Introduction to Prolog

Dr. Demetrios Glinos
University of Central Florida

CAP4630 – Artificial Intelligence

Today

- Basis of Prolog
- The Prolog Language and Interpreter
- Running Prolog
- Data Types
- Variables, Terms, and Lists
- Rules
- Facts
- Using the Interpreter
- Using Files
- Listings and Modules
- Backward Chaining
- Debug Mode
- Example: Map Coloring
- The Ultimate Query

Basis of Prolog

- General Resolution
  - Operates on unrestricted CNF clauses
  - Resolution is sound – truth-preserving
  - Resolution is complete – can resolve any entailed proposition
  - But it is NP-complete - has $O(2^n)$ time complexity
- Horn clause
  - a CNF clause that has *at most 1 positive literal*
    - Example: $\neg P \vee \neg Q \vee R$
      - which is equivalent to $(P \wedge Q) \Rightarrow R$
      - using Prolog syntax: R :- P, Q
  - Can use forward-chaining and backward-chaining algorithms
  - Deciding entailment can be done in linear time

The Prolog Language and Interpreter

- Prolog is a declarative language, not procedural, OO, or functional
  - e.g., Prolog syntax for $\neg P \vee \neg Q \vee R \equiv (P \wedge Q) \Rightarrow R$ is R :- P, Q.
- Prolog programs
  - a program is a KB of statements or *clauses*
  - statements are either facts or rules
  - statements end in period "." character
  - comment lines start with "%" character
  - variables start with uppercase letter or underscore
    - everything else starts with lowercase letter
- Prolog interpreter
  - an engine for resolving logical expressions using Horn clauses
  - we execute a program by posing a query expression (goal)
  - Prolog negates query and uses backward chaining to find resolution refutation
  - unifies variables with constants and returns the bindings that make query true
  - if more than one binding, we can cycle through them using the ";" ("or" operator)

```
fact1.
fact2.
...
factn.

rule1.
rule2.
...
rulem.
```

Running Prolog

- Starting and stopping SWI-Prolog (from a command window)

$ swipl

```
Welcome to SWI-Prolog (Multi-threaded, 64 bits, Version 7.2.3)
Copyright (c) 1990-2015 University of Amsterdam, VU Amsterdam
SWI-Prolog comes with ABSOLUTELY NO WARRANTY. This is free software,
and you are welcome to redistribute it under certain conditions.
Please visit http://www.swi-prolog.org for details.

For help, use ?- help(Topic). or ?- apropos(Word).

?-
```

?- halt. ← *note the period (".") at the end, which is required*

- Documentation: http://www.swi-prolog.org/pldoc/doc_for?object=manual

Editing a file using the SWI-Prolog IDE

For either a new or existing file:

?- edit(file('myfile.pl')).

For an existing file:

?- edit('myfile.pl').

In the GUI:

- choose File/Save buffer to save changes
- choose Compile/Compile buffer to load it so can then pose query from command line

Running Prolog

- Loading and unloading a program file:

  ?- [myfile]. ← *do not* use the ".pl" extension

  ?- unload_file('myfile.pl'). ← *must* use the ".pl" extension

- Listing active predicates:
  - all predicates: listing.
  - for a module: module_name:listing.

```
?- utils:listing.

empty_set([]).

add_to_set(B, A, A) :-
        member(B, A), !.
add_to_set(A, B, [A|B]).

remove_from_set(_, [], []).
remove_from_set(A, [A|B], B) :- !.
remove_from_set(B, [A|C], [A|D]) :-
        remove_from_set(B, C, D), !.

member_set(A, B) :-
        member(A, B).
```

Data Types

- **Alphabet**
  - upper and lower case letters, digits, underscore (“_”),
  - some special characters (+ , - , * , / , < , > , = , : , . , & , ~)

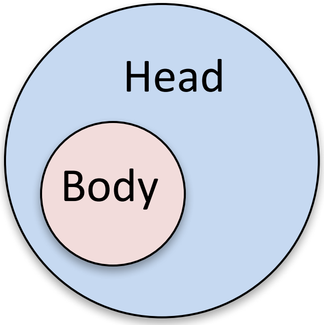
- **Atom**
  - a general-purpose name with no particular meaning
  - **must start with a lowercase letter or be enclosed in single-quotes** (‘Peter’) to distinguish it from a variable
  - the empty list (“[]”) is an atom

