Constraint Satisfaction Problems

So we've been talking about classical search in something of a "black-box" fashion, meaning we have knowledge about states, transitions, and tests for goal states, but...

...our search algorithms have treated states as atomic; there are no internal variables for us to wiggle and try to fix something that's wrong with a state and try to make it right.

á **Constraint satisfaction problems (CSPs)** use general purpose heuristic algorithms to try to find a solution using a set of variables and constraints.

What does this mean?

With CSPs, we define our states as consisting of a variety of variables that we then individually check against a variety of constraints.

If our variable **assignments** in a given state are all **consistent** with our constraints, then we say that we've found a solution!

á A CSP **variable (X)** is defined in terms of its **domains (D)**, such that a variable X_i can only attain a value in D_i in any given state.

á A CSP **constaint (C)** is a Boolean condition for success on some variable instantiation.

Therefore, we can phrase all of our constraint satisfaction problems as consisting of:

- X: a set of variables {X_1, X_2, ..., X_n}
- D: a set of domains for those variables {D_1, D_2, ..., D_n} (one for each variable)
- C: a set of Boolean constraints that tell us when we've found a solution with our variables

ň NOTE: We've only found a solution whenever all of our constraints are satisfied, and every variable has been assigned a value from its respective domains!

Later, however, we'll look at how knowing how many constraints we've failed can steer us in the right direction to a solution...

This gives us a comparison between classical search and CSPs:

Classical Search vs. CSP

Property	**Classical Search**	**CSP**
States	Problem-specific and atomic; cannot be divided into constituent components.	Variables, Domains, and Constraints
Success	Test a state to see if it meets our goal condition.	Assign values to all variables and see if that assignment meets all constraints
Cares about...	...the path from an initial state to a goal state	...an instantiation of variables that meets all constraints; path is irrelevant!
Applications	Pathfinding, optimization, etc.	SAT(isfiability), map coloring, scheduling, etc.

So, the motto of CSP might be, "I don't know what I'm looking for, but I'll know it when I see it! ...oh and I don't care about how I got there."

Let's look at some simple examples using numerical variables and algebraic constriants.

Example

- Consider the following constraint satisfaction problem specifications and find a solution for each.

```
X = {A, B}
D = { {1, 2}, {0, -1} }

#1
C = { A < B; A * B >= 0 }

#2
C = { A > B; A * B >= 0 }

#3
C = { A > 0 }



OK... Now that we're done with the warmups...

#4
X = {A, B, C, D, E, F, G, H, I, J, K, L, M, N}
D = {
        {1, 2, 3}, {6, 7, 22}, {2, 9, 1}, {9, 1, 2},
        {taco, 2, 1}, {1, 2, huh?}, {"what is this", ":D"},
        {left, up, down, right}, {2B, !2B}, {-_-, 0_o, 42},
        {4th, 5th, 999th}, {blue pill, red pill},
        {Cat, Dog, Catdog}, {Q, T, PI}
    }
C = {
        A < (B + C * D);
        B != C + D (string concatenation);
        E = {taco if A = 1, 1 otherwise};
        N = {Q if M = cat, T if I = 2B};
        L = {blue pill IFF all other variables = 1st in their set}
    }
```

...OK so I had a little fun with that last example...

The point being, of course, that what might seem like a simple task of assigning values to variables and checking constraints can quickly become difficult with a lot of variables and constraints.

Additionally, we see that not all variable domains need be the same, or even the same type / format.

So how do we tackle this problem? How do we design intelligent agents to do this?

Formulating CSP as Search

"Oh... we're back to search, Andrew?"

Yes, stop complaining!

As it turns out, we can formulate our CSPs as search problems (using a search tree) using the following tactic:

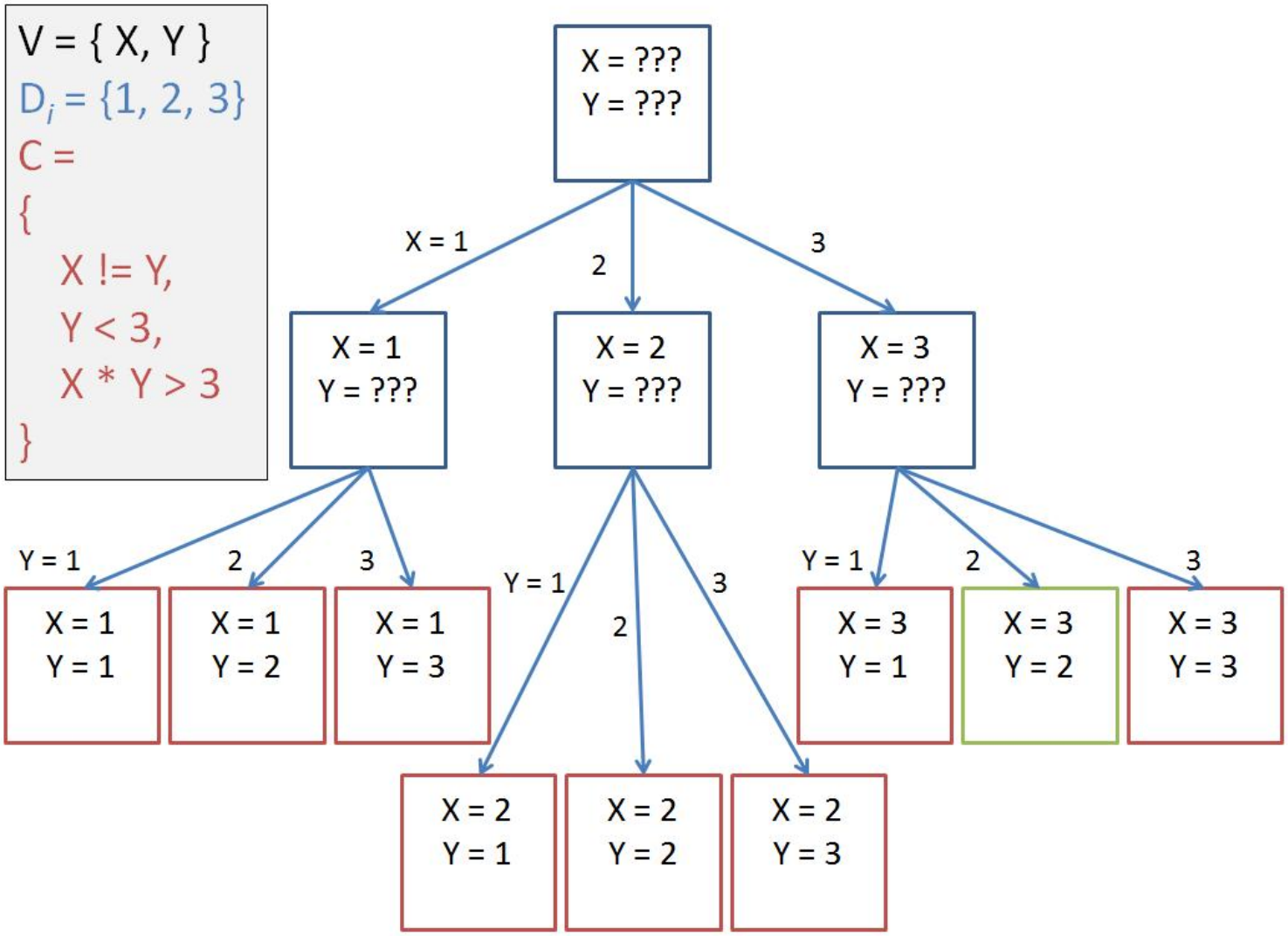
- A CSP Search Tree **state** is either a partial or complete instantiation of variables.

