Artificial Intelligence

Dr Amir Pourabdollah

Logical Inference and Programming

Resources

Based on "Artificial Intelligence — A Modern Approach" by Stuart Russel and Peter Norvig, Chapters 6, 8, and 9 (chapter numbers might differ between editions)

Suggested preparatory reading: Chapter 6 (Logical Agents) sections 1 – 5 and Chapter 8 (First-order logic), sections 2 and 3

Based on slides by Stuart Russel, `http://aima.cs.berkeley.edu`

Outline (2 sessions)

Last session:

- ♢ Knowledge-based agents
- ♢ Logic in general—models and entailment
- ♢ Propositional (Boolean) logic
- ♢ Predicate (First-order) Logic
- ♢ Inference

This session:

- ♢ **Inference methods**
  - **– forward chaining**
  - **– backward chaining**
  - **– resolution**
- ♢ **Logic Programming in NLTK**

Reminder: Inference methods

Divided into (roughly) two kinds:

1- Model checking

Truth table enumeration (sound and complete, but exponential growth by the number of variables

Some improvements are possible, such as Davis–Putnam–Logemann–Loveland heuristic
(min-conflicts-like hill-climbing algorithms)
These improvements are sound but incomplete.

- P: Today is rainy
- Q: The ground is wet
- KB: (P => Q) ∧ P
- a=Q: The ground is wet

P	Q	KB: (P->Q) ∧ P	a
T	T	T	T
T	F	F	F
F	T	F	T
F	F	F	F

"a" is proved if for all cases of KB=True, a=True

(we do not care about the cases where KB=False)

> KB is usually treated as a single sentence, i.e., the conjunction (and-operator) of some sentences.

Reminder: Inference Methods

2- Application of **inference rules**

- Legitimate (sound) generation of new sentences from old
- Proof = a sequence of inference rule applications
- Typically requires translation of sentences into a normal form

Two main inference rules

1- Modus Ponens (Latin for mode that affirms)

$$\frac{\alpha \Rightarrow \beta\ ,\ \alpha}{\beta} \quad \text{or extended as:} \quad \frac{\alpha_1 \wedge \alpha_2 \wedge \ldots \wedge \alpha_n \Rightarrow \beta\ ,\ \alpha_1,\ \alpha_2,\ldots,\ \alpha_n}{\beta}$$

2- Unit Resolution

$$\frac{\alpha \vee \beta,\ \sim\beta}{\alpha} \quad \text{or extended as:} \quad \frac{\alpha_1 \vee \alpha_2 \vee \ldots \vee \alpha_n \vee \beta \quad ,\quad \sim\beta}{\alpha_1 \vee \alpha_2 \vee \ldots \vee \alpha_n}$$

The notation means, whenever the top sentences are true, the bottom sentence is inferred.

Reminder: Inference Rules (not to be memorised!)

1. Modens Ponens or **Implication-Elimination**

$$\frac{\alpha \Rightarrow \beta \ , \quad \alpha}{\beta}$$

2. And-Elimination

$$\frac{\alpha_1 \wedge \alpha_2 \wedge \ \ldots . \wedge \alpha_n}{\alpha_i}$$

3. And-Introduction

$$\frac{\alpha_1, \ \alpha_2, \ \ldots . , \alpha_n}{\alpha_1 \wedge \alpha_2 \wedge \ \ldots . \wedge \alpha_n}$$

4. Or-Introduction

$$\frac{\alpha_i}{\alpha_1 \vee \alpha_2 \vee \ldots . \vee \alpha_i \vee \ldots . \vee \alpha_n}$$

5. Double-Negation Elimination

$$\frac{\neg\neg\alpha}{\alpha}$$

6. Unit Resolution

$$\frac{\alpha \vee \beta \ , \ \neg\beta}{\alpha}$$

Forward and backward chaining

Needs to have KB in **Horn Form** (a restriction for KB)

KB must be a **conjunction** of **Horn clauses**

Horn clause: Either:

- ♢ proposition symbol only; or
- ♢ (conjunction of symbols) $\Rightarrow$ symbol,
  in which there is at least one non-negated term on the left side.

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

Inference method: Repeatedly applying Modus Ponens:

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

The method is complete for Horn KBs

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

Inference by **Forward chaining**

Idea: to apply Modus Ponens repeatedly starting from the facts in the KB

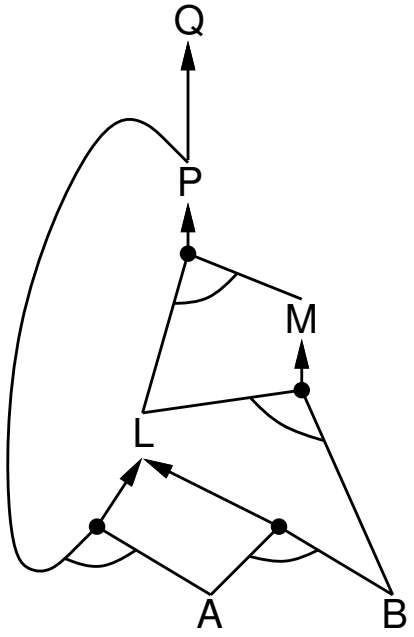
Fire any rule whose premises are satisfied in the KB,
add its conclusion to the KB, until either:
- query is inferred to be true, or:
- no more firing is possible: query cannot be inferred

(NOTE: it does not mean it is necessarily false!)