- **Numbers**
  - integers
  - floats

Variables, Terms, Lists

- **Variable**
  - **must begin with an uppercase letter or underscore**
  - can contain letters, numbers, and underscores
  - serves as a placeholder for arbitrary terms
  - can become bound to a specific term via *unification*
  - solo underscore "_" denotes an anonymous variable and means "any term" and does not mean the same value if it occurs more than once in a predicate
- **Term** (aka "predicate")
  - consists of an atom as a "functor" and zero or more arguments
  - arguments can be other compound terms
  - syntax: functor(arg1, arg2, ..., argn), where n is the "arity"
- **List**
  - enclose elements within square brackets, separated by commas
    - empty list: []
    - list with elements: [e1, e2, ..., en]

Rules



- **Prolog program**
  - a set of Horn clauses
  - the clauses can be rules or facts
- **Rule**
  - syntax: **Head :- Body**
  - meaning: "Head is true **if** Body is true" (i.e., Body $\Rightarrow$ Head)
  - Body consists of one or more atoms and/or predicates connected by
    - comma (",") – logical conjunction, or
    - semicolon (";") – logical disjunction
    - The atoms and predicates in the body are called ***goals***
  - Example: duck(X) :- looks_like_duck(X) , quacks_like_duck(X).

Facts

- **Fact**
  - A clause with an empty body
  - Can be an atom (constant).
    - Example: sunny.
  - Can be a term (predicate):
    - Example: student(tom).
      equivalent to student(tom) :- true.
    - Note: student(X) is still a fact, but it is not bound to a particular term

A fact is not the same thing as a 0-arity predicate

Using the Interpreter

- Start/stop Prolog
  - enter swipl to launch SWI-Prolog from the folder where you have (or wish to create) the file
  - use halt. to quit
- use assert/1 to add a fact or predicate
- use retractall/1 to remove facts and predicates

queries

```
?- assert(apple).
true.

?- assert(pear).
true.

?- assert(fruit(apple)).
true.

?- assert(fruit(pear)).
true.

?- fruit(X).
X = apple ;
X = pear.

?- retractall(pear).
true.

?- fruit(X).
X = apple ;
X = pear.

?- retractall(fruit(pear)).
true.

?- fruit(X).
X = apple.

?-
```

Using Files

- Much easier to have programs in files
  - create/edit using any text editor
  - create/edit using the SWI-Prolog IDE
  - comment lines start with “%” character

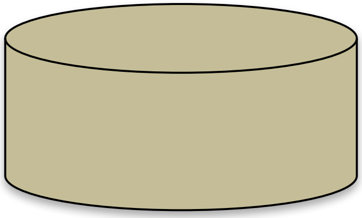
- Using a file
  - consult(‘your_file.pl’).
  - [your_file].
  - “compile buffer” from the IDE has same result
  - to unload: unload_file('your_file.pl').

- SWI-Prolog IDE
  - enter edit(file(‘your_file.pl’)). at the prompt
  - edit and save buffer

Example: KB of Facts

- Consider this simple KB of facts:

```
likes(peter, wine).
likes(peter, cheese).
likes(peter, mary).
```



Knowledge Base

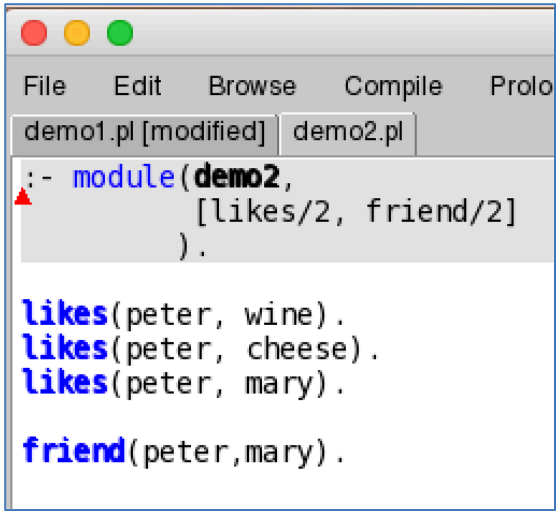
- We can pose several queries

1. likes(peter, wine).
2. likes(peter, pizza).
3. likes(peter, X).
4. likes(X, cheese).
5. likes(X, Y).

```
demo:
  $ swipl
  ?- edit('demo1.pl').
  [demo1].
  likes(X,cheese).
```

