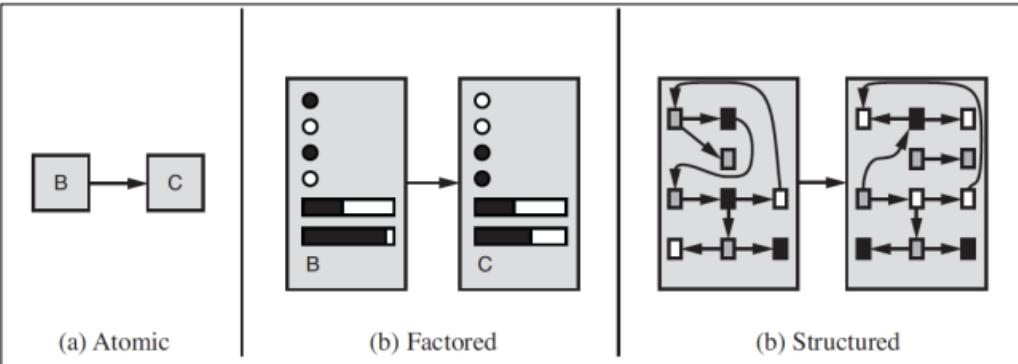


# TDDC17

Fö 4  
Constraint Satisfaction Problems

# Representing States



So far:  
Unformed search  
Heuristic search

Today:  
Constraints

- 10 binary variables can describe  $2^{10} = 1,024$  worlds
- 20 binary variables can describe  $2^{20} = 1,048,576$  worlds
- 30 binary variables can describe  $2^{30} = 1,073,741,824$
- 100 binary variables can describe  $2^{100} = 1,267,650,600,228,401,496,703,205,376$  worlds

# An Example Problem

Suppose a delivery robot must carry out a number of Delivery activities, a,b,c,d,e. Suppose that each activity Can happen in the next four hours at hour 1,2,3,or 4.

Let A,B,C,D,E, be variables representing the time activities a,b,c,d,e start, respectively.

Variable domains for each activity start-time will be {1,2,3,4}  
Assume the following constraints on start times:

$$A \neq B$$

$$A = D$$

$$E < A$$

$$B \neq D$$

$$C < D$$

$$E < D$$

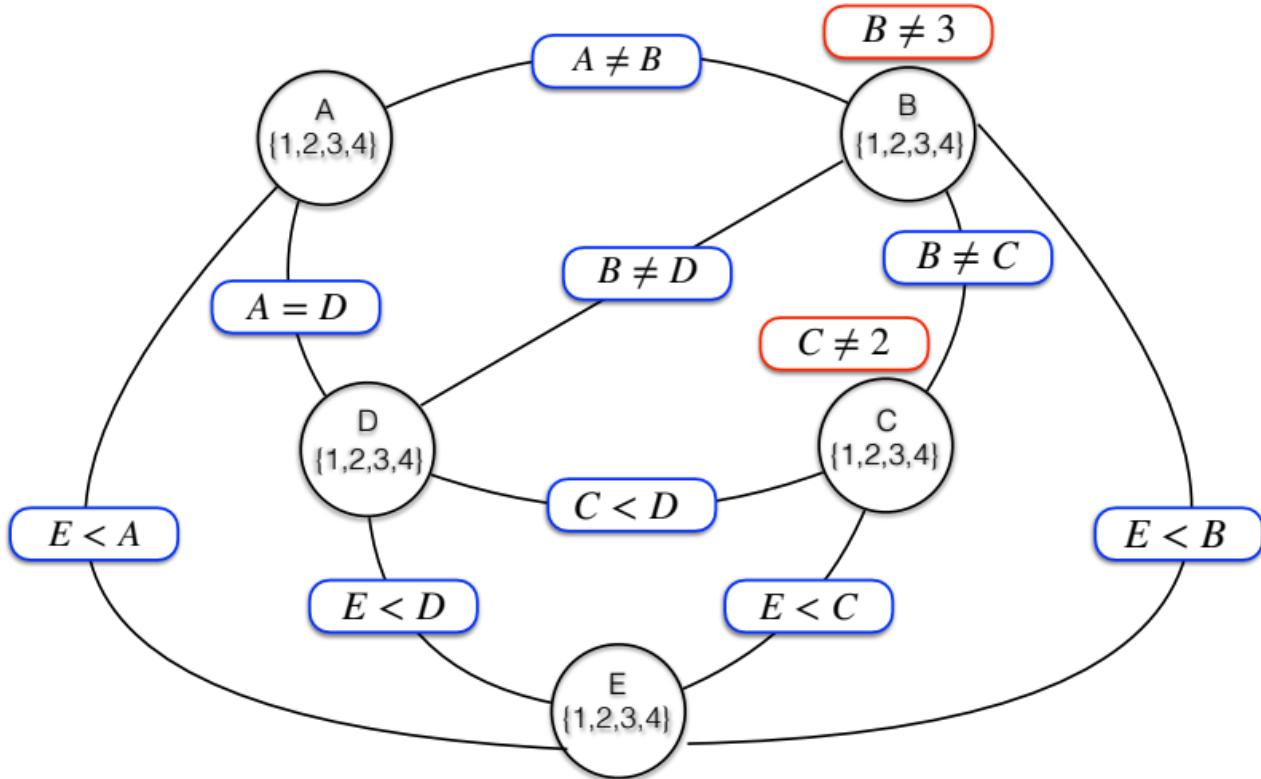
$$B \neq C$$

$$E < B$$

$$E < C$$

$$B \neq 3$$

# Delivery Robot: Constraint graph



# Constraint Satisfaction Problem

$X$  is a set of variables  $\{X_1, \dots, X_n\}$

$D$  is a set of domains  $D_1, \dots, D_n$ , one for each variable

$C$  is a set of constraints on  $X$

*The constraints restrict the values variables can simultaneously take.*

## Solution to a CSP

An assignment of a value from its domain to each variable, in such a way that all the constraints are satisfied