- A CSP Search Tree **leaf state** is therefore a complete instantiation.

- The **initial state** is therefore "no assignment" and each **action** assigns one variable to a value.

- A **goal state** is then any complete assignment such that all constraints are satisfied.

So, let's return to our simple example using numerical variables and algebraic constraints:



à What is the depth of this type of CSP tree?

Constraint Propagation

As we've seen with CSP search, the more possible variable assignments in our search tree, the larger it becomes and longer it takes to search.

So, in order to improve our efficiency, we attempt to reduce the domains of our variables to those assignment combinations that are possible.

For example, if one of our variables had the domain of integers from 1 to 10, and a constraint said that variable couldn't be 5, then it would be wasteful to keep 5 within the domain!

á **Constraint propagation** provides a means of reducing our variable domains by considering the constraints that a domain element has the possibility of satisfying. If a domain element has *no* possibility of satisfying a constraint, we simply prune it from the domain!

á **Node consistency** determines if any unary constraints on a variable are met by each element of that variable's domain.

```
; In the following example, A & B are Node Consistent:
X = { A, B }
D = { {1, 2}, {3, 4} }
C = {
       A != 3
       B != 5
     }

; In the following example, A & B are NOT Node Consistent:
X = { A, B }
D = { {1, 2}, {3, 4} }
C = {
       A != 2
       B != 3
     }

; To resolve the above node inconsistency, we
; amend the domains of A and B to be:
X = { A, B }
D = { {1}, {3} }
C = {
       A != 2
       B != 3
     }
```

á **Arc consistency** determines if any binary constraints between two variables can be met by all elements in their domains.

In other words, we want to verify that for two variables X and Y that both appear within a constraint (e.g. X < Y), all values in the domain of X can satisfy the constraint if at least one value in the domain of Y can be paired with that value in the domain of X.

```
; In the following example, A & B are Arc Consistent:
X = { A, B }
D = { {1, 2}, {3, 4} }
C = {
      A < B
      B >= 2 * A
    }

; In the following example, A & B are NOT Arc Consistent:
X = { A, B }
D = { {1, 2}, {3, 4} }
C = {
      A < B
      B > 2 * A
    }

; To resolve the above arc inconsistency, we
; amend the domains of A and B to be:
X = { A, B }
D = { {1}, {3, 4} }
C = {
      A < B
      B > 2 * A
    }
```

To automate arc consistency fixes, we can use an algorithm called AC-3. (page 209 of your book)

To picture how AC-3 works, you can imagine our CSP as a graph with nodes as the variables, and edges between any two variables that are found in the same constraint.

Let's see an example CSP and arc consistency graph.

$V = \{ W, X, Y \}$

$D_i = \{0, 1, 2\}$

$C =$

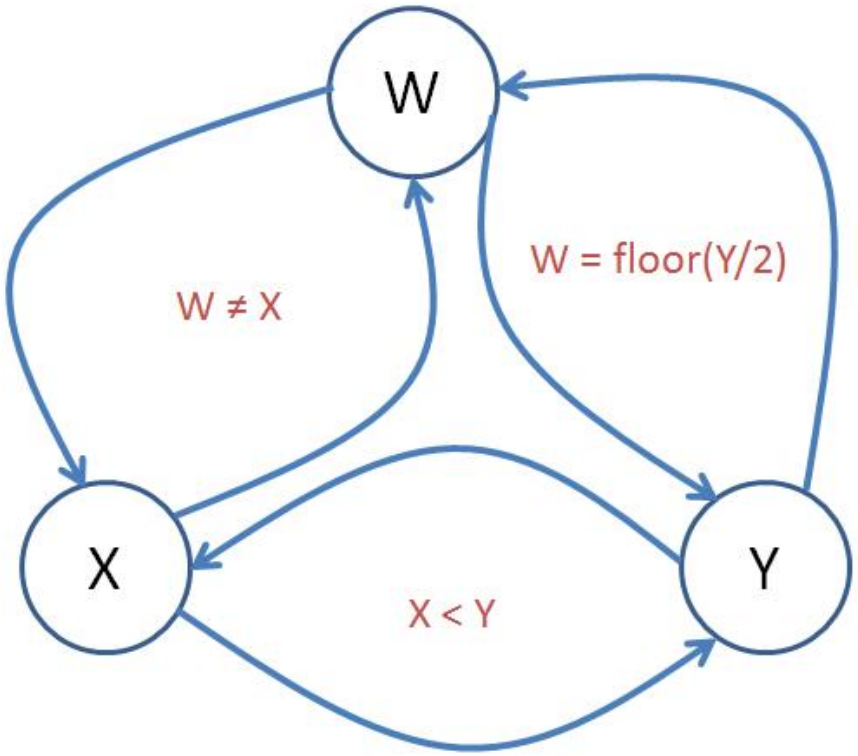
$\{$

$W \neq X$

$W = floor(Y/2)$

$X < Y$

$\}$



Now, we see that our arcs happen to involve binary constraints between all three of our variables, so we want to assess each of these arcs and determine whether or not there are elements to remove from any of the domains.

