Prolog

Forest Agostinelli
University of South Carolina

Topics Covered in This Class

- **Part 1: Search**
  - Pathfinding
    - Uninformed search
    - Informed search
  - Adversarial search
  - Optimization
    - Local search
    - Constraint satisfaction
- **Part 2: Knowledge Representation and Reasoning**
  - Propositional logic
  - First-order logic
  - Prolog
- **Part 3: Knowledge Representation and Reasoning Under Uncertainty**
  - Probability
  - Bayesian networks
- **Part 4: Machine Learning**
  - Supervised learning
    - Inductive logic programming
    - Linear models
    - Deep neural networks
    - PyTorch
  - Reinforcement learning
    - Markov decision processes
    - Dynamic programming
    - Model-free RL
  - Unsupervised learning
    - Clustering
    - Autoencoders

Outline

- Prolog Introduction
- Horn Clauses and SLD Resolution
- Prolog basics
  - Rules and Facts
  - Queries
  - Tracing
  - Negation as failure
  - Recursion
- Examples
  - Map coloring
  - Recursion example
  - Solving brainteasers
  - Lists in Prolog
  - General map coloring
  - Pathfinding

Prolog Resources

- Levesque, Hector. "Thinking as Computation." *A First Course* (2012).
- https://www.metalevel.at/prolog, Markus Triska
  - The Power of Prolog YouTube Channel
- Sterling, Leon, and Ehud Y. Shapiro. *The art of Prolog: advanced programming techniques*. MIT press, 1994.
  - PDF freely available
- Flach, Peter. *Simply logical: intelligent reasoning by example*. John Wiley & Sons, Inc., 1994.
  - https://too.simply-logical.space/src/simply-logical.html
- Other resources

Prolog

- Prolog (**Pro**gramming **log**ic)
- Declarative programming language
  - Control implements SLD resolution
- Based on first-order logic
  - Uses only Horn clauses so that it is compatible with SLD resolution
- A prolog script defines a knowledge base
- A query determines if a goal clause is entailed by the knowledge base
- Although it is a declarative language, we may have to read code procedurally from time to time
- Prolog has been used for theorem proving, natural language processing, AI planning, software verification, machine learning, amongst others.

Programming Paradigms

- Imperative
  - The programmer instructs the machine how to change its state
  - Explicit control: if, when, while, etc.
  - C, C++, Java, Python, etc.
- Declarative
  - The programmer describes the solution they would like to obtain, but not how to compute it
  - Implicit control: in Prolog, this is SLD resolution that is often implemented using a depth-first search
  - Prolog, SQL, regular expressions, etc.

https://en.wikipedia.org/wiki/Programming_paradigm

Outline

- Prolog Introduction
- Horn Clauses and SLD Resolution
- Prolog basics
  - Rules and Facts
  - Queries
  - Tracing
  - Negation as failure
  - Recursion
- Examples
  - Map coloring
  - Recursion example
  - Solving brainteasers
  - Lists in Prolog
  - General map coloring
  - Pathfinding

Horn Clauses

- A disjunction of literals (predicates in FOL) with at most one positive (unnegated) literal
- Definite clauses
  - Exactly one positive literal
  - If A1, A2, and A3 holds, then C holds
  - $\neg A1(X) \vee \neg A2(X) \vee \neg A3(X) \vee C(X)$
    - $A1(X) \wedge A2(X) \wedge A3(X) \rightarrow C(X)$
- Facts
  - Only one positive literal
  - F holds
  - $F(X)$
    - $true \rightarrow F(X)$
- Goal clauses
  - No positive literals
  - “Show that G1, G2, and G3 hold.” Each of these can be called a “goal”
  - We can then negate what we are trying to prove to do proof by contradiction
  - $\neg G1(X) \vee \neg G2(X) \vee \neg G3(X)$
  - $G1(X) \wedge G2(X) \wedge G3(X) \rightarrow false$

Horn Clauses

- Horn Clauses allow the use of backward chaining, which can be very efficient compared to doing resolution on clauses with arbitrary structure
- This is a subset of first-order predicate logic that is Turing complete

Backward Chaining

- A query is given in the form of a goal clause. We would like to do proof by contradiction. Every literal is seen as a goal. Every goal must be proven in order to show a contradiction.
  - $\neg G1(X) \vee \neg G2(X) \vee \neg G3(X)$
  - $G1(X) \wedge G2(X) \wedge G3(X) \rightarrow false$
- For each goal, we look for a definite clause or a fact that unifies with the goal
  - $A1(foo), A2(foo) \rightarrow G1(foo)$
  - $\theta = \{X/foo\}$
- Just like in resolution, we substitute foo in for all variables in the clause.
- We then substitute in antecedents in for the goal and continue
  - $A1(foo), A2(foo), G2(X) \wedge G3(X) \rightarrow false$
- If we are able to unify with a fact, then we can say that we have proven the goal
  - For example, if $A1(foo)$ is in our knowledge base
- This is just proof by contradiction, but exploiting structure for efficiency

Backward Chaining: Example

- $healthy(X) \wedge mature(X) \rightarrow ripe(X)$
- $leafygreen(X) \wedge green(X) \wedge firm(X) \rightarrow healthy(X)$
- $kale(X) \rightarrow leafygreen(X)$
- $collard(X) \rightarrow leafygreen(X)$
- $kale(kale1)$
- $collard(collard1)$
- $green(kale1)$
- $firm(kale1)$
- $mature(kale1)$

Backward Chaining: Example

- Query: $kale(X) \wedge ripe(X)$
  - Is there ripe kale?
- Negated query: $\neg kale(X) \vee \neg ripe(X)$
- Find a statement with $kale$ in the head that can unify with $kale(X)$ and find a statement with $ripe$ in the head that can unify with $ripe(X)$
- $kale(X)$
  - $\theta = \{X/kale1\}$
  - succeed
- $healthy(kale1) \wedge mature(kale1) \rightarrow ripe(kale1)$
  - $leafygreen(kale1) \wedge green(kale1) \wedge firm(kale1) \rightarrow healthy(kale1)$
    - $kale(kale1) \rightarrow leafygreen(kale1)$
      - $kale(kale1)$
        - succeed
    - $green(kale1)$
      - succeed
    - $firm(kale1)$
      - succeed
  - $mature(kale1)$
    - Succeed
- succeed $\{X/kale1\}$

Backward Chaining

- Query: $collard(X) \wedge ripe(X)$
  - Are there ripe collard greens?
- Negated query: $\neg collard(X) \vee \neg ripe(X)$
- $collard(collard1)$
  - $\theta = \{X/collard1\}$
  - Succeed
- $healthy(collard1) \wedge mature(collard1) \rightarrow ripe(collard1)$
  - $leafygreen(collard1) \wedge green(collard1) \wedge firm(collard1) \rightarrow healthy(collard1)$
    - $kale(collard1) \rightarrow leafygreen(collard1)$
      - Fail, backtrack
    - $collard(collard1) \rightarrow leafygreen(collard1)$
      - $collard(collard1)$
        - succeed
    - $green(collard1)$
      - Fail, backtrack
    - Fail, backtrack
  - Fail, backtrack
- Fail

Backward Chaining Algorithm

- Or
  - We only need one rule to match
- AND
  - We need to find a matching rule for all goals
- This can be viewed as a depth first search
- Similar to backtracking search that we saw in constraint satisfaction problems

function FOL-BC-ASK(KB, $query$) **returns** a generator of substitutions
  return FOL-BC-OR(KB, $query$, { })

function FOL-BC-OR(KB, $goal$, θ) **returns** a substitution
  for each $rule$ in FETCH-RULES-FOR-GOAL(KB, $goal$) **do**
    ($lhs \Rightarrow rhs$) ← STANDARDIZE-VARIABLES($rule$)
    for each θ' **in** FOL-BC-AND(KB, lhs, UNIFY(rhs, $goal$, θ)) **do**
      yield θ'

function FOL-BC-AND(KB, $goals$, θ) **returns** a substitution
  if $\theta = failure$ **then return**
  else if LENGTH($goals$) = 0 **then yield** θ
  else
    $first$,$rest$ ← FIRST($goals$), REST($goals$)
    for each θ' **in** FOL-BC-OR(KB, SUBST(θ, $first$), θ) **do**
      for each θ'' **in** FOL-BC-AND(KB, $rest$, θ') **do**
        yield θ''

Figure 9.6 A simple backward-chaining algorithm for first-order knowledge bases.

SLD Resolution

- Selective Linear Definite Resolution
  - Selective: We must select goals to resolve in a certain order and rules/facts from the knowledge base to try to perform resolution. In Prolog, goals are selected left to right. Rules/facts used for resolution are selected from top to bottom (the order they were written in the Prolog script). The order of goals and rules/facts can have a significant impact on performance.
  - Linear: Proof is linear in the number of clauses
  - Definite: Definite clauses
- A popular way to carry out SLD resolution is through a depth-first search
  - Saves memory
  - Although resolution is refutation complete, using a depth-first search leads to an incomplete strategy as it can get stuck in infinite loops
  - Prolog makes this choice
- However, one can also use breadth-first search, A* search, etc.

Outline

- Prolog Introduction
- Horn Clauses and SLD Resolution
- Prolog basics
  - Rules and Facts
  - Queries
  - Tracing
  - Negation as failure
  - Recursion
- Examples
  - Map coloring
  - Recursion example
  - Solving brainteasers
  - Lists in Prolog
  - General map coloring
  - Pathfinding

Facts and Rules

- In Prolog the knowledge base is made up of facts and rules (definite clauses)
- All logical statements end with a "."
- In implication, the consequent is on the left and the antecedent is on the right
- Implication is denoted by ":-"
- Conjunction is denoted ","
- Comments start with "%"
- All predicates and constants start with a lowercase letter
- All variables start with an uppercase letter

```
% facts
human(socrates).
human(plato).
human(aristotle).

% rules
mortal(X) :- human(X).
```

Queries

- The prolog interpreter uses the prompt “?”
- The user should then enter a conjunction of literals
- This can be seen as a goal clause
  - $G1(X) \wedge G2(X) \wedge G3(X) \rightarrow false$

```
?- mortal(socrates).
true.

?- mortal(plato).
true.

?- mortal(aristotle).
true.
```

Queries

- Variables can be used in queries
- This can be interpreted as asking if there exists any substitution to these variables that results in a logical sentence that is entailed by the knowledge base
- By default, Prolog searches for all substitutions using a depth-first search
- After each result one can press "enter" to stop or ";" which can be interpreted as logical or, to get the next result

```
?- mortal(X).
X = socrates ;
X = plato ;
X = aristotle.
```

Tracing Queries

- One can use trace to trace exactly how Prolog is doing its depth-first search

```
[trace]  ?- mortal(X).
   Call: (10) mortal(_17760) ? creep
   Call: (11) human(_17760) ? creep
   Exit: (11) human(socrates) ? creep
   Exit: (10) mortal(socrates) ? creep
X = socrates ;
   Redo: (11) human(_17760) ? creep
   Exit: (11) human(plato) ? creep
   Exit: (10) mortal(plato) ? creep
X = plato ;
   Redo: (11) human(_17760) ? creep
   Exit: (11) human(aristotle) ? creep
   Exit: (10) mortal(aristotle) ? creep
X = aristotle.
```

Prolog: Food Example

```
% rules
ripe(X) :- healthy(X), mature(X).
healthy(X) :- leafy_green(X), green(X), firm(X).
leafy_green(X) :- kale(X).
leafy_green(X) :- collard(X).


% facts
collard(collard1).
kale(kale1).

green(kale1).
firm(kale1).
mature(kale1).
```

Prolog: Food Example

- If there was more searching to be done, but Prolog found no matches, it will end with false.

```
?- kale(X), ripe(X).
X = kale1 ;
false.
```

Prolog: Food Example

```
% rules
ripe(X) :- healthy(X), mature(X).
healthy(X) :- leafy_green(X), green(X), firm(X).
leafy_green(X) :- kale(X).
leafy_green(X) :- collard(X).


% facts
collard(collard1).
kale(kale1).

green(kale1).
firm(kale1).
mature(kale1).
```

```
[trace]  ?- kale(X), ripe(X).
   Call: (11) kale(_15116) ? creep
   Exit: (11) kale(kale1) ? creep
   Call: (11) ripe(kale1) ? creep
   Call: (12) healthy(kale1) ? creep
   Call: (13) leafy_green(kale1) ? creep
   Call: (14) kale(kale1) ? creep
   Exit: (14) kale(kale1) ? creep
   Exit: (13) leafy_green(kale1) ? creep
   Call: (13) green(kale1) ? creep
   Exit: (13) green(kale1) ? creep
   Call: (13) firm(kale1) ? creep
   Exit: (13) firm(kale1) ? creep
   Exit: (12) healthy(kale1) ? creep
   Call: (12) mature(kale1) ? creep
   Exit: (12) mature(kale1) ? creep
   Exit: (11) ripe(kale1) ? creep
X = kale1 ;
   Redo: (13) leafy_green(kale1) ? creep
   Call: (14) collard(kale1) ? creep
   Fail: (14) collard(kale1) ? creep
   Fail: (13) leafy_green(kale1) ? creep
   Fail: (12) healthy(kale1) ? creep
   Fail: (11) ripe(kale1) ? creep
false.
```

Prolog: Food Example

```
% rules
ripe(X) :- healthy(X), mature(X).
healthy(X) :- leafy_green(X), green(X), firm(X).
leafy_green(X) :- kale(X).
leafy_green(X) :- collard(X).


% facts
collard(collard1).
kale(kale1).

green(kale1).
firm(kale1).
mature(kale1).
```

```
[trace]  ?- collard(X), ripe(X).
   Call: (11) collard(_49574) ? creep
   Exit: (11) collard(collard1) ? creep
   Call: (11) ripe(collard1) ? creep
   Call: (12) healthy(collard1) ? creep
   Call: (13) leafy_green(collard1) ? creep
   Call: (14) kale(collard1) ? creep
   Fail: (14) kale(collard1) ? creep
   Redo: (13) leafy_green(collard1) ? creep
   Call: (14) collard(collard1) ? creep
   Exit: (14) collard(collard1) ? creep
   Exit: (13) leafy_green(collard1) ? creep
   Call: (13) green(collard1) ? creep
   Fail: (13) green(collard1) ? creep
   Fail: (12) healthy(collard1) ? creep
   Fail: (11) ripe(collard1) ? creep
false.
```

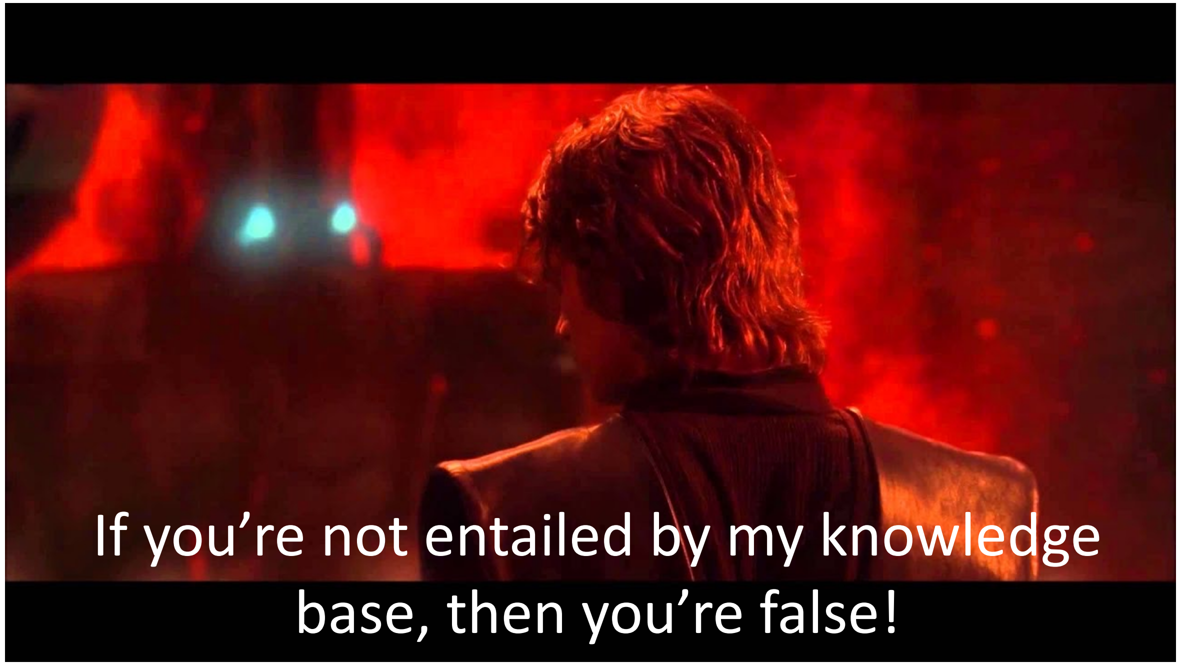
Prolog: Food Example (Swap Goals)

- It first finds kale1, which proves ripe(kale1), but cannot prove collard(kale1).
- It then backtracks and finds it cannot prove ripe(collard1).
- The swapping of the predicates, though logically equivalent, results in different execution.

```
[trace]  ?- ripe(X), collard(X).
   Call: (11) ripe(_67952) ? creep
   Call: (12) healthy(_67952) ? creep
   Call: (13) leafy_green(_67952) ? creep
   Call: (14) kale(_67952) ? creep
   Exit: (14) kale(kale1) ? creep
   Exit: (13) leafy_green(kale1) ? creep
   Call: (13) green(kale1) ? creep
   Exit: (13) green(kale1) ? creep
   Call: (13) firm(kale1) ? creep
   Exit: (13) firm(kale1) ? creep
   Exit: (12) healthy(kale1) ? creep
   Call: (12) mature(kale1) ? creep
   Exit: (12) mature(kale1) ? creep
   Exit: (11) ripe(kale1) ? creep
   Call: (11) collard(kale1) ? creep
   Fail: (11) collard(kale1) ? creep
   Redo: (13) leafy_green(_67952) ? creep
   Call: (14) collard(_67952) ? creep
   Exit: (14) collard(collard1) ? creep
   Exit: (13) leafy_green(collard1) ? creep
   Call: (13) green(collard1) ? creep
   Fail: (13) green(collard1) ? creep
   Fail: (12) healthy(_67952) ? creep
   Fail: (11) ripe(_67952) ? creep
false.
```

Negation as Failure

- In FOL, our epistemological commitments allow us to say that the truth of something is unknown
  - This is an open world assumption
- If Prolog cannot entail something from its knowledge base, then it is assumed to be false
  - This is a closed world assumption
  - Can still work with SLD resolution
- This is known as negation as failure
- This is **not** negation in the logical sense that we are used to and must be used with care



Negation as Failure

- Even though collard2 has never been mentioned, Prolog makes assertions about it due to the closed world assumption

```
% rules
ripe(X) :- healthy(X), mature(X).
healthy(X) :- leafy_green(X), green(X), firm(X).
leafy_green(X) :- kale(X).
leafy_green(X) :- collard(X).


% facts
collard(collard1).
kale(kale1).

green(kale1).
firm(kale1).
mature(kale1).
```

```
?- collard(kale1).
false.

?- \+ collard(kale1).
true.

?- collard(collard1).
true.

?- \+ collard(collard1).
false.

?- collard(collard2).
false.

?- \+ collard(collard2).
true.
```

Negation as Failure

- These two sentences are intuitively equivalent but give different results. Why?

```
% rules
ripe(X) :- healthy(X), mature(X).
healthy(X) :- leafy_green(X), green(X), firm(X).
leafy_green(X) :- kale(X).
leafy_green(X) :- collard(X).


% facts
collard(collard1).
kale(kale1).

green(kale1).
firm(kale1).
mature(kale1).
```

```
?- ripe(X), \+ collard(X).
X = kale1 ;
false.

?- \+ collard(X), ripe(X).
false.
```

Negation as Failure

- collard(kale1) fails, so the statement succeeds

```
% rules
ripe(X) :- healthy(X), mature(X).
healthy(X) :- leafy_green(X), green(X), firm(X).
leafy_green(X) :- kale(X).
leafy_green(X) :- collard(X).


% facts
collard(collard1).
kale(kale1).

green(kale1).
firm(kale1).
mature(kale1).
```

```
[trace]  ?- ripe(X), \+ collard(X).
   Call: (11) ripe(_15194) ? creep
   Call: (12) healthy(_15194) ? creep
   Call: (13) leafy_green(_15194) ? creep
   Call: (14) kale(_15194) ? creep
   Exit: (14) kale(kale1) ? creep
   Exit: (13) leafy_green(kale1) ? creep
   Call: (13) green(kale1) ? creep
   Exit: (13) green(kale1) ? creep
   Call: (13) firm(kale1) ? creep
   Exit: (13) firm(kale1) ? creep
   Exit: (12) healthy(kale1) ? creep
   Call: (12) mature(kale1) ? creep
   Exit: (12) mature(kale1) ? creep
   Exit: (11) ripe(kale1) ? creep
   Call: (11) collard(kale1) ? creep
   Fail: (11) collard(kale1) ? creep
X = kale1
```

Negation as Failure

- collard(collard1) succeeds, so the statement fails
- When using negation in Prolog, make sure that the variable is **instantiated**
  - This means it has been given a tentative value through unification

```
% rules
ripe(X) :- healthy(X), mature(X).
healthy(X) :- leafy_green(X), green(X), firm(X).
leafy_green(X) :- kale(X).
leafy_green(X) :- collard(X).


% facts
collard(collard1).
kale(kale1).

green(kale1).
firm(kale1).
mature(kale1).
```

```
[trace]  ?- \+ collard(X), ripe(X).
   Call: (11) collard(_13964) ? creep
   Exit: (11) collard(collard1) ? creep
false.
```

Recursion

- It is possible for a predicate to call itself
- In this example, an ancestor is anyone that is in a previous line of mentors

```
% facts
human(socrates).
human(plato).
human(aristotle).

mentor(socrates, plato).
mentor(plato, aristotle).

% rules
mortal(X) :- human(X).
ancestor(X, Y) :- mentor(X, Y).
ancestor(X, Y) :- mentor(X, Z), ancestor(Z, Y).
```

Recursion

- ancestor(X, Y) :- mentor(X, Z), ancestor(Z, Y).

```
% facts
human(socrates).
human(plato).
human(aristotle).

mentor(socrates, plato).
mentor(plato, aristotle).

% rules
mortal(X) :- human(X).
ancestor(X, Y) :- mentor(X, Y).
ancestor(X, Y) :- ancestor(Z, Y), mentor(X, Z).
```

```
[trace]  ?- ancestor(X, aristotle).
   Call: (10) ancestor(_18298, aristotle) ? creep
   Call: (11) mentor(_18298, aristotle) ? creep
   Exit: (11) mentor(plato, aristotle) ? creep
   Exit: (10) ancestor(plato, aristotle) ? creep
X = plato ;
   Redo: (10) ancestor(_18298, aristotle) ? creep
   Call: (11) mentor(_18298, _23852) ? creep
   Exit: (11) mentor(socrates, plato) ? creep
   Call: (11) ancestor(plato, aristotle) ? creep
   Call: (12) mentor(plato, aristotle) ? creep
   Exit: (12) mentor(plato, aristotle) ? creep
   Exit: (11) ancestor(plato, aristotle) ? creep
   Exit: (10) ancestor(socrates, aristotle) ? creep
X = socrates ;
```

Recursion

- ancestor(X, Y) :- ancestor(Z, Y), mentor(X, Z).
  - Non-terminating.
  - Why?

```
% facts
human(socrates).
human(plato).
human(aristotle).

mentor(socrates, plato).
mentor(plato, aristotle).

% rules
mortal(X) :- human(X).
ancestor(X, Y) :- mentor(X, Y).
ancestor(X, Y) :- ancestor(Z, Y), mentor(X, Z).
```

```
[trace]  ?- ancestor(X, aristotle).
   Call: (10) ancestor(_12672, aristotle) ? creep
   Call: (11) mentor(_12672, aristotle) ? creep
   Exit: (11) mentor(plato, aristotle) ? creep
   Exit: (10) ancestor(plato, aristotle) ? creep
X = plato ;
   Redo: (10) ancestor(_12672, aristotle) ? creep
   Call: (11) ancestor(_18222, aristotle) ? creep
   Call: (12) mentor(_18222, aristotle) ? creep
   Exit: (12) mentor(plato, aristotle) ? creep
   Exit: (11) ancestor(plato, aristotle) ? creep
   Call: (11) mentor(_12672, plato) ? creep
   Exit: (11) mentor(socrates, plato) ? creep
   Exit: (10) ancestor(socrates, aristotle) ? creep
X = socrates ;
   Redo: (11) ancestor(_18222, aristotle) ? creep
   Call: (12) ancestor(_25600, aristotle) ? creep
   Call: (13) mentor(_25600, aristotle) ? creep
   Exit: (13) mentor(plato, aristotle) ? creep
   Exit: (12) ancestor(plato, aristotle) ? creep
   Call: (12) mentor(_18222, plato) ? creep
   Exit: (12) mentor(socrates, plato) ? creep
   Exit: (11) ancestor(socrates, aristotle) ? creep
   Call: (11) mentor(_12672, socrates) ? creep
   Fail: (11) mentor(_12672, socrates) ? creep
   Redo: (12) ancestor(_25600, aristotle) ? creep
   Call: (13) ancestor(_33142, aristotle) ? creep
   Call: (14) mentor(_33142, aristotle) ? creep
   Exit: (14) mentor(plato, aristotle) ? creep
   Exit: (13) ancestor(plato, aristotle) ? creep
   Call: (13) mentor(_25600, plato) ? creep
   Exit: (13) mentor(socrates, plato) ? creep
   Exit: (12) ancestor(socrates, aristotle) ? creep
   Call: (12) mentor(_18222, socrates) ? creep
   Fail: (12) mentor(_18222, socrates) ? creep
   Redo: (13) ancestor(_33142, aristotle) ? creep
```

Recursion

- Swap ancestor clauses

```
% facts
human(socrates).
human(plato).
human(aristotle).

mentor(socrates, plato).
mentor(plato, aristotle).

% rules
mortal(X) :- human(X).
ancestor(X, Y) :- ancestor(Z, Y), mentor(X, Z).
ancestor(X, Y) :- mentor(X, Y).
```

```
[trace]  ?- ancestor(X, aristotle).
   Call: (10) ancestor(_11970, aristotle) ? creep
   Call: (11) ancestor(_13160, aristotle) ? creep
   Call: (12) ancestor(_13916, aristotle) ? creep
   Call: (13) ancestor(_14672, aristotle) ? creep
   Call: (14) ancestor(_15428, aristotle) ? creep
   Call: (15) ancestor(_16184, aristotle) ? creep
   Call: (16) ancestor(_16940, aristotle) ? creep
   Call: (17) ancestor(_17696, aristotle) ? creep
   Call: (18) ancestor(_18452, aristotle) ? creep
   Call: (19) ancestor(_19208, aristotle) ? creep
   Call: (20) ancestor(_19964, aristotle) ? creep
   Call: (21) ancestor(_20720, aristotle) ? creep
   Call: (22) ancestor(_21476, aristotle) ? creep
   Call: (23) ancestor(_22232, aristotle) ? creep
   Call: (24) ancestor(_22988, aristotle) ? creep
   Call: (25) ancestor(_23744, aristotle) ? creep
   Call: (26) ancestor(_24500, aristotle) ? creep
   Call: (27) ancestor(_25256, aristotle) ? creep
   Call: (28) ancestor(_26012, aristotle) ? creep
   Call: (29) ancestor(_26768, aristotle) ? creep
   Call: (30) ancestor(_27524, aristotle) ? creep
   Call: (31) ancestor(_28280, aristotle) ? creep
   Call: (32) ancestor(_29036, aristotle) ? creep
   Call: (33) ancestor(_29792, aristotle) ? creep
   Call: (34) ancestor(_30548, aristotle) ? creep
   Call: (35) ancestor(_31304, aristotle) ? creep
```

Recursion

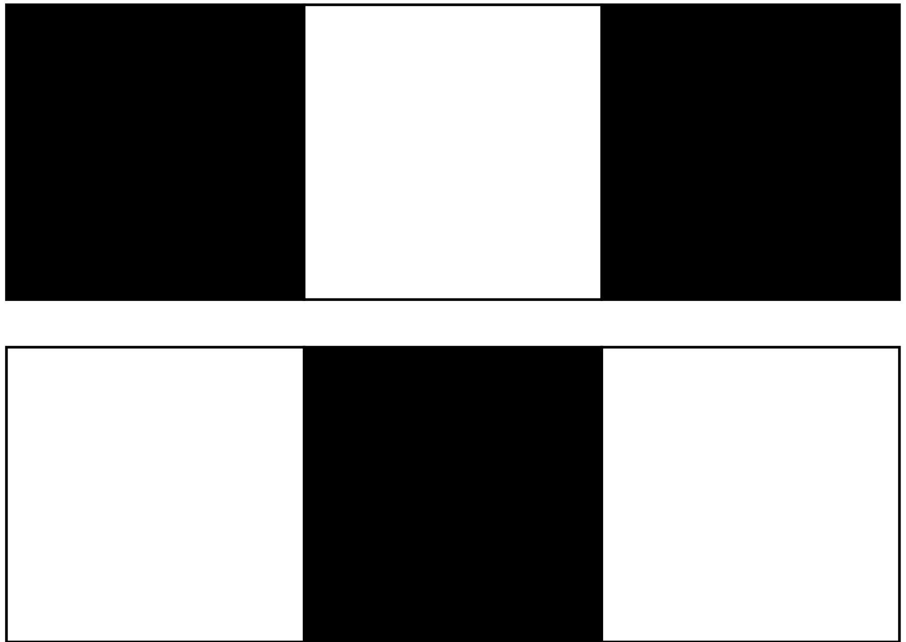
- With recursive predicates, make sure variables are instantiated by earlier atoms in the body

Outline

- Prolog Introduction
- Horn Clauses and SLD Resolution
- Prolog basics
  - Rules and Facts
  - Queries
  - Tracing
  - Negation as failure
  - Recursion
- Examples
  - Map coloring
  - Solving brainteasers
  - Lists in Prolog
  - General map coloring
  - Pathfinding

Simple Map Coloring

- Consider this simple map coloring problem
- How can we use prolog to find a solution?
  - Think about the variables, domains, and constraints



```
color(black).
color(white).

solution(A, B, C) :- color(A), color(B), color(C), \+ A=B, \+ B=C.

print_colors :- solution(A, B, C),
           maplist(write, ['A: ', A, ', B: ', B, ', C: ', C]).
```

```
?- print_colors.
A: black, B: white, C: black
A: white, B: black, C: white
```

Simple CSP Trace Comparison

- Two logically equivalent clauses
- Will their traces be any different?
- Generate and test

```
solution(A,B,C) :- color(A), color(B), color(C), \+ A=B, \+ B=C.

[trace]  ?- print_colors
 Call: (10) print_colors ? creep
   Call: (11) solution(_17174, _17176, _17178) ? creep
   Call: (12) color(_17174) ? creep
   Exit: (12) color(black) ? creep
   Call: (12) color(_17176) ? creep
   Exit: (12) color(black) ? creep
   Call: (12) color(_17178) ? creep
   Exit: (12) color(black) ? creep
   Call: (12) black=black ? creep
   Exit: (12) black=black ? creep
   Redo: (12) color(_17178) ? creep
   Exit: (12) color(white) ? creep
   Call: (12) black=black ? creep
   Exit: (12) black=black ? creep
   Redo: (12) color(_17176) ? creep
   Exit: (12) color(white) ? creep
   Call: (12) color(_17178) ? creep
   Exit: (12) color(black) ? creep
   Call: (12) black=white ? creep
   Fail: (12) black=white ? creep
   Redo: (11) solution(black, white, black) ? creep
   Call: (12) white=black ? creep
   Fail: (12) white=black ? creep
   Redo: (11) solution(black, white, black) ? creep
   Exit: (11) solution(black, white, black) ? creep
```

```
solution(A,B,C) :- color(A), color(B), \+ A=B, color(C), \+ B=C.

[trace]  ?- print_colors.
   Call: (10) print_colors ? creep
   Call: (11) solution(_13794, _13796, _13798) ? creep
   Call: (12) color(_13794) ? creep
   Exit: (12) color(black) ? creep
   Call: (12) color(_13796) ? creep
   Exit: (12) color(black) ? creep
   Call: (12) black=black ? creep
   Exit: (12) black=black ? creep
   Redo: (12) color(_13796) ? creep
   Exit: (12) color(white) ? creep
   Call: (12) black=white ? creep
   Fail: (12) black=white ? creep
   Redo: (11) solution(black, white, _13798) ? creep
   Call: (12) color(_13798) ? creep
   Exit: (12) color(black) ? creep
   Call: (12) white=black ? creep
   Fail: (12) white=black ? creep
   Redo: (11) solution(black, white, black) ? creep
```

Simple CSP Trace Comparison

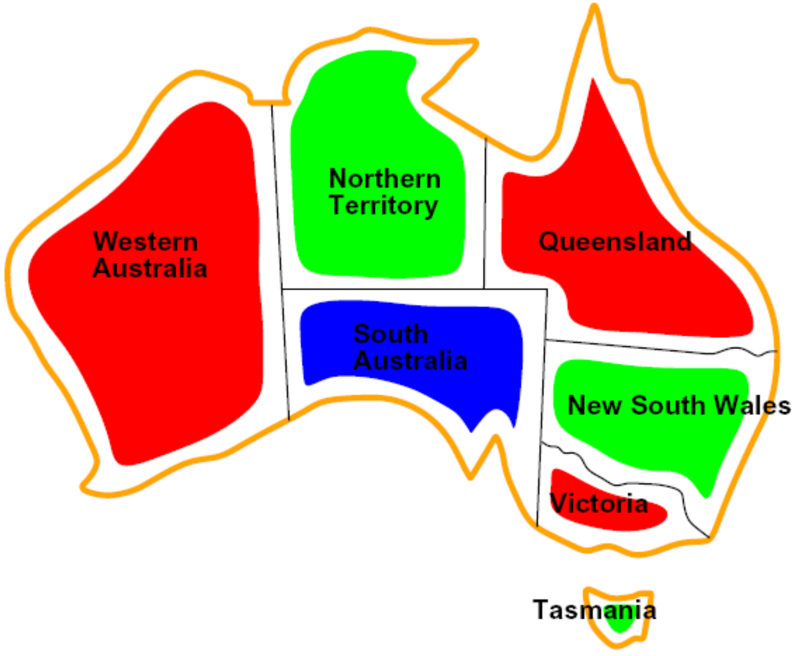
- Two logically equivalent clauses
- Will their traces be any different?

```
solution(A,B,C) :- color(A), color(B), \+ A=B, color(C), \+ B=C.

[trace]  ?- print_colors.
   Call: (10) print_colors ? creep
   Call: (11) solution(_13794, _13796, _13798) ? creep
   Call: (12) color(_13794) ? creep
   Exit: (12) color(black) ? creep
   Call: (12) color(_13796) ? creep
   Exit: (12) color(black) ? creep
   Call: (12) black=black ? creep
   Exit: (12) black=black ? creep
   Redo: (12) color(_13796) ? creep
   Exit: (12) color(white) ? creep
   Call: (12) black=white ? creep
   Fail: (12) black=white ? creep
   Redo: (11) solution(black, white, _13798) ? creep
   Call: (12) color(_13798) ? creep
   Exit: (12) color(black) ? creep
   Call: (12) white=black ? creep
   Fail: (12) white=black ? creep
   Redo: (11) solution(black, white, black) ? creep
```

```
solution(A, B, C) :- color(A), \+ A=B, color(B), color(C), \+ B=C.

[trace]  ?- print_colors.
   Call: (10) print_colors ? creep
   Call: (11) solution(_14968, _14970, _14972) ? creep
   Call: (12) color(_14968) ? creep
   Exit: (12) color(black) ? creep
   Call: (12) black=_14970 ? creep
   Exit: (12) black=black ? creep
   Redo: (12) color(_14968) ? creep
   Exit: (12) color(white) ? creep
   Call: (12) white=_14970 ? creep
   Exit: (12) white=white ? creep
   Fail: (11) solution(_14968, _14970, _14972) ? creep
   Fail: (10) print_colors ? creep
false.
```

CSP From Lecture

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

solution(WA, NT, SA, Q, NSW, V, T) :-
    color(WA), color(NT), \+ WA=NT, color(SA), \+ WA=SA, \+ NT=SA, color(Q), \+ NT=Q, \+ SA=Q,
    color(NSW), \+ Q=NSW, \+ SA=NSW, color(V), \+ SA=V, \+ NSW=V, color(T).


?- print_colors.
WA: red, NT: blue, SA: green, Q: red, NSW: blue, V: red, T: red
WA: red, NT: blue, SA: green, Q: red, NSW: blue, V: red, T: blue
WA: red, NT: blue, SA: green, Q: red, NSW: blue, V: red, T: green
WA: red, NT: green, SA: blue, Q: red, NSW: green, V: red, T: red
WA: red, NT: green, SA: blue, Q: red, NSW: green, V: red, T: blue
WA: red, NT: green, SA: blue, Q: red, NSW: green, V: red, T: green
WA: blue, NT: red, SA: green, Q: blue, NSW: red, V: blue, T: red
WA: blue, NT: red, SA: green, Q: blue, NSW: red, V: blue, T: blue
WA: blue, NT: red, SA: green, Q: blue, NSW: red, V: blue, T: green
WA: blue, NT: green, SA: red, Q: blue, NSW: green, V: blue, T: red
WA: blue, NT: green, SA: red, Q: blue, NSW: green, V: blue, T: blue
WA: blue, NT: green, SA: red, Q: blue, NSW: green, V: blue, T: green
WA: green, NT: red, SA: blue, Q: green, NSW: red, V: green, T: red
WA: green, NT: red, SA: blue, Q: green, NSW: red, V: green, T: blue
WA: green, NT: red, SA: blue, Q: green, NSW: red, V: green, T: green
WA: green, NT: blue, SA: red, Q: green, NSW: blue, V: green, T: red
WA: green, NT: blue, SA: red, Q: green, NSW: blue, V: green, T: blue
WA: green, NT: blue, SA: red, Q: green, NSW: blue, V: green, T: green
```



Solving Riddles Using Prolog

- Sandy, Chris, and Pat all have distinct occupations and play distinct instruments. The occupations are doctor, lawyer and engineer. The instruments are piano, flute, and violin.
  - Chris is married to the doctor
  - The lawyer plays the piano
  - Chris is not the engineer
  - Sandy is a patient of the violinist
- Who plays the flute?
- For Prolog, what are the variables, domains, and constraints?

Solving Riddles Using Prolog

- Variables
  - Doctor, Lawyer, Engineer, Piano, Violin, Flute
- Domains
  - chris, sandy, pat
- Constraints
  - Doctor, Lawyer and Engineer are all different
  - Piano, Violin, and Flute are all different
  - Chris is not the doctor
  - Lawyer=Piano
  - Chris is not the engineer
  - Violin=Doctor
  - Sandy does not play the violin