One may want to find 1 solution, all solutions, an optimal solution, or a good solution based on an objective function defined in terms of some or all variables.

# Delivery Robot: CSP Formulation

```
RobD = CSP({'A':{1,2,3,4}, 'B':{1,2,3,4}, 'C':{1,2,3,4},  
           'D':{1,2,3,4}, 'E':{1,2,3,4}},  
           [Constraint(('B',), ne_(3)),  
            Constraint(('C',), ne_(2)),  
            Constraint(('A','B'), ne),  
            Constraint(('B','C'), ne),  
            Constraint(('C','D'), lt),  
            Constraint(('A','D'), eq),  
            Constraint(('A','E'), gt),  
            Constraint(('B','E'), gt),  
            Constraint(('C','E'), gt),  
            Constraint(('D','E'), gt),  
            Constraint(('B','D'), ne)])
```

# Variable, Domain and Constraint Types

## Types of variables/domains

- Discrete variables
  - Finite or infinite domains
- Boolean variables
  - Finite domain
- (Continuous variables)
  - Infinite domain

## Some Special cases

- Linear programming
  - Linear inequalities forming a convex region.
  - Continuous domains.
- Solutions in time polynomial to the number of variables
- Integer programming
  - Linear constraints on integer variables.

## Types of constraints

- Unary constraints (1)
- Binary constraints (2)
- Higher-Order constraints (>2)
- Linear constraints
- Nonlinear constraints

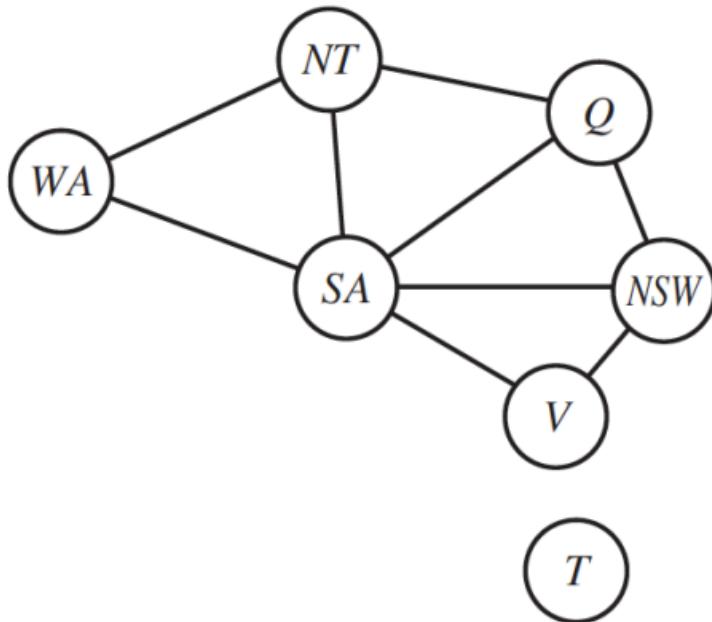
Any higher-order/finite domain CSP's can be translated into binary/finite domain CSPs! (In the book, R/N stick to these)

# Map Coloring Problem



Color each of the territories/states  
**red**, **green** or **blue** with no neighboring  
region having the same color

# Let's Abstract

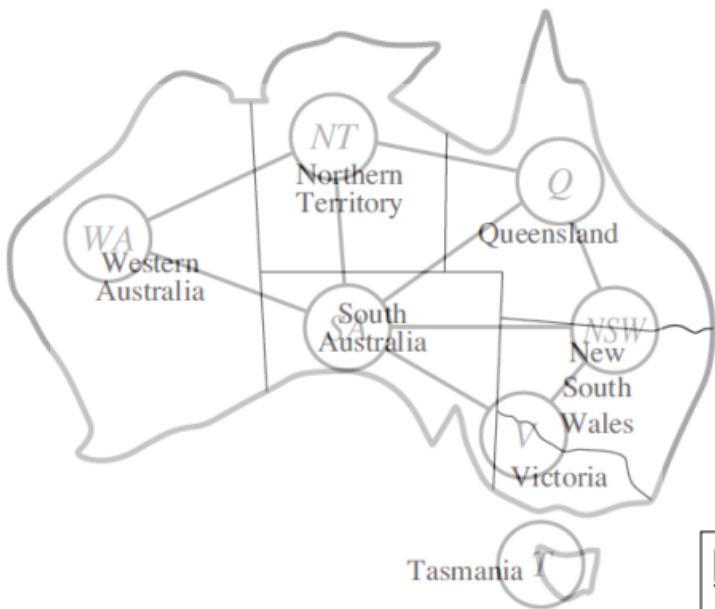


Constraint Graph

Nodes are variables  
Arcs are constraints



# Our Representation



- Associate a variable with each region.
- Introduce a set of values the variables can be bound to.
- Define constraints on the variable/value pairs