Listings and Modules

- listing.
  - lists all of the clauses currently loaded
  - module_name:listing. ← *lists all the predicates in the named module*
  - We can see our "likes/2" predicate
- abolish/2
  - deletes clauses for a particlar predicate
  - Example: abolish(likes,2).
- unload_file/1
  - deletes all clauses loaded from a particular file
  - Example: unload_file('demo1.pl').
- module
  - Prolog allows us to place predicates in modules and control their export
  - Syntax: :- module(module_name [list of predicate names with arities]).
  - use module_name:listing. to list only the clauses for the module



demo2.pl

Backward Chaining

- Prolog uses backward-chaining (goal-directed reasoning)

- Basic idea:
  - Given query q, this is the initial goal
  - If query is known to be true (i.e., it is a fact), we're done.
  - Else:
    1. Look for rules with head that match query
    2. replace goal with subgoals from body of rule
    3. look to satisfy each of those
    4. if reach a known fact, then
       a) subgoal satisfied
       b) unify variable with fact
       c) process next subgoal, if any
    5. Else backtrack to next rule that satisfies subquery

Example: KB with Rules

- Consider this program:

```
[?- demo3:listing.

is_a(A, B) :-
        looks_like(A, B),
        acts_like(A, B).

acts_like(duck, animal1).
acts_like(duck, animal3).
acts_like(dog, animal2).
acts_like(dog, animal4).

looks_like(duck, animal1).
looks_like(dog, animal2).
looks_like(duck, animal3).
looks_like(duck, animal4).
true.
```

demo3.pl

- We can ask fact-based questions: e.g., looks_like(duck,X).
- We can ask questions requiring inference (resolution): e.g., is_a(duck,X).

Example: KB with Rules

- How Prolog decides the query: is_a(duck,X).
  - There are no facts of the form is_a(duck, _)
  - But there is a rule head of form is_a(A, B)
  - Prolog matches the "duck" in the query to A and starts looking for a B that works
  - goal on stack is is_a(duck, _tmp1), where _tmp1 is a temporary variable
  - since no fact for is_a(duck, _tmp1), prolog expands that goal and add the predicates in the body of the rule to the stack
  - stack now has looks_like(duck, _tmp1) on top of acts_like(duck, _tmp1) on top of is_a(duck, _tmp1)
  - Prolog finds match with looks_like(duck, animal1) and *unifies _tmp1 with animal1*
  - subgoal satified, so now look for acts_like(duck, animal1), which is a known fact
  - subgoal satisfied, top goal on stack is is_a(duck, _tmp1)
  - since _tmp1 unified with animal1, goal is satisfied
  - Prolog reports X = animal1

```
?- demo3:listing.

is_a(A, B) :-
        looks_like(A, B),
        acts_like(A, B).

acts_like(duck, animal1).
acts_like(duck, animal3).
acts_like(dog, animal2).
acts_like(dog, animal4).

looks_like(duck, animal1).
looks_like(dog, animal2).
looks_like(duck, animal3).
looks_like(duck, animal4).
true.
```

looks_like(duck,_tmp1)
acts_like(duck,_tmp1)
is-a(duck,_tmp1)

Debug Mode

- We can follow Prolog's search process by using the trace predicate to turn on debug mode

  - trace -- turns on debug mode for the current goal
  - nodebug – turns it off
  - press 'enter' key to step to next call
  - see also spy predicte