Each arc can be considered a tuple (A, B) where A is the node/variable from which an arc originates and B is the terminal node.

This gives us 6 arcs in our problem: (W, X), (X, W), (W, Y), (Y, W), (X, Y), and (Y, X)

We begin AC-3 with all of our arcs to be assessed, and proceed as follows:

```
  ; Reduced AC-3 Pseudocode
  enqueue all arcs in your CSP
  while that queue is not empty
      check if current arc (A, B) is consistent
      ; i.e., there is at least 1 variable in B satisfying
      ; the constraint for every variable in A

      if a value in the domain of A is unsatisfiable, remove that val
      if any value was removed from A then do:
          if the domain of A is reduced to ∅, return false
          enque all arcs from C_i to A (C_i, A) where C_i = all neighbor nodes of X ex
cept B

  return true (if your queue is empty and you've made all domains arc consistent)
```

Let's try it with our example!

```
Vars: {W, X, Y}
Domains (for each): {0, 1, 2}
Constraints:
{
  W != X,
  W = floor(Y/2),
  X < Y
}

; Step 1: Queue up those arcs!
Q = { (W, X), (X, W), (W, Y), (Y, W), (X, Y), (Y, X) }

; ------------------------------------------------
; While Q is not empty...
; Step 2: Choose an arc from the queue

chosenArc = (W, X)
DomainW = {0, 1, 2}
DomainX = {0, 1, 2}
constraint = W != X

; Arc is consistent! Remove this arc from queue!

; ------------------------------------------------
Q = { (X, W), (W, Y), (Y, W), (X, Y), (Y, X) }

chosenArc = (X, W)
DomainW = {0, 1, 2}
DomainX = {0, 1, 2}
constraint = W != X

; Arc is consistent! Remove this arc from queue!

; ------------------------------------------------
Q = { (W, Y), (Y, W), (X, Y), (Y, X) }

chosenArc = (W, Y)
DomainW = {0, 1, 2}
DomainY = {0, 1, 2}
constraint = W = floor(Y/2)

; Arc is inconsistent! W can never be 2, so prune it,
; and add neighbors of W (X, W) to queue
DomainW = {0, 1}
```

```
; ---------------------------------------------------
Q = { (Y, W), (X, Y), (Y, X), (X, W) }

chosenArc = (W, Y)
DomainW = {0, 1}
DomainY = {0, 1, 2}
constraint = W = floor(Y/2)

; Arc is consistent! Remove this arc from queue!

; ---------------------------------------------------
Q = { (X, Y), (Y, X), (X, W) }

chosenArc = (X, Y)
DomainX = {0, 1, 2}
DomainY = {0, 1, 2}
constraint = X < Y

; Arc is inconsistent! X can never be 2, so prune it,
; and add neighbors of X (W, X) to queue
DomainX = {0, 1}

; ---------------------------------------------------
Q = { (Y, X), (X, W), (W, X) }

chosenArc = (Y, X)
DomainX = {0, 1}
DomainY = {0, 1, 2}
constraint = X < Y

; Arc is inconsistent! Y can never be 0, so prune it,
; and add neighbors of Y (W, Y) to queue
DomainY = {1, 2}

; ---------------------------------------------------
Q = { (X, W), (W, X), (W, Y) }

chosenArc = (X, W)
DomainX = {0, 1}
DomainW = {0, 1}
constraint = W != X

; Arc is consistent! Remove this arc from queue!

; ---------------------------------------------------
Q = { (W, X), (W, Y) }
```

```
chosenArc = (W, X)
DomainX = {0, 1}
DomainW = {0, 1}
constraint = W != X

; Arc is consistent! Remove this arc from queue!

; -----------------------------------------------------
Q = { (W, Y) }

chosenArc = (W, Y)
DomainW = {0, 1}
DomainY = {1, 2}
constraint = W = floor(Y/2)

; Arc is consistent! Remove this arc from queue!

; -----------------------------------------------------
Q = { }

; Queue is empty! We're done, return true!
; Our domains were therefore modified to:
DomainW = {0, 1}
DomainX = {0, 1}
DomainX = {1, 2}

; Reducing the size of our search space substantively!
```

Map Coloring Problem

Perhaps the most widely used example for CSPs is the map coloring problem.

á The **map coloring problem** asks if, given a map consisting of a set of states, with some adjacent to others, can you assign one of N colors to each state such that no two adjacent states have the same color? (NB: here state means the type you'd find in a country, like California)

Some map coloring solvers will go a step further and ask: "If so, what is the minimum number of colors required to complete this task?"

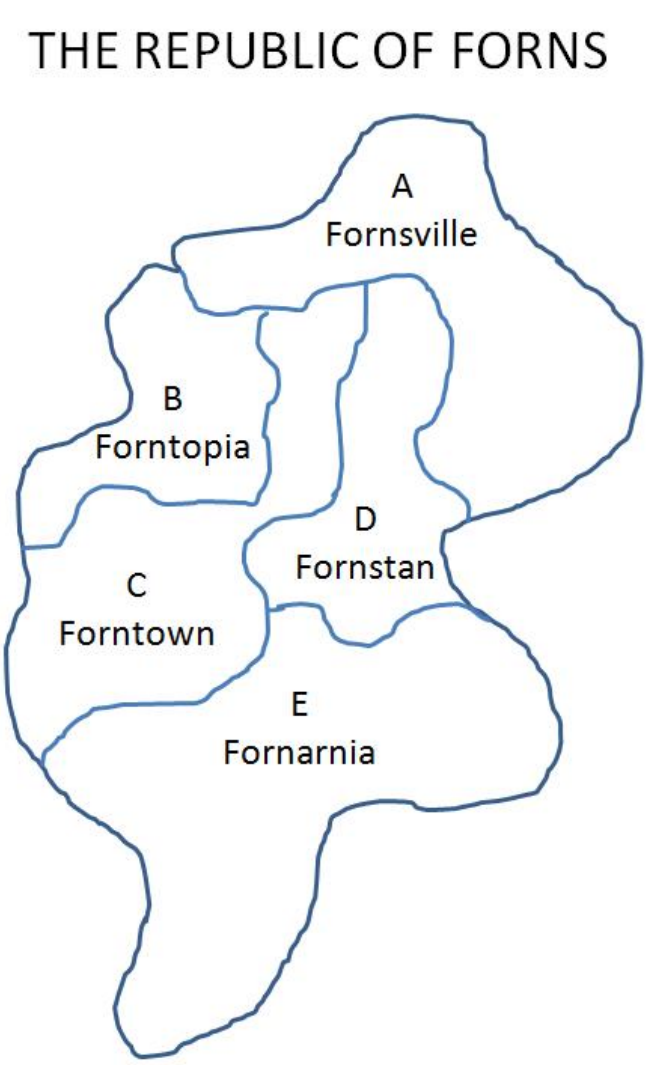
Let's formulate this as a constraint satisfaction problem!

