Chapter 5

Constraint satisfaction problems (CSP). The common examples are Map coloring and Cryptarithmetic problems.

A CSP is defined by
- set of **variables** $X_1, X_2, \ldots, X_n$
- set of **constraint** $C_1, C_2, \ldots, C_m$

Each variable X_i has nonempty **domain** D_i of possible values.
A state of the problem is defined by an **assignment** of values to dome or all variables $\{X_i=v_i, X_j=v_j\}$.

An assignment that not violate any constraints is called **consistent** or legal assignment .

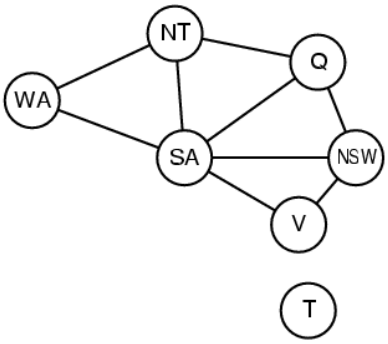
A **solution** to CSP is a complete assignment that satisfies all constraints. **Solution is more important than path to goal**

Map coloring problem

Given a map of Australia, color it using three colors (red , green, blue) such that no neighboring regions have the same color..

CSP Specification to Australia map coloring problem:
- **Variables**: *WA, NT, Q, NSW, V, SA, T*
- **Domains**: D_i*={red,green,blue}*
- **Constraints**:adjacent regions must have different colors.
  E.g. $WA \neq NT$

Constraint graph: nodes are variables, edges are constraints.



Types of Constraints

Hard Constraints:

– **Unary:** single variable, eg X>0

– **Binary**: pairs of variables, eg $X_1 > X_2$

– **Higher-Order**: 3 or more variables

Soft constraints: preferences represented by cost for each variable assignment

Cryptarithmetic Problems

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

Variables: $F, T, U, W, R, O, X_1, X_2, X_3$

Domains: $\{0,1,2,3,4,5,6,7,8,9\}$

Constraints: *All-differrent (F,T,U,W,R,O)*

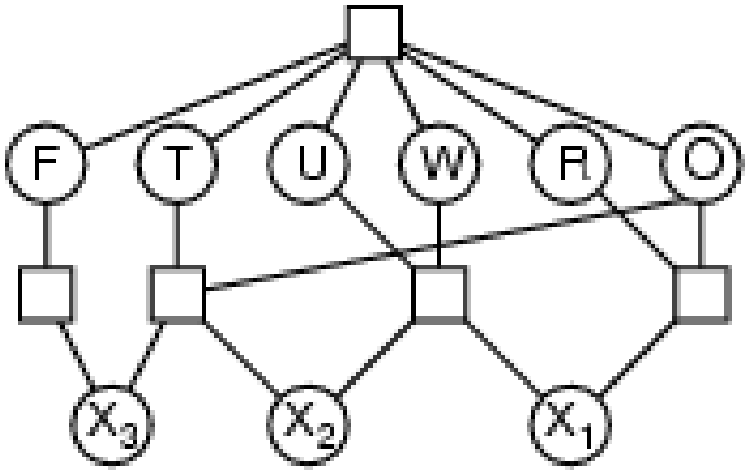
$O + O = R + 10 \cdot X_1$

$X1 + W + W = U + 10 \cdot X_2$

$X_2 + T + T = O + 10 \cdot X_3$

$X_3 = F, T \neq 0, F \neq 0$

Constraint hyper-graph : nodes are variables, squares are constraints, edges connect constriants to their variables.



Solving CSP using Relaxation

```
  S E N D
+ M O R E
----------
M O N E Y
```

To make the problem easier, suppose we are told that **E=5**.

```
M: [1,2,3,4,5,6,7,8,9]
S: [1,2,3,4,5,6,7,8,9]
O: [0,1,2,3,4,5,6,7,8,9]
E: [5]
N: [0,1,2,3,4,5,6,7,8,9]
R: [0,1,2,3,4,5,6,7,8,9]
D: [0,1,2,3,4,5,6,7,8,9]
Y: [0,1,2,3,4,5,6,7,8,9]
```

For carries, their possibilities for two-number addition are:

```
C1: [0,1]
C10: [0,1]
C100: [0,1]
```

Here are the constraints on the variables.

```
Con1: D + E = Y + (10 * C1)
Con2: N + R + C1 = E + (10 * C10)
Con3: E + O + C10 = N + (10 * C100)
Con4: S + M + C100 = O + (10 * M)
Con5: all_different(S,E,N,D,M,O,R,Y)
```

Since E=5 we can substitute that value into the constraints:

```
Con1: D + 5 = Y + (10 * C1)

Con2: N + R + C1 = 5 + (10 * C10)

Con3: 5 + O + C10 = N + (10 * C100)

Con4: S + M + C100 = O + (10 * M)

Con5: all_different (S,5,N,D,M,O,R,Y)
```

```
M: [1] (inactive)

S: [9] (inactive)

O: [0] (inactive)

E: [5] (inactive)

N: [0,1,2,3,4,5,6,7,8,9] (active)

R: [0,1,2,3,4,5,6,7,8,9] (active)

D: [0,1,2,3,4,5,6,7,8,9] (active)

Y: [0,1,2,3,4,5,6,7,8,9] (active)

C1: [0,1] (active)

C10: [0,1] (active)

C100: [0] (inactive)
```

And the sum looks like:

```
  9 5 N D
+ 1 0 R 5
_________
1 0 N 5 Y
```

The final solution is:

```
M=1, S=9, O=0, E=5, N=6, R=8, D=7, Y=2,

C1=1, C10=1, C100=0


  9 5 6 7

+ 1 0 8 5

_____________________

1 0 6 5 2
```

Solving CSP using Backtracking

Backtracking Search: DFS that chooses values for one variable at a time and backtracks when a variables has no legal values left to assign.

---

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

function RECURSIVE-BACKTRACKING(*assignment*, *csp*) **returns** a solution failure
  if *assignment* is complete **then return** *assignment*
  var ← SELECT-UNASSIGNED-VARIABLE(*Variables*[*csp*], *assignment*, *csp*)
  for each *value* **in** ORDER-DOMAIN-VALUES(*var*, *assignment*, *csp*) **do**
    if *value* is consistent with *assignment* according to Constraints[*csp*] **then**
      add { *var* = *value* } to *assignment*
      result ← RECURSIVE-BACKTRACKING(*assignment*, *csp*)
      if *result* ≠ *failue* **then return** *result*
      remove { *var* = *value* } from *assignment*
  return *failure*

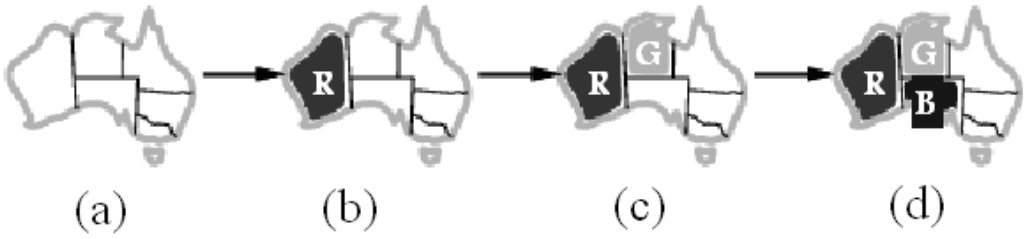
Three questions for Improving Backtracking Efficiency

1. Which variable (or value) should be assigned next?
2. In what order should its values be tried?
3. Can we detect inevitable failure early (and avoid the same failure in subsequent paths)?

1.Which variable (or value) should be assigned next?

Minimum remaining values (MRV)

choose the variable with the fewest legal values.

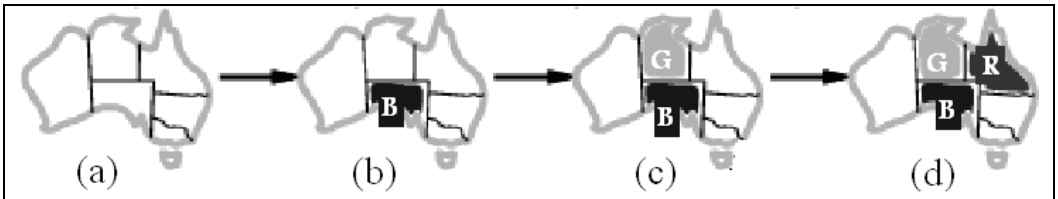


(a) (b) (c) (d)

(a) the original map
(b) all variables have 3 possible values➔ choose any,say WA
➔ WA= red
(c) NT and SA are MRV ➔ choose any , say NT
➔ NT = green
(d) SA is MRV ➔ SA = blue

Most constraining variable (MCV)

choose variable with most constraints on remaining variables



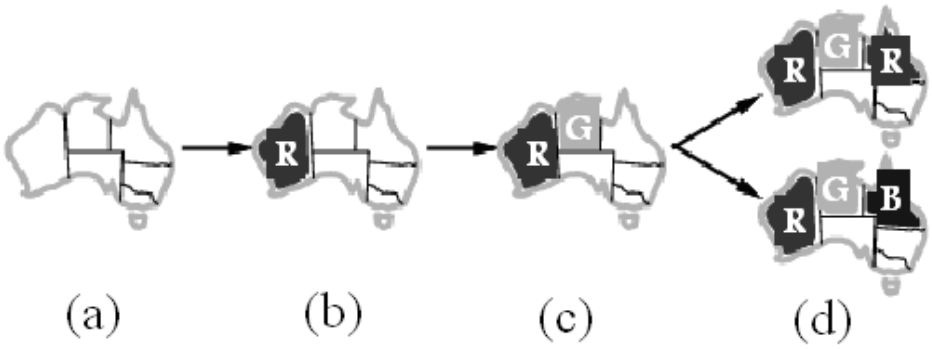
(a) (b) (c) (d)

(a) the original map
(b) from the constraint graph, MCV is SA (5 constraints)
➔ SA= blue
(c) NT, Q, NSW are MCV ➔ choose any , say NT
➔ NT = green
(d) Q, NSW are MCV ➔ choose any, say Q ➔ Q = red

2. In what order should its values be tried?

Least constraining value (LCV)

Given a variable, choose the least constraining value: the one that rules out the fewest values in the remaining variables



Assume that we shall assign variables in the following order :

WA, NT, Q, SA, NSW, V, T

Or we can use the Variable ordering heuristics (MRV or MCV)

(a) the original map

(b) WA can take any value from { red, green, blue}

➔ choose any , say red ➔ WA = red

(c) NT can take any value from {green, blue}

➔ choose any , say green ➔ NT = green

(d) Q can take any value from {red, blue}

➔ if red is choosen, this allows one value for SA (blue)

➔ however, if blue is choosen, No value left for SA.

➔ Q= red

And so on.

3. Can we detect inevitable failure early (and avoid the same failure in subsequent paths)?

Forward checking

Keep track of remaining legal values for unassigned variables, Terminate search when any variable has no legal values

(a) the original map,

(b) WA can take any value from { red, green, blue}

➔ choose any , say red ➔ WA = red

➔ remove red from neighbors {NT, SA}

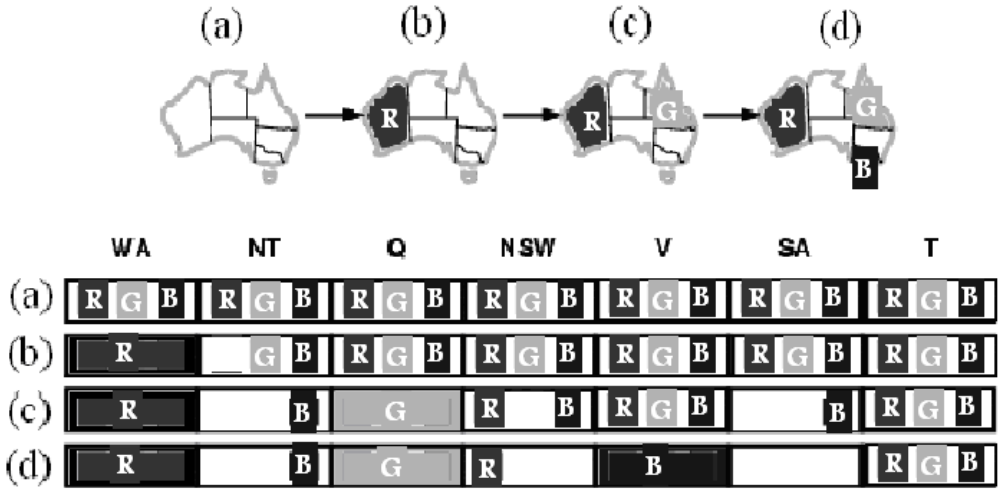
no variable has no values ➔ ok, continue

(c) assume that the next variable to assign is Q

➔Q can take any value from { red , green, blue}

➔ choose any , say green ➔ Q = green

➔ remove green from neighbors {NT, SA, NSW}
no variable has no values ➔ ok, continue
(d) assume that the next variable to assign is V
➔V can take any value from { red, green, blue}
➔ choose any , say blue ➔ V = blue
➔ remove green from neighbors {NT, SA, NSW}
➔ SA has no values ➔ failure ➔ stop



Constraint Propagation or Detecting Inconsistencies

Forward checking propagates information from assigned to unassigned variables, but doesn't provide early detection for all failures.

K-consistency

1-consistency = node consistency
2-consistency = arc consistency
3-consistency = path consistency

Strong-consistency

A graph is **strongly consistent** it is k-consistent and is also (k-1)-consistent, …down to 1-consistent

Arc consistency

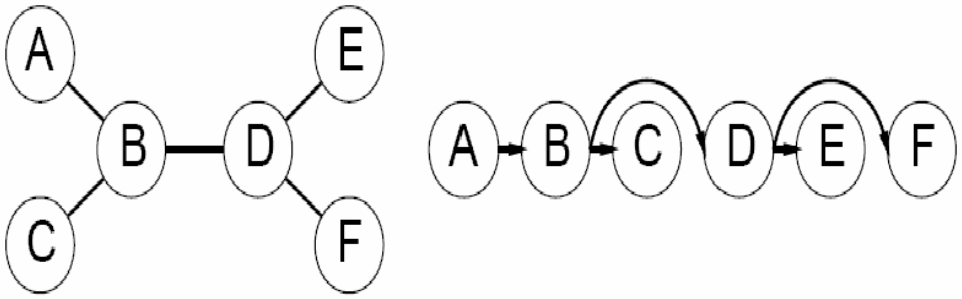
Simplest propagation makes each arc consistent
$X \rightarrow Y$ is consistent iff for every value x of X there is some allowed y

In choosing a new value for a variable, the most obvious heuristic is to select the value that results in the minimum number of conflicts with other variables. This is **min-conflicts** heuristic.

Tree-structured CSP

if the CSP constraint graph is a tree , it will be solved much quicker.

1. Choose a variable as root, order variables from root to leaves such that every node's parent precedes it in the ordering



2. For j from n down to 2, apply REMOVEINCONSISTENT$(Parent(X_j), X_j)$

3. For j from 1 to n, assign X_j consistently with $Parent(X_j)$

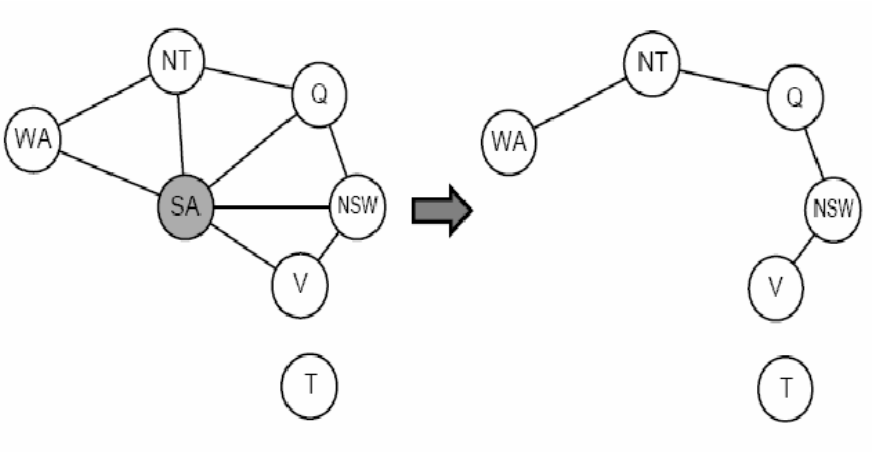
What if the constraint graph is not a tree ?

We have to convert it to a tree using one of two methods :

- Cycle Cutset
- Tree Decomposition

Cycle Cutset algorithm

1. Choose a subset S from VARIABLES[csp] such that the constraint graph becomes a tree after removal of S. S is called a **cycle cutset**.
2. For each possible assignment to the variables in S that satisfies all constraints on S,
   (a) remove from the domains of the remaining variables any values that are inconsistent with the assignment for S, and
   (b) If the remaining CSP has a solution, return it together with the assignment for S.

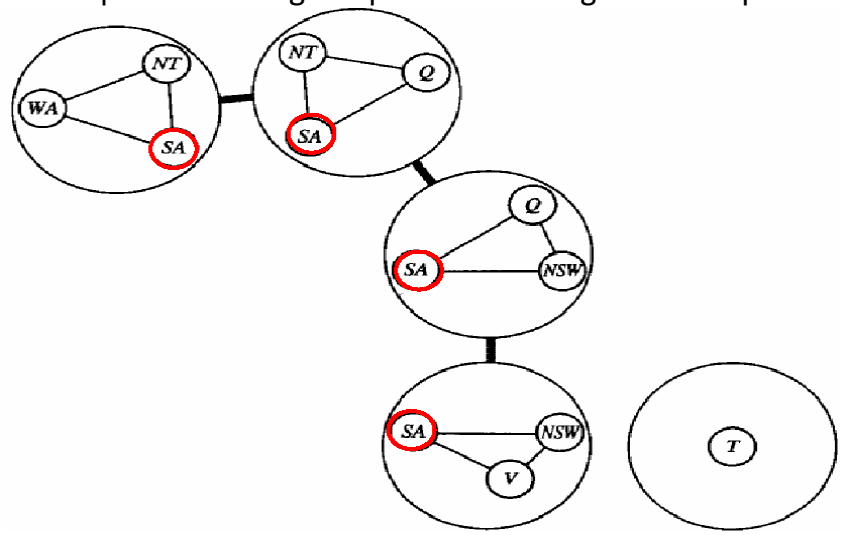


Tree Decomposition

decomposition of the constraint graph into a set of connected subproblems. Each subproblem is solved independently, and the resulting selutions are then combined. (**divide-and-conquer**).

A tree decomposition must satisfy the following three requirements:

- Every variable in the original problem appears in at least one of the subproblems
- If two variables are connected by aconstraint in the original problem, they must appear together in at least one of the subproblems.
- If a variable appears in two subproblems in the tree, it must appear in every subproblem along the path connecting those subproblems.



Performance measure of CSP

R = number of constraints ÷ number of variables