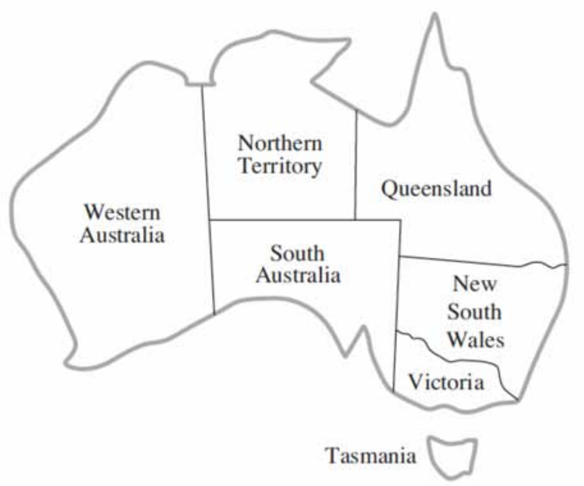
demo3.pl

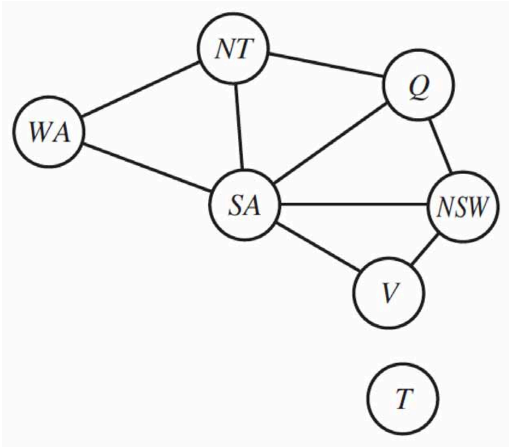
```
?- trace.
true.

[trace]  ?- is_a(duck,X).
   Call: (6) demo3:is_a(duck, _G3118) ? creep
   Call: (7) demo3:looks_like(duck, _G3118) ? creep
   Exit: (7) demo3:looks_like(duck, animal1) ? creep
   Call: (7) demo3:acts_like(duck, animal1) ? creep
   Exit: (7) demo3:acts_like(duck, animal1) ? creep
   Exit: (6) demo3:is_a(duck, animal1) ? creep
X = animal1 ;
   Redo: (7) demo3:acts_like(duck, animal1) ? creep
   Fail: (7) demo3:acts_like(duck, animal1) ? creep
   Redo: (7) demo3:looks_like(duck, _G3118) ? creep
   Exit: (7) demo3:looks_like(duck, animal3) ? creep
   Call: (7) demo3:acts_like(duck, animal3) ? creep
   Exit: (7) demo3:acts_like(duck, animal3) ? creep
   Exit: (6) demo3:is_a(duck, animal3) ? creep
X = animal3 ;
   Redo: (7) demo3:looks_like(duck, _G3118) ? creep
   Exit: (7) demo3:looks_like(duck, animal4) ? creep
   Call: (7) demo3:acts_like(duck, animal4) ? creep
   Fail: (7) demo3:acts_like(duck, animal4) ? creep
   Fail: (6) demo3:is_a(duck, _G3118) ? creep
false.

[trace]  ?-
```

Example: Map Coloring

Let's create a Prolog program for our map coloring CSP

Q: What do we need to declare to specify this problem?



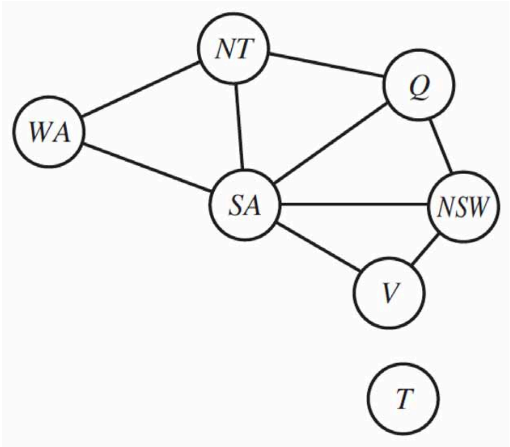


Example: Map Coloring

Let's create a Prolog program for our map coloring CSP

Q: What do we need to declare to specify this problem?

- the domains (i.e., colors)

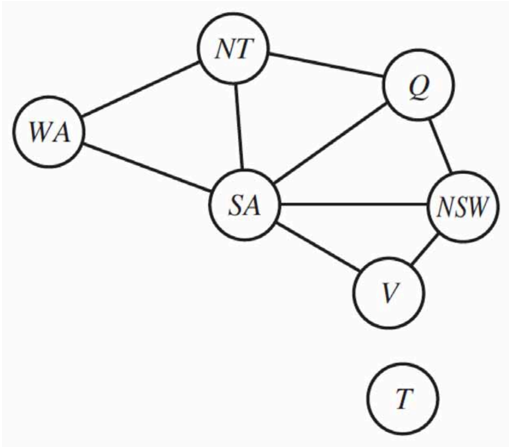


Example: Map Coloring

Let's create a Prolog program for our map coloring CSP

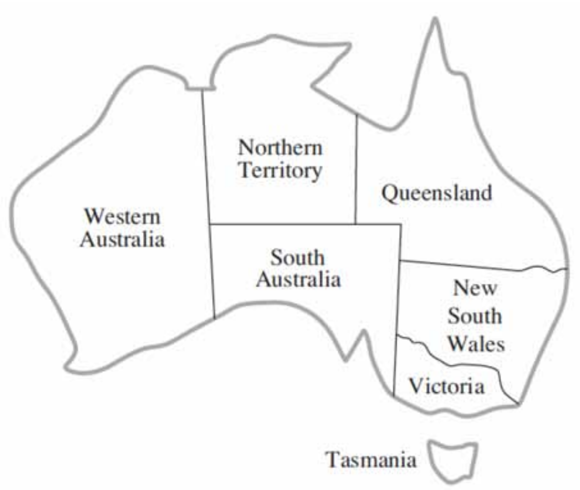
Q: What do we need to declare to specify this problem?

