Knowledge-Based Systems

Suendermann

Baden-Württemberg Cooperative State University

Stuttgart, Germany

http://suendermann.com

David Suendermann

Knowledge-Based Systems
General remarks

The most up-to-date version of this document as well as auxiliary material can be found online at http://suendermann.com

Scripts and other materials by my colleague Dirk Reichardt covering some of the topics discussed in this lecture:

Outline

- Prolog
- Expert systems and dialogue systems
- Intelligent search and problem solving strategies
- Logic and computer-assisted proof
Outline

- Prolog
- Expert systems and dialog systems
- Intelligent search and problem solving strategies
- Logic and computer-assisted proof
Propositional logic: tautology

Ludwig Wittgenstein (1921): A formula which is true in every possible interpretation is called a tautology.

Examples:
- \( A \leftrightarrow A \)
- \( A \lor \lnot A \) (excluded middle)
- \( A \rightarrow B \leftrightarrow \lnot B \rightarrow \lnot A \) (contraposition)
- \( (A \rightarrow B) \land (B \rightarrow C) \rightarrow (A \rightarrow C) \) (syllogism)
- \( (\lnot A \land B) \leftrightarrow \lnot A \lor \lnot B \) (De Morgan’s law)
- \( (A \lor B) \land (A \rightarrow C) \land (B \rightarrow C) \rightarrow C \) (proof by cases)
Precedence and associativity of logical connectives

Outermost parentheses can be dropped:

1. \((b \lor d) \iff (b \lor d)\)

2. \((a \land b) \iff (a \land b)\)

3. \(a \leftarrow b \leftarrow d \iff a \leftarrow (b \leftarrow d)\)

Assume operators of the same precedence to be left-associative:

Consider the following precedence:

\[
\begin{array}{cccc}
\text{operator} & 1 \text{(strongest)} & 2 & 3 & 4 & 5 \text{(weakest)} \\
\uparrow & \land & \lor & \rightarrow & \neg & \iff \\
\end{array}
\]

E.g., we have:

- \((p \rightarrow q) \rightarrow r \iff p \rightarrow q \rightarrow r\)

- \((x \land b) \rightarrow (b \lor d) \iff (x \land b) \leftarrow (b \lor d)\)

- \((x \leftarrow b \leftarrow d) \iff x \leftarrow (b \leftarrow d)\)
Prove that the following formula is a tautology:

\( A \land B \rightarrow C \leftrightarrow A \rightarrow (B \rightarrow C) \)

Using

a) known equivalences,

b) a truth table.

*Exercise: Tautology*
Multiple applications require sentences in propositional or 1st-order logic to be given as a (conjunctive) set of clauses. A clause is a disjunction of literals. A literal is an atomic formula or its negation. These conditions are fulfilled by the conjunctive normal form (CNF):

\[
\bigwedge_i \bigvee_j \left[ \neg \right] P_{ij}.
\]

Example for a propositional formula in CNF:

\[
\left( \neg A \lor B \lor C \right) \land \left( A \lor B \lor \neg C \right).
\]
Among other ways, we can convert a given propositional formula into CNF by:

- applying equivalences or
- establishing a truth table.

Example: We want to transform the formula

\[ A \rightarrow (B \leftrightarrow C) \]

into CNF. Another way to put this CNF is the set notation:

\[ \{\{\neg A, B, \neg C\}, \{\neg A, B, C\}\} \]

Another way to put this CNF is the set notation:

\[ \{\{\neg A, B, \neg C\}, \{\neg A, B, C\}\} \]

Example: We want to transform the formula

\[ A \rightarrow (B \leftrightarrow C) \]

into CNF. Another way to put this CNF is the set notation:

\[ \{\{\neg A, B, \neg C\}, \{\neg A, B, C\}\} \]

Another way to put this CNF is the set notation:

\[ \{\{\neg A, B, \neg C\}, \{\neg A, B, C\}\} \]

Another way to put this CNF is the set notation:

\[ \{\{\neg A, B, \neg C\}, \{\neg A, B, C\}\} \]

Another way to put this CNF is the set notation:

\[ \{\{\neg A, B, \neg C\}, \{\neg A, B, C\}\} \]

Another way to put this CNF is the set notation:

\[ \{\{\neg A, B, \neg C\}, \{\neg A, B, C\}\} \]

Another way to put this CNF is the set notation:

\[ \{\{\neg A, B, \neg C\}, \{\neg A, B, C\}\} \]

Another way to put this CNF is the set notation:

\[ \{\{\neg A, B, \neg C\}, \{\neg A, B, C\}\} \]

Another way to put this CNF is the set notation:

\[ \{\{\neg A, B, \neg C\}, \{\neg A, B, C\}\} \]
b) The conjunctive combination of all those clauses producing the result 0 in the truth table is the CNF.

\[
\begin{array}{c|cccc}
\text{clause} & A & B & C & \text{result} \\
\hline
\neg A \lor \neg B \lor \neg C & 0 & 0 & 0 & 1 \\
A \lor \neg B \lor \neg C & 1 & 0 & 0 & 1 \\
\neg A \lor B \lor \neg C & 0 & 1 & 1 & 1 \\
\neg A \lor \neg B \lor C & 0 & 0 & 1 & 1 \\
A \lor \neg B \lor C & 1 & 0 & 1 & 1 \\
\neg A \lor B \lor C & 0 & 1 & 1 & 1 \\
\neg A \lor \neg B \lor \neg C & 0 & 0 & 0 & 0 \\
\neg A \lor \neg B \lor C & 0 & 0 & 0 & 0 \\
\end{array}
\]

\(\neg \neg A \lor \neg \neg B \lor \neg \neg C\)

So, we are getting the same CNF here, too:

\[
(\neg A \lor \neg B \lor \neg C) \land (\neg A \lor \neg B \lor C)
\]

In the truth table is the CNF.
Inspector Watson is called to a jewelry store that has been subject to a robbery where three subjects, Austin, Brian, and Colin, were arrested. After evaluation of all facts, this is known:

1. At least one of the subjects is guilty:
   \[ f_1 : = A \land B \land C. \] (11)

2. If Austin is guilty, he had exactly one accomplice:
   \[ f_2 : = A \land B \land C. \] (12)

3. If Brian is innocent, so is Colin:
   \[ f_3 : = B \land C. \] (13)

4. If exactly two subjects are guilty, Colin is one of them. Hence, out of
   \[ f_4 : = B \land C. \] (14)

5. If Colin is innocent then Austin is guilty:
   \[ f_5 : = C \land A. \] (15)

After evaluation of all facts, this is known:

Inspector Watson is called to a jewelry store that has been subject to a robbery where three subjects, Austin, Brian, and Colin, were arrested.

Question: Who are the culprits?
A handy first step to approach this question is to turn all the involved formulas into CNF:

\[
\begin{align*}
\text{f}_1 \iff & \{A, B, C\} \\
\text{f}_2 \iff & A \to B \land \neg C \lor \neg B \land C \\
\text{f}_3 \iff & \neg A \lor B \land \neg C \lor \neg B \land C \\
\text{f}_4 \iff & \neg A, \neg B, C \\
\text{f}_5 \iff & \{A, C\} \\
\end{align*}
\]

(16)

b) truth table,

a) resolution,

Now, to answer our question, there are again several possibilities:

\[
\begin{align*}
\{\{A, C\}\} & \iff \text{f}_5 \\
\{\{C\}, \neg B, C\} & \iff \text{f}_4 \\
\{\neg A, \neg B\} & \iff \text{f}_3 \\
\{\neg A, C\} & \iff \text{f}_2 \\
\{\neg A, \neg B\} & \iff \text{f}_1 \\
\{\{A, B, C\}\} & \iff \text{f}_1 \\
\end{align*}
\]

Formulas into CNF:

Who are the culprits? CNF
Resolution (introduced 1965 by John Robinson) is a method to test the validity of a formula or to find a solution to a set of assumptions.

Resolution is defined in form of an algorithm and can, thus, be performed by a computer program.

We are given two clauses of a propositional formula in CNF: \( C_1 \) and \( C_2 \).

We are given two clauses of a propositional formula in CNF: \( C_1 \) and \( C_2 \).

We assume there is a literal \( T \) which exists in \( C_1 \) and whose complement \( \neg T \) exists in \( C_2 \).

Then, we can derive a resolvent \( R \) by merging the original clauses eliminating the complementary literals \( T \) and \( \neg T \).

Then, we can derive a resolvent \( R \) by merging the original clauses eliminating the complementary literals \( T \) and \( \neg T \):

\[
\begin{align*}
C_1 & = A_1 \lor \cdots \lor A_n \lor T \\
C_2 & = B_1 \lor \cdots \lor B_m \lor \neg T
\end{align*}
\]

\[
\therefore R : = A_1 \lor \cdots \lor A_n \lor B_1 \lor \cdots \lor B_m
\]
1. All sentences in the knowledge base (and the negation of the sentence we want to prove, the so-called conjecture) are conjunctively connected.

2. The resulting sentence is transformed into CNF represented by the set $S$ in set notation.

3. The resolution rule is applied to all possible pairs of clauses containing complimentary literals producing the resolvent $R$.

