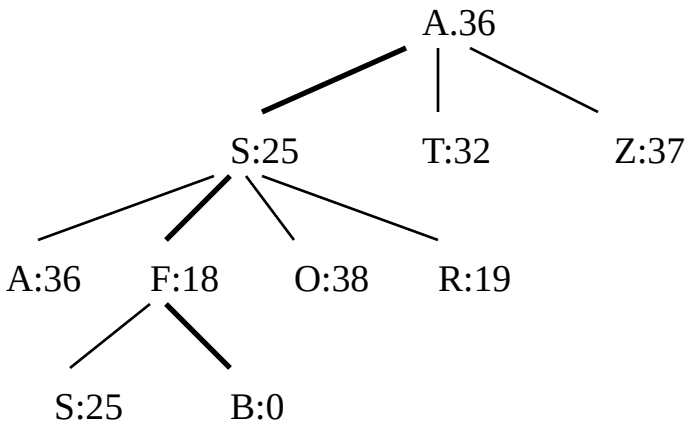
→ Features: not optimal, not complete

State space:



A:36 B:0 C:16 D:24 F:18 G:7 I:22 L:24 M:24
N:23 O:38 P:9 R:19 S:25 T:32 U:8 V:19 Z:37

Search tree:

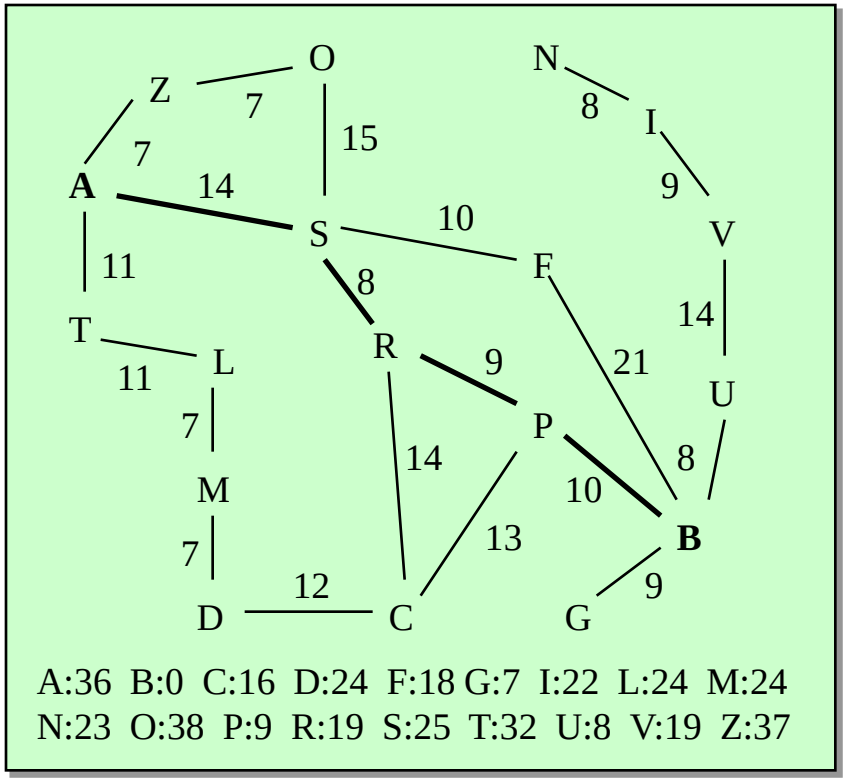


Minimisation of the former and expected costs (A*-Search):

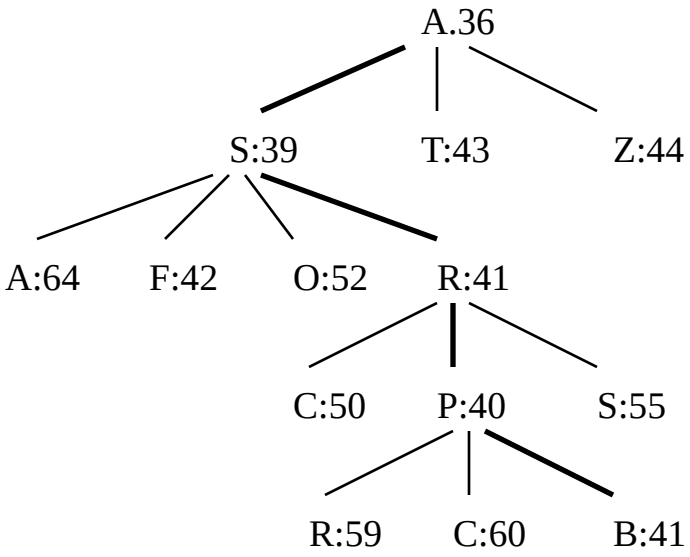
- Former costs g and expected costs h for a node n build the entire costs f
  (combination of minimal-costs and goal-directed search)
  expected minimal costs: $f(n) = g(n) + h(n)$, $f(n) \geq 0$, $f(E) = g(E)$
  real minimal costs: $f^*(n) = g^*(n) + h^*(n)$
  h optimistic: $h^*(n) \geq h(n) \geq 0$
  f monotone

→ Features: optimal, complete, efficient

Search space:



Search tree:



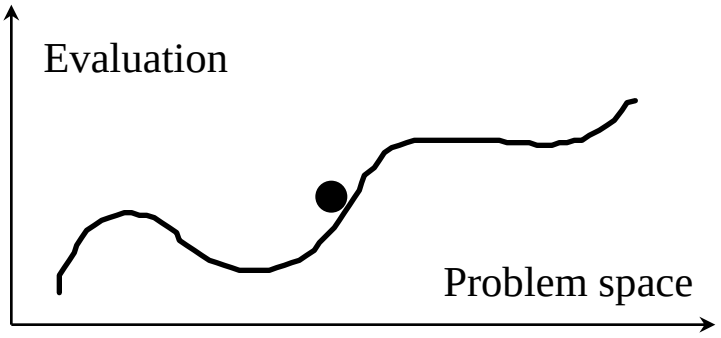
Iterative Improvement

Precondition: current problem state description
Algorithm: (1) Start with any problem state of a partial solution
(2) stepwise improvement of the solution

Hill climbing: Evaluation of the results of the operation to be carried out
Selection of the operation with the greatest ascent
Advantage: very fast
Disadvantage: only local maxima are found

Simulated annealing: Evaluation of the results of the operation to be carried out
Selection of the operation with the greatest ascent with a certain probability
Advantage: overcome local maxima
Disadvantage: very slow

Evaluation function:



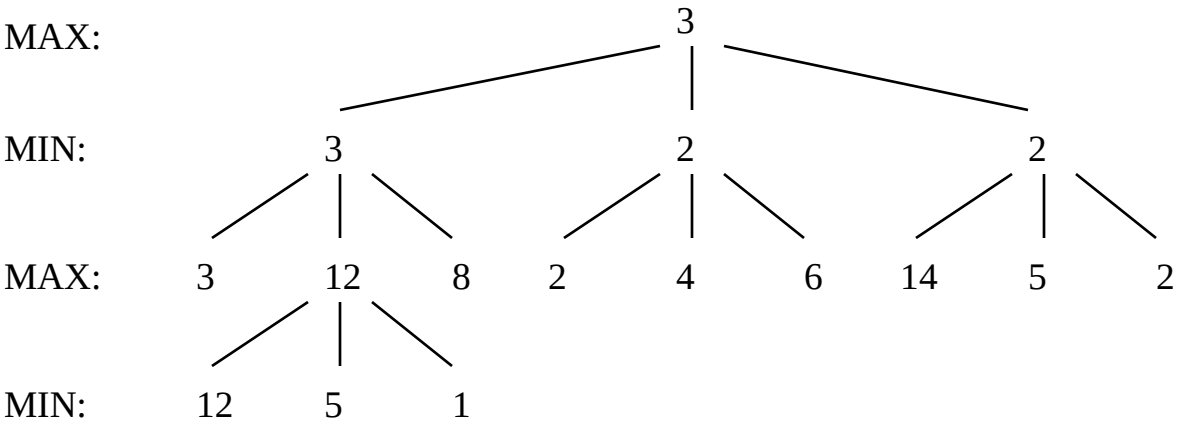
Min-Max-Approach

Goal: Determination of the optimal strategy for MAX und so the best first action

Algorithm:
(1) Generation of the entire search tree (to a certain depth)
(2) Computation of the gain for the terminal nodes
(3) Determining of the gains of all nodes of the next higher level by the determining the minima (for MIN) and the maxima (for MAX) of the subordinated nodes
(4) The gain of the root is the gain of the whole tree

Search tree:

Time complexity: $O(b^d)$



Example: Chess (Breadth b = 35, 1000 States / s) → Depth d = 4

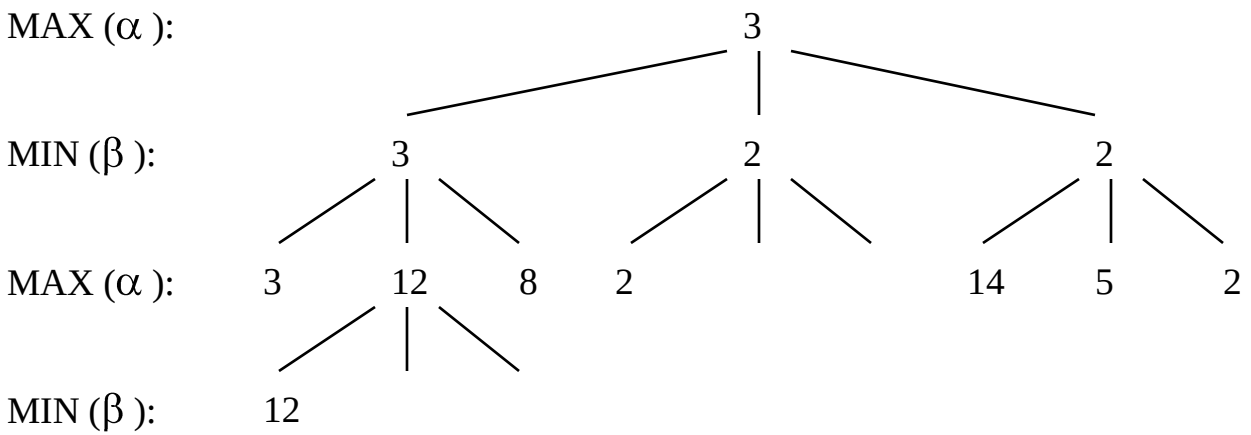
Alpha-Beta-Approach

Goal: Elimination of unimportant branches of the search tree

Algorithm: Depth-first-search with
- Computation of the node gains during the backtracking
  - α: highest gain with decision of MAX
  - β: lowest gain with decision of MIN
- Cutting the subtree if:
  - at decision of MAX: if the gain is higher than α
  - at decision of MIN: if the gain is lower than β

Search tree:

Time complexity: $O(b^{d/2})$



Example: Chess (Breadth b= 6, 1000 States / s) → Depth d = 8

12. Advanced AI Approaches

Prof. Dr.-Ing. habil. Wolfgang Oertel

Goals:

- Overview of additional intelligent concepts
- Concepts combining several technologies

Contents:

- Grammars
- Fuzzy Systems
- Learning Methods
- Case-based Reasoning
- Neural Networks
- Genetic Algorithms

Focus: Advanced Techniques

Grammars

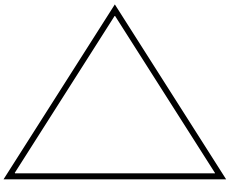
Grammar: Structure G = [T, N, R, s] with following features:
- T: not empty set of **Terminal symbols** (Alphabet)
- N: not empty set of **Nonterminal symbols**
- R: finite set of **Rules** r: $R \subseteq (T \cup N)^+ \times (T \cup N)^*$
- s: **Start symbol** $s \in N$

G defines a derivation relation $\Rightarrow$ on $(T \cup N)^*$ with $xuy \Rightarrow xvy$, if $(u,v) \in R$
The reflexive transitive closure of the relation is called $\Rightarrow^*$
by G generated formal language: set of all words on T, that can be derived by R from s:

$$L(G) = \{w | w \in T^* \wedge s \Rightarrow^* w\}$$

Natural language: G = [{art,adj,nom,verb,adv}, {s,np,npn,vp}, {(s,np vp), (np,art npn), (np,npn), (npn,nom), (npn,adj npn), (vp,verb), (vp,verb np), (vp,verb adv)}, s]
L(G) = {(art adj nom verb adv), (The big dog runs quickly)}

Graphics: G = [{a,b,c}, {s,g,h}, {(s,agh),(s,asgh),(hg,gh),(ag,ab),(bg,bb),(bh,bc),(ch,cc)}, s]
L(G) = {abc, aabbcc, aaabbbccc, ...}



Linguistic Inference

Fuzzy-System

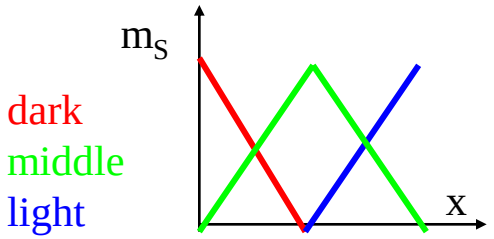
System for representation of different not exact, not precise, unsafe information and knowledge

Fuzzy Set:

set of ordered pairs of Elements and membership degrees

$$S = \{(x, m_S(x)) \mid x \in X,\ 0 \le m_S(x) \le 1\}$$

membership degree also possible by linguistic variable



Fuzzy Operation:

set-based operations

- Intersection: $A \cap B:\ m_{A\cap B}(x,y) = \min(m_A(x), m_B(y))$
- Union: $A \cup B:\ m_{A\cup B}(x,y) = \max(m_A(x), m_B(y))$
- Complement: $-A:\ m_{-A}(x) = 1 - m_A(x)$

Diagnosis:		Pain: yes:	Pain: no:
Caries:	yes:	0.04	0.06
	no:	0.01	0.89

Fuzzy Reasoning:

Conclusions with the help of fuzzy rules:

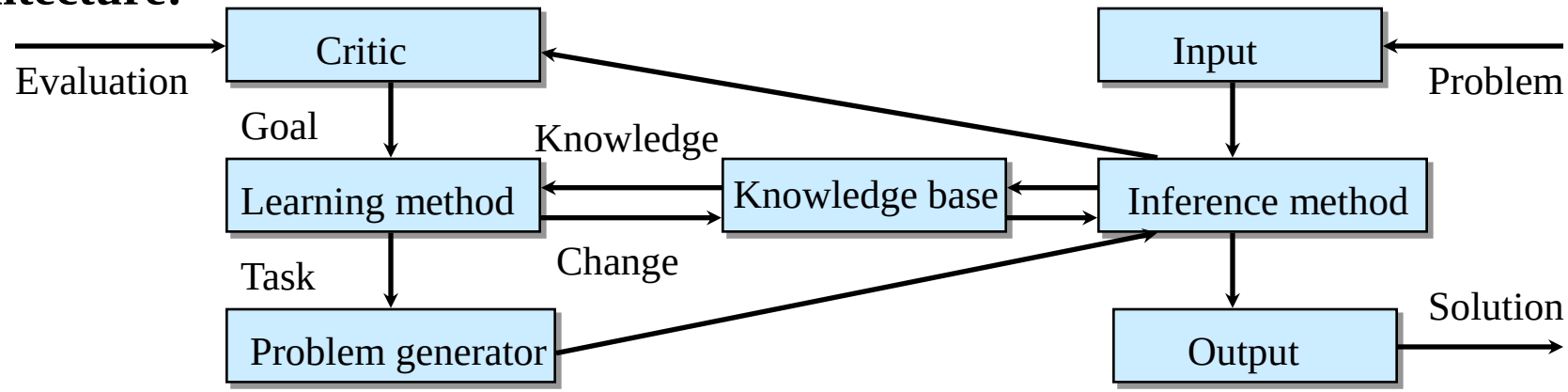
- Fact: *x is A*
- Condition: *if x is A then y is B*
- Junctions: *and, or, not*
- Fuzzycomposition: Min-Max-Rule for building complex sets
  - Correlation minimum: $m_{Bi+}(y) = \min(m_{Bi}(y), m_{Ai}(x))$
  - Correlation product: $m_{Bi+}(y) = m_{Bi}(y) \cdot m_{Ai}(x)$ $\quad m_S(w) = \max_i(m_{Bi+}(w))$
- Defuzzification: Problem solving by decomposition of complex sets

→ **Uncertain Inference**

Learning

Generalisation of concrete problem solutions for mapping of generic regularities

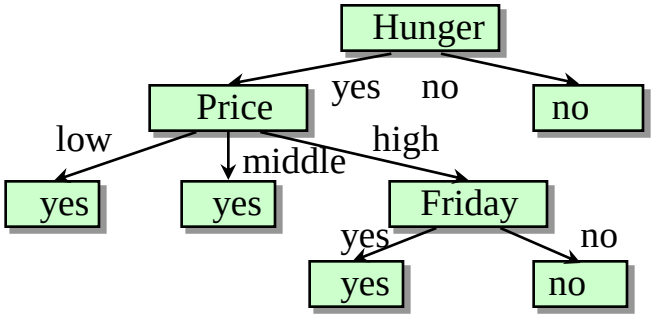
Architecture:



Classes: supervised / unsupervised

Example:

Hunger	Guests	Price	Rain	Type	Time	Friday	Wait
yes	many	high	no	chines	30-60	no	no
yes	many	high	no	chines	10-30	yes	yes
no	some	high	no	france	> 60	yes	no
yes	many	low	yes	italien	0-10	no	yes
no	no	low	yes	burger	0-10	yes	no
yes	many	middle	yes	chines	0-10	no	yes

