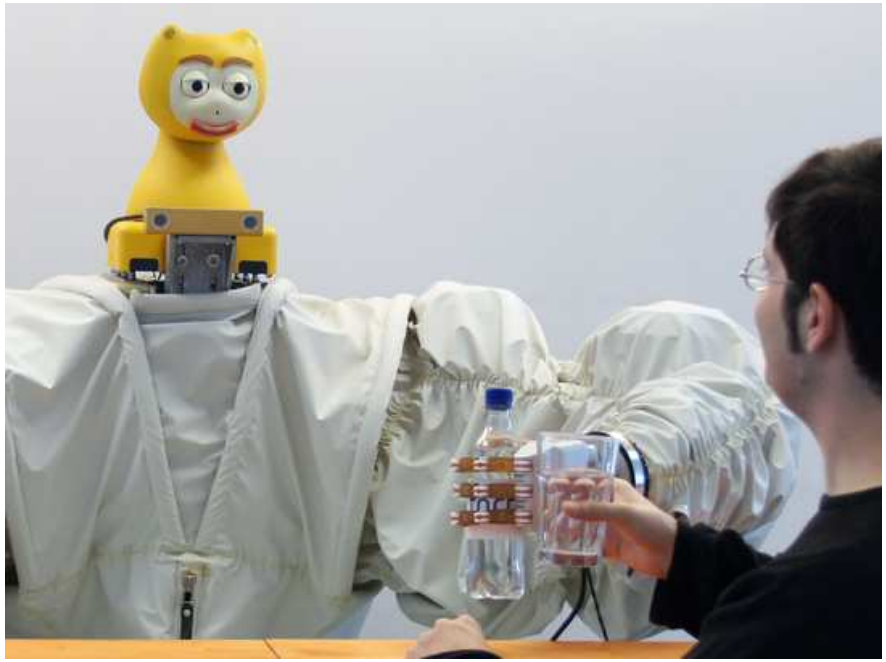
Vorlesung

Grundlagen der Künstlichen Intelligenz

Reinhard Lafrenz / Prof. A. Knoll

Robotics and Embedded Systems
Department of Informatics – I6
Technische Universität München

www6.in.tum.de
lafrenz@in.tum.de
089-289-18136
Room 03.07.055



Wintersemester 2012/13 23.11.2012

Chapter 7 (3rd ed., cont‘d)

Logics

From last lecture we know

- Wumpus world
- Propositional Logic with syntax and semantics
- Truth-table approach for proofs

Inference by enumeration

Depth-first enumeration of all models is sound and complete

```
function TT-ENTAILS?(KB, α) returns true or false
    symbols ← a list of the proposition symbols in KB and α
    return TT-CHECK-ALL(KB, α, symbols, [])

function TT-CHECK-ALL(KB, α, symbols, model) returns true or false
    if EMPTY?(symbols) then
        if PL-TRUE?(KB, model) then return PL-TRUE?(α, model)
        else return true
    else do
        P ← FIRST(symbols); rest ← REST(symbols)
        return TT-CHECK-ALL(KB, α, rest, EXTEND(P, true, model)) and
               TT-CHECK-ALL(KB, α, rest, EXTEND(P, false, model))
```

For n symbols, time complexity is $O(2^n)$,

space complexity is $O(n)$

Logical equivalence

- Two sentences are logically equivalent if and only if they are true in same models: $\alpha \equiv ß$ **iff** $\alpha \models \beta$ and $\beta \models \alpha$

$$
\begin{aligned}
(\alpha \wedge \beta) &\equiv (\beta \wedge \alpha) && \text{commutativity of } \wedge \\
(\alpha \vee \beta) &\equiv (\beta \vee \alpha) && \text{commutativity of } \vee \\
((\alpha \wedge \beta) \wedge \gamma) &\equiv (\alpha \wedge (\beta \wedge \gamma)) && \text{associativity of } \wedge \\
((\alpha \vee \beta) \vee \gamma) &\equiv (\alpha \vee (\beta \vee \gamma)) && \text{associativity of } \vee \\
\neg(\neg\alpha) &\equiv \alpha && \text{double-negation elimination} \\
(\alpha \Rightarrow \beta) &\equiv (\neg\beta \Rightarrow \neg\alpha) && \text{contraposition} \\
(\alpha \Rightarrow \beta) &\equiv (\neg\alpha \vee \beta) && \text{implication elimination} \\
(\alpha \Leftrightarrow \beta) &\equiv ((\alpha \Rightarrow \beta) \wedge (\beta \Rightarrow \alpha)) && \text{biconditional elimination} \\
\neg(\alpha \wedge \beta) &\equiv (\neg\alpha \vee \neg\beta) && \text{de Morgan} \\
\neg(\alpha \vee \beta) &\equiv (\neg\alpha \wedge \neg\beta) && \text{de Morgan} \\
(\alpha \wedge (\beta \vee \gamma)) &\equiv ((\alpha \wedge \beta) \vee (\alpha \wedge \gamma)) && \text{distributivity of } \wedge \text{ over } \vee \\
(\alpha \vee (\beta \wedge \gamma)) &\equiv ((\alpha \vee \beta) \wedge (\alpha \vee \gamma)) && \text{distributivity of } \vee \text{ over } \wedge
\end{aligned}
$$

Terminology

- Countable alphabet Σ of consisting of A, B, C, …
- Atom: atomic sentence, i.e. a symbol from the alphabet
- Literal: (possibly negated) atomic sentence, e.g. A, $\neg$B
- Clause: Disjunction of literals, e.g. (A $\vee$ $\neg$B $\vee$ C)

Validity and satisfiability

- A sentence is valid (tautology) if it is true in all models,
  e.g., *True*, $A \vee \neg A$, $A \Rightarrow A$, $(A \wedge (A \Rightarrow B)) \Rightarrow B$

- A sentence is satisfiable if it is true in some model
  e.g., $A \vee B$, C

- A sentence is unsatisfiable if it is true in no models
  e.g., $A \wedge \neg A$

- Validity is connected to inference via the Deduction Theorem:
  $KB \models \alpha$ if and only if $(KB \Rightarrow \alpha)$ is valid

- Satisfiability is connected to inference via:
  $KB \models \alpha$ if and only if $(KB \wedge \neg \alpha)$ is unsatisfiable

Proof methods

Proof methods divide into (roughly) two kinds:

- Application of inference rules
  - Legitimate (sound) generation of new sentences from existing ones
  - Proof = a sequence of inference rule applications
  - Algorithms can use inference rules as operators in a standard search algorithm
  - Typically require transformation of sentences into a normal form

- Model checking
  - truth table enumeration (always exponential in *n*)
  - improved backtracking, e.g., Davis-Putnam-Logemann-Loveland (DPLL) algorithm
  - heuristic search in model space (sound but incomplete)
    e.g., min-conflicts-like hill-climbing algorithms

Normal forms

Conjunctive normal form (CNF):

The sentence consists of a conjunction of disjunctions of literals $l_{i,j}$, i.e. it has the following form:

$$\bigwedge_{i=1..n}\left(\bigvee_{j=1..m} l_{i,j}\right)$$

e.g., (A ∨ B ∨ ¬C) ∧ (¬D ∨ B) ∧ (E ∨ C ∨ F)

Disjunctive normal form (DNF):

The sentence consists of a disjunction of conjunctions of literals $l_{i,j}$, i.e. it has the following form:

$$\bigvee_{i=1..n}\left(\bigwedge_{j=1..m} l_{i,j}\right)$$

e.g., (A ∧ B ∧ ¬C) ∨ (¬D ∧ B) ∨ (E ∧ C ∧ F)

Normal forms - properties

- For all sentences, there is at least one equivalent sentence in CNF and DNF.
- A sentence in DNF is satisfiable **iff** a disjunction is satisfiable.
- A sentence in KNF is valid **iff** every conjunction is valid.