4. Repeated literals are removed from $R$.

5. If $R$ contains complimentary literals, it is discarded. Otherwise, $R$ is added to $S$, if it is not yet an element.

6. If the empty clause can be derived after an application of the resolution rule, we have proven contradiction. This can either mean that the knowledge base is inconsistent or that the negation of the sentence we tried to prove is unsatisfiable, i.e., the conjecture follows from the knowledge base.
In our example, we want to find a solution to the facts in our knowledge base:

\[ K := \{\{A, B, C\}, \{\neg A, B, C\}, \{\neg A, \neg C, \neg B\}, \{B, \neg C\}, \{\neg A, \neg B, C\}, \{A, C\}\} \]

\[ \{A, B\} \rightarrow \{\neg C\} =: g \]
\[ \{A, \neg C\} \rightarrow \{B\} =: h \]
\[ \{\neg A, B\} \rightarrow \{\neg C\} =: i \]
\[ \{\neg A, C\} \rightarrow \{\neg B\} =: j \]
\[ \{B, \neg C\} \rightarrow \{\neg A\} =: k \]
\[ \{A, \neg C\} \rightarrow \{B\} =: l \]
\[ \{\neg A, \neg B, C\} \rightarrow \{B\} =: m \]
\[ \{A, C\} \rightarrow \{\neg B\} =: n \]
\[ \{\neg A\} \rightarrow \{\neg C\} =: o \]
\[ \{\neg A, B\} \rightarrow \{\neg C\} =: p \]

In conclusion, we find that Brian and Colin are guilty, Austin is not.

We have to systematically try all combinations of clauses when searching for a solution since if any of them had resulted in an empty clause, we would have found that the knowledge base has no solution, i.e., it is inconsistent in itself.

Resolution: example
Finding solutions using a truth table

We can derive the same solution by means of a truth table:
Let us now try to prove whether Brad or Colin are culprits, so, we now have to take the CNF of all the facts from our knowledge base

\[
\begin{align*}
\emptyset & \leftarrow \{\neg f, \neg g\} \\
\neg f & := \{c\} \leftarrow \{i, f\} \\
\neg i & := \{c, \neg a\} \leftarrow \{b, q\}
\end{align*}
\]

and try to derive an empty clause:

\[
\begin{array}{c}
\gamma \\
\wedge \\
\beta
\end{array}
\begin{array}{c}
\{\{c, \neg a\}, \{b, \neg c\}\} \\
\Leftrightarrow (c \wedge b) \neg \Leftrightarrow \neg f
\end{array}
\]

(17)

and the CNF of the negated conjecture \( \neg f \), i.e.,

\[
\begin{array}{c}
\neg f \\
\wedge \\
\neg e \\
\wedge \\
\neg p \\
\wedge \\
\neg c \\
\wedge \\
\neg q \\
\wedge \\
\neg a
\end{array}
\begin{array}{c}
\{\{c\}, \{\neg a, b, c\}, \{\neg a, b, c\}, \{\neg a, b, c\}, \{\neg a, b, c\}, \{\neg a, b, c\}, \{\neg a, b, c\}\}
\end{array}
\]

In conclusion, we were able to prove that Brad or Colin are culprits.
Again, the same result can be found when consulting the truth table:

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>( K )</th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
<th>f</th>
<th>g</th>
<th>h</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Finding solutions using a truth table (cont.)
1st-order logic (aka as predicate logic) is an extension to propositional logic. Main difference is its additional use of predicates, functions and quantifiers. A quantifier is an operator defining the scope of variables:
- $\exists$ is the existential quantifier,
- $\forall$ is the universal quantifier.

A predicate returns boolean, a function non-boolean values.
1st-order logic: examples

example functions:
– + as in \( x + y \)
– any constant such as the numeral 1
– age(\( x \)) returning the age of the object \( x \)

example predicates:
– > as in \( x > y \)
– propositional variables – \( \top \) and \( \bot \)
– isStudent(\( x \)) returning \( \top \) iff \( x \) is a student
– any constant such as the numeral 1
– \( \forall y < x \)
– \( \forall y + x \)

example terms:

1. Any variable is a term.
2. Any expression \( f(t_1, \ldots, t_n) \), with the \( n \)-ary function symbol \( f \) and the terms \( t_1, \ldots, t_n \), is a term.

terms:

\( \forall y < x \)

 pred: isStudent (\( x \)) returning \( \top \) iff \( x \) is a student

\( \top \) and \( \bot \)

prop variables:

\( \forall y < x \)

\( \forall y + x \)

ex: pred:

\( \forall y < x \)

ex: func:

\( \forall y + x \)
1st-order logic: examples on how to translate natural language into formulas

(18) \[ \forall x (\text{isStudent}(x) \rightarrow \text{isSmart}(x)) \]

There is a smart student:

(19) \[ \exists x (\text{isStudent}(x) \land \text{isSmart}(x)) \]

All students are smart:

(21) \[ \forall x (\text{isStudent}(x) 
\rightarrow \exists y (\text{isStudent}(y) \land \text{loves}(x, y))) \]

Billy has one brother:

(22) \[ \exists x (\text{isBrotherOf}(x, \text{billy}) \land \forall y (\text{isBrotherOf}(y, \text{billy}) \rightarrow x = y)) \]
Modal logic extends the standards of formal logic with elements of modality:
– possibility (Kripke 1959: “possible worlds”; operator \(\diamond\))
– necessity (operator \(\Box\)).

Each of them can be represented by the other with negation:

\[
\diamond \neg \phi \leftrightarrow \neg \Box \neg \phi, \tag{23}
\]

\[
\Box \phi \leftrightarrow \neg \diamond \neg \phi. \tag{24}
\]

\[
\Diamond \exists x \phi \leftrightarrow \exists x \Diamond \phi. \tag{25}
\]

(Wittgenstein’s son)

\[
\exists x \Diamond \phi \leftrightarrow \Diamond \exists x \phi. \tag{26}
\]
• Prolog
• expert systems and dialog systems
• intelligent search and problem solving strategies
• logic and computer-assisted proof
Problem solving is the search for a solution in a given scenario.

Questions raised by search algorithms at runtime include:

- How do I estimate what is still missing?
- How good am I at the moment?
- How good am I at the moment?

Popular search families are:

- local search (e.g. hill climbing)
- depth-first search (DFS)
- breadth-first search (BFS)
- A* graph and tree traversal

Problem solving is the search for a solution in a given scenario.
Hill climbing is an iterative algorithm that
1. starts with an arbitrary solution \( x_0 \) to the problem with a performance \( f(x_0) \),
2. incrementally changes a single element of \( x \) resulting in \( x' \),
3. if the change improved the solution (i.e., \( (x')f < (x)f \)), then the solution is updated (\( x := x' \)), and the algorithm continues at Step 2.
4. if no further improvement can be produced, the algorithm stops.

Hill-climbers are well-suited for convex surfaces. They will converge to the global optimum.

Hill climbing is an iterative algorithm that
Hill climbing: local maxima

Hill climbing will find only local maxima. Hence, if $f(x)$ is not convex, it may not find the global optimum.

E.g., if the algorithm starts at a poor location in the following example, it may not converge to the global maximum:

$$f(x) = (z(7L \cdot I - x) + z(7L \cdot I - x))^{-a} + (z + x)^{-a} = (x, y, z)$$

Hence, if $f(x)$ is not convex, it may not find the global optimum.

Hill climbing will find only local maxima.

Stochastic hill climbing, random walks, or simulated annealing try to overcome this problem.

Suendermann Knowledge-Based Systems November 19, 2012
Hill climbing: ridges

Hill climbers adjust one vector element at a time. So, each step will move in an axis-aligned direction.

Hill climbing: ridges

Another problem is when the search space is flat around the current search position (plateau).

If \( f(x) \) features a narrow ridge ascending in a non-axis direction, the climber has to zig-zag.

If the ridge’s sides are very steep, the climber has to zig-zag.

If \( f(x) \) is differentiable, gradient descent methods can over-time to ascend.

Therefore, may take an unreasonable time to ascend, therefore, may take an unreasonable time to ascend.

Gradient descend methods can over-time to ascend.

Another problem is when \( f(x) \) is differentiable.

If the ridge’s sides are very steep, the climber has to zig-zag.

Hill climbing: ridges

Suendermann
Knowledge-Based Systems
November 19, 2012 27
Simulated annealing (SA) is a probabilistic heuristic to find the global optimum of $f(x)$.

Name and inspiration come from the annealing in metallurgy.

Each step of the SA algorithm replaces $x$ with a random nearby $x'$.

The randomization is based on a probability that depends on $-f(x) - f(x')$.

Due to the randomness of picking $x'$, the method can escape local optima.

SA does not guarantee to reach the global optimum but increases chances to do so.

The parameter $T$, a temperature gradually decreased during the process, affects the probability $e^{\frac{-f(x) - f(x')}{T}}$.

Simulated annealing
Graph traversal refers to a search algorithm visiting the nodes in a graph in a particular manner. Starting at a root node, all children are generated and added to an open list. If all children of a node are generated, the node gets removed from the open list and added to a closed list. Generation and expansion are performed until a goal node is found.
Graph traversal: 8 queens puzzle

Place 8 chess queens on a chessboard such that no two queens attack each other.

But only 92 solutions.

Possible arrangements:

\[
\binom{64}{8} = 4,426,165,368
\]

There are 4,426,165,368 possible arrangements.

Knowledge-Based Systems
November 19, 2012 31

Sundermann
Graph traversal: travelling salesman problem (TSP)

Given a list of cities and their pairwise distances, find the shortest possible tour visiting each city exactly once.

