Logic Programming
Transitions

Michael Genesereth
Computer Science Department
Stanford University

Queries

True or False questions:

e.g. *Is Art the parent of Bob?*

Fill-in-the-blanks questions:

e.g. *Art is the parent of ____?*

e.g. *____ is the parent of Bob?*

e.g. *____ is the parent of ____?*

Compound questions:

e.g. *Is Art the parent of Bob* ***or*** *the parent of Bud?*

e.g. *____ has sons* ***and*** *no daughters?*

Values

Numbers and Strings:

e.g. 2 + 2?

e.g. *Concatenation of "abc" and "def"?*

Aggregates:

e.g. *Number of children of Art?*

e.g. *Number of people with daughters?*

Combinations:

e.g. *maximum of 2+3 and 2*3?*

e.g. *number Art's sons + number of Art's daughters?*

Transforms

Additions

e.g. *Add the fact that Art is the parent of Bill*.

Deletions

e.g. *Drop the fact that Art is the parent of Bill*.

Conditional Changes:

e.g. *Add grandparent facts implied by parent facts*.

e.g. *Replace parent facts with corresponding child facts*.

Transitions

Transforms

Additions

e.g. *Add the fact that Art is the parent of Bill*.

Removals

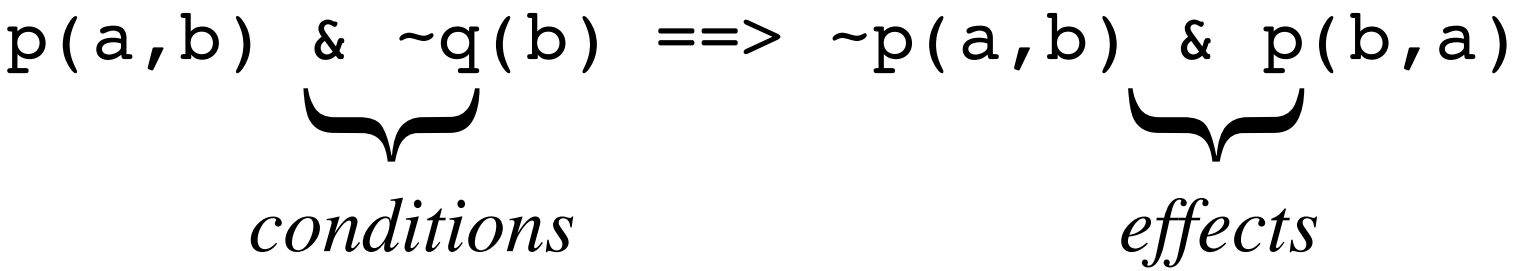
e.g. *Drop the fact that Art is the parent of Bill*.

Conditional Changes:

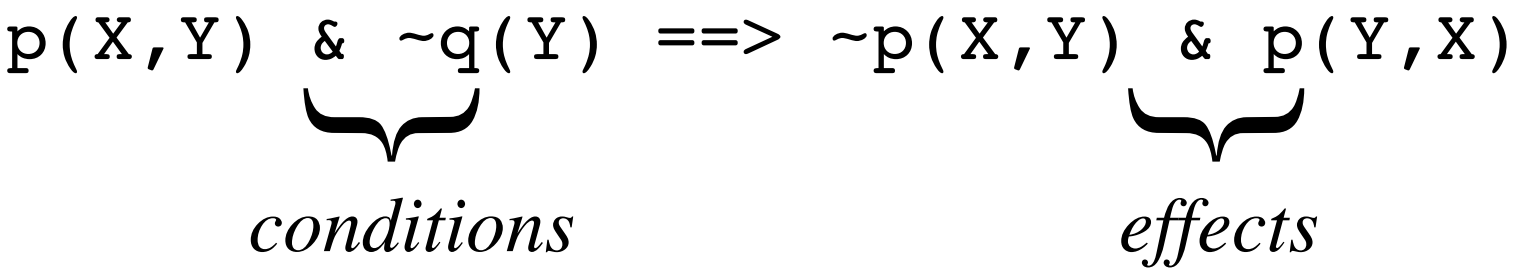
e.g. *Add grandparent facts implied by parent facts*.

e.g. *Replace parent facts with corresponding child facts*.

Update Rule



Update Rule



Safety

A transition rule is **safe** if and only if every variable in every literal on the right hand side appears in a positive literal on the left hand side. Also, every variable in a negative literal on the left hand side appears in a prior positive literal.

Safe Transition Rule

```
p(X) & ~q(X) ==> ~p(X) & q(X)
```

Unsafe Transition Rules

```
p(X) & ~q(X) ==> ~p(X) & q(Y)
p(X) & ~q(Y) ==> ~p(X) & q(X)
```

Updates

An **update** is a finite collection of update rules.