Example:

Sentences in KB:

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

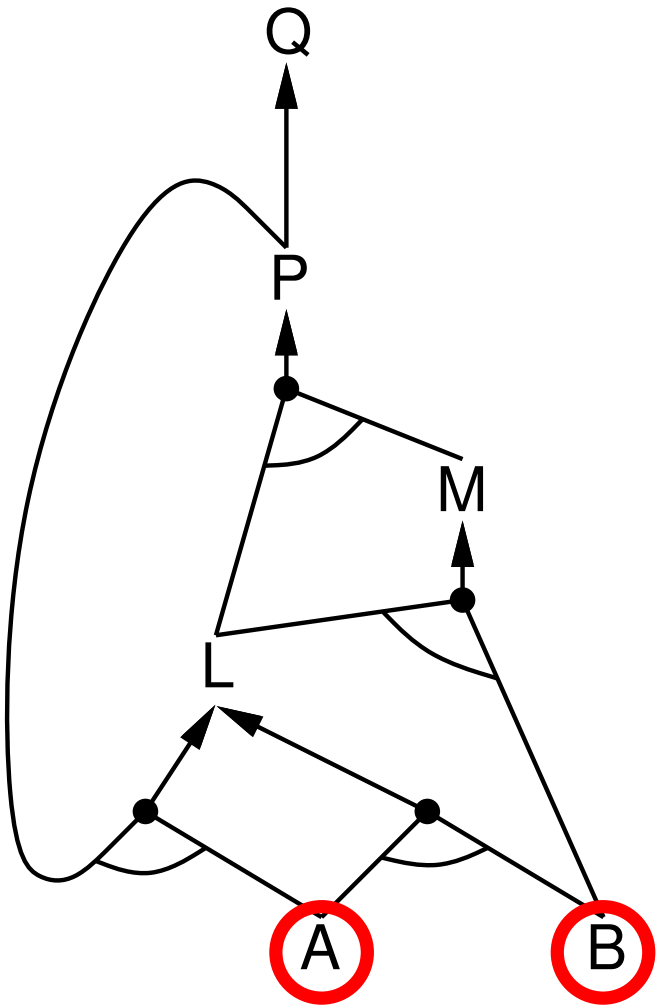


Query: Q (i.e., is Q true?)