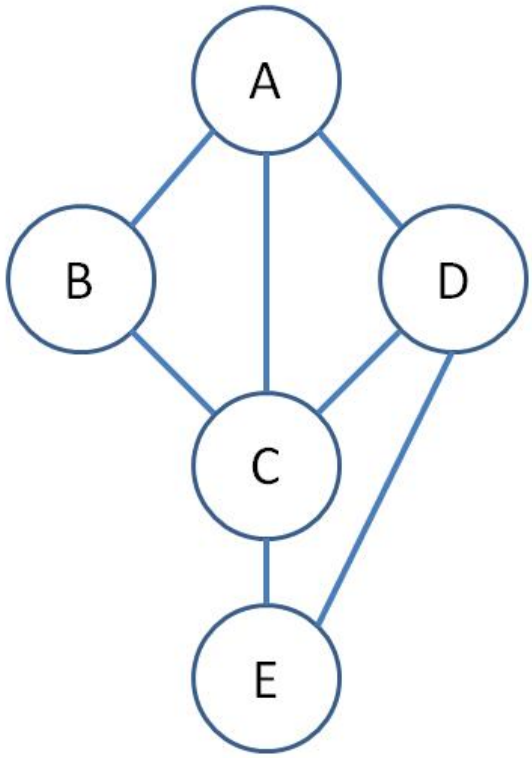
- What will a state (here, state means the state in our search tree) look like in our map coloring CSP?

- What will our constraints look like?

Example

- What is the minimum N (where N is the number of colors) you could use to color the great Republic of Forns, below?





Constraints:

A != B, A != C, A != D,
B != C
C != D, C != E
D != E

Example

- Sketch a search tree for the CSP above, given the minimum N you decided for the map coloring.

- What do we notice about CSP node expansion that will save us some work?

CSP as Local Search

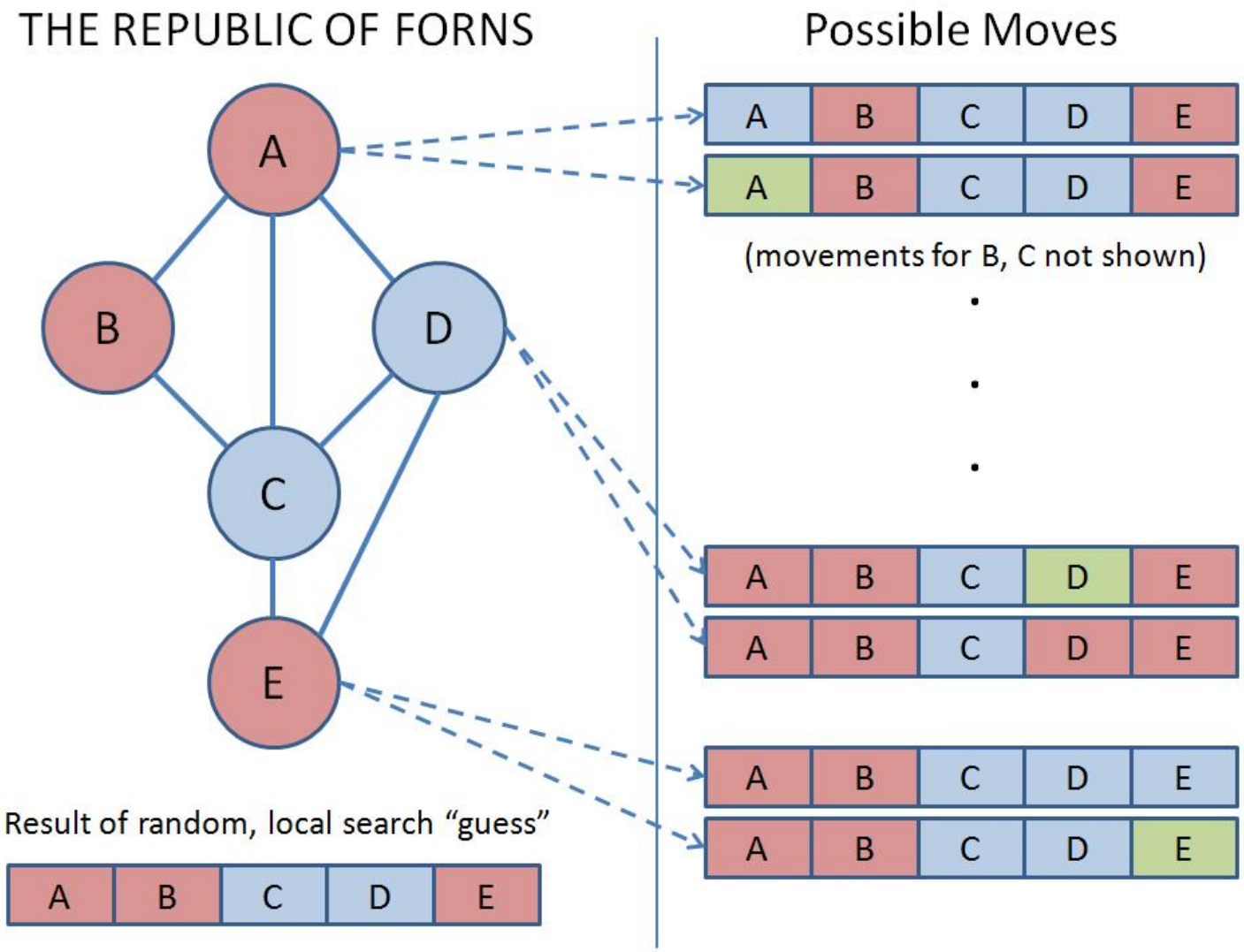
Now... what would happen if the proud people of The Republic of Forns all declared independence and separated the states into millions of sovereign bodies?

