# BIM309 Artificial Intelligence

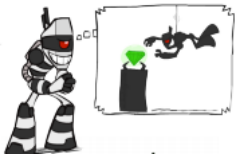
## Week 8 – Constraint Satisfaction Problems (CSP)

## Outline

- Constraint Satisfaction Problems (CSPs)
- Crypt-arithmetic Puzzle
- Backtracking
- Constraint Propagation
- Problem Structure

# Recall

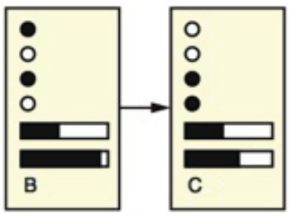
**Planning:** sequences of actions



- **Search Problems**
  - Find the **sequence of actions** that leads to the goal
  - Sequence of actions means a **path** in the search space
  - The **path to the goal** is the important thing
  - Paths come with different costs and depths
  - We use "rules of thumb" aka **heuristics** to guide the search efficiently
- **Constraint Satisfaction Problems (CSP)**
  - A search problem too! But, a special subset of search problems
  - We care about the **goal itself**, ***not the path***
  - CSPs are specialized for identification problems

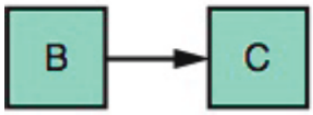
**Identification:** assignments to variables

# Search vs. CSPs




## Search Problems

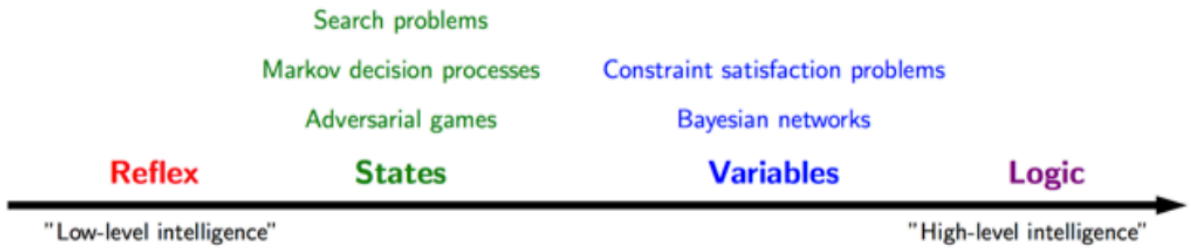
- A ***state*** is a **black box**, implemented as some data structure that supports successor function, heuristic function and goal test
- Recall **atomic representation**
- A **goal test** is any function over the states




## CSP Problems

- A ***state***: defined by variables $X_i$ with values from domain $D_i$ → attribute-value pairs
- Recall **factored representation**
- A **goal test** is ***a set of constraints*** specifying ***allowable combinations*** of values for subsets of variables

## *Recall:* The Intelligence Level Graph




**Credit: Courtesy Percy Liang**

## CSPs Definition

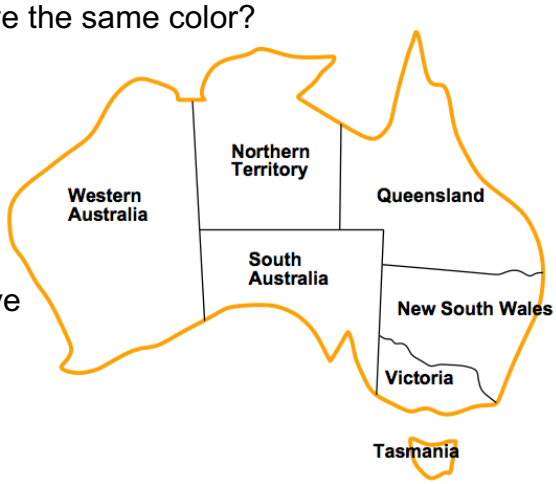
- A constraint satisfaction problem (CSP) consists of **three elements**:
  - A set of **variables**, $X = \{X_1, X_2, \ldots, X_n\}$
  - A set of **domains** (possible values) for each variable: $D = \{D_1, D_2, \ldots, D_n\}$
  - A set of **constraints** *C* that specify allowable combinations of values
- Solving the CSP: **finding variable assignment(s)** that **satisfy all constraints**
- Concepts: problem formalization, backtracking search, arc consistency, etc.
- We call "**a solution**" → a **consistent assignment**
  - Why consistent? – find the assignment that does not violate the constraints

# Example: Map Coloring

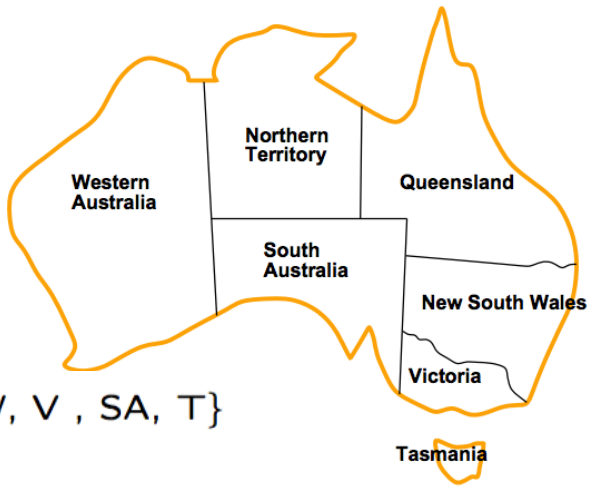
***Question:*** How can we color each of the 7 provinces {**red**, **green**, **blue**} so that no two neighboring provinces have the same color?

***Task:*** Color each region {**red**, **green**, **blue**}

***Constraint:*** No two adjacent regions have the same color




# Example: Map Coloring




**Variables:** X ={WA, NT, Q, NSW, V , SA, T}

**Domains:** $D_i$ = {red, green, blue}

**Constraints:** adjacent regions must have different colors;

e.g., WA ≠ NT or (W A, N T ) ∈ {(red, green), (red, blue), etc..}

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

# Example: Map Coloring

***One Possible Solution***


**Example:**

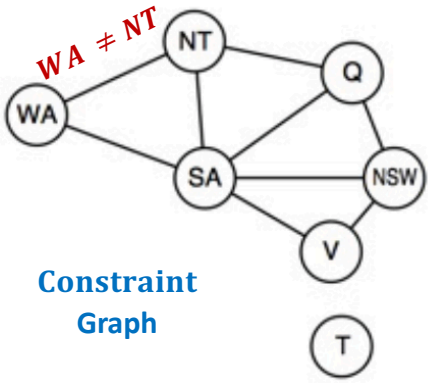
{WA=red, NT=green, Q=red, NSW=green,V=red, SA=blue,T=green}

# Real-World CSPs

- Assignment problems (e.g., who teaches what class?)
- Timetabling problems (e.g., which class is offered when and where?)
- Hardware configuration
- Spreadsheets
- Transportation scheduling
- Factory scheduling
- Floor planning
- Notice that many real-world problems involve real-valued variables, rather than discrete variables such as the color problem in the map coloring

# Constraint Graph

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


- **Binary CSP:** each constraint relates at most two variables
- **Binary Constraint Graph:** nodes are variables, arcs (edges) show constraints
- **CSP algorithms:** use the graph structure to speed up search
  - E.g., Tasmania is an independent sub-problem!

# Varieties of Variables

- **Discrete Variables**
  - **Finite Domains**
    - Assume ***n*** variables, ***d*** values, then the number of complete assignments is $\boldsymbol{O(d^n)}$
    - E.g., map coloring, 8-queens problem
  - **Infinite Domains** (integers, strings, etc.)
    - Cannot enumerate all combinations of values
    - Need to use a constraint language
    - E.g., job scheduling (variables are start/end days for each job) - $T_1 + d \leq T_2$
- **Continuous Variables**
  - Common in operations research
  - Linear programming problems with linear or non-linear equalities/inequalities

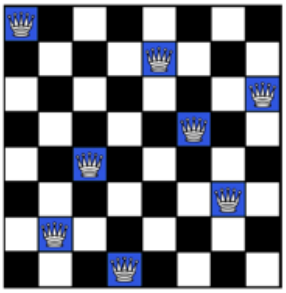
# Varieties of Constraints

Constraints may have different scope:

- **Unary constraints:** constraint on a single variable, e.g., $SA \neq green$
- **Binary constraints:** constraints between pairs of variables, e.g., $SA \neq WA$
- **Global constraints:** involve 3 or more variables, e.g., ***Alldiff*** constraint specifies that all variables must have different values (e.g. of this kind of constraint involves, *crypt-arithmetic puzzles*, *Sudoku*)
- **Preferences (soft constraints):**
  - *Example:* red is better than green
  - Each variable assignment is associated with a **cost**
  - *Goal*: Find the assignment with the lowest overall cost that is consistent with all constraints
  - Involves problems such as "*constrained optimization problems*"

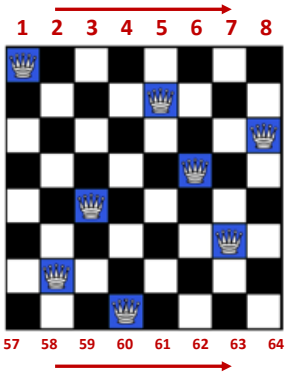
# Example: 8-Queen Problem

***8-Queen:*** Place 8 queens on an 8x8 chess board so no queen can attack another one



- Can treat this as a CSP
  - What are the variables, domains, and constraints?

# Example: 8-Queen CSP




***8-Queen:*** Place 8 queens on an 8x8 chess board so no queen can attack another one

- **Problem Formulation 1:**
  - **Variables:** 1 variable per queen

$$\{Q_1, Q_2, Q_3, Q_4, Q_5, Q_6, Q_7, Q_8\}$$

  - **Domains:** Each variable could have a value between 1 and 64

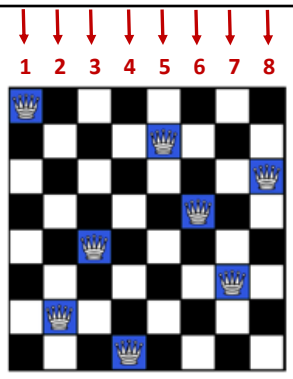
$$1 \leq Q_n \leq 64$$

  - **Solution:**

$$Q_1 = 1, Q_2 = 13, Q_3 = 24, \dots, Q_8 = 60$$

  - **Constraints?**

# Example: 8-Queen CSP




***8-Queen:*** Place 8 queens on an 8x8 chess board so no queen can attack another one

- **Problem Formulation 2:**
  - **Variables:** 1 variable per queen

$$\{Q_1, Q_2, Q_3, Q_4, Q_5, Q_6, Q_7, Q_8\}$$

  - **Domains:** Each variable could have a value between 1 and 8 (Column for the queen in the n-th row)

$$1 \leq Q_n \leq 8$$

  - **Solution:**

$$Q_1 = 1, Q_2 = 7, Q_3 = 5, \dots, Q_8 = 3$$

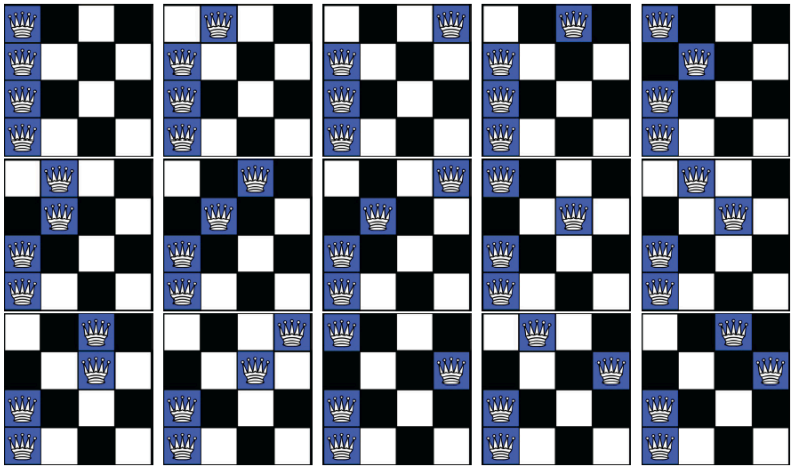
  - **Constraints:**

$$Q_1 \neq Q_2, Q_1 \neq Q_3, Q_1 \neq Q_4, \dots, Q_7 \neq Q_8$$

diagonals: $|i - j| \neq |Q_i - Q_j|$

# Brute Force?

Should we simply generate and test all configurations?



...

- **4-Queen Problem**
  $4x4x4x4 = 4^4 = 256$

***Is this brute force search scalable?***

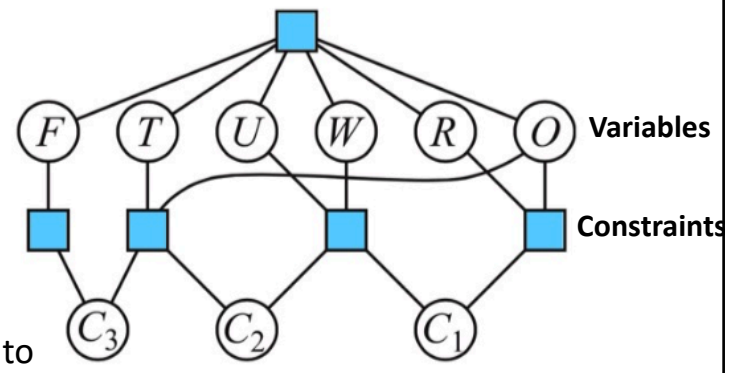
- **8-Queen Problem**
  $8x8x8x8x8x8x8x8 = 8^8 = 16.7\ million$
- **16-Queen Problem**
  $16^{16} \approx 10^{20}$

# Example: Crypt-arithmetic

$C_3\ C_2\ C_1$

$$\begin{array}{r} T\ W\ O \\ +\ T\ W\ O \\ \hline F\ O\ U\ R \end{array}$$

***Alldiff***




**Hypergraph**

**Crypt-arithmetic Puzzle:** Problem is to find a substitution of letters with digits

- There's ten digits, such that we have the arithmetic sum here correct
- We want to find substitutions for T, W, O, F, U, and R

# Example: Crypt-arithmetic

$C_3\ C_2\ C_1$

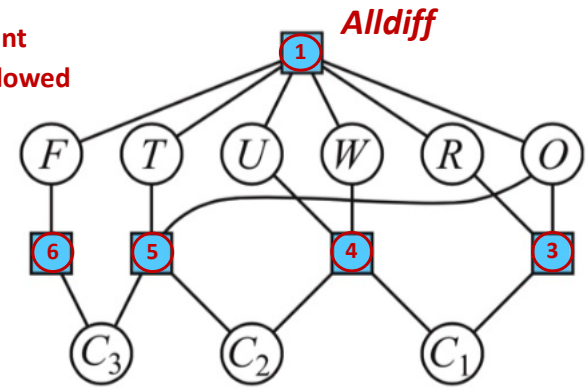
$$\begin{array}{cccc} & T & W & O \\ + & T & W & O \\ \hline F & O & U & R \end{array} \qquad \begin{array}{c} 734 \\ 734 \\ \\ 1468 \end{array}$$

**Variables:** $X = \{F, T, U, W, R, O, C_1, C_2, C_3\}$

**Domain:** $D = \{0, 1, 2, \cdots, 9\}$

**Constraints:**

1. Alldiff$(F, T, U, W, R, O)$ — n-ary constraint
2. $T \neq 0, \quad F \neq 0$ — leading zeros are not allowed
3. $O + O = R + 10 * C_1$
4. $C_1 + W + W = U + 10 * C_2$
5. $C_2 + T + T = O + 10 * C_3$
6. $C_3 = F$




# Solving CSPs

- **State-space search algorithms: search!**
- **CSP Algorithms:** Algorithm can do two things:
  - **Search:** choose a new variable assignment from many possibilities (e.g. backtracking search)
  - **Inference: constraint propagation**, *use the constraints to spread the word*: reduce the number of values for a variable which will reduce the legal values of other variables etc.
- Inference can be used
  - As a preprocessing step - constraint propagation sometimes can solve the problem entirely without search
  - But, constraint propagation can also be intertwined with search in more difficult problems

# Solving CSPs

- **BFS:** Develop the complete tree
- **DFS:** Fine, but time consuming
- **BTS: Backtracking search** is the basic uninformed search for CSPs It's a **DFS** with these two improvements
  1. ***Assign one variable at a time:***
     - Variable assignments are commutative, so fix ordering
     - E.g., (WA=red then NT=green) is same as (NT=green then WA=red)
     - Only need to consider assignments to a single variable at each step
  2. ***Check constraints on the go:***
     - consider values that do not conflict with previous assignments

***DFS with these two improvements is called BTS***

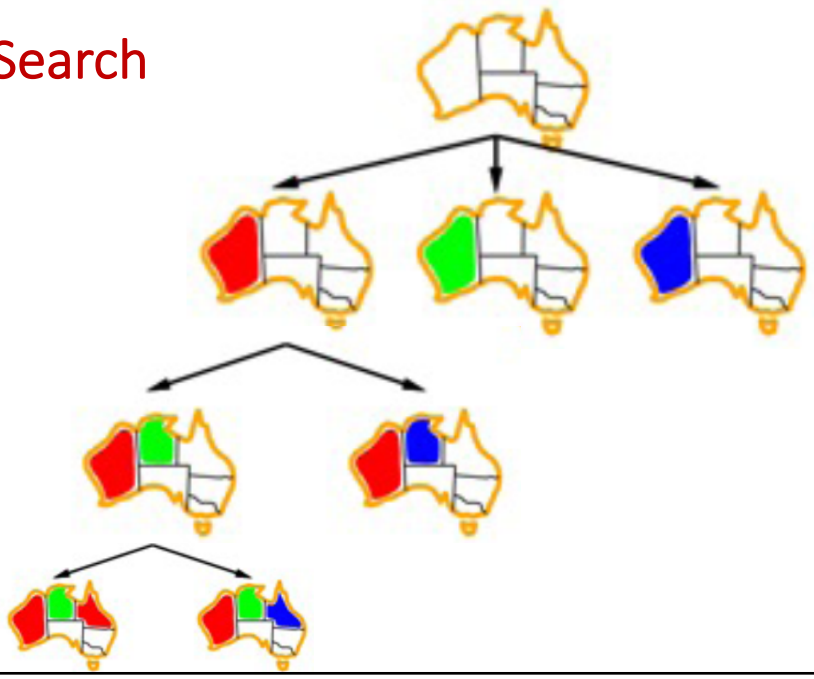
***Typically, CSPs is a DFS with some backtracking search, which means that we're going to step back whenever we make an assignment that violates the previous assignments***

# Solving CSPs

*Standard search formulation of CSPs*

- **Initial state:** empty assignment {} - we did not assign any value to variables
- **States:** defined by the values assigned so far (partial assignments)
  - E.g., providing some variables with digits in the crypt-arithmetic puzzle
- **Successor function:** assign a value to an unassigned variable *(var=value)* that does not conflict with current assignment
  - Fails if no legal assignments
- **Goal test:** the current assignment is complete and satisfies all constraints – complete and consistent

## Backtracking Search



## Improving BTS

**Heuristics are back!**

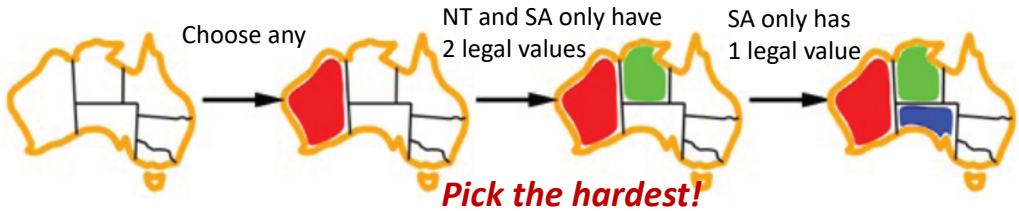
1. **Which variable should be assigned next?** *(ordering)*
2. **In what order should its values be tried?** *(ordering)*
3. **Can we detect inevitable failure early?** *(filtering)*

**Use General Purpose Heuristics, which give huge gains in speed**

# Minimum Remaining Values

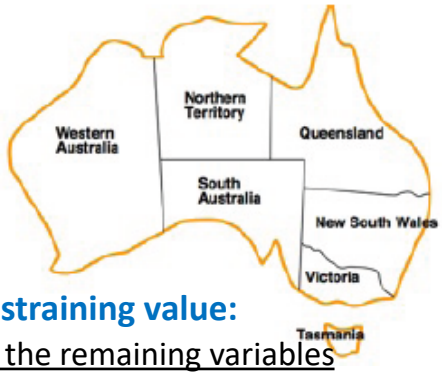

1. **Which variable should be assigned next?**

- **MRV:** Choose the variable with the **fewest legal values in its domain**
- This is the variable that is most likely to cause failure. Fail first!




***Pick the hardest!***

# Least Constraining Values

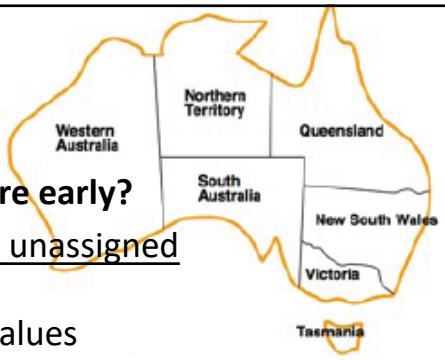



2. **In what order should its values be tried?**

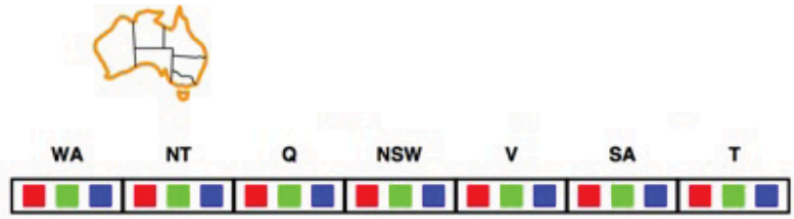
- **LCV:** Given a variable, **choose the least constraining value:**
  - The one that rules out the <u>fewest values in the remaining variables</u>


***Pick the ones that are likely to work!***

# Forward Checking

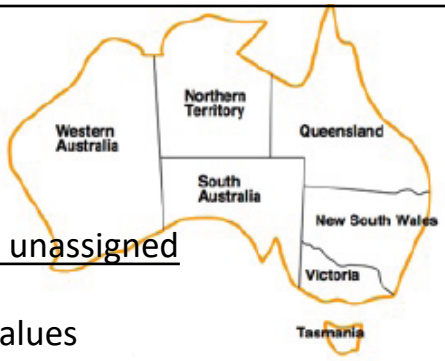



3. **Can we detect inevitable/unavoidable failure early?**
   - **FC:** Keep track of remaining legal values for unassigned variables (using immediate constraints)
   - Terminate when any variable has no legal values

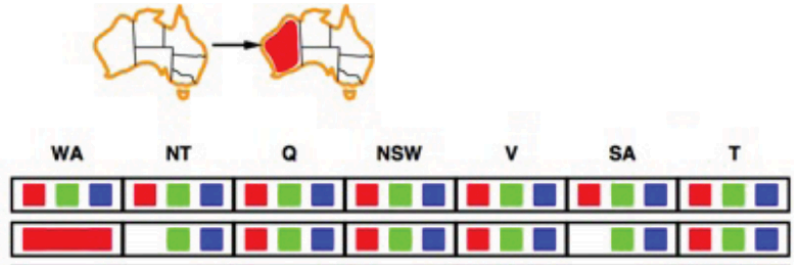



*Helps to detect failure early*

# Forward Checking

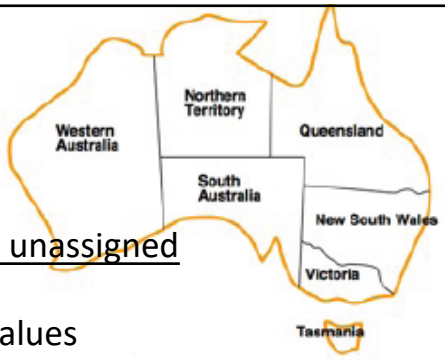



3. **Can we detect inevitable failure early?**
   - **FC:** Keep track of remaining legal values for unassigned variables (using immediate constraints)
   - Terminate when any variable has no legal values




*Helps to detect failure early*

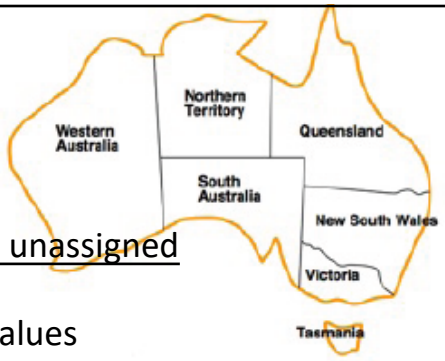
# Forward Checking



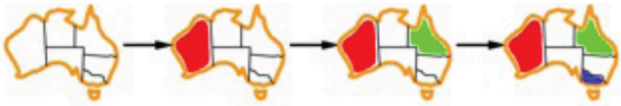

3. **Can we detect inevitable failure early?**
   - **FC:** Keep track of remaining legal values for unassigned variables (using immediate constraints)
   - Terminate when any variable has no legal values

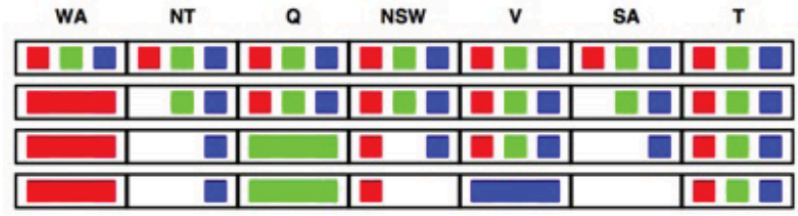

*Helps to detect failure early*

# Forward Checking




3. **Can we detect inevitable failure early?**
   - **FC:** Keep track of remaining legal values for unassigned variables (using immediate constraints)
   - Terminate when any variable has no legal values






*Helps to detect failure early*

# Backtracking Search

***Input*** **is CSP formalization**, which means the domain, the variables, and the constraints

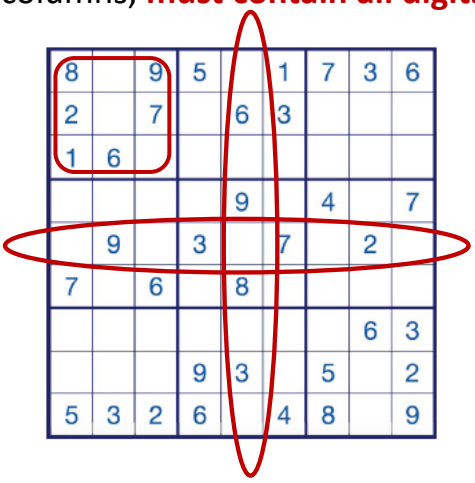
```
function BACKTRACKING_SEARCH(csp) returns a solution, or failure
    return BACKTRACK({}, csp)

function BACKTRACK(assignment, csp) returns a solution, or failure
    if assignment is complete then return assignment
    var = SELECT_UNASSIGNED-VARIABLE(csp)
    for each value in ORDER_DOMAIN_VALUES (var, assignment, csp)
        if value is consistent with assignment then
            add {var = value} to assignment
            result = BACKTRACK(assignment, csp)
            if result ≠ failure then return result
        remove {var = value} from assignment
    return failure
```

**Pick a variable – one at a time**

**Backtracking = DFS + variable-ordering + fail-on-violation**

# Solving CSPs: Sudoku

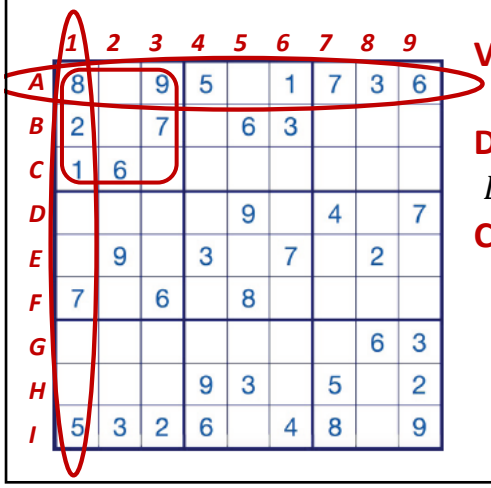
All 3x3 boxes, rows, columns, **must contain all digits 1…9**




**← units**

# Solving CSPs: Sudoku

All 3x3 boxes, rows, columns, **must contain all digits 1…9**




**Variables:**

$$V = \{A_1, \dots, A_9, B_1, \dots, B_9, I_1, \dots, I_9\}, |V| = 81$$

**Domain:**

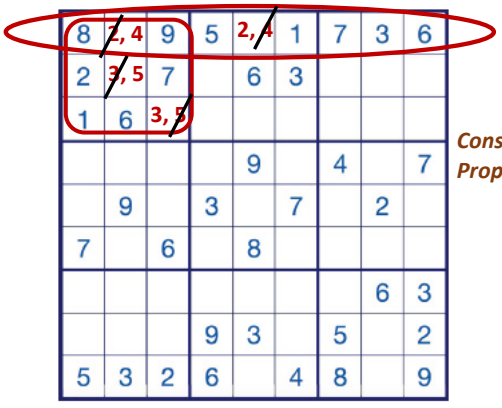
$D = \{1, 2, \dots, 9\}$, the filled squares have a single value

**Constraints:** 27 constraints

- Alldiff$(A_1, A_2, A_3, A_4, A_5, A_6, A_7, A_8, A_9)$
  … (8 more)
- Alldiff$(A_1, B_1, C_1, D_1, E_1, F_1, G_1, H_1, I_1)$
  … (8 more)
- Alldiff$(A_1, A_2, A_3, B_1, B_2, B_3, C_1, C_2, C_3)$
  … (8 more)

# Solving CSPs: Sudoku

All 3x3 boxes, rows, columns, **must contain all digits 1…9**



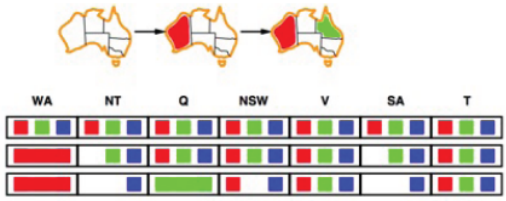

*Constraint Propagation*

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

- **Naked doubles (triples):** find two (three) cells in a 3x3 grid that have only the same candidates left, eliminate these two (three) values from all possible assignments in that box
- Locked pair, Locked triples, etc.

# Constraint Propagation

- Forward checking propagates information from **assigned to adjacent unassigned variables, but doesn't detect more distant failures**
- Observe:




  - **Forward checking** does not check interaction between unassigned variables!
  - Here **SA** and **NT**! (They both must be blue, but they cannot both be blue!)
  - Why didn't we detect this yet?
- Forward checking improves backtracking search but does not look very far in the future, hence does not detect all failures, and sometimes can make mistakes
  - There is no propagation between unassigned variables
- We use **constraint propagation**, reasoning from constraint to constraint
  - *Constraint propagation* repeatedly enforces constraints (locally)
  - E.g., arc consistency test

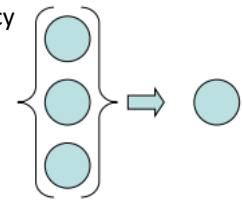
# Types of Consistency

Constraint propagation aims to make the graph, or the CSP problem, consistent.
But there are different kinds of consistency – different levels

- **Node-consistency** (or **1-Consistency**) - (***unary constraints***): Each single node's domain has a value which meets that node's unary constraints
  - E.g, if we want Tasmania to be absolutely not green, add the unary constraint $T \neq green$ , and use that constraint to make the node Tasmania consistent
  - If each single node in the CSP graph is node-consistent, which means cannot take any forbidden value → Node-consistency in CSP

- **Arc-consistency** (or **2-Consistency**) - (***binary constraints***): is the consistency between edges. For each pair of nodes, any consistent assignment to one can be extended to the other
  - To make a graph arc-consistent, check all possible edges for their arc-consistency status, if they are all arc-consistent, → call the problem arc-consistent

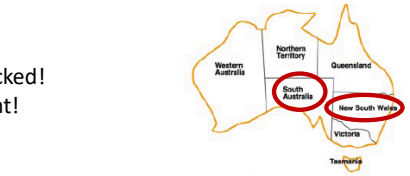
- **Path-consistency** (or **n-Consistency**) - (***n-ary constraints***): generalizes arc-consistency from binary to multiple constraints. For each n nodes, any consistent assignment to k-1 can be extended to the kth node
- Higher ***n*** more expensive to compute
- ***Note:*** It is always possible to transform all n-ary constraints into binary constraints
  - Often, CSPs solvers are designed to work with binary constraints



# Arc Consistency

- **AC:** Simplest form of propagation makes each arc consistent
- $X \rightarrow Y$ is consistent **IFF** for every value $x$ of X, there is some allowed $y$

***Important:*** If *X* loses a value, neighbors of *X* need to be rechecked! Must rerun after each assignment!

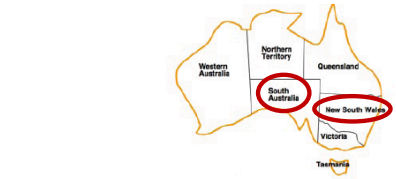



**SA → NSW**

*SA → NSW* is consistent since *SA=blue* and *NSW=red* is a consistent assignment.


# Arc Consistency

- **AC:** Simplest form of propagation makes each arc consistent
- $X \rightarrow Y$ is consistent IFF for every value $x$ of X, there is some allowed $y$

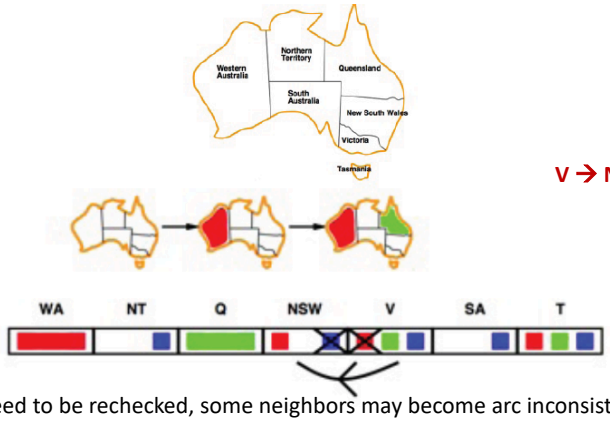



**NSW → SA**

*NSW → SA* is not consistent since for *NSW=blue* there is no consistent assignment to *SA*.
Arc can be made consistent by removing *blue* from *NSW*

# Arc Consistency

- **AC:** Simplest form of propagation makes each arc consistent
- $X \rightarrow Y$ is consistent IFF for every value $x$ of X, there is some allowed $y$



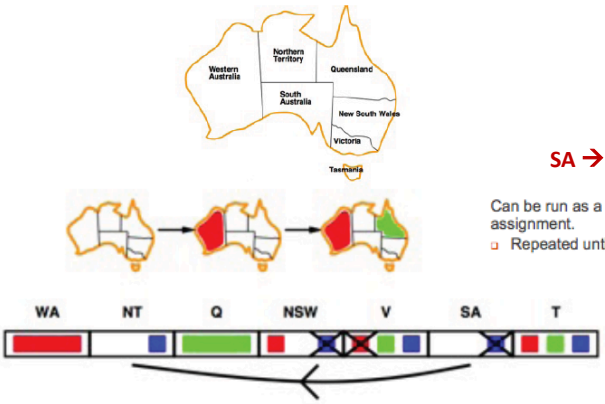

V → NSW

RECHECK neighbours !!
- Remove red from *V*

If X loses a value, neighbors of X need to be rechecked, some neighbors may become arc inconsistent

# Arc Consistency

- **AC:** Simplest form of propagation makes each arc consistent
- $X \rightarrow Y$ is consistent IFF for every value $x$ of X, there is some allowed $y$




SA → NT

Can be run as a preprocessing before the search or after each assignment.
- Repeated until no inconsistency remains

# Arc Consistency

```
function AC-3( csp) returns the CSP, possibly with reduced domains
  inputs: csp, a binary CSP with variables {X_1, X_2, ..., X_n}
  local variables: queue, a queue of arcs, initially all the arcs in csp

  while queue is not empty do
    (X_i, X_j) ← REMOVE-FIRST(queue)
    if REMOVE-INCONSISTENT-VALUES(X_i, X_j) then
      for each X_k in NEIGHBORS[X_i] do
        add (X_k, X_i) to queue
```

```
function REMOVE-INCONSISTENT-VALUES( X_i, X_j) returns true iff succeeds
  removed ← false
  for each x in DOMAIN[X_i] do
    if no value y in DOMAIN[X_j] allows (x,y) to satisfy the constraint X_i ↔ X_j
      then delete x from DOMAIN[X_i]; removed ← true
  return removed
```

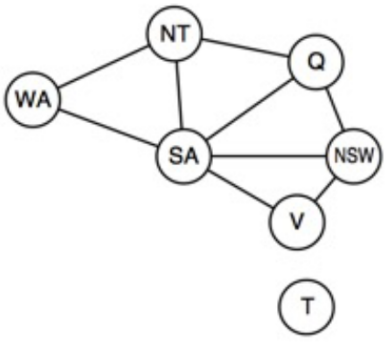
Algorithm that makes a CSP arc-consistent!

Time complexity: $O(n^2 d^3)$

# Backtracking w / 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 = SELECT_UNASSIGNED-VARIABLE(csp)
    for each value in ORDER_DOMAIN_VALUES (var, assignment, csp)
        if value is consistent with assignment then
            add {var = value} to assignment
            inferences = INFERENCE(csp, var, value)
            if inferences ≠ failure then
                add inferences to assignment
                result = BACKTRACK(assignment, csp)
                if result ≠ failure then return result
        remove {var = value} and inferences from assignment
    return failure
```

# Problem Structure




- **Idea:** Leverage the problem structure to make the search more efficient
- ***Example:*** Tasmania is an independent problem
- Identify the connected component of a graph constraint
- Work on independent sub-problems, which can reduce the complexity a lot

# Problem Structure

**Complexity**

- Let $\boldsymbol{d}$ be the size of the domain and $\boldsymbol{n}$ be the number of variables
- Time complexity for BTS is $\boldsymbol{O(d^n)}$
- Suppose we **decompose into sub-problems**, with $c$ variables per sub-problem
  - Then we have $\frac{\boldsymbol{n}}{\boldsymbol{c}}$ sub-problems
  - $c$ variables per sub-problem takes $\boldsymbol{O(d^c)}$
  - The total for all sub-problems takes $\boldsymbol{O\left(\frac{n}{c}d^c\right)}$ in the worst case

## Example: Problem Structure

- Assume $n$ = 80, $d$ = 2
- Assume we can decompose into 4 sub-problems with $c$ = 20
- Assume processing at 10 million nodes per second

- Without decomposition of the problem we need:

$O(d^c)$ $2^{80} = 1.2 \times 10^{24}$ $nodes$ → when divided by 10 million = **3.83 million years!**

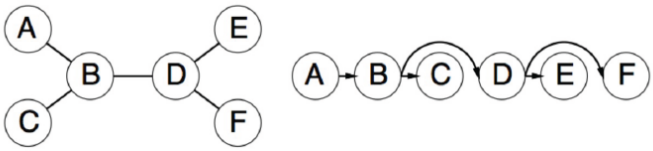
- With decomposition of the problem we need:

$O\left(\frac{n}{c}d^c\right)$ $4 \times 2^{20} = 4.2 \times 10^{6}$ $nodes$ → when divided by 10 million = **0.4 seconds!**

## Problem Structure

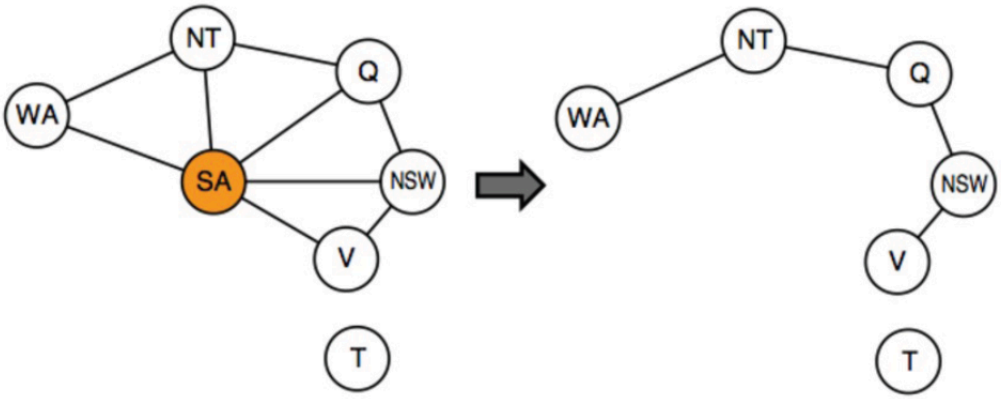
- Turning a problem into independent sub-problems is not always possible
- Can we leverage other graph structures?
  - Yes, if the graph is tree-structured or nearly tree-structured
  - A graph is a **tree** if any two variables are connected by **only one path**
- **Idea:** use DAC (Directed Arc Consistency)
- A CSP is said to be **directed arc-consistent** under an ordering $X_1, X_2, \ldots, X_n$ IFF every $X_i$ is arc-consistent with each $X_j$ for $j > i$

# Problem Structure




- First pick a variable to the be the root
- Do a **topological sorting**: choose an ordering of the variables, *s.t.* each variable appears after its parent in the tree
- For $n$ nodes, we have $n-1$ edges
- Make the tree directed arc-consistent takes $\boldsymbol{O(n)}$
- Each consistency check takes up to $\boldsymbol{O(d^2)}$ (compare d possible values for 2 variables)
- The CSP can be solved in $\boldsymbol{O(nd^2)}$ - linear complexity

# Nearly tree-structured CSPs




- Assign a variable or a set of variables and prune all the neighbors domains
- This will turn the constraint graph into a tree :)
- Assign {SA= x }
  - Remove any values from the other variables that are inconsistent
  - The selected value for SA could be wrong: have to try all of them

## Summary

- CSPs are a special kind of search problems:
  - states defined by values of a fixed set of variables
  - goal test defined by constraints on variable values
  - assign values to the variables without violating the set of constraints
- Backtracking = depth-first search with one variable assigned per node
- Variable ordering and value selection heuristics help
- Forward checking prevents assignments that guarantee later failure

## Summary

- Constraint propagation (e.g., arc consistency) is an important mechanism in CSPs
- It does additional work to constrain values and detect inconsistencies
- Tree-structured CSPs can be solved in linear time
- Further exploration: How can local search be used for CSPs?
- **The power of CSPs: domain-independent, that is you only need to define the problem and then use a solver that implements CSPs mechanisms**
- Play with CSP solver? Try ***http://aispace.org/constraint/***