Introduction to AI

Philippe Laroque

UCP/ETIS/CNRS

Oct. 2008

Outline I

Outline II

- Formal systems
- Propositional calculus PC(0)
- First-order predicate calculus PC(1)
- Introduction to PROLOG
- Introduction to fuzzy logic

Outline

Brief History of AI

Goal: Analyse and mimic human behavior in a machine. Intelligence? (Turing test).

- Cybernetics (Wiener, Rosenblatt...), NN (perceptron).
- 1960, Mc Carthy & al: computer can be used to manipulate symbols (Ada Lovelace, 1842). ELIZA (Weiezbaum 1960): dialog with a psy
- 1969, Minsky/Papert: limitations of perceptron: NN frozen
- 1978, Newell & Simon: the GPS
- 1982, 5th-generation computers (Japan). Goal: parallel thinking machine by '92

Languages

- 1958, LISP (J. McCarthy, MIT): program from data
- 1973, PROLOG (A. Colmerauer): inference engines and expert systems generators
- 80s: production rules, frame languages, script languages, logical programming (PLANNER: goal generation for problem solving)

Applications

- Computer-aided programming, diagnosis (MYCIN 76), design (R1 83), planning, education (LOGO)...
- Problem solving (DENDRAL 71, organic chemistry) (AM 79, mathematical concepts discovery)
- Games (chess, poker, bridge...)
- Simulation (qualitative physics)
- ...

Outline

Problems

- "Common sense" modelling?
- On real-sized applications: incompleteness of expertise, errors in rules, inconsistency within rule sets...
- Learning
- Combinatory explosion (NP-complete problems): ex. Chess
  - $\simeq 40$ legal config each turn
  - 7 turns: $40^7 = 163,840,000,000$
  - if 100000 config/s, since epoch: $100000 \times 3600 \times 24 \times 366 \times 4.6 \times 10^9 \simeq 40^{14}$, 7 turns for both players!

Intelligence and Knowledge

Intelligence needs knowledge, which is by nature

- huge
- hard to define precisely
- subject to change in time

Intelligence and Knowledge

Intelligence needs knowledge, which is by nature

- huge
- hard to define precisely
- subject to change in time

Conclusion:

Need for a representation model of knowledge

Knowledge Representation

Desired features for knowledge representation:

- general (apply in most cases)
- understandable (by people who need it)
- easily maintenable
- can serve as a tool to improve knowledge about knowledge

Most common techniques:

- state space
- formal systems, as proposition calculus ($PC(0)$) and predicate calculus ($PC(1)$)

Outline

Basic Notions

- State: symbolic description of manipulated objects and their properties, and relations between these objects at a given time. Common data structures: lists, arrays, graphs, databases...
- Goal: the state of the system when the problem is solved
- Operator: make state change. Describe atomic actions needed to switch from state A to state B. Defined by its application domain. Common representations: functions, rewriting rules, algorithms...

Basic Notions

- State: symbolic description of manipulated objects and their properties, and relations between these objects at a given time. Common data structures: lists, arrays, graphs, databases...
- Goal: the state of the system when the problem is solved
- Operator: make state change. Describe atomic actions needed to switch from state A to state B. Defined by its application domain. Common representations: functions, rewriting rules, algorithms...

Problem solving

By applying *rules* that use operators, we start from an inital state to the goal. Rules are *fired* following a given *strategy*: it's a *production system*

Outline

Production system

- Set of *production rules*. Left part defines the conditions for applying the rule, right part describes actions to run if rule is fired.
- *Data* (or *facts*) *base*. Contains informations needed to activate the actions. Dynamic structure: applying rules can add information to the base.
- *Command strategy*: defines how rules are fired according to the base contents.

Command strategies

- Contain criteria to choose rules in order to be as efficient as possible.
- Induce state changes: need for exhaustivity, but risk of *combinatory explosion*.
- Heuristic functions can help avoid C.E. Good heuristic functions demand good knowledge of the problem: no general rule to find them.

Problem Analysis

- Is the problem breakable into easier sub-problems?
- Can some states be ignored/removed if search fails?
- Must we find a “good” solution or the “best” solution?
- Is the base coherent? Do we need all of the base all the time?
- May the user help the computer find the solution?

State space enumeration

State spaces can be represented by a directed graph where states are vertices (nodes) and operators are edges.
Problem solving = find a path from initial state to goal state.
Actual building of the complete graph is seldom necessary: implicitely defined by production rules
Important aspects:

- search direction
- order of enumeration
- state representation
- candidate rule selection
- heuristic function definition

Direction of search

- *forward* search: starts from initial state. Fired rules have a left part compatible with current state of the problem. Right parts provide new states.
- *backward* search (or *backward chaining*): starts from goal state. Fired rules have a right part compatible with current state of the problem. Left parts provide new states.

Direction choice criteria

Rules of thumb:

1. If there are n initial states and m goal states, choose direction towards $max(n, m)$
2. From current state, choose direction with the lowest *branching factor*
3. If system is interactive, choose direction that fits best user reasoning mode

Order of enumeration

- *Breadth* first: nodes are visited in the order in which they are created. From current state, possible candidates are states created by applying rules; they are placed in a queue (FIFO).
- *Depth* first: nodes are visited in the reverse order of their creation; they are placed in a stack (LIFO).

Outline

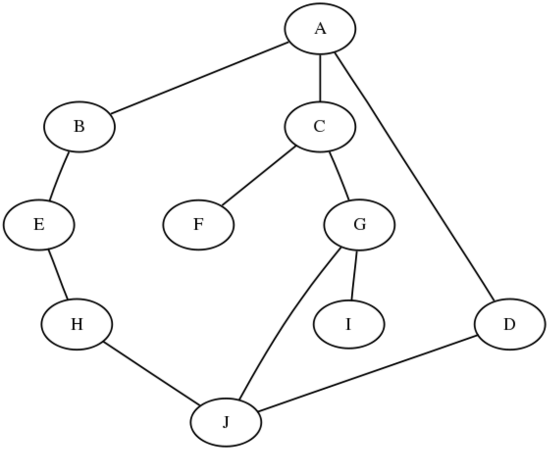
Enumeration algorithms

- Two lists are used: OPEN and CLOSED
    - OPEN contains known nodes waiting to be visited
    - CLOSED contains already visited nodes

Breadth-first algorithm

1. Place initial node n_0 in OPEN
2. if OPEN is empty, stop (failure)
3. get and remove first element of OPEN, call it n and add it to CLOSED
4. if no successor to n, go to 2
5. append every successor s_i to the end of OPEN if it is not already in OPEN or CLOSED (and initialize backpath pointer $s_i \longrightarrow n$)
6. if one of the successors is the goal, stop (success): use pointer chain to retrieve the solution path.
7. go to 2

Example graph



Notion of depth

- For a tree: distance from root node
- For a graph, recursive definition: depth of nearest ancestor + 1 (update necessary during traversal)

Depth-first algorithm

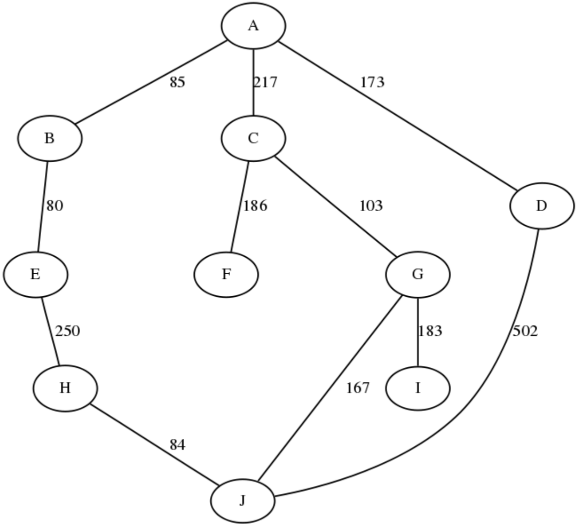
1. Place initial node n_0 in OPEN
2. if OPEN is empty, stop (failure)
3. get and remove first element of OPEN, call it n and add it to head of CLOSED (update depths if necessary)
4. if current depth exceeds max depth, go to 2
5. if no successor to n, go to 2
6. add every successor to the top of OPEN if not already in CLOSED (update depths if necessary and initialize/set associated pointers to n)
7. if one of the successors is the goal, stop (success): use pointer chain to retrieve the solution path.
8. for vertices already in CLOSED, recompute depth of successors
9. go to 2

Path criteria

- state switching involves certain operations: each edge has an associated cost
- Order of the visits can be determined to minimize global cost of the solution
- depth-first: the successors of current vertex are sorted on this criterion
- breadth-first: the wole set of nodes waiting to be visited is sorted that way (ex. Dijkstra)

UNIVERSITÉ de Cergy-Pontoise

Example graph (2)



Dijkstra

- No need for CLOSED, since nodes are only visited once
- Principle:
    - OPEN initially contains all nodes
    - A distance from the source node is maintained
    - Each time a node is visited, that distance may be updated
- This algorithm gives the shortest path under the condition that no weight is negative

Algorithm

```
OPEN <- all nodes
for each node $i$ set $d(i) = \infty$ except $d(n_0) = 0$
while OPEN not empty
  1. n <- remove-min(OPEN)
  2. for each successor $s_i$ of n
     1. if $d(s_i) > d(n) + w(n, s_i)$ then
        1. $d(s_i)$ <- $d(n) + w(n, s_i)$
        2. update backpath pointer associated with $s_i$: $s_i \longrightarrow n$
```

Notes on Dijkstra

- Simple optimization: stop when $n = goal$
- Performance:
  - Using adjacency matrices, $O(V^2 + E)$
  - For sparse graphs, using adjacency lists and a heap for OPEN: $O\left((V + E)\, log(V)\right)$
  - using a Fibonacci heap: $O\left(E + V.log(V)\right)$
- Ford-Bellman can be used when some edges have a negative weight but worse performance $O(EV)$
- Sometimes, one only needs a "good" solution (not the best), but faster: need for an evaluation function

Evaluation function

two parts: $f(n) = g(n) + h(n)$

- $g(n)$ represents the cost of the path from initial state to current state n
- $h(n)$ represents the cost of the path from current state to goal state
- from now on, the above formula stands for the cost of the *optimal* path P containing n

optimal path property

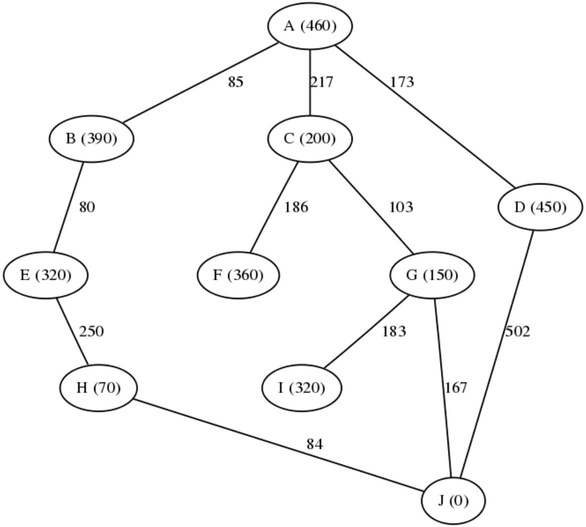
$\forall n \in P, f(n) = f(n_0)$

Estimation functions

Since we ignore if n is on the optimal path, we estimate the evaluation function: $\widehat{f}(n) = \widehat{g}(n) + \hat{h}(n)$

- $\hat{g}(n)$ represents the min. cost from n_0 to *n at the time n is visited* (can only be $\geq$ final value of $g(n)$)
- $\hat{h}(n)$ estimates the cost from n to the goal assuming n is on the optimal path.