TSP is an NP-hard problem and belongs to the most intensively studied ones in optimization.

Suche zurückkehrt

A* Suendermann

Knowledge-Based Systems

November 19, 2012 32
Graph search algorithms

- Exhaustive search algorithms
  - depth-first search (DFS)
  - breadth-first search (BFS)

- Heuristic and statistical search algorithms
  - backtracking
  - A∗
  - best-first search
  - minimax algorithm
Exhaustive search algorithms: DFS vs. BFS
DFS: example

<table>
<thead>
<tr>
<th>OPEN (stack)</th>
<th>CLOSED</th>
</tr>
</thead>
<tbody>
<tr>
<td>4, 3, 2, 9, 8, 5, 7, 10</td>
<td>6, 10</td>
</tr>
<tr>
<td>4, 3, 2, 9, 8, 5, 7</td>
<td>10, 10</td>
</tr>
<tr>
<td>7, 10, 10</td>
<td>5, 10</td>
</tr>
<tr>
<td>8, 9, 5, 10, 2</td>
<td>4, 3, 2</td>
</tr>
<tr>
<td>9, 8, 5, 10, 4</td>
<td>3, 5, 10</td>
</tr>
<tr>
<td>3, 5, 10</td>
<td>4, 5, 10</td>
</tr>
<tr>
<td>4, 5, 10</td>
<td>1</td>
</tr>
</tbody>
</table>
Exhaustive search algorithms: modifications

- **Depth-limited search** – works exactly like DFS, but imposes a maximum limit on the depth of search. A solution is found when both instances hit an identical node.

- **Iterative deepening** – runs exactly like DFS, but imposes a maximum limit on the depth of the search. Each attempt increases the maximum depth until a solution is found.

- **Bidirectional search** – runs two instances of BFS, one from the initial node, one from the goal node. Reaching the maximum depth, it increases the depth limit with each iteration until a solution is found. Compared to pure BFS, the complexity of this algorithm can be significantly lower, e.g. \( O\left(\frac{b^d}{2}\right) \) rather than \( O(b^d) \) with the branching factor \( b \).
PATH(x): path to node x
- OPEN: open set
- CLOSED: closed set
- \(N(x)\): set of neighbor nodes of x
- visit nodes in the tree
- \(f(x)\): distance-plus-cost heuristic to determine the order in which to visit nodes in the tree
- \(h(x)\): a heuristic estimate of the distance to the goal
- \(g(x)\): cost from the starting node to the current node x
- \(c(x)\): cost from x to y
- \(T\): the set of goal nodes
- \(s\): starting node

Involved terms: •

A*: Terms
Algorithm:

1. **OPEN** = 0; **CLOSED** = ∅
2. If *x* ∈ **CLOSED**:
   - **OPEN** = **OPEN** + {x}; **CLOSED** = **CLOSED** - {x}
3. If *x* ∈ **OPEN** or *x* ∈ **CLOSED**
   - **PATH** = (λ) 最短路径
4. If *y* ∈ **OPEN** or *y* ∈ **CLOSED**
   - **OPEN** = **OPEN** + {y}; **CLOSED** = **CLOSED** - {y}
5. For each *y* in **N(x)**:
   - If *x* ∈ **T**
     - Solution found
   - Else
     - **OPEN** = **OPEN** + {x}; **CLOSED** = **CLOSED** - {x}
     - Select *x* from **OPEN** minimizing h(s)
     - g(s) = 0; OPEN = {}; CLOSED = ∅
     - Compute h(s) = h(s) + h(s)
     - If *x* ∈ **T**
       - Solution found
     - Else
       - **OPEN** = **OPEN** + {x}; **CLOSED** = **CLOSED** - {x}
     - **PATH** = (λ) 最短路径
6. No solution
\begin{array}{ccccccccc}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \\
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \\
\end{array}

\begin{array}{c}
(y(x)) \\
x \\
\end{array}

\textbf{Example}

\begin{center}
\includegraphics[width=\textwidth]{example.png}
\end{center}
A: example (cont.)

<table>
<thead>
<tr>
<th>Step</th>
<th>x</th>
<th>f(x)</th>
<th>g(x)</th>
<th>OPEN</th>
<th>CLOSED</th>
<th>N(x)</th>
<th>PATH(x)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>4</td>
<td>1</td>
<td>8</td>
<td>3,5,8,10</td>
<td>6,9</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>1</td>
<td>8</td>
<td>3,5,8,10</td>
<td>6,9</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>6</td>
<td>1</td>
<td>8</td>
<td>3,5,8,10</td>
<td>6,9</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>7</td>
<td>3</td>
<td>5,10</td>
<td>6,9</td>
<td>6,9</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>6</td>
<td>1</td>
<td>8</td>
<td>3,5,8,10</td>
<td>6,9</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>7</td>
<td>3</td>
<td>5,10</td>
<td>6,9</td>
<td>6,9</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>6</td>
<td>1</td>
<td>8</td>
<td>3,5,8,10</td>
<td>6,9</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>7</td>
<td>3</td>
<td>5,10</td>
<td>6,9</td>
<td>6,9</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>8</td>
<td>7</td>
<td>10</td>
<td>6,9</td>
<td>6,9</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>9</td>
<td>7</td>
<td>10</td>
<td>6,9</td>
<td>6,9</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>10</td>
<td>7</td>
<td>10</td>
<td>6,9</td>
<td>6,9</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>11</td>
<td>7</td>
<td>10</td>
<td>6,9</td>
<td>6,9</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>12</td>
<td>7</td>
<td>10</td>
<td>6,9</td>
<td>6,9</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>13</td>
<td>7</td>
<td>10</td>
<td>6,9</td>
<td>6,9</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>14</td>
<td>7</td>
<td>10</td>
<td>6,9</td>
<td>6,9</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>15</td>
<td>7</td>
<td>10</td>
<td>6,9</td>
<td>6,9</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>17</td>
<td>16</td>
<td>7</td>
<td>10</td>
<td>6,9</td>
<td>6,9</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>17</td>
<td>7</td>
<td>10</td>
<td>6,9</td>
<td>6,9</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>19</td>
<td>18</td>
<td>7</td>
<td>10</td>
<td>6,9</td>
<td>6,9</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>19</td>
<td>7</td>
<td>10</td>
<td>6,9</td>
<td>6,9</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>21</td>
<td>20</td>
<td>7</td>
<td>10</td>
<td>6,9</td>
<td>6,9</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>22</td>
<td>21</td>
<td>7</td>
<td>10</td>
<td>6,9</td>
<td>6,9</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>23</td>
<td>22</td>
<td>7</td>
<td>10</td>
<td>6,9</td>
<td>6,9</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>24</td>
<td>23</td>
<td>7</td>
<td>10</td>
<td>6,9</td>
<td>6,9</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>25</td>
<td>24</td>
<td>7</td>
<td>10</td>
<td>6,9</td>
<td>6,9</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>26</td>
<td>25</td>
<td>7</td>
<td>10</td>
<td>6,9</td>
<td>6,9</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>27</td>
<td>26</td>
<td>7</td>
<td>10</td>
<td>6,9</td>
<td>6,9</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>28</td>
<td>27</td>
<td>7</td>
<td>10</td>
<td>6,9</td>
<td>6,9</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>29</td>
<td>28</td>
<td>7</td>
<td>10</td>
<td>6,9</td>
<td>6,9</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>30</td>
<td>29</td>
<td>7</td>
<td>10</td>
<td>6,9</td>
<td>6,9</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>31</td>
<td>30</td>
<td>7</td>
<td>10</td>
<td>6,9</td>
<td>6,9</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>32</td>
<td>31</td>
<td>7</td>
<td>10</td>
<td>6,9</td>
<td>6,9</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>33</td>
<td>32</td>
<td>7</td>
<td>10</td>
<td>6,9</td>
<td>6,9</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>34</td>
<td>33</td>
<td>7</td>
<td>10</td>
<td>6,9</td>
<td>6,9</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>35</td>
<td>34</td>
<td>7</td>
<td>10</td>
<td>6,9</td>
<td>6,9</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>36</td>
<td>35</td>
<td>7</td>
<td>10</td>
<td>6,9</td>
<td>6,9</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>37</td>
<td>36</td>
<td>7</td>
<td>10</td>
<td>6,9</td>
<td>6,9</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>38</td>
<td>37</td>
<td>7</td>
<td>10</td>
<td>6,9</td>
<td>6,9</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>39</td>
<td>38</td>
<td>7</td>
<td>10</td>
<td>6,9</td>
<td>6,9</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>40</td>
<td>39</td>
<td>7</td>
<td>10</td>
<td>6,9</td>
<td>6,9</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>41</td>
<td>40</td>
<td>7</td>
<td>10</td>
<td>6,9</td>
<td>6,9</td>
<td>7</td>
<td></td>
</tr>
</tbody>
</table>

**Notes:**
- The table shows the progression of steps in a process, with the values for each step listed.
- The columns represent different aspects of the process, such as the variable values and conditions.
- The process appears to be iterative, with each step building upon the previous one.

**Example:**

- **Step 1:** Initial values are set for the process.
- **Step 2:** Iteration starts with the first set of values.
- **Step 3:** Values are updated, and the process continues iteratively.