Construction of CNF sentences

1. Eliminate $\Rightarrow$ and $\Leftrightarrow$: $A \Rightarrow B$ ~> $(\neg A \vee B)$
2. Move $\neg$ to the inside: $\neg(A \wedge B)$ ~> $(\neg A \vee \neg B)$
3. Distribute $\vee$ over $\wedge$: $(A \wedge B) \vee C$ ~> $(A \vee C) \wedge (B \vee C)$
4. Simplify: $(A \vee A)$ ~> A

The result is a conjunction of disjunction of literals.

- An analogous procedure transforms any sentence into an equivalent sentence in DNF.
- Sentences can be expanded exponentially during the transformation.

Inference rules

Modus ponens

$$\frac{A \Rightarrow B,\ A}{B}$$

Conjunction Elimination

$$\frac{A_1 \wedge A_2 \wedge A_3 \wedge \ldots \wedge A_n}{A_i}$$

Conjunction Introduction

$$\frac{A_1,\ A_2,\ A_3,\ \ldots,\ A_n}{A_1 \wedge A_2 \wedge A_3 \wedge \ldots \wedge A_n}$$

Disjunction Introduction

$$\frac{A_i}{A_1 \vee A_2 \vee A_3 \vee \ldots \vee A_n}$$

Inference rules - Resolution

Unit resulotion

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

Resolution

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

or, more general

$$\frac{A_1 \vee A_2 \vee \ldots \vee A_n,\quad \neg A_1 \vee B_2 \vee \ldots \vee B_n}{A_2 \vee \ldots \vee A_n \vee B_2 \vee \ldots \vee B_n}$$

Proof by Resolution - Idea

- We now want to study a deduction technique that does not rely on explicitly testing all interpretations.
- Idea: You try to show that a set of sentences is unsatisfiable.
- However: It is required that all sentences are given in CNF.
- But: In most cases, the sentences are close to their CNF (and there is a quick transformation that preserves satisfyability).
- Nevertheless: In the worst case, this deduction technique also requires an exponential amount of time (probably, you cannot avoid this).

Proof by Resolution

To prove that KB |= A , show that (KB $\wedge \neg$A) is unsatisfiable

Use the inference rules including th resolution rules to infer the empty clause $\square$ from (KB $\wedge \neg$A)

E.g. show that in the Wumpus world there is no pit at [1,2],

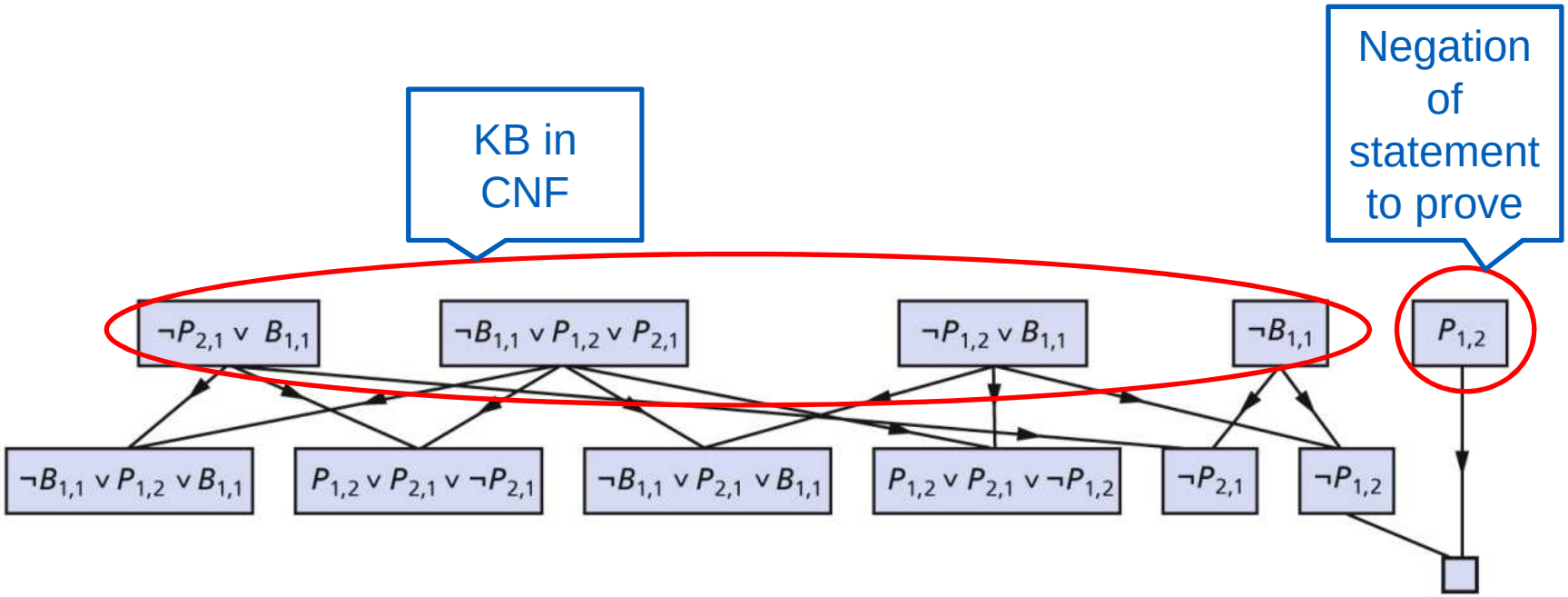
formal: $\neg P_{1,2}$

The Knowledge base is

$KB = (B_{1,1} \Rightarrow (P_{1,2} \vee P_{2,1})) \wedge \neg B_{1,1}$

Proof by Resolution

Add the negated clause you want to show to the KB:



A simple resolution algorithm

function PL-RESOLUTION(KB, α) **returns** *true* or *false*

clauses ← the set of clauses in the CNF representation of $KB \wedge \neg\alpha$
new ← { }
loop do
  for each C_i, C_j **in** *clauses* **do**
    resolvents ← PL-RESOLVE(C_i, C_j)
    if *resolvents* contains the empty clause **then return** *true*
    new ← *new* ∪ *resolvents*
  if *new* ⊆ *clauses* **then return** *false*
  clauses ← *clauses* ∪ *new*

Completeness of the resolution

- Resolution theorem:

If a set of clauses is unsatisfiable, the resolution closure of these clauses contains the empty clause

- i.e., the resolution is complete, KB |= A can be shown by deriving $\square$ from (KB $\wedge \neg$A).

