Propositional inference, propositional agents

Chapter 7.5–7.7

Outline

- ◇ Inference rules and theorem proving
  - – forward chaining
  - – backward chaining
  - – resolution
- ◇ Efficient model checking algorithms
- ◇ Boolean circuit agents

Proof methods

Proof methods divide into (roughly) two kinds:

Application of inference rules
- – Legitimate (sound) generation of new sentences from old
- – Proof = a sequence of inference rule applications
  Can use inference rules as "actions" in a standard search alg.
- – Typically require translation of sentences into a normal form

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

Forward and backward chaining

Horn Form (restricted)
KB = **conjunction** of **Horn clauses**
Horn clause =
- ◇ proposition symbol; or
- ◇ (conjunction of symbols) $\Rightarrow$ symbol

E.g., $C \wedge (B \Rightarrow A) \wedge (C \wedge D \Rightarrow B)$

Modus Ponens (for Horn Form): complete for Horn KBs

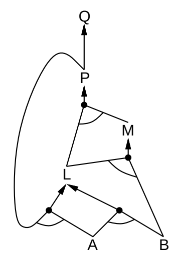
$$\frac{\alpha_1, \ldots, \alpha_n, \qquad \alpha_1 \wedge \cdots \wedge \alpha_n \Rightarrow \beta}{\beta}$$

Can be used with forward chaining or backward chaining.
These algorithms are very natural and run in **linear** time

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

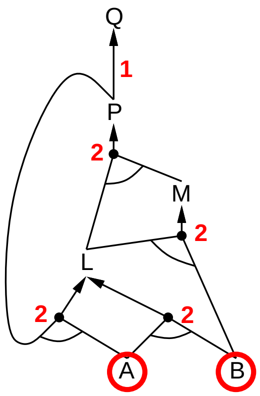


Forward chaining algorithm

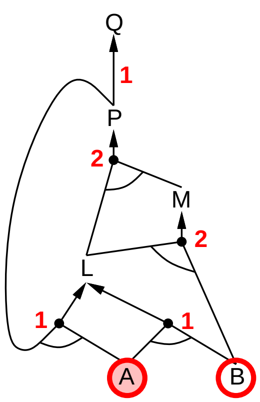
```
function PL-FC-Entails?(KB, q) returns true or false
   inputs: KB, the knowledge base, a set of propositional Horn clauses
           q, the query, a proposition symbol
  local variables: count, a table, indexed by clause, initially the number of premises
                   inferred, a table, indexed by symbol, each entry initially false
                   agenda, a list of symbols, initially the symbols known in KB

   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
```

