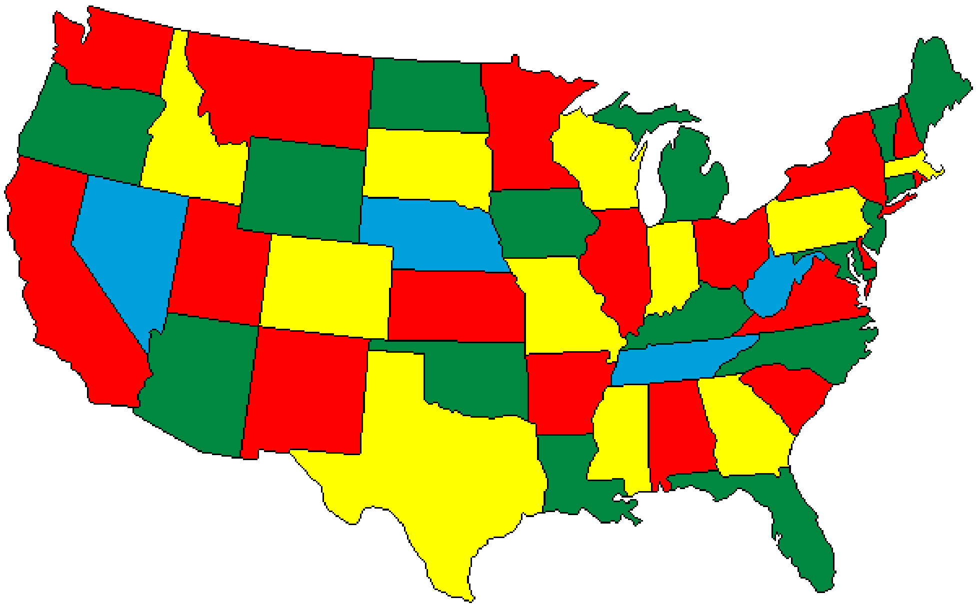
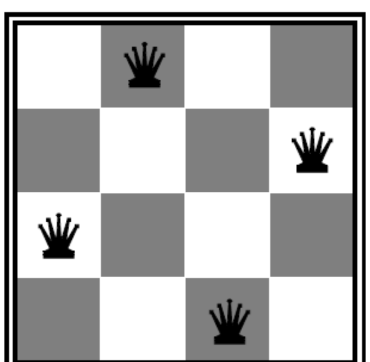
# Constraint Satisfaction Problems

Artificial Intelligence

Slides are mostly adapted from AIMA, MIT Open Courseware
Svetlana Lazebnik (UIUC) and Manuela Veloso (CMU)



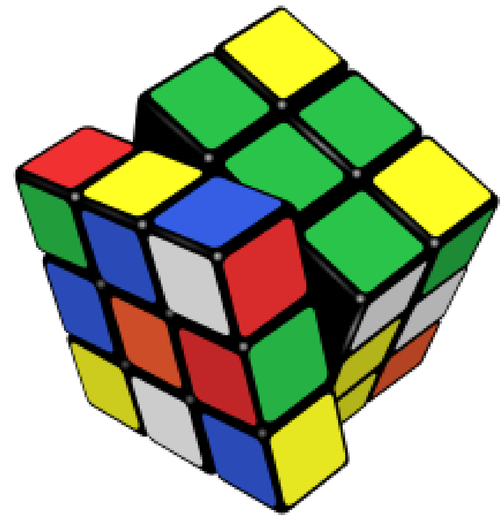
```
    T W O
  + T W O
  -------
  F O U R
```



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

# What is search for?

- Assumptions: single agent, deterministic, fully observable, discrete environment
- **Search for *planning***
  - The path to the goal is the important thing
  - Paths have various costs, depths
- **Search for *assignment***
  - Assign values to variables while respecting certain constraints
  - The goal (complete, consistent assignment) is the important thing



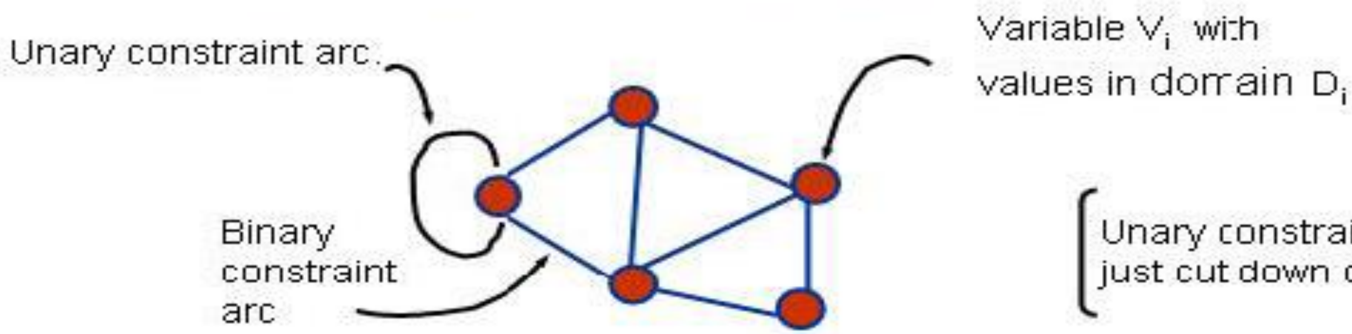
| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 8 | | | 4 | | 6 | | | 7 |
| | | | | | | 4 | | |
| | 1 | | | | | 6 | 5 | |
| 5 | | 9 | | 3 | | 7 | 8 | |
| | | | | 7 | | | | |
| | 4 | 8 | | 2 | | 1 | | 3 |
| | 5 | 2 | | | | | 9 | |
| | | 1 | | | | | | |
| 3 | | | 9 | | 2 | | | 5 |

# Constraint satisfaction problems (CSPs)

- Definition:
  - **State** is defined by variables $X_i$ with values from domain $D_i$
  - **Goal test** is a set of constraints specifying allowable combinations of values for subsets of variables
  - **Solution** is a complete, consistent assignment
- How does this compare to the "generic" tree search formulation?
  - A more structured representation for states, expressed in a formal representation language
  - Allows useful general-purpose algorithms with more power than standard search algorithms

