CS161 Discussion 4
CSP

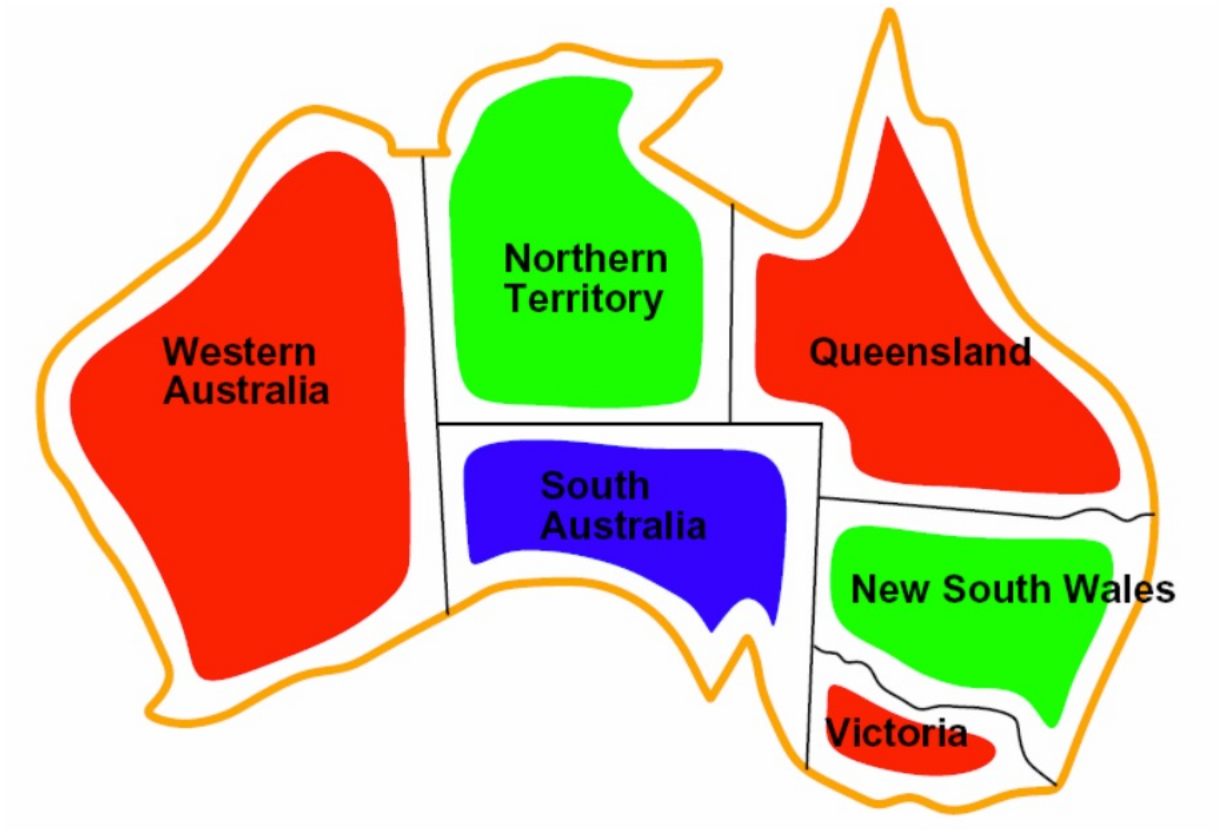
Jinghao Zhao

10/22/2021

Constraint Satisfaction Problems (CSP)

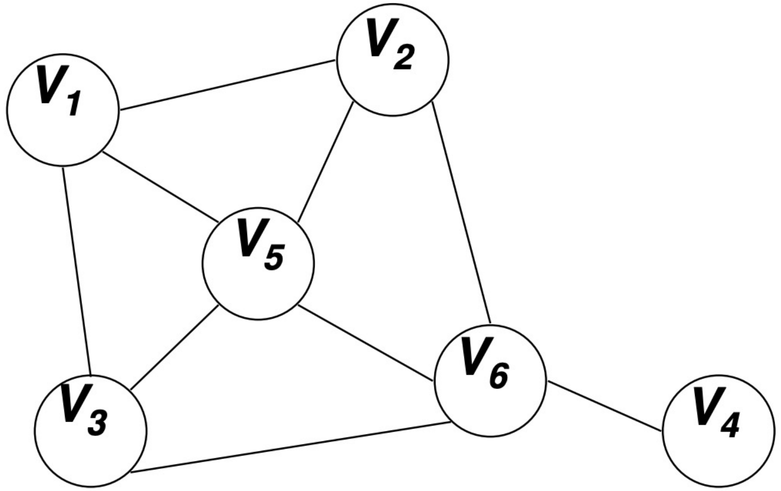
- CSP = {V, D, C}
- Variables: V = {V1 ,..,VN }
  - Example: The values of the nodes in the graph
- Domain: The set of d values that each variable can take
  - Example: D = {R, G, B}
- Constraints: C = {C1 ,..,CK }
- Each constraint consists of a tuple of variables and a list of values that the tuple is allowed to take for this problem
  - Example: [(V2 ,V3),{(R,B),(R,G),(B,R),(B,G),(G,R),(G,B)}]
- Constraints are usually defined implicitly: A function is defined to test if a tuple of variables satisfies the constraint
  - Example: Vi ≠ Vj for every edge (i,j)