The table continues with similar iterations, each building on the previous one.
Minimax search

Originally formulated for two-player game theory.

Each game situation is a state, i.e. a node in a graph.

- The opponent's move minimizes one's winning probability.
- One's move maximizes one's winning probability.

Minimax principle:

Assumption: The opponent always chooses the best-possible move.

- Originally formulated for two-player game theory.
Can minimax be applied to chess?

Not without further assumptions since the state space is too large.

Possible solutions:
- limited search depth,
- heuristic cost/reward functions.
- Heuristic cost/reward functions do not reward 1/0 for winning/losing but try to find a reasonable approximation.

Minimax heuristic
Minimax algorithm: example
- Prolog
- Expert systems and dialog systems
- Intelligent search and problem solving strategies
- Logic and computer-assisted proof
Expert systems: introduction

An expert system (XPS) is a computer program emulating the decision-making of human experts. XPSs are one of the most popular applications of artificial intelligence.

At runtime, an XPS has to communicate with a human user, so it also requires human-machine interfaces for in- and output.

Accordingly, the two main components of an XPS are

- the knowledge base
- the inference engine

In contrast to conventional software, an XPS is designed to solve complex problems by reasoning about knowledge.
Expert systems: architecture

Knowledge base:

- Semantic generation
- Inference engine
- Semantic analysis
- Human-machine interface

From human to human:

- Knowledge base
- Semantic generation
- Inference engine
- Semantic analysis
- Human-machine interface

From human to human:
Mycin was designed to identify bacteria causing severe infections (e.g., meningitis). Mycin also recommended medication (antibiotics) adjusted to the patient’s characteristics. Based on the PhD thesis of a student at Stanford’s medical school in the early 1970s, the knowledge base consisted of about 600 rules established with the help of medical experts. A performance test resulted in 69% good recommendations outperforming infectious disease experts from Stanford’s medical school. One of the reasons is that the reliability of medical decisions made in the U.S. is particularly crucial in the real world. Mycin was not released to the real world.
Deep Blue

Deep Blue is a chess-playing computer by IBM. On May 11, 1997, Deep Blue won a six-game match against Garry Kasparov. The evaluation function contained multiple parameters tuned on 700,000 grandmaster games. The evaluation function was written in C under AIX and based on brute-force computing power. Deep Blue won a six-match against Garry Kasparov. Based on brute-force computing power, Deep Blue won a six-match against Garry Kasparov.

Pic: [Source: https://flickr.com/photos/22453761@N00/592436598/ - Author: James the photographer - License: Creative Commons Attribution 2.0 Generic]
Watson is an AI computer system from IBM for question answering. It combines applications of – machine learning, – NLP, – information retrieval, – knowledge representation, – reasoning.

To showcase its abilities, in February 2011, Watson competed on the show Jeopardy! against the human champions and won.

During the quiz, Watson had no access to the Internet.

It had access to 200M pages of structured and unstructured data, amounting to 4TB, including a copy of the entire Wikipedia.

Hardware consisted of – 90 IBM Power 750 servers with 2880 processors and 16TB of RAM.

Watson is an AI computer system from IBM for question answering.
Expert systems: knowledge base

Traditionally, the knowledge base stores knowledge in a computer-readable manner (e.g. using SQL, logical formulas).

In the case of an XPS, the knowledge base can be composed of:

- Probabilistic models (automatically) learned from structured and unstructured data (e.g., statistics of survival rates given patient’s symptoms and medication, winning probabilities given a game scenario, caller behavior and state etc.);
- Structured data derived from encyclopedias, directories, catalogues, or the WWW (e.g., Wikipedia, etc.);
- Unstructured data (as provided by FAQs, scientific articles, Wikipedia, etc.);
- Heterogeneous sources such as experts—depending on the XPS’s domain;
- Structured data derived from encyclopedias, directories, catalogues, or the WWW (e.g., statistics of survival rates given patient’s symptoms and medication, winning probabilities given a game scenario, caller behavior and state etc.);
- Structured data derived from encyclopedias, directories, catalogues, or the WWW (e.g., Wikipedia, etc.);
- Unstructured data (as provided by FAQs, scientific articles, Wikipedia, etc.);
- Heterogeneous sources such as experts—depending on the XPS’s domain;

and (e.g. statistics of survival rates given patient’s symptoms and medication, winning probabilities given a game scenario, caller behavior and state etc.);

In the case of an XPS, the knowledge base can be composed of:

- Computer-readable manner (e.g. using SQL, logical formulas).

Traditionally, the knowledge base stores knowledge in a computer-readable manner (e.g. using SQL, logical formulas).

In the case of an XPS, the knowledge base can be composed of:

- Probabilistic models (automatically) learned from structured and unstructured data (e.g., statistics of survival rates given patient’s symptoms and medication, winning probabilities given a game scenario, caller behavior and state etc.);
- Structured data derived from encyclopedias, directories, catalogues, or the WWW (e.g., Wikipedia, etc.);
- Unstructured data (as provided by FAQs, scientific articles, Wikipedia, etc.);
- Heterogeneous sources such as experts—depending on the XPS’s domain;
- Structured data derived from encyclopedias, directories, catalogues, or the WWW (e.g., Wikipedia, etc.);
- Unstructured data (as provided by FAQs, scientific articles, Wikipedia, etc.);
- Heterogeneous sources such as experts—depending on the XPS’s domain;
If something is living then it is mortal.

(turn into 1st-order logic)

IF something is living THEN it is mortal.

(turn into SQL)

If somebody's age is known then his birth date is today's date minus his age.

(IF the identity of the germ is not known with certainty AND the germ is
gram-positive AND the morphology of the organism is "rod" AND the germ is
aerobic THEN there is a strong probability (0.8) that the germ is of type
enterobacteraeae. (Mycin rule))
The inference engine evaluates rules and/or statistics provided by the knowledge base to produce reasoning. It can be based on (a combination of):

- propositional logic (0th-order XPS)
- other types of logic (predicative, modal, temporal, fuzzy)
- classification (e.g., decision trees)
- regression
- batch (all input variables for a query are given at once)
- conversational (input variables are provided one after the other, this way, non-salient variables can be skipped) (dialog systems)

In general, an inference engine can run in two modes:

- batch
- conversational (e.g., decision trees)

Knowledge base to produce a reasoning.

The inference engine evaluates rules and/or statistics provided by the knowledge base.
Spoken dialog systems: architecture

Knowledge-based systems

November 19, 2012

55

Suendermann
An example call flow...
In commercial spoken dialog systems, call flows are built by call flow designers. They implement a predefined business logic.

Manual vs. automatic design

Manual vs. automatic design

Manual vs. automatic design

Manual vs. automatic design

Manual vs. automatic design

Manual vs. automatic design

Manual vs. automatic design

Manual vs. automatic design

Manual vs. automatic design

Manual vs. automatic design

Manual vs. automatic design

Manual vs. automatic design

Manual vs. automatic design

Manual vs. automatic design

Manual vs. automatic design

Manual vs. automatic design

Manual vs. automatic design

Manual vs. automatic design

Manual vs. automatic design

Manual vs. automatic design

Manual vs. automatic design

Manual vs. automatic design

Manual vs. automatic design

Manual vs. automatic design

Manual vs. automatic design

Manual vs. automatic design

Manual vs. automatic design

Manual vs. automatic design

Manual vs. automatic design

Manual vs. automatic design

Manual vs. automatic design

Manual vs. automatic design

Manual vs. automatic design

Manual vs. automatic design

Manual vs. automatic design

Manual vs. automatic design

Manual vs. automatic design

Manual vs. automatic design

Manual vs. automatic design
Call flow and business logic

... an example call flow...
Call flow and business logic (cont.)

...and the underlying business logic table:

<table>
<thead>
<tr>
<th>Destination</th>
<th>Amount</th>
<th>Account Type</th>
<th>Service Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>give checking balance</td>
<td>$x$</td>
<td>checking</td>
<td>transfer</td>
</tr>
<tr>
<td>give savings balance</td>
<td>$x$</td>
<td>savings</td>
<td>transfer</td>
</tr>
<tr>
<td>transfer checking</td>
<td>savings</td>
<td>checking</td>
<td>balance</td>
</tr>
<tr>
<td>transfer savings</td>
<td>checking</td>
<td>savings</td>
<td>balance</td>
</tr>
</tbody>
</table>
Real-world call flows can be very complex (1000s of nodes and transitions).

- Columns of the business logic table represent questions and routing destinations (or final call flow action).
- Rows contain individual answers and destinations.
- An additional column contains the probability with which every row is visited.
- Without loss of generality, we consider a (sub-)call flow to be of a question-answer-destination type.

Without loss of generality, we can consider a (sub-)call flow to be of a question-answer-destination type.

- Columns of the business logic table can be represented by individual business logic tables.
- Complex call flows can be broken down in sub-call flows each of which transitions.

- Real-world call flows can be very complex (1000s of nodes and transitions).
Call flow and business logic (cont.)