# Constraint Satisfaction Problems

**General class of Problems:** **Binary CSP**




[Unary constraints just cut down domains]

**This diagram is called a constraint graph**

**Basic problem:**

**Find a $d_j \in D_i$ for each $V_i$ s.t. all constraints satisfied (finding consistent labeling for variables)**

# Varieties of CSPs

- Discrete variables
  - finite domains:
    - *n* variables, domain size *d* -> *O(dn)* complete assignments
    - e.g., Boolean CSPs, incl.~Boolean satisfiability (NP-complete)
  - infinite domains:
    - integers, strings, etc.
    - e.g., job scheduling, variables are start/end days for each job
    - need a constraint language, e.g., $StartJob1 + 5 \leq StartJob3$
- Continuous variables
  - e.g., start/end times for Hubble Space Telescope observations
  - linear constraints solvable in polynomial time by linear programming

# CSP definition

CSP = {*V*, *D*, *C*}

- *Variables*: *V* = {*V*1,..,*V*N}
  - 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* = {*C*1,..,*C*K}
- Each constraint consists of a tuple of variables and a list of values

that the tuple is allowed to take for this problem

  - Example: [(*V*2,*V*3),{(*R*,*B*),(*R*,*G*),(*B*,*R*),(*B*,*G*),(*G*,*R*),(*G*,*B*)}]
- Constraints are usually defined implicitly a□ A function is defined to

test if a tuple of variables satisfies the constraint

  - Example: $V_i \neq V_j$ for every edge ($i$,$j$)

Unary constraints involve a single variable,

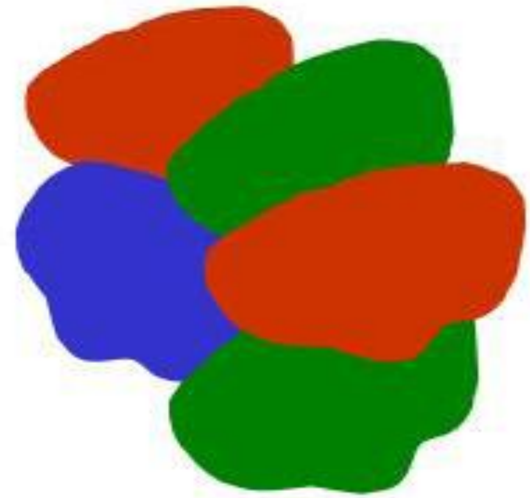
– e.g., SA ≠ green

Binary constraints involve pairs of variables,

– e.g., SA ≠ WA

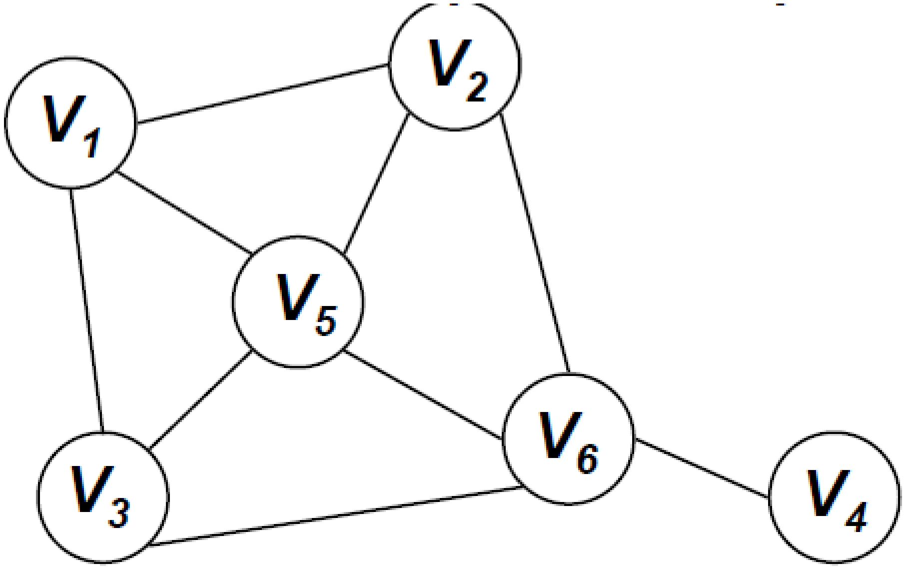
# Graph Coloring as CSP

Pick colors for map regions, avoiding coloring adjacent regions with the same color