Canonical Example: Graph Coloring

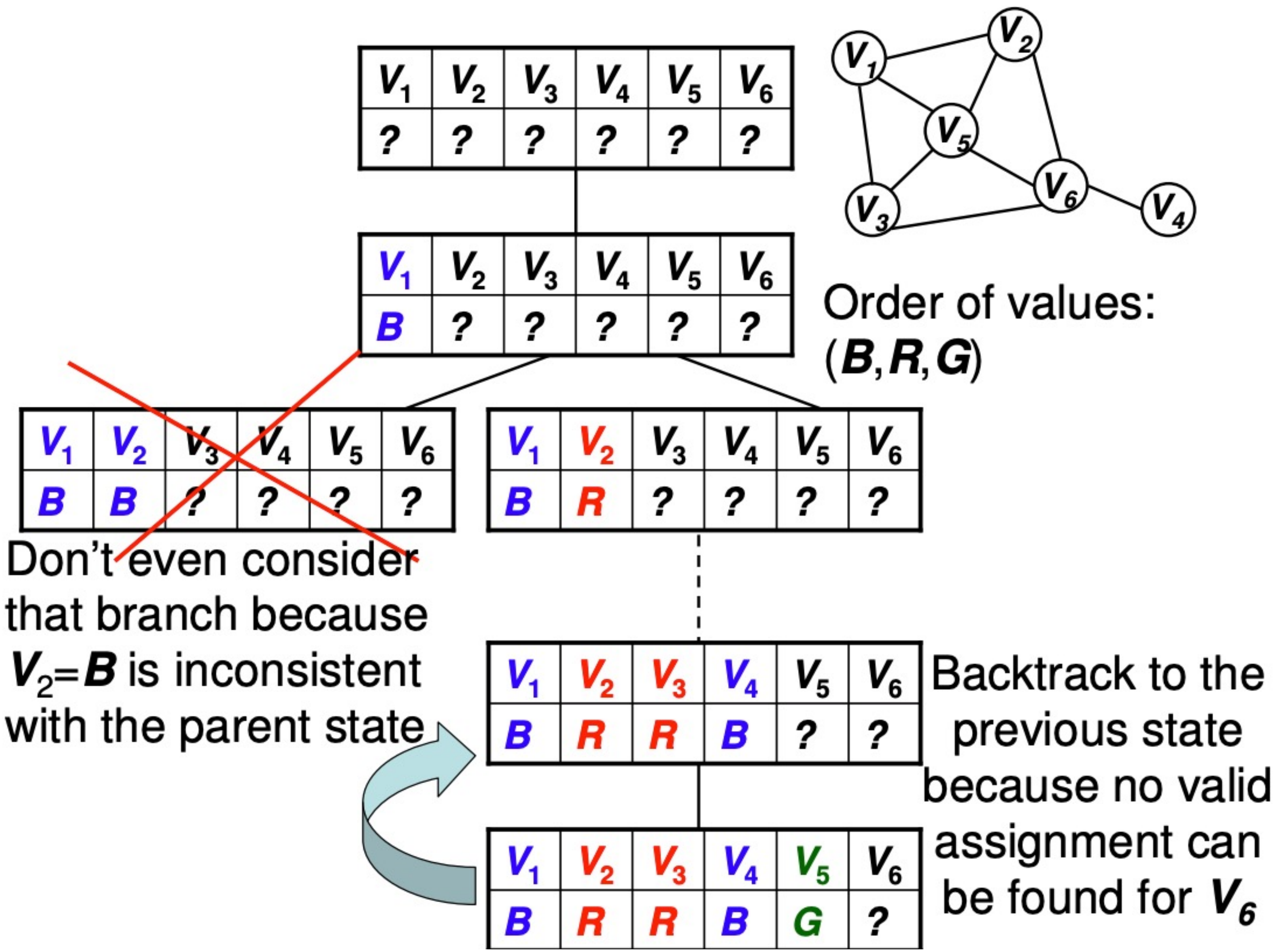


Canonical Example: Graph Coloring

- Consider N nodes in a graph
- Assign values V1 ,..,VN to each of the N nodes
- The values are taken in {R,G,B}
- Constraints: If there is an edge between i and j, then Vi must be different of Vj