Consider the following table for a general business logic:

<table>
<thead>
<tr>
<th>( p_n )</th>
<th>( d_n )</th>
<th>( a_{n1} )</th>
<th>( a_{n2} )</th>
<th>( a_{n3} )</th>
<th>( a_{n4} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( d_1 )</td>
<td>( a_{11} )</td>
<td>( a_{12} )</td>
<td>( a_{13} )</td>
<td>( a_{14} )</td>
<td></td>
</tr>
<tr>
<td>( d_2 )</td>
<td>( a_{21} )</td>
<td>( a_{22} )</td>
<td>( a_{23} )</td>
<td>( a_{24} )</td>
<td></td>
</tr>
<tr>
<td>( \ldots )</td>
<td>( \ldots )</td>
<td>( \ldots )</td>
<td>( \ldots )</td>
<td>( \ldots )</td>
<td></td>
</tr>
</tbody>
</table>

- \( p_n \) is the prior probability of a call ending at \( d_n \).
- \( d_n \) is the \( n \)th routing destination.
- \( a_{nm} \) is an answer to \( Q_m \).
- \( D_n \) is the \( n \)th routing destination.
- \( P_n \) is the prior probability of a call ending at \( d_n \).

This is a general business logic table:

Call flow and business logic (cont.)
Why does the order of questions matter?

In an example, we want to determine which of these modem types a caller has:

1. black Ambit
2. white Ambit
3. black Arris

A. Is your modem black or white?
B. Do you have an Ambit or an Arris modem?

The system designer considers the two questions:

- In an example, we want to determine which of these modem types a caller has.
Why does the order of questions matter? (cont.)

In our example, $x$ has two possible values ($x_1 = 1$ and $x_2 = 2$); i.e.,

Our random variable $x$ is the number of asked questions.

\[
\mathbb{E} = \sum_i x_i p(x_i)
\]

One way to estimate this number is to use the definition of the expected value.

Here, optimality means that the expected number of questions asked is minimal.

Since $A=\text{white}$ and $B=\text{Arris}$, the optimal order is $\text{A, B vs. B, A}$.
Why does the order of questions matter? (cont.)

Consequently, the optimal order in this example is A,B.

\[
E(A', B) = 1.6; \quad E(A', B) = 1.7.
\]

This results in

\[
E(A, B) = 1.6; \quad E(A, B) = 1.7.
\]

As an example, assume

\[
p(1) = 0.3; \quad p(2) = 0.4; \quad p(3) = 0.3.
\]

Consequently, the optimal order is minimal, thereby determining the optimal order.

\[
E(A', B) = 1.6; \quad E(A', B) = 1.7.
\]

Depending on the specific values of \(p(2)\) and \(p(3)\) either \(E(A', B)\) or \(E(A', B)\) is minimal, thereby determining the optimal order.

In case of the order A,B, we ask one question with the probability \(p(2)\), and two questions with \(1 - p(2)\), i.e.,

\[
(d) - 2 = ((d) - 1) \cdot 2 + (d) \cdot 1 = (B', A)E
\]

Similarly, for the order B,A, we get

\[
(d) - 2 = ((d) - 1) \cdot 2 + (d) \cdot 1 = (B', A)E
\]
A call flow resembles a decision tree.

We can use well-established machine learning techniques.

To find the most relevant questions, i.e. the ones providing maximum information gain, let's use the information gain measure:

\[ I(D; A_m) = H(A_m) + H(D) - H(A_m, D) \]

where \( A_m \) are the distinct destinations in the currently processed business logic table.

At every node in the call flow, we determine which question leads to the maximum information gain:

\[ \hat{m} = \text{arg max}_{m=1,\ldots,M} I(D; A_m) \]

maximum information gain.

Shannon's entropy defined as:

\[ H(D) = -\sum_{\delta=1}^{\Delta} P(\delta) \log_2 P(\delta) \]

where \( \delta \in \{1, \ldots, \Delta\} \) are the distinct destinations in the currently processed business logic table.

At every node in the call flow, we determine which question leads to the maximum information gain:

\[ \hat{m} = \text{arg max}_{m=1,\ldots,M} I(D; A_m) \]
Automatic call flow design: an experiment

We took the business logic table from a mature call routing application processing about 4M calls per month.

Based on call logs of an entire month, the probabilities $P_n$ were estimated. We took the business logic table from a mature call routing application.

| $\Delta = 20$ | number of distinct destinations |
| $N = 31$ | number of rows |
| $M = 4$ | number of questions |
| 3,868,014 | number of calls |

Experiment parameters:

- Estimated.
- Processing about 4M calls per month.
- Based on call logs of an entire month, the probabilities $P_n$ were estimated.
Automatic call flow generation with maximum information gain strategy resulted in $M = 2.87$.

- 30% reduction of average number of asked questions
- Possible savings of five- to six-figure US$ per month
- Possible savings of five- to six-figure US$ per month

The original app asked 4 questions: $M = 4$. The original app asked 4 questions: $M = 4$. 

(– modifiers (credit card, pay-per-view, digital TV conversion, etc.)
– actions (cancel, schedule, make a payment, etc.)
– product (Internet, cable TV, telephone)
– service type (orders, billing, technical support, etc.)

Automatic call flow design: an experiment (cont.)
A reward function

The main argument for using commercial spoken dialog systems is to replace the human agent to save costs.

Can we quantify the savings? And if so, what can we do to optimize them?

Consequently, the overall savings/reward is

\[ R = \frac{W}{W_A}  \]

The per-time-unit cost is \( \frac{W}{T} \).

On the other hand, automated calls produce costs (hosting, licensing or time is money: From \( S \) to \( R \)).

Conversely, an average cost \( \frac{W}{W_A} \) associated with a call successfully handled by a human agent.

Consider an average cost \( \frac{W}{W_A} \) associated with the automation rate \( A \).

Principally, an application’s performance is determined by the fraction of calls completed without agent intervention (the automation rate \( A \)).

The trade-off parameter is

\[ T \]

\[ S = \frac{W}{W_A} A - \frac{W}{T} T \]
Adaptation and optimization

– dialog management

– speech recognition and understanding

adaptation and optimization:

This talk is about two components of a spoken dialog system subject to a high saturation point.

This is to continually increase performance over time (or keep it at a high saturation point).

optimize spoken dialog systems.

To avoid this situation, there are several techniques to adapt and optimize and automatic rate falls below $\frac{V_t}{L}$, saving turns negatively (i)

When automatic rate falls below $\frac{V_t}{L}$, saving turns negative (i)
Speech recognition and understanding problems cause:
- poor user experience.
- going down the wrong path leading to a dead end.
- escalating to a human upon reaching a max number of "speech errors".
- misunderstand human speech.

The major criticism on spoken dialogue systems is their tendency to
Commercial applications use rule-based grammars for directed dialog:

- Please tell me the reason you are calling about today.
- Do you have more than one TV at home?
- What brand of modem do you have?
- Open prompt: callers are invited to use their own expressions

Practice: developers feel statistical grammars are out of their control
- Lack of knowledge and tools to build statistical grammars
- Data for statistical grammars initially unavailable

Rule-based grammars
The curse of the unexpected

Grammars are designed to match prompts – Do you have more than one TV at home? (yes | yeah | yup | no | nope)
– What brand of modem do you have? (Motorola | Toshiba | Sony | ...)

Unfortunately, a significant portion of users speak out-of-grammar.

- 90% of calls in grammar
- 10% of calls out-of-grammar
- UNIQUE CALLER EXPRESSIONS IN GRAMMAR
- 90% of calls no-match recognition errors
- 10% of calls rule based

The curse of the unexpected
Grammar tuning

- Conventionally, commercial tuning is performed sporadically by “speech scientists” looking at small samples of transcribed calls.

- Manually tuned grammars cannot perform as well as statistical ones (certain conditions apply).

- Manual tuning is expensive and does not scale.

  - Changing, adding, removing rules to match observed data
  - Cannot systematically tune all grammars in large applications
  - Cannot take advantage of large amounts of data

- Manually tuned grammars cannot ever perform as well as statistical ones (certain conditions apply).

- Convientionally, commercial tuning is performed sporadically (performed)
Building statistical language models and classifiers

TRANSCRIPTIONS

ANNOTATIONS
Continuous Improvement Cycle
Data conditioning process

- Completeness
- Correlation
- Consistency
- Confusion

- Reduce confusion across classifier categories
- Similar utterances need to be annotated consistently
- Consistency across multiple annotators
- Correlation analysis guarantees that utterances are annotated
- Transcribed/annotated data to match the distribution

C7 data conditioning process
enure statistical significance

– minimum training, development, and test corpus sizes are required to

Corpus size

– minimize out-of-grammar utterance by adding new semantic

Coverage

– Rule based grammars, if available, need to be congruent with

Conformance

C7 data conditioning process (cont.)
Continuous improvement of semantic accuracy
<table>
<thead>
<tr>
<th></th>
<th>True Total (Sep. 2008)</th>
<th>True Total (Jun. 2008)</th>
</tr>
</thead>
<tbody>
<tr>
<td>%</td>
<td>90.5%</td>
<td>78.0%</td>
</tr>
<tr>
<td>#nodes</td>
<td>2,021</td>
<td></td>
</tr>
<tr>
<td>#calls</td>
<td>533,343</td>
<td></td>
</tr>
<tr>
<td>#utterances</td>
<td>2,184,203</td>
<td></td>
</tr>
</tbody>
</table>

![Diagram](image-url)
Equation 35 (R = T_A - T) suggests that R can be increased by decreasing call duration. This holds true for automated and non-automated calls. However, non-automated calls can be shortened aggressively by escalating to an agent as early as possible. Escalators can be based on:

1. Manual rules (unsolicited agent requests, speech recognition problems, situations the system does not know how to handle, etc.),
2. The probability of the call ending unsuccessfully,
3. Events, situations the system does not know how to handle, etc.