Would our CSP search tree for the map coloring problem be tractable?

Errr... no... millions of variables? Not gonna work...

á **CSP Local Search** operates under the same principles of classical local search: start off with a random complete instantiation, and then try to tweak erroneous assignments into a constraint-consistent solution

So, just as an example using our 5-state map from before, we could arrive at the following partial instantiation for, say, N = 3 colors:



à How should we choose which, amongst these adjacent possible solutions, to move to next?

à What were some of the improvements we talked about with classical local search that might apply to CSP local search?

So far, with our search strategies, we've begun our problems under the assumption that we don't know anything about a possible solution...

In classical search, all we could do was guess with some amount of confidence where in the search tree a solution was likely to be found relative to our initial state...

With CSPs, we started with blank variable instantiations and tried to find one that satisfied all of our constraints.

What if we knew some evidence that could direct our search in a more intelligent manner? (FOREBODING!!!)

Propositional Logic

> "Humans, it seems, know things; and what they know helps them do things."
>
> Our Textbook's Start to Chapter 7

Some humans really know how to start a textbook chapter!

What the quote means to express is that human cognition has two important qualities that would be nice to recreate:

- Humans have some sort of **knowledge** representation that spans the breadth of human experience from episodic (event-related) to semantic (factual) memory.
- Using this knowledge, humans rationalize and perform **inference** from the facts that they know and extrapolate unto the facts that they don't.

For example, if I told you that "If it is raining, then the pavement will be wet. Oh, by the way, it's raining." What might you infer about the pavement?

Hopefully, you infer that the pavement is wet! (and that I'm bad at concisely relaying information)

This is a small gap that humans bridge very easily using reasoning about what they know and what they can infer from what they know.

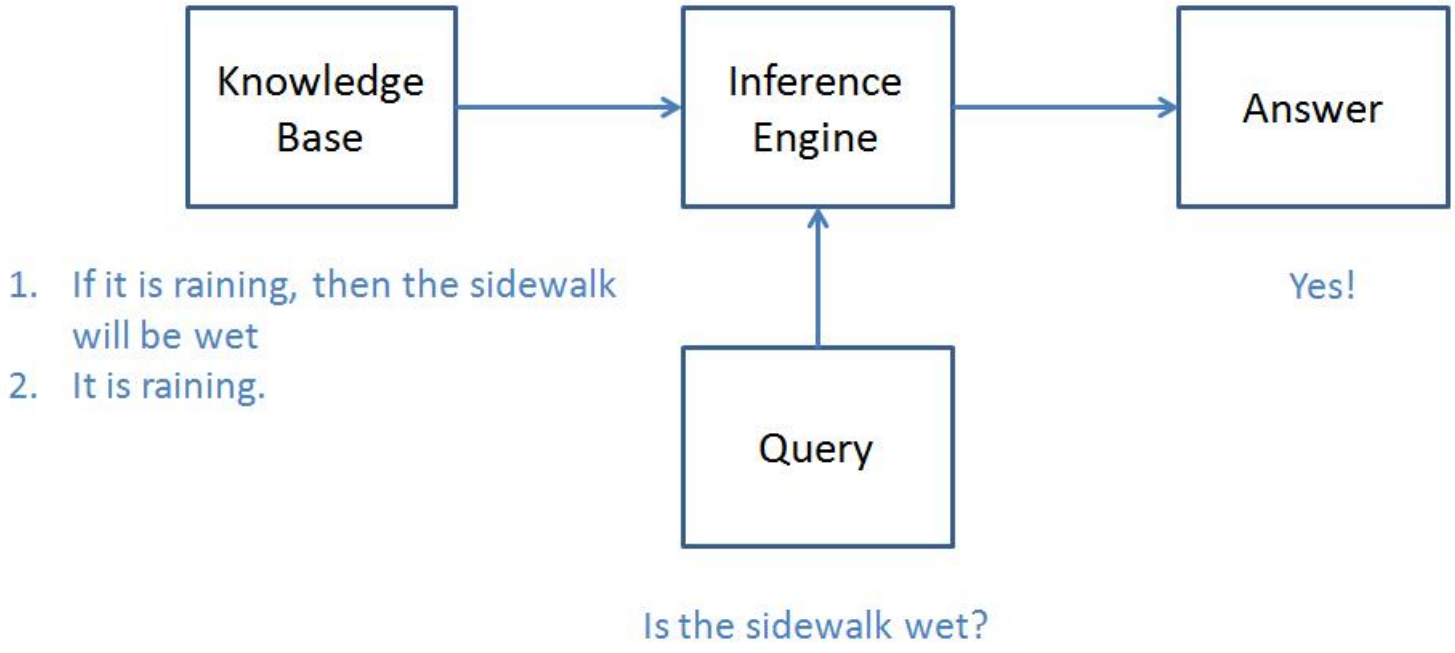
Computers, however, need a lot of help with this task; to start our quest to implement a reasoning engine, we'll talk about some definitions:

- An intelligent system's **knowledge base (KB)** consists of a set of logical **sentences** that represent some assertions of the world.

- A KB's **sentences** are not English sentences so much as they are logical expressions relating some properties of the environment.

In **Propositional Logic**, a KB's sentences are combinations of boolean variables that express our knowledge about the world.

So pictorially, our task will be to design an inference engine that takes what we know, questions about what we know, and produce an answer:



Above, we say that the items that we start off with in our KB are axioms, and the facts that we derived from them are inferred.