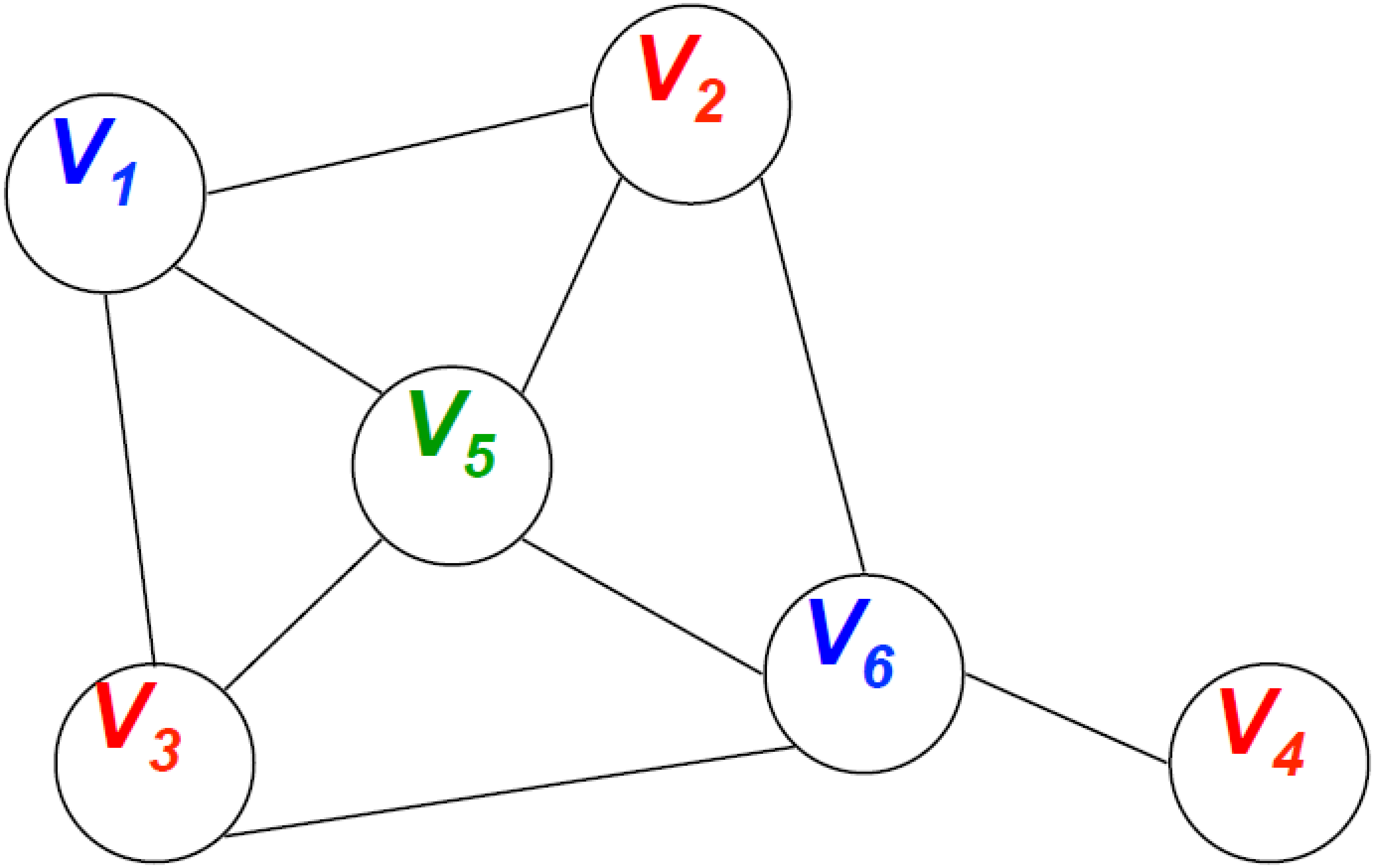
| | |
|---|---|
| **Variables** | regions |
| **Domains** | colors allowed |
| **Constraints** | adjacent regions must have different colors |

# Graph Coloring

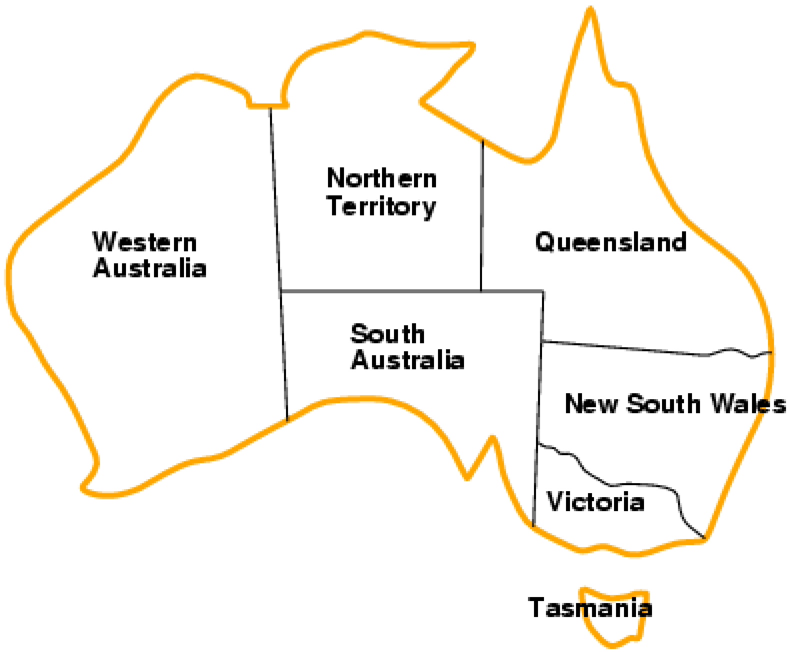



- Consider *N* nodes in a graph
- Assign values $V_1,..,V_N$ to each of the *N* nodes
- The values are taken in {*R*,*G*,*B*}
- Constraints: If there is an edge between *i* and *j*, then $V_i$ must be different from $V_j$

# Graph Coloring

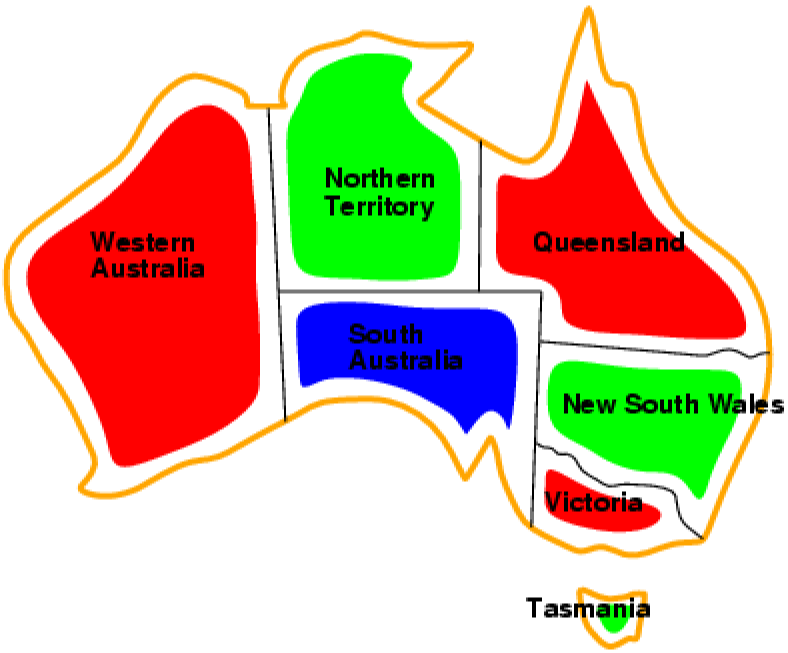


# Example: Map Coloring




- **Variables:** WA, NT, Q, NSW, V, SA, T
- **Domains:** {red, green, blue}
- **Constraints:** adjacent regions must have different colors e.g., WA ≠ NT, or (WA, NT) in {(red, green), (red, blue), (green, red), (green, blue), (blue, red), (blue, green)}