We call an algorithm that deliberately escalates calls based on its opinion about the call outcome Escalator. Escalators can be based on:

1. Manual rules (unsolicited agent requests, speech recognition problems, situations the system does not know how to handle, etc.),
2. The probability of the call ending unsuccessfully,
3. Events, situations the system does not know how to handle, etc.

This holds true for automated and non-automated calls.
Features used to estimate Escalator probabilities can be based on the dialog history including:

- transitions taken
- textual and acoustic speech input
- acoustic and semantic confidence scores
- transitions taken
- number of no-match, no-input, or dis-confirmation
- etc.

An example implementation is described in [Levin and Pieraccini, 2006] and another example is based on pruning the call flow:

An example implementation is described in [Levin and Pieraccini, 2006].
### Escalator Example

<table>
<thead>
<tr>
<th>Time</th>
<th>#calls (tokens)</th>
<th>#nodes (types)</th>
<th>#nodes pruned</th>
<th>#calls (tokens) pruned</th>
</tr>
</thead>
<tbody>
<tr>
<td>13.3s</td>
<td>45,631</td>
<td>847</td>
<td>176</td>
<td></td>
</tr>
<tr>
<td>196.8s</td>
<td>5,000s</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>183.5s</td>
<td>w/o pruning</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>w/ pruning</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>176</td>
<td>847</td>
<td>45,631</td>
<td></td>
<td></td>
</tr>
<tr>
<td>847</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>45,631</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The graph shows the number of pruned nodes over time, with a significant decrease after pruning. The table details the call counts, node counts, and pruned nodes for different time intervals.
Then we route certain portions of traffic to each of the contender paths.

To find out which strategy is best, we can implement all of the above.

... –

- What is the best recovering strategy after a no-match?
- When do I time out?
- What is the ideal voice activity detection sensitivity?
- How much time should I wait until I offer a backup menu?
- Or a y/n question followed by an open prompt with examples?
- Is directed dialog or open prompt better in this context?

There can be 1000 things impacting automation, e.g.

What can we do to boost automation rate?

Escalator focuses on reducing handling time.
Alternative 1
Alternative 2
Alternative 3
Randomizer

Randomization
Weights

(cont.)
As shown in [Suendermann et al., 2010], the amount of traffic hitting path $A$ should be the winning probability $p(A)$. This approach maximizes the accumulated reward.

As shown in [Suendermann et al., 2010], the amount of traffic hitting path $A$ should be the winning probability $p(A)$. This approach maximizes the accumulated reward.

<table>
<thead>
<tr>
<th>#calls (tokens)</th>
<th>( T )</th>
<th>( H ) after contending</th>
<th>( H ) baseline</th>
</tr>
</thead>
<tbody>
<tr>
<td>38,004</td>
<td>29.4s</td>
<td>( \forall R )</td>
<td>229.9s</td>
</tr>
<tr>
<td>5,000s</td>
<td>253.4s</td>
<td></td>
<td>282.9s</td>
</tr>
</tbody>
</table>

Integral over Contender probability distributions is required.

In case of an $n$-way split, the numerical solution of an $n$-dimensional integral over Contender probability distributions is required.
Outline

- Prolog
- Expert systems and dialog systems
- Intelligent search and problem solving strategies
- Logic and computer-assisted proof
Prolog

Prolog (programming in logic) is a programming language associated with artificial intelligence as well as computer linguistics.

In accordance with the architecture of XPS, the main components of logical programming are:

1. A knowledge base (facts and rules),

Logical programming are advantageous because this job is done by the inference engine, and one does not have to develop an algorithm to solve the problem. Instead, we describe the problem by means of logical formulas.

Advantage of logical programming is that one can be obtained through distributions as well as Cygwin (http://cygwin.com) or can be obtained as part of the major Linux distributions.


Prolog (programming in logic) is associated with artificial intelligence as well as computer linguistics.
Facts and rules

Facts are atomic formulas with the Prolog syntax \( p(t_1, \ldots, t_n) \) (36).

All the variables in facts are universally bound, i.e., Eq. 36 represents the logical formula \( \forall x_1, \ldots, x_m (p(t_1, \ldots, t_n)) \).

Rules are conditional propositions with the Prolog syntax (38).

Rules generally require formulas to be given as Horn clauses.

(39) \[ (A \leftarrow \bigwedge x_1, \ldots, x_m (B_1, \ldots, B_n)) \]

This represents the formula \( A \leftarrow \bigwedge B_1, \ldots, B_n \).

Again, all the variables in rules are universally bound, so, Eq. 38 represents the logical formula \( \forall x_1, \ldots, x_m (B_1 \land \ldots \land B_n \rightarrow A) \).

This generally requires formulas to be given as Horn clauses.

(37) \[ (\forall x_1, \ldots, x_m (p(t_1, \ldots, t_n))) \]

Features the predicate \( p \) and the terms \( t_1, \ldots, t_n \).

(36) \[ (\forall x_1, \ldots, x_m (p(t_1, \ldots, t_n))) \]

Features the atomic formulas with the Prolog syntax.
Some conventions

- The first character of variables is a capital letter or an underscore.
- The first character of predicates or functions is a lower-case letter.
- The predicate true represents validity.
- The symbols +, (or alternatively, not ( ) ) is the negation operator.
- The symbol \% is used for comments.
- The symbols , and ; is used for conjunction and disjunction, respectively.
- The symbols <, >, =, =<, >=, \=, == are predicate symbols you can use in infix notation. Note that == tests for equality, and \= tests for inequality.
- The symbol + (or, alternatively, not() ) is the unification operator.
- The symbols , and ; is used for conjunction and disjunction, respectively.
- The symbols +, -, *, / are function symbols you can use in infix notation.
- The predicate true represents validity.

November 19, 2012

Suendermann
The following derivation shows that disjunctions in Prolog rules are effectively no additional feature:

\[
\begin{align*}
A & : \neg B_1 \\
\cdots \\
A & : \neg B_1 \\
(A \land \neg B_1) & \lor \cdots \lor (A \land \neg B_n) \\
A & \lor (\neg B_1 \land \cdots \land \neg B_n) \\
A & \iff A : \neg B_1 \\
\end{align*}
\]

Therefore, disjunctions in Prolog rules are effectively no additional feature. The following diversion shows that disjunctions in Prolog rules are:

- On Prolog's disjunction

Suendermann Knowledge-Based Systems November 19, 2012 93
– Colin is a professor.
– Colin is a computer scientist.
– Brad is a student.
– Alan is a student.
– Computer scientists are crazy.
– Whoever is computer scientist and professor is powerful.
– Whoever is smart is powerful.
– All students are smart.

Let us now consider a realistic example:
An example (cont.)

This is the respective Prolog code (see student.pl in the auxiliary package kbs*_zip):

1 smart(X):-student(X).
2 powerful(X):-smart(X).
3 powerful(X):-cs(X),prof(X).
4 crazy(X):-cs(X).
5 student(alan).
6 student(brad).
7 cs(colin).
8 prof(colin).

This is the respective Prolog code (see student_pl in the auxiliary package kbs*_zip):
We want to find out whether there is a powerful and crazy individual. The respective logical formula is

$$\exists x \text{ (powerful} (x) \land \text{crazy} (x))$$.

The respective logical formula is

$$\exists x \text{ (powerful} (x) \lor \text{crazy} (x))$$.

In order to find out, we first launch Prolog with the command

```
pl
```

To load our knowledge base, we type

```
consult(student).
```

and get the command prompt

```
p1
```

To load our knowledge base, we type

```
consult(student).
```

Now, we can use the Prolog syntax of Eq. 41 to check the validity of our conjecture:

```
powerful(X), crazy(X).
```

We want to find out whether there is a powerful and crazy individual.
An example (cont.)

We obtain the response \( X = \text{colin} \) telling us that Colin is a powerful and crazy individual.

In order to identify other potential candidates, we type the following command:

\[ \text{No} \]

resulting in the response \( \text{No} \).

We obtain the response \( X = \text{colin} \) which indicates that there are no more solutions to the problem.
Prolog's inference algorithm works as follows:

1. Search (in order of appearance) all the rules \( A \) in \( P \), for which there exists a unifier \( \mu \) such that

\[
\begin{cases}
\text{true} = \mu(1, O) \\
\text{otherwise}
\end{cases}
\]

where \( \mu(1, O) \) is the most general unifier (mgu) of the query \( Q \) and the head of the rule.

Here, facts are expanded to rules by

\[ A : \text{true} \leftrightarrow A \]

and a query of the form

\[ w, \ldots, w \cdot B : \text{true} \leftrightarrow R \]

are given the Prolog program \( P \) consisting of a number of rules of the form

\[ \text{true} \]

We are given the Prolog program \( P \) consisting of a number of rules of the form

\[ w, \ldots, w \cdot B : \text{true} \leftrightarrow R \]

The inference algorithm works as follows:

1. Search (in order of appearance) all the rules \( A \) in \( P \), for which there exists a unifier \( \mu \) such that

\[
\begin{cases}
\text{true} = \mu(1, O) \\
\text{otherwise}
\end{cases}
\]

where \( \mu(1, O) \) is the most general unifier (mgu) of the query \( Q \) and the head of the rule.