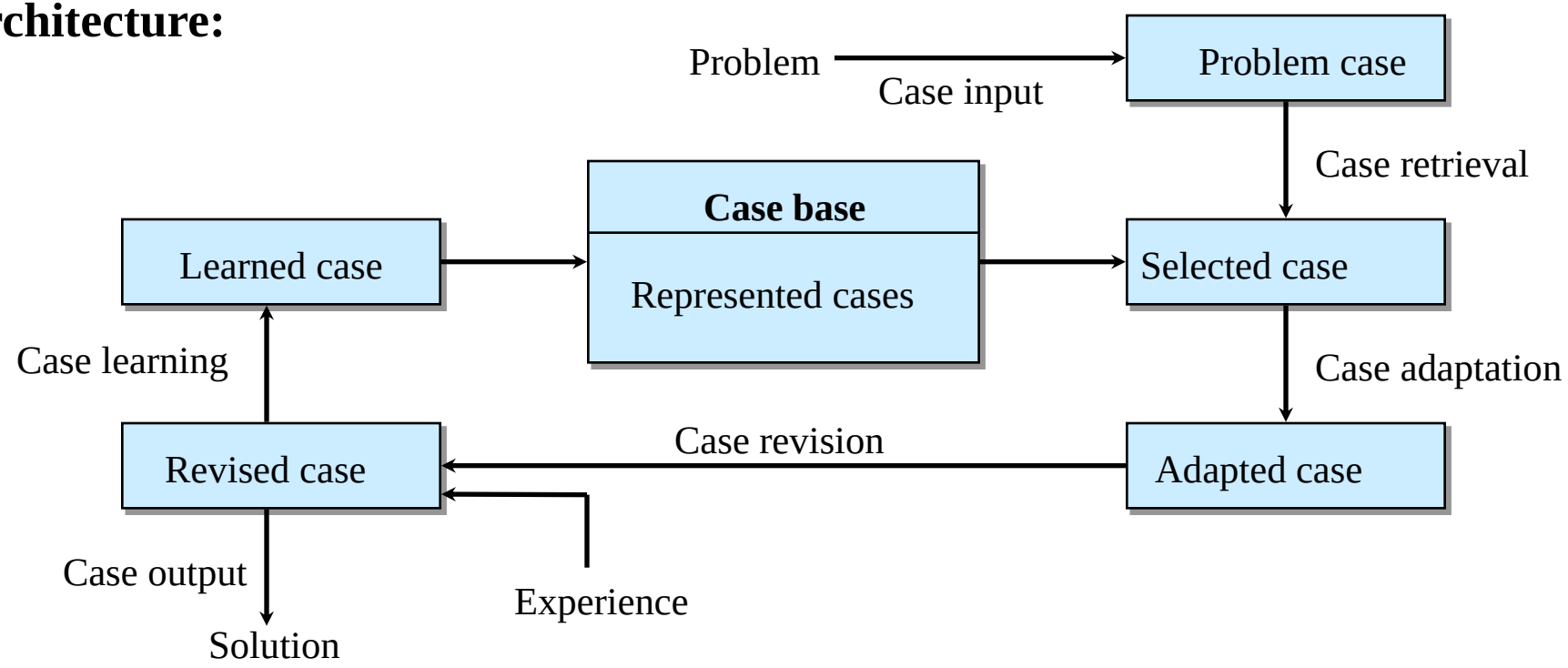


Inductive Inference

Case-Based Reasoning

Cyclic Approach that bases on the storage of former problems and solutions and their Reuse for the solution of similar current problems

Architecture:



Example:

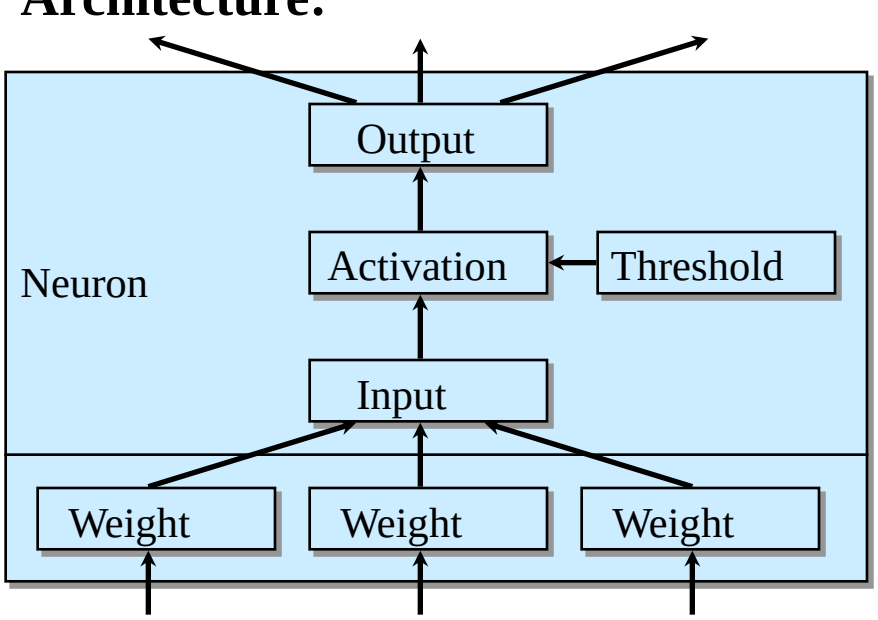
Problem: (Anamnesis Pain) (Alter 6) (Test ?) (Therapy ?))
Cases: (Anamnesis Pain) (Alter 45) (Test CT) (Therapy Antibiotica))
Solution: (Anamnesis Pain) (Alter 6) (Test MRT) (Therapy Antibiotica))

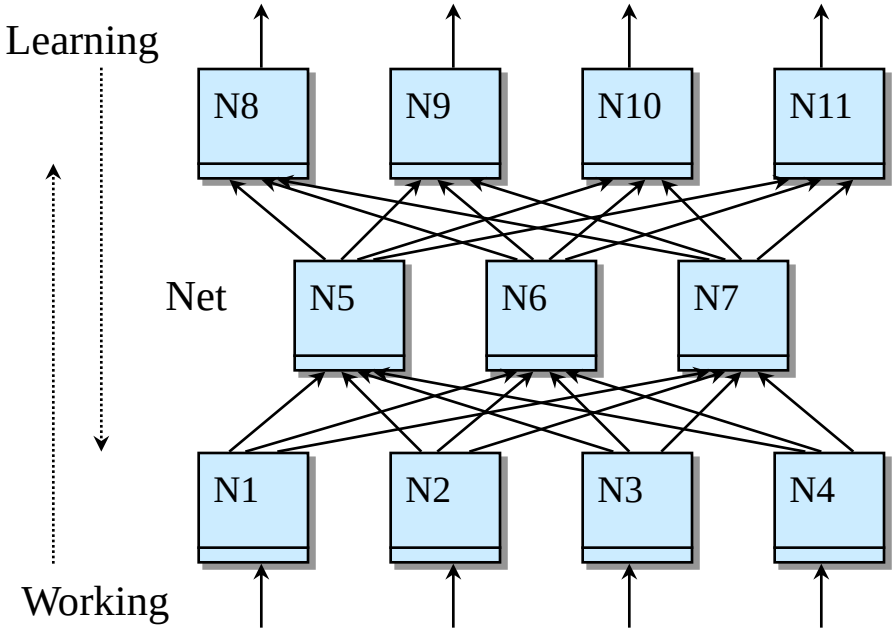
Analogue Inference

Neural Network

System that consists of many simple units that activate each other by sending of messages via changeable weighted connections with regard to thresholds

Architecture:





Special Networks:

- Feed-forward Net
- Back-propagation Net
- Kohonen Net
- Hopfield Net

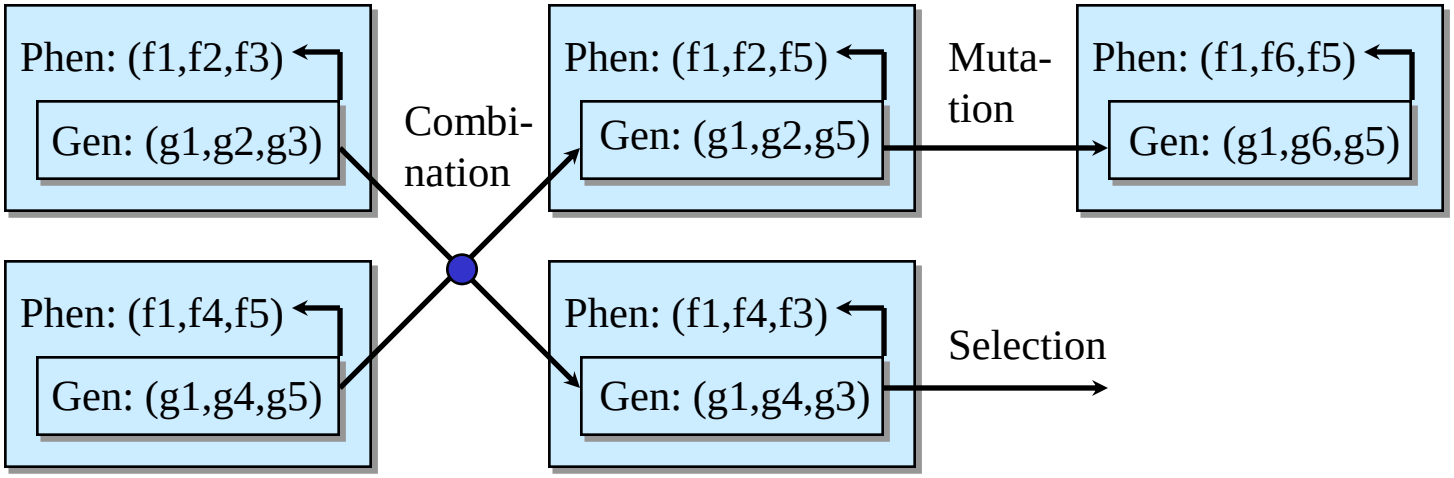
Dimension: Neurons: 10^{12} ; Synapses per Neuron: 10^{3}
Frequency: 10^{2} Hz; Velocity: 10^{2} m/s

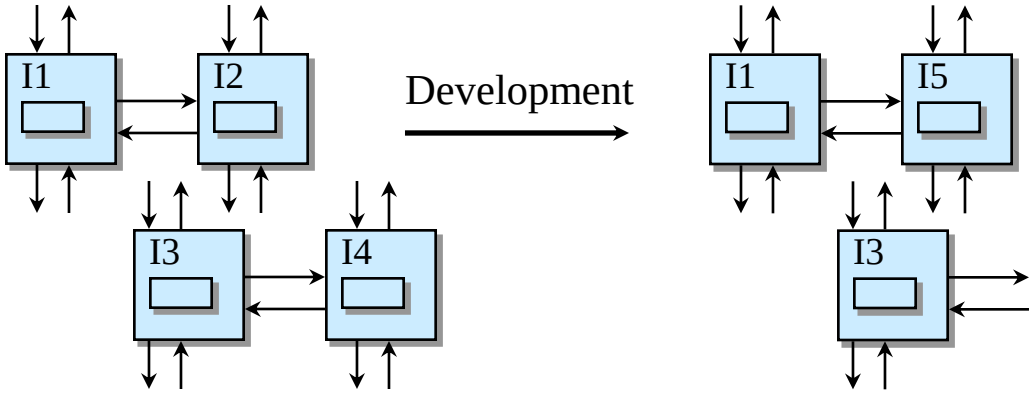
Neural Inference

Genetic Algorithm

System that consists of a population of individuals. Their behaviour is determined by genetic structures that develop evolutionary by combination, mutation, and selection

Architecture:





Natural intelligence: Individual: 10^5 Gene (* 10^{13} Instances)
Population: 10^9 Individuals
Species: 10^9; Time: 10^9 Years; Mutations: 10^{-5}

Evolutionary Inference

13. Programming Language Lisp

Prof. Dr.-Ing. habil. Wolfgang Oertel

Goals:

- Getting to know the basic concepts of an AI programming language
- Simple AI programming

Contents:

Focus: Lisp basics

Basics of the Language Lisp

What is Lisp?

LISP = LIStProcessor
- originally: developed by J. McCarthy for symbolic processing
- today: usable as higher universal programming language, especially for AI problems
- functional language: Programs compute functions that map input data to output data
- related languages: Scheme, ML, Miranda, Logo, Smalltalk, Forth

History:
- Predecessors 1956 - 1958: IPL, FORTRAN
- Kernel 1959 - 1965: LISP1, LISP1.5, StanfordLISP1.6
- Extensions, Dialects 1966 - 1983: StandardLISP, FranzLISP, ZetaLISP, InterLISP, MacLISP, muLISP, SCHEME
- Standardisation 1984: CommonLISP, EuLISP, ISO LISP
- Objektorientation 1988: CLOS

Concepts:
- Processing of complex dynamic data structures (lists as passive structures)
- Functional programming (functions as active structures)
- Unified handling of data and programs
- interactive, interpretative programming
- integrated programming environment
- language embedding
- compromise between adequate programming and efficient implementation

Intuitive Introduction

Numeric:

```
2                                       >   2
(+ 2 3)                                 >   5
(+ (* 3 4) 2 0.5)                       >   14.5
(setf a 2.5)                            >   2.5
a                                       >   2.5
(- a 3)                                 >   -0.5
((lambda (x y) (-(* x x)y))3 a)         >   6.5
(defun f1 (x y) (-(* x x)y))            >   f1
(f1 3 a)                                >   6.5
```

Symbolic:

```
'(* 3 4)                                    >   (* 3 4)
(car '(* 3 4))                              >   *
(cdr '(* 3 4))                              >   (3 4)
(cons '/ '(3 4))                            >   (/ 3 4)
(car (cdr '(* 3 4)))                        >   3
(setf b '(* 3 4))                           >   (* 3 4)
b                                           >   (* 3 4)
(cons '+ (cons b '(2 0.5)))                 >   (+ (* 3 4) 2 0.5)
((lambda (x y) (cons x (cdr y))) '/ b)      >   (/ 3 4)
(defun f2 (x y) (cons x (cdr y)))           >   f2
(f2 '/ b)                                   >   (/ 3 4)
```

Evaluation:

```
(quote a)                    >  a
a                            >  2.5
(eval a)                     >  2.5
(quote b)                    >  b
b                            >  (* 3 4)
(eval b)                     >  12
(quote (f2 '/ b))            >  (f2 '/ b)
(f2 '/ b)                    >  (/ 3 4)
(eval (f2 '/ b))             >  0.75
```

Syntax (Data structure)

Object:

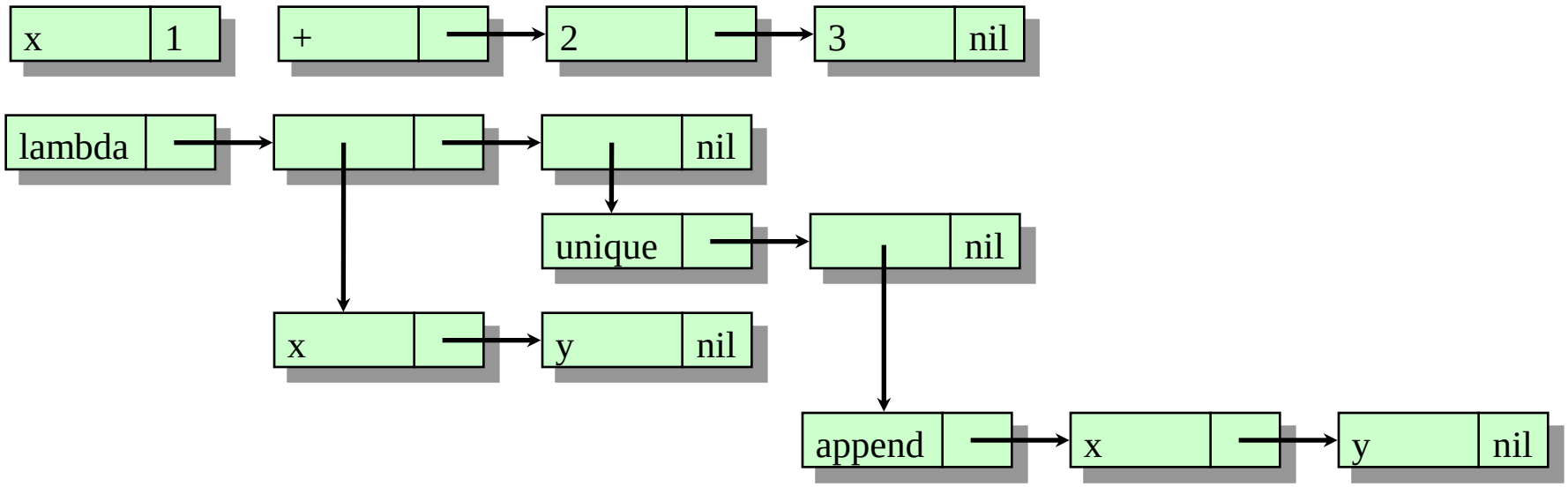
OBJECT (syntactically allowed / representable expression
- ATOM (elementary object): NUMBER, SYMBOL
- CONS (composite object): ordered pair of objects OBJECT

Internal Representation: Network of Cells and Pointers

Example:

```
1       -4       2.3       -44.55

ax      +        +-!       append
```



External Representation: linear, use of names and brackets

- ATOM: Print name (Sequence of characters without space and bracket)
  NUMBER: defined syntax SYMBOL: else
- CONS: dotted pair (OBJECT . OBJECT)