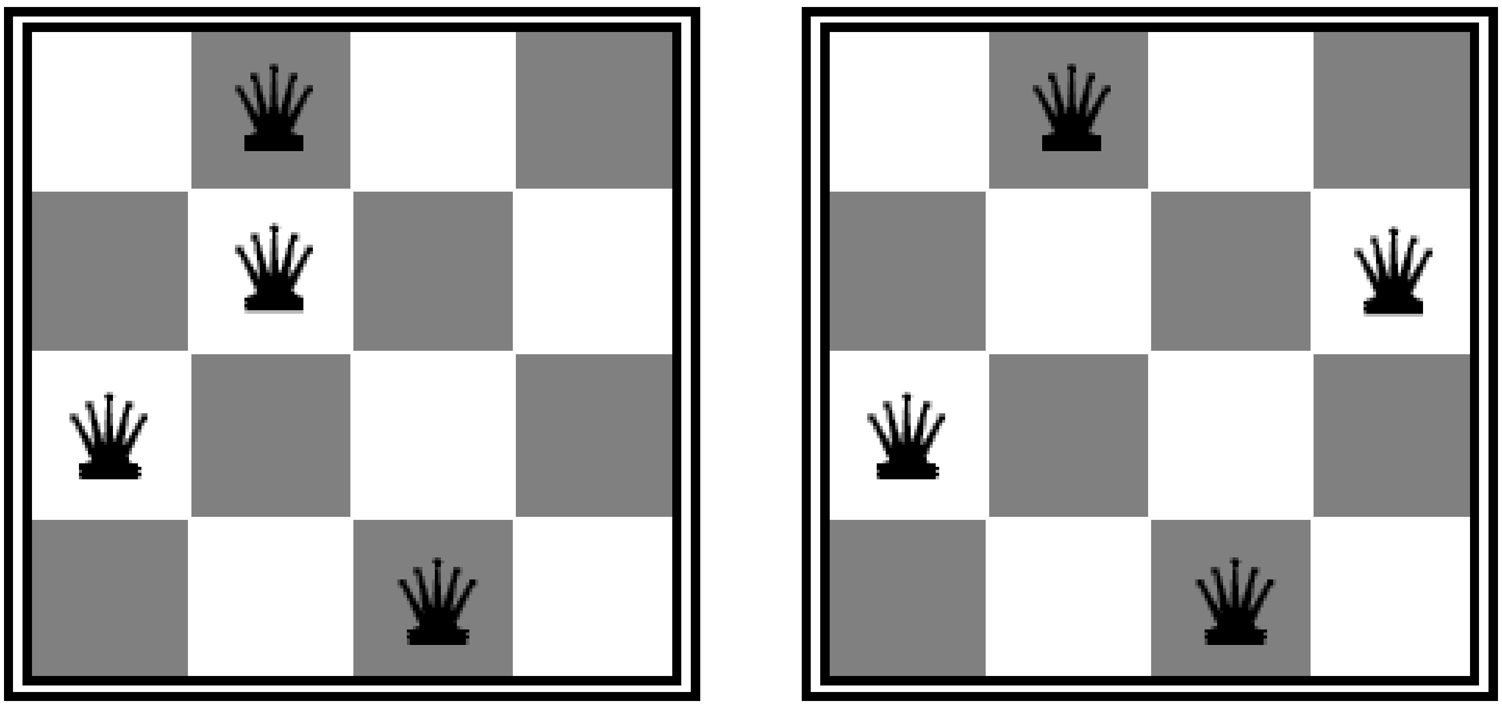
# Example: Map Coloring




- **Solutions** are *complete* and *consistent* assignments, e.g., WA = red, NT = green, Q = red, NSW = green, V = red, SA = blue, T = green

# Example: *n*-queens problem

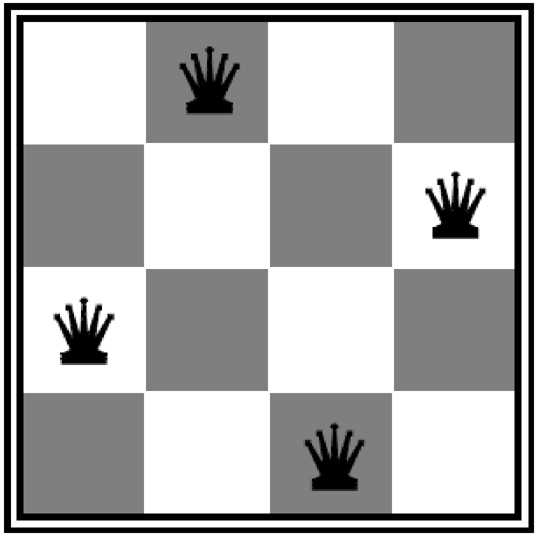
- Put $n$ queens on an $n \times n$ board with no two queens on the same row, column, or diagonal



# N-Queens:

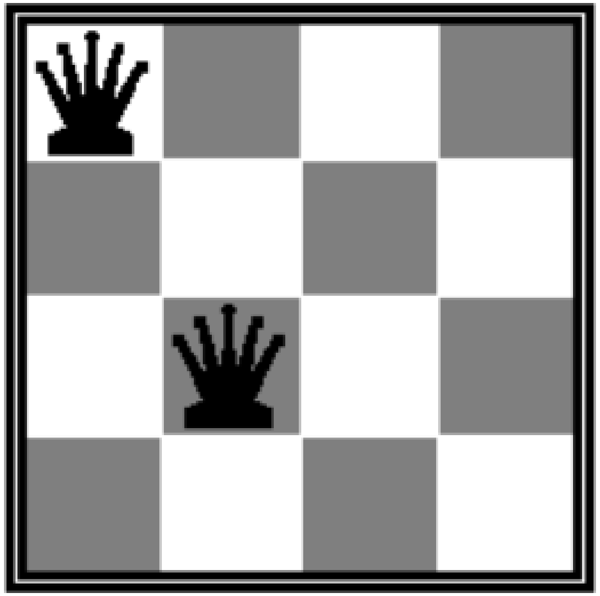
- **Variables:** $Q_i$
- **Domains:** $\{1, \ldots, N\}$
- **Constraints:**