Forward chaining example

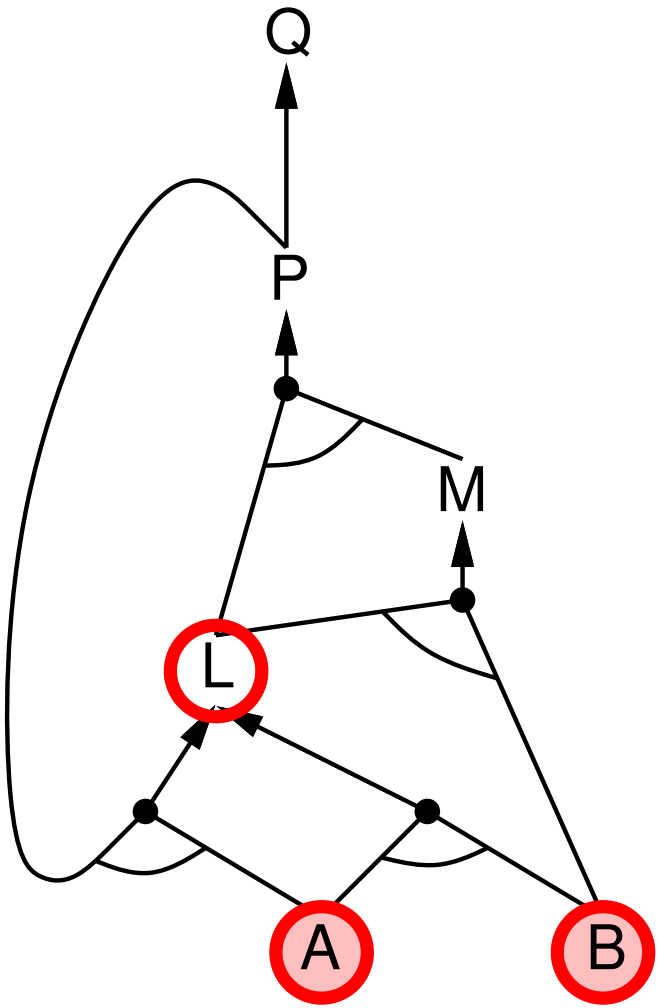
A is true

B is true



Forward chaining example

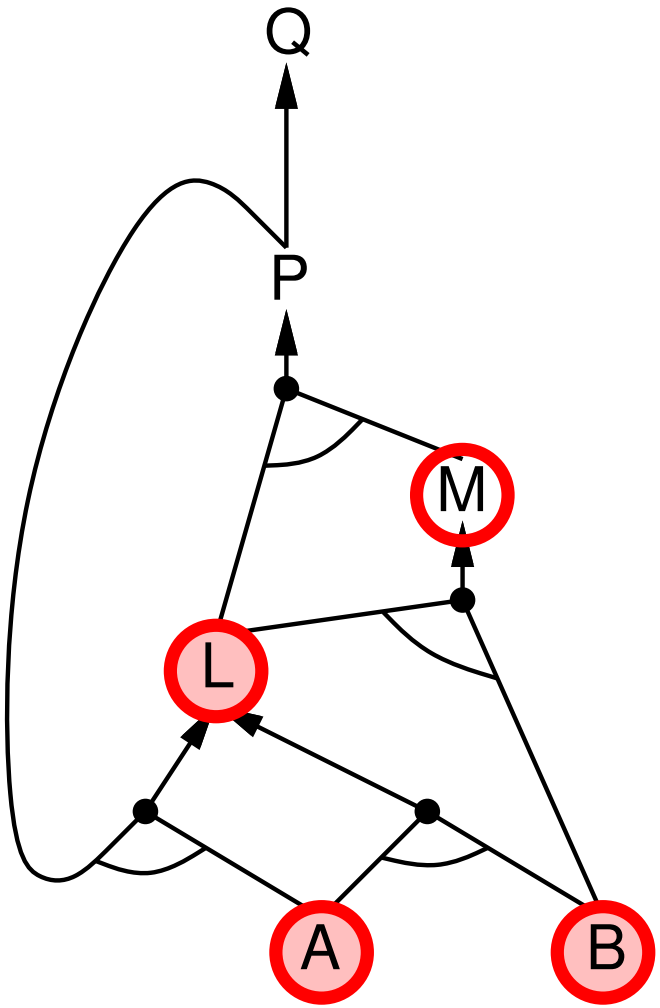
So, L is true



Forward chaining example

L is added to KB

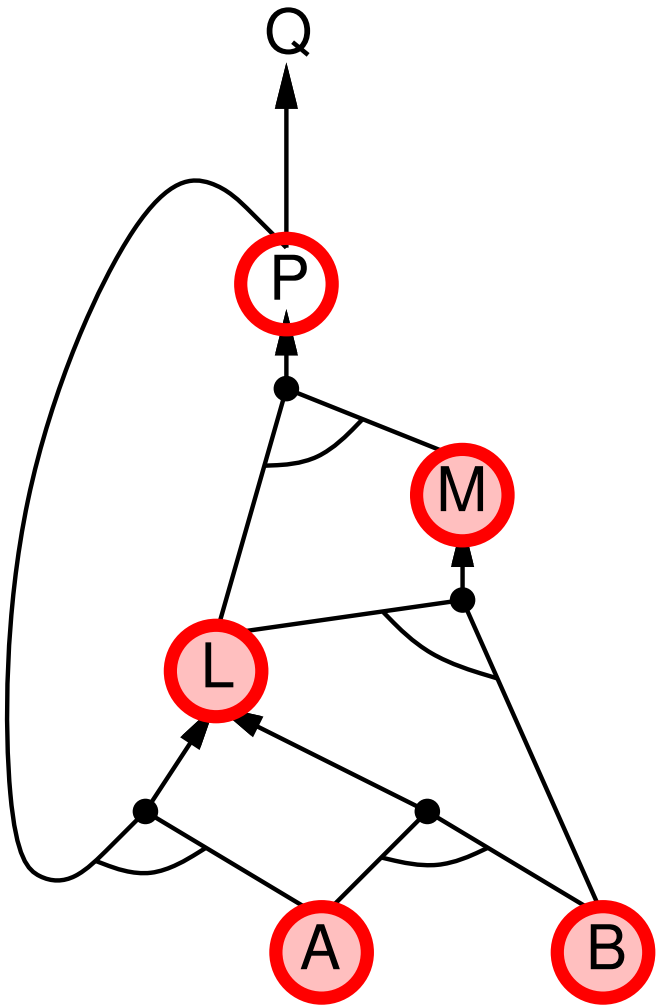
L, B are true
So, M is true.



Forward chaining example

M is added to KB

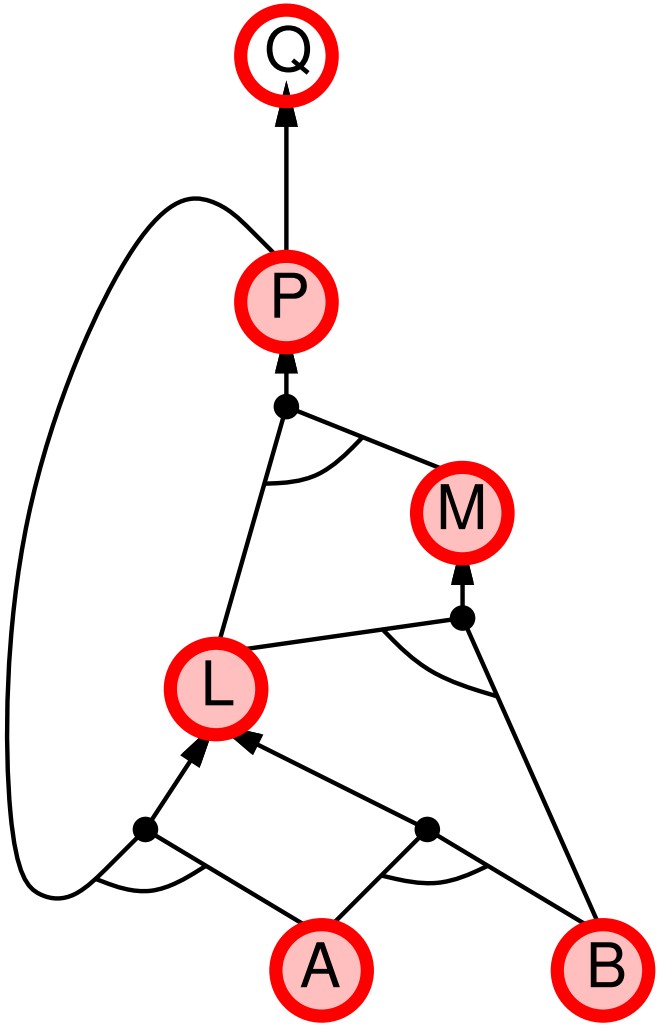
L, M are true
So, P is true.



Forward chaining example

P is added to KB

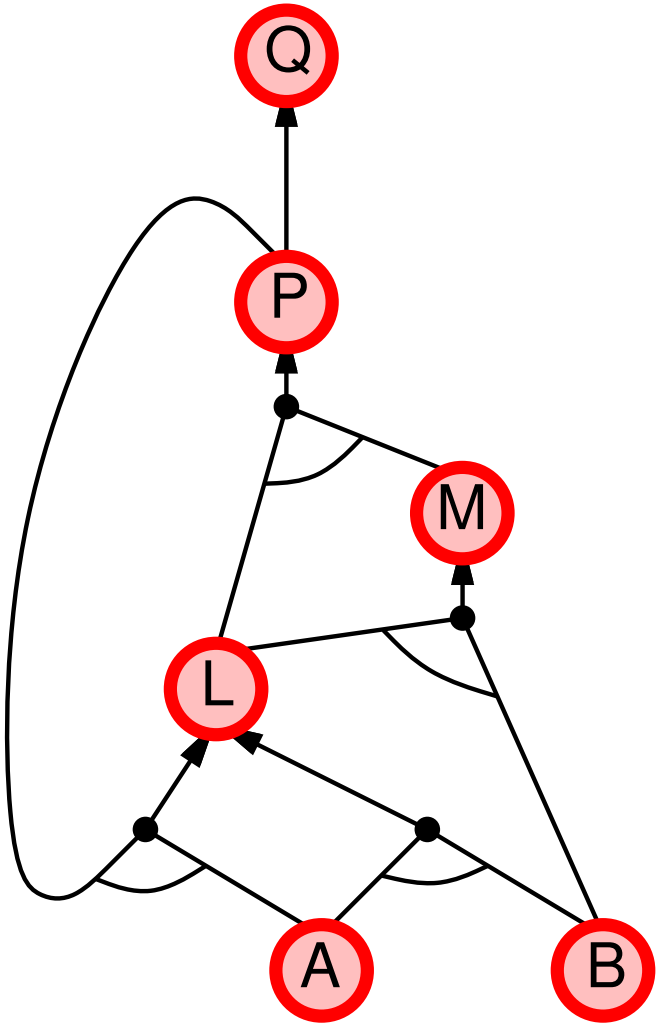
P is true
So, Q is true.



Forward chaining example

Q is added to KB

Inference: Q=True



Inference by **Backward chaining** (BC)

Idea: to apply Modus Ponens backwards from the query Q:

Check any rules whose conclusions are satisfied by Q. For each rule:

- Check if all the rule's premises are already in the KB, or can be inferred by applying the BC (recursively): Then Q is true.
- No (new) rule is found: query cannot be inferred

(NOTE: it does not mean Q is necessarily false!)

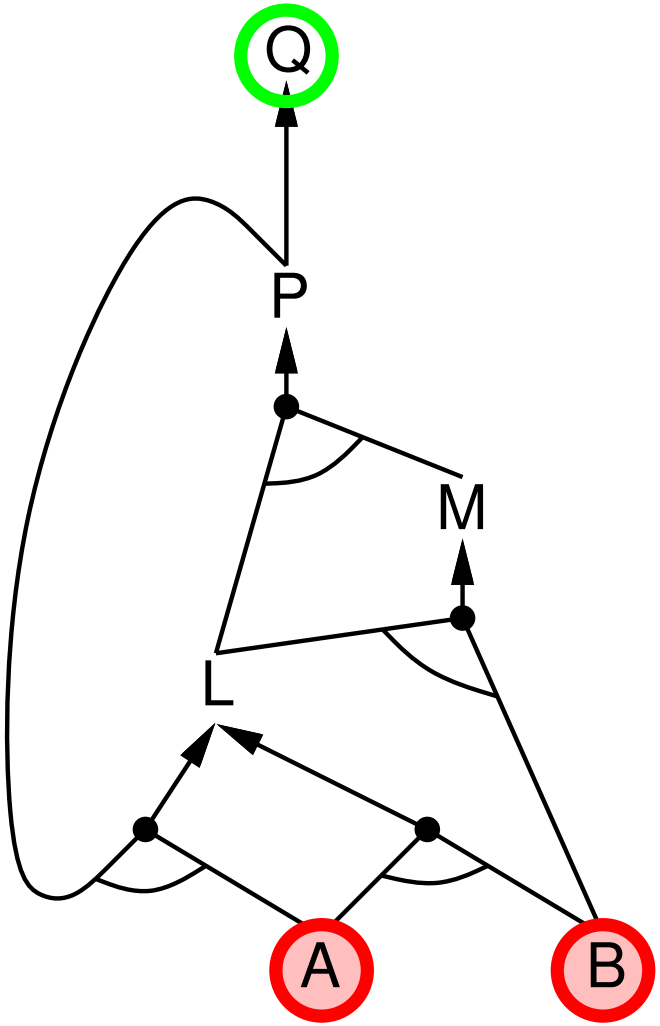
Note: Avoid loops: check if new subgoal is already on the goal stack

Avoid repeated work: check if new subgoal

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

Backward chaining example

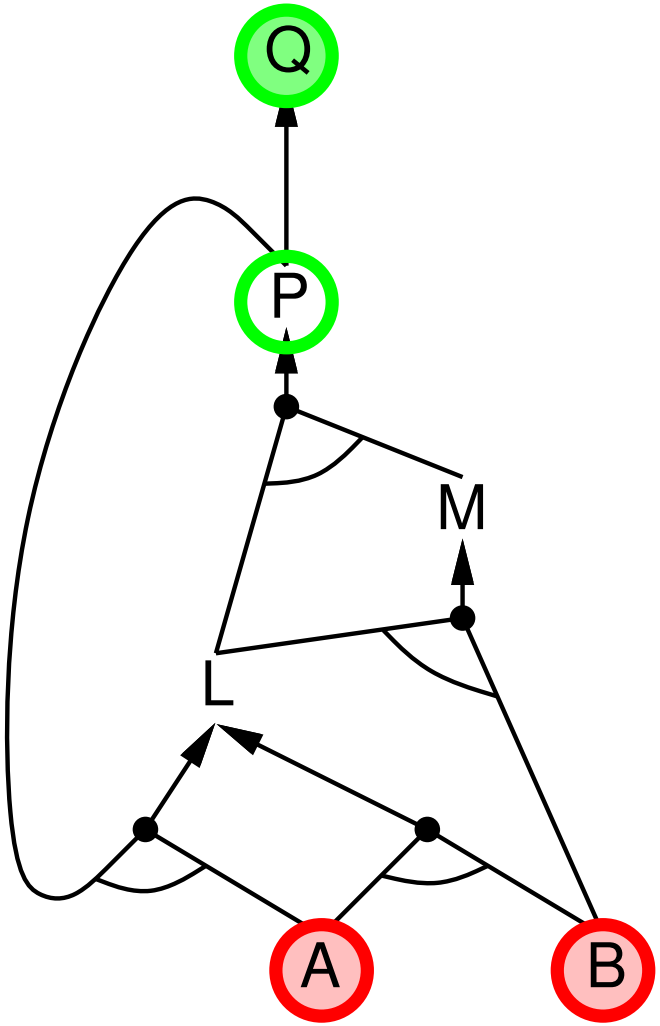
Is Q True?



Backward chaining example

For Q being True,
P must be true

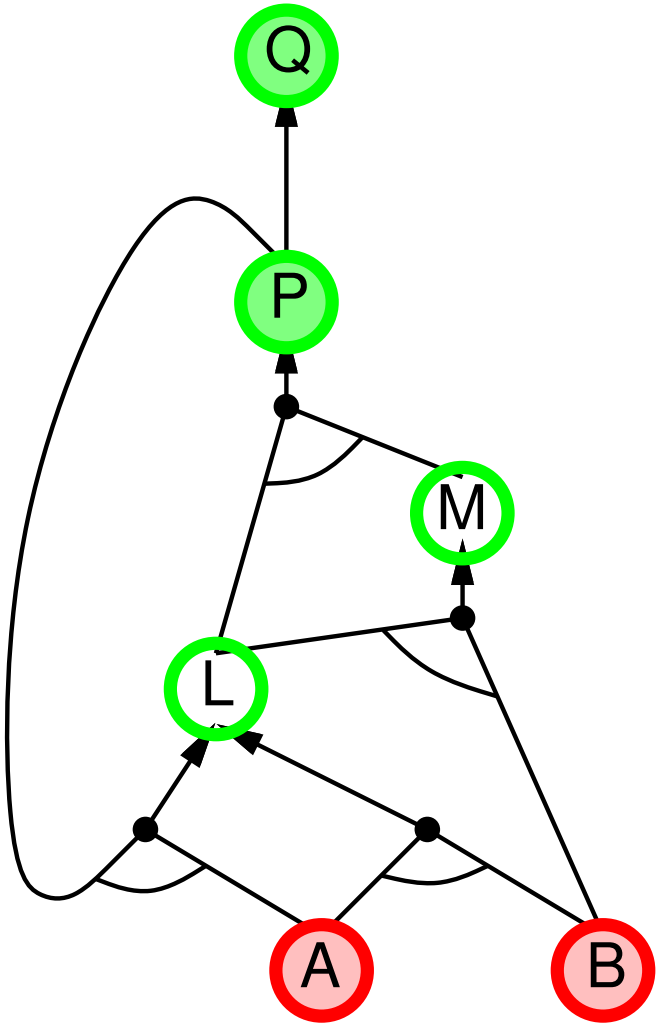
Is P True?
(recursion)



Backward chaining example

For P being True,
L and M must be true

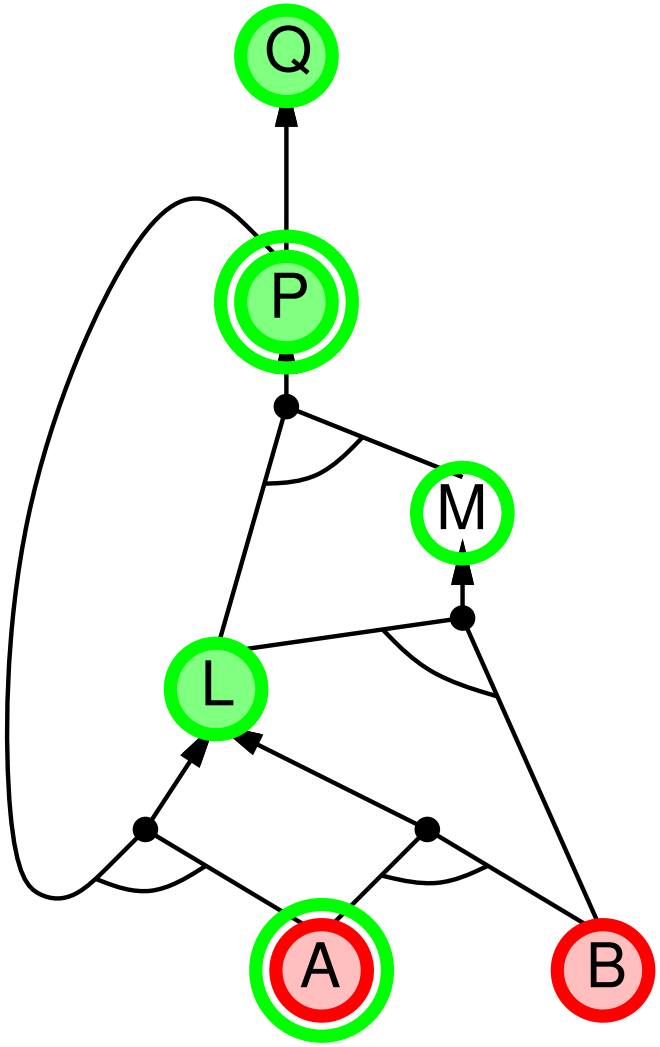
Are L and M True?
(recursion)



Backward chaining example

For L being True,
P and A must be true
Or
A and B must be true

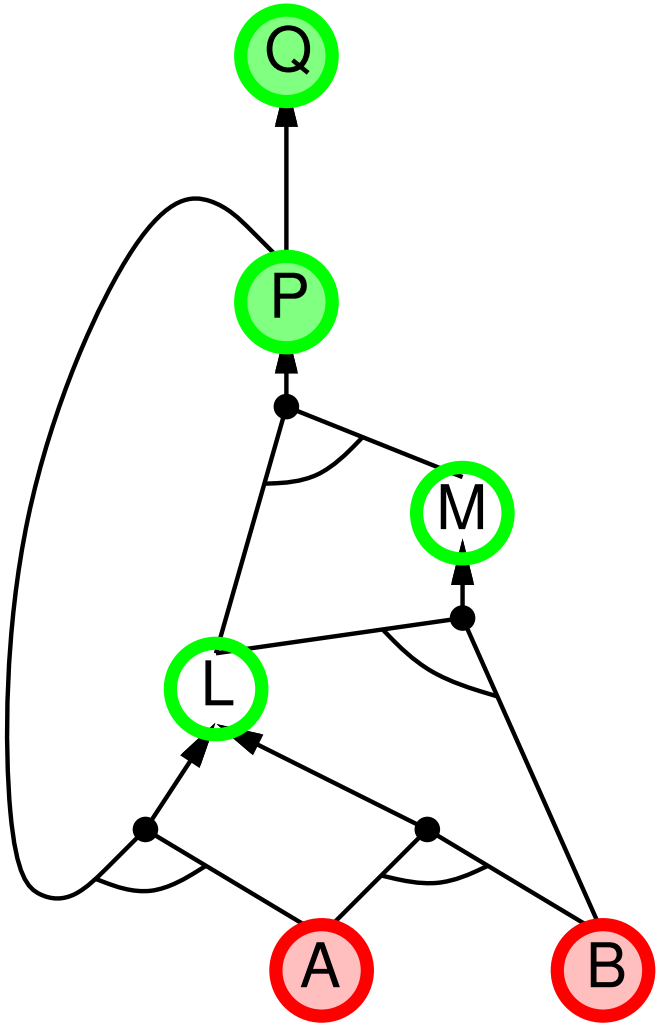
Are P and A True?
(loop on P – no action)



Backward chaining example

For L being True,
P and A must be true
Or
A and B must be true

Are A and B True?
Yes (already in KB)
=> L is true



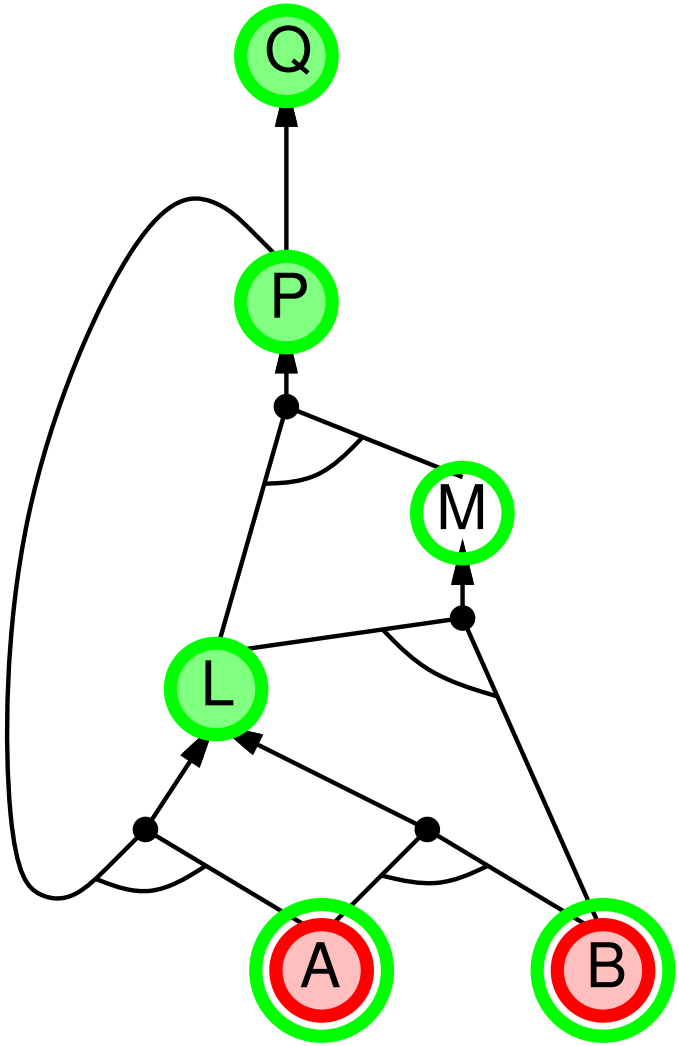
Backward chaining example

For M being True,
L and B must be true

Are L and B True?Yes

=> M is true

All passed rules'
premises are true
=> P is true
=> Q is true
(inference completed)



Inference by **Resolution**

Needs to have KB in **CNF** (Conjunctive Normal Form)
conjunction of **disjunctions** (AND of ORs)

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

Resolution inference rule (for CNF): complete for propositional logic

$$\frac{\alpha \vee \beta, \sim\beta}{\alpha}$$

or extended as:

$$\frac{\alpha_1 \vee \alpha_2 \vee \ldots \vee \alpha_n \vee \beta \quad , \quad \sim\beta}{\alpha_1 \vee \alpha_2 \vee \ldots \vee \alpha_n}$$

(Proof by Contradiction)

Resolution is sound and complete for propositional logic

Conversion to CNF

Initial KB

$R_1: s \Rightarrow p \vee q$

$R_2: p \vee t \Rightarrow r$

$R_3: \sim r$

$R_4: s$

Key operation:

$\alpha \Rightarrow \beta$ is equal to $\neg\alpha \vee \beta$.

$R_1: \sim s \vee p \vee q$

$R_2: \sim (p \vee t) \vee r \equiv (\sim p \wedge \sim t) \vee r$

$\equiv (\sim p \vee r) \wedge (\sim t \vee r)$

$R_3: \sim r$

$R_4: s$

$R_1: \sim s \vee p \vee q$

$R_{2\text{-}1}: \sim p \vee r$

$R_{2\text{-}2}: \sim t \vee r$

$R_3: \sim r$

$R_4: s$

KB in CNF

$$\frac{\alpha \vee \beta,\ \sim\beta}{\alpha}$$

Resolution Algorithm

Idea: To add the negation of query Q to the KB, then checking for any contradiction. If found, Q must have been true.

Algorithm:

1- **KB' =KB$\wedge\sim$Q** (KB': a temporary copy of KB)

2- Look for any patterns of ($\alpha \vee \beta$, $\sim\beta$) in each two sentences of the KB'.

2.1- If found, α is inferred. Check if $\sim\alpha$ is already in the KB.

2.1.1- If not found, add α to the KB': **KB' =KB' $\wedge$ α**. Goto step 2.

2.1.2- else, the algorithm is successful, means **Q is true**.

2.2 - else, the algorithm is unsuccessful, means **Q cannot be inferred** (does not mean Q is false)

Note: Resolution algorithm cannot find if Q is false.
To find this, one can change the query to ~Q and apply the algorithm.
If successful, ~Q must have been true, i.e., Q must have been false.

$$\frac{\alpha \vee \beta,\ \sim\beta}{\alpha}$$

KB Integrity Check

Resolution can also be used to check the KB integrity, i.e. to check if there is any contradiction in it.

A solution: Running the resolution algorithm without adding any new statement to it (i.e., from step 2).

In other words, Apply the resolution for Q=NULL.

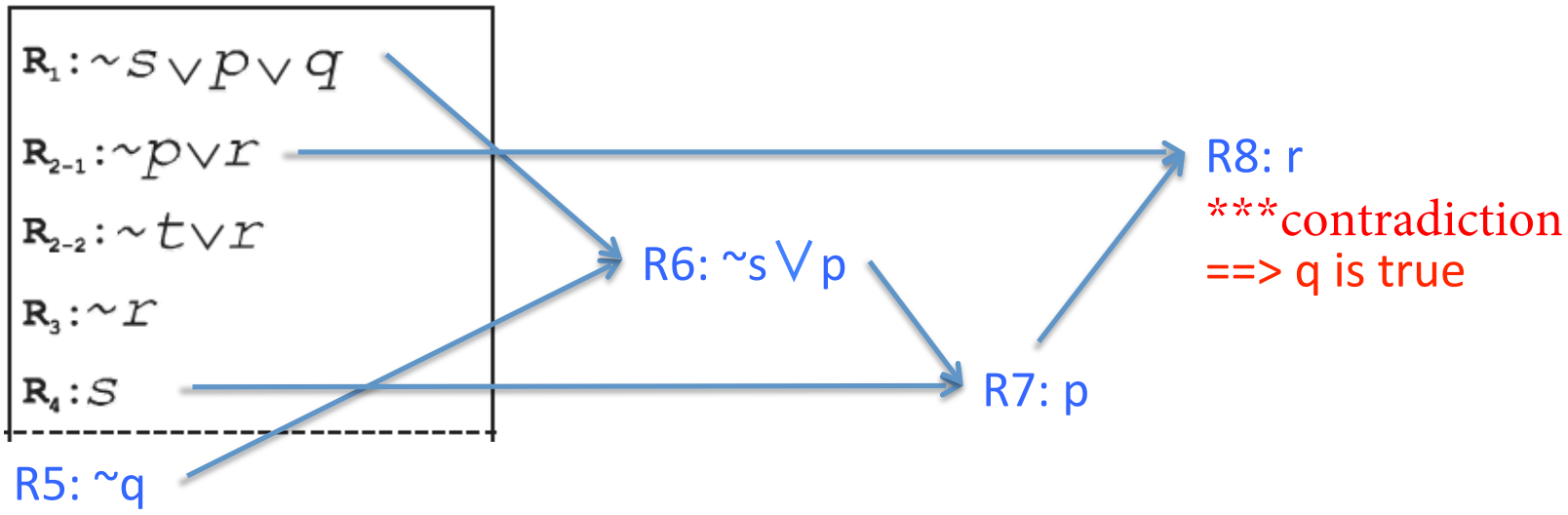
- If resolution is successful, it means the NULL sentence is necessarily true: something must have been wrong with the KB!
  - This is an indication of KB being inconsistent, i.e., having an internal contradiction.

$$\frac{\alpha \vee \beta,\ \sim\beta}{\alpha}$$

Example

Query: q

KB:



The negate of the query is added to the end of the KB

------>

Finding patterns of
α ∨ β, ~β
(this will infers α)
If found, adding α to KB

------>

No more pattern found -> no success
Any contradiction -> query was true

Inference in first-order Logic

The inference methods in propositional logic can be extended to the first-order logic.