- the domains (i.e., colors)
- the constraints that no two adjacent territories have the same color



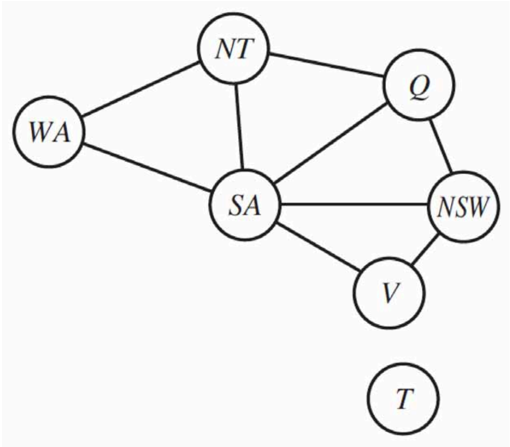
Example: Map Coloring

Let's create a Prolog program for our map coloring CSP



Q: What do we need to declare to specify this problem?

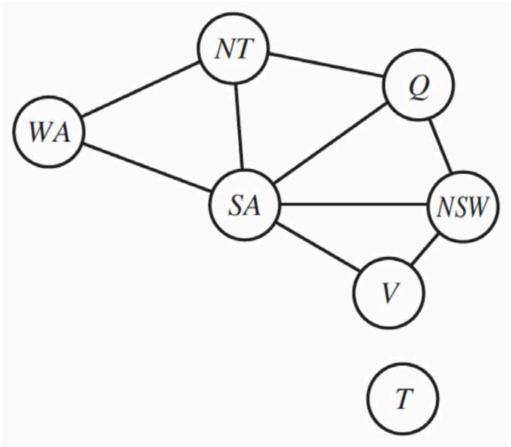
- the domains (i.e., colors)
- the constraint that no two adjacent territories have the same color
- the topology of Australia



Example: Map Coloring

Let's create a Prolog program for our map coloring CSP

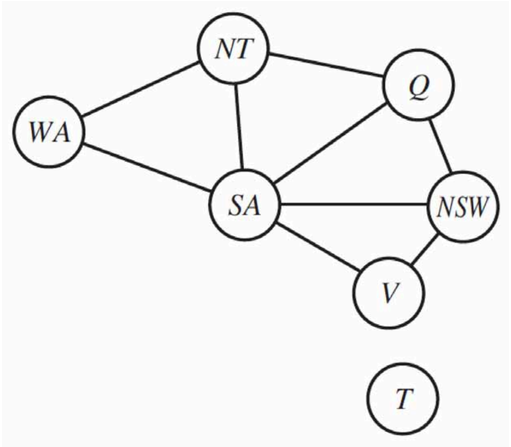
```
color( red ).
color( green ).
color( blue ).
```



Example: Map Coloring

Let's create a Prolog program for our map coloring CSP

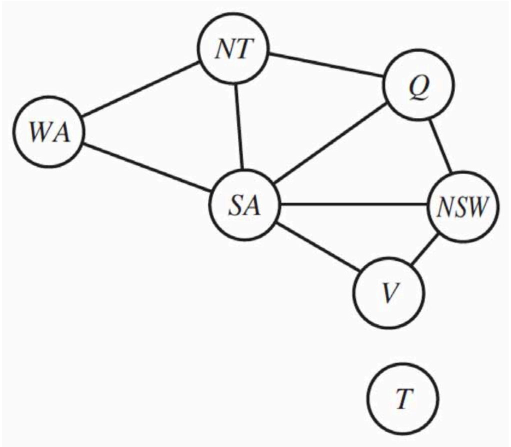
```
color( red ).
color( green ).
color( blue ).

nextto( Acolor, Bcolor ) :-
        color( Acolor ),
        color( Bcolor ),
        Acolor \= Bcolor.
```



Example: Map Coloring

Let's create a Prolog program for our map coloring CSP

```
color( red ).
color( green ).
color( blue ).

nextto( Acolor, Bcolor ) :-
        color( Acolor ),
        color( Bcolor ),
        Acolor \= Bcolor.

australia(WA,NT,SA,Q,NSW,V,T) :-
        nextto( WA, NT ), nextto( WA, SA ),
        nextto( NT, Q ), nextto( NT, SA ),
        nextto( Q, NSW ), nextto( Q, SA ),
        nextto( NSW, V ), nextto( NSW, SA ),
        nextto( V, SA ).
```



Example: Map Coloring

Q: What is the query?