Example:
```
(x . 1)
(+ . (2 . (3 . nil)))
(lambda . ((x . (y . nil)) .
           ((unique . ((append . (x . (y . nil))) . nil)) .
            nil)))
```

Additional rules for 2. Lisp-Object:
- NIL: "." and 2. Object not necessary
- CONS: "." and brackets of the 2. Object not necessary

Example:
```
(x . 1)
(+ 2 3)
(lambda (x y) (unique(append x y)))
```

LIST: NIL, CONS

TRUE LIST: each CONS contains as 2. object a list

Semantics (Program structure)

Form:

FORM (semantically allowed / evaluable expression)
- NUMBER (self-evaluating): Number value
- T, NIL (self evaluating): T, NIL
- SYMBOL (Variable): Variable value (binding by (setf x y))
- TRUE LIST (Function call): Function value, side effect

Evaluation of a true list (f x1 x2 . . . xn):
1. Sequential evaluation of the arguments x1, x2, ..., xn
2. Application of the function f to the results of the evaluation
3. Providing of the function value

Example:

```
1                          >   1
t                          >   t
a                          >   2.5
b                          >   (* 3 4)
(+ 1 2)                    >   3
(+ (* 2 a) -4)             >   1.0
(cons  1(car(cdr b)))      >   (1 . 3)
(setf a 3)                 >   3
```

Function:

FUNCTION (Object, applicable to objects)
- LAMBDA-EXPRESSION (Function description): (lambda l k)
  - Lambda-List l: List of symbols as formal Parameters
  - Lambda-Body k: Form in which symbols of l are bounded as local variables
- SYMBOL (Function name): f (Binding by (defun f l k))

Example:

```
((lambda (x y) (+ (* x x) y)) 2 3)              >   7
(defun f3 (x y) (+ (* x x) y))                  >   f3
(f3 2 3)                                        >   7
(defun f4 (x) (- (f3 x 4) 3))                   >   f4
(/ (f3 2 3) (f4 1))                             >   3.5

(defun faculty (x)
  (if(= x 0) 1 (* x (faculty (- x 1)))))
(faculty 5)                                     >   120
```

Predefined Function Packages

Interpreter:

(read) - Object transformation extern > intern
(print x) - Object transformation intern > extern
(eval x) - Object evaluation (standard)
(quote x) - Object evaluation avoiding (short: ’x)

Top-Level-Interpreter-Loop:
(loop (print (eval (read))))

Predicates:

(atom x)	(consp x)	(null x)	(listp x)	
(symbolp x)	(numberp x)	(eq x y)	(equal x y)	

Arithmetic:

(+ x1 ... xn)	(- x1 ... xn)	(* x1 ... xn)	(/ x1 ... xn)	(= x1 ... xn)
(< x1 ... xn)	(> x1 ... xn)	(<= x1 ... xn)	(>= x1 ... xn)	(/= x1 ... xn)

Logic:

(not x) (and x1 ... xn) (or x1 ... xn)

Conditional Forms:

(if x y z) (cond (x1 y11 ... y1m) ... (xn yn1 ... ynm))

Truth Values:

false: NIL
true: T or any other object unequal NIL

List Processing

Basic Functions:

(car x)	- 1. object of a list x (1. object of a cons)
(cdr x)	- List x without 1. object (2. object of a cons)
(cons x y)	- Adding object x in front of a list y (building a cons of the objects x and y)
(list x1 ... xn)	- Building a list of objects x1, ..., xn
(append x1 ... xn)	- List of objects from lists x1, ..., xn
(remove x y)	- List of objects from list y without object x

Principal: car, cdr deliver pointers to existing objects
cons, ... generate new lists and deliver the pointer on them
No changing of arguments

Destructive Functions:

(setf x y)	- Object x of a list is replaced by object y
(nconc x1 ... xn)	- Concatenation of the lists x1, ..., xn
(delete x y)	- Deletion of object x from list y

Principal: Changing of arguments

Set Functions:

(member x y)	- Element
(subsetp x y)	- Subset
(union x y)	- Union
(intersection x y)	- Intersection
(set-difference x y)	- Difference

Examples of Objects and Forms

Objects:

```
5x341
(m(n)((a))(p r))
”1&-/A”
139.2
(nil nil)
10.3E-2
(lambda(x y z)(-x z y))
(setf q 2)
(1 - 2 - 3 - 4)
(- 1 2 3 4)
(1 2 3 4 . 5)
(3.4.5)
2...4
(1 \( \( 4)
3\.14
```

```
(1(2))(3 4)5)
888(
(1 . 2 3 4 5)
(3 . 4 . 5)
```

Forms:

```
(atom 'lisp)                          >   t
(consp 12.456)                        >   nil
(null 0)                              >   nil
(listp nil)                           >   t
(symbolp '(d r e s d e n))            >   nil
(numberp 'zwei)                       >   nil
(eq '(1 2) '(1 2))                    >   nil
(equal '(1 2) '(1 2))                 >   t

(+(-(*(/ 12 3 2)5)1 -2)-11)           >   0
(* 2.5 3 1)                           >   7.5
(< 1 2 3.2 7)                         >   t
(= 3 3.0 (/ 6 2))                     >   t

(not 23.5)                            >   nil
(not nil)                             >   t
(and 1 t(setf a 3)nil(setf b 4)a)     >   nil
(or nil(setf a nil)1(set b t))        >   1

(if t 1 2)                            >   1
(if(and(< a 5)(> a 5))3 4)            >   4
(cond ((< 3 2) 4 5 6)
          ((= 3 3)(setf b 5)7)
          (t(- 1 2)))                 > 7
```

Forms:

```
(setf y (cons 'x 5))                    >   (x . 5)
(car y)                                 >   x
(cdr y)                                 >   5
(setf x '(2 3))                         >   (2 3)
(setf x (cons 1 x))                     >   (1 2 3)
(+ (car(cdr(cdr x))) (caddr x))         >   6
(list 1 2 x 3)                          >   (1 2(1 2 3)3)
(list x '(4 5))                         >   ((1 2 3)(4 5))
(append x '(4 5))                       >   (1 2 3 4 5)
(remove 2 x)                            >   (1 3)
x                                       >   (1 2 3)
(nconc x '(4 5))                        >   (1 2 3 4 5)
x                                       >   (1 2 3 4 5)
(delete 2 x)                            >   (1 3 4 5)
(setf x(cdr x))                         >   (3 4 5)
(setf(cadr x)6)                         >   6
(setf(caddr x x)                        >   (3 6(3 6(3 6(...))))
(setf x '(3 6 5))                       >   (3 6 5)
(setf(cddr x)x)                         >   (3 6 3 6 3 6 ...)
(member 4 '(1(4 5)2 3))                 >   nil
(member 2 '(1(4 5)2 3))                 >   (2 3)
(intersection '(1 2 3) '(3 4 5))        >   (3)
(union '(1 2 3) '(3 4 5))               >   (1 2 3 4 5)
```

Program Example

Union of sets: Sets are represented as lists

Data structure: `(1 2)  (2 3)  (1 2 3 4)  (a b(c)(d e))  (h b(d e)(a))`

Program structure:

```
(defun my-union (x y) (unique (append x y)))
 (defun unique (x)
  (cond ((null x) nil)
   ((member (car x) (cdr x)) (unique (cdr x)))
   (t (cons (car x) (unique (cdr x))))))
```

Trace protocol:

```
>   (my-union '(1 2) '(2 3))
  >   (append '(1 2) '(2 3))
  <   (1 2 2 3)
  >   (unique '(1 2 2 3))
    >   (unique '(2 2 3))
      >   (unique '(2 3))
        >   (unique '(3))
          >   (unique nil)
          <   nil
        <   (3)
      <   (2 3)
    <   (2 3)
  <   (1 2 3)
<   (1 2 3)
```

14. Common Lisp

Prof. Dr.-Ing. habil. Wolfgang Oertel

Goals:

- Overview of a professional system to implement artificial intelligent system
- Standardised Programming language for AI applications