Here, facts are expanded to rules by

\[ A : \text{true} \leftrightarrow A \]

and a query of the form

\[ w, \ldots, w \cdot B : \text{true} \leftrightarrow R \]

are given the Prolog program \( P \) consisting of a number of rules of the form

\[ w, \ldots, w \cdot B : \text{true} \leftrightarrow R \]
Prolog’s inference algorithm (cont.)

2. In case there are multiple such rules,
   a) select the first rule (in order of appearance),
   b) set a choice point (CP) to perform a different selection at this point in case it becomes necessary at a later moment.

3. Here, two cases are distinguished:

   a) set a choice point (CP) to perform a different selection at this point in case it becomes necessary at a later moment.
   b) select the first rule (in order of appearance);

   3. Here, two cases are distinguished:

   a) This means success, and Prolog returns the last non-empty µ.
   b) Otherwise, we recursively continue with the query.

   (46)

Negation is implemented in Prolog as negation as failure.

Reversing the replacements G := G µ accordingly.

If we do not find a solution, we return to the last choice point.

Otherwise, we recursively continue with the query.

If we find a solution, we return to the last choice point reversing the replacements G := G µ accordingly.

Negation is implemented in Prolog as negation as failure.

I.e., if Q1 in I is of the syntax not (Q1), the algorithm tries to prove Q1.

Negation is implemented in Prolog as negation as failure.

\[ G = B_1 \mu, \ldots, B_m \mu, Q_2 \mu, \ldots, Q_n \mu. \]
Let us sketch a proof of Prolog's inference rule.

For the sake of simplicity, we limit ourselves to propositional logic and

assume a Prolog rule

Let us sketch a proof of Prolog’s inference rule.
Let us prove that this inference rule is a tautology:

\( \Downarrow \Leftrightarrow \land \)

\( \mathcal{R} \land \mathcal{B} \land \mathcal{O} \land \mathcal{A} \Leftrightarrow \top \)

\( (\mathcal{O} \land \mathcal{A}) \lor (\mathcal{A} \land \mathcal{O} \land \mathcal{B}) \lor (\mathcal{O} \land \mathcal{A} \land \mathcal{B}) \Leftrightarrow \top \)

\( (\mathcal{A} \land \mathcal{B}) \lor (\mathcal{A} \land \mathcal{B}) \Leftrightarrow \top \)

\( \mathcal{R} \lor \mathcal{O} \lor \mathcal{A} \lor \mathcal{B} \lor \mathcal{O} \lor \mathcal{A} \lor \mathcal{B} \lor \mathcal{O} \lor \mathcal{A} \lor \mathcal{B} \Leftrightarrow \top \)

\( \mathcal{R} \lor \mathcal{O} \lor \mathcal{A} \lor \mathcal{B} \lor (\mathcal{O} \land \mathcal{A}) \lor (\mathcal{O} \land \mathcal{A}) \lor (\mathcal{A} \land \mathcal{B}) \lor (\mathcal{A} \land \mathcal{B}) \Leftrightarrow \top \)

\( \mathcal{R} \lor \mathcal{O} \lor \mathcal{A} \lor \mathcal{B} \lor (\mathcal{O} \land \mathcal{A}) \lor (\mathcal{O} \land \mathcal{A}) \lor (\mathcal{A} \land \mathcal{B}) \lor (\mathcal{A} \land \mathcal{B}) \Leftrightarrow \top \)

\( \mathcal{R} \lor \mathcal{O} \leftarrow (\mathcal{R} \lor \mathcal{B}) \lor (\mathcal{O} \leftarrow \mathcal{A}) \lor (\mathcal{A} \leftarrow \mathcal{B}) \Rightarrow \top \)

\( \mathcal{R} \lor \mathcal{O} \leftarrow (\mathcal{R} \lor \mathcal{B}) \lor (\mathcal{O} \leftarrow \mathcal{A}) \lor (\mathcal{A} \leftarrow \mathcal{B}) \Rightarrow \top \)
<table>
<thead>
<tr>
<th>U</th>
<th>(X)(powerful) : true</th>
<th>(X)(powerful) : true</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>(X)</td>
<td>(X) : (X)(crazy) : (X)(smart) : (X)(powerful) : (X)(student)</td>
<td>(X) : (X)(crazy) : (X)(smart) : (X)(powerful) : (X)(student)</td>
<td>9</td>
</tr>
<tr>
<td>[brad ← X]</td>
<td>(X)(crazy) : (X)(student)</td>
<td>(X)(crazy) : (X)(student)</td>
<td>8</td>
</tr>
<tr>
<td>[]</td>
<td>(X)(crazy) : (X)(student)</td>
<td>(X)(crazy) : (X)(student)</td>
<td>7</td>
</tr>
<tr>
<td>[brad ← X]</td>
<td>(X)(crazy) : (X)(student)</td>
<td>(X)(crazy) : (X)(student)</td>
<td>6</td>
</tr>
<tr>
<td>U</td>
<td>(X)(powerful) : true</td>
<td>(X)(powerful) : true</td>
<td>5</td>
</tr>
<tr>
<td>(X)</td>
<td>(X) : (X)(crazy) : (X)(smart) : (X)(powerful) : (X)(student)</td>
<td>(X) : (X)(crazy) : (X)(smart) : (X)(powerful) : (X)(student)</td>
<td>4</td>
</tr>
<tr>
<td>[alan ← X]</td>
<td>(X)(crazy) : (X)(student)</td>
<td>(X)(crazy) : (X)(student)</td>
<td>3</td>
</tr>
<tr>
<td>[]</td>
<td>(X)(crazy) : (X)(student)</td>
<td>(X)(crazy) : (X)(student)</td>
<td>2</td>
</tr>
<tr>
<td>[alan ← X]</td>
<td>(X)(crazy) : (X)(student)</td>
<td>(X)(crazy) : (X)(student)</td>
<td>1</td>
</tr>
</tbody>
</table>

**Prolog's Inference Algorithm: Example**

<table>
<thead>
<tr>
<th>ID</th>
<th>CP</th>
<th>G</th>
<th>R</th>
</tr>
</thead>
</table>
Prolog's inference algorithm: example (cont.)

<table>
<thead>
<tr>
<th>ID</th>
<th>CP</th>
<th>G</th>
<th>R</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td></td>
<td>powerful ((X))</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>crazy ((X))</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>(X)</td>
<td>powerful ((X))</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>cs ((X)), prof ((X))</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>colin</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>true ((X)), crazy ((X))</td>
<td></td>
</tr>
<tr>
<td>14</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>17</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>18</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Prolog's response is hence: \([X \mapsto \text{colin}]\).
Drawbacks of Prolog's inference algorithm

Consider the Prolog program:

\begin{align*}
1 & \text{a} : \text{not(true)}. \\
2 & \text{a} : \text{true}. \\
3 & \text{a} : \text{false}. \\
4 & \text{a} \leftarrow \text{not} \left( \text{true} \right). \\
5 & \text{a} \leftarrow \text{false}.
\end{align*}

The query infers `false` by negation as failure (indicated by `*`).

Even worse, the response to the query `not(a)` is true due to two
applications of inversion.

The reason is Prolog's closed-world assumption: It assumed the
database is complete; i.e., if the answer cannot be deduced, it is false.

This, however, does not coincide with our understanding of the
semantics of the implication: $\bot \rightarrow a$ is true independent of
whether $a$ or not.

Even worse, the response to the query `not(a)` is true due to two
applications of inversion.

Let us query whether $a$:

$\text{a} : \text{not} \left( \text{true} \right)$.

Consider the Prolog program:

\[ \text{ID} \]

\[ \text{CP} \]

\[ G \]

\[ R \]
Drawbacks of Prolog's inference algorithm (cont.)

The nature of Prolog being based on Horn logic and its negation and loop handling show a considerable weakness of its inference algorithm.

Consider the Prolog program:

1. `a :- b.`
2. `b :- a.`

Let us query whether `a`:

- 1. `a :- a.`
- 2. `a :- a.`
- 3. `b :- a.`
- 4. `q :- a.`

The program enters an infinite loop even though the query could be proven true in a few steps:

\[ (b \rightarrow a) \land (a \rightarrow b) \rightarrow \bot \]

The nature of Prolog being based on Horn logic and its negation and loop handling show a considerable weakness of its inference algorithm.

Suendermann

Knowledge-Based Systems
November 19, 2012 105
Apart from Prolog’s inference engine, a predominant feature is its list handling. Lists can be written in three ways:

1. \( (s, t) \) defines a list with the element \( s \) and the tail \( t \);
2. \( [s | t] \) does the same;
3. \( \{ s_1, \ldots, s_n \} \) defines a list with the elements \( s_1, \ldots, s_n \) and the tail \( t \).

Accordingly, these are equivalent lists:

- \( 1, 2, 3 \)
- \( [[[[] | 3] | 2] | 1] \)
- \( (1, 2, 3) \)

Lists can be written in three ways:

1. \( (s, t) \) defines a list with the element \( s \) and the tail \( t \);
2. \( [s | t] \) does the same;
3. \( \{ s_1, \ldots, s_n \} \) defines a list with the elements \( s_1, \ldots, s_n \) and the tail \( t \).
Lists: example function

In the following, we run an example to understand the program's functionality.

(Rule 1).

An empty list concatenated with a list results in the same list (Fact 2).