á **Axioms** are the "given" elements of the KB that we assume are true before reasoning; they represent our background knowledge.

á **Inferred** terms are derived from those initial axioms to extend our knowledge base and answer queries.

à What are the axioms and derived terms from our example above about whether the sidwalk is wet when it's raining?

So, we begin to see how to fit the pieces together for our inference engine, but we need a formalization.

Propositional Syntax

á In propositional logic, we'll represent our knowledge with Boolean variables called **propositions**, i.e., variables that can either be true or false.

```
Boolean variable examples:

Let S = whether or not Andrew's socks are matching today
    H = whether or not it is over 80 degrees out today
    R = whether or not it is raining
    W = whether or not the sidewalk is wet
```

Thus, every proposition can attain either the value True or False, or T and F as you'll often see them displayed.

Now, we can build logical sentences comprised of our propositions and logical connectives.

á A **logical operator** simply defines some relationship between some number of propositions.

á Our knowledge representation is therefore composed of:

- **Atoms:** T, F, Prop, ¬Prop, where Prop is some propositional variable.
- **Sentences:** atoms or atoms used with logical operators: ¬, ∨, ∧, ⇒

Logical operators are defined as follows:

Negation (¬) is a unary logical operator that flips the Boolean value of a sentence from T to F or from F to T.

```
; Definition:
¬(sentence)

; Examples:
¬T = F
¬F = T

; If Prop = T, then:
¬Prop = F
```

DeMorgan's Law (although not strictly a logical operator) is an important aspect of negating non-atom sentences (defined below).

```
; Definition:
¬(prop1 ∧ prop2) = (¬prop1) ∨ (¬prop2)
¬(prop1 ∨ prop2) = (¬prop1) ∧ (¬prop2)

; Examples:
¬(T ∧ F)
  = (¬T) ∨ (¬F)
  = (F) ∨ (T)
  = T
```

Conjunction (∧) AKA "and" is a binary operator that returns T only if the sentence on its left AND right are T.

```
; Definition:
(sentence1) ∧ (sentence2)

; Examples:
T ∧ F = F
T ∧ T = T
(T ∧ F) ∧ T = F
Prop1 ∧ Prop2 ∧ Prop3
```

Disjunction (∨) AKA "or" is a binary operator that returns T if at least one of the sentences on its left OR right is T.

```
; Definition:
(sentence1) ∨ (sentence2)

; Examples:
T ∨ F = T
T ∨ T = T
(T ∨ F) ∨ F = T
F ∨ F = F
Prop1 ∨ Prop2 ∨ Prop3
```

Implication (⇒) is a logical shorthand indicating an if-then relationship between sentences.

```
; Definition:
(sentence1) ⇒ (sentence2)
; ...is shorthand for:
¬(sentence1) ∨ (sentence2)

; Examples:
F ⇒ F
  = ¬F ∨ F
  = T ∨ F
  = T
```

If-and-only-if (⇔) is a logical shorthand indicating if-then relationships of the format: (if A then B) AND (if B then A).

```
; Definition:
(sentence1) ⇔ (sentence2)
; ...is shorthand for:
((sentence1) ⇒ (sentence2)) ∧ ((sentence2) ⇒ (sentence1))
```

Alright, so now that we have the tools at our disposal, we can talk about the semantics of propositional logic.

Propositional Semantics

Since we're dealing with propositions, which are Boolean variables, then we can imagine our variables describing a scenario by being instantiated to either T or F.

Let's say we returned to the example of rain and the sidewalk. I'll define two propositions:

```
Let R = whether or not it's raining
    S = whether or not the sidewalk is wet
```

The fact that I have two propositional variables of interest means that I have 4 possible worlds consisting of:

R	**S**	**Interpretation of (R ∧ S)**
F	F	It isn't raining and the sidewalk isn't wet
F	T	It isn't raining but the sidewalk IS wet
T	F	It IS raining, but the sidewalk isn't wet
T	T	It IS raining, and the sidewalk IS wet

à In general, then, if I have N propositional variables, how many possible worlds will I have?

Now, let's say I observed some facts about my environment and also knew some background knowledge that are relevant to my observations.

Example

t Convert the following English sentences into propositional logic sentences:

```
Let R = whether or not it's raining
    S = whether or not the sidewalk is wet

;; #1
"If it is raining, then the sidewalk will be wet."

;; #2
"It is raining."

;; #3
"It is NOT raining."
```

OK, so let's say that I observe the following: "If it is raining, then the sidewalk will be wet. It is raining."

à How can I phrase this observation as a conjunctive sentence?

Knowing these facts about my environment, and also the set of all 4 possible worlds over instantiation combinations of R and S, I can ask if a world is consistent with my observations.

á Formally, we say that for some logical sentence α, a world is a **model** of that sentence if α is true given the instantiations of variables in that world.

H We use the notation M(α) to indicate the set of all models of α; this is referred to as the **meaning** of α

So, for our example, if α = (¬R ∨ S) ∧ R = "If it is raining, then the sidewalk will be wet. It is raining."

...then M(α) = the set of all worlds consistent with (¬R ∨ S) ∧ R

Let's look again at our truth table:

World	R	S	α = (¬R ∨ S) ∧ R	Model of α?
W1	F	F	(¬F ∨ F) ∧ F = F	No
W2	F	T	(¬F ∨ T) ∧ F = F	No
W3	T	F	(¬T ∨ F) ∧ T = F	No
W4	T	T	(¬T ∨ T) ∧ T = T	Yes!

So, since W4 is the only model of our sentence α, then we say M(α) = { W4 }

We can derive a few interesting set properties of models based on some atoms and sentence formats:

First, some notational specifications:

- **W** (uppercase, bolded) is the set of all worlds.
- w (lowercase, unbolded) is one world in **W**.
- The notation w.Prop for some given propositional variable Prop is the instantiation of Prop in world w.