Example

```
p(X) & ~q(X) ==> ~p(X) & q(X)
p(X) & q(X) ==> ~p(X) & ~q(X)
```

Instances

An **instance of an update rule** is a rule in which all variables have been consistently replaced by ground terms.

Rule

```
p(X) & ~q(X) ==> ~p(X) & q(X)
```

Herbrand Universe

```
{a, b, c, d}
```

Instances

```
p(a) & ~q(a) ==> ~p(a) & q(a)
p(b) & ~q(b) ==> ~p(b) & q(b)
p(c) & ~q(c) ==> ~p(c) & q(c)
p(d) & ~q(d) ==> ~p(d) & q(d)
```

Active and Inactive Rule Instances

A dataset Δ, we say that update rule instance *r* is **active** if and only if Δ contains all of the *r*'s positive conditions and none of its negative conditions. Otherwise, it is **inactive**.

Rule: `p(X) & ~q(X) ==> ~p(X) & q(X)`

Dataset: `{p(b),p(c),p(d),q(d)}`

Instances (active black, inactive grey):

```
p(a) & ~q(a) ==> ~p(a) & q(a)
p(b) & ~q(b) ==> ~p(b) & q(b)
p(c) & ~q(c) ==> ~p(c) & q(c)
p(d) & ~q(d) ==> ~p(d) & q(d)
```

Positive and Negative Updates

The **positive update set** A(r,Δ) for an update *rule r* on a dataset Δ is set of all positive conclusions in some active instance of r. The **negative update set** D(r,Δ) for an update *rule r* on a dataset Δ is set of all negative conclusions in some active instance of r.

The **positive update set** A(Ω,Δ) for a *set of rules* Ω on a dataset Δ is the union of the positive updates for all of the rules in Δ. The **negative update set** D(Ω,Δ) for a *set of rules* Ω on a dataset Δ is the union of the negative updates of all of the rules in Δ.

Update

The **result** of applying an update Ω to a dataset Δ (written $u(\Omega,\Delta)$) is the set obtained by removing the negative updates and adding in the positive updates, i.e. the result is.

$$u(\Omega,\Delta) = \Delta - D(\Omega,\Delta) \cup A(\Omega,\Delta)$$

Rule: `p(X) & ~q(X) ==> ~p(X) & q(X)`

Dataset: `{p(b),p(c),p(d),q(d)}`

Negative Update: `{p(b),p(c)}`
Positive Update: `{q(b),q(c)}`

Result: `{p(d),q(b),q(c),q(d)}`

Interesting Example

Update Rule:

```
p(X,a) ==> ~p(X,a) & p(a,X)
```

Positive Update Set: {p(a,a)}
Negative Update Set: {p(a,a)}

Dataset:

```
p(a,a)
p(a,b)
p(b,c)
p(c,c)
p(c,d)
```

New Dataset:

```
p(a,a)
p(a,b)
p(b,c)
p(c,c)
p(c,d)
```

Interesting Example

Update Rule:

```
p(X,a) ==> ~p(X,a) & p(a,X)
```

Positive Update Set: {p(a,a)}
Negative Update Set: {p(a,a)}

Dataset:

```
p(a,a)
p(a,b)
p(b,c)
p(c,c)
p(c,d)
```

New Dataset:

```
p(a,a)
p(a,b)
p(b,c)
p(c,c)
p(c,d)
```

Production Systems

A **production system** is a set of condition-action rules. On each step in the execution of a production system, an active rule instance is chosen and the actions are performed. The cycle then repeats on the new state.

```
if p(X), then del p(X) and add p(X+10)
```