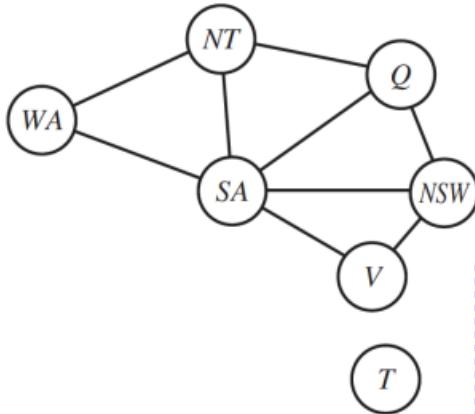
Goal:

Find a set of legal bindings satisfying the constraints!

# Map Coloring: Australia

## Map Coloring Specification

- $X = \{\text{WA}, \text{NT}, \text{SA}, \text{Q}, \text{NSW}, \text{V}, \text{T}\}$
- $D = \{\{\text{red}, \text{green}, \text{blue}\}, \dots, \{\text{red}, \text{green}, \text{blue}\}\}$
- $C$  is a set of constraints on  $X$



## Binary Constraints

### Constraints

$\text{WA} \neq \text{NT}$ ,  
 $\text{WA} \neq \text{SA}$ ,  
 $\text{NT} \neq \text{Q}$ ,  
 $\text{NT} \neq \text{SA}$ ,  
 $\text{Q} \neq \text{SA}$ ,  
 $\text{Q} \neq \text{NSW}$ ,  
 $\text{V} \neq \text{SA}$ ,  
 $\text{V} \neq \text{NSW}$

```
CSP({{'WA':{'red','blue','green'},'NT':{'red','blue','green'},  
'SA':{'red','blue','green'},'Q':{'red','blue','green'},  
'NSW':{'red','blue','green'},'V':{'red','blue','green'},  
'T':{'red','blue','green'}},  
[Constraint(('WA','NT'),ne),Constraint(('WA','SA'),ne),  
Constraint(('NT','SA'),ne),Constraint(('NT','Q'),ne),  
Constraint(('SA','Q'),ne),Constraint(('SA','NSW'),ne),  
Constraint(('SA','V'),ne), Constraint(('Q','NSW'),ne)])]
```

# Sudoku

	1	2	3	4	5	6	7	8	9
A			3		2		6		
B	9			3		5			1
C			1	8		6	4		
D			8	1		2	9		
E	7								8
F			6	7		8	2		
G			2	6		9	5		
H	8			2		3			9
I			5		1		3		

	1	2	3	4	5	6	7	8	9
A	4	8	3	9	2	1	6	5	7
B	9	6	7	3	4	5	8	2	1
C	2	5	1	8	7	6	4	9	3
D	5	4	8	1	3	2	9	7	6
E	7	2	9	5	6	4	1	3	8
F	1	3	6	7	9	8	2	4	5
G	3	7	2	6	8	9	5	1	4
H	8	1	4	2	5	3	7	6	9
I	6	9	5	4	1	7	3	8	2

Variables

81(one for each cell)

Constraints:

Alldiff() for each row

Alldiff() for each column

Alldiff for each 9 cell area

# Another Example

- Suppose our territories are coverage areas, each with a sensor that monitors the area.
- Each sensor has  $N$  possible radio frequencies
- Sensors overlap if they are in adjacent areas
- If sensors overlap, they can not use the same frequency

Find a solution where each sensor uses a frequency that does not interfere with adjacent coverage areas

This is an  $N$ -map coloring problem!

# Advantages of CSPs

- Representation is closer to the original problem.
- Representation is the same for all constraint problems.
- Algorithms used are domain independent with the same general purpose heuristics for all problems
- Algorithms are simple and often find solutions quite rapidly for large problems
  - CSPs often more efficient than regular state-space search because it can quickly eliminate large parts of the search space
  - Many problems intractable for regular state-space search can be solved efficiently with a CSP formulation.

# Solving a CSP: Types of Algorithms

## Search

(choose a new variable assignment)

## Inference

### Constraint Propagation

(reduce the # of legal values for a variable and propagate to other variables)

## Constraint Propagation

Sometimes solves  
the problem  
without search!