Example graph (3)



A*

1. Place n_0 in OPEN. compute $\hat{h}(n_0)$ and set $\hat{g}(n_0) = 0$. All other $\hat{g} = \infty$
2. if OPEN is empty, stop (failure)
3. remove from OPEN the vertex with minimal $\hat{f}$, call it n and add it to CLOSED
4. if n is the goal, stop (success): use pointer chain to retrieve the solution path.
5. For each successor s_i of n:
    1. compute $\hat{g}(n) + c(n, s_i)$
    2. if s_i is in OPEN or in CLOSED and $\hat{g}(n) + c(n, s_i) > \hat{g}(s_i)$, skip to next successor
    3. remove s_i from OPEN and CLOSED if present
    4. insert s_i in OPEN and update $g(\hat{s_i})$ and backpath pointer
6. go to 2

Admissibility of A*

Definition

An algorithm *a* is *admissible* if, for every graph representing a possible problem, *a* finds the optimal path

Admissibility of A*

Definition

An algorithm *a* is *admissible* if, for every graph representing a possible problem, *a* finds the optimal path

Theorem

A is admissible if* $\forall n, \hat{h}(n) \leq h(n)$ *and*
$(\exists \delta > 0, \forall n, s_i), c(n, s_i) > \delta$

Several notes about A*

- It is possible to set $g(n)$ to 0 systematically. We choose then each time the vertex that minimizes $\hat{h}$ in OPEN (or in the successors of n – “hill climbing” strategy)
- Concerning function h:
    - if $h(n) = 0$, search is guided by g
        - if g is null too, the search is random
        - if $g(n) = 1$, (id. depth) the search is breadth-first

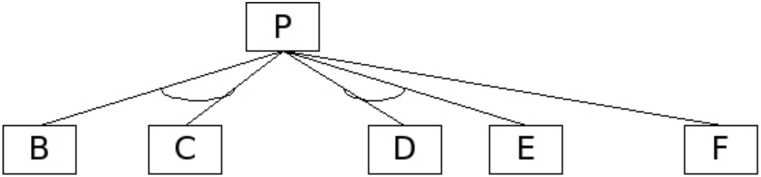
Solving problems by decomposition

The idea is to repeatedly break a problem into easier-to-solve subproblems, until each subproblem is trivial.

Outline

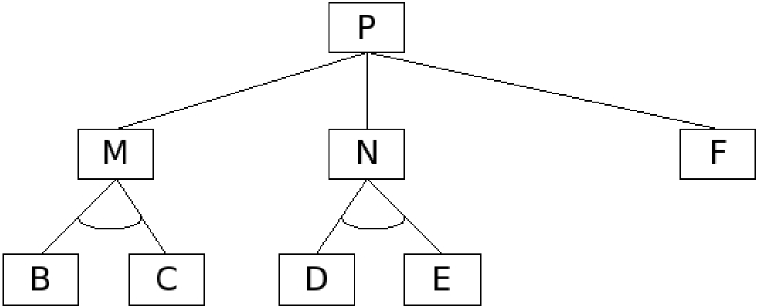
AND-OR Trees

- Previous trees and graphs can be viewed as OR graphs: the algorithm stops as soon as only one solution path is needed
- AND-OR graphs and trees are suitable to search solutions to breakable problems, such as: “to solve P, one has to solve B and C, or D and E, or F”



Standard representation

At a given level, there are only “OR” nodes or “AND” nodes: add intermediate nodes



Node types

- "OR" nodes: solved if one of the children is solved
- "AND" nodes: solved if all children are solved
- Initial node (root) correponds to the formulation of the problem
- Terminal nodes are solved problems, non-terminal nodes without successors are unsolved problems

Production rule analogy

Problem decomposition can be represented with a rule of the form

$$Q \rightarrow A, B, C$$

which means “to solve Q, one must solve A, B and C”
A set of such rules is called a *rule base*. Initially solved problems form the *facts base*.

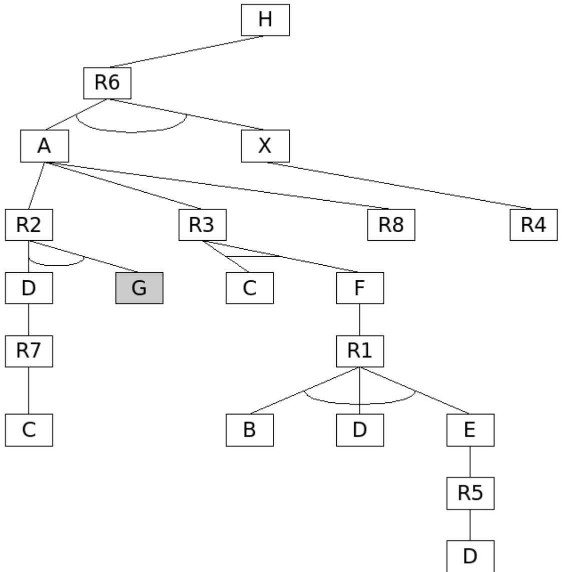
Example

Rules base:

$R_1 : F \longrightarrow B, D, E$
$R_2 : A \longrightarrow D, G$
$R_3 : A \longrightarrow C, F$
$R_4 : X \longrightarrow B$
$R_5 : E \longrightarrow D$
$R_6 : H \longrightarrow A, X$
$R_7 : D \longrightarrow C$
$R_8 : A \longrightarrow X, C$
$R_9 : D \longrightarrow X, B$

Facts base: $\{B, C\}$
Problem to solve: H

Corresponding AND-OR tree



Cost of a solution tree

As for classical “OR” trees, it is possible to use an evaluation function $h(n)$ to estimate the cost of a solution tree rooted at current node:

1. if n is terminal, $h(n) = 0$
2. if n is a non-terminal “OR”, $h(n) = min_{i=1..k}\{c(n, s_i) + h(s_i)\}$
3. if n is a non-terminal “AND”, $h(n) = \Sigma_{i=1..k}\{c(n, s_i) + h(s_i)\}$
4. if n is unsolved, $h(n)$ is undefined

Estimation of the evaluation function

During search phase, h cannot be computed, only estimated (using $\hat{h}$).
At each step of the search tree building phase, *extrema* vertices fall into four categories:

1. terminals: $\hat{h}(n) = 0$
2. non-terminals whose successors have not yet been visited: $\hat{h}(n)$ is an estimation of the solution tree rooted at n.
3. non-terminals whose successors have been visited:
    1. if n is an "OR" node: $\hat{h}(n) = min_{i=1..k}\{c(n, s_i) + \hat{h}(s_i)\}$
    2. if n is an "AND" node: $\hat{h}(n) = \Sigma_{i=1..k}\{c(n, s_i) + \hat{h}(s_i)\}$

Game algorithms

Good application domain for AI:

- They use a strategy whose accuracy can be easily measured.
- They demand some domain-specific knowledge to define heuristics leading to winning configurations.

In complex games, CE must definitely be avoided. To do so, one must have:

1. A *procedure to generate good movements* in search space, which must select the most “promising” moves.
2. A *static evaluation function* which measures the quality of a given configuration.

1- and 2- player games

- 1-player games can use A* algorithm
- 2-player games often need an AND-OR graph-like structure:

graph	game
vertex, problem state	game state
terminal node, solved problem	winning configuration
extremum vertex, unsolved problem	loosing configuration
OR vertex	I's turn to play
AND vertex	HE's turn to play

Simultaneous moves

- In the case of zero-sum games: choose by solving a set of equations (J. von Neumann, 1928):

	B1	B2	B3
A1	+3	-2	+2
A2	-1	0	+4
A3	-4	-3	+1

(same - negated payoff matrix for player B)

- Read: "if A plays 1 and then B plays 1 too, then A wins 3 (and B looses 3)"
- Simple choice: A2 (worst case costs 1) and B2 (0 cost)
- But A2 $\rightarrow$ B1 $\rightarrow$ A1 $\rightarrow$ B2: unstable!
- By solving a set of equations, the system can be made stable

Example of stabilization

A's point of view: A3 will never be chosen because always worse than A2.
B's point of view: B3 will never be chosen because always worse than both B1 and B2.
A: Call $p =_{def} p(A_1)$, then

- If B plays B1 we get $3p - (1 - p) = 4p - 1$
- If B plays B2 we get $-2p$
- Hence $-2p = 4p - 1 \Rightarrow p = \frac{1}{6}$, and cost is $\frac{1}{3}$

B: Call $p =_{def} p(B_1)$, then

- If A plays A1 we get $-3p + 2(1 - p) = -5p + 2$
- If A plays A2 we get p
- Hence $p = -5p + 2 \Rightarrow p = \frac{1}{3}$, and gain is $\frac{1}{3}$

Outline

MinMax Algorithm

- Game tree developped to depth d: leaves are evaluated using static evaluation function.
- The algorithm tries to make I (usually, the computer) win: I tries to maximise the evaluation function, HE tries to minimize it (hence the name, MinMax)
- Goal: anticipate several turns and evaluate best turn according to d.
- OR vertices are associated with MAX, AND vertex with MIN

Search space size

- Exhaustive exploration of search space is not realistic
- Example: connect4:
  - branching factor = 7
  - max depth = 42
  - # configs $= 7^{42} \simeq 3.10^{35}$
  - assuming 10^8 configs visited per second:
    $10^{27} s \simeq 3.10^{23} h \simeq 10^{22} days \simeq 3.10^{19} years$
- Need to stop exploration at given depth d

Principle of MinMax

1. Build search tree to depth d
2. Compute evaluation function $v(n)$ on leaves
3. Bottom-up-compute values $V(n)$ for internal nodes using following rule:
   - $V(n) = v(n)$ if n is an extremum
   - $V(n) = \max_i\{V(s_i)\}$ if n is a MAX vertex
   - $V(n) = \min_i\{V(s_i)\}$if n is a MIN vertex

Example of evaluation function

For connect4:

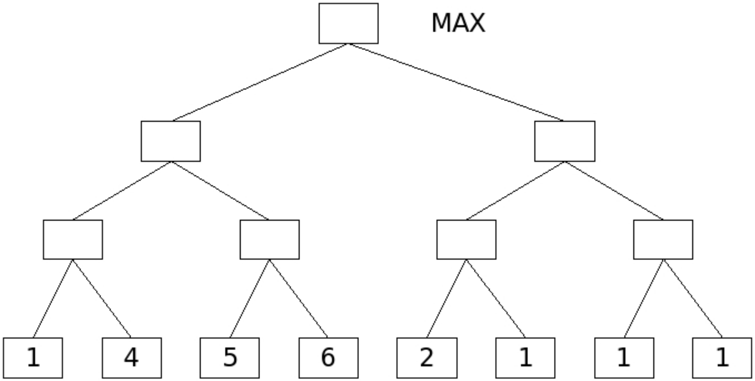
- let n_1 be the number of “potential ones” (a token and 3 spaces in a row)
- let n_2 be the number of “potential twos” and n_3 the number of “potential threes”.
- Since potential threes are of much greater value than potential ones, give them higher weights, for instance $f(conf, player) = n_1 + 5n_2 + 50n_3$
- Then $v(conf)$ can be defined as follows:

$$v(conf) = f(conf, I) - f(conf, HE)$$

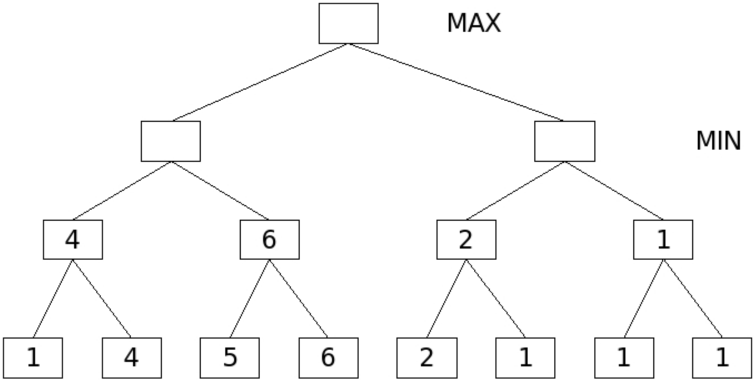
MinMax algorithm

```
function minimax(node, depth)
  if node is a terminal node or depth = 0
    return $v(node)$
  if the adversary is to play at node
    let $\alpha$ := MAXVAL //+ infinity
    foreach child of node
      $\alpha$ := min($\alpha$, minimax(child, depth-1))
  else {we are to play at node}
    let $\alpha$ := -MAXVAL
    foreach child of node
      $\alpha$ := max($\alpha$, minimax(child, depth-1))
  return $\alpha$
```

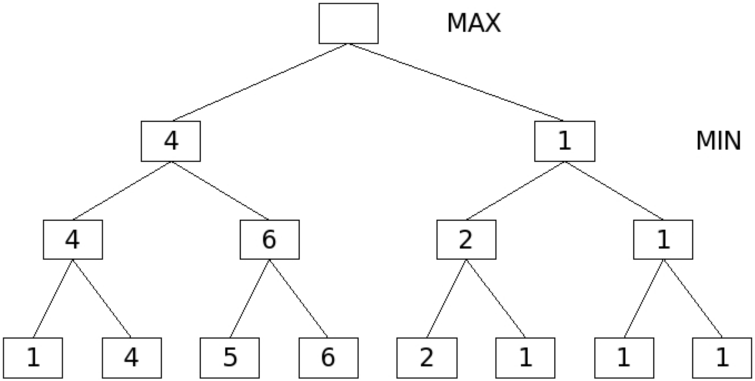
A Simple Example



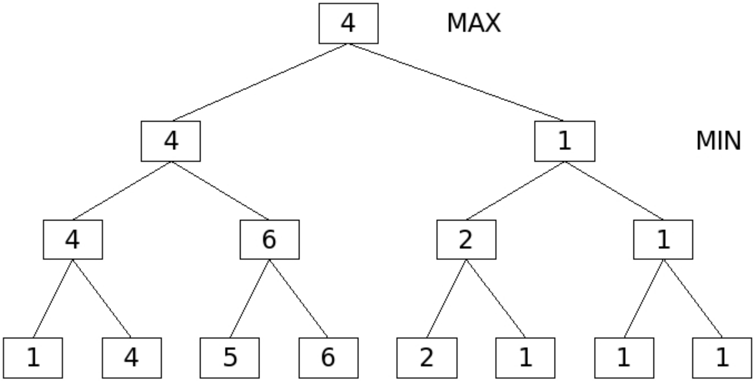
A Simple Example



A simple Example



A simple Example

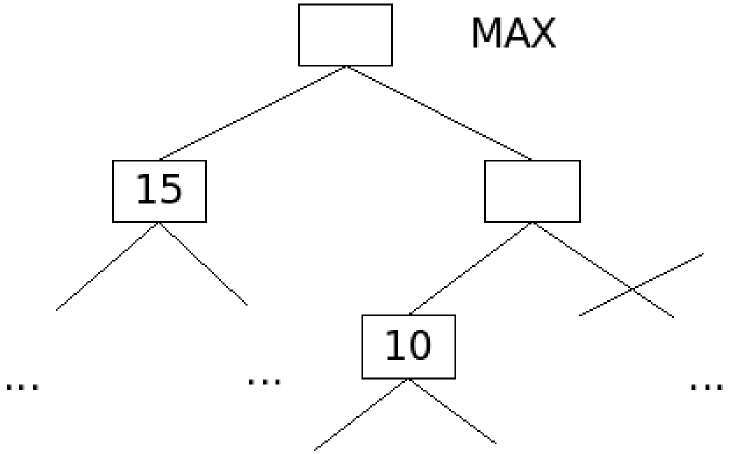


Outline

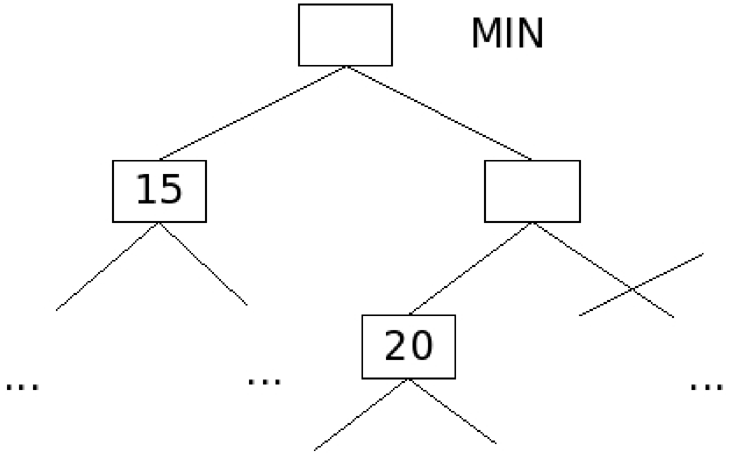
Alpha-Beta: improvement to MinMax

- MinMax visits all of the search tree, which is not always necessary
- Alpha-Beta detects the possibility of cut-offs in the tree
- Two more variables:
    - α represents the minimum value MAX is sure to reach
    - β represents the maximum value MAX can hope to reach

Alpha cutoff (MIN)



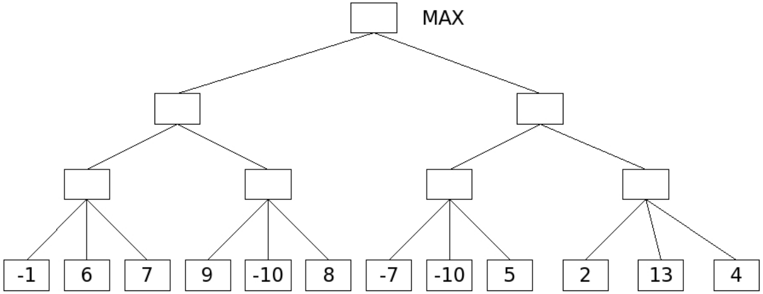
Beta cutoff (MAX)



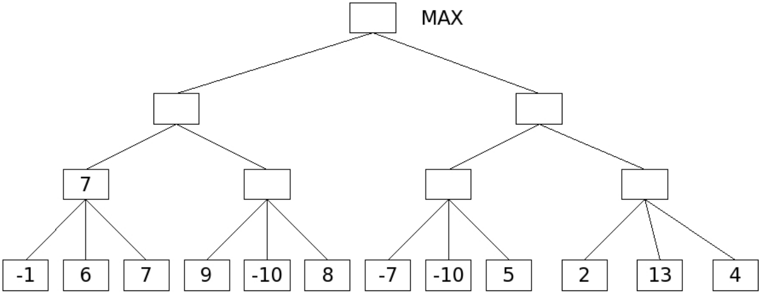
Alpha-Beta algorithm

```
alphaBeta (n,d,$\alpha$,$\beta$) { //$\alpha = -\infty, \beta = +\infty$
  if (d = 0) return v(n)
  if 'HE' plays {
    for each child $c_i$ of n {
      val = alphaBeta($c_i$,d-1,$\alpha$,$\beta$)
      if (val < $\beta$) $\beta$ = val
      if ($\alpha$>=$\beta$) break
    }
    return $\beta$
  } else { // 'I' plays
    for each child $c_i$ of n {
      val = alphaBeta($c_i$,d-1,$\alpha$,$\beta$)
      if (val > $\alpha$) $\alpha$ = val
      if ($\alpha$>=$\beta$) break
    }
    return $\alpha$
  }
}
```

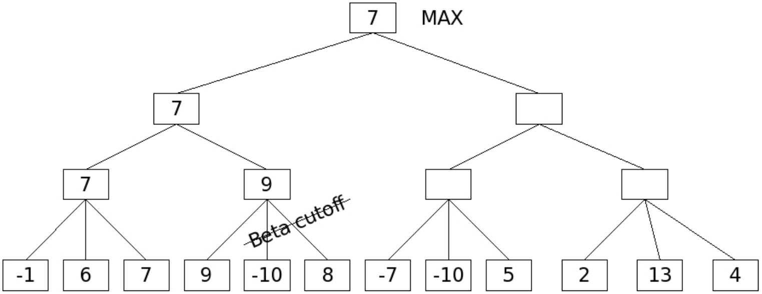
Example



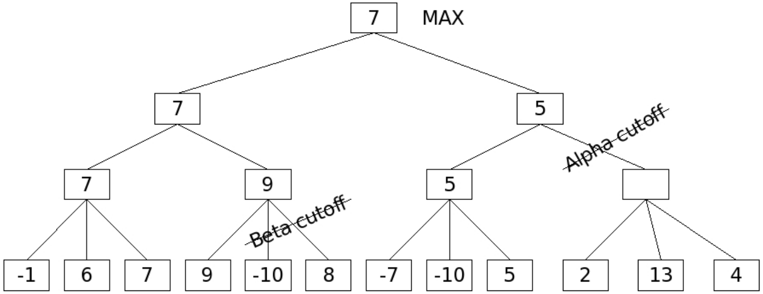
Example



Example



Example



Expectable benefit of alpha-beta

- algorithm is heavily dependent upon the order in which moves are searched.
- If program always manages to pick best move first, effective branching factor is equal to approximately the square root of the expected branching factor (best possible case)
- massive improvement: allows to search *twice as deeply* in the same number of nodes:

$$\sqrt{n}^h = \left(n^{\frac{1}{2}}\right)^h = n^{\frac{h}{2}}$$

Example of connect4

- assuming a depth of 12 (6 turns for each player)
- number of configs to examine with minmax: $7^{12} \simeq 14$billions
- if 10^8 config visited per second: 2 minutes!
- number of config to examine with alpha-beta (opt.): $7^6 = 117649$, which takes approx. 1ms!