Resticted forms of clauses

- Definite clause: dicjunction of literals with exactly one positive literal, e.g. $(A \vee \neg B \vee \neg C)$
- Horn clause: dicjunction of literals with at most one positive literal, e.g. $(A \vee \neg B \vee \neg C)$, $(\neg B \vee \neg C)$

The resolvent of two Horn clauses is again a Horn clause.

Horn clauses can be read as implications:

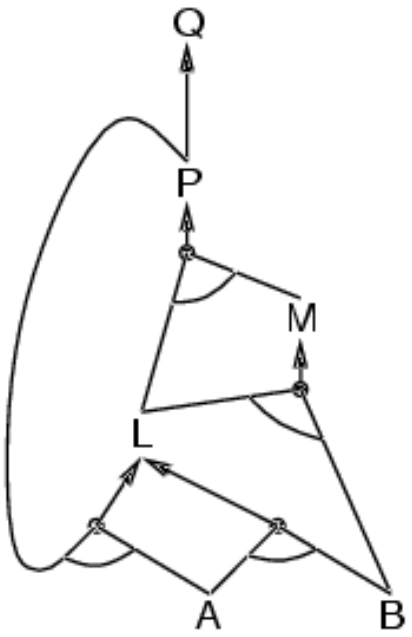
$A \vee \neg B \vee \neg C \quad \equiv \quad (B \wedge C) \Rightarrow A$

Horn clauses can be used for forward- and backward chaining

Forward chaining

- Idea: fire any rule whose premises are satisfied in the *KB*,
  - add its conclusion to the *KB*, until query is found

$$P \Rightarrow Q$$
$$L \wedge M \Rightarrow P$$
$$B \wedge L \Rightarrow M$$
$$A \wedge P \Rightarrow L$$
$$A \wedge B \Rightarrow L$$
$$A$$
$$B$$

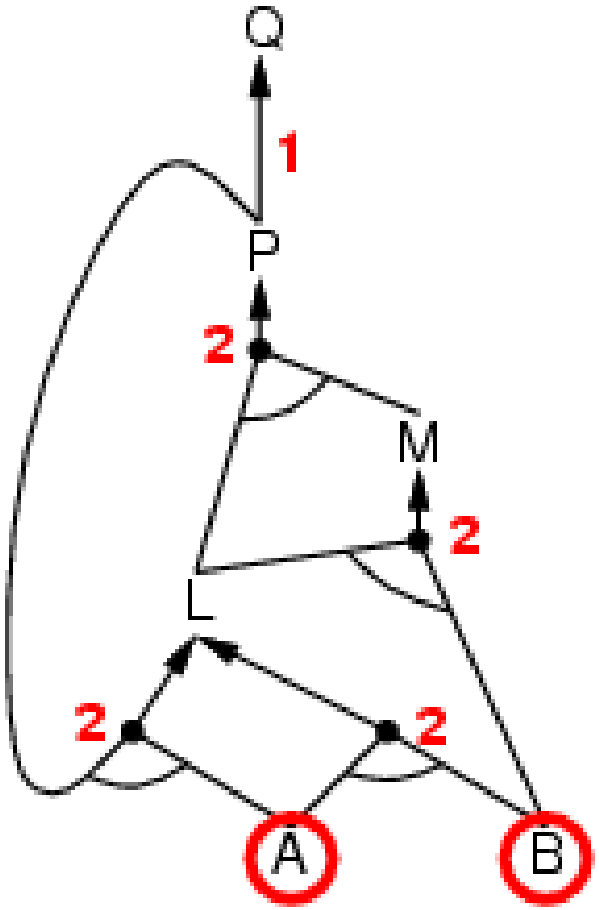


Forward chaining algorithm

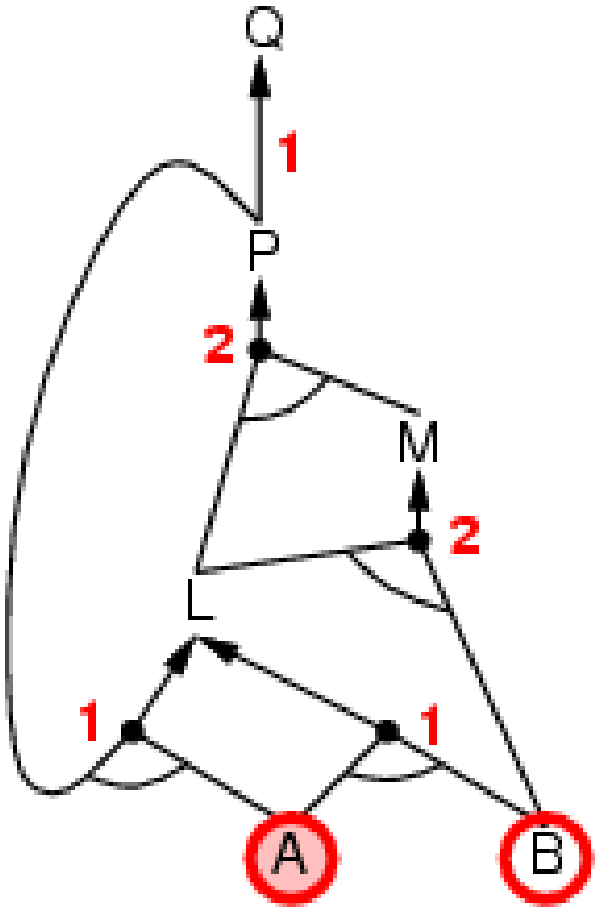
```
function PL-FC-ENTAILS?(KB, q) returns true or false
  local variables: count, a table, indexed by clause, initially the number of premises
                   inferred, a table, indexed by symbol, each entry initially false
                   agenda, a list of symbols, initially the symbols known to be true

  while agenda is not empty do
      p ← POP(agenda)
      unless inferred[p] do
          inferred[p] ← true
          for each Horn clause c in whose premise p appears do
              decrement count[c]
              if count[c] = 0 then do
                  if HEAD[c] = q then return true
                  PUSH(HEAD[c], agenda)
  return false
```