There are extended versions of the propositional inference techniques, such as Generalised Modes Ponens (GMP) and Generalised Resolution that can work with variable terms.

Although the underlying theory may look a bit complex, it is computationaly easy: Keep the variable terms in the quantifiers ($\exists$, $\forall$) as wildcards, and replace them with different values within inference algorithms.

For example, "$\forall$ x King(x) $\wedge$ Greedy(x) $\Rightarrow$ Evil(x)" can be used for inference just by "King(x) $\wedge$ Greedy(x) $\Rightarrow$ Evil(x)" where x can be replaced by a constant as needed within the program - examples to follow.

Existential qualifiers ($\exists$) can be expressed with their equivalent universal quantifier ($\forall$) forms, and shown as above.

Logic Programming in NLTK

NLTK library provides some functions and classes for FOL.

It has a specific syntax for representing logical statements.
Examples at http://www.nltk.org/howto/logic.html

```
negation            -
conjunction         &
disjunction         |
implication         ->

Example KB:
human(x) -> mortal(x)
human(Tim)
day(y) & !cloudy(y) -> sunny(y)
```

In NLTK, terms starting with a single letter are treated as wildcards (e.g., x or z2).

- Important: Do not use same variable names for different KB statement.

Logic Programming in NLTK

```
from nltk.sem import Expression
from nltk.inference import ResolutionProver
read_expr = Expression.fromstring

p1 = read_expr('British (x) -> European (x)')
p2 = read_expr('British (Tim)')
p3 = read_expr('-European (Jim)')
kb=[p1,p2,p3]
q  = read_expr('European(Tim)')
r=ResolutionProver().prove(q, kb, verbose=True)
print(r)
```

```
[1] {-European(Tim)}                    A
[2] {-British(z21), European(z21)}      A
[3] {British(Tim)}                      A
[4] {-European(Jim)}                    A
[5] {-British(Tim)}                     (1, 2)
[6] {European(Tim)}                     (2, 3)
[7] {}                                  (1, 6)

True
```

- With verbose=True, ResolutionProver prints the inference steps
- z21: An arbitrary variable name internally given to x (x is used as a wildcard)
- Comma in [2] means OR (since the algorithm works with CNF)
- p1 is converted to CNF (reminder: "$\alpha \Rightarrow \beta$" is the same as "$\neg\alpha \vee \beta$")
- step [1]: Added ~q to the KB
- steps [2-4] were already in the KB (noted by "A")
- [5] Added to the KB by resolution between [1],[2]
- [6] added to the KB by resolution between [2],[3]
- step [7]: Contradiction found between 1,6
- => q is true because of the found contradiction

Logic Programming in NLTK

```
p1 = read_expr('British (x) -> European (x)')
p2 = read_expr('British (Tim)')
p3 = read_expr('-European (Jim)')
kb=[p1,p2,p3]
q  = read_expr('British (Jim)')
r=ResolutionProver().prove(q, kb, verbose=True)
print(r)
```

```
[1] {-British(Jim)}                 A
[2] {-British(z30), European(z30)}  A
[3] {British(Tim)}                  A
[4] {-European(Jim)}                A
[5] {European(Tim)}                 (2, 3)
[6] {-British(Jim)}                 (2, 4)

False
```

- Same KB with different q
- The algorithm stopped with no contradiction found.
- False here means the algorithm was unsuccessful, but does not mean q is false (although q is actually false in this case)

Logic Programming in NLTK

```
p1 = read_expr('brother(Jim, Tim)')
p2 = read_expr('brother(x,y) & man(y) -> brother(y,x)')
p3 = read_expr('man(Tim)')
kb=[p1,p2,p3]
q  = read_expr('brother(Tim, Jim)')
r=ResolutionProver().prove(q, kb, verbose=True)
print(r)
```

Another example with a multi-term predicate: brother(x,y)

q: Is Tim brother of Jim?

```
[ 1] {-brother(Tim,Jim)}                                  A
[ 2] {brother(Jim,Tim)}                                   A
[ 3] {-brother(z39,z40), -man(z40), brother(z40,z39)}     A
[ 4] {man(Tim)}                                           A
[ 5] {-brother(Jim,Tim), -man(Tim)}                       (1, 3)
[ 6] {-man(Tim), brother(Tim,Jim)}                        (2, 3)
[ 7] {-man(Tim)}                                          (1, 6)
[ 8] {-man(Tim)}                                          (2, 5)
[ 9] {brother(Tim,z39), -brother(z39,Tim)}                (3, 4)
[10] {-brother(Jim,Tim)}                                  (1, 9)
[11] {brother(Tim,Jim)}                                   (2, 9)
[12] {}                                                   (1, 11)

True
```

Logic Programming in Prolog

A much earlier programming language specific to logics from 1972 (also LISP of 60s).

Still active. IBM Watson also used Prolog.

FOL example:

```
 1: orbits(mercury, sun).
 2: orbits(venus,   sun).
 3: orbits(earth,   sun).
 4: orbits(mars,    sun).
 5:
 6: orbits(moon, earth).
 7:
 8: orbits(phobos, mars).
 9: orbits(deimos, mars).
10:
11: planet(P)    <= orbits(P, sun).
12: satellite(S) <= orbits(S, P) and planet(P).
?satellite(moon)

 --- running ---
satellite(moon) yes
```

Summary

- Inference algorithms:
  - Forward/backward chaining (complete if KB in Horn clauses)
  - Resolution for propositional/FOL logic (complete if KB in CNF)
  - Logic programming: Python (NLTK), Prolog