Outline

Introduction to Expert Systems

- Human-like reasoning, if limited knowledge domain
- As humans, able to explain their conclusions
- ES building in two phases:
  1. Analysis: understand the underlying domain-specific knowledge mechanisms.
  2. Synthesis: program a machine to behave like a domain expert
     1. Structuring level
     2. Conceptual level
     3. Cognitive level

Structuring level

Goal: model expert method using AI techniques. Need to evaluate complexity, which roughly falls into three classes:

1. *diagnosis* systems: classify a situation using (constant) descriptor(s) $\longrightarrow$ propositional calculus
2. *problem solving* systems: input is parameterized by variables. Find a series of legal transformations to find correct values $\longrightarrow$ 1st-order predicate calculus
3. *planning* systems: try to optimally execute a set of tasks subject to a set of constraints. Most complex class, because
    1. constraint optimization
    2. context dynamically evolves

Conceptual level

- Defines the semantics of the language to express knowledge (structuring level defines syntax).
- Describes descriptors and predicates with which laws, states and operators modelling knowledge will be defined.

Cognitive level

- Uses the language defined in previous levels to represent knowledge of the expert

Outline

Components of an ES

- Knowledge base: domain-specific. Describes manipulated concepts, their relations, resolution strategies, particular cases. Some uncertain knowledge can be probabilistically defined
- Fact base: current situation of the system. Can contain proven facts or facts to prove (goals).
- Inference engine: process which solves the problem specified by input facts of the FB, using knowledge contained in the KB.

Operating mode of an IE

- Most of the time, KB contains *production rules.*
- Each rule contains a *condition* part and a *body* (describes the effects of firing the rule).
- IE runs several *evaluation-execution* cycles.
  - evaluation phase determines candidate rules after current state of FB;
  - execution phase updates FB after firing the rule.
- IE stops if no candidate rule in evaluation phase (or in execution phase, on an explicit *stop* statement)

Contents of evaluation phase

- *restriction*: according to current state of problem, select a subset of FB and a subset of KB (optional).
- *pattern-matching*: condition part of rules of KB are compared to facts of FB (systematic).
- *conflict resolution*: determines actual subset of rules that will be fired (optional; for instance, if two rules have the same “condition” part and lead to contradictory “body” parts: can rely on a *measure of confidence* in rules to choose which rule to fire)

Contents of execution phase

IE executes body part of selected rules. When the rule set is empty,

1. either IE stops (in simple cases)
2. or IE defines a new subset by reconsidering the set of rules elaborated during pattern matching

Performance of an ES

- Solving a problem involves chaining several cycles, called *inference cycles.*
- The number of inference cycles per time unit (LIPS: Logical Inferences Per Second) is one of the performance indicators for 5th generation computers.
- IE can work using *forward* and/or *backward* chaining

Forward chaining

- IE starts from proven facts to find the solution
- When condition (left) part of a rule is is FB, its right part is added to FB (which thus only contains proven facts)

Backward chaining

- IE starts form goal and finds needed facts to prove it (AND-OR tree)
- Matching operates on right parts of the rules: when right part of a rule is in FB, its left part is added to FB
- Initial problem is solved when every problem it depends on is solved (*i.e.* is in FB)

NB: some IE use *mixed chaining*, according to context.

Monotonous mode

- An SE runs in monotonous mode if
  1. no knowledge (rule or proven fact) can be removed;
  2. new knowledge never induces contradiction
- Most PC(0) and PC(1) systems are monotonous
- In non-monotonous systems, firing a rule can modify KB (and even RB)

Brief History

- 4th c. BC: Aristote (variable, quantization, terms), stoïcists (Modus ponens $p \wedge (p \Rightarrow q) \vdash q$, modus tollens $\neg q \wedge (p \Rightarrow q) \vdash \neg p$):
- 13th c.: Scholastic logic (G. d'Occam - *Entia non sunt multiplicanda praeter necessitatem*[1], ...)
- 15th c.: stagnation (exception: Leibniz, thinking=calculus on signs: step from *thought* $\rightarrow$ *speech* $\rightarrow$ *writing* to *writing* $\rightarrow$ *thought*)
- 1850: Boole, de Morgan
- 20th c.: Peano (axiomatization), Russel (*Principia Mathematica*), Hilbert (problem of the non-contradiction of mathematics), Gödel (1931: incompleteness)...

[1] Occam's razor: do not use new hypotheses as existing ones are sufficient (*principle of economy*)

Outline

Definition of a FS

a FS S is made of

- An alphabet Σ, finite or countable, numbered
- A recursive[2] subset $F \subseteq \Sigma*$ called the *well-formed formulas* of S.
- A recursive subset $A \subseteq F$ called the *axioms* of S.
- A finite set R of decidable predicates defined on F, called the *inference rules* of S.

Notation: $f_1, ..., f_n \vdash^r g$ rather than $r(f_1, ..., f_n, g)$

[2] One can build a program that, given a formula f, says wether f is well-formed or not

Example

- $\Sigma = \{1, +, =\}$
- $F = \{1^+ + 1^+ = 1^+\}$
- $A = \{1 + 1 = 11\}$
- $R = \left\{ \begin{array}{lll} 1^n + 1^m = 1^p & \vdash^{r_1} & 1^{n+1} + 1^m = 1^{p+1} \\ 1^n + 1^m = 1^p & \vdash^{r_2} & 1^n + 1^{m+1} = 1^{p+1} \end{array} \right.$

Deduction and theorem

Let $(h_i) \subseteq F$ be a set of formulas called *hypotheses*

Definition

a *Deduction* from (h_i) is a finite family (f_i) in F such as

- $f_i \in A$, or
- $f_i \in (h_i)$, or
- $\exists (f_j) \subseteq (f_i), \exists r_k \in R / (f_j) \vdash^{r_k} f_i$

A *Theorem* is a deduction from $\emptyset$. The set of S's theorems is called T_S

A simple analogy: chess

- Σ: chessboard and figures
- F: configurations of figures on chessboard
- A: initial configuration
- T: allowed configurations
- R: rules of chess game

Theorem and truth

Attention

No necessary coincidence between definition of the theorems and the interpretation human mind makes from formulas

- What is known to be true can not be a theorem (*completeness* problem)
- A theorem can not reflect a reality of our interpretation (*consistency* problem)

Ex: $1+1+1=111$ is “intuitively” true, but is no theorem (nor a wff)

Properties of a FS

Definition

- A FS is Coherent if $T \neq F$
- A FS is Decidable if T is recursive
- A FS is Consistent if there is no wff $f \in F / f \in T, \neg f \in T$

Gödel's theorem: one cannot prove the consistency of a "complex"[3] FS, but with tools more powerful than the FS itself. Moreover, if a FS is consistent and its theorems are all "true", then there exist arithmetical formulas that are true and are not theorems of the FS.
There are FS in which T is not recursive (automatic proof pb)

[3] including arithmetics

Exercise

Let S be defined with

- $\Sigma = \{A, B, C\}$
- $F = \{A_n B C_m (n, m \geq 0)\}$
- $A = \{A_{2i} B C_{2i}, (i \geq 0)\}$
- $R =$
$$\left\{ \left\{ \begin{array}{c} A_n B C_m \\ A_{n'} B C_{m'} \end{array} \right\} \vdash A_{n+n'} B C_m \right\}$$

Exercise

Let S be defined with

- $\Sigma = \{A, B, C\}$
- $F = \{A_n BC_m (n, m \geq 0)\}$
- $A = \{A_{2i} BC_{2i}, (i \geq 0)\}$
- $R = \left\{ \left\{ \begin{array}{c} A_n BC_m \\ A_{n'} BC_{m'} \end{array} \right\} \vdash A_{n+n'} BC_m \right\}$

1. Prove that $A_6 BC_2$and $A_{10} B$ are theorems of S
2. Characterize T and show that any theorem can be derived in at most 3 steps
3. Show that if any axiom is removed, T is changed
4. Give a FS with same Σ, F, T, only 1 axiom and 2 inference rules

Solution

Let's call $F_{n,m}$ the formula A_nBC_m. Axioms are thus $A = \{A_i = F_{2i,2i}\}$

1. $A_1, A_2 \vdash F_{6,2}$ and $A_0, A_{10} \vdash F_{10,0}$
2. $T = \{T_{i,j} = F_{2i,2j}, 0 \leq j \leq i\}$. Steps: 1/ A_i, 2/ A_{i-j}, 3/ $A_j, A_{i-j} \vdash^r T_{i,j}$
   reciprocal: let $F_{i,j} \in T$ then $i < j$ is impossible: we would have $F_{i-j,0} \in T$ and $i - j < 0$, which is not in F. So $i \geq j$. Then $F_{i-j,0} \in T$ (appl. of R), and thus $(i - j)$ and j are even: OK
3. $A' = A - \{A_i = T_{i,i}\}$: $T_{i,i}$ cannot be proven any more
4. $A = \{A_0\}$, $r_1 = r$, $A_nBC_n \vdash^{r_2} A_{n+2}BC_{n+2}$

Outline

PC(0)

- $\Sigma(0) = \{P_1, ..., P_n, ...\} \cup \{\neg, \wedge, \vee, \Rightarrow, \Leftrightarrow\} \cup \{T, F\}$[4]
- $F(0) =$<WFF>=<P> | ¬<WFF> | (<WFF><BCon><WFF>)
- $A(0) =$
$$\begin{cases} A_1 & (p \Rightarrow (q \Rightarrow p)) \\ A_2 & ((p \Rightarrow (q \Rightarrow r)) \Rightarrow ((p \Rightarrow q) \Rightarrow (p \Rightarrow r))) \\ A_3 & ((\neg p \Rightarrow \neg q) \Rightarrow (q \Rightarrow p)) \end{cases}$$
- $R(0) = MP$

[4] ¬and ⇒are sufficient

Exercises

- Show that $\neg$ and $\Rightarrow$ are sufficient
- Model $(p?q : r)$
- Proove $(p \Rightarrow p)$

Exercises

- Show that $\neg$ and $\Rightarrow$ are sufficient
- Model $(p?q : r)$
- Proove $(p \Rightarrow p)$

$p \vee q : (\neg p \Rightarrow q)$,$p \wedge q : \neg(p \Rightarrow \neg q)$,$(p \Leftrightarrow q) : \neg((p \Rightarrow q) \Rightarrow \neg(q \Rightarrow p))$

Exercises

- Show that $\neg$ and $\Rightarrow$ are sufficient
- Model $(p?q:r)$
- Proove $(p \Rightarrow p)$