- Forward chaining is sound and complete for Horn KBs

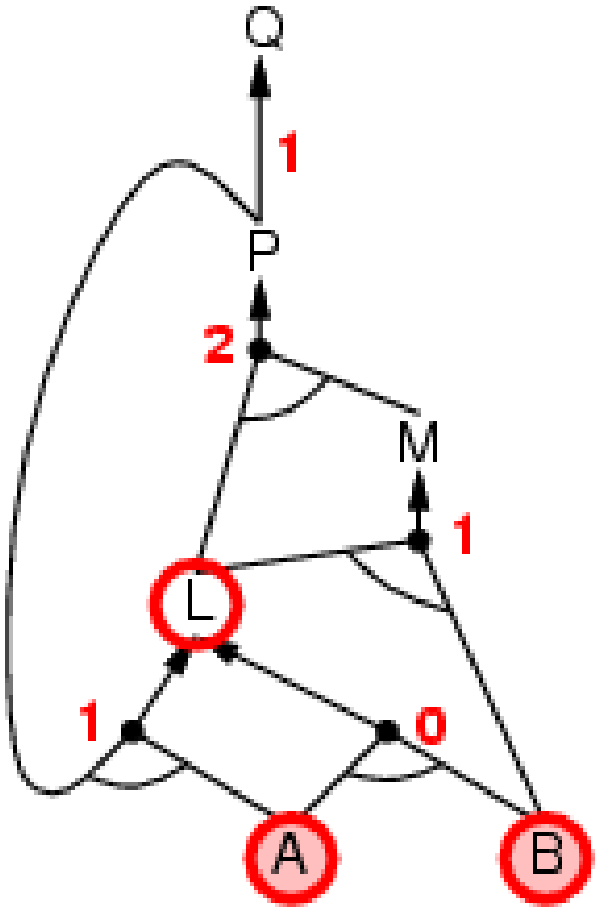
Forward chaining example



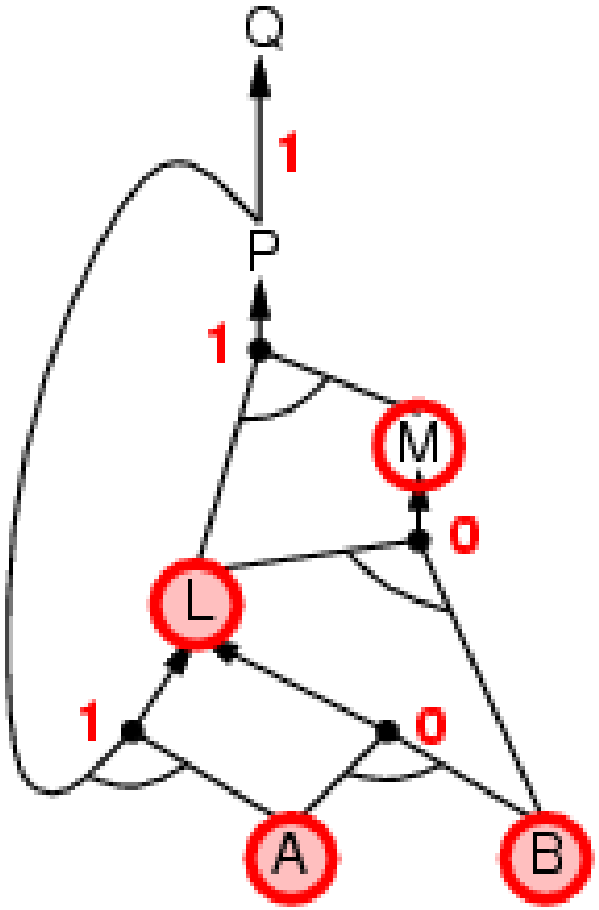
Forward chaining example



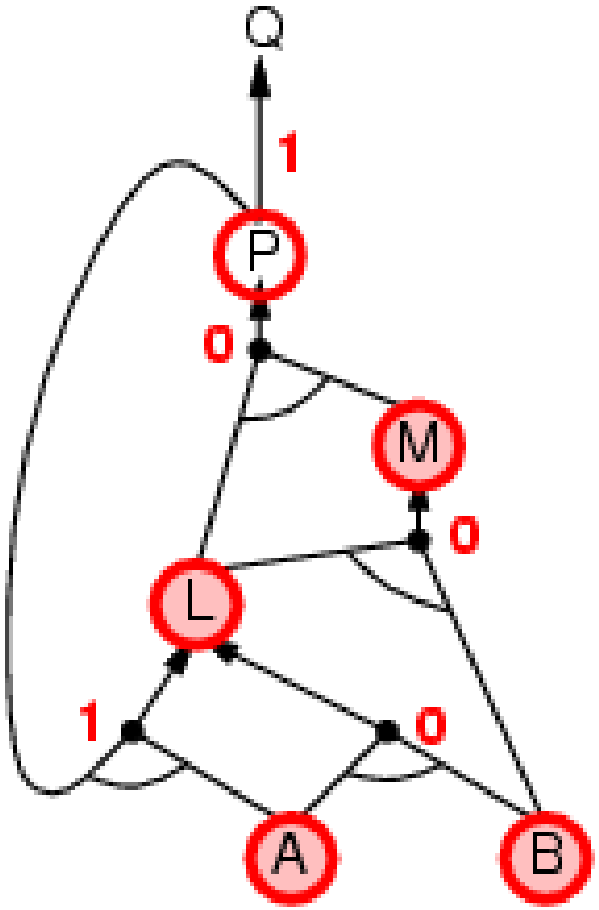
Forward chaining example



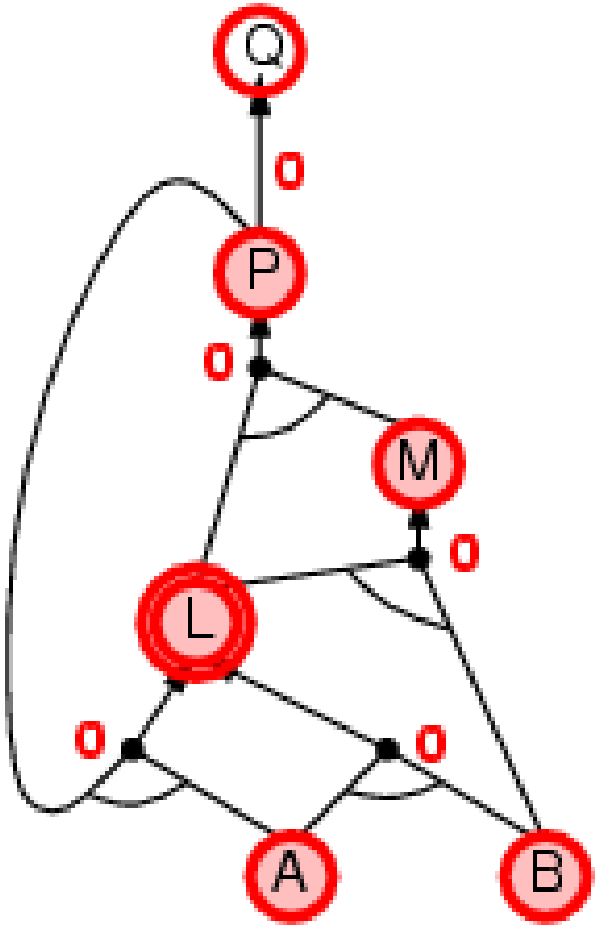
Forward chaining example



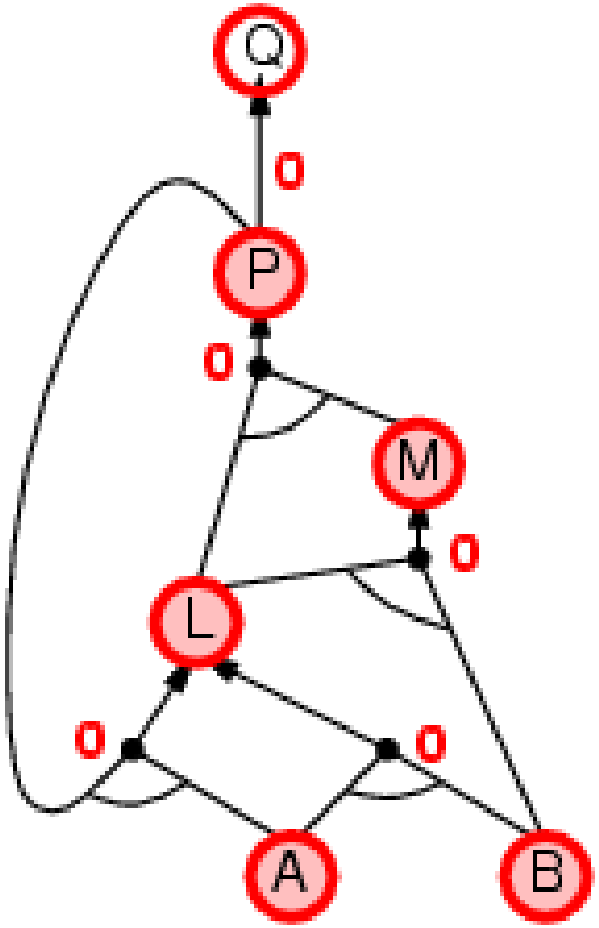