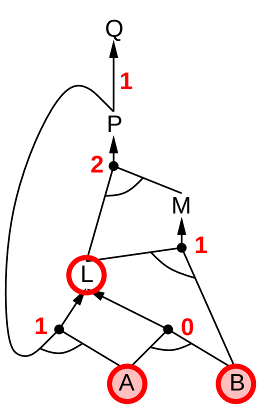
Forward chaining example



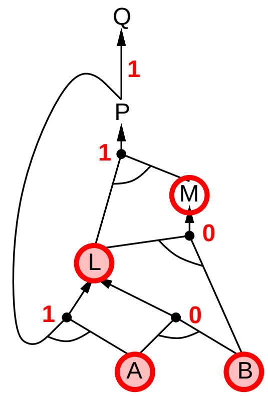
Forward chaining example



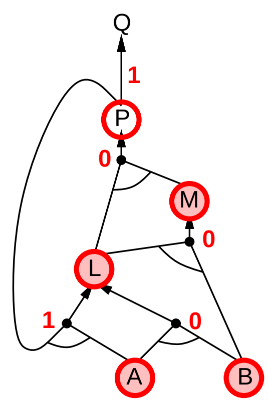
Forward chaining example



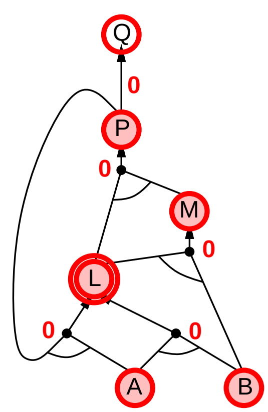
Forward chaining example



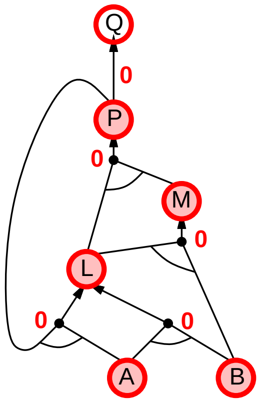
Forward chaining example



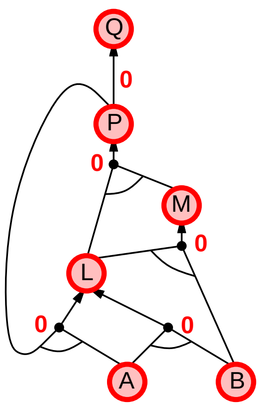
Forward chaining example



Forward chaining example



Forward chaining example



Proof of completeness

FC derives every atomic sentence that is entailed by KB

1. FC reaches a fixed point where no new atomic sentences are derived

2. Consider the final state as a model m, assigning true/false to symbols

3. Every clause in the original KB is true in m
   Proof: Suppose a clause $a_1 \wedge \ldots \wedge a_k \Rightarrow b$ is false in m
   Then $a_1 \wedge \ldots \wedge a_k$ is true in m and b is false in m
   Therefore the algorithm has not reached a fixed point!

4. Hence m is a model of KB

5. If $KB \models q$, q is true in **every** model of KB, including m

General idea: construct any model of KB by sound inference, check α

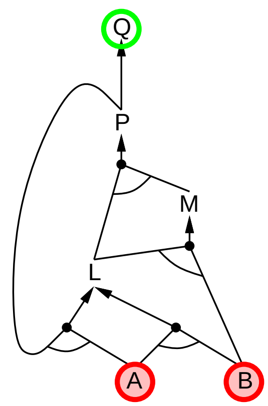
Backward chaining

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

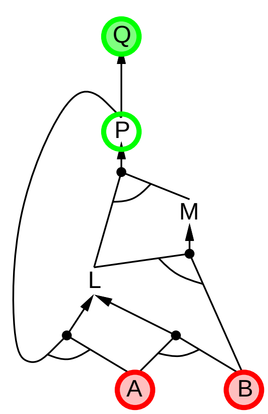
Avoid loops: check if new subgoal is already on the goal stack

Avoid repeated work: check if new subgoal
1) has already been proved true, or
2) has already failed