$p \vee q : (\neg p \Rightarrow q), p \wedge q : \neg(p \Rightarrow \neg q), (p \Leftrightarrow q) :$
$\neg((p \Rightarrow q) \Rightarrow \neg(q \Rightarrow p))$
$((p \Rightarrow q) \wedge (\neg p \Rightarrow r))$

Exercises

- Show that $\neg$ and $\Rightarrow$ are sufficient
- Model $(p?q : r)$
- Proove $(p \Rightarrow p)$

$p \vee q : (\neg p \Rightarrow q)$,$p \wedge q : \neg(p \Rightarrow \neg q)$,$(p \Leftrightarrow q) :$
$\neg((p \Rightarrow q) \Rightarrow \neg(q \Rightarrow p))$
$((p \Rightarrow q) \wedge (\neg p \Rightarrow r))$
$A_1 \quad (p \Rightarrow ((p \Rightarrow p) \Rightarrow p)) \quad q : (p \Rightarrow p)$

Exercises

- Show that $\neg$ and $\Rightarrow$ are sufficient
- Model $(p?q : r)$
- Proove $(p \Rightarrow p)$

$p \vee q : (\neg p \Rightarrow q), p \wedge q : \neg(p \Rightarrow \neg q), (p \Leftrightarrow q) :$
$\neg((p \Rightarrow q) \Rightarrow \neg(q \Rightarrow p))$
$((p \Rightarrow q) \wedge (\neg p \Rightarrow r))$

$A_1 \quad (p \Rightarrow ((p \Rightarrow p) \Rightarrow p)) \quad q : (p \Rightarrow p)$

$A_2 \quad ((p \Rightarrow ((p \Rightarrow p) \Rightarrow p)) \Rightarrow ((p \Rightarrow (p \Rightarrow p)) \Rightarrow (p \Rightarrow p))) \quad q$

Exercises

- Show that $\neg$ and $\Rightarrow$ are sufficient
- Model $(p?q : r)$
- Proove $(p \Rightarrow p)$

$p \vee q : (\neg p \Rightarrow q), p \wedge q : \neg(p \Rightarrow \neg q), (p \Leftrightarrow q) :$
$\neg((p \Rightarrow q) \Rightarrow \neg(q \Rightarrow p))$
$((p \Rightarrow q) \wedge (\neg p \Rightarrow r))$

$A_1 \quad (p \Rightarrow ((p \Rightarrow p) \Rightarrow p)) \quad q : (p \Rightarrow p)$
$A_2 \quad ((p \Rightarrow ((p \Rightarrow p) \Rightarrow p)) \Rightarrow ((p \Rightarrow (p \Rightarrow p)) \Rightarrow (p \Rightarrow p))) \quad q$
$MP \quad ((p \Rightarrow (p \Rightarrow p)) \Rightarrow (p \Rightarrow p))$

Exercises

- Show that $\neg$ and $\Rightarrow$ are sufficient
- Model $(p?q : r)$
- Proove $(p \Rightarrow p)$

$p \vee q : (\neg p \Rightarrow q), p \wedge q : \neg(p \Rightarrow \neg q), (p \Leftrightarrow q) :$
$\neg((p \Rightarrow q) \Rightarrow \neg(q \Rightarrow p))$
$((p \Rightarrow q) \wedge (\neg p \Rightarrow r))$

$$\begin{array}{lll} A_1 & (p \Rightarrow ((p \Rightarrow p) \Rightarrow p)) & q : (p \Rightarrow p) \\ A_2 & ((p \Rightarrow ((p \Rightarrow p) \Rightarrow p)) \Rightarrow ((p \Rightarrow (p \Rightarrow p)) \Rightarrow (p \Rightarrow p))) & q \\ MP & ((p \Rightarrow (p \Rightarrow p)) \Rightarrow (p \Rightarrow p)) & \\ A_1 & (p \Rightarrow (p \Rightarrow p)) & q : p \end{array}$$

Exercises

- Show that $\neg$ and $\Rightarrow$ are sufficient
- Model $(p?q : r)$
- Proove $(p \Rightarrow p)$

$p \vee q : (\neg p \Rightarrow q)$,$p \wedge q : \neg(p \Rightarrow \neg q)$,$(p \Leftrightarrow q) :$
$\neg((p \Rightarrow q) \Rightarrow \neg(q \Rightarrow p))$
$((p \Rightarrow q) \wedge (\neg p \Rightarrow r))$

$$\begin{array}{ll}
A_1 & (p \Rightarrow ((p \Rightarrow p) \Rightarrow p)) \quad q : (p \Rightarrow p) \\
A_2 & ((p \Rightarrow ((p \Rightarrow p) \Rightarrow p)) \Rightarrow ((p \Rightarrow (p \Rightarrow p)) \Rightarrow (p \Rightarrow p))) \quad q \\
MP & ((p \Rightarrow (p \Rightarrow p)) \Rightarrow (p \Rightarrow p)) \\
A_1 & (p \Rightarrow (p \Rightarrow p)) \quad q : p \\
MP & (p \Rightarrow p)
\end{array}$$

Interpretation

Definition

An *interpretation* i is an application

$$i : \{P_1, ..., P_n\} \rightarrow \{T, F\}$$

By extension, concept of interpretation of a *formula* $f \in F(0)$

- $f \in F(0)$ is *consistent* if $\exists i, i(f) = T$
- $f \in F(0)$ is *valid* (or is a *tautology*) if $\forall i, i(f) = T$ (notation $\models f$)
- c is a logical consequence of h: if $i(h) = T$, then $i(c) = T$. Notation $h \models c$ (tautologies are logical consequences of $\emptyset$).

Properties of an axioms schema

- An AS is *consistent* if $\forall f$, if $\vdash f$ then $\models f$ (everything that is demonstrable is true)
- An AS is *complete* if $\forall f$, if $\models f$ then $\vdash f$ (everything that is true is demonstrable)

Validity of a formula

- Always decidable, BUT
- N variables give 2^N distinct interpretations
- Simplification algorithms exist, but inefficient (except clause resolution)

Principle of Deduction

Definition

c is a logical consequence of a set of hypotheses if and only if adding $\neg c$ to this set makes it inconsistent :

$$(h_i) \models c \text{"} \iff \text{"} (h_i) \cup \{\neg c\} \models F$$

Principle of uniform substitution

Definition

Let t be a tautology, p a proposition of t and f a formula. Let

$$t' = \sigma(t, p, f)$$

be the formula obtained when replacing every occurrence of p in t with f.
Then t' is a tautology

Clause and clausal form

Definition

- A *literal* is either a proposition, or its negation.
- A *clause* is a finite disjunction of literals (empty clause is writen $\emptyset$).
- A *normal conjunctive form* (NCF) is a finite conjunction of clauses

```
<L>   = <P>  |  ¬<P>
<C>   = <L>  [∨<L>]*  |  ∅
<NCF> = <C>  [∧<C>]*
```

Formulas and NCF

Theorem

Any fomula of $F(0)$ has an equivalent NCF

Proof.

(by construction):

1. transform $\Leftrightarrow$ into $(\Rightarrow, \wedge)$ pairs
2. transform $p \Rightarrow q$ into $\neg p \vee q$
3. put $\neg$ inside formulas, remove $\neg\neg$
4. distribute ORs: $\begin{cases} p \vee (q \wedge r) & \longrightarrow & (p \vee q) \wedge (p \vee r) \\ (p \wedge q) \vee r & \longrightarrow & (p \vee r) \wedge (q \vee r) \end{cases}$

$\square$

Notes about NCF

- Any clause containing a literal and its negation is valid (no other valid clause), and can be removed from the NCF
- A “pure” NCF contains at most 1 occurrence of any literal
- If one clause of an NCF is a subclause of another, the latter can be removed
- If $\emptyset$ is in a NCF, the NCF is inconsistent

Exercise

put $((p \Rightarrow (q \Rightarrow r)) \Rightarrow ((p \wedge s) \Rightarrow r))$ into NCF

Exercise

put $((p \Rightarrow (q \Rightarrow r)) \Rightarrow ((p \wedge s) \Rightarrow r))$ into NCF

$(\neg(p \Rightarrow (q \Rightarrow r)) \vee ((p \wedge s) \Rightarrow r))$

Exercise

put $((p \Rightarrow (q \Rightarrow r)) \Rightarrow ((p \wedge s) \Rightarrow r))$ into NCF
$(\neg\,(p \Rightarrow (q \Rightarrow r)) \vee ((p \wedge s) \Rightarrow r))$
$(\neg\,(\neg p \vee (\neg q \vee r)) \vee (\neg(p \wedge s) \vee r))$

Exercise

put $((p \Rightarrow (q \Rightarrow r)) \Rightarrow ((p \wedge s) \Rightarrow r))$ into NCF
$(\neg (p \Rightarrow (q \Rightarrow r)) \vee ((p \wedge s) \Rightarrow r))$
$(\neg (\neg p \vee (\neg q \vee r)) \vee (\neg(p \wedge s) \vee r))$
$((\neg\neg p \wedge (\neg\neg q \wedge \neg r)) \vee ((\neg p \vee \neg s) \vee r))$

Exercise

put $((p \Rightarrow (q \Rightarrow r)) \Rightarrow ((p \wedge s) \Rightarrow r))$ into NCF
$(\neg (p \Rightarrow (q \Rightarrow r)) \vee ((p \wedge s) \Rightarrow r))$
$(\neg (\neg p \vee (\neg q \vee r)) \vee (\neg(p \wedge s) \vee r))$
$((\neg\neg p \wedge (\neg\neg q \wedge \neg r)) \vee ((\neg p \vee \neg s) \vee r))$
$((p \wedge (q \wedge \neg r)) \vee ((\neg p \vee \neg s) \vee r))$

Exercise

put $((p \Rightarrow (q \Rightarrow r)) \Rightarrow ((p \wedge s) \Rightarrow r))$ into NCF

$(\neg(p \Rightarrow (q \Rightarrow r)) \vee ((p \wedge s) \Rightarrow r))$

$(\neg(\neg p \vee (\neg q \vee r)) \vee (\neg(p \wedge s) \vee r))$

$((\neg\neg p \wedge (\neg\neg q \wedge \neg r)) \vee ((\neg p \vee \neg s) \vee r))$

$((p \wedge (q \wedge \neg r)) \vee ((\neg p \vee \neg s) \vee r))$

$(p \vee \neg p \vee \neg s \vee r) \wedge (q \vee \neg p \vee \neg s \vee r) \wedge (\neg r \vee \neg p \vee \neg s \vee r)$

Exercise

put $((p \Rightarrow (q \Rightarrow r)) \Rightarrow ((p \wedge s) \Rightarrow r))$ into NCF
$(\neg (p \Rightarrow (q \Rightarrow r)) \vee ((p \wedge s) \Rightarrow r))$
$(\neg (\neg p \vee (\neg q \vee r)) \vee (\neg (p \wedge s) \vee r))$
$((\neg\neg p \wedge (\neg\neg q \wedge \neg r)) \vee ((\neg p \vee \neg s) \vee r))$
$((p \wedge (q \wedge \neg r)) \vee ((\neg p \vee \neg s) \vee r))$
$(p \vee \neg p \vee \neg s \vee r) \wedge (q \vee \neg p \vee \neg s \vee r) \wedge (\neg r \vee \neg p \vee \neg s \vee r)$
$(q \vee \neg p \vee \neg s \vee r)$

Principle of resolution

- Let f be a NCF, c_1, c_2 two clauses of f, l a literal. If $\left\{ \begin{array}{c} l \in c_1 \\ \neg l \in c_2 \end{array} \right.$, then $r = c_1 - \{l\} \cup c_2 - \{\neg l\}$ is the *resolvent* clause of c_1 and c_2.
- In this situation, f and $f \cup \{r\}$ are equivalent: it is the basis for the *resolution algorithm*.