Forward chaining example



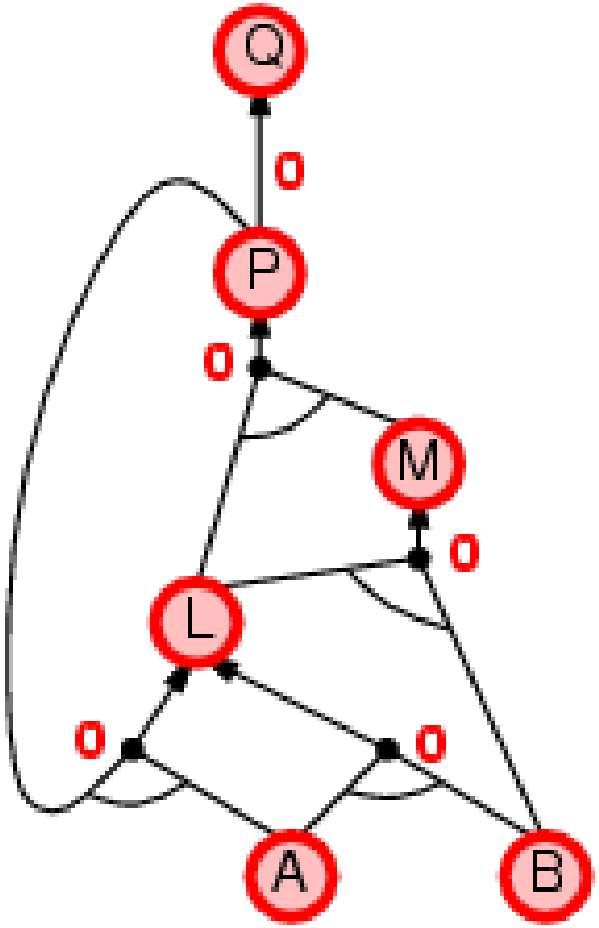
Forward chaining example



Forward chaining example



Forward chaining example



Backward chaining

Idea: work backwards from the query *q*:

to prove *q* by BC,

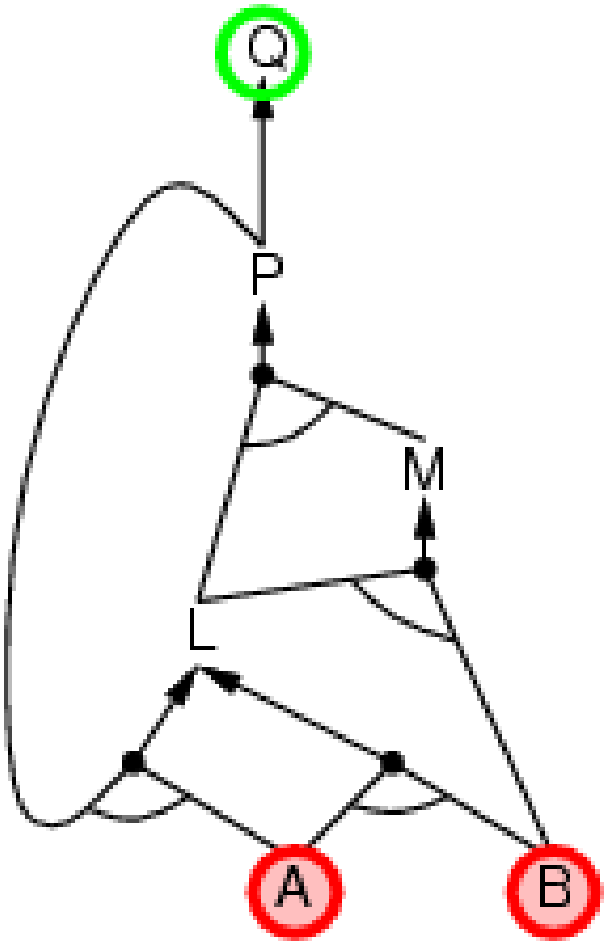
- check if *q* is known already, or
- prove by BC all premises of some rule concluding *q*