Backtrack search

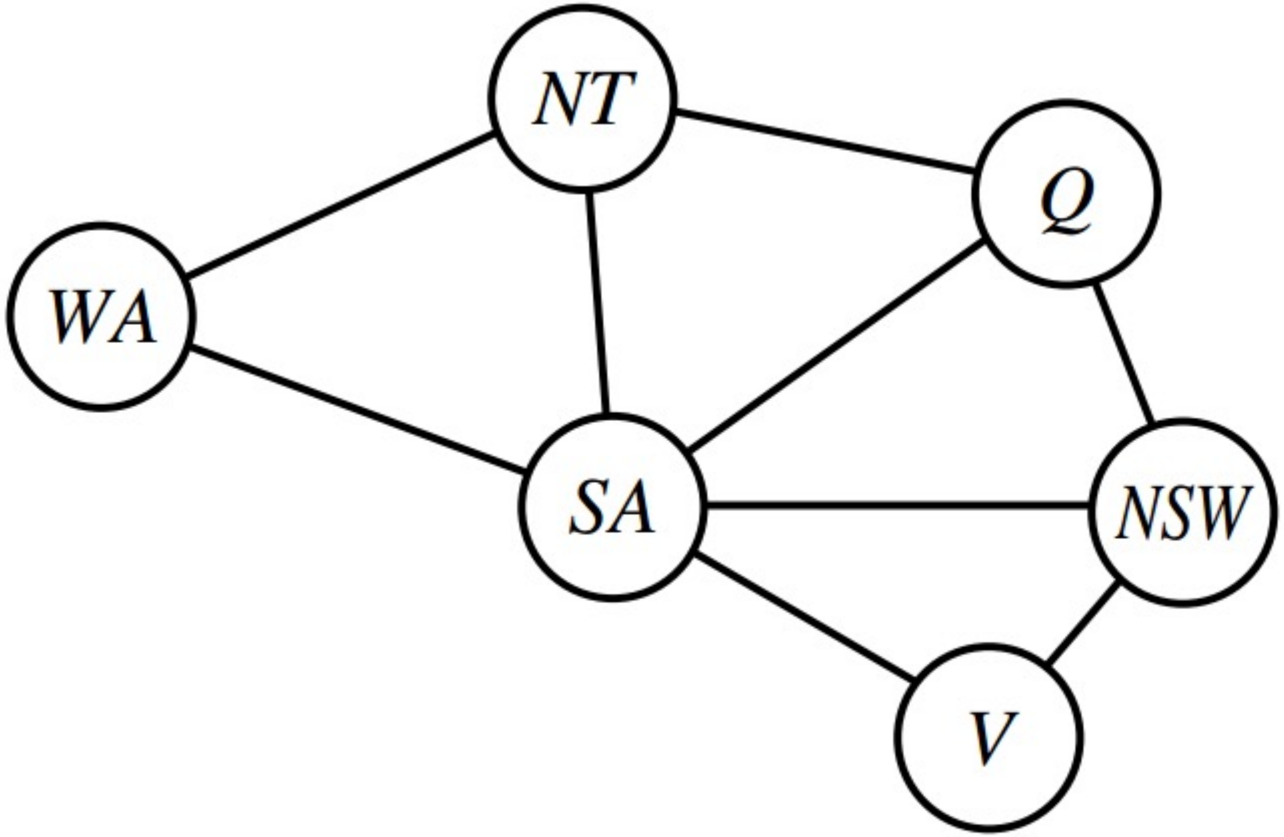


Constraint Satisfaction Problem

X is a set of variables, $\{X_1, \ldots, X_n\}$.
D is a set of domains, $\{D_1, \ldots, D_n\}$, one for each variable.
C is a set of constraints that specify allowable combinations of values.

- A **state** in CSP: an assignment of values to some or all variables
  - Consistent/Legal assignment: an assignment that does not violate any constraints
  - Complete assignment: every variable is assigned (otherwise partial assignment)
- A **solution** in CSP: a consistent, complete assignment

Constraint Graph



Exercise – CSP Formulations

- Class scheduling
  - A fixed number of professors
  - A fixed number of classrooms
  - A list of classes to be offered
  - A list of possible time slots for classes.
  - Each professor has a set of classes that he or she can teach.

Exercise – CSP Formulations

- Hamiltonian tour
  - Given a network of cities connected by roads, choose an order to visit all cities in a country without repeating any.

Main Algorithm for CSP - Backtracking DFS

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*) **do**
    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*

Backtrack

    remove {*var* = *value*} and *inferences* from *assignment*
  return *failure*

Main Algorithm for CSP - Backtracking DFS

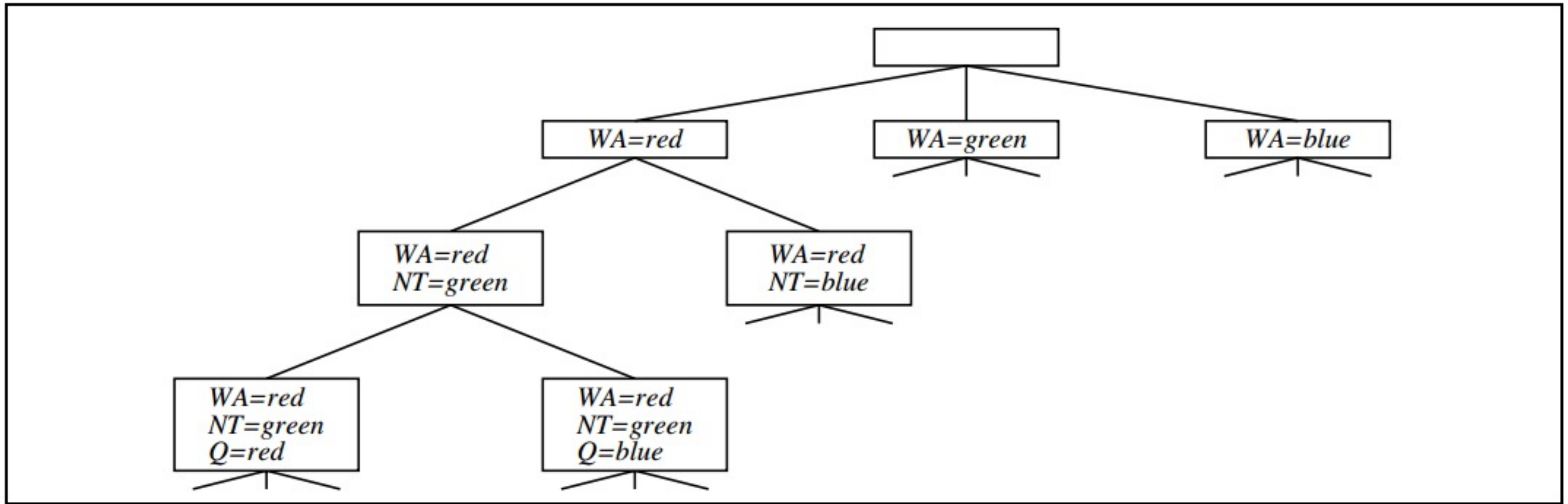


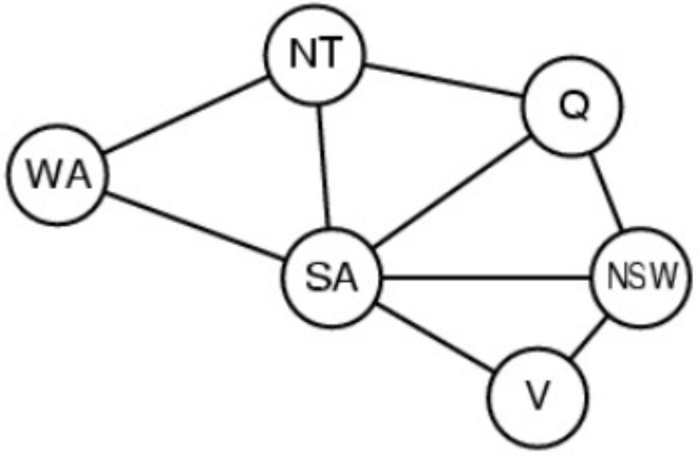
Figure 6.6 Part of the search tree for the map-coloring problem in Figure 6.1.