Before: `{p(10), p(20), p(30)}`
Step 1: `{p(20), p(30)}`
Step 2: `{p(30)}`

Updates

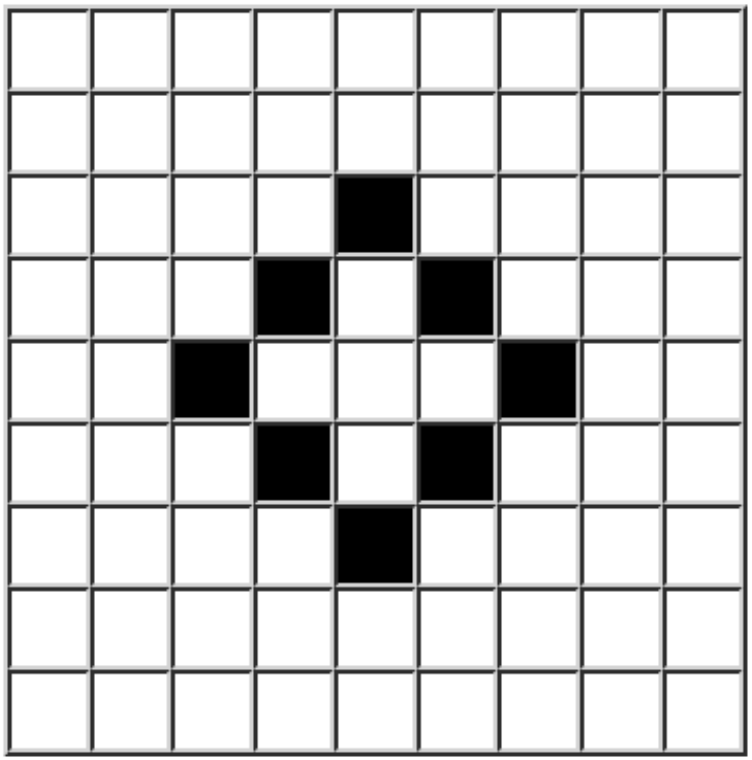
Updates differ from production systems in that all active update rules "fire" at the same time. (1) All updates are computed *before* any changes are made, and (2) all changes are made simultaneously.

```
p(X) & evaluate(plus(X,10),Y) ==> ~p(X) & p(Y)
```

Before: {p(10), p(20), p(30)}
Step 1: {p(20), p(30), p(40)}
Step 2: {p(30), p(40), p(50)}

The Game of Life

World



Rules of the Game

(1) Any *live* cell with *two or three* live neighbors lives on to the next generation.

(2) Any *live* cell with *fewer than two* live neighbors dies (as if caused by underpopulation).

(3) Any *live* cell with *more than three* live neighbors dies (as if by overpopulation).

(4) Any *dead* cell with *exactly three* live neighbors becomes a live cell (as if by reproduction).

Vocabulary

Symbols: `c11, c12, …`

Unary Predicates:

`on` - cell is live

`cell` - true of cells

Binary Predicates:

`neighbor` - cells are neighbors

Social Starvation

Any *live* cell with *fewer than two* live neighbors dies.

```
on(Y) & countofall(X,neighbor(X,Y)&on(X),0)
 ==> ~on(Y)

on(Y) & countofall(X,neighbor(X,Y)&on(X),1)
 ==> ~on(Y)
```

Overcrowding

Any *live* cell with *more than three* live neighbors dies.

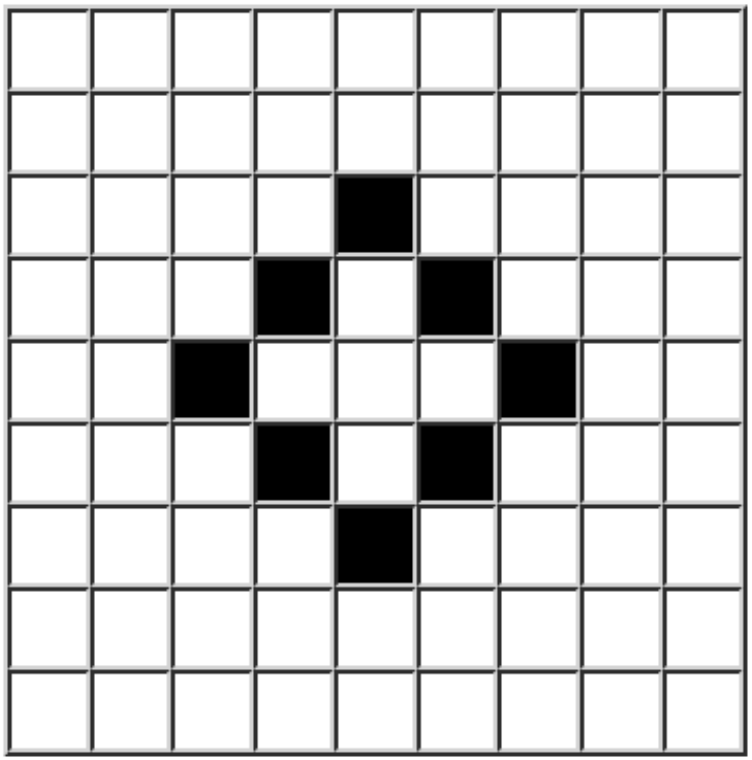
```
on(Y) & countofall(X,neighbor(X,Y)&on(X),N) & leq(4,N)
  ==> ~on(Y)
```

Reproduction

Any *dead* cell with *exactly three* live neighbors becomes live.

```
cell(Y) & ~on(Y) & countofall(X,neighbor(X,Y)&on(X),3)
  ==> on(Y)
```