Avoid loops: check if new subgoal is already on the goal stack

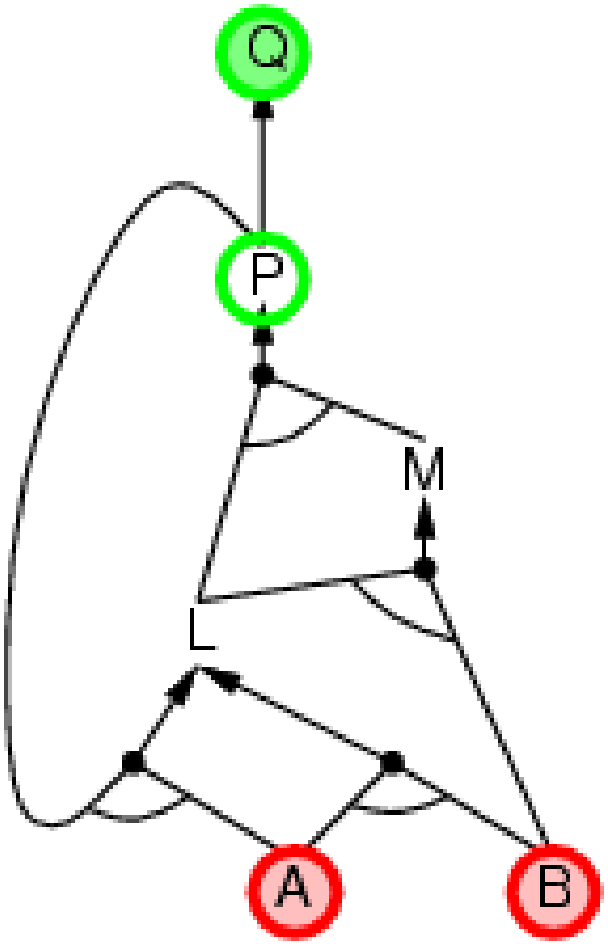
Avoid repeated work: check if new subgoal