  $\forall\ i, j$ non-threatening $(Q_i, Q_j)$



# N- Queens

- Variables: $Q_i$
- Domains: $D_i = \{1, 2, 3, 4\}$
- Constraints
  - $Q_i \neq Q_j$ (cannot be in the same row)
  - $|Q_i - Q_j| \neq |i - j|$ (or same diagonal)



$Q_1 = 1 \quad Q_2 = 3$

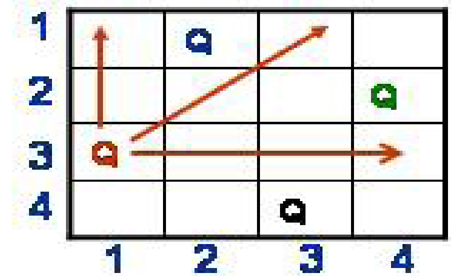
- Valid values for $(Q_1, Q_2)$ are (1,3) (1,4) (2,4) (3,1) (4,1) (4,2)

# Alternative formulation

## N-Queens as CSP

**Classic "benchmark" problem**

Place N queens on an NxN chessboard so that none can attack the other.




**Variables** are board positions in NxN chessboard

**Domains** Queen or blank

**Constraints** Two positions on a line (vertical, horizontal, diagonal) cannot both be Q

# Example: N-Queens

- **Variables:** $X_{ij}$
- **Domains:** $\{0, 1\}$
- **Constraints:**

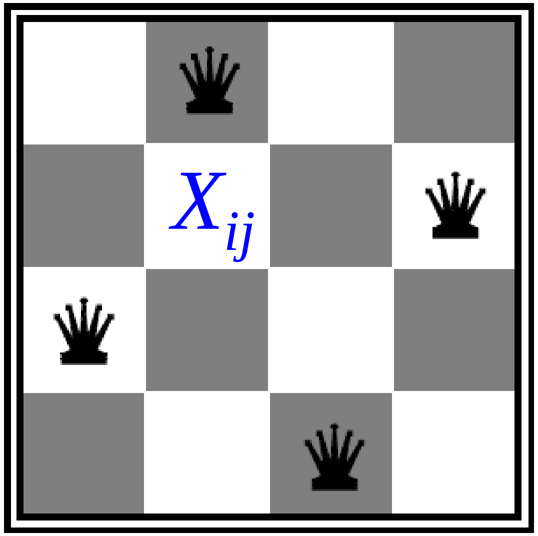
$\Sigma_{i,j} X_{ij} = N$

$(X_{ij}, X_{ik}) \in \{(0, 0), (0, 1), (1, 0)\}$

$(X_{ij}, X_{kj}) \in \{(0, 0), (0, 1), (1, 0)\}$

$(X_{ij}, X_{i+k, j+k}) \in \{(0, 0), (0, 1), (1, 0)\}$

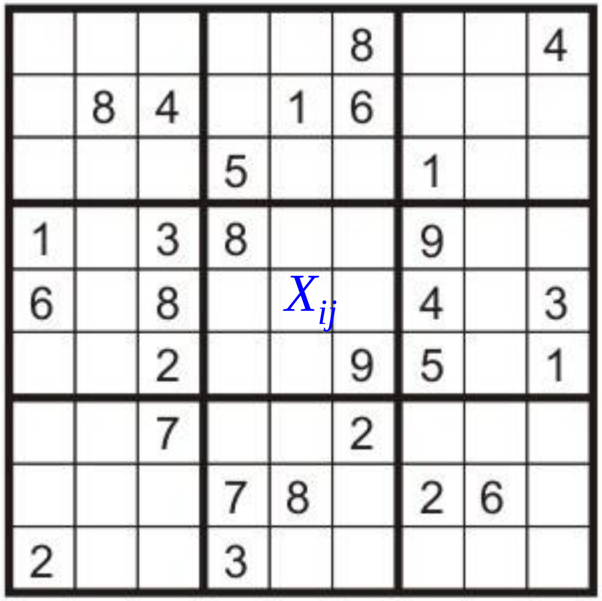
$(X_{ij}, X_{i+k, j-k}) \in \{(0, 0), (0, 1), (1, 0)\}$



# Example: Sudoku

- **Variables:** $X_{ij}$
- **Domains:** {1, 2, …, 9}
- **Constraints:**

  Alldiff($X_{ij}$ in the same *unit*)



# Example: Cryptarithmetic

- **Variables:** T, W, O, F, U, R

  $X_1$, $X_2$
- **Domains**: {0, 1, 2, …, 9}
- **Constraints:**

  $O + O = R + 10 * X_1$

  $W + W + X_1 = U + 10 * X_2$

  $T + T + X_2 = O + 10 * F$

  Alldiff(T, W, O, F, U, R)

  $T \neq 0$, $F \neq 0$

```
    T W O
+   T W O
---------
  F O U R
```

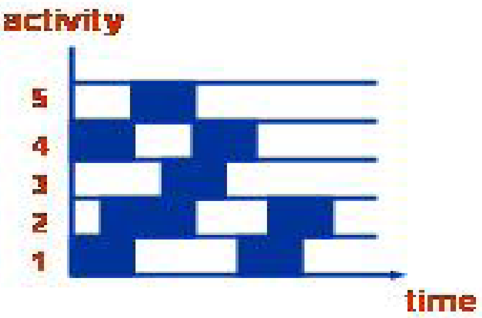
# Real-world CSPs

- Assignment problems
  - e.g., who teaches what class
- Timetable problems
  - e.g., which class is offered when and where?
- Transportation scheduling
- Factory scheduling

- More examples of CSPs: http://www.csplib.org/

# Scheduling as CSP

Choose time for activities e.g. observations on Hubble telescope, or terms to take required classes.