- When to backtrack?
  - When a variable has no legal values left to assign (No possible consistent assignment)

Variable and Value Ordering

Backtracking DFS:

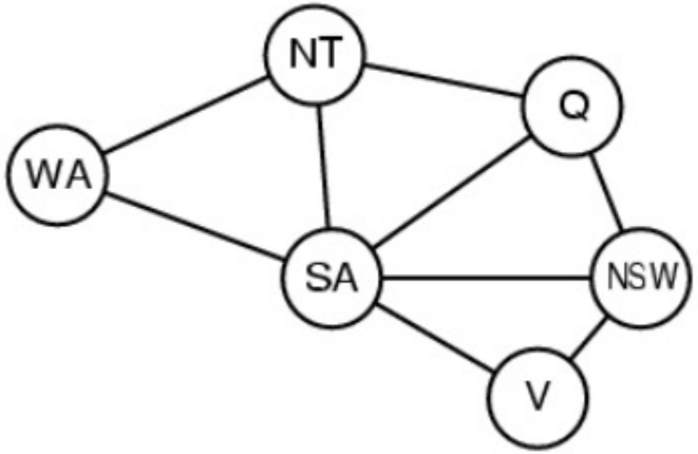
- Choose a variable and assign a value.
- Backtrack when no legal values left.
- Keep trying until it fails



- **How to select unassigned variable?**
- **In order what should its values be tried?**

Variable and Value Ordering

- **How to select unassigned variable?**
  - Minimum-remaining-values (MRV) heuristic
    - a.k.a. "most constrained variable", or "fail-first"
    - If no legal values left, fail immediately
  - Degree heuristic
    - Pick variable with most constraints on remaining variables
    - Attempt to reduce branching factor on future choice
    - Useful as a tie-breaker



- **In order what should its values be tried?**
  - Least-constraining-value
    - Leave the maximum flexibility for subsequent variable assignments

Arc Consistency

Backtracking DFS:

Choose a variable, try a value in the variable's domain.

Can we eliminate impossible values according to constraints before further search?

- **Arc consistency**
  - Variable is arc consistent: Every value in its domain satisfies the variable’s binary constraints
    - X_i is arc-consistent with respect to another variable X_j if for every value in the current domain D_i there is some value in the domain D_j that satisfies the binary constraint on the arc (X_i, X_j)
  - Network is arc consistent: every variable is arc consistent with every other variable

Example – Arc Consistency

- The domain of both X and Y is the set of digits (0~9).
- $Y = X^2$

<u>What will the domains of X and Y be after enforcing arc consistency?</u>

Example – Arc Consistency

- The initial domain of both X and Y is the set of digits (0~9). $Y = X^2$.

- **Consider the arc (X $\leftarrow$ Y)**
  - Make X arc-consistent with respect to Y
    - For any value in X's domain, there should be at least one value in Y's domain that satisfied the constraint
  - X: $\{0,1,2,3\}$
- **Consider the arc (Y $\leftarrow$ X)**
  - Y: $\{0,1,4,9\}$

Result

X: $\{0,1,2,3\}$

Y: $\{0,1,4,9\}$

Arc Consistency Algorithm: AC-3

- Maintains a queue (set) of arcs
- Pop an arbitrary arc $\left(X_i \leftarrow X_j\right)$ and check D_i (the domain of X_i)
  - D_i unchanged
    - Move to next
  - D_i becomes smaller
    - Add to queue all arcs$(X_k \leftarrow X_i)$ where X_k is a neighbor of X_i
  - D_i is empty
    - Fail!

Finally, we get an CSP that is equivalent to the original CSP(with same solutions).

But now variables have smaller domains!

Arc Consistency Algorithm: AC-3

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 $revised$

Complexity of AC-3

- n variables
- d: largest domain size
- c binary constraints
- Each arc (X_k, X_i) can be inserted at most d times
  - Xi has at most d values to delete
- Checking consistency of one arc: $O(d^2)$

- $O(cd^3)$

Complexity of AC (binary constraints)

- n: variables
- d: largest domain size
- binary constraints

- Each arc (X_k, X_i) can be inserted at most d times
  - Xi has at most d values to delete
- Checking consistency of one arc: $O(d^2)$
- Binary CSPs: $O(n^2)$

- Complexity of AC $is\ O(n^2 d^3)$

Maintaining Arc Consistency (MAC)

After assigning a value to a variable, we can use either of the following algorithm to eliminate impossible values before next round's search starts:

- **Forward checking** only makes current variable arc-consistent
- **MAC** maintains global arc consistency

Difference between Forward Checking and MAC:

<u>Which arcs to check?</u>

Forward Checking and MAC

- We can apply AC-3 before search starts
  - $Y = X^2$ => X: {0,1,2,3}, Y: {0,1,4,9}

- Can we do domain reductions according to arc consistency during search (**after assigning a value to a variable**)?
  - And detect inevitable failure early

Forward Checking

Which arcs to check after assigning a value to variable X?

- Only the arcs of X

Forward Checking

- Keep track of remaining legal values for unassigned variables that are connected to current variable. (**Variable-level arc consistency**)
- Terminates when any variable has no legal values
  - Then backtrack!