1. has already been proved true, or
2. has already failed

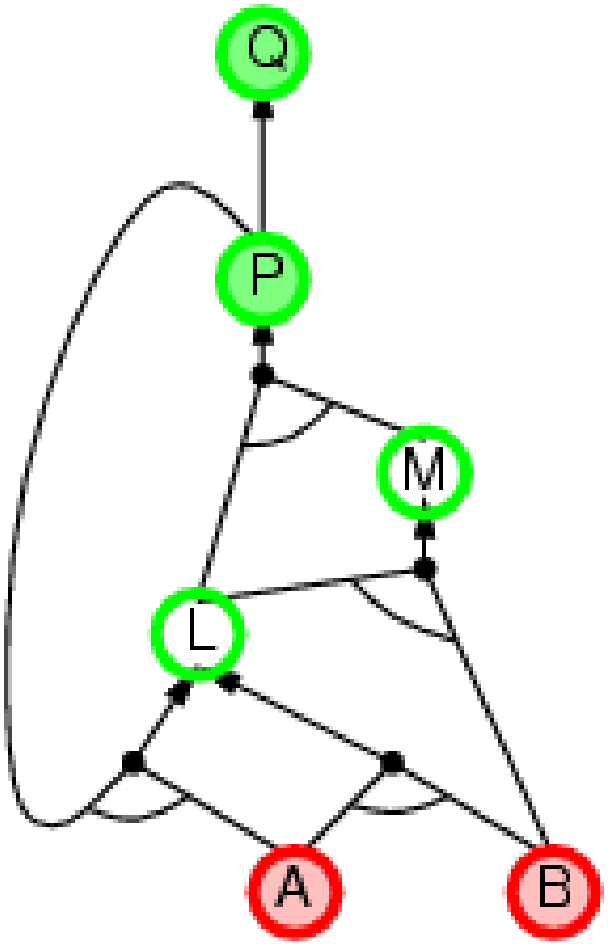
Backward chaining example



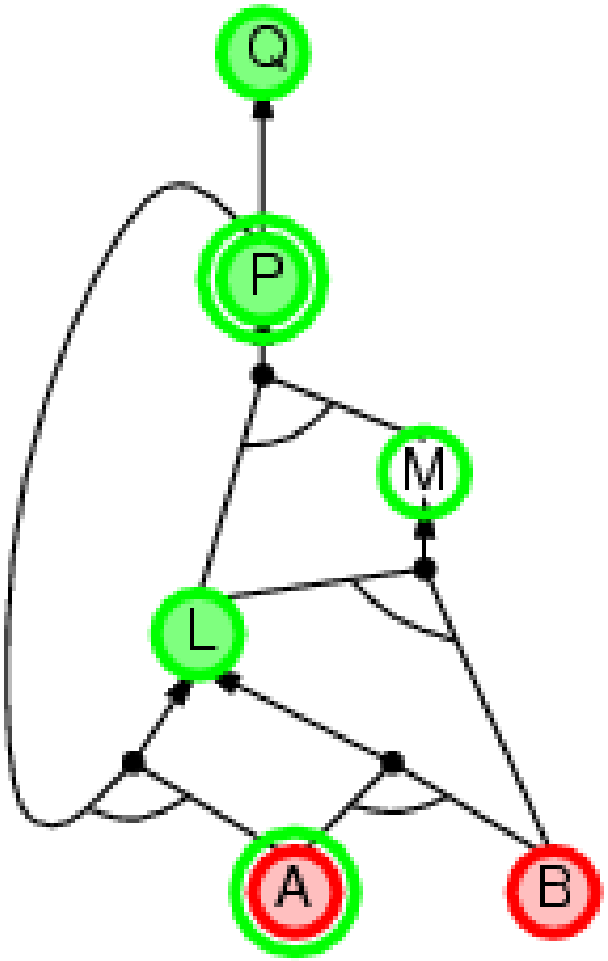
Backward chaining example



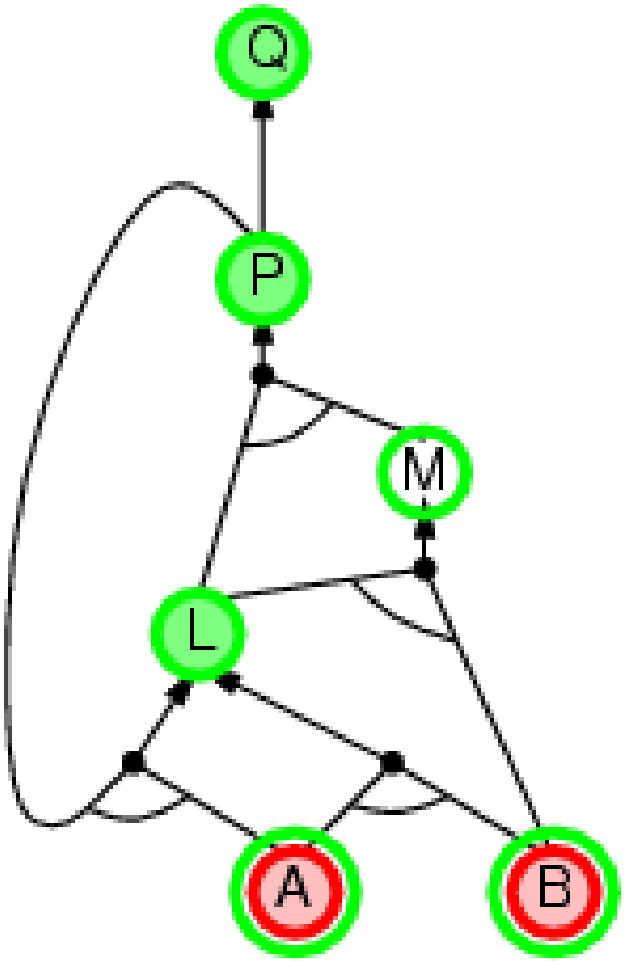
Backward chaining example



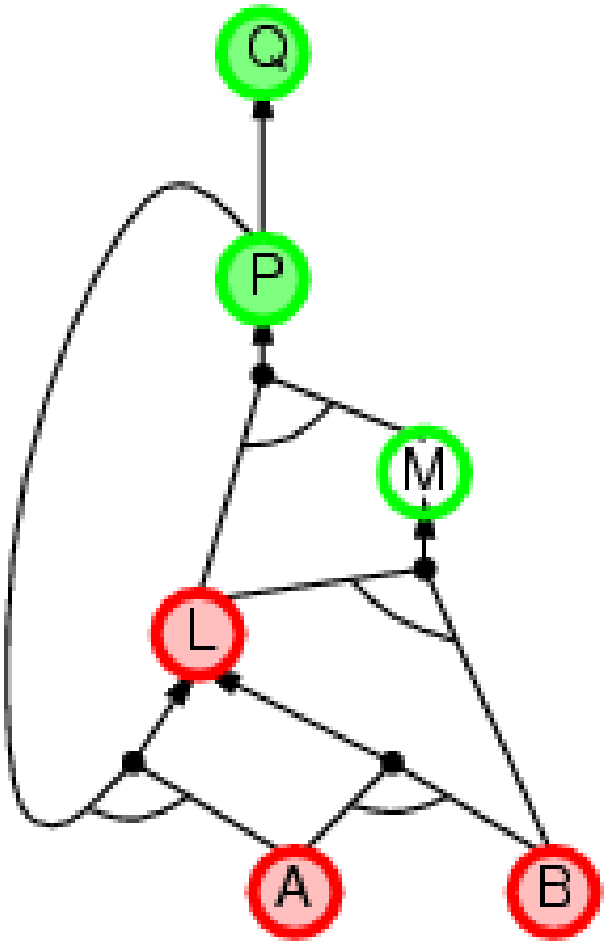
Backward chaining example



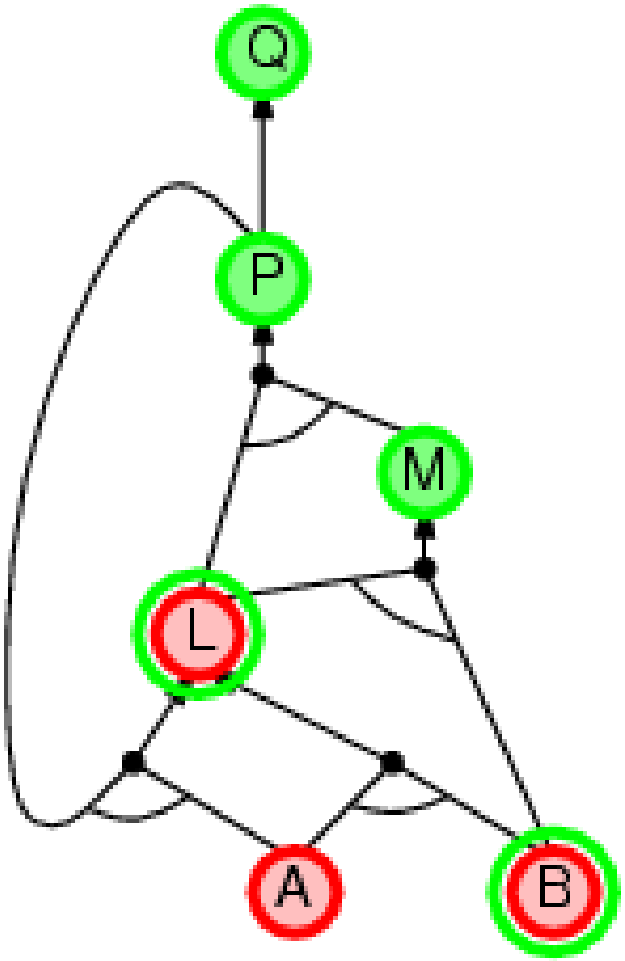
Backward chaining example



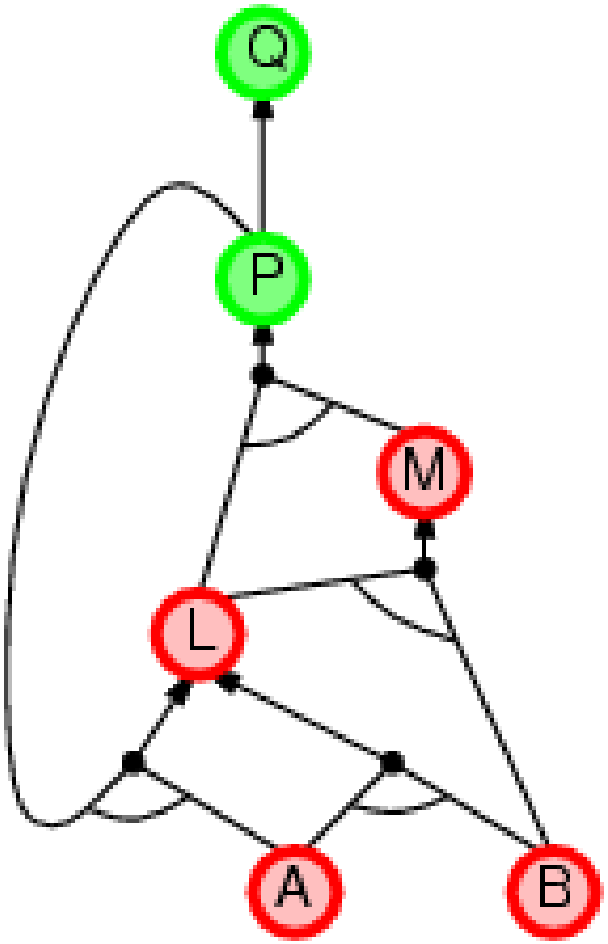
Backward chaining example



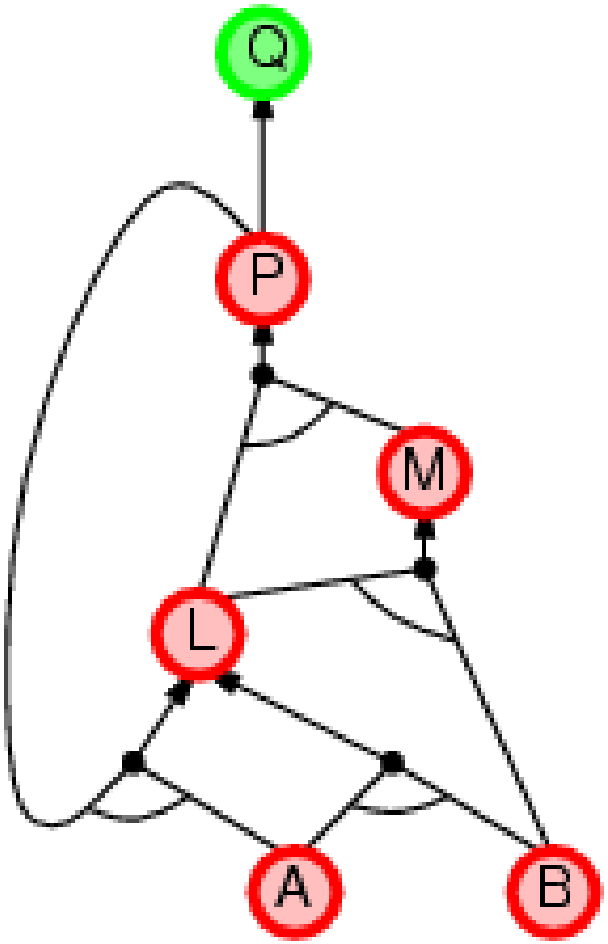
Backward chaining example



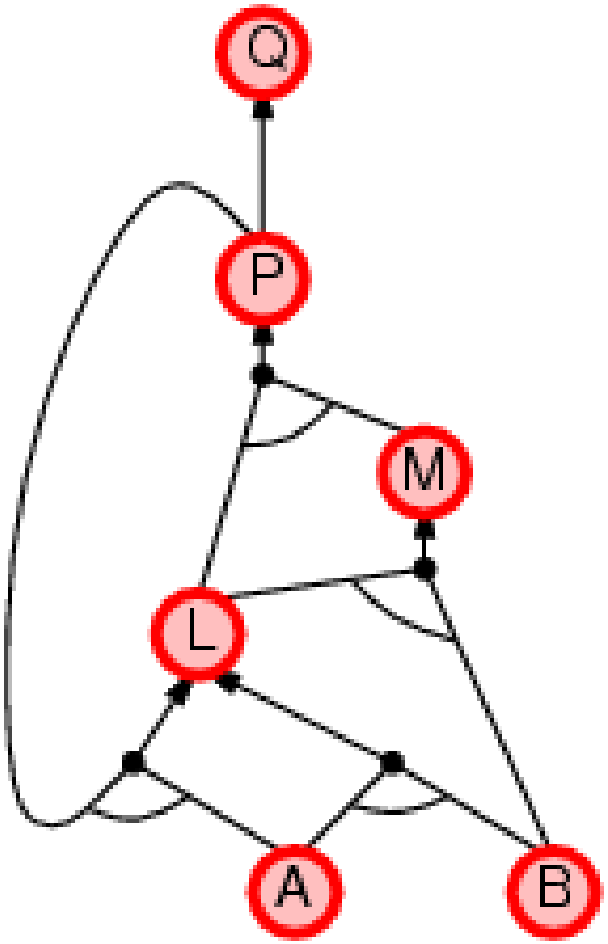
Backward chaining example



Backward chaining example



Backward chaining example



Forward vs. backward chaining

- FC is data-driven, automatic, unconscious processing,
  - e.g., object recognition, routine decisions
- May do lots of work that is irrelevant to the goal
- BC is goal-driven, appropriate for problem-solving,
  - e.g., Where are my keys? How do I get into a PhD program?
- Complexity of BC can be much less than linear in size of KB

Efficient propositional inference

Two families of efficient algorithms for propositional inference:

Complete backtracking search algorithms

- DPLL algorithm (Davis, Putnam, Logemann, Loveland)

- Incomplete local search algorithms
  - `WalkSAT` algorithm

The DPLL algorithm

Determine if an input propositional logic sentence (in CNF) is satisfiable. Improvements over truth table enumeration:

1. Early termination
   A clause is true if any literal is true.
   A sentence is false if any clause is false.

2. Pure symbol heuristic
   Pure symbol: always appears with the same "sign" in all clauses.
   e.g., In the three clauses $(A \vee \neg B)$, $(\neg B \vee \neg C)$, $(C \vee A)$, A and B are pure, C is impure. Make a pure symbol literal true.

3. Unit clause heuristic
   Unit clause: only one literal in the clause
   The only literal in a unit clause must be true.

The DPLL algorithm

function DPLL-SATISFIABLE?(s) **returns** *true* or *false*

inputs: s, a sentence in propositional logic

clauses ← the set of clauses in the CNF representation of s

symbols ← a list of the proposition symbols in s

return DPLL(*clauses*, *symbols*, [])

---

function DPLL(*clauses*, *symbols*, *model*) **returns** *true* or *false*