**Variables** are activities

**Domains** sets of start times (or "chunks" of time)

**Constraints**

1. Activities that use same resource cannot overlap in time
2. Preconditions satisfied

# CSP Example

Given 40 courses (8.01, 8.02, . . . . 6.840) & 10 terms (Fall 1, Spring 1, . . . . , Spring 5). Find a legal schedule.

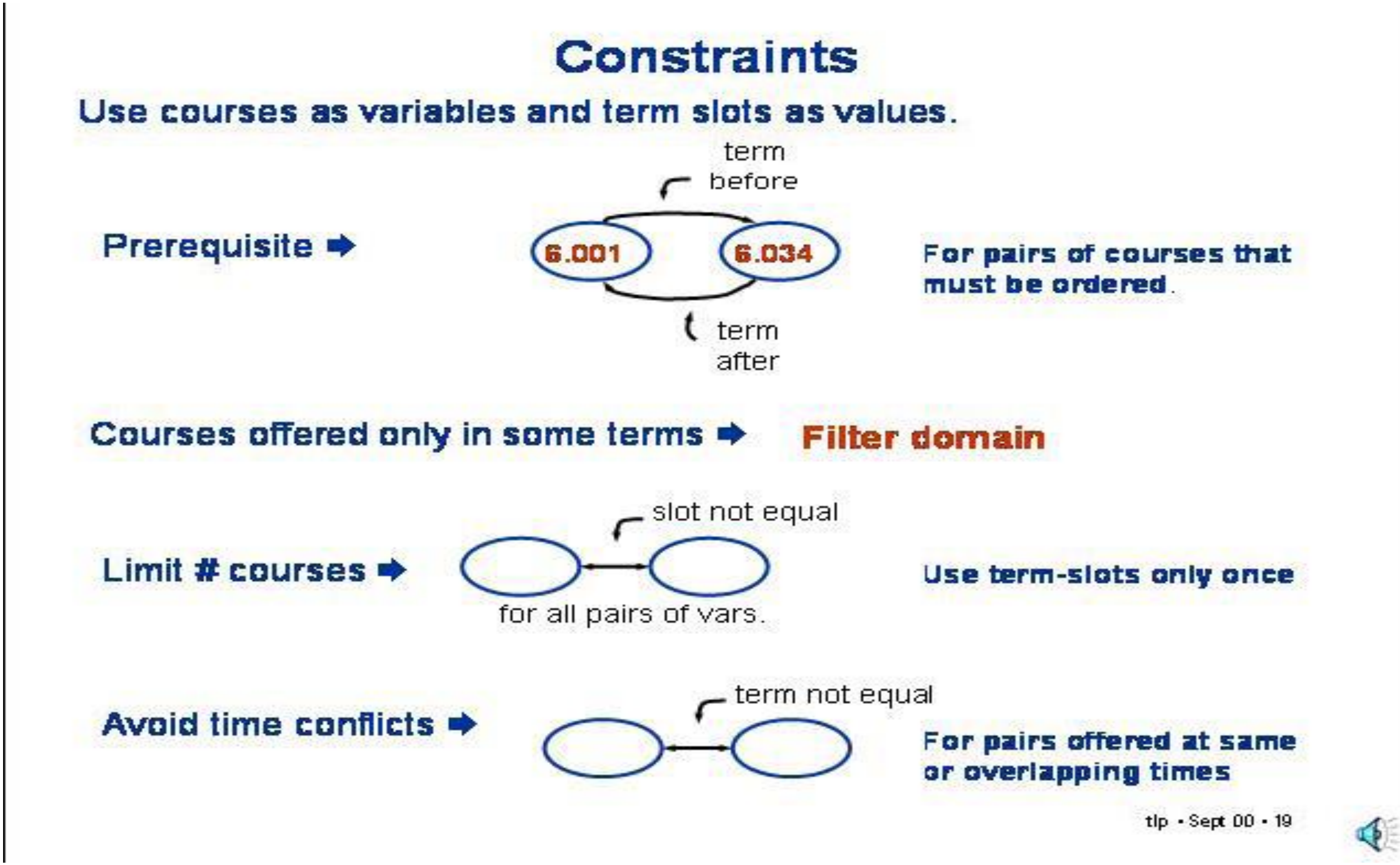
Constraints

- Pre-requisites
- Courses offered on limited terms
- Limited number of courses per term
- Avoid time conflicts

Note, CSPs are not for expressing (soft) preferences e.g., minimize difficulty, balance subject areas, etc.

# Choice of variables & values

| VARIABLES | DOMAINS |
|---|---|
| A. Terms? | Legal combinations of for example 4 courses (but this is huge set of values). |
| B. Term Slots? subdivide terms into slots e.g. 4 of them (Fall 1,1) (Fall 1,2) (Fall1,3) (Fall 1,4) | Courses offered during that term |
| C. Courses? | Terms or term slots (Term slots allow expressing constraint on limited number of courses / term.) |


Constraints
Use courses as variables and term slots as values.
Prerequisite ➡
term before
6.001
6.034
term after
For pairs of courses that must be ordered.
Courses offered only in some terms ➡
Filter domain
Limit # courses ➡
slot not equal
for all pairs of vars.
Use term-slots only once
Avoid time conflicts ➡
term not equal
For pairs offered at same or overlapping times
tlp • Sept 00 • 19