```
person(chris).
person(sandy).
person(pat).

uniq_people(A, B, C) :- person(A), person(B), person(C),
           \+ A=B, \+ A=C, \+ B=C.

solution(Flute) :-
    uniq_people(Doctor, Lawyer, Engineer),
    uniq_people(Piano, Violin, Flute),
    \+ chris = Doctor, Lawyer = Piano, \+ Engineer = chris,
    Violin = Doctor, \+ sandy = Violin.
```

Lists in Prolog

- Lists are sequences of objects and a list itself is an object
  - []
  - [cat1, cat2, cat3]
  - [[cat1, cat2], cat3, cat4]
  - [[]]

Head and Tail of List

- To access the head of a list, one can use the following notation [H|T]
- This is very useful for defining predicates recursively

```
name([g,a,r,r,e,t]).
name([s,a,m]).
name([d,o,c]).
name([d,a,r,r,e,l]).
name([j,a,m,e,s]).

?- name(X).
X = [g, a, r, r, e, t] ;
X = [s, a, m] ;
X = [d, o, c] ;
X = [d, a, r, r, e, l] ;
X = [j, a, m, e, s].

?- name([H|T]).
H = g,
T = [a, r, r, e, t] ;
H = s,
T = [a, m] ;
H = d,
T = [o, c] ;
H = d,
T = [a, r, r, e, l] ;
H = j,
T = [a, m, e, s].

?- name([d|T]).
T = [o, c] ;
T = [a, r, r, e, l].

?- name([d,a|T]).
T = [r, r, e, l].
```

Head and Tail of List

- One can use "_" to denote a variable without a name
- It behaves the similar to other variables, but does not get returned in a query
- Also, all instances of "_" are assumed to be different

```
?- name(Name), Name=[d|_].
Name = [d, o, c] ;
Name = [d, a, r, r, e, l] ;

?- name(Name), Name=[d,a|_].
Name = [d, a, r, r, e, l] ;
```

General Map Coloring

- From http://cs603.cs.ua.edu/lectures/chapter10b-prolog.pdf
- Let's understand, critique, and propose changes

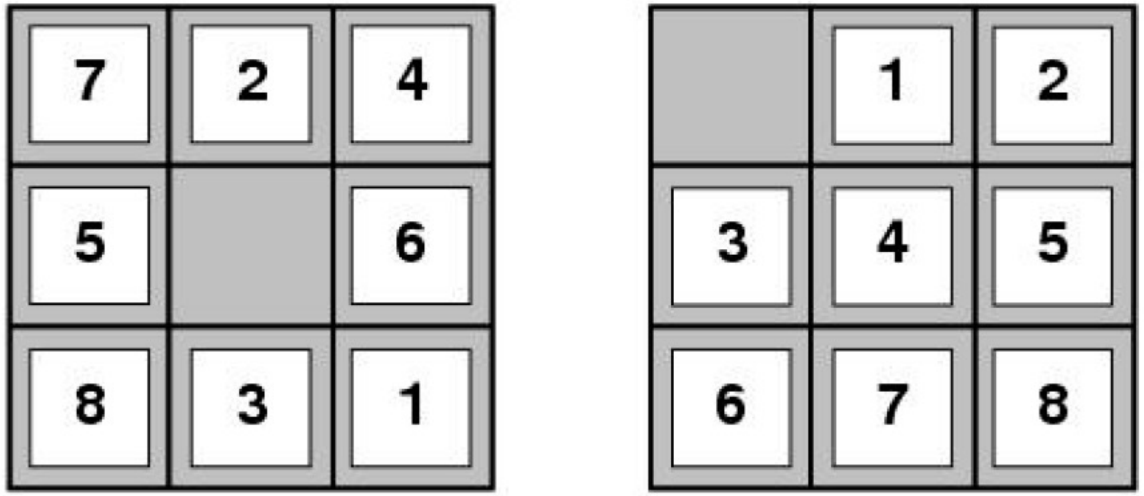
```
?- coloring(M, [adj(wa, [nt, sa]), adj(nt, [wa, sa]), adj(sa, [wa, nt, q, nsw, v]), adj(q, [nt, nsw,
sa]), adj(nsw, [q, sa, v]), adj(v, [sa, nsw]), adj(t, [])]).

M = [assign(wa, yellow), assign(nt, blue), assign(sa, red), assign(q, yellow), assign(nsw, blue),
assign(v, yellow), assign(t, _)] ;
M = [assign(wa, blue), assign(nt, yellow), assign(sa, red), assign(q, blue), assign(nsw, yellow),
assign(v, blue), assign(t, _)] ;
M = [assign(wa, yellow), assign(nt, red), assign(sa, blue), assign(q, yellow), assign(nsw, red),
assign(v, yellow), assign(t, _)] ;
M = [assign(wa, red), assign(nt, yellow), assign(sa, blue), assign(q, red), assign(nsw, yellow),
assign(v, red), assign(t, _)] ;
M = [assign(wa, blue), assign(nt, red), assign(sa, yellow), assign(q, blue), assign(nsw, red), assign(v,
blue), assign(t, _)] ;
M = [assign(wa, red), assign(nt, blue), assign(sa, yellow), assign(q, red), assign(nsw, blue), assign(v,
red), assign(t, _)] ;
```

- assign and adj are compound terms

General Map Coloring

```
different(yellow,blue).
different(blue,yellow).
different(yellow,red).
different(red,yellow).
different(blue,red).
different(red,blue).

lookup(R,[assign(R,C)|_],C).
lookup(R,[_|T],C) :- lookup(R,T,C).