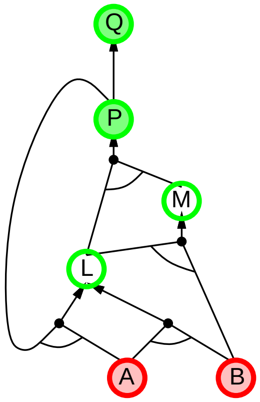
Backward chaining example



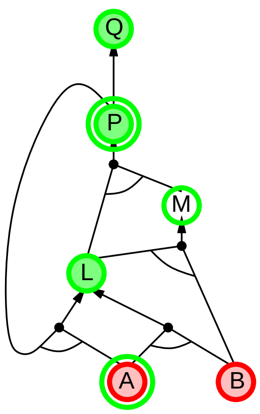
Backward chaining example



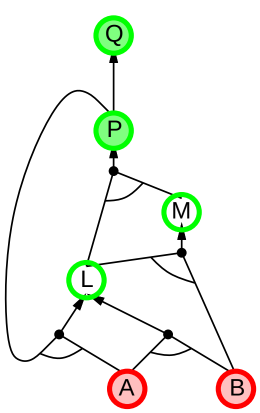
Backward chaining example



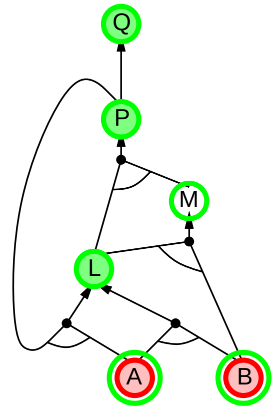
Backward chaining example



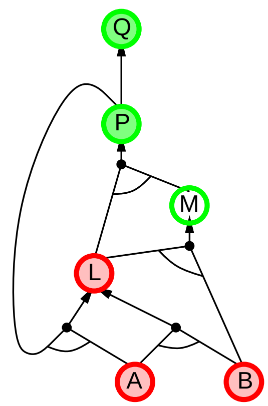
Backward chaining example



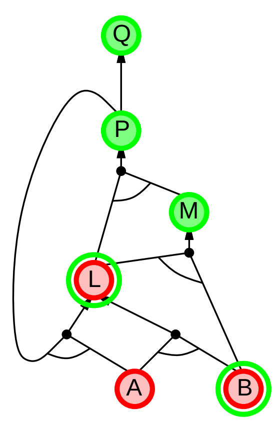
Backward chaining example



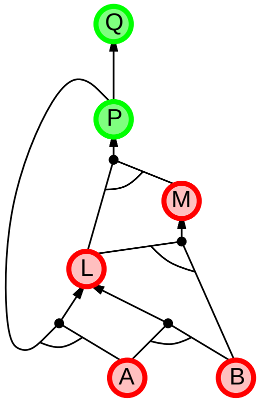
Backward chaining example



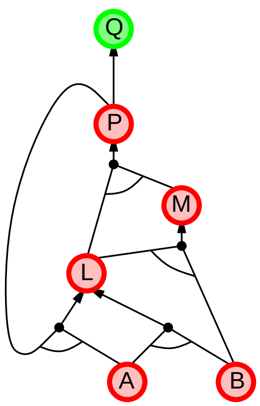
Backward chaining example



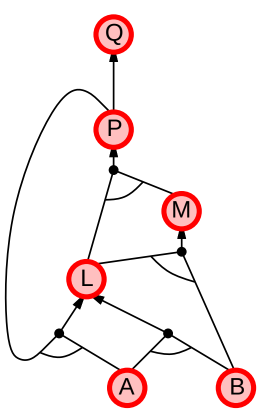
Backward chaining example



Backward chaining example



Backward chaining example



Forward vs. backward chaining

FC is data-driven, cf. 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

Resolution

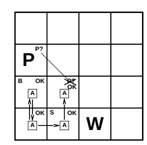
Conjunctive Normal Form (CNF—universal)
$\text{conjunction of } \underbrace{\text{disjunctions of literals}}_{\text{clauses}}$
E.g., $(A \vee \neg B) \wedge (B \vee \neg C \vee \neg D)$

Resolution inference rule (for CNF):

$$\frac{\ell_1 \vee \cdots \vee \ell_k, \qquad m_1 \vee \cdots \vee m_n}{\ell_1 \vee \cdots \vee \ell_{i-1} \vee \ell_{i+1} \vee \cdots \vee \ell_k \vee m_1 \vee \cdots \vee m_{j-1} \vee m_{j+1} \vee \cdots \vee m_n}$$

where ℓ_i and m_j are complementary literals. E.g.,

$$\frac{P_{1,3} \vee P_{2,2}, \qquad \neg P_{2,2}}{P_{1,3}}$$



Resolution is sound and complete for propositional logic

Conversion to CNF

$B_{1,1} \Leftrightarrow (P_{1,2} \vee P_{2,1})$

1. Eliminate $\Leftrightarrow$, replacing $\alpha \Leftrightarrow \beta$ with $(\alpha \Rightarrow \beta) \wedge (\beta \Rightarrow \alpha)$.

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

2. Eliminate $\Rightarrow$, replacing $\alpha \Rightarrow \beta$ with $\neg\alpha \vee \beta$.

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

3. Move $\neg$ inwards using de Morgan's rules and double-negation:

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

4. Apply distributivity law ($\vee$ over $\wedge$) and flatten:

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

Resolution algorithm

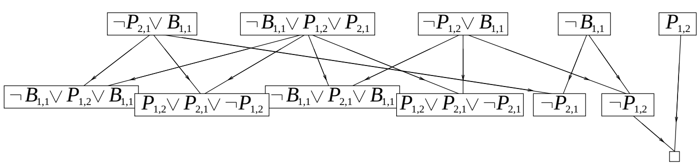
Proof by contradiction, i.e., show $KB \wedge \neg\alpha$ unsatisfiable

```
function PL-Resolution(KB, α) returns true or false
   inputs: KB, the knowledge base, a sentence in propositional logic
           α, the query, a sentence in propositional logic

   clauses ← the set of clauses in the CNF representation of KB ∧ ¬α
   new ← { }
   loop do
       for each Ci, Cj in clauses do
           resolvents ← PL-Resolve(Ci, Cj)
           if resolvents contains the empty clause then return true
           new ← new ∪ resolvents
       if new ⊆ clauses then return false
       clauses ← clauses ∪ new
```


Resolution example

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



DPLL: backtracking++

Backtracking applied to SAT problems:
– variables are proposition symbols, clauses are constraints

Several key improvements:

1. Early termination: stop if all clauses true or any clause false
   e.g., $\{A = true\}$ satisfies $(A \vee B) \wedge (A \vee C)$

2. Pure symbols: symbol has same sign in all as-yet-unsatisfied clauses
   e.g., A and B are pure in $(A \vee \neg B) \wedge (\neg B \vee \neg C) \wedge (C \vee A)$
   $\Rightarrow$ assign symbol to make literals true

3. Unit clauses: clause has exactly one as-yet-unfalsified literal
   e.g., if $\{A = true\}$ already, $(\neg A \vee \neg B)$ is a unit clause
   $\Rightarrow$ assign symbol to make clause true (cf. forward chaining, MRV)

DPLL algorithm

```
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 true
   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])
```

Highly optimized implementation + caching unsolvable subassignments
$\Rightarrow$ modern solvers handle tens of millions of clauses
$\Rightarrow$ practical for large hardware and medium software verification

Propositions and time

Suppose the wumpus-world agent wants to keep track of its location

A sentence such as $L_{1,1} \wedge FacingRight \wedge Forward \Rightarrow L_{2,1}$
doesn't work: after one inference step, $L_{1,1}$ and $L_{2,1}$ are in KB!!

Changeable aspects of world need separate symbols for each time step
e.g., $L^1_{1,1}$ means "Agent is at [1,1] at time step 1", and

$$L^1_{1,1} \wedge FacingRight^1 \wedge Forward^1 \Rightarrow L^2_{2,1}$$

Reflex rules: for every t, we have, e.g., $Glitter^t \Rightarrow Grab^t$

☹Need copies of all axioms involving temporal symbols
for **every** time step (might be infinitely many!)

Tracking changes in the world

State estimation is the general task of keeping track of environment state given a stream of percepts

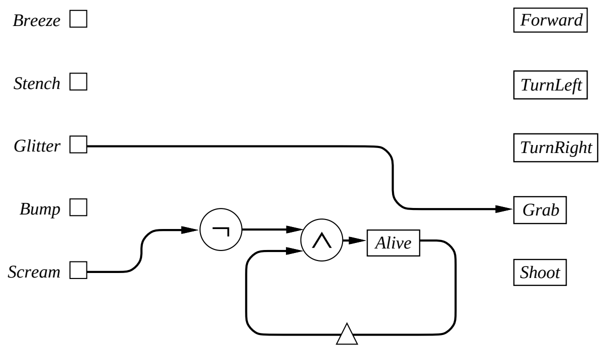
For logic-based systems: maintain a representation of the set of all logically possible world states, given axioms and percepts

Basic trick: successor-state axioms **define** truth of proposition at $t+1$ from propositions at t

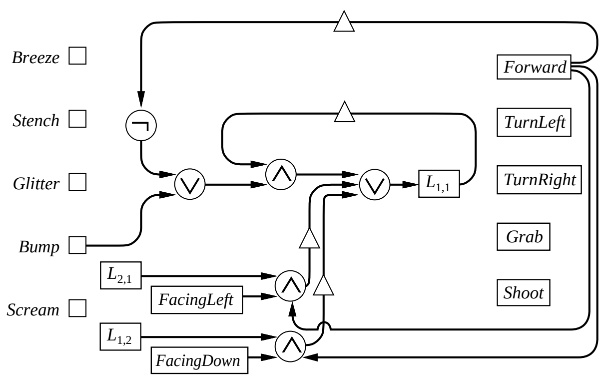
E.g., $Alive^t \Leftrightarrow \neg Scream^t \wedge Alive^{t-1}$

$$\begin{aligned} L^t_{1,1} \Leftrightarrow\ & (L^{t-1}_{1,1} \wedge (\neg Forward^{t-1} \vee Bump^t)) \\ & \vee (L^{t-1}_{1,2} \wedge (FacingDown^{t-1} \wedge Forward^{t-1})) \\ & \vee (L^{t-1}_{2,1} \wedge (FacingLeft^{t-1} \wedge Forward^{t-1})) \end{aligned}$$

Boolean circuit agents



Boolean circuit agents contd.



Summary

Inference methods work by theorem proving or model checking

Forward, backward chaining are linear-time, complete for Horn clauses

Resolution is complete for propositional logic

DPLL is an efficient, complete model checker;
WalkSAT is incomplete but often very fast in practice

Circuit-based agents provide a simple way to handle time
but are usually less complete than inference-based agents