# Good News / Bad News

Good News - very general & interesting class problems

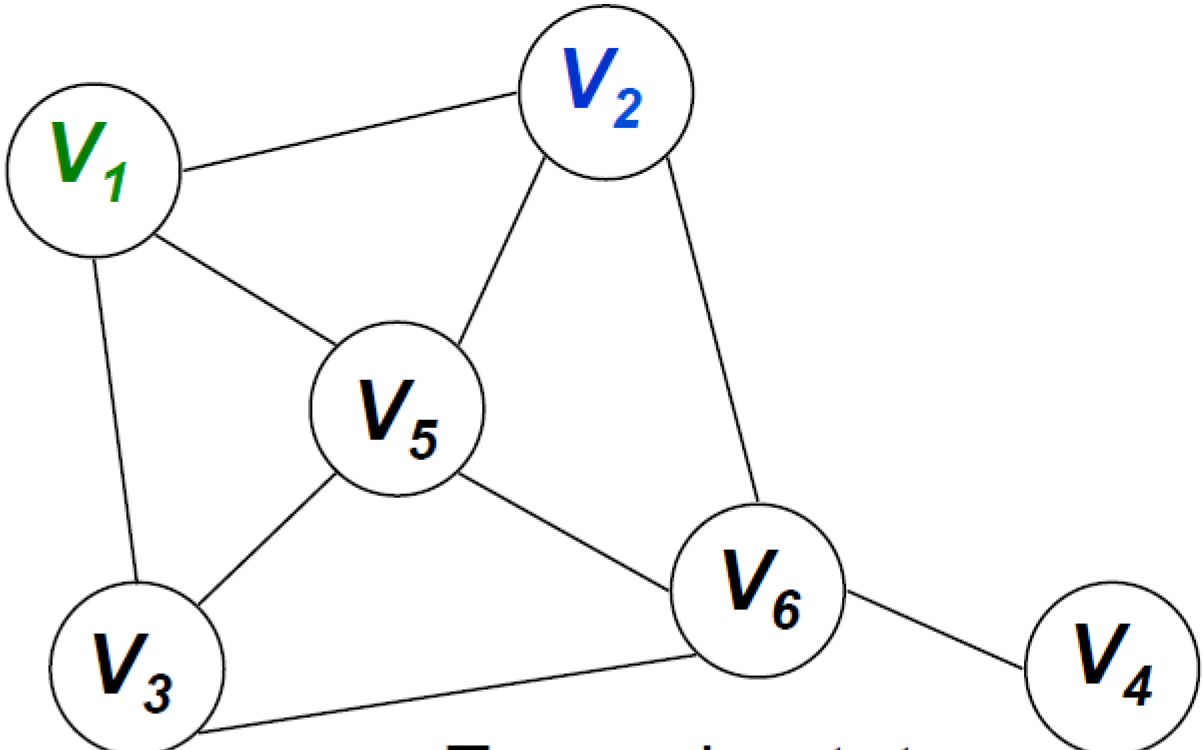
Bad News - includes NP-Hard (intractable) problems

So, good behavior is a function of domain not the formulation as CSP.

# Standard search formulation (incremental)

- **States:**
  - Variables and values assigned so far
- **Initial state:**
  - The empty assignment
- **Action:**
  - Choose any unassigned variable and assign to it a value that does not violate any constraints
    - Fail if no legal assignments
- **Goal test:**
  - The current assignment is complete and satisfies all constraints

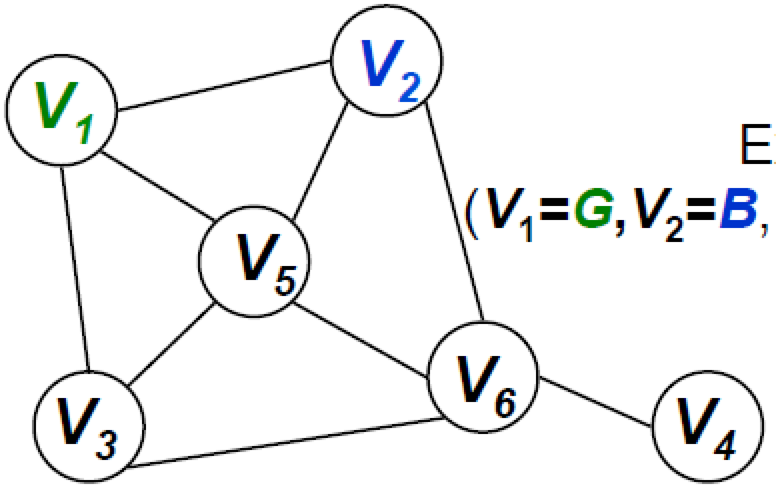
# CSP as a Standard search problem




Example state:

($V_1$=G,$V_2$=B, $V_3$=?, $V_4$=?, $V_5$=?, $V_6$=?)