Maintaining Arc Consistency (MAC)

Which arcs to check after assigning a value to variable X?

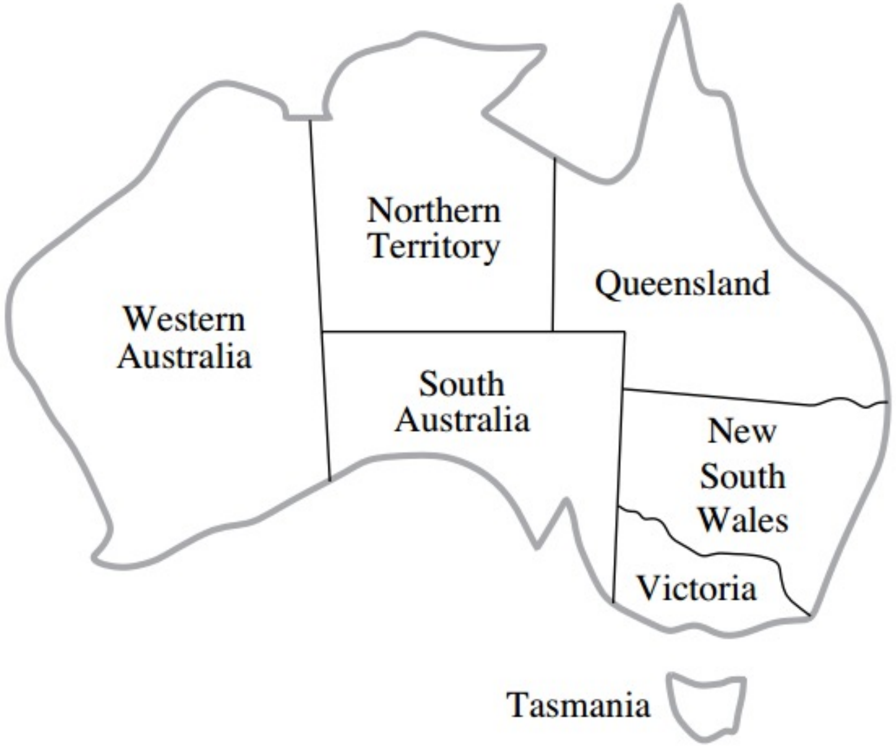
- First, push all the arcs of X
  - Same as forward checking
- Pop an arc $(Y \leftarrow X)$ and perform domain reduction
  - Y: a neighbor of X, unassigned
  - If domain size of Y reduces, push all the arcs of Y (constraint propagation)
  - If (Z <- Y) reduces the domain reduction of Z, all the arcs of Z are also pushed
- Keeps popping and pushing until the arc queue is empty

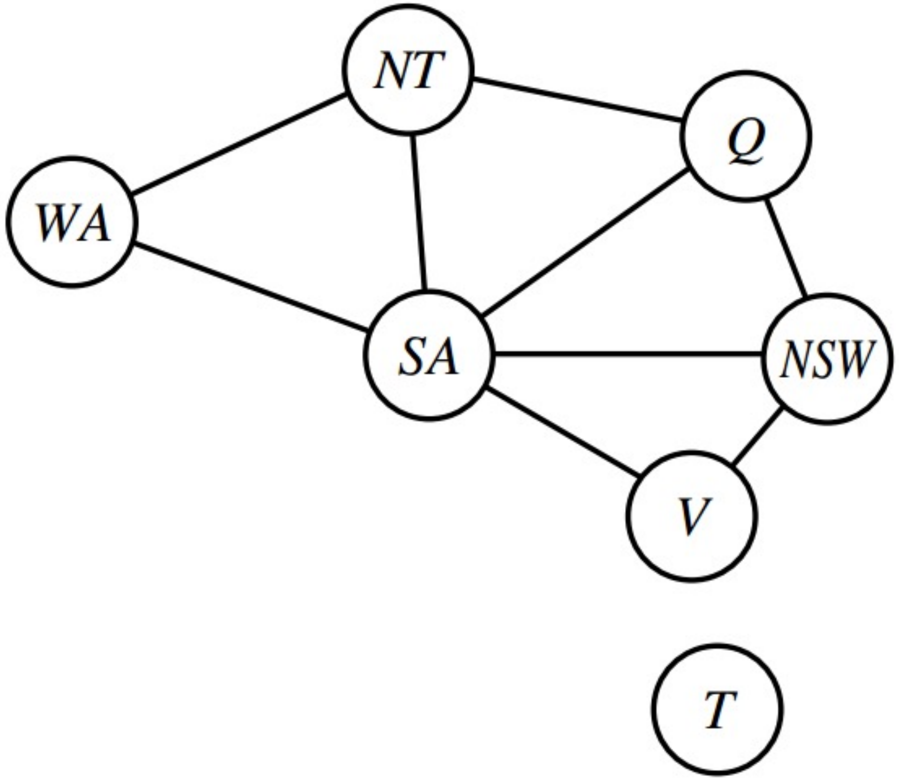
(Similar to AC-3)

MAC is strictly more powerful than forward checking.