valid(_, [ ]).
valid(M, [adj(_, [ ])|R]) :- valid(M,R).
valid(M, [adj(X, [Y|T])|R]) :- lookup(X,M,Xc), lookup(Y,M,Yc), different(Xc,Yc), valid(M, [adj(X,T)|R]).

assignment([ ],[ ]).
assignment([assign(R,_)|M], [adj(R,_)|T]) :- assignment(M,T).

coloring(M,G) :- assignment(M,G), valid(M,G).
```

Pathfinding



- We want to find a path from one state to another
- How can we do this using Prolog?
  - How do we model states?
  - How do we define actions?
  - How do we search for a sequence of actions to solve the problem?

States

- States are objects
- We can define the 8-puzzle as a list

```
initial_state([2,0,3,1,5,6,4,7,8]).
goal_state([1,2,3,4,5,6,7,8,0]).
```

Actions

- Define legal actions as relationships that hold between states (which are objects)
- A simple strategy is to define every legal move
  - 8-puzzle: move the blank tile left, up, right, or down

```
move( [A,0,C,D,E,F,H,I,J] ,l, [0,A,C,D,E,F,H,I,J] ).
move( [A,B,C,D,0,F,H,I,J] ,l, [A,B,C,0,D,F,H,I,J] ).
move( [A,B,C,D,E,F,H,0,J] ,l, [A,B,C,D,E,F,0,H,J] ).
move( [A,B,0,D,E,F,H,I,J] ,l, [A,0,B,D,E,F,H,I,J] ).
move( [A,B,C,D,E,0,H,I,J] ,l, [A,B,C,D,0,E,H,I,J] ).
move( [A,B,C,D,E,F,H,I,0] ,l, [A,B,C,D,E,F,H,0,I] ).

move( [A,B,C,0,E,F,H,I,J] ,u, [0,B,C,A,E,F,H,I,J] ).
move( [A,B,C,D,0,F,H,I,J] ,u, [A,0,C,D,B,F,H,I,J] ).
move( [A,B,C,D,E,0,H,I,J] ,u, [A,B,0,D,E,C,H,I,J] ).
move( [A,B,C,D,E,F,0,I,J] ,u, [A,B,C,0,E,F,D,I,J] ).
move( [A,B,C,D,E,F,H,0,J] ,u, [A,B,C,D,0,F,H,E,J] ).
move( [A,B,C,D,E,F,H,I,0] ,u, [A,B,C,D,E,0,H,I,F] ).

move( [A,0,C,D,E,F,H,I,J] ,r, [A,C,0,D,E,F,H,I,J] ).
move( [A,B,C,D,0,F,H,I,J] ,r, [A,B,C,D,F,0,H,I,J] ).
move( [A,B,C,D,E,F,H,0,J] ,r, [A,B,C,D,E,F,H,J,0] ).
move( [0,B,C,D,E,F,H,I,J] ,r, [B,0,C,D,E,F,H,I,J] ).
move( [A,B,C,0,E,F,H,I,J] ,r, [A,B,C,E,0,F,H,I,J] ).
move( [A,B,C,D,E,F,0,I,J] ,r, [A,B,C,D,E,F,I,0,J] ).

move( [A,B,C,0,E,F,H,I,J] ,d, [A,B,C,H,E,F,0,I,J] ).
move( [A,B,C,D,0,F,H,I,J] ,d, [A,B,C,D,I,F,H,0,J] ).
move( [A,B,C,D,E,0,H,I,J] ,d, [A,B,C,D,E,J,H,I,0] ).
move( [0,B,C,D,E,F,H,I,J] ,d, [D,B,C,0,E,F,H,I,J] ).
move( [A,0,C,D,E,F,H,I,J] ,d, [A,E,C,D,0,F,H,I,J] ).
move( [A,B,0,D,E,F,H,I,J] ,d, [A,B,F,D,E,0,H,I,J] ).
```

https://www.cpp.edu/~jrfisher/www/prolog_tutorial/5_2.html

Search for a Sequence of Actions

- Use Head|Tail list functionality to find a path to the goal
- However, are there any pitfalls given how Prolog performs SLD resolution?

```
plan(L) :- initial_state(I), goal_state(G), reachable(I,L,G).
reachable(S, [], S).
reachable(S1, [M|L], S3) :- move(S1, M, S2), reachable(S2, L, S3).
```

Iterative Deepening Depth-First Search

- Because Prolog Prolog uses depth-first search to do SLD resolution plan(L) is doing a depth-first search
- Therefore, plan(L) will get stuck in an infinite loop
- First try to find paths of depth 0, then depth 1, etc.
  - `bplan(L) :- tryplan([], L).`
  - `tryplan(L, L) :- plan(L).`
  - `tryplan(X, L) :- tryplan([_|X], L).`
- This is iterative-deepening depth-first search
- Because L is a variable, it can unify with any object, even a list. tryplan([], L) ensures it first tries a depth-limited search with limit 0

```
bplan(L) :- tryplan([], L).

tryplan(L, L) :- plan(L).
tryplan(X, L) :- tryplan([_|X], L).

initial_state([2,0,3,1,5,6,4,7,8]).
goal_state([1,2,3,4,5,6,7,8,0]).
plan(L) :- initial_state(I), goal_state(G), reachable(I,L,G).

reachable(S, [], S).
reachable(S1, [M|L], S3) :- move(S1, M, S2), reachable(S2, L, S3).

?- bplan(L).
L = [l, d, d, r, r]
```

Iterative Deepening Depth-First Search

- What happens if we switch the order of the rules
  - `tryplan(X, L) :- tryplan([_|X], L).`
  - `tryplan(L, L) :- plan(L).`
- Because Prolog tries rules in the order they appear in the file, this will lead to us getting stuck in an infinite loop

Limitations

- Given the theoretical properties of uninformed search strategies, we know the time complexity increases exponentially with the depth of the solution
- Therefore, this cannot scale up to larger puzzles, such as the 15 or 24 puzzle
- However, we can try to solve problems with a larger state space by using sub-goals

A* Search

- One can also define A* search declaratively in Prolog
  - https://www.cse.sc.edu/~mgv/csce580sp17/index.html
- Variants of A* search, such as IDA* can significantly reduce memory usage

Summary

- Prolog
  - A declarative language that must sometimes be viewed procedurally
  - Describe the solution, not what to do to find the solution
- SLD Resolution
- Facts, Rules, and Queries
- Negation as Failure
  - Caveats
- Recursion
  - Caveats
- Generate and test
  - Test sooner to prune earlier
  - Make sure all variables involved in the test are instantiated