if every clause in *clauses* is true in *model* **then return** *true*

if some clause in *clauses* is false in *model* **then return** *false*

P, *value* ← FIND-PURE-SYMBOL(*symbols*, *clauses*, *model*)

if P is non-null **then return** DPLL(*clauses*, *symbols*–P, [$P =$ *value*|*model*])

P, *value* ← FIND-UNIT-CLAUSE(*clauses*, *model*)

if P is non-null **then return** DPLL(*clauses*, *symbols*–P, [$P =$ *value*|*model*])

P ← FIRST(*symbols*); *rest* ← REST(*symbols*)

return DPLL(*clauses*, *rest*, [$P =$ *true*|*model*]) **or**
DPLL(*clauses*, *rest*, [$P =$ *false*|*model*])

The `WalkSAT` algorithm

- Incomplete, local search algorithm
- Evaluation function: The min-conflict heuristic of minimizing the number of unsatisfied clauses
- Balance between greediness and randomness

The WalkSAT algorithm

```
function WalkSAT(clauses, p, max-flips) returns a satisfying model or failure
    inputs: clauses, a set of clauses in propositional logic
            p, the probability of choosing to do a "random walk" move
            max-flips, number of flips allowed before giving up

    model ← a random assignment of true/false to the symbols in clauses
    for i = 1 to max-flips do
        if model satisfies clauses then return model
        clause ← a randomly selected clause from clauses that is false in model
        with probability p flip the value in model of a randomly selected symbol
                from clause
        else flip whichever symbol in clause maximizes the number of satisfied clauses
    return failure
```


Summary

- Logical agents apply inference to a knowledge base to derive new information and make decisions

Basic concepts of logic:

- syntax: formal structure of sentences
- semantics: truth of sentences wrt models
- entailment: necessary truth of one sentence given another
- inference: deriving sentences from other sentences
- soundness: derivations produce only entailed sentences
- completeness: derivations can produce all entailed sentences

- Wumpus world requires the ability to represent partial and negated information, reason by cases, etc.
- Resolution is complete for propositional logic
- Forward, backward chaining are linear-time, complete for Horn clauses
- DPLL and WalkSAT algorithms