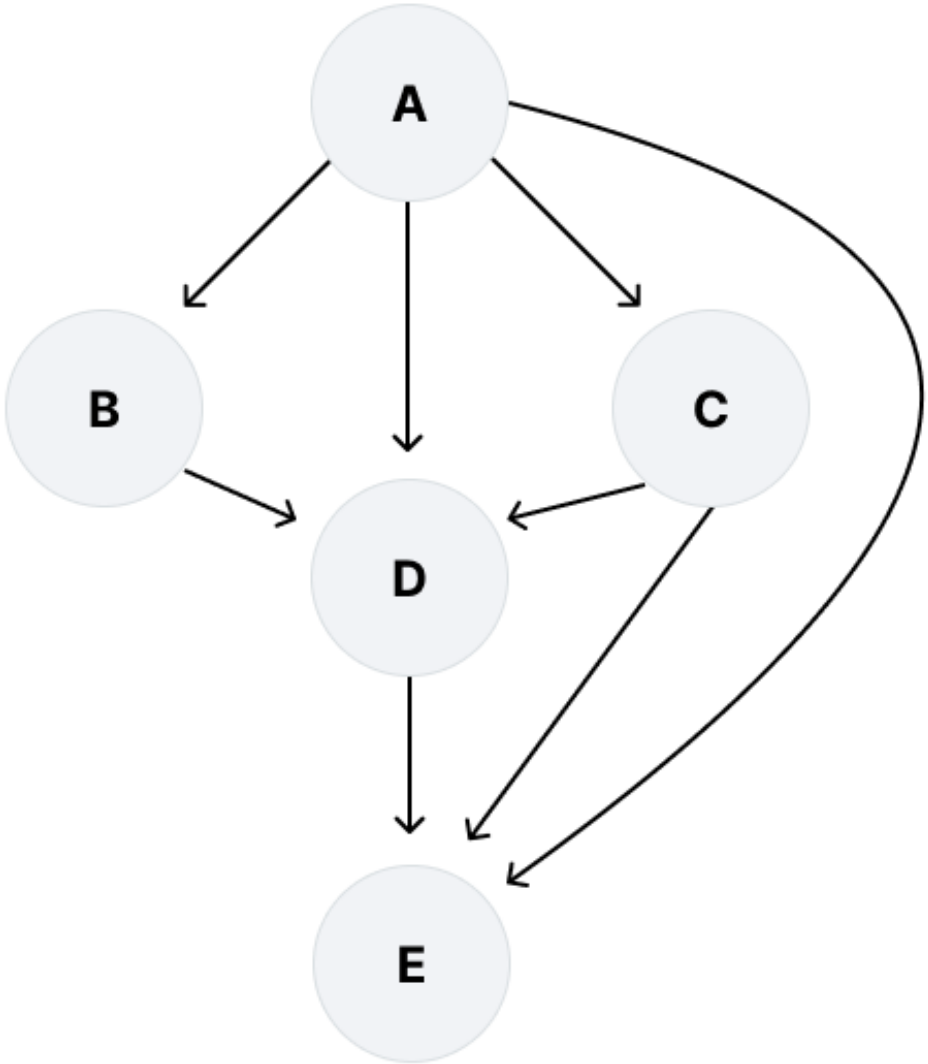
Example



Demonstration

Directed Graphs

Example



Copying

Dataset:

`{edge(a,b),edge(b,d),edge(b,e)}`

Operation Definition:

`edge(b,Z) ==> edge(c,Z)`

Negative Updates: {}

Positive Updates: `{edge(c,d),edge(c,e)}`

Result:

`{edge(a,b),edge(b,d),edge(b,e),edge(c,d),edge(c,e)}`

Reversing Arguments

Dataset:

{edge(a,b), edge(b,d), edge(b,e)}

Operation Definition:

```
edge(X,Y) ==> ~edge(X,Y) & edge(Y,X)
```

Negative Updates: {edge(a,b), edge(b,d), edge(b,e)}

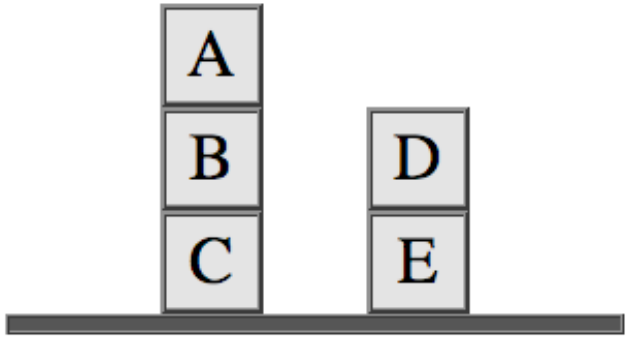
Positive Updates: {edge(b,a), edge(d,b), edge(e,b)}