Resolution algorithm

```
while $\emptyset \notin f$
    choose $l, c_1, c_2 / \begin{cases} l \in c_1 \\ \neg l \in c_2 \end{cases}$
    if impossible exit(failure)
    compute $r$
    replace $f$ with $f \cup \{r\}$
```

Exercise

show that p is a logical consequence of
$(p \vee q) \wedge (p \vee r) \wedge (\neg q \vee \neg r)$
We then try to show that
$f = \{(p \vee q)_{(1)}, (p \vee r)_{(2)}, (\neg q \vee \neg r)_{(3)}, \neg p_{(4)}\}$ is inconsistent.

Exercise

show that p is a logical consequence of
$(p \vee q) \wedge (p \vee r) \wedge (\neg q \vee \neg r)$
We then try to show that
$f = \{(p \vee q)_{(1)}, (p \vee r)_{(2)}, (\neg q \vee \neg r)_{(3)}, \neg p_{(4)}\}$ is inconsistent.

automatic:
5:$p \vee \neg r$ (1,3)
6:q (1,4)
7:$p \vee \neg q$ (2,3)
8: r (2,4)
9: p (2,5)
...
13:$\neg q$ (4,7)
14:$\emptyset$ (4,9)

manual:
5: q (1,4)
6: r (2,4)
7: $\neg r$ (3,5)
8: $\emptyset$ (6,7)

Exercise

Proove the “case disjunction”: if a (true) hypothesis implies a disjunction, and each member of the disjunction imply the same conclusion, then the conlusion is true:
$\{h, h \Rightarrow (p \vee q), p \Rightarrow c, q \Rightarrow c\} \models c$

Exercise

Proove the "case disjunction": if a (true) hypothesis implies a disjunction, and each member of the disjunction imply the same conclusion, then the conlusion is true:
$\{h, h \Rightarrow (p \vee q), p \Rightarrow c, q \Rightarrow c\} \models c$
$f = \{h_{(1)}, (\neg h \vee p \vee q)_{(2)}, (\neg p \vee c)_{(3)}, (\neg q \vee c)_{(4)}, \neg c_{(5)}\}$

Exercise

Proove the "case disjunction": if a (true) hypothesis implies a disjunction, and each member of the disjunction imply the same conclusion, then the conlusion is true:
$\{h, h \Rightarrow (p \vee q), p \Rightarrow c, q \Rightarrow c\} \models c$
$f = \{h_{(1)}, (\neg h \vee p \vee q)_{(2)}, (\neg p \vee c)_{(3)}, (\neg q \vee c)_{(4)}, \neg c_{(5)}\}$
6: $p \vee q$ $(1,2,h)$
7: $\neg p$ $(3,5,c)$
8: $\neg q$ $(4,5,c)$
9: q $(6,7,p)$
10: $\emptyset(8,9,\ q)$.

Horn clauses

Definition

A *Horn clause* is a clause which contains at most 1 positive literal.

- Base for “if ..... then *conclusion*”
- If no hypothesis, the clause is called a *fact*
- Advantage: resolution algorithm is simpler

Resolution algorithm for Horn clauses

```
while $\emptyset \notin f$
    choose $p, c / \neg p \in c$
    if impossible exit(failure)
    replace $f$ with $f - c \cup (c - \{\neg p\})$
```

- NB: always terminate, since 1 literal less at each iteration
- If N literals in f, $C = O\left(N^2\right)$

Outline

PC(1) I

- $\Sigma(1) = \{x, y, ...\}$ (variables: `<var>`)
  $\cup\{a, b, ...\}$ (individual constants `<var>`)
  $\cup\{f, g, ...\}$ (functional constants `<fct>`)
  $\cup\{p, q, ...\}$ (predicate constants `<pct>`)
  $\cup\{\vee, \wedge, \Rightarrow, \Leftrightarrow\}$ (binary logical connectors `<blc>`)
  $\cup\{\exists, \forall\}$ (quantifiers `<q>`)
  $\cup\{\neg\}$
- $F(1)$:

```
<wff> = <at> | ¬<wff> |
(<wff><blc><wff>) | (<q><var>)<wff>
<at> = <pf> | (<t> = <t>)
<t> = <var> | <ff>
<ff> = <fct>([<t>[,<t>]*]?)
<pf> = <pct>([<t>[,<t>]*]?)
```

PC(1) II

- $A(1) = A(0)$ (except formulas in $F(1)$), plus:
  $(\forall x, p(x)) \Rightarrow p(t)$(any term)
  $((p \Rightarrow q) \Rightarrow (p \Rightarrow (\forall x, p(x))))$ (x is not free in p)
- $R(1) = MP+$
  $A \vdash (\forall x, A)$ (generalization rule)

NB: $T(1)$ is *not* recursive (an infinity of possible interpretations for formulas)

Bound variables

Definition

Let $V_b(f)$ be the set of "bound" variables of formula f. It is defined constructively:

- $V_b(< at >) = \emptyset$
- $V_b(f \Rightarrow g) = V_b(f) \cup V_b(g)$
- $V_b(\neg f) = V_b(f)$
- $V_b(\forall x, f) = V_b(f) \cup \{x\}$

Free variables

Definition

Let $V_f(f)$ be the set of "free" variables of formula f. It is defined constructively:

- $V_b(< at >) = V(< at >)$
- $V_f(f \Rightarrow g) = V_f(f) \cup V_f(g)$
- $V_f(\neg f) = V_f(f)$
- $V_f(\forall x, f) = V_f(f) - \{x\}$

A formula without any free variable is said to be *CLOSED*

Examples

- $p(f(x,y)) \vee (\forall z, r(a,z))$
  $V_b = \{z\},\ V_f = \{x,y\}$
- $(\forall x, p(x,y,z)) \vee (\forall z, (p(z) \Rightarrow r(z)))$
  $V_b = \{x,z\},\ V_f = \{y,z\}$
- $\forall x, \exists y, (p(x,y) \Rightarrow (\forall z, r(x,y,z)))$
  $V_b = \{x,y,z\},\ V_f = \emptyset$

Substitution

Definition

A substitution is an application

$$\begin{aligned}\sigma : \quad < \quad var > &\longrightarrow < t > \\ x &\longmapsto t\end{aligned}$$

σ is said to be *finite* if $\sigma(x) = x$ almost everywhere.
By extension: $\sigma(t)$ is the term obtained by replacing each variable in t with its image by σ

Example

Example

$\sigma = \{(x, f(x)), (y, g(x,z))\},\ t = g\left(f(x), g\left(f(z), y\right)\right)$ gives
$\sigma(t) = g\left(f\left(f(x)\right), g\left(f(z), g(x,z)\right)\right)$

NB: $\circ$(*composition law of substitutions*) is internal in <t>, associative and has a neutral element, the identity (it's a *monoïd*)

Instanciation

Definition

Let t_1, t_2 be two terms. t_2 is an *instance* of t_1 if $\exists\sigma, t_2 = \sigma(t_1)$.
If $V(t) = \emptyset$, t is said to be *completely instanciated*.
A formula f is *valid* iff all of its instances are valid.
A formula f is *consistent* iff one of its instances is consistent.

Prenex form

Definition

A sentence is in *prenex* form if all its quantifiers come at the very start, i.e., no quantifiers are within the scope of a truth-functional connective.

Theorem

Any formula has a prenex form which is equivalent

Equivalent prenex form

Proof.

By construction:

1. eliminate $\Leftrightarrow$ and $\Rightarrow$
2. rename bound variables until $V_b \cap V_f = \emptyset$
3. remove useless quantifiers
4. put $\neg$ as close as possible to <pct>: $\neg\forall x, p \longrightarrow \exists x, \neg p$, $\neg(p \wedge q) \longrightarrow (\neg p \vee \neg q)$, etc.
5. reject quantifiers to the beginning of the formula:
   $(\forall x, p \wedge \forall x, q) \longrightarrow \forall x, (p \wedge q)$
   $((\forall x, p) \wedge q) \longrightarrow \forall x, (p \wedge q)$ (if q does not contain x), etc.

$\square$

Example

$\forall x\,(p(x) \land \forall y, \exists x\,(\neg q(x,y) \Rightarrow \forall z, r(a,x,y)))$
$\forall x\,(p(x) \land \forall y, \exists x\,(\neg\neg q(x,y) \lor \forall z, r(a,x,y)))$
$\forall x\,(p(x) \land \forall y, \exists u\,(q(u,y) \lor \forall z, r(a,u,y)))$
$\forall x \forall y\,(p(x) \land \exists u\,(q(u,y) \lor r(a,u,y)))$
$\forall x \forall y \exists u\,(p(x) \land (q(u,y) \lor r(a,u,y)))$
NB: the prenex form is not unique

NCF in CP(1)

```
<lit> = <at> | ¬<at>
<cl> = <lit> [∨ <lit>]⋆
<ncf> = <qf>⋆ (<cl> [∧ <cl>]⋆)
<qf> = <q> <var>
```

Theorem

Any wff has a ncf which is equivalent

Skolemization

- Further simplification of a ncf.
- Principle:
  1. Replace any existencially-quantified variable with a function of the universally-quantified variables than come before it in the formula
  2. Remove all occurrences of $\forall$ (all variables are implicitly universally quantified)

Example of skolemization

Example

$\forall x\,(p(x) \wedge \forall y, \exists x\,(\neg q(x,y) \Rightarrow \forall z, r(a,x,y)))$ gives the ncf $\forall x \forall y \exists u\,(p(x) \wedge (q(u,y) \vee r(a,u,y)))$ which is “skolemized” in $(p(x) \wedge (q(f(x,y),y) \vee r(a,f(x,y),y)))$

Unification

Definition

Unification is resolution applied to skolem forms. It is the basic mechanism of PROLOG

Principle:

- $c_1, c_2 / c_1 \ni l_1, c_2 \ni \neg l_2,\ V(c_1) \cap V(c_2) = \emptyset$ (possibly after some renaming) and l_1 and l_2 are *unifiable* (*i.e.* they have a common instance).
- Consider $c'_1, c'_2 / l'_1 = l'_2 = l'$ and $r = (c'_1 - \{l'\} \cup c'_2 - \{l'\}$
- Then $f \cup r$ is a logical consequence of f

Algorithm

```
while $\emptyset \notin f$
   choose $l_1, l_2, c_1, c_2 / \begin{cases} l_1 \in c_1 \\ \neg l_2 \in c_2 \end{cases}$ and $(l_1, l_2)$
unifiable
   if impossible exit(failure)
   compute $r$
   replace $f$ with $f \cup \{r\}$
```

Outline

Introduction to PROLOG

- Unification between skolemized Horn clauses of $PC(1)$
- The positive literal is separated from the other (negative) literals with a "$\vdash$" sign ("if")
- Example: compute the gcd of two positive integers x and y
  - if x is equal to y, the result is x
  - if x is greater (*resp.* less) than y, the result is the same as the gcd of $(x - y)$ and y (*resp.* x and $(y - x)$).