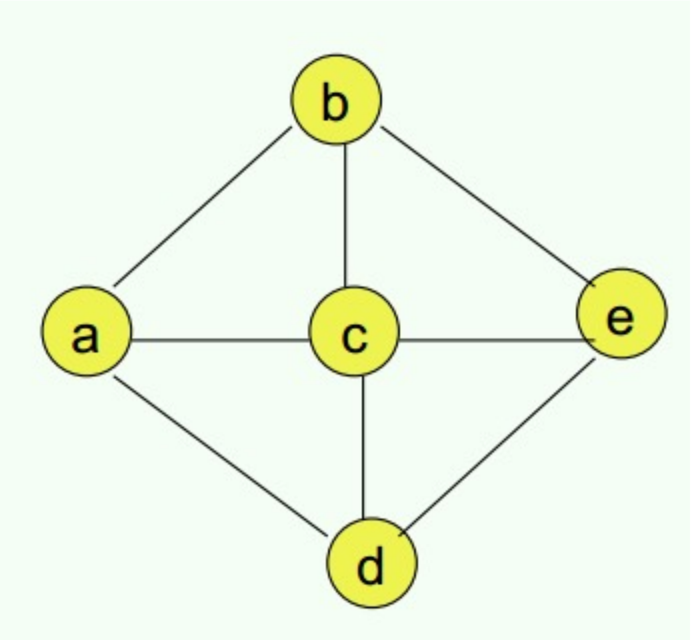
Exercise

- Use the AC-3 algorithm to show that arc consistency can detect the inconsistency of the partial assignment {WA = green, V = red}





Exercise – Solve CSP



- The domain for every variable is [1,2,3,4].
- 2 unary constraints:
  - variable “a” cannot take values 3 and 4.
  - variable “b” cannot take value 4.
- Variables connected by an edge cannot have the same value.

===========================

Heuristics:

===========================

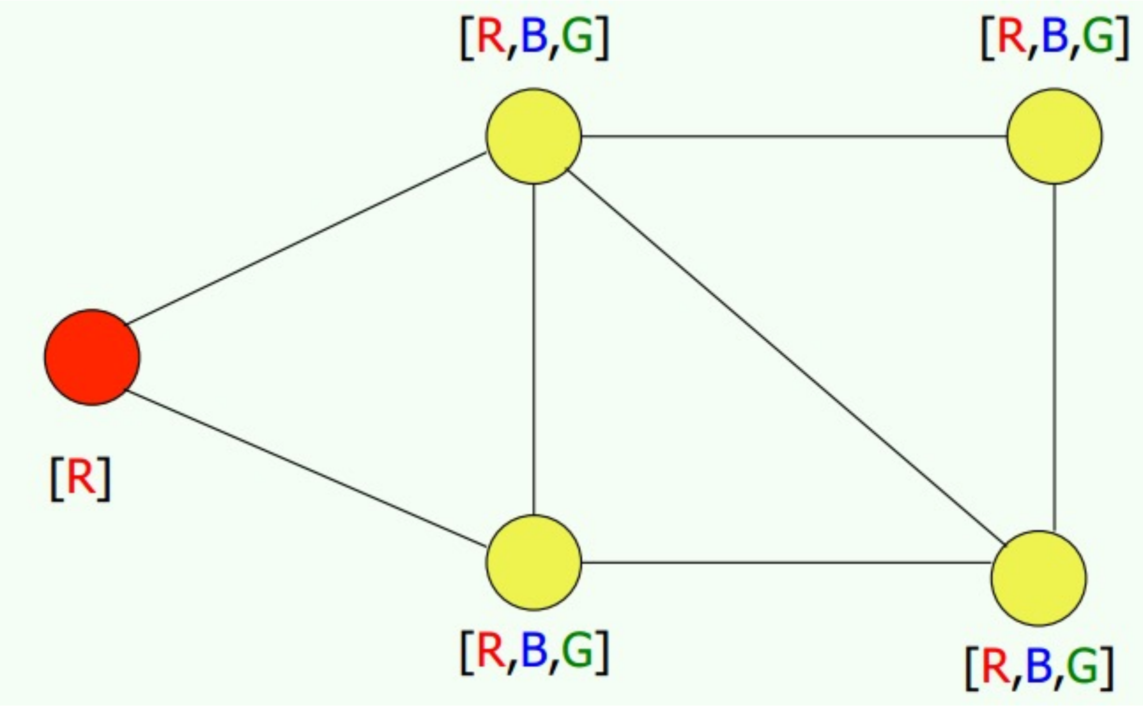
Find solution for this CSP. Show each step and explain.

Exercise – Solve CSP

- Connected variables cannot share color

Solve this CSP and explain each step

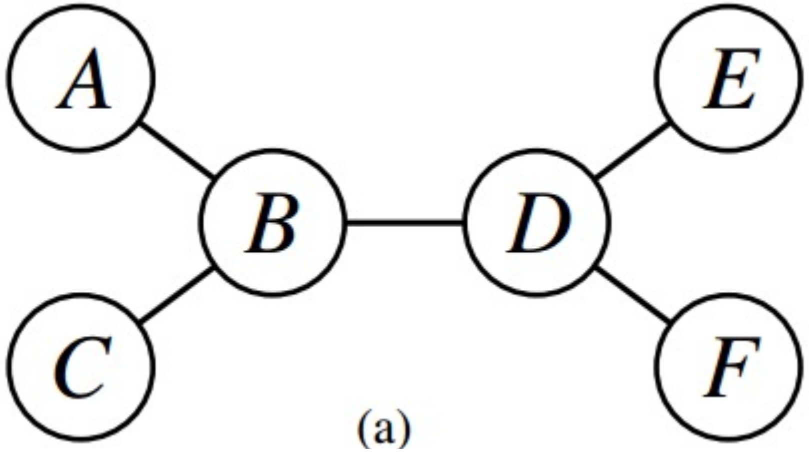
- Use all heuristics



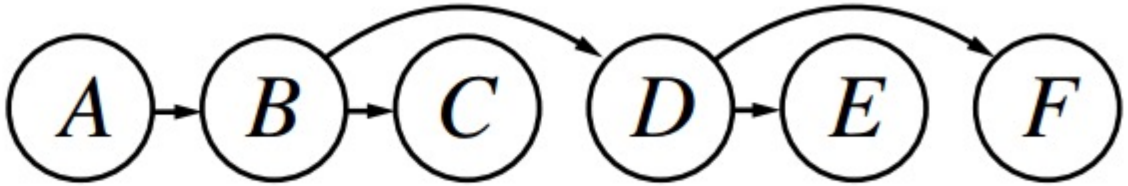
Tree-structured CSP

Tree-structured CSP could be solved in time linear in # of variables

Method: **Topological sorting**



(a)



(b)

- Assign values to variable in order
- No need to backtrack!