Result:

{edge(b,a), edge(d,b), edge(e,b)}

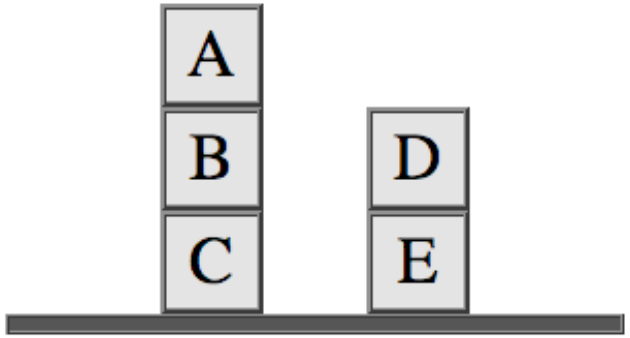
Blocks World

Data




```
block(a)
block(b)      on(a,b)
block(c)      on(b,c)
block(d)      on(d,e)
block(e)
```

Support and Clutter




```
block(a)
block(b)      on(a,b)
block(c)      on(b,c)
block(d)      on(d,e)
block(e)
```

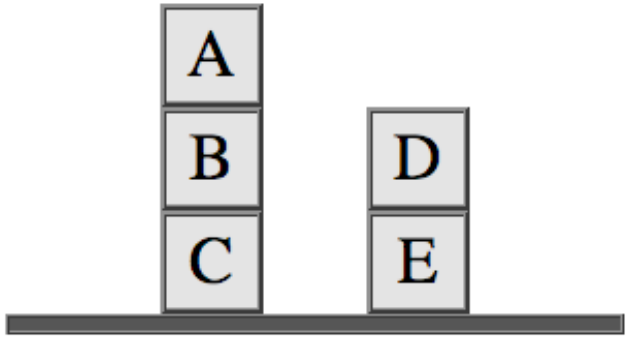
```
goal(X) :- on(X,Y)

      goal(a)
      goal(b)
      goal(d)

goal(Y) :- on(X,Y)

      goal(b)
      goal(d)
      goal(e)
```

Materializing Relations

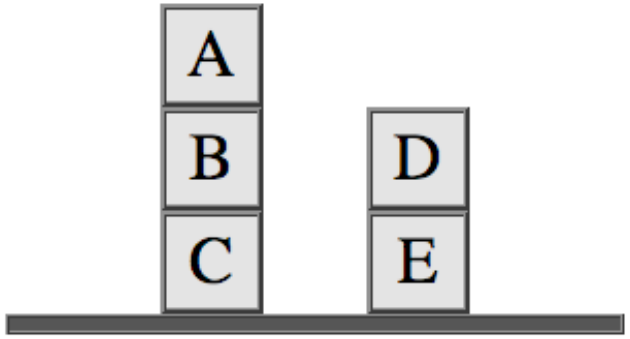



```
block(a)
block(b)      on(a,b)
block(c)      on(b,c)
block(d)      on(d,e)
block(e)
```

```
on(X,Y) ==> supported(X) & cluttered(Y)
```

```
supported(a)
supported(b)
supported(d)
cluttered(b)
cluttered(c)
cluttered(e)
```

Blocks World - above

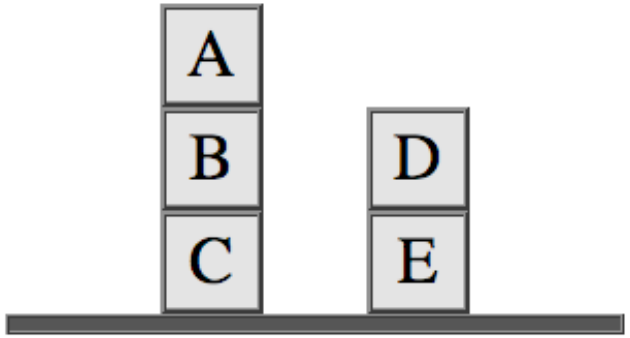



```
block(a)
block(b)      on(a,b)
block(c)      on(b,c)
block(d)      on(d,e)
block(e)
```

```
goal(X,Y) :- on(X,Y)
goal(X,Z) :- on(X,Y) & on(Y,Z)
goal(X,W) :- on(X,Y) & on(Y,Z) & on(Z,W)
...

goal(a,b)
goal(b,c)
goal(a,c)
goal(d,e)
```