PROLOG-like notation

Let's write $gcd(X, Y, Z)$ for "Z is the gcd of X and Y". We get 3 clauses:

```
1: gcd(X,X,X). % variables are in uppercase
2: gcd(X,Y,Z) ⊢ X>Y, gcd(X-Y,Y,Z).
3: gcd(X,Y,Z) ⊢ Y>X, gcd(X,Y-X,Z).
```

Attention

Expressions in predicates must be "matchable":
Rule 2 for instance must be written
`gcd(X,Y,Z) ⊢ X>Y, DIFF is X-Y, gcd(DIFF,Y,Z).`

Let's compute the gcd of 4 et 6: we add the goal
`⊢gcd(4,6,Z).`

Derivation Example

```
4:  gcd(4,6,Z)
5:  gt(6,4),gcd(4,2,Z)  //  3:X=4,Y=6
6:  gt(4,2),gcd(2,2,Z)  //  2:X=4,Y=2
7:  ∅ //  1:X=2,Z=2
```

So $gcd(4,6) = 2$ (final value of Z)

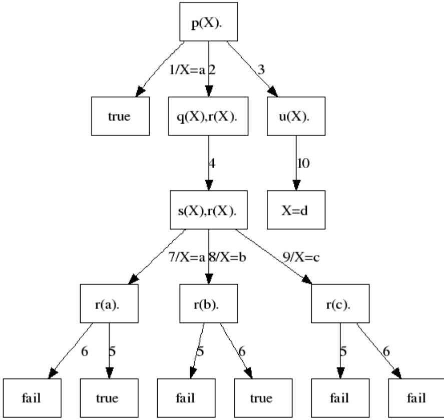
Derivation tree

- The goal is matched against the goal part of each rule
- If a rule matches, all its hypotheses are added as subgoals
- This leads to a tree-like structure (the *derivation tree*) which is visited using a depth-first, left-handed method: the order in which rules are written *is* significant!

Simple example

Program P	cl. #
p(a).	1
p(X) :- q(X),r(X).	2
p(X) :- u(X).	3
q(X) :- s(X).	4
r(a).	5
r(b).	6
s(a).	7
s(b).	8
s(c).	9
u(d).	10

We want to see what happens for goal $\vdash$p(X).:



Unification in PROLOG

- depth-first traversal of derivation tree
- if left-most subgoal unifies with head of side clause, then the subgoal is replaced with the body of the side clause:

```
g1,g2,...
h :- b1,b2,...
---------         // if g1 unifies with h
b1,b2,...,g2,...
```

- N.B.: some variables in (b_i) and (g_j) have been bound during unification
- If the tail of a rule is empty $(b_i) = \emptyset$ then subgoal g_1 can be removed
- When all subgoals are removed along a path, a "yes" is generated

Simple examples

```
?- p(X,f(Y),a) = p(a,f(a),Y).
X = a Y = a
?- p(X,f(Y),a) = p(a,f(b),Y).
No
?- p(X,f(Y),a) = p(Z,f(b),a).
X = _G182 Y = b Z = _G182
?- p(X,f(Y),a) = p(Z,f(b),a), X is d.
X = d Y = b Z = d
```

Several built-in PROLOG goals

- `trace`, `notrace`
- `true`, `fail`
- `[fileName]` loads fileName.pl (syn. `consult('fileName.pl')`)
- Numerical comparisons $< \;\; <= \;\; >= \;\; >$
- `is` : logical variable (numerical) binding
- Type predicates `integer(X)`, `real(X)`, `string(X)`...

Examples (2)

- matching and equality: `=` `\=` `==` `\==`

```
?- [X,Y|R] = [a,b,c]
X = a, Y = b, R = [c]
?- [X,Y,Z] = [a,b]
No
```

- `call(P)` forces P to be a goal; same success/failure
- `!` cut predicate
- `not` as if defined by (*exercise after cut def.*)

```
not(P) :- call(P), !, fail.
not(P).
```

The “!” (cut) predicate

- branches of the derivation tree preceeding the “!” are eliminated from the backtrack process
- Variables bound at the time the “!” is encountered stay bound to the same value
- Ex: previous set of clauses, and goals “`p(X),!.`”, “`r(X),!,s(Y).`” and “`r(X),s(Y),!.`”:

```
?- p(X),!.
X = a ;
No
?- r(X),s(Y).
X = a Y = a ;
X = a Y = b ;
X = a Y = c ;
...
X = b Y = c ;
No
```

The cut operator (2)

```
?- r(X),!,s(Y).
X = a Y = a ;
X = a Y = b ;
X = a Y = c ;
No
?- r(X),s(Y),!.
X = a Y = a ;
No
```

The cut operator (3)

```
red(a). black(b).
color(P,red) :- red(P),!.
color(P,black) :- black(P),!.
color(_,unknown).
```

- What happens if no “!” ? (examine `color(X,red)` and `color(a,Y)`)[5]
- What happens to goal `p(X)` if clause #2 is replaced with `p(X) :- q(X), !, r(X)`?

```
?- p(X).
X = a ? ;
X = a
yes
```

[5] respectively “a” then “No”, and “red” then “unknown”

Hanoi towers

- Simple example of recursion: move N disks from pin p_1 to pin p_2 using pin p_3, with a constraint: a larger disk can never be placed above a narrow one.
- Predicate: `hanoi(N, from, to, using)`

```
hanoi(1,I,F,_) :-
   format("moving from %d to %d\n",[I,F]).
hanoi(N,I,F,AUX) :- N>1,
   N1 is N-1,
   hanoi(N1,I,AUX,F),
   hanoi(1,I,F,AUX),
   hanoi(N1,AUX,F,I).
```

Outline

Brief history

- Original paper: L.A. Zadeh 65
- Fuzzy logic & neural networks: E. Mamdani (1973)
- 1st "fuzzy" VLSI: 1989

Principle and applicability

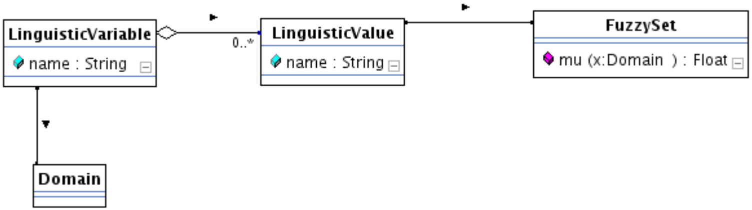
- Idea: switch from binary, “true/false” logic to a measure of uncertainty in truth
- Base: theory of sets $\longrightarrow$ continuum of grades of membership (*membership function* in $[0, 1]$)
- Accurate if
  - very complex process without simple mathematical model
  - non-linearity
  - must deal with linguistic, human expert knowledge

Definitions

Definitions

fuzzy set: set of pairs $(x, \mu(x))$ where μ takes values in $[0, 1]$
linguistic variable: variable which represent process / control state, and whose value are defined in linguistic terms
linguistic value: fuzzy set mapping crisp values to degree of membership to this value of the linguistic variable
universe of discourse: set of possible linguistic values

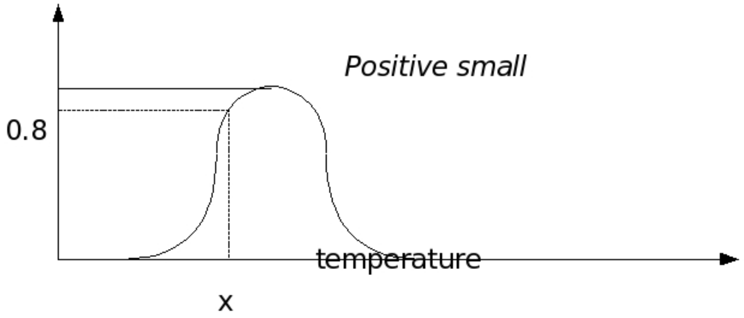
Relationships



Example

Example

T(emperature) = { negative big, negative medium, negative small, close to zero, positive small, positive medium, positive big }



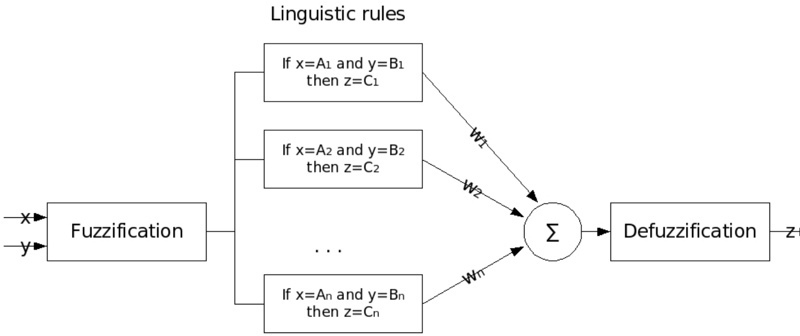
Operators

- Several possible sets of operators. Most common:
  - $\mu(\neg p) = 1 - \mu(p)$
  - $\mu(p \vee q) = max(\mu(p), \mu(q))$ (algebraic sum,...)
  - $\mu(p \wedge q) = min(\mu(p), \mu(q))$ (algebraic product,...)
- Hedges (modifiers):
  - very: $\mu(very(x)) =_{def} \mu(x)^2$
  - more or less: $\mu(mol(x)) =_{def} \sqrt{\mu(x)}$

Linguistic rules

- Two parts, *antecedent* (premise): `if ...`, and *consequent*: `then ...`
- $\mu(cons) =_{def} \mu(premise)$
- *Fuzzy controller*: set of fuzzy linguistic rules.

Structure of a classical fuzzy system


Linguistic rules
x
y
Fuzzification
If x=A1 and y=B1 then z=C1
If x=A2 and y=B2 then z=C2
. . .
If x=An and y=Bn then z=Cn
W1
W2
Wn
Σ
Defuzzification
z

Classical steps

1. Fuzzification: measure of input variable $\longrightarrow$ degree of membership for every fuzzy set of the universe of discourse
2. Computation of each rule *firing strength* (or *weight*) using operators (min)
3. Generation of *consequent value* for each rule and computation of $\mu_C(z)$
4. Defuzzification: generation of the crisp output value(s)

Types of fuzzy reasoning

- Tsukamoto: if output membership function is increasing, then the overall output can be a weighted average of generated crisp output values
- Lee: operation MAX on the qualified fuzzy outputs, overall output is the center of gravity (most common)
- Takagi and Sugeno: each rule's output is a linear combination of input variables; overall crisp output is their weighted average
- ...