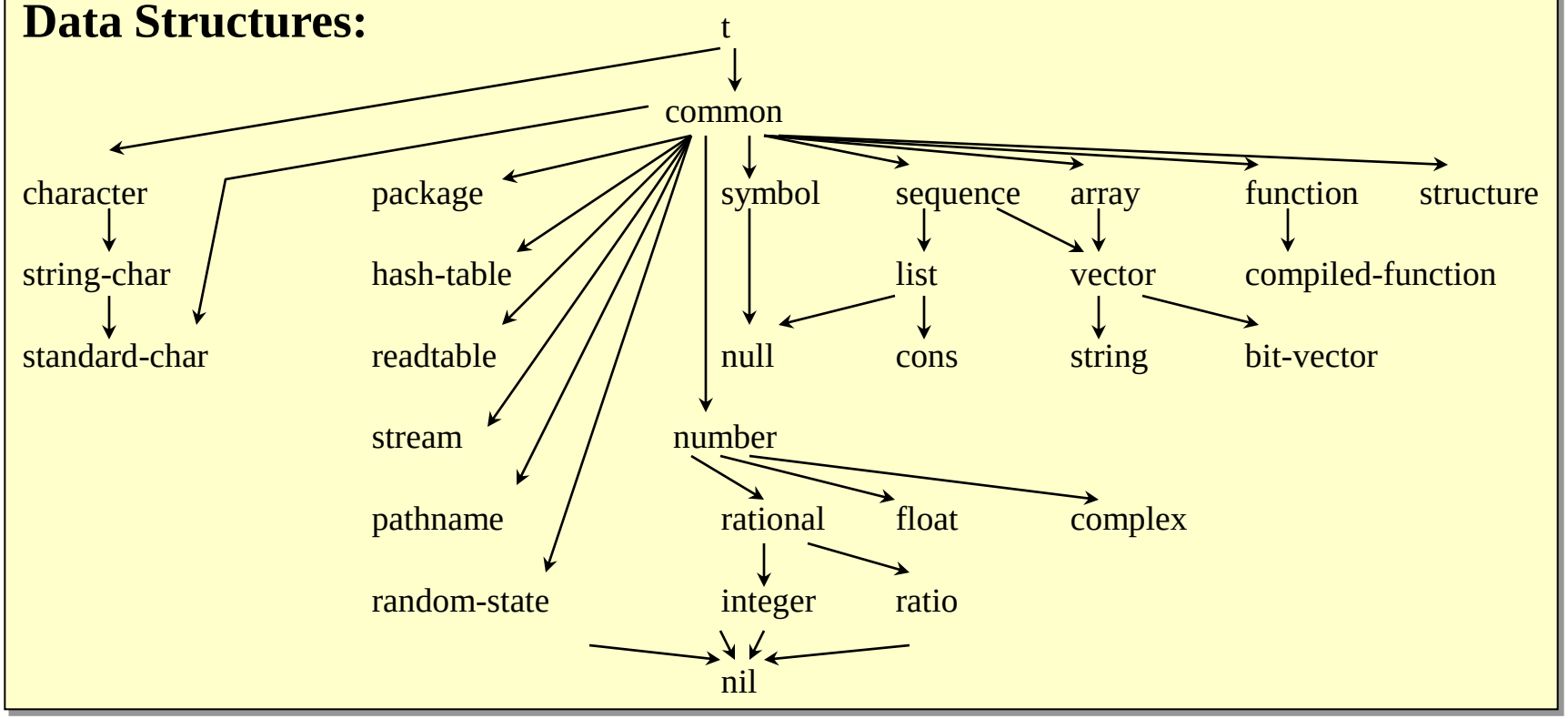
Contents:

- Language Overview
- Data Structures
- Program Structures
- Environment

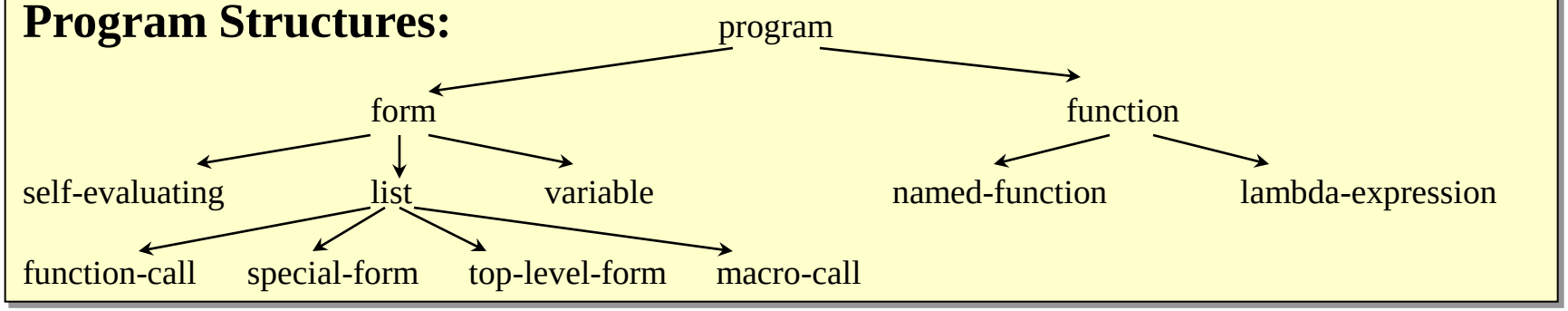
Focus: Standard AI language

Language Overview

Data Structures:



Program Structures:



Data Structures

character: Characters with Bits, Font, Code attribute and name, syntax: #font\bits-name
Example: `#\A` `#\Control-B` `#\Control-Meta-1` `#\Excape` `#3\4`

string-char: Characters for Strings; Font- and Bits attribute 0 Example: `#\a #\backspace #\tab`

standard-char: Standard characters (94 ASCII) Example: `#\3` `#\Z` `#\space` `#\newline`

readtable: Determining of the syntax type of a character

hash-table: efficient mapping between two sets, hash function
(make-hash-table test size) (sxhash object)

random-state: Random state generator (make-random-state state) (random number state)

pathname: File description Example: `#p"e:\\lisp\\test.lsp"`
(make-pathname host device directory name type version) (pathname-device pathname)

stream: Source or sink of data
(open file) (make-string-output-stream) (make-string-input-stream string)
(print object stream) (read stream) (format stream string objects)
(read-byte stream) (read-char stream) (read-line stream)
(write-byte int stream) (write-char char stream) (write-line string stream)
(close stream) Example: `(setf file(open"data.lsp"))(print(read file))(close file)`

package: Name space for symbols
(make-package name) (use-package packages) (unuse-packages packages)
(intern string) (unintern symbol)
(export symbols package) (import symbols package)

number: Number of any length and basis
<, >, =, <=, /=, +, -, *, /, expt, log, sqrt, abs, sin, cos, atanh, round, mod, logand, logor, ...
(coerce object type)
Example: `1234  #xf1b  #7r165  88888888888888888888888888888888888888`

rational: Rational number Example: `1.2  -99.99`

integer: Fixed number (fixnum oder bignum) Example. `33  -44`

ratio: Relation of two fixed numers Example: `2/3  -10/7`

float: Floating point number with different precisions (E, S, F, D, L) Example: `-0.77e10  33.2d-1`

complex: Complex number Example: `#c(0 1) #c(5/3 7.0f2)`
(complex re im) (realpart nu)

array: Multi-dimensional field of objects with integer indexes
Example: `#2a((1 2)(2 3)(3 4)) #3a(((NIL NIL)(NIL NIL))((NIL NIL)(NIL NIL)))`
(make-array '(dim1 dim2 ...) (aref array idx1 idx2 ...) (setf (aref array idx1 idx2 ...) value)

vector: One-dimensional field Example: `#(1 2.3 wo (3(4 5)))`

string: Character string Example: `"Work"  "1 + 2 - 3"`
(make-string size) (char string idx) (string= string1 string2)

bit-vector: Sequence of bits Example: `#*011001`
(bit vector idx) (bit-not vector) (bit-and vector1 vector2) (bit-or vector1 vector2)

structure: Composite data object with a fixed number of named objects
Example: `#s(gear performance 1000 rotations 50 components (shaft bearing))`
(defstruct name slot1 value1 slot2 value2 ...) (make-name structure) (name-slot structure)

symbol: Named object consisting of the components printname, value, function, propertylist, and package Example: `car  user::f1  #:a12`

Generation:	(make-symbol string)	x (gensym)
Access:	(symbol-name x)	(print x)
	(symbol-value x)	x
	(symbol-function x)	(x y)
	(symbol-plist x)	(get x y)
	(symbol-package x)	
Changing:	(setf (symbol-... x) y)	(setf(get x y) z)
	(setf x y) (makeunbound x)	(defun x y z) (fmakunbound x)

Example:
```
(setf x 1) (defun x(a b)(+ a b)) (setf(get 'x 4)2) (setf(get 'x 1)3)
(symbol-name 'x)          >   "x"
(symbol-value 'x)         >   1
(symbol-function 'x)      >   (lambda(a b) (+ a b)
(symbol-plist 'x)         >   (1 3 4 2)
(symbol-package 'x)       >   "user"
(x x (get 'x 4))          >   3
```

sequence: Ordered sequence of objects Example: `(1 2 3)  #(1 2 3)  "123"`
(concatenate type seq1 seq2) (length seq) (sort seq test) (some pred seq) (every pred seq)

cons: Ordered pair of objects
list: List of objects
null: Contains only NIL as objects

nil: Empty data type
t: All data type
common: Data types of the Common Lisp Standard

function: Function Example: `#'append` `#'(lambda(x y)(* x y))`

Lambda-List: (x1 x2 — Position parameter obligatory
&optional x3 (x4 v4) — Position parameter optional
&rest x5 — Rest parameter
&key x6 (x7 v7) — Keyword parameter
&aux x8 (x9 v9) — Local variable