A:

```
color( red ).
color( green ).
color( blue ).

nextto( Acolor, Bcolor ) :-
        color( Acolor ),
        color( Bcolor ),
        Acolor \= Bcolor.

australia(WA,NT,SA,Q,NSW,V,T) :-
        nextto( WA, NT ), nextto( WA, SA ),
        nextto( NT, Q ), nextto( NT, SA ),
        nextto( Q, NSW ), nextto( Q, SA ),
        nextto( NSW, V ), nextto( NSW, SA ),
        nextto( V, SA ).
```

Example: Map Coloring

Q: What is the query?

A: australia(WA,NT,SA,Q,NSW,V,T).

```
color( red ).
color( green ).
color( blue ).

nextto( Acolor, Bcolor ) :-
        color( Acolor ),
        color( Bcolor ),
        Acolor \= Bcolor.

australia(WA,NT,SA,Q,NSW,V,T) :-
        nextto( WA, NT ), nextto( WA, SA ),
        nextto( NT, Q ), nextto( NT, SA ),
        nextto( Q, NSW ), nextto( Q, SA ),
        nextto( NSW, V ), nextto( NSW, SA ),
        nextto( V, SA ).
```

demo:

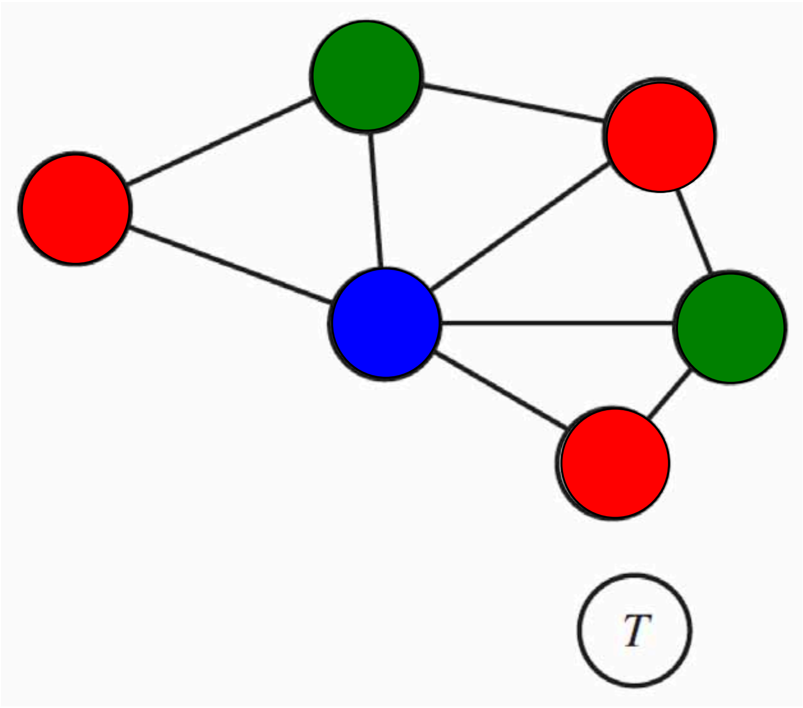
```
edit('australia.pl').
[australia]
australia:listing.
australia(WA,NT,SA,Q,NSW,V,T).
```

Example: Map Coloring

Q: What is the query?

A: australia(WA,NT,SA,Q,NSW,V,T).

```
?- australia(A, B, C, D, E, F, _).
A = D, D = F, F = red,
B = E, E = green,
C = blue ;
A = D, D = F, F = red,
B = E, E = blue,
C = green ;
A = D, D = F, F = green,
B = E, E = red,
C = blue ;
A = D, D = F, F = green,
B = E, E = blue,
C = red ;
A = D, D = F, F = blue,
B = E, E = red,
C = green ;
A = D, D = F, F = blue,
B = E, E = green,
C = red ;
false.

?- 
```



(first of 6 solutions shown)

Note how Prolog treats Tasmania

The Ultimate Query

Explain this query and Prolog's response:

```
?- X.
% ... 1,000,000 ............ 10,000,000 years later
%
%       >> 42 << (last release gives the question)
?-
```