For atoms, we have the following properties:

- M(True) = **W**
- M(False) = ∅
- M(Prop) = { w ⊆ **W** | w.Prop = T }
  "The set of all worlds in W such that each world w in the set has instantiation Prop = T"
- M(¬Prop) = { w ⊆ **W** | w.Prop = F }
  "The set of all worlds in W such that each world w in the set has instantiation Prop = F"

For sentences, we have the following properties:

- **Intersection:** M(α ∧ β) = M(α) ∩ M(β)
- **Union:** M(α ∨ β) = M(α) ∪ M(β)
- **Subtraction:** M(¬α) = **W** \ M(α)

There a couple of accompanying definitions with propositional models where we interpret M(α) as the *meaning* of α

á If M(α) = **W**, then we say that α is **valid**.

Example
t Are the following sentences over worlds for variables S and R valid? If not, give an instantiation that is inconsistent with α

```
;; #1
α = S ∨ ¬S

;; #2
α = S ∧ ¬S

;; #3
α = S ∨ (¬S ∧ R)
```

á If M(α) = ∅, then we say that α is **inconsistent**.

Example
t Are the following sentences over worlds for variables S and R inconsistent? If not, give an instantiation that is consistent with α

```
;; #1
α = S ∨ ¬S

;; #2
α = S ∧ ¬S

;; #3
α = S ∨ (¬S ∧ R)
```

á If M(α) ≠ ∅, then we say that α is **consistent**.

Example

t Are the following sentences over worlds for variables S and R consistent?

```
;; #1
α = S ∨ ¬S

;; #2
α = S ∧ ¬S

;; #3
α = S ∨ (¬S ∧ R)
```

We also have some sentence properties over pairs of sentences α and β:

á If M(α) ⊆ M(β), then we say that α **entails** β, written α ⊨ β.

Another way of thinking about this is that all of the worlds consistent with α are also consistent with β such that we know whenever some world w ⊨ α, then that same world w ⊨ β.

That said, we are NOT guaranteed that every world consistent with β is consistent with α

Example

t Does α entail β in the following examples? If not, provide a world where α is consistent but β isn't.

```
;; #1
α = S
β = R

;; #2
α = ¬R ∧ (S ∨ R)
β = S
```

á If M(α) ⊆ M(β) AND M(β) ⊆ M(α), then we say that α and β are **equivalent**

Another way of thinking about this is that all of the worlds consistent with α are also consistent with β AND all worlds consistent with β are also consistent with α

Example
t Is α = β in the following sentences?

```
;; #1
α = S
β = R

;; #2
α = ¬R ∧ (S ∨ R)
β = S

;; #3
α = ¬R ∧ (S ∨ R)
β = ¬R ∧ S
```

Formalizing the Knowledge Base

OK, so we've got our propositional logic syntax and semantics down... let's apply them to an intelligent reasoning system!

Our quest begins by defining a knowledge base:

á A **propositional knowledge base (KB)** is a conjunction of sentences that entails a set of possible worlds.

In other words, if M(KB) = **W'**, then **W'** ⊆ **W**

To illustrate this, let's expand on our previous example to include a new variable, L = "The light from the sun is shining."

Example
t Convert the sentences about variables S, L, and R into propositional sentences, and then form the KB:

```
Let R = whether or not it's raining
    S = whether or not the sidewalk is wet
    L = whether or not the sunlight is shining

;; #1
sent1 = "If the sunlight is shining, then it is not raining."

;; #2
sent2 = "If it is not raining, then the sidewalk is dry."

;; #3
sent3 = "The sunlight is shining."

;; #4:
KB = The conjunction of all of our sentences
   = sent1 ∧ sent2 ∧ sent3
```

Inference Rules

Think about what "inference" means... that is, what it means when you infer something?

It means you take the information you know, and then make an assertion based on that information to what you haven't observed.

In the context of a knowledge base, inference rules define semantics to take the sentences of a KB and then return a NEW sentence that is entailed by the KB!

The first inference rule is called Modus Ponens:

á **Modus Ponens** is an inference rule defined as:

$$\alpha \models \beta;\ \alpha$$

$$\beta$$

Frankly, I find this syntactic definition to be one of the worst known to man; let's unpack it:

Modus Ponens says, "If α entails β in my KB, and I also have α somewhere in my KB (by itself), then I'm allowed to add β (by itself) to the knowledge base."

Another, more friendly formulation is that if you ever have two sentence structures in your KB: (sent1 ⇒ sent2) ∧ sent1, then you can add sent3 = sent2

So, for example, if I have the following sentences, sent1 and sent2 in my KB, I can use modus ponens to infer sent3:

```
;; Assume sent1 and sent2 are in KB

;; #1
sent1 = (R ⇒ S)
sent2 = R

; Since, in our definition for Modus Ponens,
; α = R and β = S, then I can infer:
sent3 = S


;; #2
sent1 = X
sent2 = (Y ∨ ¬X)

sent3 = ???
```

The second inference rule is called Resolution:

á **Resolution** is an inference rule defined as:

For any two sentences X and Y in our KB with propositions x_i and y_i of the format: X = x1 ∨ x2 ∨ ... ∨ xn; Y = y1 ∨ y2 ∨ ... ∨ yn

If X and Y contain some propositional variable Z such that Z ∈ X AND ¬Z ∈ Y, then we can combine the two sentences on everything except the disagreement on Z such that:

$$x1 \vee x2 \vee Z \vee ... \vee xn; \qquad y1 \vee y2 \vee \neg Z \vee ... \vee yn$$

$$x1 \vee x2 \vee ... \vee xn \vee y1 \vee y2 \vee ... \vee yn$$

In other words, we're allowed to infer the sentence composed of all disjoined propositions of X and Y *except* for element Z.

Given sentences sent1 and sent2 in my KB, I can use resolution to infer sent3:

```
;; Assume sent1 and sent2 are in KB

;; #1
sent1 = (X ∨ Y ∨ ¬Z)
sent2 = (M ∨ N ∨ Z)

; Since sent1 and sent2 disagree on proposition Z
; we can resolve the two into the third sentence:
sent3 = (X ∨ Y ∨ M ∨ N)

;; #2
sent1 = (X ∨ Y ∨ ¬Z)
sent2 = Z
sent3 = (X ∨ Y)

;; #3
sent1 = ((X ⇒ Y) ∨ Z)
sent2 = (¬X ⇒ Y)
sent3 = ???
```

There are a couple things to note about resolution:

á Resolution expects our sentences to be in the form of **clauses**, which are sentences composed only of disjunction between propositions.