Furthermore, if the concatenation of the lists results in \( L_3 \) and \( L_1 \) with an preceding element \( X \) concatenated with \( L_2 \) must result in \( L_3 \) with the same preceding element \( X \) as the concatenation of the lists \( L_1 \) and \( L_2 \).

This program reads:

1. `cat([],L,L).`
2. `cat([X|L1],L2,[X|L3]):-cat(L1,L2,L3).`

A respective Prolog program is:

\( L_2 \) and \( L_2 \).

In the world of logical programming, this could be conceived as the 3-ary function `cat(\( L_1 \), \( L_2 \), \( L_3 \))` which becomes true if \( L_3 \) is the concatenation of function `cat(\( L_1 \), \( L_2 \), \( L_3 \))` resulting in the list \( L_2 \). We want to design a function `cat` that concatenates two lists \( L_1 \) and \( L_2 \).
\[(54) \quad [\begin{array}{c}
1, 3, 4, T
\end{array}] =
[\begin{array}{c}
[[[3, 4], T, 2, 1], 1, 1, 4, X, 2, 3],
[3, 4, T, 2, 1], 1, 1, 4, X, 2, 3]
\]

**Prolog's response is hence:**

\[
\begin{array}{c|c|c|c}
\text{ID} & \text{CP} & \text{G} & \text{R} \\
\hline
3 & 4 & 2 & 1 \\
\end{array}
\]
You may perceive some flavor of Prolog's elegance if you consider which use cases the above example function features:

- Concatenate two lists:
  ```prolog
cat([X, Y], [1, 2], [3, 4], []).  
```

- Check whether a list resulted from another list by way of concatenation:
  ```prolog
cat([1, 2], Y, [1, 2, 3, 4], []).  
```

- Find all possible splits of a list into two lists:
  ```prolog
cat(X, Y, [1, 2, 3, 4], []).  
```

- Concatenate two lists:
  ```prolog
cat([1, 2], [3, 4], X, []).  
```

You may perceive some flavor of Prolog’s elegance if you consider which use cases the above example function features.
How order matters

Consider the following program:

1. a:-b.
2. a.
3. b:-a.

We get the query result

\[ a \rightarrow \text{Yes} \]  \quad (58)

Now, we reorder the rules:

\[ a \rightarrow \text{Yes} \]  \quad (59)

We get the query result

\[ a \rightarrow \text{Yes} \]  \quad (58)

This, time, the query result is

\[ \begin{array}{c}
3: b=a \\
2: a \\
1: a=b \\
\end{array} \]

This is because Prolog has to create a choice point for every recursion loop, but a stack overflow.

Other than the example on Page 105, this time, we do not get an infinite loop but a stack overflow.

The inference algorithm keeps accessing Rule 1 over and over again.

Due to the presence of the alternative Rule 3.

This is because Prolog has to create a choice point for every recursion.
How order matters (cont.)

Consider the following program:

1. \( s([X],[X]) \).
2. \( s([A,B],[A,D]) :- s([B],[D]), A < D. \)

We get the query result

\[
\text{\( s([1,2],[X]) \).}
\]

\[
\rightarrow X = [1,2].
\]

The inference algorithm tries to evaluate \( A > D \) first, before \( D \) had been determined by way of evaluating \( s([B],[D]) \).

ERROR: Arguments are not sufficiently instantiated.

Now, we switch the elements in Rule 2's body:

1. \( s([X],[X]) \).
2. \( s([A,B],[A,D]) :- A < D, s([B],[D]). \)

This time, we get

\[
\text{\( s([1,2],[X]) \).}
\]

\[
\rightarrow \text{ERROR: Arguments are not sufficiently instantiated.}
\]

We get the query result

\[
\text{\( s([1,2],[X]) \).}
\]

We consider the following program:

\[
\text{\( s([1,2],[X]) \).}
\]
Symbolic vs. numerical computation

Consider the following program:

1. `p(X,Y):-Y==X+1.`
2. `q(X,Y):-Y>X+0.9999999,Y<X+1.0000001.`
3. `r(X,Y):-Y is X+1.`
4. `s(X,Y):-Y=X+1.`

We get the following query results:

- `p(1,2) → No`.
- `q(1,2) → Yes`.
- `r(1,2) → Yes`.
- `s(1,2) → No`.

We get the following query results:

- `ERROR: Arguments are not sufficiently instantiated.`
- `y = 2`.
- `b(1,2) → Yes`.
- `b(1,2) → No`.
- `d(1,2) → Yes`.
- `d(1,2) → No`.

Consider the following program:

```
Symbolic vs. numerical computation
```

Suendermann

Knowledge-Based Systems

November 19, 2012 112
Symbolic vs. numerical computation (cont.)

The check for equality (=) fails due to issues with Prolog’s numerical precision.

Rather than for equality, q checks for a small range around the expected precision.

The unification operator = tries to solve the syntactical equation

\[ 2 = 1 + 1 \]

\[ x + 1 \]

... and succeeds. Accordingly, the free parameter y gets assigned the sum of 1

Prolog’s keyword is assigns the exact value of x + 1 to y and therefore

complains about insufficient instantiation.

However, Prolog is not able to limit the (real-valued) search space and

when queried with the free parameter y, succeeds. When queried with a free parameter, however, when

Hence, it fails. When queried with a free parameter, however, whose solution is the result set.

The syntactical equation is

\[ y = 1 + 1 \]

which is not possible since different function symbols cannot be unified.

The unification operator = tries to solve the syntactical equation...
Write programs to

1) Determine the maximum of two numbers (2 lines)
2) Calculate the factorial (2 lines)
3) Union a list (3 lines)
4) Find identical elements in two lists (3 lines)
5) Sort a list (4 lines)
Notes for the Prolog programming project

**Deadlines:**

<table>
<thead>
<tr>
<th>Group ID(s) (including the year)</th>
<th>Group Proposal</th>
<th>Code Due</th>
<th>Presentation Due</th>
</tr>
</thead>
<tbody>
<tr>
<td>AIA09</td>
<td>March 23</td>
<td>April 17</td>
<td>April 18</td>
</tr>
<tr>
<td>AIB18</td>
<td>March 26</td>
<td>April 20</td>
<td>April 18</td>
</tr>
<tr>
<td>AIC18</td>
<td>March 26</td>
<td>April 20</td>
<td>April 18</td>
</tr>
<tr>
<td>AID18</td>
<td>March 24</td>
<td>April 18</td>
<td>April 16</td>
</tr>
</tbody>
</table>

Proposals have to be submitted to all of the following e-mail addresses:

- david@suendermann.com
- david@speechcycle.com
- suendermann@dhbw-stuttgart.de

The subject line of the e-mail has to contain:

- the word "proposals",
- your first and last name(s),
- your matriculation number(s),
- and your group ID(s) including the year (e.g. AIA09).

---

Suendermann
Knowledge-Based Systems
November 19, 2012 115
Notes for the Prolog programming project (cont.)

Up to two students can collaborate on each project.

Proposals have to contain a brief (no more than 200 words) but clear description of what your program is supposed to achieve and how typical rules and queries are expected to look like.

The Prolog code of your project needs to be well-documented according to common coding standards. For a Prolog coding style reference, see: http://www.ai.uga.edu/mc/plcoding.pdf

Submit your program to the above listed e-mail addresses with the above described subject line, replacing „proposall by „code“.

There is no need for any documentation outside of the code itself.

Make sure the code clearly shows example queries to run the code.

Do not dare to copy anybody else’s code. Every identified attempt will cause failure of the project.

So does missing a deadline without prior permission or medical certificate.

Do not copy anyone else’s code. Every identified attempt will cause failure of the project.

Submit your program to the above listed e-mail addresses with the above described subject line, replacing „proposall by „code“.

There is no need for any documentation outside of the code itself.

Make sure the code clearly shows example queries to run the code.

Do not dare to copy anybody else’s code. Every identified attempt will cause failure of the project.

So does missing a deadline without prior permission or medical certificate.
November 19, 2012

Notes for the Prolog programming project (cont.)

There is no right to claim the 20% associated with difficulty, either. Presentations will be held in the classroom during the time of the regular lecture. An exact schedule will be compiled shortly before.

Presentations are 15 minutes in duration. During this time, you need to convince me to respond positively to the questions in the table below.

To derive the coding project’s final score, I will consider answers to the following questions:

<table>
<thead>
<tr>
<th>Percentage</th>
<th>Question</th>
</tr>
</thead>
<tbody>
<tr>
<td>20%</td>
<td>Was the project well presented?</td>
</tr>
<tr>
<td>20%</td>
<td>Was the code well documented?</td>
</tr>
<tr>
<td>40%</td>
<td>Does the program fulfill the proposed task?</td>
</tr>
<tr>
<td>20%</td>
<td>Is the proposed task difficult to be implemented in Prolog?</td>
</tr>
</tbody>
</table>

In doing so, I will generally apply a weighting scheme according to the percentages of the table (exceptions are possible, e.g., if somebody proposes a very difficult task and does not submit anything useful, there is no right to claim the 20% associated with difficulty, either).
Example projects include:

- Crossword puzzle
- Sudoku
- n-Queens puzzle
- g-Puzzle
- Traveling Salesman problem
- Applications of dynamic programming (e.g., Levenshtein distance)
- Resolution in predicate logic
- Davis and Putnam algorithm for propositional logic
- A* search

Example projects include: