Logic Programming Languages

Chapter 16

featuring

Prolog, your new favorite language

Prolog
PROgramming in LOGic

- It is the most widely used logic programming language
- Its development started in 1970
- What's it good for?
  - Knowledge representation
  - Natural language processing
  - State-space searching (Rubik's cube)
  - Expert systems, deductive databases, Agents

Overview of Logic Programming

- Main idea: Ask the computer to solve problems using principles of logic:
  - Program states the known facts
  - To ask a question, you make a statement and ask the computer to search for a proof that the statement is true
  - Additional mechanisms are provided to guide the search to find a proof

Declarative vs. Imperative

- Languages used for logic programming are called *declarative* languages because programs written in them consist of declarations rather than assignment and flow-of-control statements. These declarations are statements, or *propositions*, in symbolic logic.
- Programming in imperative languages (e.g., Pascal, C) and functional languages (e.g., Lisp) is *procedural*, which means that the programmer knows what is to be accomplished by the program and instructs the computer on exactly how the computation is to be done.

Logic Programming

- Programming in logic programming languages is non-procedural.
- Programs in such languages do not state how a result is to be computed. Instead, we supply the computer with:
  - relevant information (facts and rules)
  - a method of inference for computing desired results.
- Logic programming is based on the *predicate calculus.*

Logic background

- Horn clauses
- General form:

IF (A1 and A2 and A3 …) THEN H

  - Head = H
  - Body = A1 and A2 and ….
- E.g. “If X is positive, and Y is negative, then Y is less than X”

The Predicate Calculus: Proposition

- Proposition:
  - A proposition is a logical statement, made up of objects and their relationships to each other, which may or may not be true.
  - Examples:
    - man(john)
    - likes(pizza, baseball)

parameters

functor

The Predicate Calculus: Logical Connectors

- A compound proposition consists of 2 or more propositions connected by logical connectors, which include
  - Negation: $\neg a$ ("not a")
  - Conjunction: $a \cap b$ ("a and b")
  - Disjunction: $a \cup b$ ("a or b")
  - Equivalence: $a \equiv b$ ("a is equivalent to b")
  - Implication: $a \supset b$ ("a implies b")
    $a \subset b$ ("b implies a")

The Predicate Calculus: Quantifiers

- Variables may appear in formulas, but only when introduced by **quantifiers**:
  - Universal quantifier: $\forall$ (for all)
  - Existential quantifier: $\exists$ (there exists)
- Examples:
  - $\forall X\ (woman(X) \supset human(X))$
    "All women are human"
  - $\exists X\ (likes(bill,X) \cap sport(X))$
    "There is some sport that bill likes"

Resolution and Unification

- Resolution: how we do logical deduction from multiple horn clauses:
  - If the head of horn clause #1 matches one of the terms in horn clause #2, then we can replace that term with the body of clause #1
- Unification: How to determine when the hypotheses are satisfied

The Predicate Calculus: Resolution

- Resolution is an inference rule that allows inferred propositions to be computed from given propositions.
- Suppose we have two propositions

  A ⊂ B *(B implies A)* **and** **C ⊂ D** *(D implies C)*

  and that A is identical to D. Suppose we rename A and D as T:

  T ⊂ B **and** **C ⊂ T**

  From this we can infer: **C ⊂ B**

The Predicate Calculus: Unification and Instantiation

- **Unification** is the process of finding values for variables during resolution so that the matching process can succeed
- **Instantiation** is the temporary binding of a value (and type) to a variable to allow unification. A variable is instantiated only during the resolution process. The instantiation lasts only as long as it takes to satisfy one goal.

Prolog syntax and terminology

- *clause* = Horn clauses assumed true, represented “head :- [term [,term]*]”
  - **comma represents logical “and”**
- clause is *fact* if: no terms on right of :-

head and term are both structures**:**

- *structure* = *functor* (arg1, arg2,..)
  - Represents a **logical assertion**, e.g

teaches(Barbara, class)

Prolog syntax and terminology

- Constants are numbers or represented by strings starting with lower-case
- Variables start with upper-case letters
- A *goal* or *query* is a clause with no left-hand side: ?- rainy(seattle)
  - Tells the Prolog interpreter to see if it can prove the clause.

Facts, Rules And Queries

- A collection of facts and rules is called a Knowledge Base.
- Prolog programs *are* Knowledgebases
- You use a Prolog program by posing queries.

Using KnowledgeBase

- Woman(janet)
- Woman(stacy)
- playsFlute(stacy)

- We can ask Prolog
- ?- woman(stacy).
- Prolog Answers yes

Using KnowledgeBase

- ?- playsFlute(stacy)
- Prolog Answers yes
- ?- playsFlute(mary)
- Prolog Answers Are you kidding me?

Elements of Prolog

- **Fact** statements—propositions that are assumed to be true, such as

  female(janet).

  male(steve).

  brother(steve, janet).

- Remember, these propositions have no intrinsic semantics--they mean what the programmer intends for them to mean.

Elements of Prolog

- Rules combine facts to increase knowledge of the system

```
son(X,Y):-
        male(X),child(X,Y).
```

- X is a son of Y if X is male and X is a child of Y

Elements of Prolog

- **Rule** statements take the form:

  <consequent> :- <antecedent>

- The consequent must be a single term, while the antecedent may be a single term or a conjunction.
- Examples:

```
parent (X, Y) :- mother (X, Y).
parent (X, Y) :- father (X, Y).
grandparent (X, Z) :-
    parent (X, Y),
    parent (Y, Z).
```

Elements of Prolog

- **Goal**—a proposition that we want the system to either prove or disprove.
- When variables are included, the system identifies the instantiations of the variables which make the proposition true.
- As Prolog attempts to solve goals, it examines the facts and rules in the database in top-to-bottom order.

Elements of Prolog

- Ask the Prolog virtual machine questions
- Composed at the ?- prompt
- Returns values of bound variables and yes or no

```
?- son(bob, harry).
yes
?- king(bob, france).
no
```

Elements of Prolog

- Can bind answers to questions to variables
- Who is bob the son of? (X=harry)

```
?- son(bob, X).
```

- Who is male? (X=bob, harry)

```
?- male(X).
```

- Is bob the son of someone? (yes)

```
?- son(bob, _).
```

- No variables bound in this case!

Backtracking

- How are questions resolved?

?- son(X,harry).

- Recall the rule:

```
son(X,Y):-
      male(X),child(X,Y).
```

Forward chaining (bottom-up)

(starts with each rule and checks the facts)

- Forward chaining
  - Use database to systematically generate new theorems until one matching query is found
  - Example:
    - father(bob).
    - man(X) :- father(X)
    - Given the goal: man(bob)
  - Under forward chaining, father(bob) is matched against father(X) to derive the new fact man(bob) in the database.
  - This new fact satisfies the goal.

Backward chaining (Top-Down)

(start with the facts, use the rules that apply)

- Use goal to work backward to a fact
- Example:
  - Given man(bob) as goal
  - Match against man(X) to create new goal: father(bob).
  - father(bob) goal matches pre-existing fact, thus the query is satisfied.

Forward vs. Backward chaining

- Bottom-Up Resolution (Forward Chaining)
  Searches the database of facts and rules, and attempts to find a sequence of matches within the database that satisfies the goal. Works more efficiently on a database that does not hold a lot of facts and rules.
- Top-Down Resolution (Backward Chaining)

  It starts with the goal, and then searches the database for matching sequence of rules and facts that satisfy the goal.

2nd Part of Resolution Process

- If a goal has more than one sub-goal, then the problem exists as to how to process each of the sub-goals to get the goal.
- Depth-First Search – it first finds a sequence or a match for the first sub-goal, and then continues down to the other sub-goals.
- Breath-First Search – process all the sub-goals in parallel.
- Prolog designers went with a top-down (backward chaining), depth-first resolution process.

Backward Chaining Example

uncle(X,thomas):- male(X), sibling(X,Y), has_parent(thomas,Y).

To find an *X* to make *uncle(X,thomas)* true:

1. first find an X to make *male(X)* true
2. then find a Y to make *sibling(X,Y)* true
3. then **check** that *has_parent(thomas,Y)* is true

Recursive search until rules that are facts are reached.

This is called **backward chaining**

A search example

- Consider the database:

```
mother (betty, janet).
mother (betty, steve).
mother (janet, adam).
father (steve, dylan).
parent (X, Y) :- mother (X, Y).
parent (X, Y) :- father (X, Y).
grandparent (X, Z) :-
parent (X, Y),
parent (Y, Z).
```

- and the goal

```
? grandparent (X, adam).
```

? grandparent (X, adam).

- Prolog proceeds by attempting to match the goal clause with a fact in the database.
- Failing this, it attempts to find a rule with a left-hand-side (consequent) that can be unified with the goal clause.
- It matches the goal with grandparent (X, Z) where Z is instantiated with the value adam to give the goal

  grandparent (X, adam).
- To prove this goal, Prolog must satisfy the sub-goals

  parent (X,Y) and
  parent (Y,adam)

```
mother (betty, janet).
mother (betty, steve)
mother (janet, adam).
father (steve, dylan).
parent (X, Y) :-
    mother (X, Y).
parent (X, Y) :-
    father (X, Y).
grandparent (X, Z) :-
    parent (X, Y),
    parent (Y, Z).
```

Sub-goal: parent (X,Y)

- Prolog uses a depth-first search strategy, and attempts to satisfy the first sub-goal. The first "parent" rule it encounters is

  parent (X,Y) :-
  mother (X,Y).

- To satisfy the sub-goal mother(X,Y), Prolog again starts at the top of the database and first encounters the fact

  mother (betty, janet).

- This fact matches the sub-goal with the instantiation

  X = betty, Y = janet

```
mother (betty, janet).
mother (betty, steve)
mother (janet, adam).
father (steve, dylan).
parent (X, Y) :-
    mother (X, Y).
parent (X, Y) :-
    father (X, Y).
grandparent (X, Z) :-
    parent (X, Y),
    parent (Y, Z).
```

Sub-goal: parent (X,Y)

- The instantiation
  X = betty, Y = janet
  is returned so that the 2 sub-goals of the grandparent rule are now:
  grandparent (betty, adam) :-
  parent (betty, janet),
  parent (janet, adam).

- The sub-goal parent(betty, janet) was inferred by Prolog.
- Next, Prolog must solve the sub-goal parent(janet, adam).

  Once again, Prolog uses the first matching rule it encounters:
  parent (X,Y) :-
  mother (X,Y).
  with the instantiation
  X = janet, Y = adam

```
mother (betty, janet).
mother (betty, steve)
mother (janet, adam).
father (steve, dylan).
parent (X, Y) :-
    mother (X, Y).
parent (X, Y) :-
    father (X, Y).
grandparent (X, Z) :-
    parent (X, Y),
    parent (Y, Z).
```

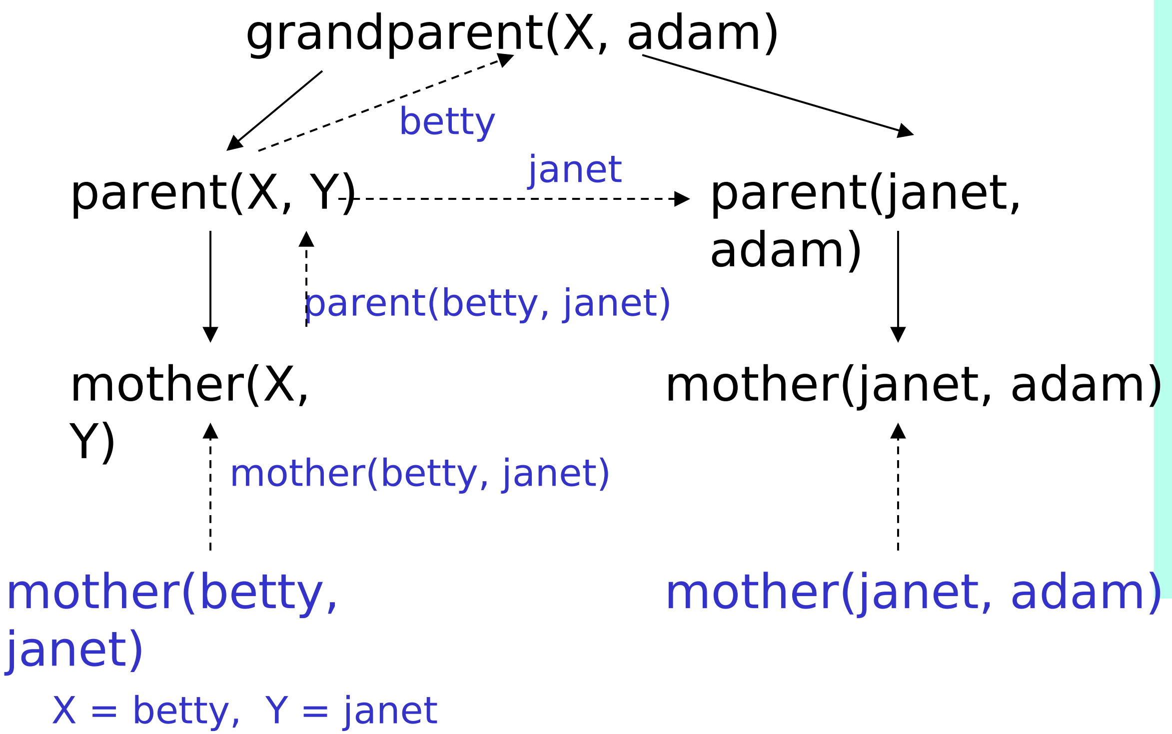
X = betty Y = janet Z = adam

Sub-goal: parent (janet,adam)

- Substituting the values X = janet,
  Y = adam in the rule parent (X, Y) :-
  mother (X, Y).
  results in
  parent (janet, adam) :-
  mother (janet, adam).
- The consequent of this rule matches the 3rd fact in the database.
- Since both sub-goals are now satisfied, the original goal grandparent(X, adam) is now proven with the instantiation
  X=betty
- Prolog returns with success:
  Yes
  X = betty

```
mother (betty, janet).
mother (betty, steve)
mother (janet, adam).
father (steve, dylan).
parent (X, Y) :-
    mother (X, Y).
parent (X, Y) :-
    father (X, Y).
grandparent (X, Z) :-
    parent (X, Y),
    parent (Y, Z).
```

Solving grandparent(X,adam)




```
mother (betty, janet).
mother (betty, steve)
mother (janet, adam).
father (steve, dylan).
parent (X, Y) :-
    mother (X, Y).
parent (X, Y) :-
    father (X, Y).
grandparent (X, Z) :-
    parent (X, Y),
    parent (Y, Z).
```

Prolog lists

- A Prolog list consists of 0 or more elements, separated by commas, enclosed in square brackets:
  - Example: [1,2,3,a,b]
  - Empty list: []
- Prolog list notation:
  - [H | T]
    - H matches the first element in a list (the Head)
    - T matches the rest of the list (the Tail)
  - For the example above,
    - H = 1
    - T = [2,3,a,b]

Prolog lists

- To further illustrate the Prolog list notation, consider the following goals:

```
?- [H | T] = [1, 2, 3, 4].
H = 1
T = [2, 3, 4]
Yes

?- [H1, H2 | T] = [1, 2, 3, 4].
H1 = 1
H2 = 2
T= [3, 4]
Yes
```

The append predicate (1)

```
?- append([1, 2, 3], [a, b], X).
X = [1, 2, 3, a, b]
Yes

?- append([1, 2, 3], X, [1, 2, 3, a, b]).
X = [a, b]
Yes

?- append(X,[a,b],[1,2,3,a,b]).
X = [1, 2, 3]
Yes
```

The append predicate (2)

```
?- append(X, Y, [1,2,3]).

X = []
Y = [1, 2, 3] ;

X = [1]
Y = [2, 3] ;

X = [1, 2]
Y = [3] ;

X = [1, 2, 3]
Y = []

Yes
```

Semicolon instructs Prolog to find another solution.

Defining myappend

```
% myappend.pl

myappend([], List, List).
myappend([H | T], List, [H | List2]) :-
    myappend(T, List, List2).
```

Execution of myappend

```
?- consult('myappend.pl').
% myappend.pl compiled 0.00 sec, 588 bytes
Yes
?- myappend([1,2,3],[a,b],X).
X = [1, 2, 3, a, b]
Yes
?- myappend([1,2,3],X,[1,2,3,a,b]).
X = [a, b]
Yes
?- myappend(X,[a,b],[1,2,3,a,b]).
X = [1, 2, 3]
Yes
```

Defining myreverse

```
% myreverse.pl

:- consult('myappend.pl').

myreverse([], []).
myreverse([H | T], X) :-
    myreverse(T, R),
    myappend(R, [H], X).
```

Execution of myreverse

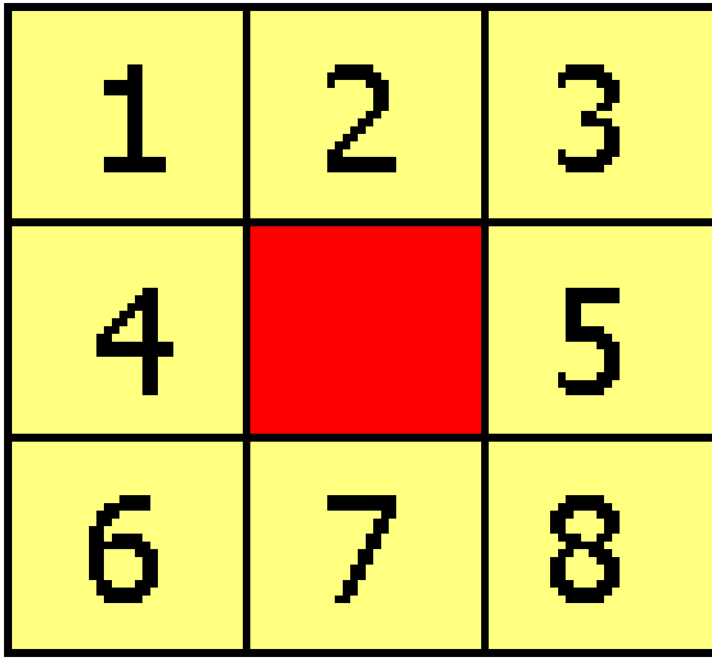
```
?- consult('myreverse.pl').
%  myappend.pl compiled 0.00 sec, 524 bytes
%  myreverse.pl compiled 0.00 sec, 524 bytes
Yes

?- myreverse([1, 2, 3], X).
X = [3, 2, 1]
Yes

?- myreverse(X, [1, 2, 3, 4]).
X = [4, 3, 2, 1]
Yes
```

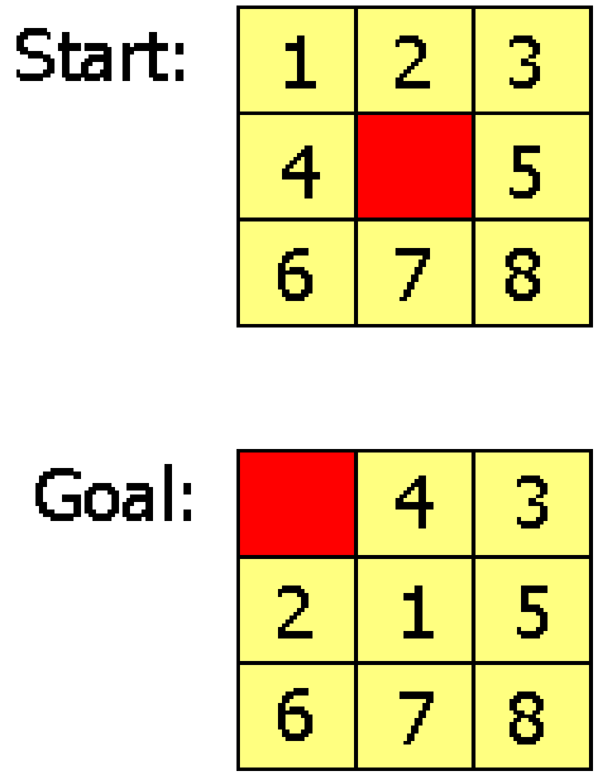
The Eights Puzzle

- The Eights Puzzle is a classic search problem which is easily solved in Prolog.
- The puzzle is a 3 x 3 grid with 8 tiles numbered 1 – 8 and an empty slot.
- A possible configuration is shown at right.

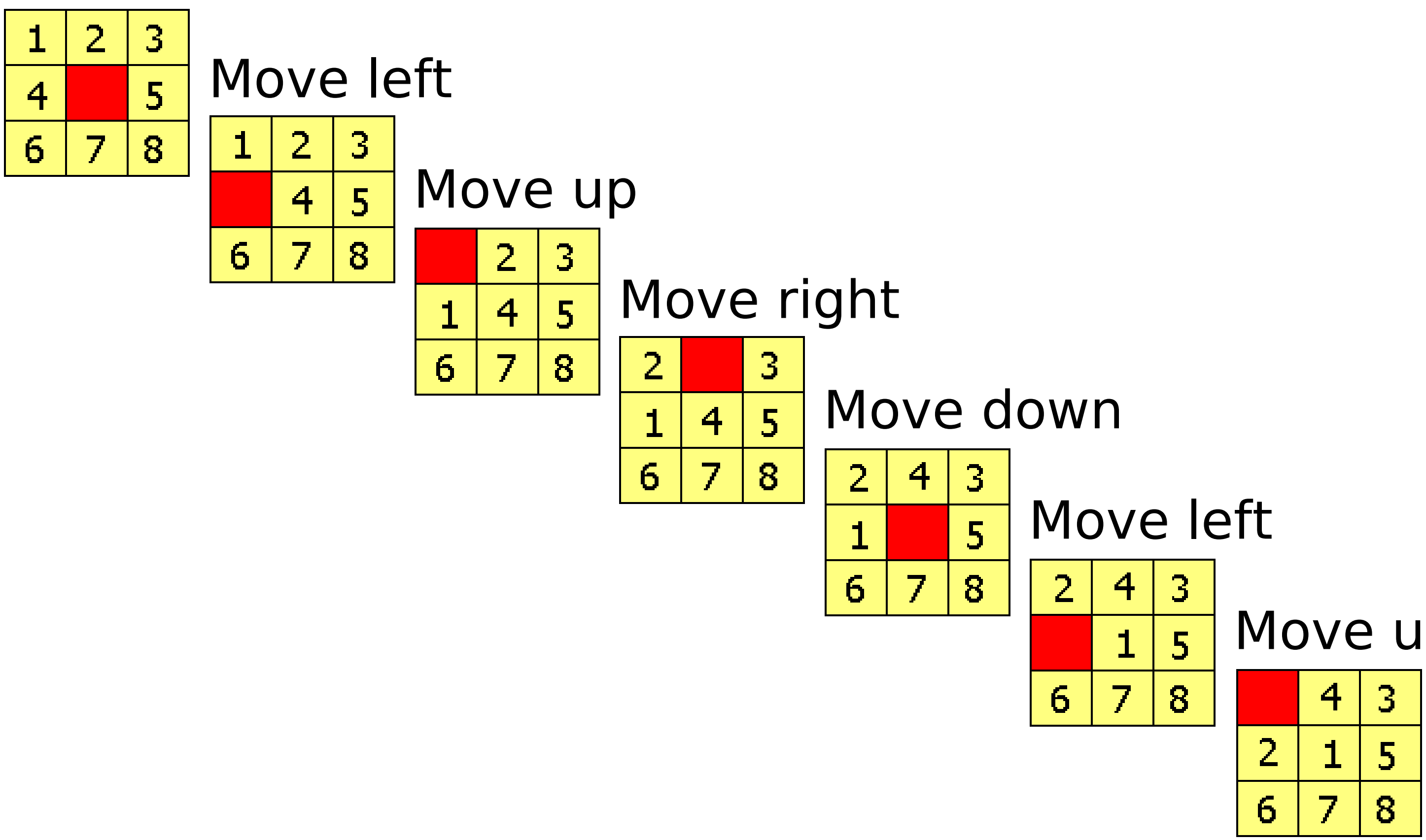


A sample problem

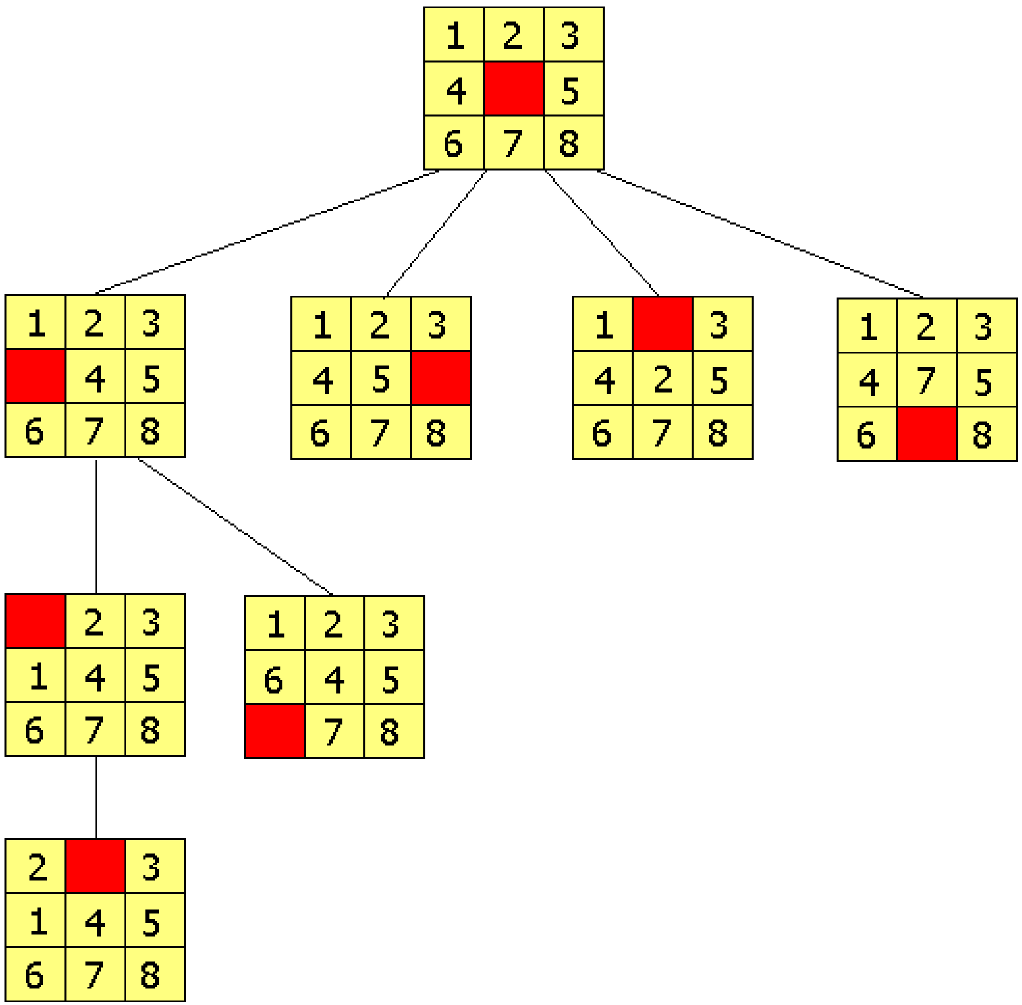
- The 8s Puzzle problem consists of finding a sequence of moves that transform a starting configuration into a goal configuration.
- Possible start and goal configurations are shown in the figure at right.



One solution to the problem



A partial depth-first search tree



The Eights Puzzle

?- solve.
Enter a starting puzzle:

|: 1 2 3
|: 4 0 5
|: 6 7 8
Enter a goal puzzle:

|: 0 4 3
|: 2 1 5
|: 6 7 8
Enter a depth bound (1..9): |: 6

1 2 3
4 0 5
6 7 8

1 2 3
0 4 5
6 7 8

0 2 3
1 4 5
6 7 8

2 0 3
1 4 5
6 7 8

2 4 3
1 0 5
6 7 8

2 4 3
0 1 5
6 7 8

0 4 3
2 1 5
6 7 8

Yes

Solving the problem

- We represent the puzzle using a list of 9 numbers, with 0 representing the “empty tile”.

1	2	3
4		5
6	7	8

= [1,2,3,4,0,5,6,7,8]

Solving the problem

- We make moves using rules of the form **move(puzzle1, puzzle2).** There are 24 of these in all.
- For example, the following two rules describe all moves that can be made when the empty tile is in the upper left corner:

```
move([0,B2,B3,B4,B5,B6,B7,B8,B9],  % move right
     [B2,0,B3,B4,B5,B6,B7,B8,B9]).

move([0,B2,B3,B4,B5,B6,B7,B8,B9],  % move down
     [B4,B2,B3,0,B5,B6,B7,B8,B9]).
```

The solve rule

- The workhorse of the program is the solve rule, with form:

  solve(S, G, SL1, SL2, Depth, Bound)
- Where
  - S = start puzzle
  - G = goal puzzle
  - SL1 is a list of states (input)
  - SL2 is a list of states (output)
  - Depth is the current depth of the search
  - Bound is the depth bound

The solve rule

```
solve(S,S,L,[S|L],Depth,Bound).

solve(S, G, L1, L2, Depth,Bound) :-
  Depth < Bound,
  not(S == G),
  move(S, S1),
  not(member(S,L1)),
  D is Depth+1,
  solve(S1,G,[S|L1],L2,D,Bound).
```

Prolog arithmetic

- Arithmetic expressions are evaluated with the built in predicate **is** which is used as an infix operator :

  variable **is** *expression*
- Example,

  ?- X is 3 * 4.
  X = 12
  yes
- Prolog has standard arithmetic operators:
  - +, -, *, / (real division), // (integer division), mod, and **
- Prolog has relational operators:
  - =, \=, >, >=, <, =<

Applications

- Intelligent systems
- Complicated knowledge databases
- Natural language processing
- Logic data analysis

Conclusions

Strengths:

- Strong ties to formal logic
- Many algorithms become trivially simple to implement

Weaknesses:

- Complicated syntax
- Difficult to understand programs at first sight

Issues

- What applications can Prolog excel at?
- Is Prolog suited for large applications?
- Would binding the Prolog engine to another language be a good idea?

End of Lecture