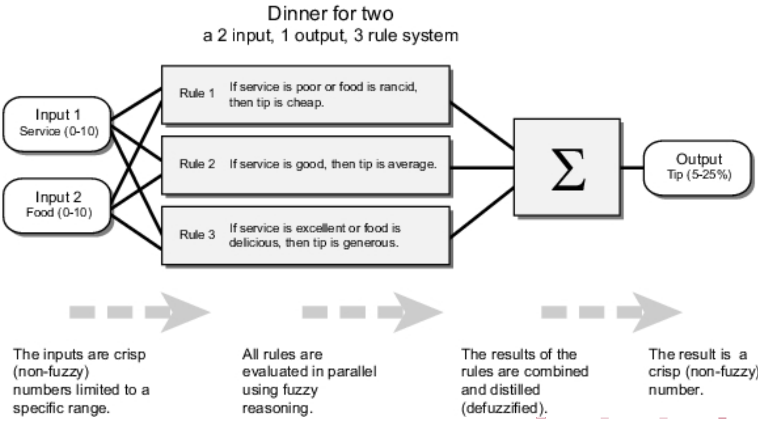
Concrete application example

- Fuzzy air-conditioned system (Mitsubishi) handling weather changing conditions
  - 50 rules, 6 linguistic variables (room and wall temperature, ...)
  - prototype: 4 man.days, tests and integration: 20 man.days, optimization: 80 man.days
  - implemented on a standard micro-controller
  - results: startup process time reduced by 40%, much more robust to interferences (window opening...), less sensors, 24% energy saved.

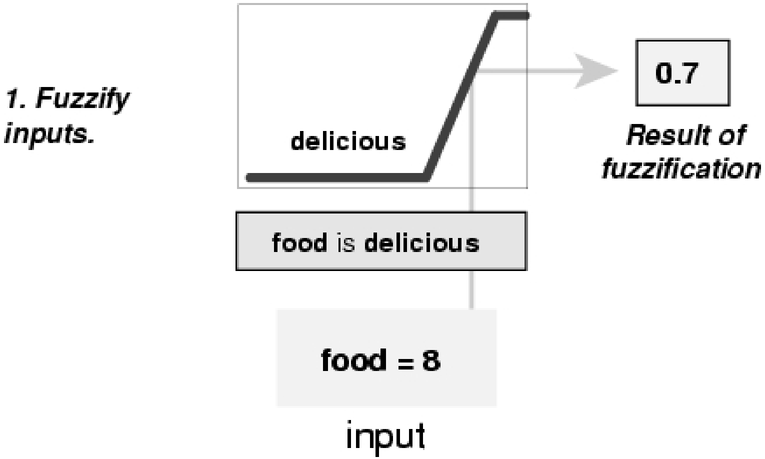
A simple but complete example

Drawn from the *mathworks* site (`http://www.mathworks.com`):
the system computes the tip to give after

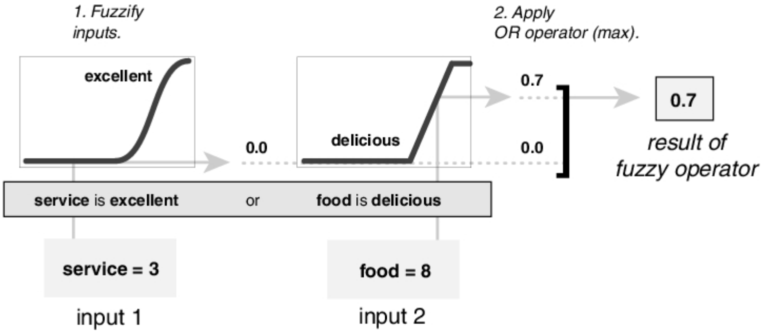
- quality of food
- quality of service


Dinner for two
a 2 input, 1 output, 3 rule system
Input 1
Service (0-10)
Input 2
Food (0-10)
Rule 1
If service is poor or food is rancid, then tip is cheap.
Rule 2
If service is good, then tip is average.
Rule 3
If service is excellent or food is delicious, then tip is generous.
Σ
Output
Tip (5-25%)
The inputs are crisp (non-fuzzy) numbers limited to a specific range.
All rules are evaluated in parallel using fuzzy reasoning.
The results of the rules are combined and distilled (defuzzified).
The result is a crisp (non-fuzzy) number.

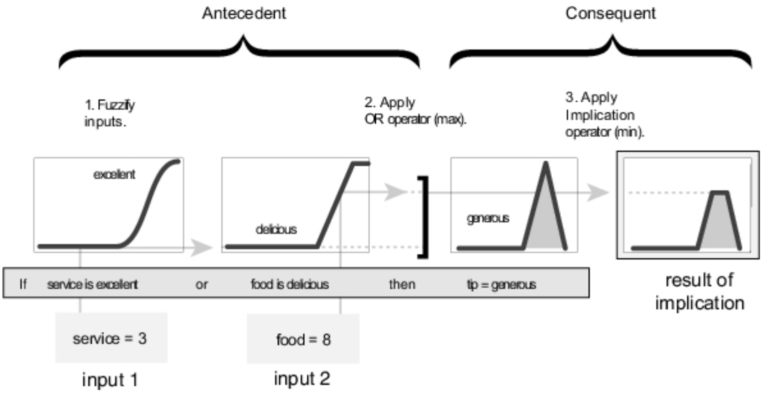
Step 1: fuzzification



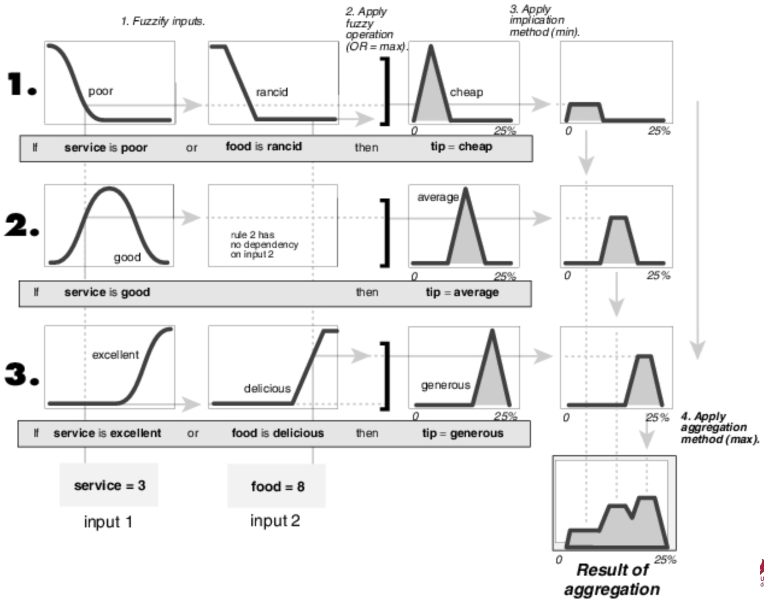
Combining operators



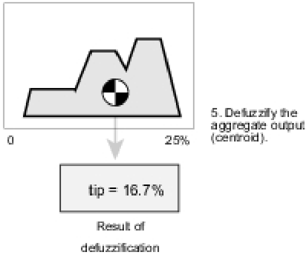
Step 2: Firing rules


Antecedent
Consequent
1. Fuzzify inputs.
2. Apply OR operator (max).
3. Apply Implication operator (min).
excellent
delicious
generous
result of implication
If service is excellent or food is delicious then tip = generous
service = 3
input 1
food = 8
input 2

Step 4: Output aggregation


1. Fuzzify inputs.
2. Apply fuzzy operation (OR = max).
3. Apply implication method (min).
1.
poor
rancid
cheap
0
25%
If service is poor or food is rancid then tip = cheap
2.
good
rule 2 has no dependency on input 2
average
If service is good then tip = average
3.
excellent
delicious
generous
If service is excellent or food is delicious then tip = generous
4. Apply aggregation method (max).
service = 3
input 1
food = 8
input 2
Result of aggregation

Step 5: defuzzification


0
25%
5. Defuzzify the aggregate output (centroid).
tip = 16.7%
Result of defuzzification

Bibliography I

Colmerauer, A. “Prolog in 10 figures”, *in* Communications of the Association for Computing Machinery, 28(12):1296-1310, 1985

Kleene, S. C. “Mathematical Logic”, New York: Dover, 2002.

C.C. Lee , “Fuzzy logic in control systems: fuzzy logic controller - part 1 & 2”, IEEE Trans. on Systems, Man and Cybernetics, 20 (2), pp 404–435, 1990.

Levy, David N.L. “How Computers Play Chess, New York”, Computer Science Press, 1991[6]

J. McCarthy, “A Basis for a Mathematical Theory of Computation”, *in* Computer Programming and Formal Systems, North Holland, 1961

Bibliography II

- E. Mamdani, S. Assilian, “An experiment in Linguistic Synthesis with a Fuzzy Logic Controller”, Int. Journal on Man-Machine Studies, 7, 1973, pp 1–13.
- Minsky, M. “The Society of Mind”, Simon & Schuster (March 15, 1988)
- A. Newell & H. Simon, “The Theory of Human Problem Solving”, reprinted in Collins & Smith (eds.), Readings in Cognitive Science, section 1.3.
- N.J. Nilsson “Principles of Artificial Intelligence”, Tioga Publishing Co., 1980[7]
- Papert, S. “Mindstorms: Children, Computers, and Powerful Ideas”, New York, Basic Books, 1980
- Russel, S., Norvig, P., “Intelligence artificielle”, Pearson education, 2006 (2e edition)

Bibliography III

- Sterling L., Shapiro E., “The Art of PROLOG”, MIT Press
- T. Takagi, M. Sugeno, “Derivation of fuzzy control rules from human operator’s control actions”, Proc. of the IFAP Symp. on fuzzy information, knowledge representation and decision analysis, pp 55–60, july 1983
- S. Tanimoto, “The Elements of Artificial Intelligence Using Common Lisp”, W.H. Freeman & Co, 1995[8]
- Tsukamoto T., “An approach to fuzzy reasoning method”, in Advances in Fuzzy Set Theory and Applications, M. Gupta, R.K. Ragade & R.R. Yager, eds., North Holland, 1979, pp. 137–149

Bibliography IV

J. Weizenbaum, “ELIZA–A Computer Program For the Study of Natural Language Communication Between Man and Machine”, Communications of the ACM Volume 9, Number 1 (January 1966): 36-35.

N. Wiener, “Cybernetics - Control and Communication in the Animal and the Machine”, MIT Press, 1948 (reed. 1961).

L.A. Zadeh, “Fuzzy sets”, Inf. Control 8, 338-353, 1965.

[6] This reference introduces the game of chess and have well explanation of minimax algorithm and alpha_beta cutoff

[7] Good general reference on artificial intelligence and on minimax trees.

[8] Very clear and complete, though using a - now - esoteric programming language