Forward and Backward Chaining

Outline

I. Forward chaining in FOL

II. Backward chaining

III. Prolog

* Figures are from the textbook site.

I. First-Order Definite Clauses

A *first-order definite clause* is a disjunction of literals of which exactly one is positive.

I. First-Order Definite Clauses

A *first-order definite clause* is a disjunction of literals of which exactly one is positive.

- single positive literal

$$Bird(Ostrich)$$

I. First-Order Definite Clauses

A *first-order definite clause* is a disjunction of literals of which exactly one is positive.

- single positive literal

$$Bird(Ostrich)$$

- an implication whose premise is a conjunction of positive literals and whose conclusion is a single positive literal

$$Human(Socrates) \Rightarrow Fallible(Socrates)$$

I. First-Order Definite Clauses

A *first-order definite clause* is a disjunction of literals of which exactly one is positive.

- single positive literal

$$\mathit{Bird}(\mathit{Ostrich})$$

- an implication whose premise is a conjunction of positive literals and whose conclusion is a single positive literal

$$\mathit{Human}(\mathit{Socrates}) \Rightarrow \mathit{Fallible}(\mathit{Socrates})$$

- variables allowed and implicitly under universal quantification

$\mathit{Human}(x) \Rightarrow \mathit{Fallible}(x)$ // interpreted as $\forall x \; \mathit{Human}(x) \Rightarrow \mathit{Fallible}(x)$

I. First-Order Definite Clauses

A *first-order definite clause* is a disjunction of literals of which exactly one is positive.

- single positive literal

$$Bird(Ostrich)$$

- an implication whose premise is a conjunction of positive literals and whose conclusion is a single positive literal

$$Human(Socrates) \Rightarrow Fallible(Socrates)$$

- variables allowed and implicitly under universal quantification

$Human(x) \Rightarrow Fallible(x)$ // interpreted as $\forall x \; Human(x) \Rightarrow Fallible(x)$

$Gate(g) \land Terminal(t) \; \Rightarrow g \neq t$ // interpreted as $\forall g, t \; Gate(g) \land Terminal(t) \; \Rightarrow g \neq t$

I. First-Order Definite Clauses

A *first-order definite clause* is a disjunction of literals of which exactly one is positive.

- single positive literal

$$\mathit{Bird}(\mathit{Ostrich})$$

- an implication whose premise is a conjunction of positive literals and whose conclusion is a single positive literal

$$\mathit{Human}(\mathit{Socrates}) \Rightarrow \mathit{Fallible}(\mathit{Socrates})$$

- variables allowed and implicitly under universal quantification

$\mathit{Human}(x) \Rightarrow \mathit{Fallible}(x)$ // interpreted as $\forall x\ \mathit{Human}(x) \Rightarrow \mathit{Fallible}(x)$

$\mathit{Gate}(g) \wedge \mathit{Terminal}(t) \Rightarrow g \neq t$ // interpreted as $\forall g, t\ \mathit{Gate}(g) \wedge \mathit{Terminal}(t) \Rightarrow g \neq t$

- existential quantifiers not allowed

Translation of Sentences

The law says that it is a crime for an American to sell weapons to hostile nations. The country Nono, an enemy of America, has some missiles, and all of its missiles were sold to it by Colonel West, who is American.

Translation of Sentences

The law says that it is a crime for an American to sell weapons to hostile nations. The country Nono, an enemy of America, has some missiles, and all of its missiles were sold to it by Colonel West, who is American.

"… *it is a crime for an American to sell weapons to hostile nations*":

Translation of Sentences

The law says that it is a crime for an American to sell weapons to hostile nations. The country Nono, an enemy of America, has some missiles, and all of its missiles were sold to it by Colonel West, who is American.

"… *it is a crime for an American to sell weapons to hostile nations*": // $\forall x$ ...

Translation of Sentences

The law says that it is a crime for an American to sell weapons to hostile nations. The country Nono, an enemy of America, has some missiles, and all of its missiles were sold to it by Colonel West, who is American.

"… *it is a crime for an American to sell weapons to hostile nations":* // $\forall x$ ...

American$(x) \wedge$ *Weapon*$(y) \wedge$ *Hostile*$(z) \wedge$ *Sells*$(x, y, z) \Rightarrow$ *Criminal*(x)

Translation of Sentences

The law says that it is a crime for an American to sell weapons to hostile nations. The country Nono, an enemy of America, has some missiles, and all of its missiles were sold to it by Colonel West, who is American.

"… *it is a crime for an American to sell weapons to hostile nations":* // $\forall x$ ...

$$American(x) \land Weapon(y) \land Hostile(z) \land Sells(x, y, z) \Rightarrow Criminal(x)$$

"*Nono … has some missles":*

Translation of Sentences

The law says that it is a crime for an American to sell weapons to hostile nations. The country Nono, an enemy of America, has some missiles, and all of its missiles were sold to it by Colonel West, who is American.

"… *it is a crime for an American to sell weapons to hostile nations":* // $\forall x$ ...

$$\mathit{American}(x) \wedge \mathit{Weapon}(y) \wedge \mathit{Hostile}(z) \wedge \mathit{Sells}(x, y, z) \Rightarrow \mathit{Criminal}(x)$$

"*Nono … has some missles":* // $\exists x\ \mathit{Owns}(\mathit{Nono}, x) \wedge \mathit{Missile}(x)$

Translation of Sentences

The law says that it is a crime for an American to sell weapons to hostile nations. The country Nono, an enemy of America, has some missiles, and all of its missiles were sold to it by Colonel West, who is American.

"*… it is a crime for an American to sell weapons to hostile nations*": // $\forall x$ ...

American$(x) \wedge$ *Weapon*$(y) \wedge$ *Hostile*$(z) \wedge$ *Sells*$(x, y, z) \Rightarrow$ *Criminal*(x)

"*Nono … has some missles*": // $\exists x$ *Owns*$($*Nono*$, x) \wedge$ *Missile*(x)

Owns$($*Nono*$, M_1)$

Missile(M_1)

introducing a Skolem constant to eliminate ∃

Translation of Sentences

The law says that it is a crime for an American to sell weapons to hostile nations. The country Nono, an enemy of America, has some missiles, and all of its missiles were sold to it by Colonel West, who is American.

"… *it is a crime for an American to sell weapons to hostile nations*": // $\forall x$ ...

$\mathit{American}(x) \wedge \mathit{Weapon}(y) \wedge \mathit{Hostile}(z) \wedge \mathit{Sells}(x, y, z) \Rightarrow \mathit{Criminal}(x)$

"*Nono … has some missles*": // $\exists x\ \mathit{Owns}(\mathit{Nono}, x) \wedge \mathit{Missile}(x)$

$\mathit{Owns}(\mathit{Nono}, M_1)$

$\mathit{Missile}(M_1)$

introducing a Skolem constant to eliminate ∃

"*All of its missiles were sold by Colonel West*": // $\forall x$ ...

Translation of Sentences

The law says that it is a crime for an American to sell weapons to hostile nations. The country Nono, an enemy of America, has some missiles, and all of its missiles were sold to it by Colonel West, who is American.

"… *it is a crime for an American to sell weapons to hostile nations*": // $\forall x$ ...

$American(x) \land Weapon(y) \land Hostile(z) \land Sells(x, y, z) \Rightarrow Criminal(x)$

"*Nono … has some missles*": // $\exists x \; Owns(Nono, x) \land Missile(x)$

$Owns(Nono, M_1)$

$Missile(M_1)$

introducing a Skolem
constant to eliminate ∃

"*All of its missiles were sold by Colonel West*": // $\forall x$ ...

$Missile(x) \land Owns(Nono, x) \Rightarrow Sells(West, x, Nono)$

Completing the KB

Completing the KB

Need to know that missiles are weapons.

$$Missile(x) \Rightarrow Weapon(x)$$

Completing the KB

Need to know that missiles are weapons.

$$Missile(x) \Rightarrow Weapon(x)$$

Also need to know that an enemy of America counts as “hostile”.

$$Enemy(x, America) \Rightarrow Hostile(x)$$

Completing the KB

Need to know that missiles are weapons.

$$\textit{Missile}(x) \Rightarrow \textit{Weapon}(x)$$

Also need to know that an enemy of America counts as "hostile".

$$\textit{Enemy}(x, \textit{America}) \Rightarrow \textit{Hostile}(x)$$

"*West, who is American* …

$$\textit{American}(\textit{West})$$

Completing the KB

Need to know that missiles are weapons.

$$\mathit{Missile}(x) \Rightarrow \mathit{Weapon}(x)$$

Also need to know that an enemy of America counts as “hostile”.

$$\mathit{Enemy}(x, \mathit{America}) \Rightarrow \mathit{Hostile}(x)$$

“*West, who is American* …

$$\mathit{American}(\mathit{West})$$

“*The country Nono, an enemy of America* …”

$$\mathit{Enemy}(\mathit{Nono}, \mathit{America})$$

Completing the KB

Need to know that missiles are weapons.

$$Missile(x) \Rightarrow Weapon(x)$$

Also need to know that an enemy of America counts as "hostile".

$$Enemy(x, America) \Rightarrow Hostile(x)$$

"*West, who is American* …

$$American(West)$$

"*The country Nono, an enemy of America* …"

$$Enemy(Nono, America)$$

The *KB* consists of first-order definite clauses with no function symbols.
It is called a *Datalog*.

Simple Forward Chaining

1. Start from the known facts.
2. Trigger all the rules whose premises are satisfied.
3. Add their conclusions to the known facts.
4. Repeat steps 2 and 3 until one of the following situations occurs:
    a. The query is answered.
    b. No new facts are added.

Simple Forward Chaining

1. Start from the known facts.
2. Trigger all the rules whose premises are satisfied.
3. Add their conclusions to the known facts.
4. Repeat steps 2 and 3 until one of the following situations occurs:
   a. The query is answered.
   b. No new facts are added.

A *new fact* is not a renaming of a known fact.

Simple Forward Chaining

1. Start from the known facts.
2. Trigger all the rules whose premises are satisfied.
3. Add their conclusions to the known facts.
4. Repeat steps 2 and 3 until one of the following situations occurs:
   a. The query is answered.
   b. No new facts are added.

A *new fact* is not a renaming of a known fact.

Likes(x,*IceCream*) is a renaming of *Likes*(y,*IceCream*).
Both have the meaning: "Everyone likes ice cream".

Execution of Forward Chaining

KB:

$American(x) \land Weapon(y) \land Hostile(z) \land Sells(x, y, z) \Rightarrow Criminal(x)$

$Owns(Nono, M_1)$

$Missile(M_1)$

$Missile(x) \land Owns(Nono, x) \Rightarrow Sells(West, x, Nono)$

$Missile(x) \Rightarrow Weapon(x)$

$Enemy(x, America) \Rightarrow Hostile(x)$

$American(West)$

$Enemy(Nono, America)$

Execution of Forward Chaining

KB:

$American(x) \land Weapon(y) \land Hostile(z) \land Sells(x, y, z) \Rightarrow Criminal(x)$

$Owns(Nono, M_1)$

$Missile(M_1)$

$Missile(x) \land Owns(Nono, x) \Rightarrow Sells(West, x, Nono)$

$Missile(x) \Rightarrow Weapon(x)$

$Enemy(x, America) \Rightarrow Hostile(x)$

$American(West)$

$Enemy(Nono, America)$

Iteration 1 adds:

Execution of Forward Chaining

KB:

$American(x) \wedge Weapon(y) \wedge Hostile(z) \wedge Sells(x, y, z) \Rightarrow Criminal(x)$

$Owns(Nono, M_1)$

$Missile(M_1)$

$Missile(x) \wedge Owns(Nono, x) \Rightarrow Sells(West, x, Nono)$

$Missile(x) \Rightarrow Weapon(x)$

$Enemy(x, America) \Rightarrow Hostile(x)$

$American(West)$

$Enemy(Nono, America)$

Iteration 1 adds:

$Sell(West, M_1, Nono)$

Execution of Forward Chaining

KB:

$American(x) \wedge Weapon(y) \wedge Hostile(z) \wedge Sells(x, y, z) \Rightarrow Criminal(x)$

$Owns(Nono, M_1)$

$Missile(M_1)$

$Missile(x) \wedge Owns(Nono, x) \Rightarrow Sells(West, x, Nono)$

$Missile(x) \Rightarrow Weapon(x)$

$Enemy(x, America) \Rightarrow Hostile(x)$

$American(West)$

$Enemy(Nono, America)$

Iteration 1 adds:

$Sell(West, M_1, Nono)$

$Weapon(M_1)$

Execution of Forward Chaining

KB:

$American(x) \wedge Weapon(y) \wedge Hostile(z) \wedge Sells(x, y, z) \Rightarrow Criminal(x)$

$Owns(Nono, M_1)$

$Missile(M_1)$

$Missile(x) \wedge Owns(Nono, x) \Rightarrow Sells(West, x, Nono)$

$Missile(x) \Rightarrow Weapon(x)$

$Enemy(x, America) \Rightarrow Hostile(x)$

$American(West)$

$Enemy(Nono, America)$

Iteration 1 adds:

$Sell(West, M_1, Nono)$

$Weapon(M_1)$

Execution of Forward Chaining

KB:

$American(x) \land Weapon(y) \land Hostile(z) \land Sells(x, y, z) \Rightarrow Criminal(x)$

$Owns(Nono, M_1)$

$Missile(M_1)$

$Missile(x) \land Owns(Nono, x) \Rightarrow Sells(West, x, Nono)$

$Missile(x) \Rightarrow Weapon(x)$

$Enemy(x, America) \Rightarrow Hostile(x)$

$American(West)$

$Enemy(Nono, America)$

Iteration 1 adds:

$Sell(West, M_1, Nono)$

$Weapon(M_1)$

$Hostile(Nono)$

Execution of Forward Chaining

KB:

$American(x) \land Weapon(y) \land Hostile(z) \land Sells(x, y, z) \Rightarrow Criminal(x)$

$Owns(Nono, M_1)$

$Missile(M_1)$

$Missile(x) \land Owns(Nono, x) \Rightarrow Sells(West, x, Nono)$

$Missile(x) \Rightarrow Weapon(x)$

$Enemy(x, America) \Rightarrow Hostile(x)$

$American(West)$

$Enemy(Nono, America)$

Iteration 1 adds:

$Sell(West, M_1, Nono)$

$Weapon(M_1)$

$Hostile(Nono)$

Iteration 2 adds:

Execution of Forward Chaining

KB:

$American(x) \wedge Weapon(y) \wedge Hostile(z) \wedge Sells(x, y, z) \Rightarrow Criminal(x)$

$Owns(Nono, M_1)$

$Missile(M_1)$

$Missile(x) \wedge Owns(Nono, x) \Rightarrow Sells(West, x, Nono)$

$Missile(x) \Rightarrow Weapon(x)$

$Enemy(x, America) \Rightarrow Hostile(x)$

$American(West)$

$Enemy(Nono, America)$

Iteration 1 adds:

$Sell(West, M_1, Nono)$

$Weapon(M_1)$

$Hostile(Nono)$

Iteration 2 adds:

$Criminal(West)$

Execution of Forward Chaining

KB:

$American(x) \wedge Weapon(y) \wedge Hostile(z) \wedge Sells(x, y, z) \Rightarrow Criminal(x)$

$Owns(Nono, M_1)$

$Missile(M_1)$

$Missile(x) \wedge Owns(Nono, x) \Rightarrow Sells(West, x, Nono)$

$Missile(x) \Rightarrow Weapon(x)$

$Enemy(x, America) \Rightarrow Hostile(x)$

$American(West)$

$Enemy(Nono, America)$

Iteration 1 adds:

$Sell(West, M_1, Nono)$

$Weapon(M_1)$

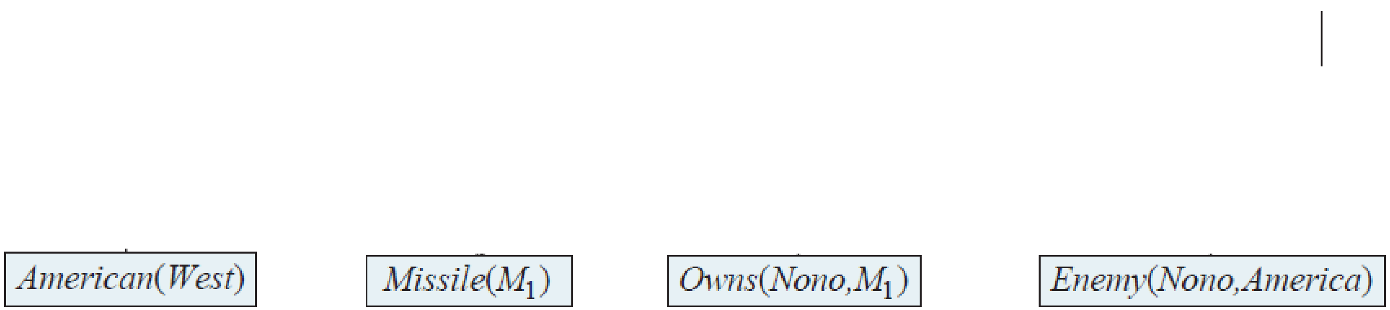
$Hostile(Nono)$

Iteration 2 adds:

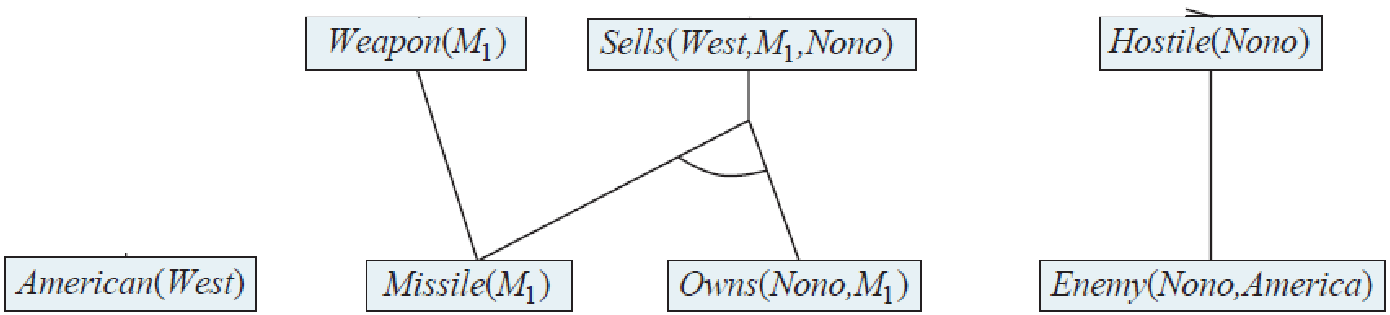
$Criminal(West)$

KB has now reached a *fixed point*, meaning that no new sentences are possible.

Proof Tree

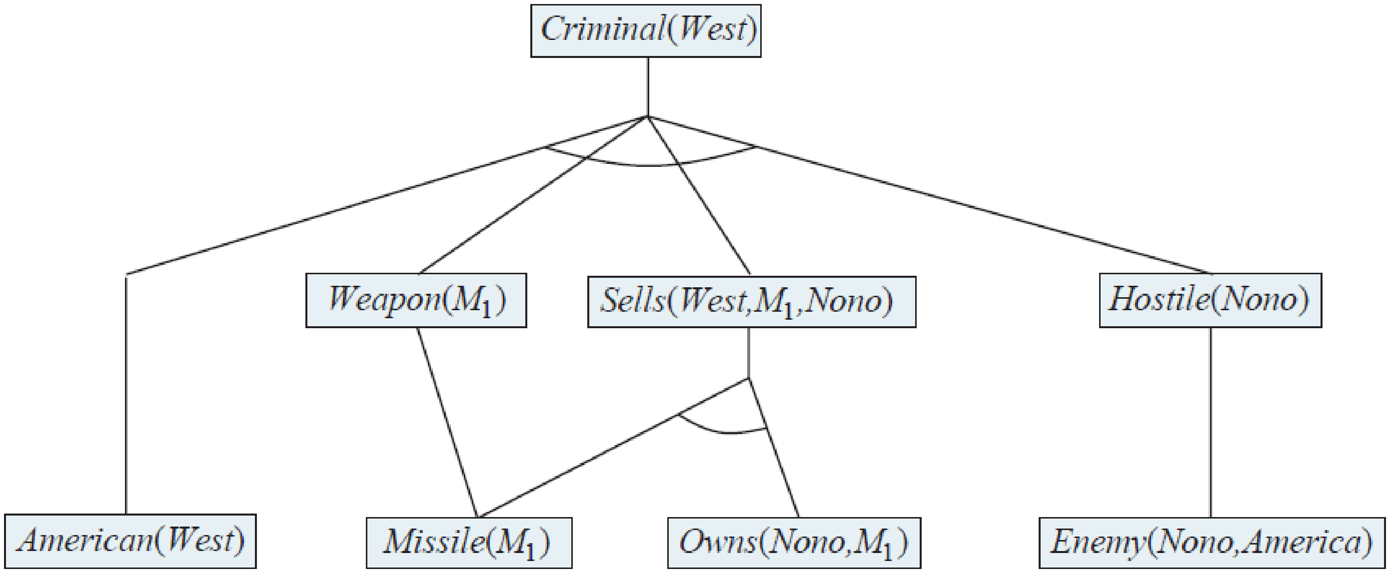


Proof Tree

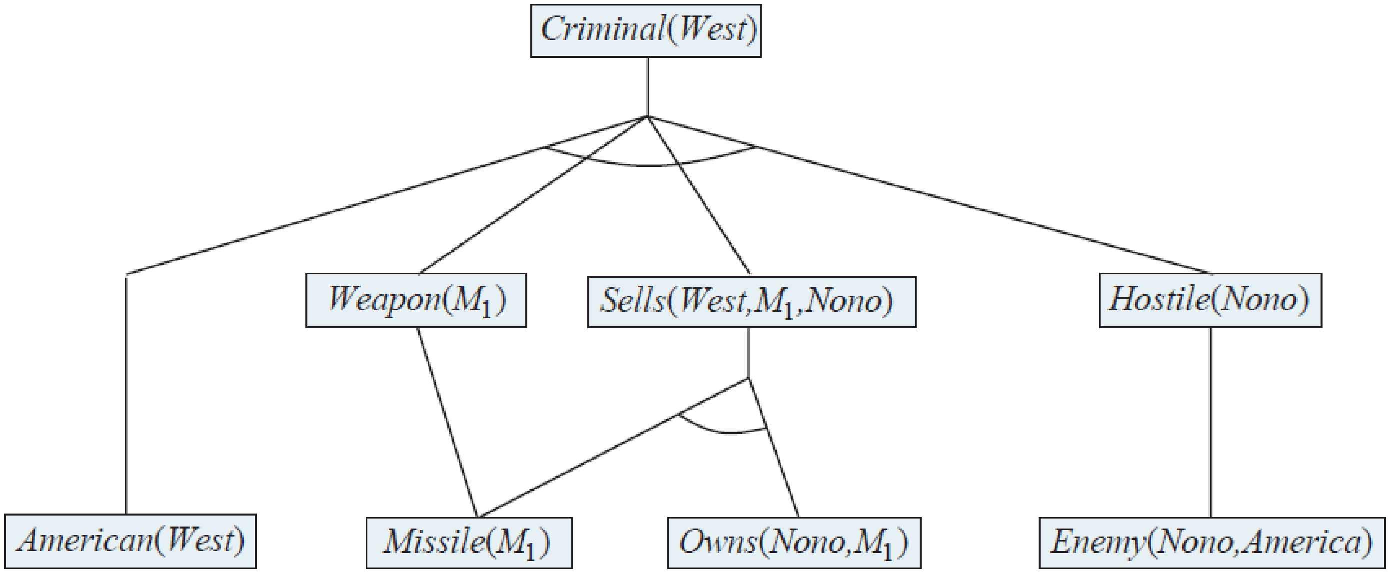


Weapon(M1)
Sells(West,M1,Nono)
Hostile(Nono)
American(West)
Missile(M1)
Owns(Nono,M1)
Enemy(Nono,America)

Proof Tree


Criminal(West)
American(West)
Weapon(M1)
Sells(West,M1,Nono)
Hostile(Nono)
Missile(M1)
Owns(Nono,M1)
Enemy(Nono,America)

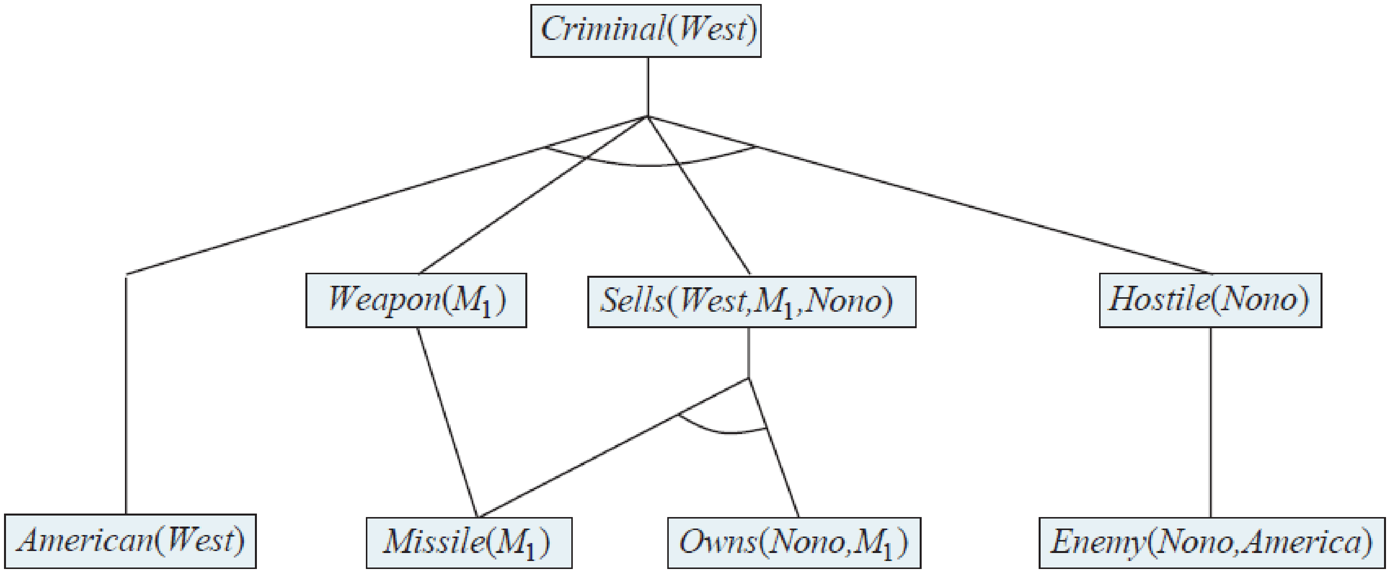
Proof Tree



- Soundness of forward chaining

  Every inference is an application of Generalized Modus Ponens.

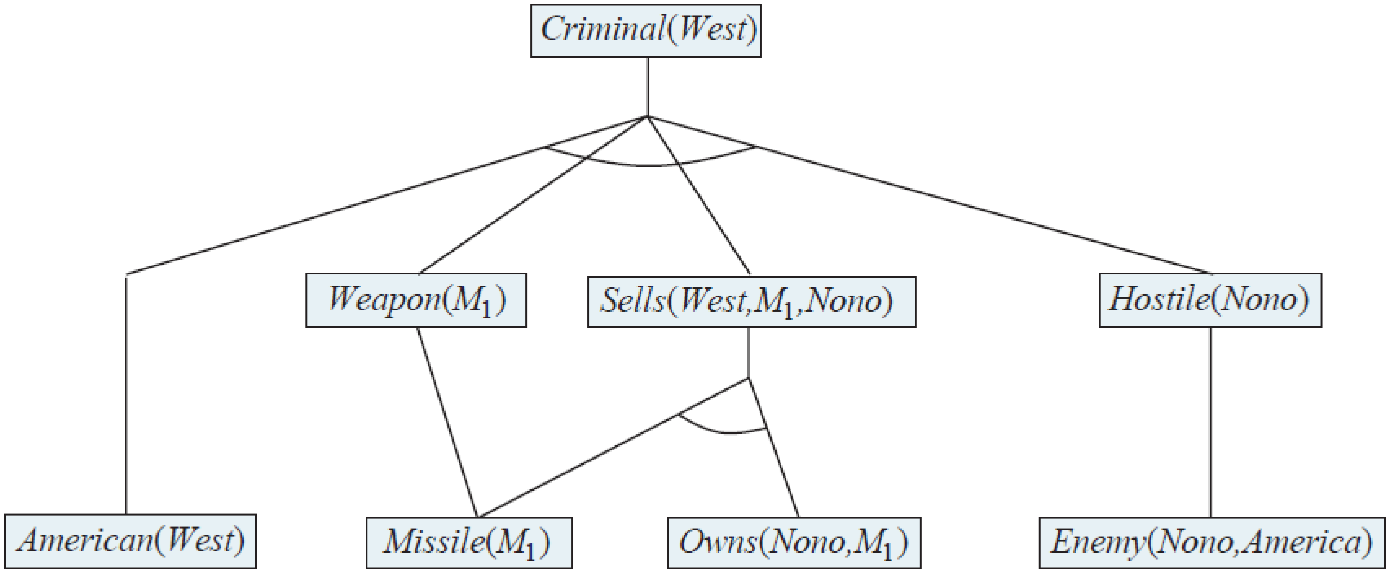
Proof Tree



- Soundness of forward chaining

  Every inference is an application of Generalized Modus Ponens.

Proof Tree

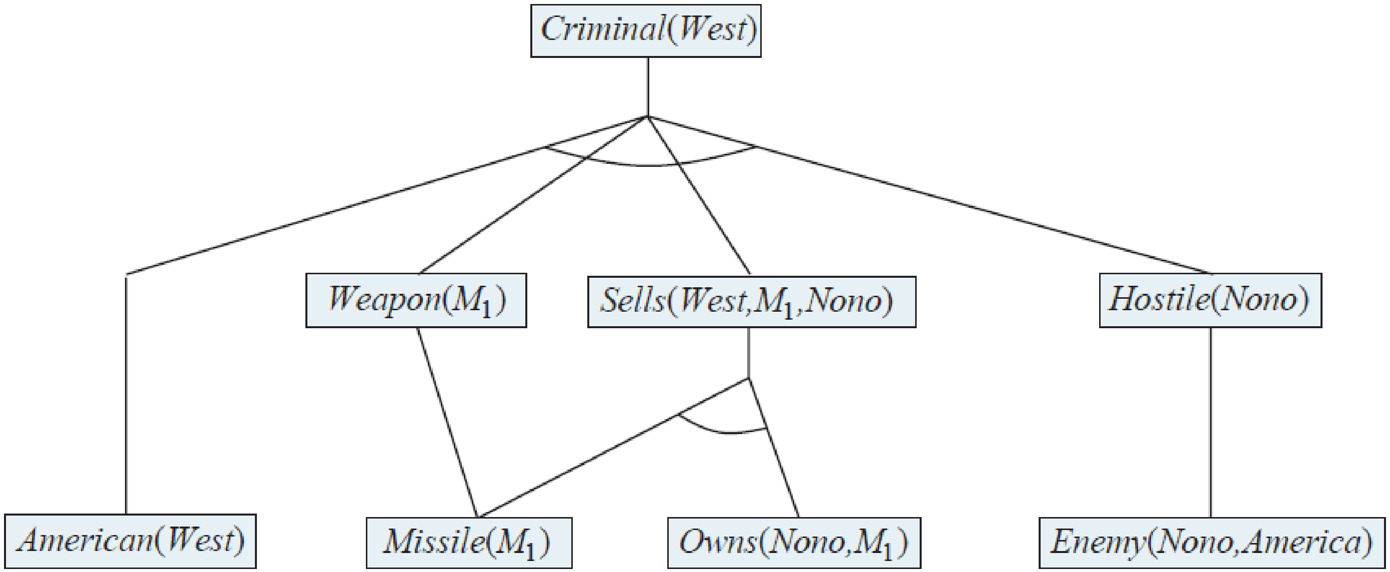


- **Soundness** of forward chaining

  Every inference is an application of Generalized Modus Ponens.

- **Completeness**
  - Easy to establish if no function symbols appears in the KB.

Proof Tree



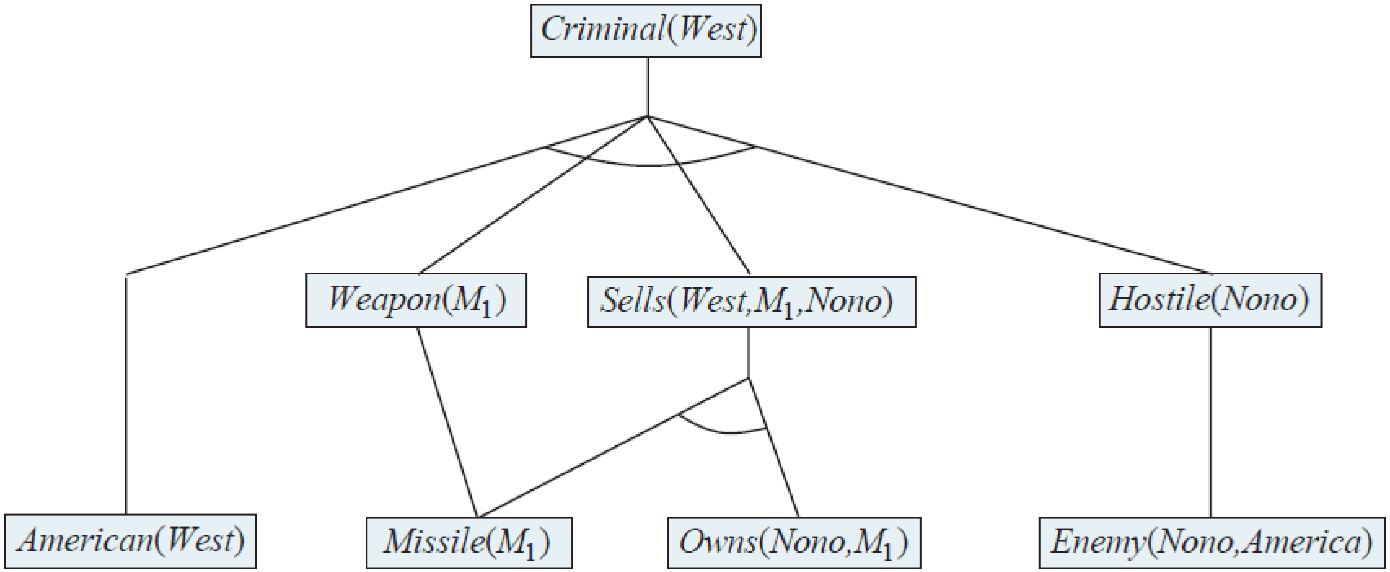
♦ Soundness of forward chaining

Every inference is an application of Generalized Modus Ponens.

♦ Completeness

- Easy to establish if no function symbols appears in the KB.
- Guaranteed except for a query with no answer, if function symbols appear in the KB.

Proof Tree



- Soundness of forward chaining

  Every inference is an application of Generalized Modus Ponens.

- Completeness
  - Easy to establish if no function symbols appears in the KB.
  - Guaranteed except for a query with no answer, if function symbols appear in the KB.
  - Entailment with definite clauses is semi-decidable (Turing 1936, Church 1936).

Improvement 1: Matching Rules Against Known Facts

Inefficiency of simple forward chaining:

- ♠ Exhaustively matches every rule against every fact.
- ♠ Rechecks every rule on each iteration (even with very few additions to *KB*).
- ♠ Generates many facts that are irrelevant to the goal.

Improvement 1: Matching Rules Against Known Facts

Inefficiency of simple forward chaining:

- ♠ Exhaustively matches every rule against every fact.
- ♠ Rechecks every rule on each iteration (even with very few additions to *KB*).
- ♠ Generates many facts that are irrelevant to the goal.

Improvement 1: Matching Rules Against Known Facts

Inefficiency of simple forward chaining:

- ♠ Exhaustively matches every rule against every fact.
- ♠ Rechecks every rule on each iteration (even with very few additions to *KB*).
- ♠ Generates many facts that are irrelevant to the goal.

Improvement 1: Matching Rules Against Known Facts

Inefficiency of simple forward chaining:

- ♠ Exhaustively matches every rule against every fact.
- ♠ Rechecks every rule on each iteration (even with very few additions to *KB*).
- ♠ Generates many facts that are irrelevant to the goal.

$$Missile(x) \wedge Owns(Nono, x) \Rightarrow Sells(West, x, Nono)$$

Improvement 1: Matching Rules Against Known Facts

Inefficiency of simple forward chaining:

- ♠ Exhaustively matches every rule against every fact.
- ♠ Rechecks every rule on each iteration (even with very few additions to *KB*).
- ♠ Generates many facts that are irrelevant to the goal.

$$Missile(x) \land Owns(Nono, x) \Rightarrow Sells(West, x, Nono)$$

- ♦ Suppose Nono owns many objects among which very few are missiles. They are two approaches:

Improvement 1: Matching Rules Against Known Facts

Inefficiency of simple forward chaining:

- ♠ Exhaustively matches every rule against every fact.
- ♠ Rechecks every rule on each iteration (even with very few additions to *KB*).
- ♠ Generates many facts that are irrelevant to the goal.

$$Missile(x) \wedge Owns(Nono, x) \Rightarrow Sells(West, x, Nono)$$

- ♦ Suppose Nono owns many objects among which very few are missiles. They are two approaches:
  - ♣ Find all the objects owned by Nono and, for each, check if it is a missile.

Improvement 1: Matching Rules Against Known Facts

Inefficiency of simple forward chaining:

♠ Exhaustively matches every rule against every fact.

♠ Rechecks every rule on each iteration (even with very few additions to *KB*).

♠ Generates many facts that are irrelevant to the goal.

$$Missile(x) \land Owns(Nono, x) \Rightarrow Sells(West, x, Nono)$$

- Suppose Nono owns many objects among which very few are missiles. They are two approaches:
  - ♣ Find all the objects owned by Nono and, for each, check if it is a missile.
  - ♣ Find all the missiles first and check if they are owned by Nono.

Improvement 1: Matching Rules Against Known Facts

Inefficiency of simple forward chaining:

$\Longrightarrow$ ♠ Exhaustively matches every rule against every fact.

♠ Rechecks every rule on each iteration (even with very few additions to *KB*).

♠ Generates many facts that are irrelevant to the goal.

$$Missile(x) \wedge Owns(Nono, x) \Rightarrow Sells(West, x, Nono)$$

♦ Suppose Nono owns many objects among which very few are missiles. They are two approaches:

♣ Find all the objects owned by Nono and, for each, check if it is a missile.

♣ Find all the missiles first and check if they are owned by Nono. More efficient!

Improvement 1: Matching Rules Against Known Facts

Inefficiency of simple forward chaining:

- ♠ Exhaustively matches every rule against every fact.
- ♠ Rechecks every rule on each iteration (even with very few additions to *KB*).
- ♠ Generates many facts that are irrelevant to the goal.

$$\mathit{Missile}(x) \wedge \mathit{Owns}(\mathit{Nono}, x) \Rightarrow \mathit{Sells}(\mathit{West}, x, \mathit{Nono})$$

- ♦ Suppose Nono owns many objects among which very few are missiles. They are two approaches:
  - ♣ Find all the objects owned by Nono and, for each, check if it is a missile.
  - ♣ Find all the missiles first and check if they are owned by Nono. More efficient!
- ♦ How to order the conjuncts of the rule premise so they can be solved with the minimum total cost?

Improvement 1: Matching Rules Against Known Facts

Inefficiency of simple forward chaining:

♠ Exhaustively matches every rule against every fact.

♠ Rechecks every rule on each iteration (even with very few additions to *KB*).

♠ Generates many facts that are irrelevant to the goal.

$$Missile(x) \wedge Owns(Nono, x) \Rightarrow Sells(West, x, Nono)$$

- Suppose Nono owns many objects among which very few are missiles. They are two approaches:
  - ♣ Find all the objects owned by Nono and, for each, check if it is a missile.
  - ♣ Find all the missiles first and check if they are owned by Nono. More efficient!
- How to order the conjuncts of the rule premise so they can be solved with the minimum total cost?

NP-hard!

Improvement 1: Matching Rules Against Known Facts

Inefficiency of simple forward chaining:

- ♠ Exhaustively matches every rule against every fact.
- ♠ Rechecks every rule on each iteration (even with very few additions to *KB*).
- ♠ Generates many facts that are irrelevant to the goal.

$$Missile(x) \wedge Owns(Nono, x) \Rightarrow Sells(West, x, Nono)$$

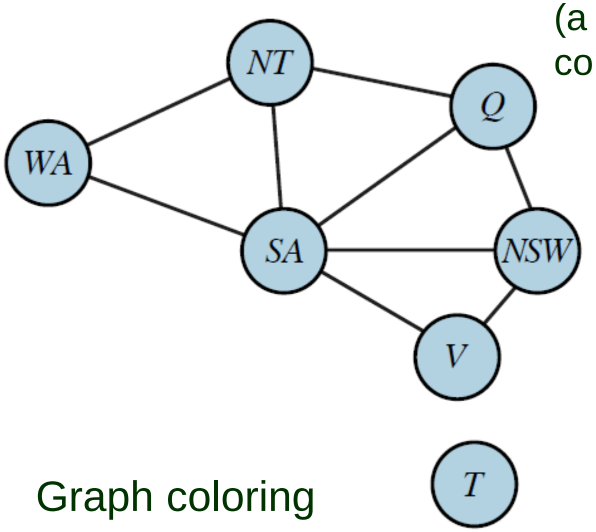
- ♦ Suppose Nono owns many objects among which very few are missiles. They are two approaches:
  - ♣ Find all the objects owned by Nono and, for each, check if it is a missile.
  - ♣ Find all the missiles first and check if they are owned by Nono. More efficient!
- ♦ How to order the conjuncts of the rule premise so they can be solved with the minimum total cost?

NP-hard!

Use a heuristic, e.g., the minimum-remaining-values (MRV) heuristic for CSPs.

CSP as a Definite Clause

View every conjunct in the premise as a constraint on the variables it contains.



Graph coloring

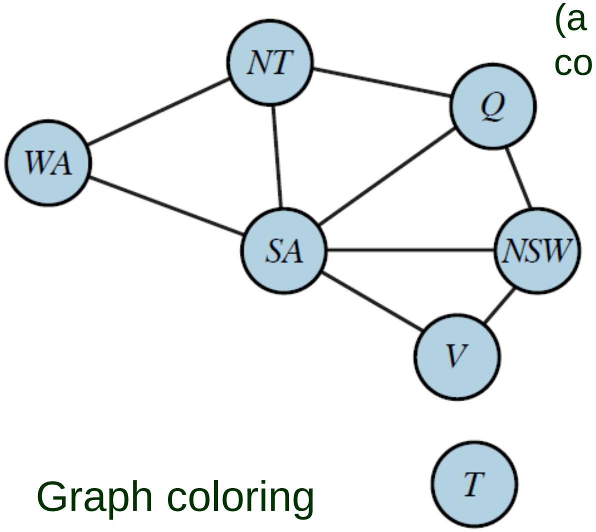
color of WA (a Skolem constant)

$Diff(wa, nt) \wedge Diff(wa, sa) \wedge$
$Diff(nt, q) \wedge Diff(nt, sa) \wedge$
$Diff(q, nsw) \wedge Diff(q, sa) \wedge$
$Diff(nsw, v) \wedge Diff(nsw, sa) \wedge$
$Diff(v, sa) \Rightarrow true$

$Diff(Red, Blue)$	$Diff(Red, Green)$
$Diff(Green, Red)$	$Diff(Green, Blue)$
$Diff(Blue, Red)$	$Diff(Blue, Green)$

CSP as a Definite Clause

View every conjunct in the premise as a constraint on the variables it contains.



Graph coloring

color of WA
(a Skolem
constant)

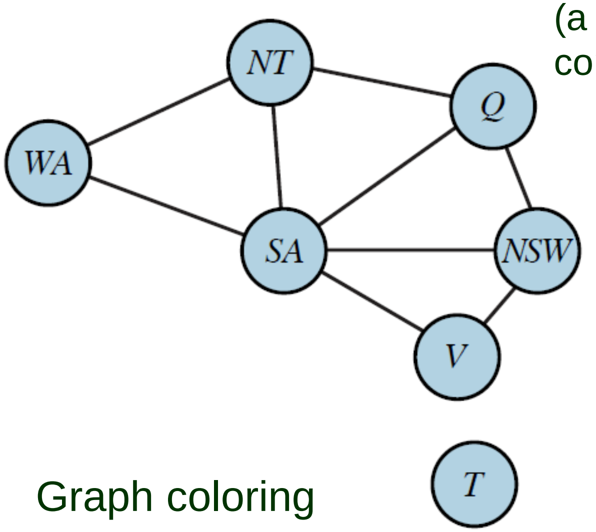
$$
\begin{aligned}
&Diff(wa, nt) \wedge Diff(wa, sa) \wedge \\
&Diff(nt, q) \wedge Diff(nt, sa) \wedge \\
&Diff(q, nsw) \wedge Diff(q, sa) \wedge \\
&Diff(nsw, v) \wedge Diff(nsw, sa) \wedge \\
&Diff(v, sa) \Rightarrow true
\end{aligned}
$$

$Diff(Red, Blue)$	$Diff(Red, Green)$
$Diff(Green, Red)$	$Diff(Green, Blue)$
$Diff(Blue, Red)$	$Diff(Blue, Green)$

Constraint satisfaction is NP-hard.

CSP as a Definite Clause

View every conjunct in the premise as a constraint on the variables it contains.



Graph coloring

color of WA (a Skolem constant)

$Diff(wa, nt) \land Diff(wa, sa) \land$
$Diff(nt, q) \land Diff(nt, sa) \land$
$Diff(q, nsw) \land Diff(q, sa) \land$
$Diff(nsw, v) \land Diff(nsw, sa) \land$
$Diff(v, sa) \Rightarrow true$

$Diff(Red, Blue)$	$Diff(Red, Green)$
$Diff(Green, Red)$	$Diff(Green, Blue)$
$Diff(Blue, Red)$	$Diff(Blue, Green)$

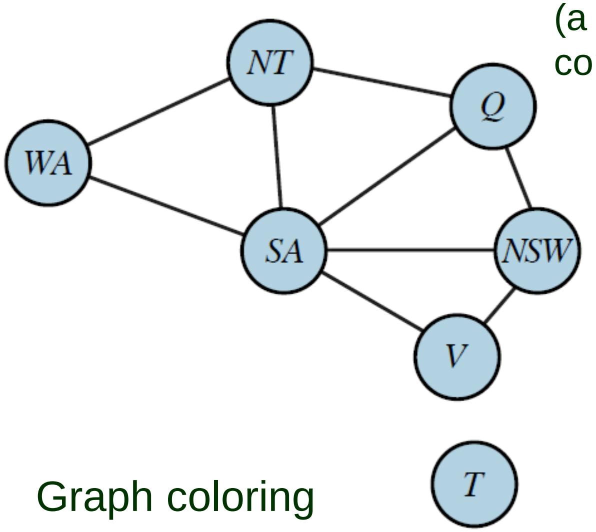
Constraint satisfaction is NP-hard.

⇩

Matching a definite clause against a set of facts is NP-hard.

CSP as a Definite Clause

View every conjunct in the premise as a constraint on the variables it contains.



Graph coloring

color of WA (a Skolem constant)

Diff(*wa*, *nt*) ∧ *Diff*(*wa*, *sa*) ∧
Diff(*nt*, *q*) ∧ *Diff*(*nt*, *sa*) ∧
Diff(*q*, *nsw*) ∧ *Diff*(*q*, *sa*) ∧
Diff(*nsw*, *v*) ∧ *Diff*(*nsw*, *sa*) ∧
Diff(*v*, *sa*) ⇒ *true*

Diff(*Red*, *Blue*)	*Diff*(*Red*, *Green*)
Diff(*Green*, *Red*)	*Diff*(*Green*, *Blue*)
Diff(*Blue*, *Red*)	*Diff*(*Blue*, *Green*)

Constraint satisfaction is NP-hard.

⇩

Matching a definite clause against a set of facts is NP-hard.

Good news View every Datalog clause as a CSP and apply heuristics for CSPs (e.g., tree structure, cutset conditioning, etc.).

Improvement 2: Incremental FC

Observations

- Every new fact inferred on iteration i must be derived from at least one new fact inferred on iteration $i - 1$.
- Only a small fraction of the rules are triggered by a fact.

Improvement 2: Incremental FC

Observations

- Every new fact inferred on iteration i must be derived from at least one new fact inferred on iteration $i-1$.
- Only a small fraction of the rules are triggered by a fact.

Incremental forward chaining does the following during iteration i:

Improvement 2: Incremental FC

Observations

- Every new fact inferred on iteration i must be derived from at least one new fact inferred on iteration $i-1$.
- Only a small fraction of the rules are triggered by a fact.

Incremental forward chaining does the following during iteration i:

1. Check a rule only if its premise includes a conjunct p_i that unifies with a fact p_i' inferred at iteration $i-1$.

Improvement 2: Incremental FC

Observations

- Every new fact inferred on iteration i must be derived from at least one new fact inferred on iteration $i-1$.
- Only a small fraction of the rules are triggered by a fact.

Incremental forward chaining does the following during iteration i:

1. Check a rule only if its premise includes a conjunct p_i that unifies with a fact p_i' inferred at iteration $i-1$.
2. Extends the substitution to match p_i with p_i'.

Improvement 2: Incremental FC

Observations

- Every new fact inferred on iteration i must be derived from at least one new fact inferred on iteration $i-1$.
- Only a small fraction of the rules are triggered by a fact.

Incremental forward chaining does the following during iteration i:

1. Check a rule only if its premise includes a conjunct p_i that unifies with a fact p_i' inferred at iteration $i-1$.
2. Extends the substitution to match p_i with p_i'.
3. Repeat for every such conjunct in the premise of the same rule.

Improvement 2: Incremental FC

Observations

- Every new fact inferred on iteration i must be derived from at least one new fact inferred on iteration $i-1$.
- Only a small fraction of the rules are triggered by a fact.

Incremental forward chaining does the following during iteration i:

1. Check a rule only if its premise includes a conjunct p_i that unifies with a fact p_i' inferred at iteration $i-1$.
2. Extends the substitution to match p_i with p_i'.
3. Repeat for every such conjunct in the premise of the same rule.
4. The remaining conjunts are matched with facts from iterations before $i-1$.

Improvement 2: Incremental FC

Observations

- Every new fact inferred on iteration i must be derived from at least one new fact inferred on iteration $i-1$.
- Only a small fraction of the rules are triggered by a fact.

Incremental forward chaining does the following during iteration i:

1. Check a rule only if its premise includes a conjunct p_i that unifies with a fact p_i' inferred at iteration $i-1$.
2. Extends the substitution to match p_i with p_i'.
3. Repeat for every such conjunct in the premise of the same rule.
4. The remaining conjunts are matched with facts from iterations before $i-1$.

E.g., the Rete algorithm

Improvement 2: Incremental FC

Observations

- Every new fact inferred on iteration i must be derived from at least one new fact inferred on iteration $i-1$.
- Only a small fraction of the rules are triggered by a fact.

Incremental forward chaining does the following during iteration i:

1. Check a rule only if its premise includes a conjunct p_i that unifies with a fact p_i' inferred at iteration $i-1$.
2. Extends the substitution to match p_i with p_i'.
3. Repeat for every such conjunct in the premise of the same rule.
4. The remaining conjunts are matched with facts from iterations before $i-1$.

E.g., the Rete algorithm

II. Backward Chaining

Works like AND/OR search:

- OR
  - ♣ The goal query can be proved by any rule in the *KB*.
  - ♣ A query containing a variable, e.g., *Person*(x) can be proved in multiple ways.
- AND: all the conjuncts in the premise of a clause must be proved.

II. Backward Chaining

Works like AND/OR search:

- OR
  - ♣ The goal query can be proved by any rule in the *KB*.
  - ♣ A query containing a variable, e.g., $Person(x)$ can be proved in multiple ways.
- AND: all the conjuncts in the premise of a clause must be proved.

How does it work?

II. Backward Chaining

Works like AND/OR search:

- OR
  - ♣ The goal query can be proved by any rule in the *KB*.
  - ♣ A query containing a variable, e.g., $Person(x)$ can be proved in multiple ways.
- AND: all the conjuncts in the premise of a clause must be proved.

How does it work?

- ♦ Fetch all clauses that unify with the goal.
- ♦ Rename the variables in every such clause to be brand-new.
- ♦ Prove every conjunct in the clause by keeping track of the expanded substitution as it goes.

Depth-First Proof Tree

$American(x) \wedge Weapon(y) \wedge Sells(x,y,z) \wedge Hostile(z) \Rightarrow Criminal(x)$

$Missile(x) \wedge Owns(Nono,x) \Rightarrow Sells(West,x,Nono)$

$Missile(x) \Rightarrow Weapon(x)$

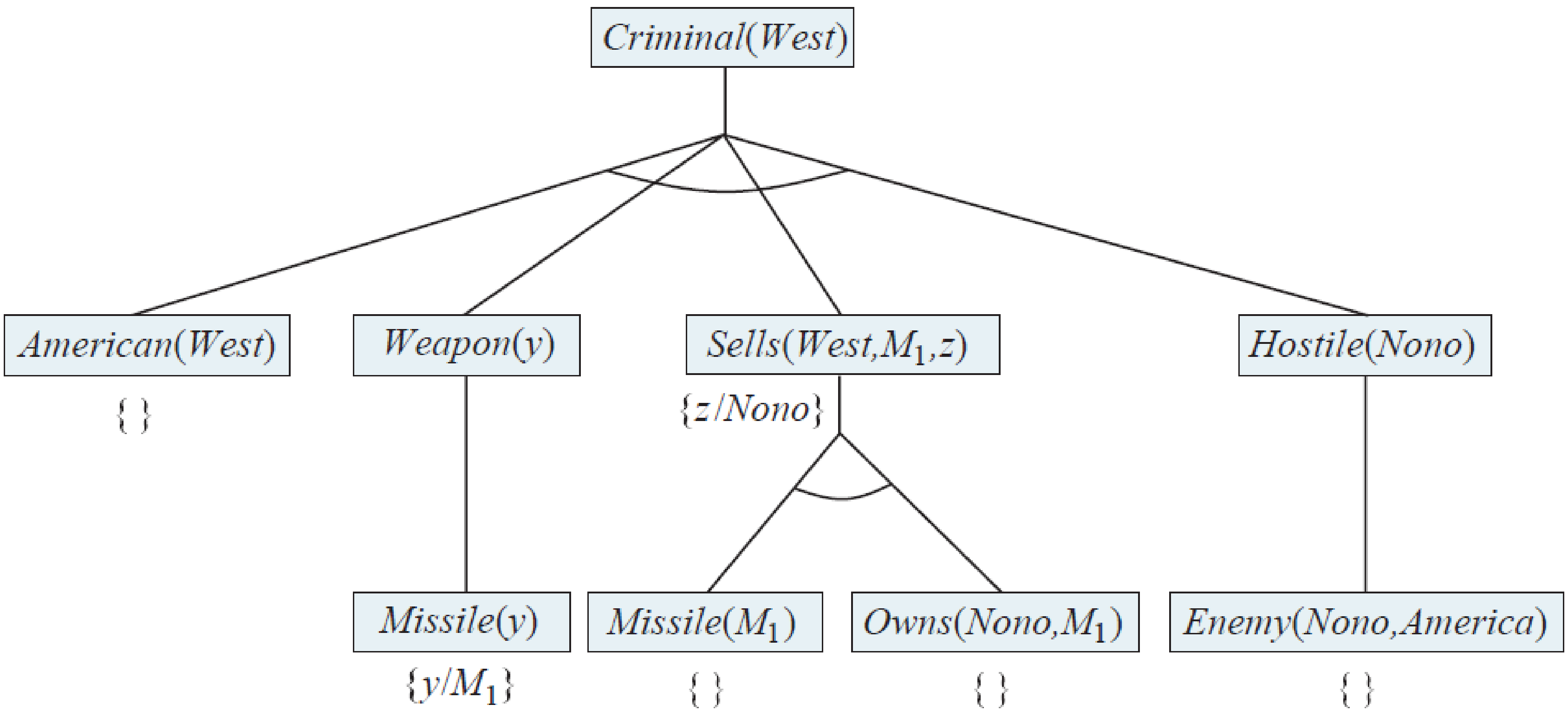
$Enemy(x,America) \Rightarrow Hostile(x)$

$Owns(Nono,M_1)$

$Missile(M_1)$

$American(West)$

$Enemy(Nono,America)$



Depth-First Proof Tree

$American(x) \land Weapon(y) \land Sells(x,y,z) \land Hostile(z) \Rightarrow Criminal(x)$

$Missile(x) \land Owns(Nono,x) \Rightarrow Sells(West,x,Nono)$

$Missile(x) \Rightarrow Weapon(x)$

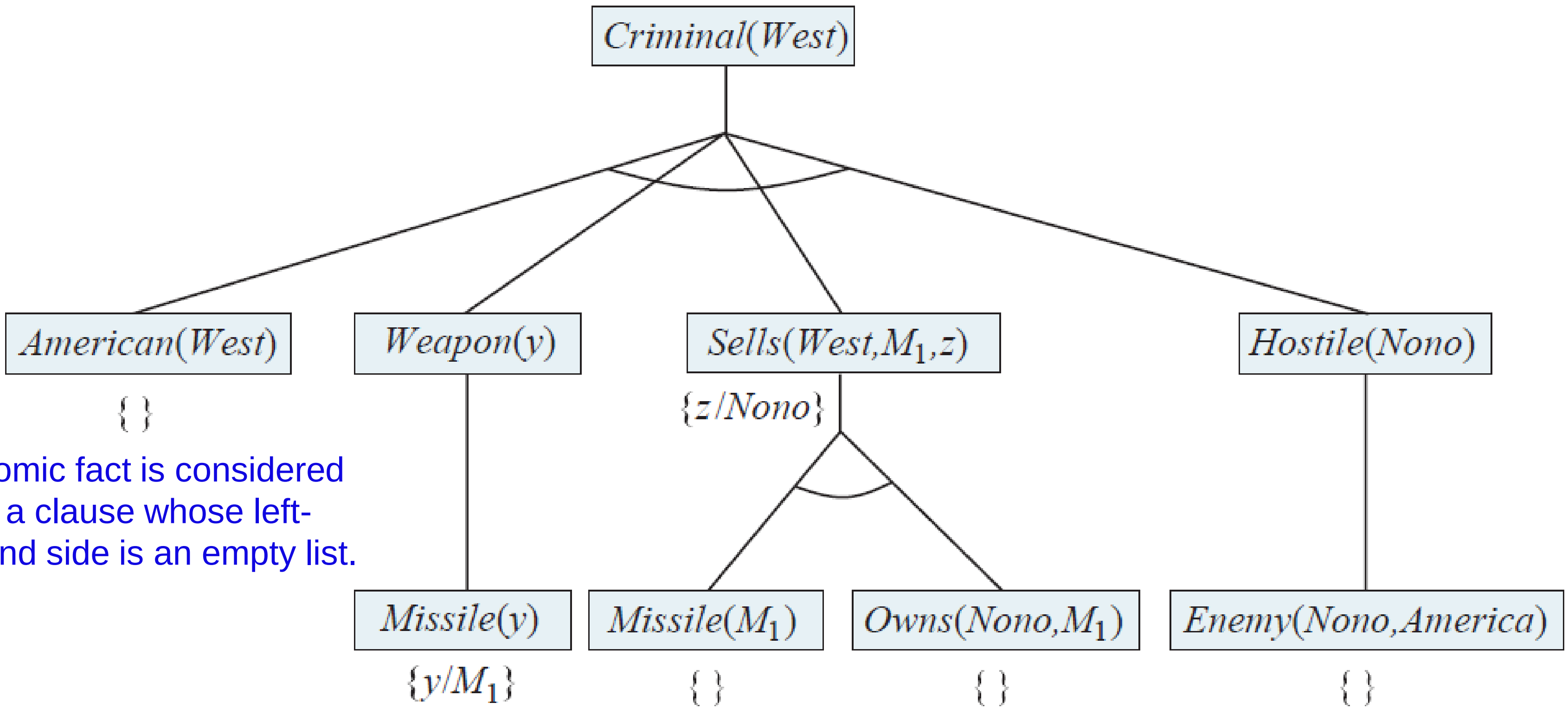
$Enemy(x,America) \Rightarrow Hostile(x)$

$Owns(Nono,M_1)$

$Missile(M_1)$

$American(West)$

$Enemy(Nono,America)$



Atomic fact is considered as a clause whose left-hand side is an empty list.

III. Logic Programming

Algorithm = Logic + Control (Robert Kowalski)

Prolog (1972) is the most widely used logic programming language.

- Rapid prototyping
- Symbolic manipulation (e.g., writing compliers, parsing natural languages)

III. Logic Programming

Algorithm = Logic + Control (Robert Kowalski)

Prolog (1972) is the most widely used logic programming language.

- Rapid prototyping
- Symbolic manipulation (e.g., writing compliers, parsing natural languages)

♦ A Prolog program is a set of definite clauses.

III. Logic Programming

Algorithm = Logic + Control (Robert Kowalski)

Prolog (1972) is the most widely used logic programming language.

- Rapid prototyping
- Symbolic manipulation (e.g., writing compliers, parsing natural languages)

♦ A Prolog program is a set of definite clauses.

// $\mathit{American}(x) \wedge \mathit{Weapon}(y) \wedge \mathit{Hostile}(z) \wedge \mathit{Sells}(x, y, z) \Rightarrow \mathit{Criminal}(x)$

III. Logic Programming

Algorithm = Logic + Control (Robert Kowalski)

Prolog (1972) is the most widely used logic programming language.

- Rapid prototyping
- Symbolic manipulation (e.g., writing compliers, parsing natural languages)

♦ A Prolog program is a set of definite clauses.

// *American*$(x) \wedge$ *Weapon*$(y) \wedge$ *Hostile*$(z) \wedge$ *Sells*$(x, y, z) \Rightarrow$ *Criminal*(x)

```
criminal(X) :- american(X), weapon(Y), sells(X,Y,Z), hostile(Z).
```

III. Logic Programming

Algorithm = Logic + Control (Robert Kowalski)

Prolog (1972) is the most widely used logic programming language.

- Rapid prototyping
- Symbolic manipulation (e.g., writing compliers, parsing natural languages)

♦ A Prolog program is a set of definite clauses.

// $American(x) \wedge Weapon(y) \wedge Hostile(z) \wedge Sells(x, y, z) \Rightarrow Criminal(x)$

```
criminal(X) :- american(X), weapon(Y), sells(X,Y,Z), hostile(Z).
```

$\Leftarrow$

III. Logic Programming

Algorithm = Logic + Control (Robert Kowalski)

Prolog (1972) is the most widely used logic programming language.

- Rapid prototyping
- Symbolic manipulation (e.g., writing compliers, parsing natural languages)

♦ A Prolog program is a set of definite clauses.

// $American(x) \wedge Weapon(y) \wedge Hostile(z) \wedge Sells(x, y, z) \Rightarrow Criminal(x)$

```
criminal(X) :- american(X), weapon(Y), sells(X,Y,Z), hostile(Z).
```

$\Leftarrow$ $\wedge$

III. Logic Programming

Algorithm = Logic + Control (Robert Kowalski)

Prolog (1972) is the most widely used logic programming language.

- Rapid prototyping
- Symbolic manipulation (e.g., writing compliers, parsing natural languages)

♦ A Prolog program is a set of definite clauses.

// $\mathit{American}(x) \land \mathit{Weapon}(y) \land \mathit{Hostile}(z) \land \mathit{Sells}(x, y, z) \Rightarrow \mathit{Criminal}(x)$

```
criminal(X) :- american(X), weapon(Y), sells(X,Y,Z), hostile(Z).
```

$\Leftarrow$ $\land$ uppercase letters for variables

III. Logic Programming

Algorithm = Logic + Control (Robert Kowalski)

Prolog (1972) is the most widely used logic programming language.

- Rapid prototyping
- Symbolic manipulation (e.g., writing compliers, parsing natural languages)

♦ A Prolog program is a set of definite clauses.

// $American(x) \wedge Weapon(y) \wedge Hostile(z) \wedge Sells(x, y, z) \Rightarrow Criminal(x)$

```
criminal(X) :- american(X), weapon(Y), sells(X,Y,Z), hostile(Z).
```

$\Leftarrow$ $\wedge$ uppercase letters for variables end of a clause

Backward Chaining in Prolog

- Prolog recursively defines a function.

Backward Chaining in Prolog

- Prolog recursively defines a function.

Example List appending.

```
append([], Y, Y).
```

```
append( [A|X], Y, [A|Z]) :- append(X,Y,Z)
```

Backward Chaining in Prolog

- Prolog recursively defines a function.

Example List appending.

```
// appending the empty list and the list Y produces the same list Y.
append([], Y, Y).

append( [A|X], Y, [A|Z]) :- append(X,Y,Z)
```

Backward Chaining in Prolog

- Prolog recursively defines a function.

Example List appending.

// appending the empty list and the list Y produces the same list Y.

```
append([], Y, Y).
```

```
append( [A|X], Y, [A|Z]) :- append(X,Y,Z)
```

a list whose first element
is X and rest is L.

Backward Chaining in Prolog

- Prolog recursively defines a function.

Example List appending.

// appending the empty list and the list Y produces the same list Y.

```
append([], Y, Y).
```

// [A | Z] is the result of appending [A | X] and Y provided that Z is
// the result of appending X and Y.

```
append( [A|X], Y, [A|Z]) :- append(X,Y,Z)
```

a list whose first element
is X and rest is L.

Backward Chaining in Prolog

- Prolog recursively defines a function.

Example List appending.

// appending the empty list and the list Y produces the same list Y.

```
append([], Y, Y).
```

// [A | Z] is the result of appending [A | X] and Y provided that Z is
// the result of appending X and Y.

```
append( [A|X], Y, [A|Z]) :- append(X,Y,Z)
```

a list whose first element
is X and rest is L.

Describes the relations among the three arguments of append.

Query Example

(1) `append([], Y, Y).`

(2) `append( [A|X], Y, [A|Z]) :- append(X,Y,Z)`

Query: `append(X, Y, [1, 2, 3])`

Query Example

(1) `append([], Y, Y).`

(2) `append( [A|X], Y, [A|Z]) :- append(X,Y,Z)`

Query: `append(X, Y, [1, 2, 3])`

Solutions returned by Prolog:

`X=[]` `Y=[1,2,3]` // matches (1)

Query Example

(1) `append([], Y, Y).`

(2) `append( [A|X], Y, [A|Z]) :- append(X,Y,Z)`

Query: `append(X, Y, [1, 2, 3])`

Solutions returned by Prolog:

```
X=[]        Y=[1,2,3]
```
// matches (1)

```
X=[1]       Y=[2,3]
```
// matches (2) to obtain substitution {A/1}, and then
// matches `append(X, Y, [2, 3])` against (1).

Query Example

(1) `append([], Y, Y).`

(2) `append( [A|X], Y, [A|Z]) :- append(X,Y,Z)`

Query: `append(X, Y, [1, 2, 3])`

Solutions returned by Prolog:

`X=[]`	`Y=[1,2,3]`	// matches (1)
`X=[1]`	`Y=[2,3]`	// matches (2) to obtain substitution {A/1}, and then // matches `append(X, Y, [2, 3])` against (1).
`X=[1,2]`	`Y=[3]`	// applies (2) twice and then (1).

Query Example

(1) `append([], Y, Y).`

(2) `append( [A|X], Y, [A|Z]) :- append(X,Y,Z)`

Query: `append(X, Y, [1, 2, 3])`

Solutions returned by Prolog:

`X=[]`	`Y=[1,2,3]`	// matches (1)
`X=[1]`	`Y=[2,3]`	// matches (2) to obtain substitution {A/1}, and then // matches `append(X, Y, [2, 3])` against (1).
`X=[1,2]`	`Y=[3]`	// applies (2) twice and then (1).
`X=[1,2,3]`	`Y=[]`	// applies (2) thrice and then (1).

Infinite Loop

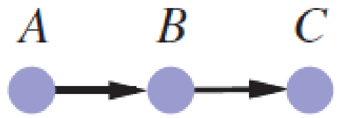
Finds if a path exists between two nodes in a directed graph.

```
path(X,Z) :- link(X, Z).
path(X,Z) :- path(X,Y), link(Y,Z).
```

Infinite Loop

Finds if a path exists between two nodes in a directed graph.

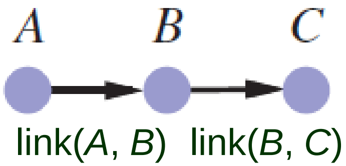
```
path(X,Z) :- link(X, Z).
path(X,Z) :- path(X,Y), link(Y,Z).
```



Infinite Loop

Finds if a path exists between two nodes in a directed graph.

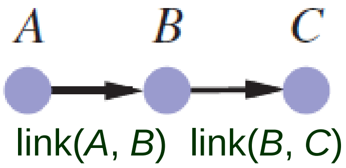
```
path(X,Z) :- link(X, Z).
path(X,Z) :- path(X,Y), link(Y,Z).
```



Infinite Loop

Finds if a path exists between two nodes in a directed graph.

```
path(X,Z) :- link(X, Z).
path(X,Z) :- path(X,Y), link(Y,Z).
```

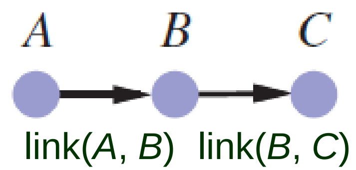


Query `path(a, c)`

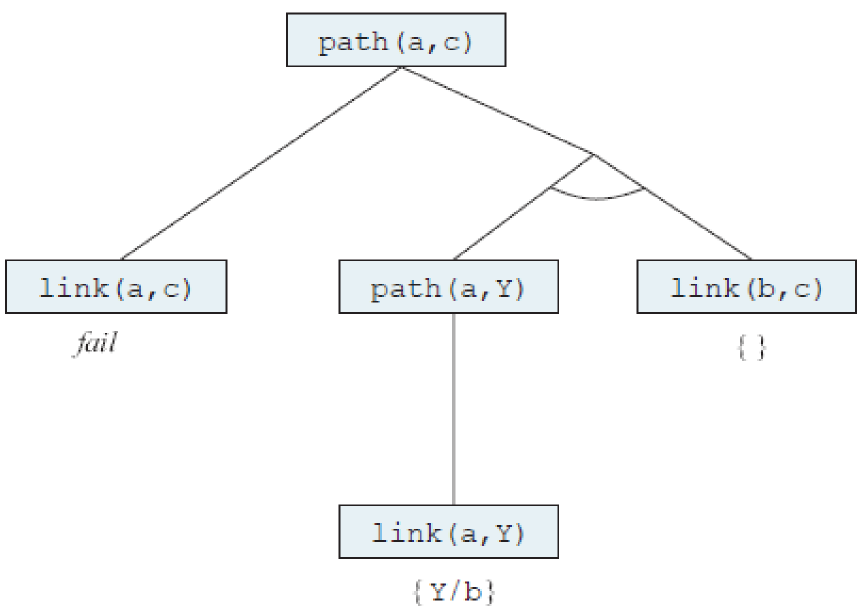
Infinite Loop

Finds if a path exists between two nodes in a directed graph.

```
path(X,Z) :- link(X, Z).
path(X,Z) :- path(X,Y), link(Y,Z).
```



Query `path(a, c)`



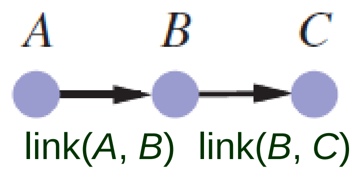
Infinite Loop

Finds if a path exists between two nodes in a directed graph.

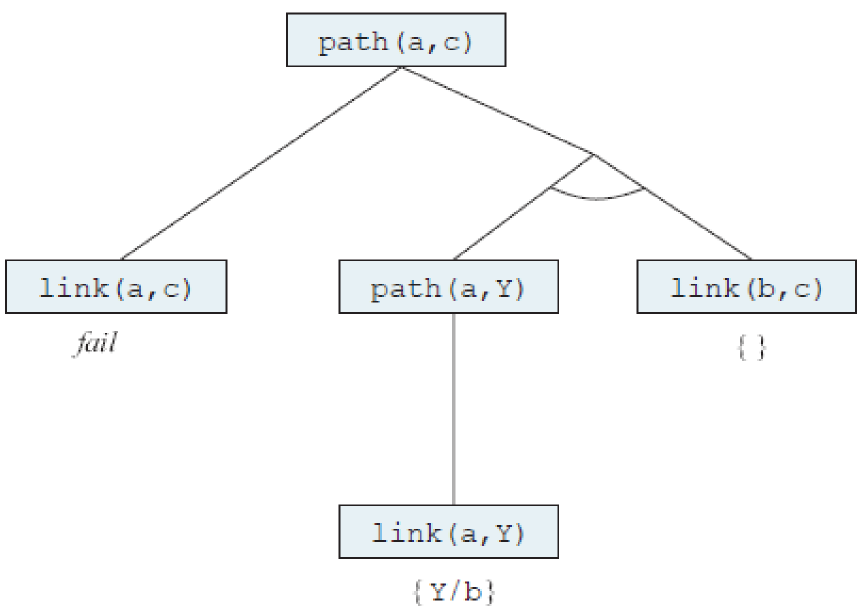
```
path(X,Z) :- link(X, Z).
path(X,Z) :- path(X,Y), link(Y,Z).
```

switch order

```
path(X,Z) :- path(X,Y), link(Y,Z).
path(X,Z) :- link(X, Z).
```



Query `path(a, c)`



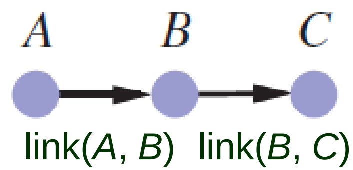
Infinite Loop

Finds if a path exists between two nodes in a directed graph.

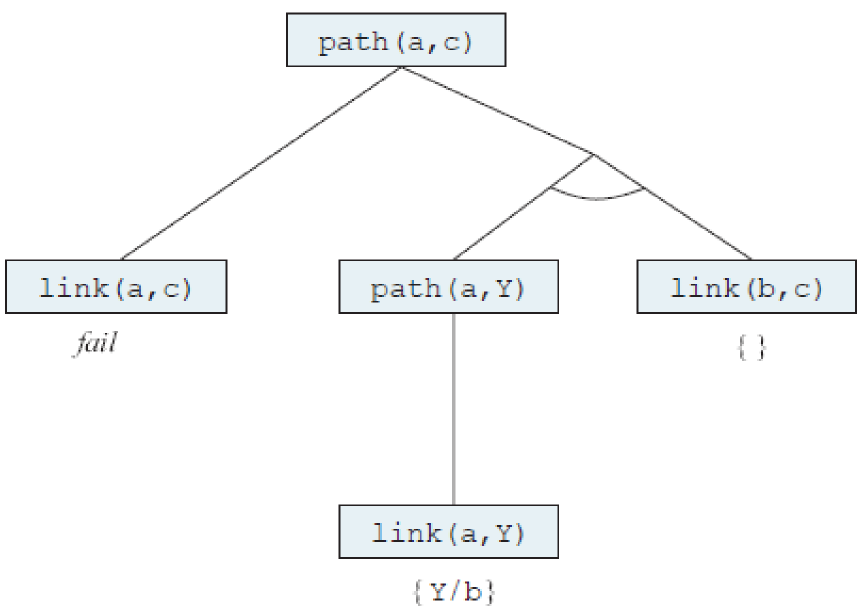
```
path(X,Z) :- link(X, Z).
path(X,Z) :- path(X,Y), link(Y,Z).
```

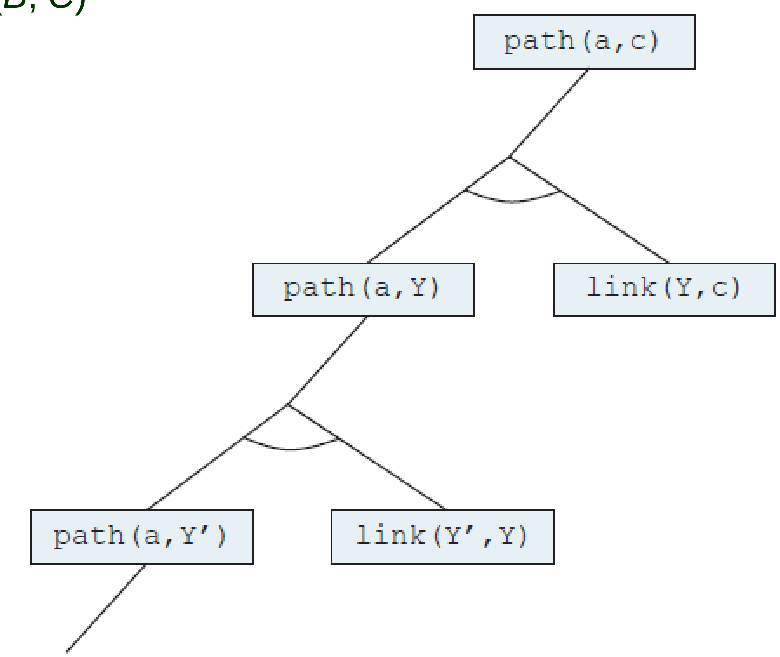
switch order ⇒

```
path(X,Z) :- path(X,Y), link(Y,Z).
path(X,Z) :- link(X, Z).
```



Query `path(a, c)`





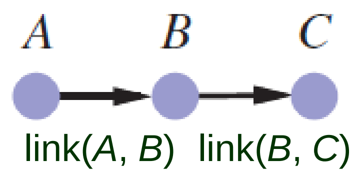
Infinite Loop

Finds if a path exists between two nodes in a directed graph.

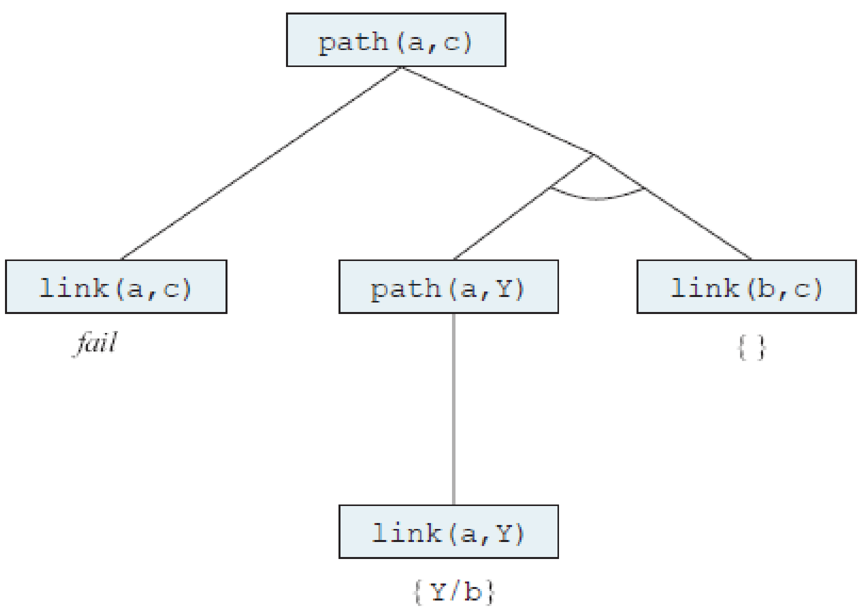
```
path(X,Z) :- link(X, Z).
path(X,Z) :- path(X,Y), link(Y,Z).
```

switch order ⟹

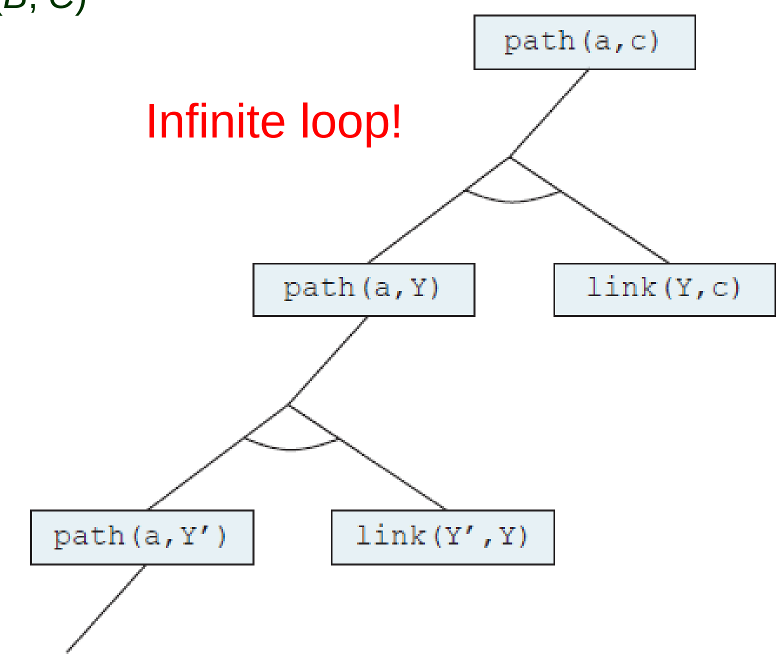
```
path(X,Z) :- path(X,Y), link(Y,Z).
path(X,Z) :- link(X, Z).
```



Query `path(a, c)`

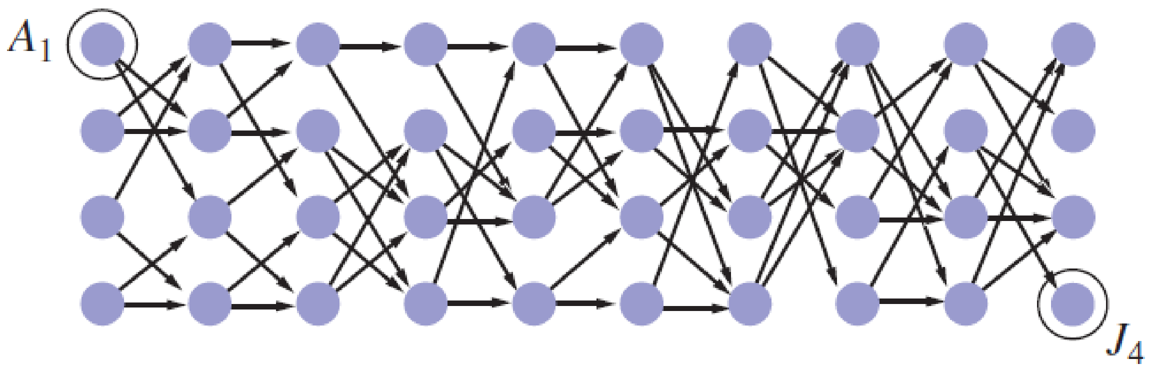


Infinite loop!



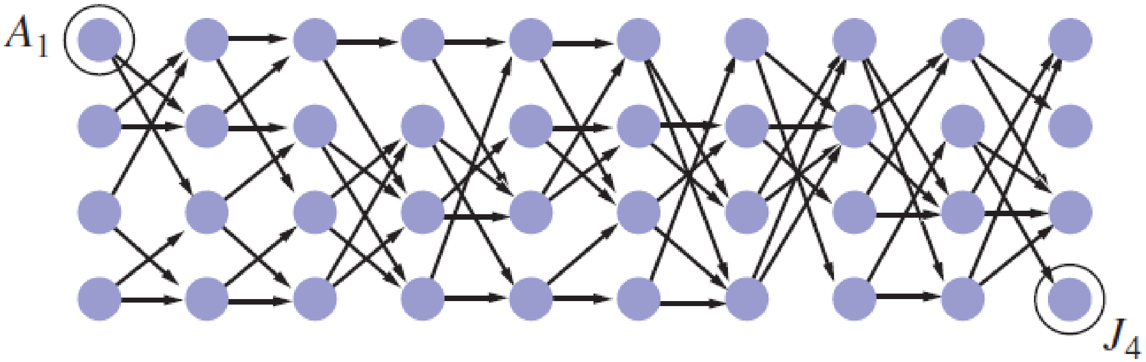
Redundant Inference

```
path(X,Z) :- link(X, Z).
path(X,Z) :- path(X,Y), link(Y,Z).
```



Redundant Inference

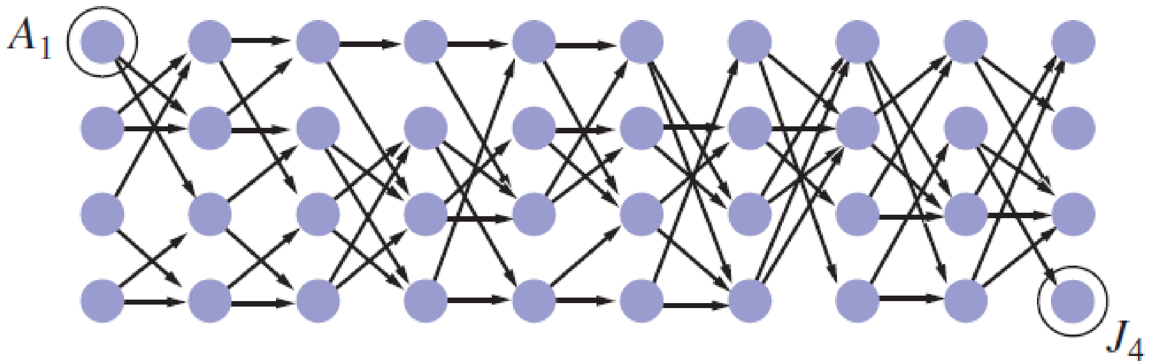
```
path(X,Z) :- link(X, Z).
path(X,Z) :- path(X,Y), link(Y,Z).
```



Query `path(A1, J4)`

Redundant Inference

```
path(X,Z) :- link(X, Z).
path(X,Z) :- path(X,Y), link(Y,Z).
```

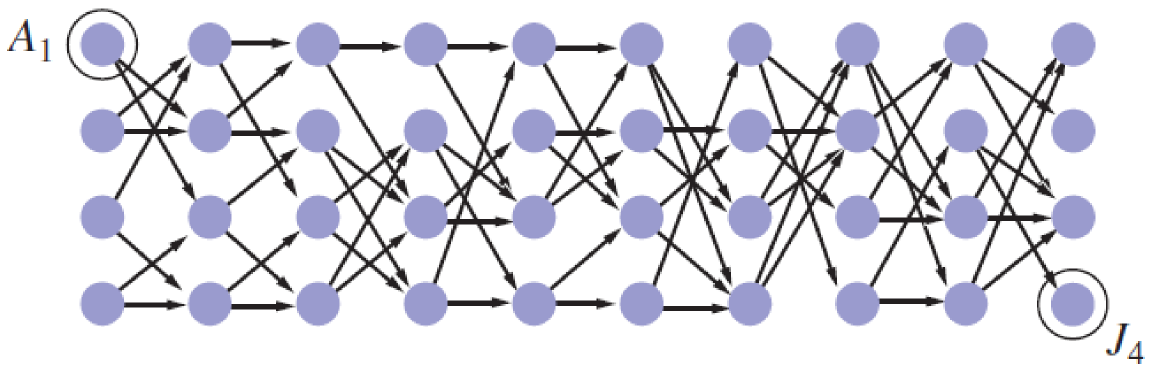


Query `path(A1, J4)`

- ♠ Prolog performs 877 inferences (most of which involve nodes from which the goal is unreachable).

Redundant Inference

```
path(X,Z) :- link(X, Z).
path(X,Z) :- path(X,Y), link(Y,Z).
```

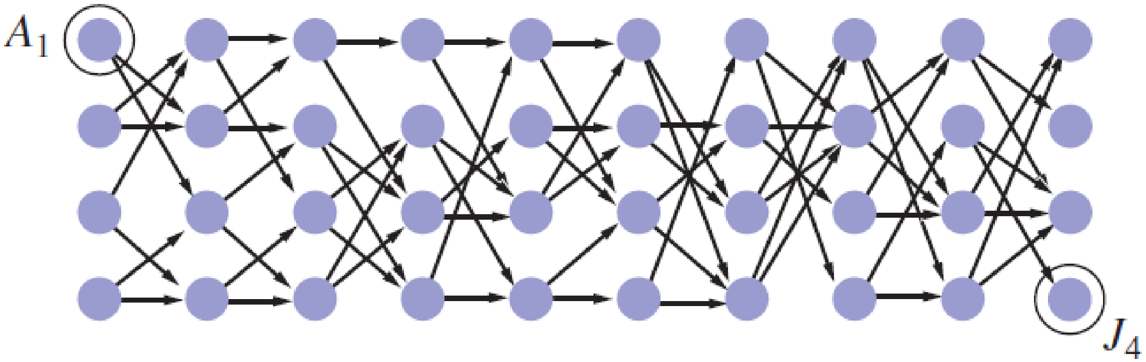


Query `path(A1, J4)`

- ♠ Prolog performs 877 inferences (most of which involve nodes from which the goal is unreachable).
- ♦ Forward chaining performs only 62 inferences.

Redundant Inference

```
path(X,Z) :- link(X, Z).
path(X,Z) :- path(X,Y), link(Y,Z).
```



Query `path(A1, J4)`

- ♠ Prolog performs 877 inferences (most of which involve nodes from which the goal is unreachable).
- ♦ Forward chaining performs only 62 inferences.