Blocks World - above




```
block(a)
block(b)      on(a,b)
block(c)      on(b,c)
block(d)      on(d,e)
block(e)
```

```
on(X,Y) ==> above(X,Y)
```

```
above(a,b)
above(b,c)
above(d,e)
```

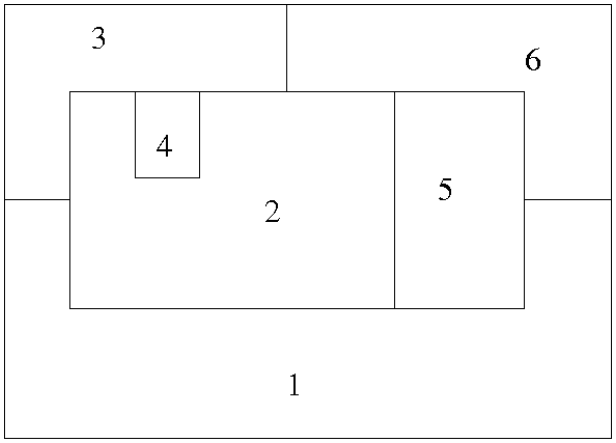
```
on(X,Y) & above(Y,Z) ==> above(X,Z)
```

```
above(a,c)
```

Repeat till fixpoint ("*Bottom-up*" *computation of* `above`.)

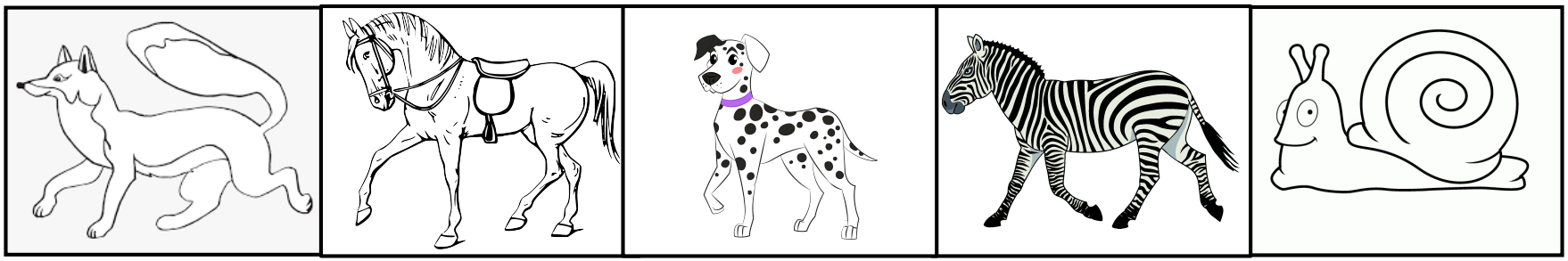
Constraint Propagation

Constraint Satisfaction Problems



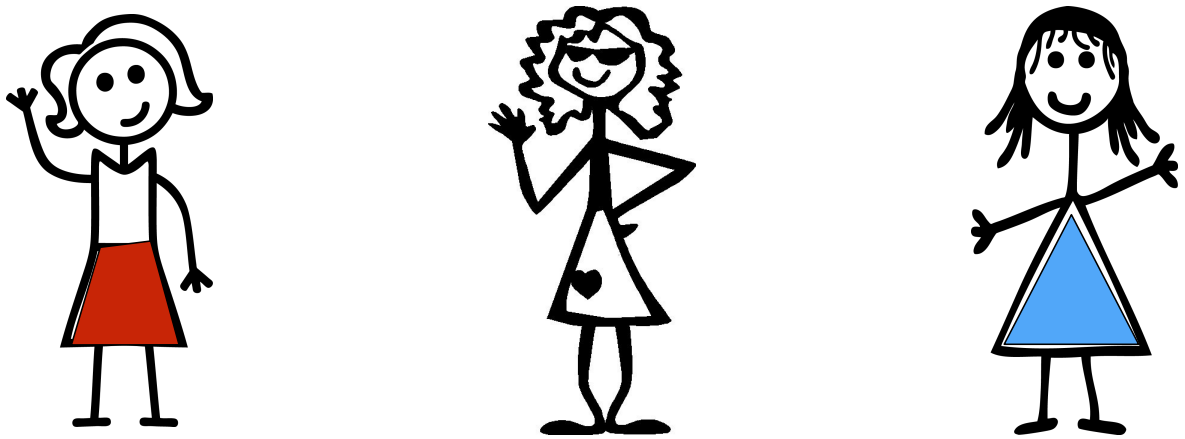

```
  SEND
+MORE
-----
MONEY
```

	6		1		4		5	
		8	3		5	6		
2								1
8			4		7			6
		6				3		
7			9		1			4
5								2
		7	2		6	9		
	4		5		8		7	



Problem Statement

Ruby Red, Willa White, Betty Blue are having lunch.
One is wearing a red skirt, one white, one blue.