## Constraint Propagation

Pre-Processing

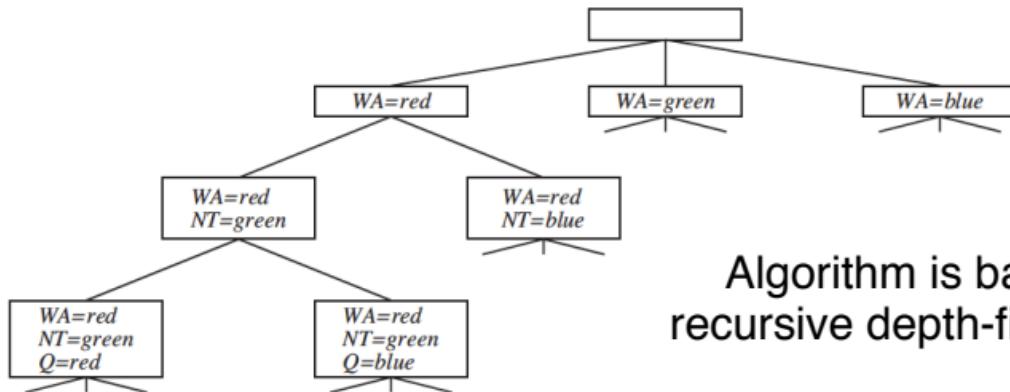
## Search

## Constraint Propagation

Interleave

## Search

# Simple Backtracking Search Algorithm for CSPs



Algorithm is based on recursive depth-first search

If a value assignment to a variable leads to failure then it is removed from the current assignment and a new value is tried (backtrack)

The algorithm will interleave inference with search

# Backtracking Algorithm (Search with Inference)

```
function BACKTRACKING-SEARCH(csp) returns a solution, or failure
    return BACKTRACK({ }, csp)

function BACKTRACK(assignment, csp) returns a solution, or failure
    if assignment is complete then return assignment
    var  $\leftarrow$  SELECT-UNASSIGNED-VARIABLE(csp)
    for each value in ORDER-DOMAIN-VALUES(var, assignment, csp) do
        if value is consistent with assignment then
            add {var = value} to assignment
            inferences  $\leftarrow$  INFERENCE(csp, var, value)
            if inferences  $\neq$  failure then
                add inferences to assignment
                result  $\leftarrow$  BACKTRACK(assignment, csp)
                if result  $\neq$  failure then
                    return result
            remove {var = value} and inferences from assignment
    return failure
```

Domain  
Independent  
Heuristics

Inference

# Backtracking Algorithm (Search with Inference)

```
def backtrack_search_constraints(cspproblem):  
  
    def backtrack(assignment,cspproblem):  
        if assignment.complete(cspproblem):  
            return assignment  
        var = select_unassigned_variable(cspproblem,assignment)  
        for value in order_domain_values(var,assignment,cspproblem):  
            if assignment.consistent_with(var,value, cspproblem):  
                assignment.add(var,value)  
                infer = inferences(cspproblem,var,assignment)  
                if not infer == 'failure':  
                    assignment.add_inferences(infer)  
                    result = backtrack(assignment, cspproblem)  
                    if not result == 'failure':  
                        return result  
                    assignment.remove(var,value)  
                    assignment.remove_inferences(infer)  
    return 'failure'  
  
    return backtrack(assignment(),cspproblem)
```

b\_s\_f calls:

```
def select_unassigned_variable(cspproblem,assignment):  
    return assignment.unassigned_variables(cspproblem)[0]
```

```
def inferences(cspproblem,var,assignment):  
    return {}
```

```
def order_domain_values(var,assignment,cspproblem):  
    return list(cspproblem.domains[var])
```

Domain Independent  
Heuristics

*Need to instantiate!*

# Potential Problems with backtracking search

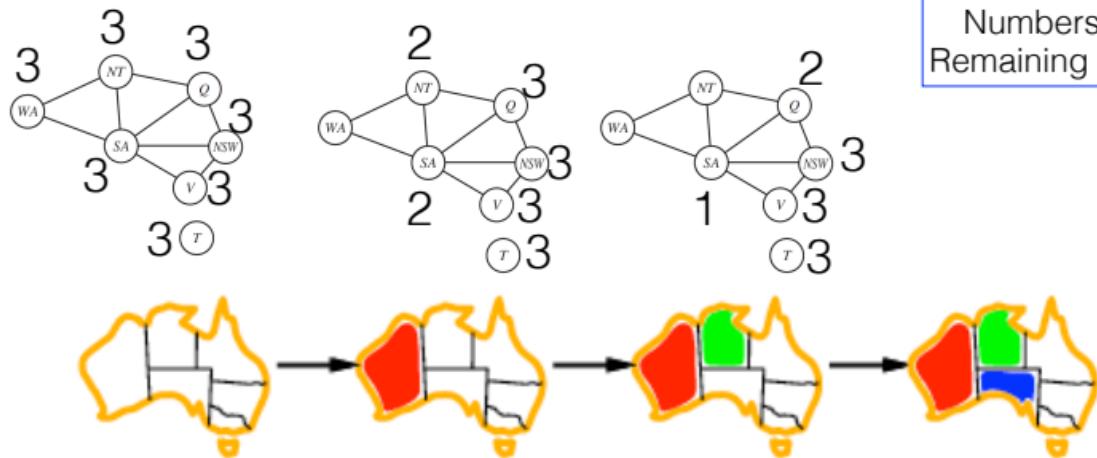
- Variable choice and value assignment is arbitrary
  - Which variable should be assigned?
    - `SELECT-UNASSIGNED-VARIABLE()`
  - Which values should be assigned first?
    - `ORDER-DOMAIN-VALUES()`
  - Conflicts detected too late (empty value domain)
    - Conflicts not detected until they actually occur.
    - What are the implications of current variable assignments for the other unassigned variables?
      - `INFERENCE()`
  - Thrashing
    - Major reason for failure is conflicting variables, but these conflicts are continually repeated throughout the search
    - When a path fails, can the search avoid repeating the failure in subsequent paths?
      - One solution: Intelligent Backtracking

# Variable Selection Strategies

- Variable Selection Strategy
  - **SELECT-UNASSIGNED-VARIABLE():**
  - Minimum Remaining Values (MRV) heuristic
    - Choose the variable with the fewest remaining legal values.
    - Try first where you are most likely to fail (fail early!..hard cases 1st)
      - Fail-First heuristic
      - Will knock out large parts of the search tree.
  - Degree Heuristic
    - Select the variable that is involved in the largest number of constraints on other unassigned variables.
    - Hard cases first!
    - Tie breaker when MRV can't be applied.

# Minimal Remaining Values (MRV)

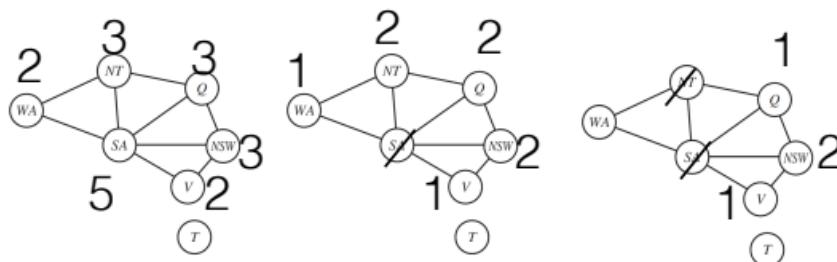
“Attempts to fail early, thus removing parts of the search tree”



Actually, if we used  
degree heuristic  
to break ties: then SA  
would be chosen here  
instead of WA

# Degree Heuristic

“Attempts to reduce the branching factor in search tree”



Numbers are  
arcs pointing to  
Variables with  
remaining values

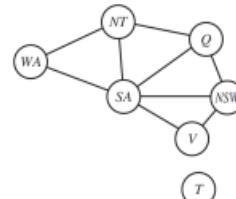


Can't use MRV:  
All have same number

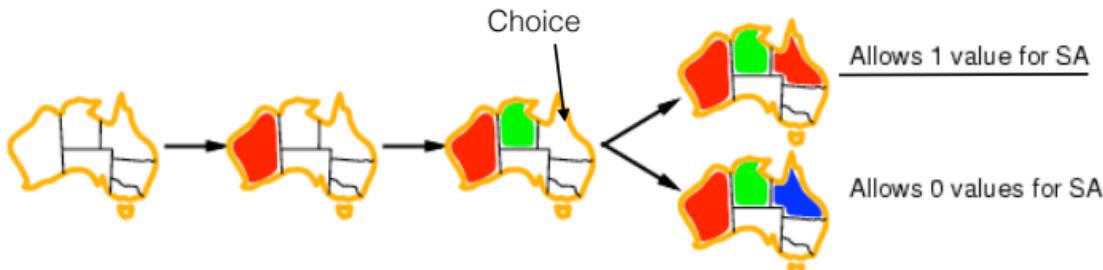
MRV: Only one choice  
of color!

# Value Selection Strategies

- Value Selection Strategy
  - ORDER-DOMAIN-VALUES()
  - Least-constraining-value heuristic



- Choose the value that rules out the fewest choices of values for the neighboring variables in the constraint graph.
  - Fail Last heuristic
- Maximize the number of options....least commitment.
  - Only useful when searching for one solution.



# Inference in CSPs

## Key Idea:

- Treat each variable as a node and each binary constraint as an arc in our constraint graph.
- Enforcing local consistency in each part of the graph eliminates inconsistent values throughout the graph.
- The less local we get when propagating the more expensive inference becomes.

### Node Consistency

A single variable is *node consistent* if all values in the variable's domain satisfy the variable's *unary* constraints

WA  $\neq$  green

WA={red, ~~green~~, blue}

# Arc Consistency

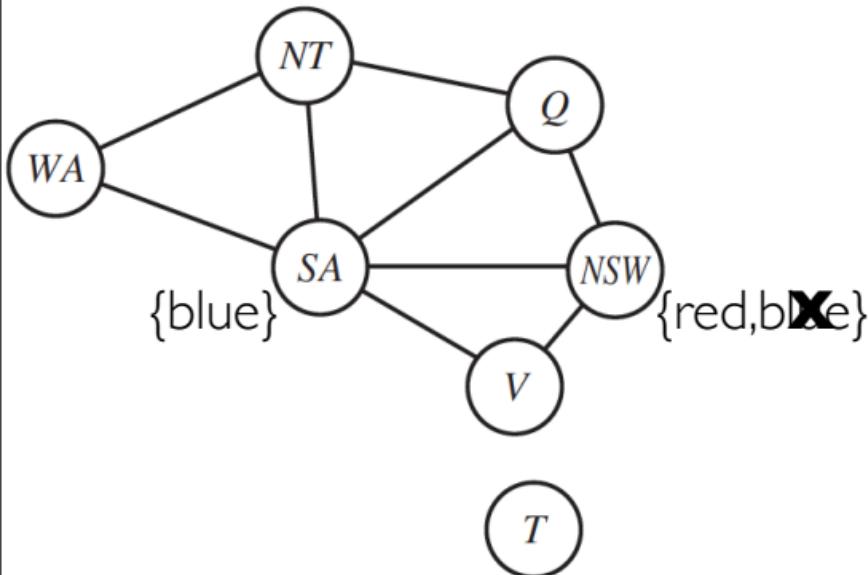
## Definition

Arc  $(V_i, V_j)$  is arc consistent if for every value  $x$  in the domain of  $V_i$  there is some value  $y$  in the domain of  $V_j$  such that  $V_i = x$  and  $V_j = y$  satisfies the constraints between  $V_i$  and  $V_j$ .

A constraint graph is *arc-consistent* if all its arcs are arc consistent

- The property is not symmetric.
- Arc consistent constraint graphs do not guarantee consistency of the constraint graph and thus guarantee solutions. They do help in reducing search space and in early identification of inconsistency.
- AC-3 ( $O(n^2d^3)$ ), AC-4 ( $O(n^2 * d^2)$ ) are polynomial algorithms for arc consistency, but 3SAT (in NP) is a special case of CSPs, so it is clear that AC-3, AC-4 do not guarantee (full) consistency of the constraint graph.

# Arc Consistency is not Symmetric



SA → NSW

Is arc-consistent

NSW → SA

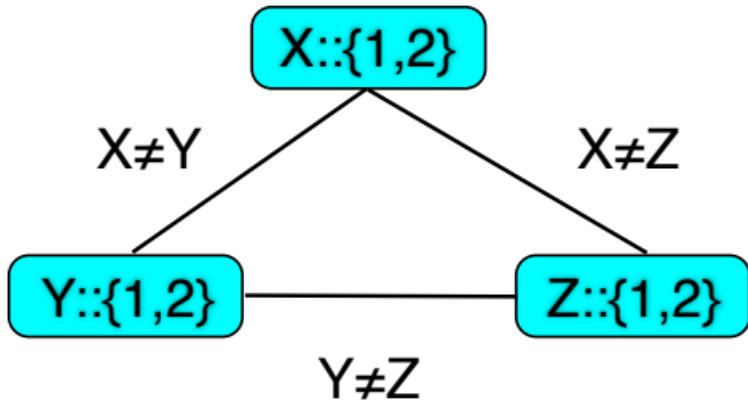
Is not arc-consistent

Remove blue from NSW

NSW → SA

Is now arc-consistent

Arc Consistency does not guarantee a solution



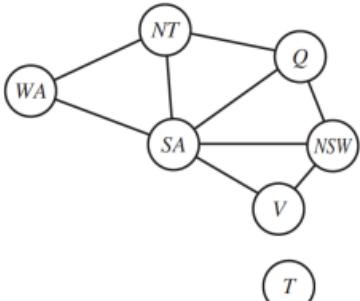
Arc consistent constraint graph  
with no solutions

# Simple Inference: Forward Checking

Whenever a variable X is assigned, look at each unassigned variable Y that is connected to X by a constraint and delete from Y's domain any value that is inconsistent with the value chosen for X. [make all Y's arc consistent with X]

Forward check each time  
A variable binding is added:

	WA	NT	Q	NSW	V	SA	T
Initial domains	R G B	R G B	R G B	R G B	R G B	R G B	R G B
After WA=red	(R)	G B	R G B	R G B	R G B	G B	R G B
After Q=green	(R)	B	(G)	R B	R G B	B	R G B
After V=blue	(R)	B	(G)	R	(B)		R G B



Note 1: After WA=red, Q=green, NT and SA both have single values. This eliminates branching.

Note 2: After WA=red, Q=green, there is an inconsistency between NT, SA, but it is not noticed.

Note 3: After V=blue, an inconsistency is detected

# AC3 Algorithm

AC-3 propagates inferences

**function** AC-3(*csp*) **returns** false if an inconsistency is found and true otherwise

**inputs:** *csp*, a binary CSP with components (*X*, *D*, *C*)

**local variables:** *queue*, a queue of arcs, initially all the arcs in *csp*

**while** *queue* is not empty **do**

    ( $X_i$ ,  $X_j$ )  $\leftarrow$  REMOVE-FIRST(*queue*)

**if** REVISE(*csp*,  $X_i$ ,  $X_j$ ) **then**

**if** size of  $D_i = 0$  **then return** false

**for each**  $X_k$  **in**  $X_i$ .NEIGHBORS -  $\{X_j\}$  **do**

            add ( $X_k$ ,  $X_i$ ) to *queue*

**return** true

**function** REVISE(*csp*,  $X_i$ ,  $X_j$ ) **returns** true iff we revise the domain of  $X_i$

*revised*  $\leftarrow$  false

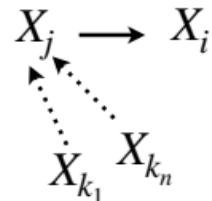
**for each**  $x$  **in**  $D_i$  **do**

**if** no value  $y$  in  $D_j$  allows  $(x,y)$  to satisfy the constraint between  $X_i$  and  $X_j$  **then**

            delete  $x$  from  $D_i$

*revised*  $\leftarrow$  true

**return** *revised*



Returns an arc consistent binary constraint graph or false because a variable domain is empty (and thus no solution)

# Version of AC-3 in Python

```
def ac3(csp):
    queue = fifo_queue()
    init_queue(csp,queue)
    domains = dict(csp.domains)

    def revise(xi,xj):
        revised = False
        for con in csp.constraints:
            constraint = [con for con in csp.constraints if con.scope == (xi,xj)][0]
        for value in domains[xi]:
            consistent = any(constraint.holds({xi:value,xj:val}) for val in domains[xj])
            if not consistent:
                domain = list(domains[xi])
                domain.remove(value)
                domains[xi] = set(domain)
                revised = True
        return revised

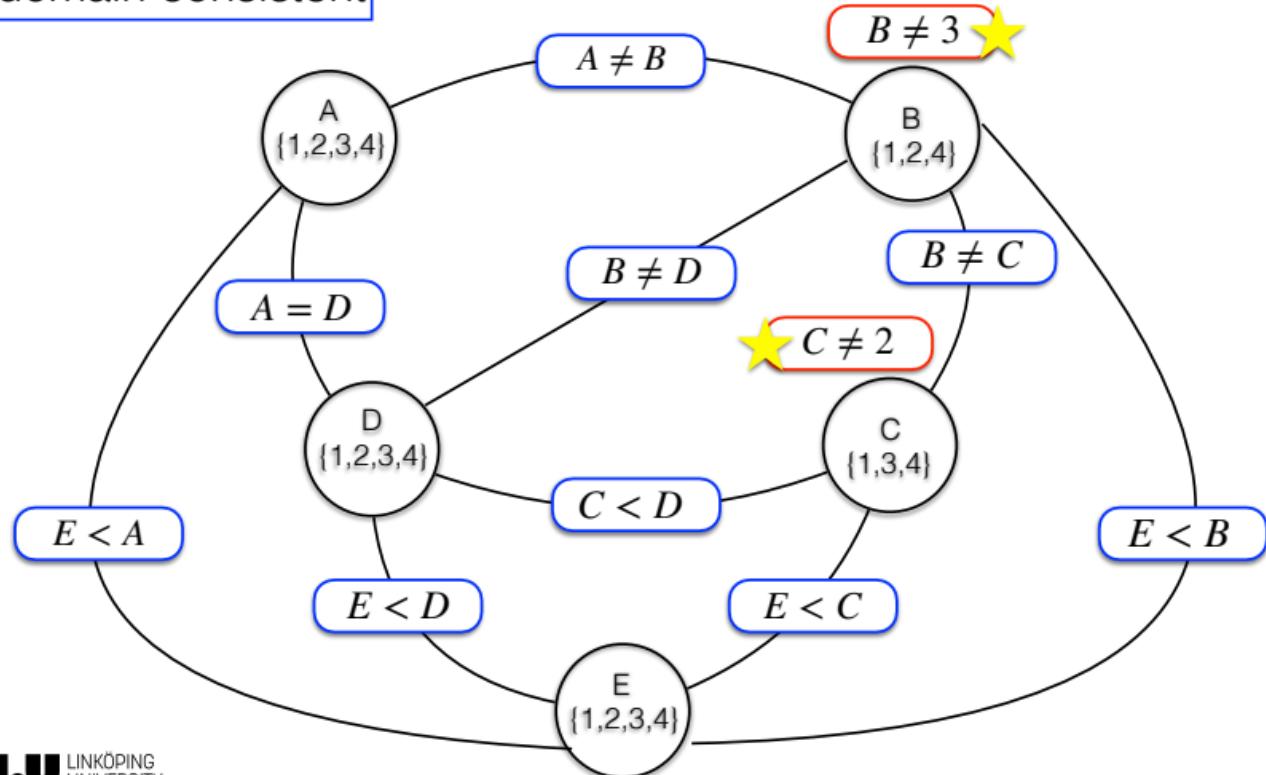
    def neighbors(xi,xj):
        neighbors = [con.scope[0] for con in csp.constraints if con.scope[1] == xi and not con.scope[0] == xj]
        return neighbors

    while queue.queue():
        arc_ij = queue.pop()
        xi = arc_ij[0]
        xj = arc_ij[1]
        if revise(xi,xj):
            if len(csp.domains[xi]) == 0:
                return [False,None]
            for xk in neighbors(xi,xj):
                queue.insert((xk,xi))

    return [True,domains]
```