(funcall f obj1 obj2 ...) (apply f objlist) (mapc f list1 list2 ...) (mapcar f list1 list2 ...)
(member obj list :test f) (sort seq f)
(values form1 form2 ...) (multiple-value-setq (var1 var2 ...) form)

Example: (

```
(funcall (caddr '(+ - / *)) 1 2 4)          >   0.125
(apply #'+ '(1 2 3 4))                      >   10
(mapcar #'numberp '(1 a (2 3)))             >   (t nil nil)
(mapcar #'+ '(1 2 3) '(2 3 4) '(3 4 5))     >   (6 9 12)
(defun f1 (x1 &optional x2 &rest x3) (list x1 x2 x3))
(f1)                                        >   break
(f1 1)                                      >   (1 nil nil)
(f1 1 2 3 4 5)                              >   (1 2 (3 4 5))
(defun f2 (x1 x2 &key x3 x4 &aux x5 (x6 6)) (list x1 x2 x3 x4 x5 x6))
(f2 1 2 :x4 4)                              >   (1 2 nil 4 nil 6)
```

compiled-function: Compiled function

(compile function-name lambda-expression) (compile-file file)

Backquote-Macro: Evaluation within a quotation

Example:

```
`(a b ,(+ 3 4) d)                           >   (a b 7 d)
(setf x `(c d))
`(a b ,x (,(cadr x)) (e ,x))                >   (a b (c d) (d) (e (c d)))
```

Program Structures

function: Function that can be applied to objects

Binding: - unbound

Example: `(funcall #'(lambda (x y) (+ x y)) 1 2)`

- special (global, temporally limited, by Heap realised, symbol-function)

Example: `(defun f (x y) (+ x y))  (f 1 2)  (fmakunbound 'f)`

- lexical (local, spatially limited, by Stack realised)

Example: `(labels ((f (x y) (+ x y))) (f 2 1))`

Search strategy: lexical > special > break

named-function: Named function

lambda-expression: Lambda-expression

form: Form, evaluable obect

self-evaluating: Object with itself as value (number, character, string, bit-vector, t, nil)

variable: Variable, object with bound value

Binding: - unbound Example: `1`

- special (symbol-value) Example: `(setf x 1)  x  (makunbound 'x)`
- lexical Example: `(lambda (x) (- x 1)) (let ((x 1)) x)`

Example: `(setf x 1)  (defun f1 (x y) (+ x y (f2 x y)))`

```
(defun f2 (a b) (- a b x y (funcall #'(lambda (a) (* a b x) 2)))
(f1 3 4)          >   break
(setf y 5)
(f1 3 4)          >   -8
```

Search strategy: lexical > special > break

list: List, kind of evaluation depends on first element, further elements are forms
(object form1 form2 ...)

function-call: Function call, standard evaluation strategy (application of an object to evaluated forms)

special-form: Special form with separate evaluation strategy
(quote form) (setf symbol value) (if cond form1 form2) (function func)
(labels functionlist form) (progn form1 form2 ...) (let (var1 var2 ...) form1 form2 ...)
(unwind-protect form1 form2) (catch tag form) (throw tag result)
(go tag) (return-from name result)

top-level-form: On top-level (empty lexical environment) evaluable form considering by compiler
(defun name lambdalist forms) (defvar name value) (defparameter name value)
(defconstant name value) (declare specifications) (defmacro name lambdalist forms)

macro-call: Macro call
Macro = Function for transformation of forms with subsequent evaluation
- Macro expansion (transformation: form → form)
- Macro evaluation (evaluation of the expanded form)

Goal: Increased Efficiency, Changed evaluation strategy

Example:
```
(push x y)   =   (setf y (cons x y))
(pop y)      =   (prog1 (car y) (setf y (cdr y)))
(defmacro pot (x y) (let (h) (dotimes (i y) (push x h)) (cons '* h))))
(defun kubik (x) (pot x 3))
```
Expansion: `>  (* x x x)`

Predefined Macros:
(loop forms) (do ((var1 init1 step1) ...) (end) forms) (dotimes (var number) forms)
(dolist (var list) forms) (prog (var1 var2 ...) forms)

Environment

Practical Systems:

Lisp machines as universal computer: not successful

- MIT, LMI : CADR; Symbolics: 3600; Xerox: Dorado, Dolphin

Common Lisp implementations with programming environments and additional tools

→ available on all hardware and software platforms:

- Allegro Common Lisp (Franz)
- LispWorks (Harlequin)
- CLisp (Steingold, Haible)

large number of implementations and applications

Programming Environment:

- Programming paradigms:	Functional, imperative, rule-, logic-, object-oriented
- Object orientation:	CLOS
- Interpreter / Compiler:	Interpretative working, partial compilation, executables
- Editor / Debugger:	File-Editor, Structure editor, Trace, Break, Step, Inspect, Documents
- Interface:	Programming language C; operating system access; user interfaces; interface builder
- Database access:	Relational systems by ODBC: ORACLE, SYBASE, ACCESS, ... Object-oriented systems: ITASCA, STATICE, ALLEGRO-STORE
- Networking:	Sockets, Internet

Applications

Fields:
- Prototyping
- AI systems, knowledge-based systems
- Dynamic flexible systems
- Teaching

Applications:
- Formula manipulation (REDUCE)
- Knowledge processing (MYCIN)
- Tools (KEE, BABYLON)
- Problem solving (MOLGEN)
- Language processing (VERBMOBIL)
- Programming environment (EMACS)
- Diagnosis (D3)
- Planning (PRODIGY)
- Logic interpreter (KRIS)
- Software technology (COSEM)
- Development environment (FAENSY)

Other basic languages:
- Prolog
- Java (Java Rules engines, Drools)

15. AI Programming Examples

Prof. Dr.-Ing. habil. Wolfgang Oertel

Goals:

- Implementation of knowledge bases and respective interpreter
- Solving of concrete knowledge-based application problems

Contents:

- Recursive Programming
- Semantic Network
- Production Rules
- General Problem Solver
- VRML Script Generation

Focus: Intelligent Applications

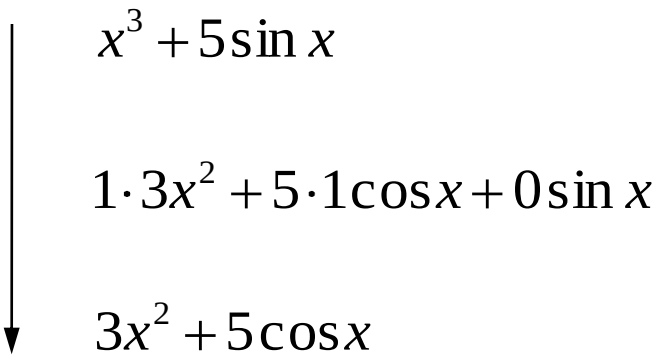
Example: Symbolic Differentiation

Problem:

```
(setf formula1 '(+ (pot x 3) (* (sin x) 5)))

(diff formula1 'x)        >   (+ (* 1 (* 3 (pot x 2)))
                                 (+ (* (* 1 (cos x)) 5) (* (sin x) 0)))

(simp (diff formula1 'x))   >   (+ (* 3 (pot x 2))
                                   (* (cos x) 5))
```



Solution:

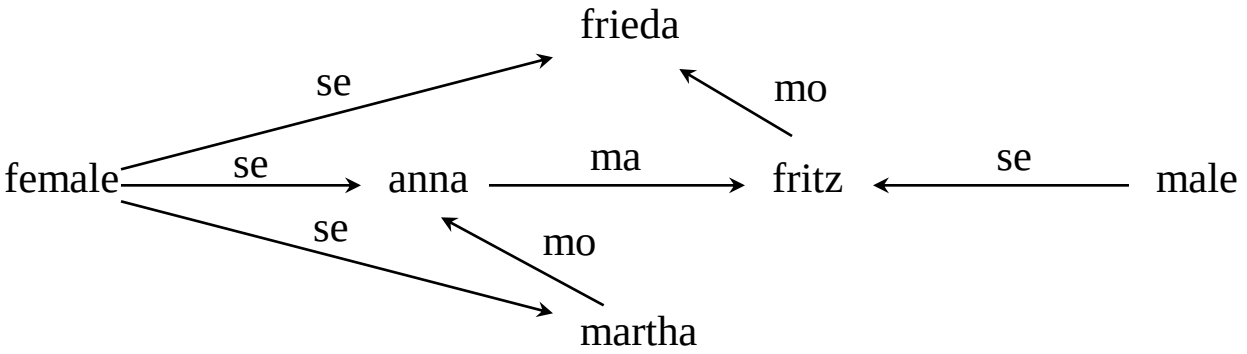
```
(defun diff (x v)
 (cond ((eq x v) 1)
  ((atom x) 0)
  ((member(car x)'(+ -)) (list(car x)(diff(cadr x)v)(diff(caddr x)v)))
  ((eq(car x)'*) (list '+(list '*(diff(cadr x)v)(caddr x))
                       (list '*(cadr x)(diff(caddr x)v))))
  ((eq(car x)'/) (list '/(list '-(list '*(diff(cadr x)v)(caddr x))
                                 (list '*(cadr x)(diff(caddr x)v)))
                       (list 'pot(caddr x)2)))
  ((eq(car x)'sin) (list '*(diff(cadr x)v)(list 'cos(cadr x))))
  ((eq(car x)'cos) (list '*(diff(cadr x)v)(list '-(list 'sin(cadr x)))))
  ((and(eq(car x)'pot)(numberp(caddr x)))
   (list '* (diff(cadr x)v)
    (list '*(caddr x)(list 'pot(cadr x)(-(caddr x)1)))))
  . . .
  (t (append '(no rule for) (list x)))))

(defun simp (x)
 (cond ((atom x) x)
  ((and(member(car x)'(+ -))(member 0 x)) (simp(car(remove 0(cdr x)))))
  ((and(member(car x)'(*))(member 1 x)) (simp(car(remove 1(cdr x)))))
  ((and(member(car x)'(*))(member 0 x)) 0)
  ((and(numberp(cadr x))(numberp(caddr x))) (eval x))
  . . .
  (t (cons(car x)(mapcar #'simp(cdr x))))))
```