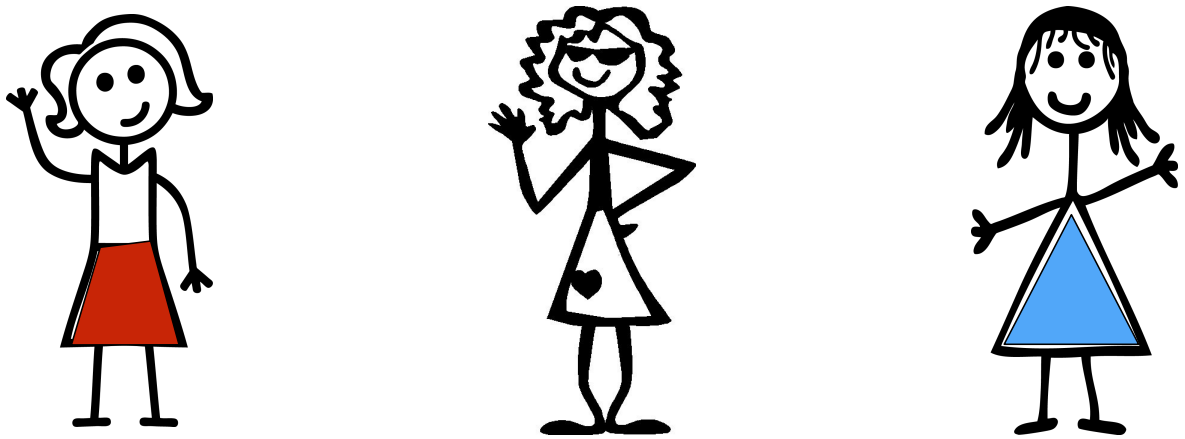


Betty to woman in white skirt: *Have you noticed that our skirts have colors different from our names?*

Which woman is wearing which skirt?

Problem Statement

Ruby Red, Willa White, Betty Blue are having lunch.
One is wearing a red skirt, one white, one blue.



Betty to woman in white skirt: *Have you noticed that our skirts have colors different from our names?*

Why is this problem so easy to solve?

Vocabulary

Symbols: `red`, `white`, `blue`, `v`, `x`

Unary predicates Predicates:

`color(C)` is true if C is a color.

Ternary Predicates:

`c(P,C,v)` is true if person P is wearing color C.

`c(P,C,x)` is true if person P is not wearing color C.

Constraint Propagations Operations:

`c1` - No person is wearing same color as name.

`c2` - Betty Blue spoke to person in white.

`c3` - Every person has one color and vice versa.

Colors

```
color(red)
color(white)
color(blue)
```

First Given

No person is wearing same color as name.

```
color(C) ==> c(C,C,x)
```

Second Given

Betty Blue spoke to person in white.

```
c(blue,white,x)
```

Existence Constraints

Every person is wearing some color.

```
c(P,C1,x) & c(P,C2,x) & color(C3) & mutex(C1,C2,C3)
  ==> c(P,C3,v)
```

Every color is worn.

```
c(P1,C,x) & c(P2,C,x) & color(P3) & mutex(P1,P2,P3)
  ==> c(P3,C,v)
```

Uniqueness Constraints

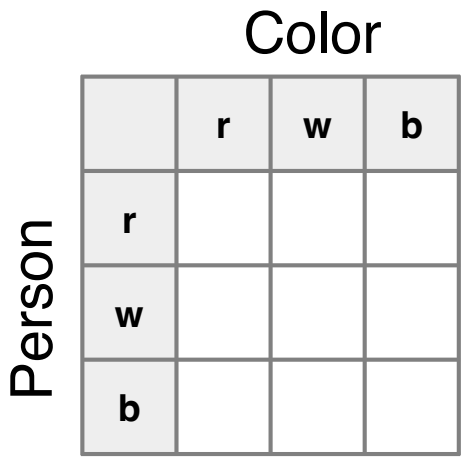
Nobody is wearing two colors.

```
c(P,C1,v) & color(C2) & distinct(C1,C2)
  ==> c(P,C2,x)
```

No color is worn by two people.

```
c(P1,C,v) & color(P2) & distinct(P1,P2)
  ==> c(P2,C,x)
```