Example

t Are the following sentences clauses? Can they be converted to clauses?

```
;; #1
(X ∨ Y ∨ Z)

;; #2
(X ∧ Y ∨ Z)

;; #3
((X ∧ Y) ⇒ Z)
```

Because we want to reason using resolution (which we've just seen operates on cluases), it would be nice to have our knowledge base in a format that consisted ONLY of clauses!

á **Conjunctive Normal Form (CNF)** is a sentence format consisting solely of a conjunction of clauses. This is the format we'd like to have our KB in!

Example
t Are the following sentences in CNF?

```
;; #1
(X ∨ Y) ∧ (Z ∨ X) ∧ (¬W ∨ D)

;; #2
(X ⇒ Y) ∧ (Z ⇒ X)

;; #3
¬(X ∧ Y) ∨ Z
```

H Fortunately, we can convert any arbitrary knowledge base in CNF; the procedure on page 253 of the textbook describes this.

Because of the assumption that our KB is in CNF, we often simply list a KB's sentences like:

```
KB = (X ∨ Y ∨ Z) ∧ (W ∨ M) ∧ (¬X ∨ W)

; Notationally represented as:
KB =
   1. (X ∨ Y ∨ Z)
   2. (W ∨ M)
   3. (¬X ∨ W)
```

Inference Using Refutation

OK, now that we have our knowledge base, the goal of the system is to prove certain facts about what we know.

We'll do this by illustrating that for some query sentence α, we will determine whether or not our KB ⊨ α

However, since it is difficult to show that, for all worlds a KB satisfies, α is also satisfied, we'll use a clever trick:

á **Refutation** AKA proof by contradiction, is an inference strategy whereby for query sentence α, we say KB ⊨ α IFF M(KB ∧ ¬α) = ∅

"Ugh... maths..." you might remark.

But it's really quite intuitive! Let's translate refutation inference into English:

H Refutation says, "Add the *opposite* of my query to the Knowledge Base, and if it makes the KB inconsistent, then I know my query had to have been entailed in the first place!"

Example
t Are the following KB's inconsistent?

```
;; #1
KB =
   1. ¬A
   2. (A ∨ B)
   3. A

;; #2
KB =
   1. ¬A
   2. (A ∨ B)
   3. ¬B
```

Let's try a simple example with our previous raining and sidwalk-wetness stuff:

```
Let R = whether or not it's raining
    S = whether or not the sidewalk is wet
    L = whether or not the sunlight is shining

KB =
   ; The sidewalk is wet IFF it is raining
   1. (R ⇒ S)
   2. (S ⇒ R)

   ; If the sun is shining, then it is not raining
   3. (L ⇒ ¬R)
    = (¬L ∨ ¬R)

   ; The sun is shining
   4. L

; ------------------------------------------------
; KB sentences 1 - 4 represent our axioms.
; Now, let's ask our query: Is the sidewalk dry?
; α = ¬S
; Therefore, we add ¬α = S to our KB:

   5. S

; ------------------------------------------------
; Now we use resolution to see if a contradiction
; exists in our KB! If it does, then KB ⊨ α

   6. R [Modus Ponens: #2 and #5]
   7. ¬L [Resolution: #3 and #6]
   8. ∅ [Resolution: #4 and #7]

→ ← Contradiction: We had both L and ¬L in
    the KB, which derived ∅

∴ KB ⊨ α
```

So, since we reached a contradiction, we conclude that KB ⊨ α since we added ¬α and made it inconsistent.

ň Note: We can make this assumption BECAUSE we assume the KB is consistent to begin with.

That said... what does it mean if we fail to reach a contradiction using resolution and we've exhausted all of our resolution possibilities in the KB?

Well, we know nothing! We can't conclude that our KB *doesn't* entail α, we just know that resolution didn't find the answer.

á For this reason, we say that resolution is **refutation complete**, meaning that if a contradiction exists in the KB from addition of our query ¬α, then resolution will find it... but if it doesn't find a contradiction, then we've learned nothing.

The other property of resolution is that it is sound:

á **Soundness** is a resolution strategy property meaning that only entailed sentences are ever derived during use of that algorithm; i.e., no sentences are ever made up.

We don't want our reasoning system just making things up!

So that's resolution in a nutshell!

Other Sentence Formats

As an appendix to this section, I wanted to mention two special logical sentence formats that have special meaning for inference:

á A **positive literal** is any non-negated proposition in a sentence.

á **Horn Clauses** are still clauses (disjunction of propositions) that have the additional constraint of allowing *at most* one positive literal.

Example

t Which of the following are Horn clauses?

```
;; #1
(¬X ∨ Y ∨ ¬Z)

;; #2
(X ⇒ Y)

;; #3
(¬X)

;; #4
(X ∨ Y ∨ Z)
```

á A **definite clause** has *exactly* one positive literal.

Example
t Which of the following are definite clauses?

```
;; #1
(¬X ∨ Y ∨ ¬Z)

;; #2
(X ⇒ Y)

;; #3
(¬X)

;; #4
(X ∨ Y ∨ Z)
```

The significance of these forms are discussed on page 256 of the book and can be used for efficient inference algorithms.

Homework 2

We'll save the last 5 - 10 minutes of discussion going over HW2 here!

Any time left will be devoted to practice, below.

Practice

Let's try out some of the propositional logic stuff we learned today!

Example

t For variables P and Q, show, using truth tables, that the following sentences are equivalent:

```
P ⇒ ¬Q
Q ⇒ ¬P
```

Example

t Use Boolean logic properties to convert the following sentence into CNF:

```
;; #1
¬(X ∨ Y) ∧ Z

;; #2
(X ∧ Y) ⇒ Z

;; #3
(X ∨ Y) ⇒ Z
```

Example

t Perform resolution on the following KB to answer the question as to whether KB $\models \alpha$

$\alpha = \neg X \land Y$
$\neg\alpha = X \lor \neg Y$

KB =
1. $X \lor Z \lor Y$
2. $\neg Z \lor W \lor X$
3. $\neg X \lor W$
4. $\neg W$