Example: Semantic Network

Problem:

```
(defarc 'martha 'mother 'anna)
(defarc 'anna 'married 'fritz)
(defarc 'fritz 'mother 'frieda)
(defarc 'female 'sex 'martha)
(defarc 'female 'sex 'anna)
(defarc 'female 'sex 'frieda)
(defarc 'male 'sex 'fritz)
```




```
(mother '(martha))            >   (anna)
(father '(martha))            >   (fritz)
(parents '(martha))           >   (fritz anna)
(grandmother '(martha))       >   (frieda)
(child '(anna))               >   (martha)
(daughter '(fritz))           >   (martha)
(ancestor '(martha))          >   (frieda fritz anna)
```

Solution:

```
(defun defarc (x r y) (setf (get x r) (cons y (get x r))))
(defun delarc (x r y) (setf (get x r) (remove y (get x r))))
(defun getnod (x r) (if x (union (get (car x) r) (getnod (cdr x) r))))
(defun restr (x f) (remove-if #'(lambda (s) (null (funcall f s))) x))

(defun mother (x) (getnod x 'mother))
(defun married (x) (getnod x 'married))
(defun sex (x) (getnod x 'sex))
(defun father (x) (married (mother x)))
(defun parents (x) (union (mother x) (father x)))
(defun grandmother (x) (mother (parents x)))
(defun child (x) (restr (sex '(male female))
                  #'(lambda(s) (intersection x (parents(list s))))))
(defun daughter (x) (intersection (child x) (sex '(female))))
(defun ancestor (x) (if x (union (parents x) (ancestor (parents x)))))
```

Example: Production Rules

Problem:

```
(setf rules
      '((((it rains)) -> (wet streets))
        (((wet streets)) -> (slip danger))
        (((bad tires)) -> (slip danger))))

(setf facts '((it rains)))

(pri '((slip danger)) rules facts)
```

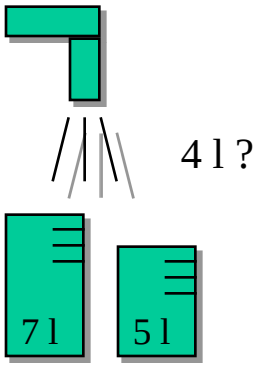
Solution:

```
(defun pri (condi ruleset factset)
  (prog (rule rules match matches trace (facts factset))
   loop0 (setf rules ruleset)
         (if (end-condition condi facts) (return (list 'solution facts)))
   loop1 (cond ((setf rule (pop rules))
                (setf matches (rule-condition rule facts)))
               ((multiple-value-setq (rule rules match matches facts)
                (apply 'values (pop trace))))
               ((return (list 'no-solution facts))))
         (unless (setf match (pop matches)) (go loop1))
         (push (list rule rules match matches facts) trace)
         (setf facts (rule-action rule match facts))
         (go loop0)))

(defun end-condition (condi facts) (subsetp condi facts :test #'equal))
(defun rule-condition (rule facts)
  (and (null (subsetp (cddr rule) facts :test #'equal))
    (subsetp (car rule) facts :test #'equal) (car rule)))
(defun rule-action (rule match facts) (cons (caddr rule) facts))
```

Example: General Problem Solver

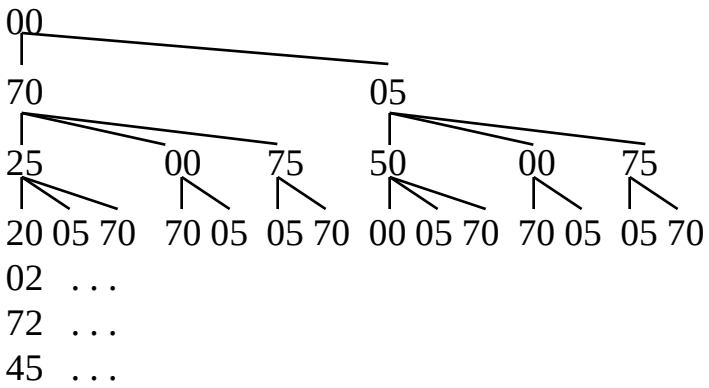
Specific Problem: Measuring an amount of liquid with two glasses



States:
z = (ContentGlas1 ContentGlas2)

Operations:
f1: Fill Glass1
f2: Fill Glass2
e1: Empty Glass1
e2: Empty Glass2
g1: Give Glass1 in Glass2
g2: Give Glass2 in Glass1

Search tree:



Problem:

```
(defun e1(state)(list 0(cadr state)))
(defun e2(state)(list(car state)0))
(defun f1(state)(list w1(cadr state)))
(defun f2(state)(list(car state)w2))
(defun g1(state)
  (list(-(car state)(min(car state)(- w2(cadr state))))
    (+(cadr state)(min(car state)(- w2(cadr state))))))
(defun g2(state)
  (list(+(car state)(min(cadr state)(- w1(car state))))
    (-(cadr state)(min(cadr state)(- w1(car state))))))
(defun c(operation)1)
(setf w1 7)
(setf w2 5)
(setf problem '((0 0)(e1 e2 f1 f2 g1 g2)(4 0)c)))
(genericsearch problem)

→   ((4 0)
      (e2(4 5)g1(7 2)f1(0 2)g1(2 0)e2(2 5)g1(7 0)f1(0 0))
      7 7)
```

Solution:

```
(defun initstate (problem) (car problem))
(defun operations (problem) (cadr problem))
(defun goalstate (problem) (caddr problem))
(defun costfunction (problem) (cadddr problem))

(defun state (node) (car node))
(defun path (node) (cadr node))
(defun depth (node) (caddr node))
(defun cost (node) (cadddr node))

(defun genericsearch (problem &aux node newnode nodes states)
   (push(list(initstate problem)nil 0 0)nodes)
   (loop (unless nodes(return)) (setf node(pop nodes))
     (when(equal(state node)(goalstate problem))(return node))
     (dolist(operation(operations problem))
       (setf newnode(list(funcall operation(state node))
                         (cons operation(cons(state node)(path node)))
                         (+(depth node)1)
                         (+(cost node)(funcall(costfunction problem)
                                              operation))))
       (unless(member(state newnode)states :test #'equal)
         (push newnode nodes)(push(state newnode)states)))))
```

VRML Script Generation

Generation of VRML prototype instances within a Lisp program

Problem and Solution:

```
(setf vrmlscript01
"#VRML V2.0 utf8
EXTERNPROTO WALL [
        exposedField SFColor material
        exposedField SFVec3f translate
        exposedField SFRotation rotate
        exposedField SFVec3f scale
        exposedField SFVec3f bbox
        field MFString name
        field MFString reference
        exposedField SFInt32 on
        ]
[\"Prototypes9a.wrl#WALL\"]
...
DEF Objects Transform {children [
")

(setf vrmlscript03
"
]}
")
```

```
(setf vrmlscript02
"
DEF Wal00 WALL {material 1 1 1 translate 500 180 15 rotate 0 1 0 0
  scale 1000 300 30
  bbox 1 1 1 name \"Wal00\" reference \"Wal00.wrl\" on 0}
DEF Wal01 WALL {material 1 1 1 translate 15 180 300 rotate 0 1 0 0
  scale 30 300 540
  bbox 1 1 1 name \"Wal01\" reference \"Wal01.wrl\" on 0}
...
")

(setf vrmlscript03
"
]}
")

(defun writevrml()
 (setf file(open "c://Test//vrml15-01.wrl" :direction :output))
 (princ vrmlscript01 file)
 (princ vrmlscript02 file)
 (princ vrmlscript03 file)
 (close file))

(writevrml)
```

The End