Time Complexity: $O(nd^2)$

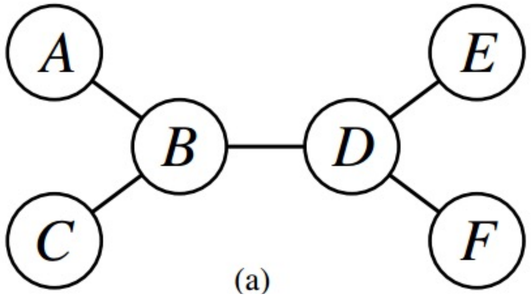
Tree-structured CSP

- n nodes => n-1 edges (tree-structure)
- domain size: d

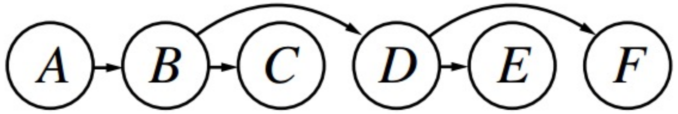
1. topological sorting: $O(|V| + |E|) = O(n)$

2. Assigning values to variables: $O(nd^2)$
- O(n) arcs
- Each arc will only be used once for consistency
- Enforcing arc consistency for a single arc: d^2

Time Complexity: $O(nd^2)$



(a)



(b)

Reduce general CSP to tree-structured CSP

- **Tree-structured CSP:** $O(nd^2)$
- **General CSP:** $O(d^n)$ in the worst-case
  - n: solution depth, d: branching factor

Tree-structured CSP is <u>easy to solve but rare</u>

Can we somehow reduce general (or nearly-tree) constraint graphs to trees?

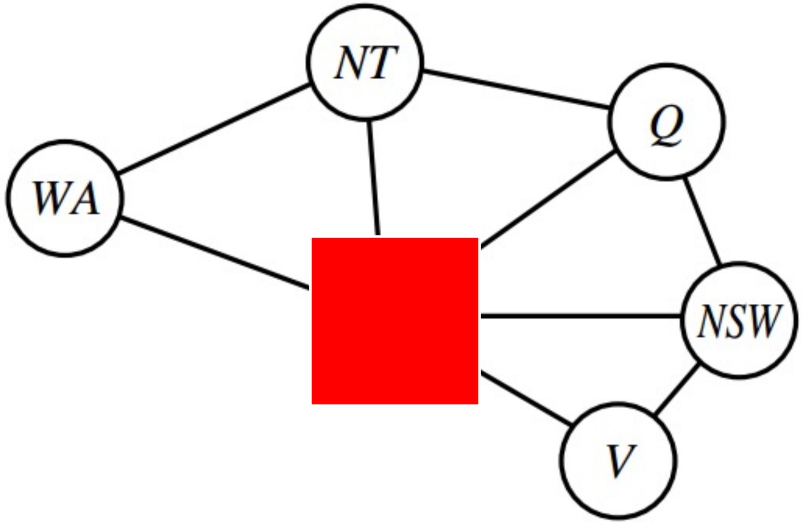
Reduce general CSP to tree-structured CSP

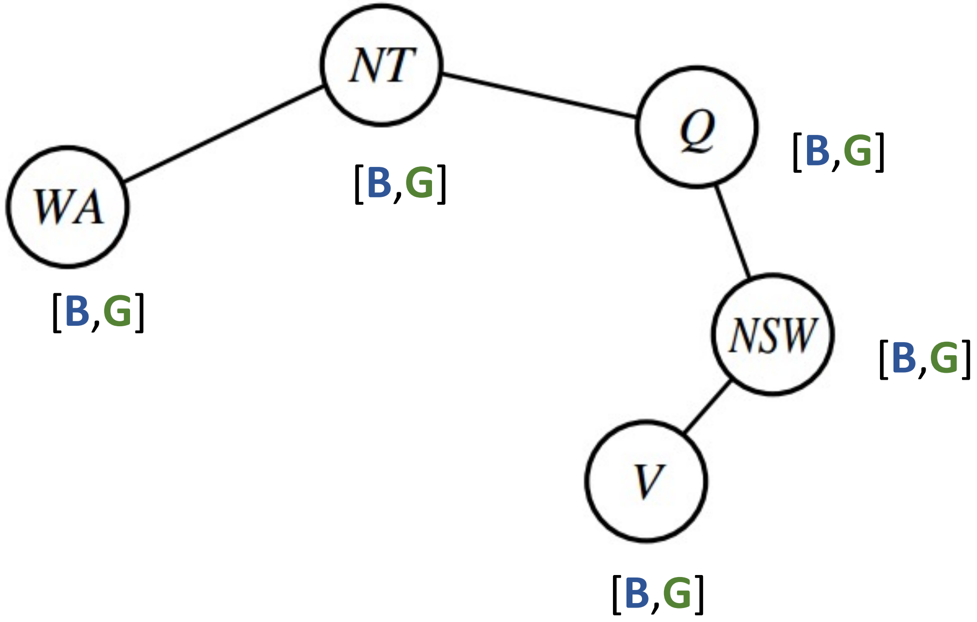
Methods:

- **Cutset conditioning**
  - Assign values to some variables so that the remaining variables form a tree
- **Tree decomposition**
  - Decompose the original problem into a set of connected subproblems

Cutset Conditioning

- Assign values to some variables so that the remaining variables form a tree
  - Cycle cutset: a set of variables. Removing the set of variables and graphs will have no cycle
  - Conditioning: Assign values to certain variables and prune neighbors' domains





Cutset Conditioning

- How to find the minimum cycle cutset?
  - NP-hard! But efficient approximation are known.