# Robot Delivery Example

First make  
domain consistent



# Apply AC-3

Initial Queue

<del>{1, 2, 3, 4} : ('A', 'B') : {1, 2, 4}</del>
<del>{1, 2, 3, 4} : ('A', 'D') : {1, 2, 3, 4}</del>
<del>{1, 2, 3, 4} : ('A', 'E') : {1, 2, 3, 4}</del>
<del>{1, 2, 4} : ('B', 'A') : {1, 2, 3, 4}</del>
<del>{1, 2, 4} : ('B', 'C') : {1, 3, 4}</del>
<del>{1, 2, 4} : ('B', 'D') : {1, 2, 3, 4}</del>
<del>{1, 2, 4} : ('B', 'E') : {1, 2, 3, 4}</del>
<del>{1, 3, 4} : ('C', 'B') : {1, 2, 4}</del>
<del>{1, 3, 4} : ('C', 'D') : {1, 2, 3, 4}</del>
<del>{1, 3, 4} : ('C', 'E') : {1, 2, 3, 4}</del>
<del>{1, 2, 3, 4} : ('D', 'A') : {1, 2, 3, 4}</del>
<del>{1, 2, 3, 4} : ('D', 'B') : {1, 2, 4}</del>
<del>{1, 2, 3, 4} : ('D', 'C') : {1, 3, 4}</del>
<del>{1, 2, 3, 4} : ('D', 'E') : {1, 2, 3, 4}</del>
<del>{1, 2, 3, 4} : ('E', 'A') : {1, 2, 3, 4}</del>
<del>{1, 2, 3, 4} : ('E', 'B') : {1, 2, 4}</del>
<del>{1, 2, 3, 4} : ('E', 'C') : {1, 3, 4}</del>
<del>{1, 2, 3, 4} : ('E', 'D') : {1, 2, 3, 4}</del>

Final Result:

{'A': {4}, 'B': {2}, 'C': {3}, 'D': {4}, 'E': {1}}

{2, 3, 4} : ('A', 'B') : {2, 4}
{1, 2, 4} : ('B', 'C') : {1, 3, 4}
{2, 4} : ('B', 'D') : {2, 3, 4}
{3, 4} : ('C', 'D') : {2, 3, 4}

Note: Domains are not part of the  
Arch queue.

# Path Consistency

```
{r,g,b} WA --> NT {r,g,b}  
{r,g,b} NT --> WA {r,g,b}  
{r,g,b} WA --> SA {r,g,b}  
{r,g,b} SA --> WA {r,g,b}  
{r,g,b} NT --> SA {r,g,b}  
{r,g,b} SA --> NT {r,g,b}
```

Note that arc consistency does not help us out for the map coloring problem!

It only looks at pairs of variables

## Definition

A two variable set  $\{X_i, X_j\}$  is path consistent with respect to a 3rd variable  $X_m$  if, for every assignment  $\{X_i=a, X_j=b\}$  consistent with the constraints on  $\{X_i, X_j\}$ , there is an assignment to  $X_m$  that satisfies the constraints on  $\{X_i, X_m\}$  and  $\{X_m, X_j\}$ .



# K-Consistency

A CSP is **k-consistent** if, for any set of **k-1** variables and for any consistent assignment to those variables, a consistent value can always be found for the **kth** variable.

**1-consistency:** node consistency

**2-consistency:** arc consistency

**3-consistency:** path consistency

A CSP is strongly k-consistent if it is k-consistent and is also k-1 consistent, k-2 consistent, ...., 1-consistent.

In this case, we can find a solution in  $O(n^2d)!$  but establishing n-consistency takes time exponential in n in the worst case and space exponential in n!

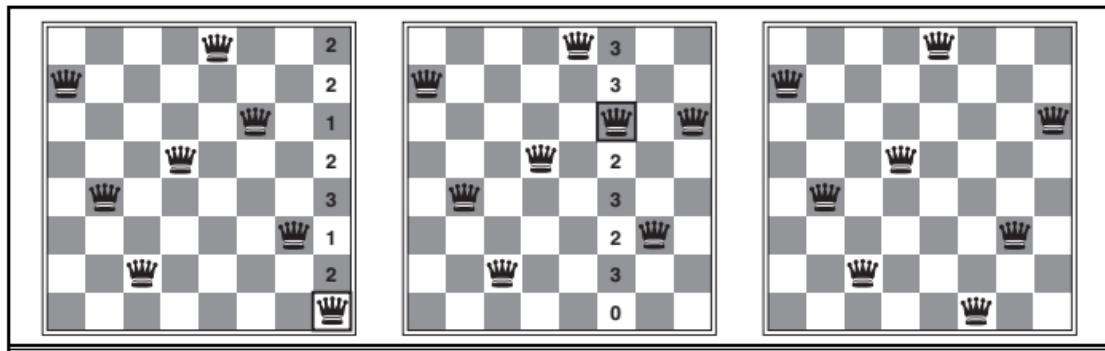
# Local Search for CSPs

```
function MIN-CONFLICTS(csp, max_steps) returns a solution or failure
  inputs: csp, a constraint satisfaction problem
            max_steps, the number of steps allowed before giving up

  current  $\leftarrow$  an initial complete assignment for csp
  for i = 1 to max_steps do
    if current is a solution for csp then return current
    var  $\leftarrow$  a randomly chosen conflicted variable from csp.VARIABLES
    value  $\leftarrow$  the value v for var that minimizes CONFLICTS(var, v, current, csp)
    set var = value in current
  return failure
```

**Conflicts(var, v, current, csp)** - Counts the number of constraints violated by a particular value, given the rest of the current assignment.

# A Local Search CSP Example



- At each step a queen is chosen for reassignment in its column
- The number of conflicts is shown in each square (# of attacking queens for a square)
- Move the queen to the min-conflicts square
- Above: A two step solution

For n-queens: runtime is independent of problem size!!!!  
Can solve million queens problem in an average of 50 steps!!!

Hubble Space Telescope: Reduced time to schedule a week of observations  
from three weeks to 10 minutes!!