Initial State



First Given

No person is wearing same color as name.

```
color(C) ==> c(C,C,x)
```

	r	w	b
r			
w			
b			

	r	w	b
r	X		
w		X	
b			X

Second Given

Betty Blue spoke to person in white.

```
c(blue,white,x)
```

	r	w	b
r	X		
w		X	
b			X

	r	w	b
r	X		
w		X	
b		X	X

Existence Constraints

```
c(P,C1,x) & c(P,C2,x) & color(C3) & mutex(C1,C2,C3)
  ==> c(P,C3,v)

c(P1,C,x) & c(P2,C,x) & human(P3) & mutex(P1,P2,P3)
  ==> c(P3,C,v)
```

	r	w	b
r	X		
w		X	
b		X	X

	r	w	b
r	X	V	
w		X	
b	V	X	X

Uniqueness Constraints

```
c(P,C1,v) & color(C2) & distinct(C1,C2)
  ==> c(P,C2,x)
```

```
c(P1,C,v) & person(P2) & distinct(P1,P2)
  ==> c(P2,C,x)
```

	r	w	b
r	X	V	
w		X	
b	V	X	X

	r	w	b
r	X	V	X
w	X	X	
b	V	X	X

Existence Constraints

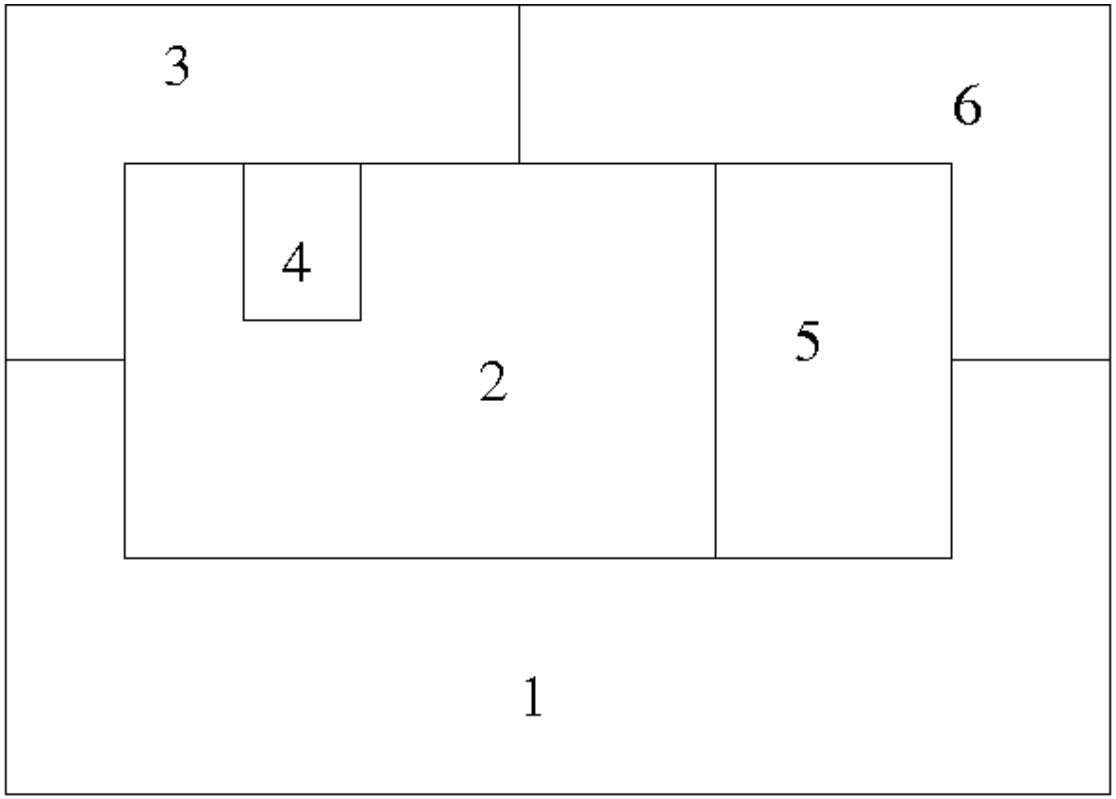
```
c(P,C1,x) & c(P,C2,x) & color(C3) & mutex(C1,C2,C3)
  ==> c(P,C3,v)
```

```
c(P1,C,x) & c(P2,C,x) & human(P3) & mutex(P1,P2,P3)
  ==> c(P3,C,v)
```

	r	w	b
r	X	V	X
w	X	X	
b	V	X	X

	r	w	b
r	X	V	X
w	X	X	V
b	V	X	X

Map Coloring



Sudoku

	6		1		4		5	
		8	3		5	6		
2								1
8			4		7			6
		6				3		
7			9		1			4
5								2
		7	2		6	9		
	4		5		8		7