Cutset Conditioning Time complexity Analysis

n variables. Cycle cutset size: c

- We have to try d^c combinations for variables in cutset
- The rest of the graph is a tree-structured CSP of (n-c) variables$O((n-c)d^2)$

$$\boldsymbol{O(d^c(n-c)d^2)}$$

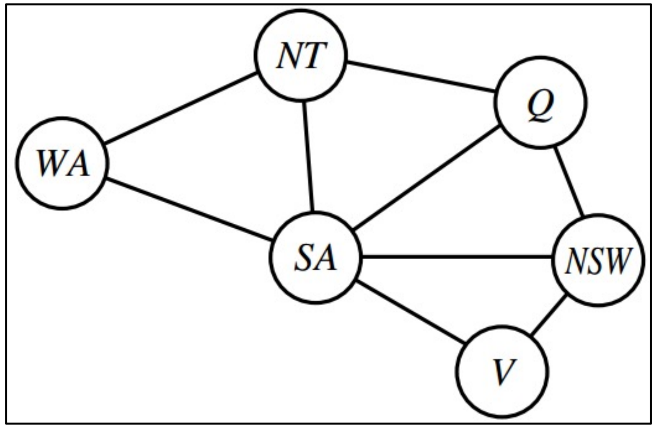
Tree Decomposition

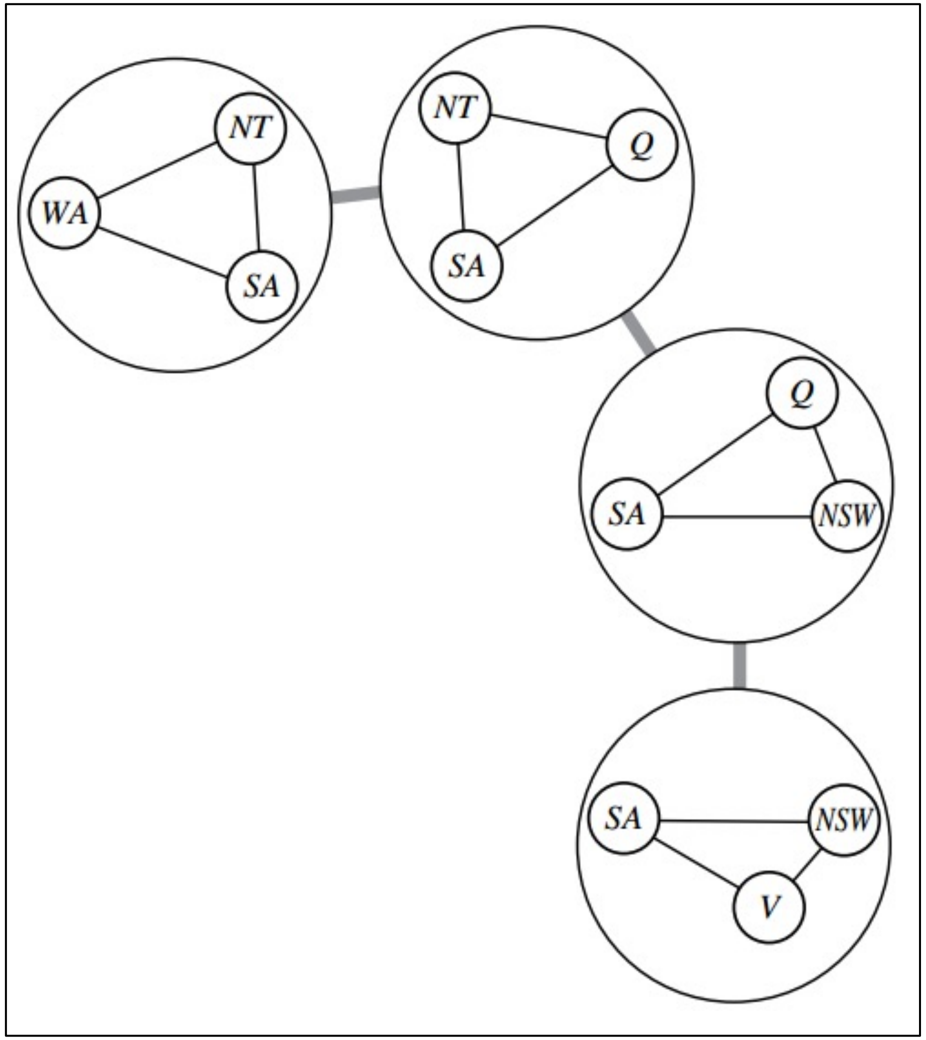
- Decompose the original problem into a set of connected subproblems

1) Every variable in the original problem appears in at least one of the subproblems.
2) If two variables are connected by a constraint in the original problem, they must appear together (along with the constraint) in at least one of the subproblems.
3) If a variable appears in two subproblems in the tree, it must appear in every subproblem along the path connecting those subproblems.

Tree Decomposition

1) Every variable in the original problem appears in at least one of the subproblems.
2) If two variables are connected by a constraint in the original problem, they must appear together (along with the constraint) in at least one of the subproblems.
3) If a variable appears in two subproblems in the tree, it must appear in every subproblem along the path connecting those subproblems.





Tree Decomposition

- Each subproblem is a "mega-variable"
  - Domain: all the solutions to the subproblem
- Constraints between subproblems:
  - Subproblem solutions agree on their shared variables.

A constraint graph may correspond to multiple tree decomposition

- When choosing the decomposition, make the subproblems as small as possible