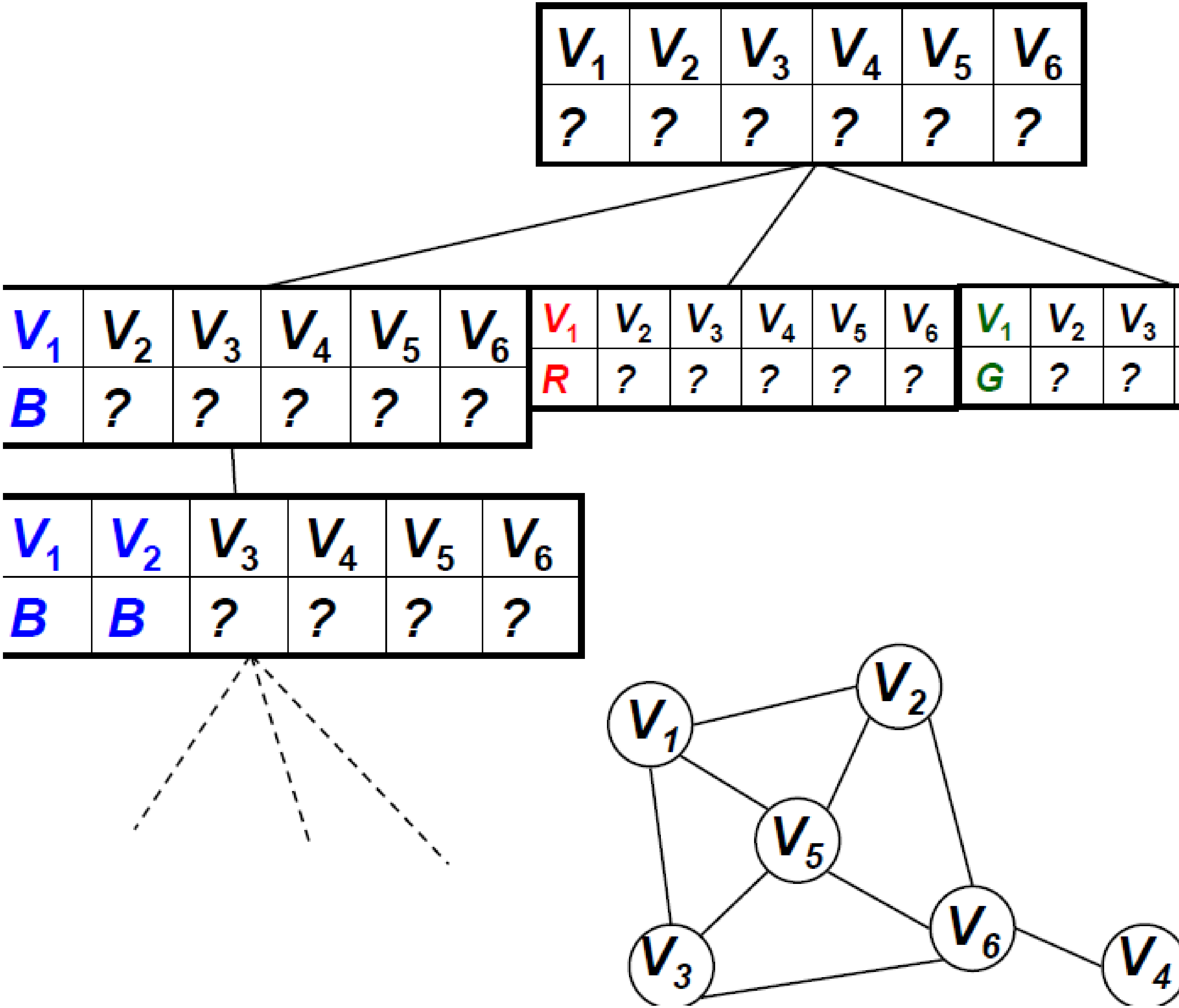
# CSP as a Standard search problem




Example state:

$(\mathbf{V_1}=\mathbf{G}, \mathbf{V_2}=\mathbf{B}, V_3=?, V_4=?, V_5=?, V_6=?)$

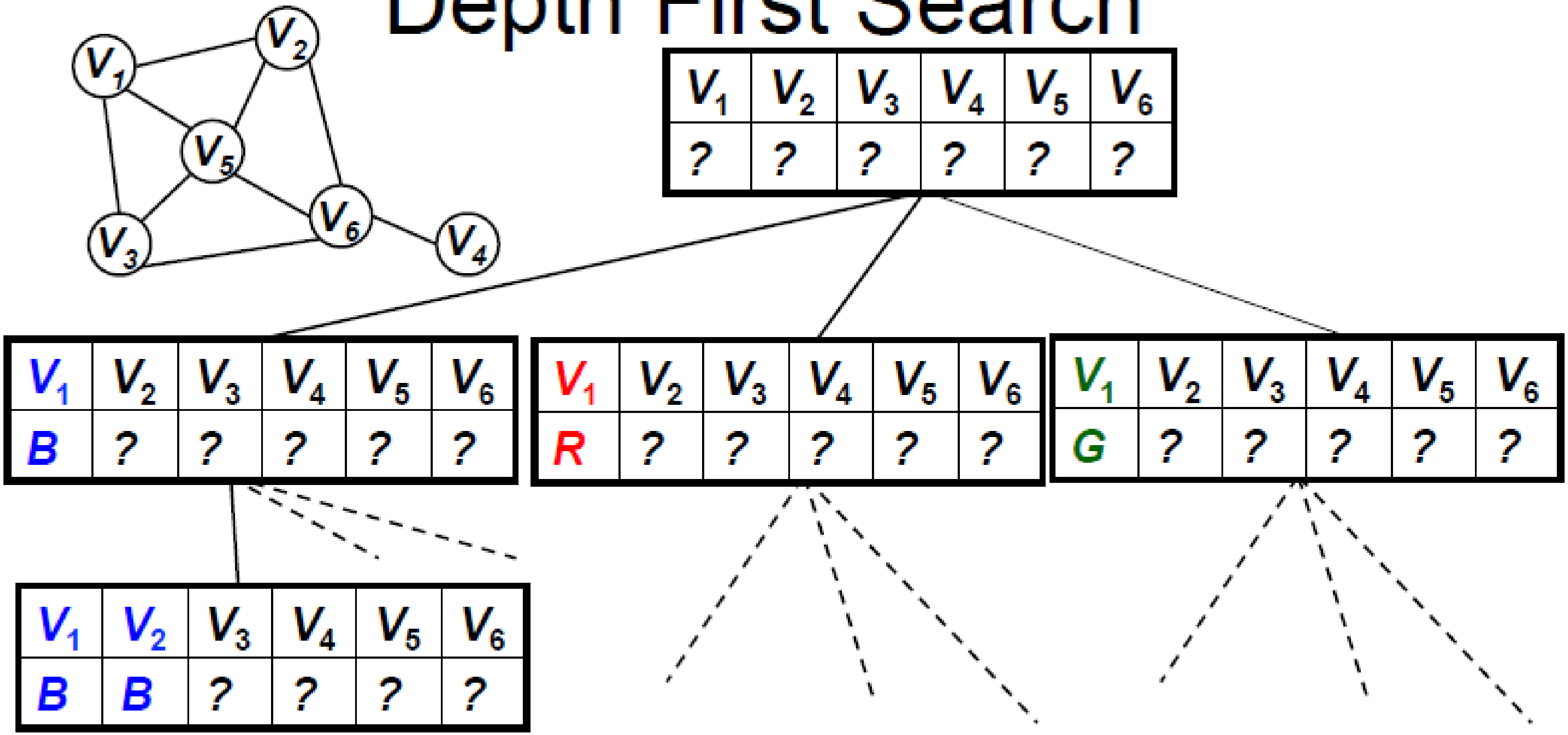
- *State*: assignment to *k* variables with *k*+1,..,*N* unassigned
- *Successor*: Assignment of a value to variable *k*+1, keeping the others unchanged
- *Start state*: $(V_1=?, V_2=?, V_3=?, V_4=?, V_5=?, V_6=?)$
- *Goal state*: All variables assigned with constraints satisfied
- No concept of cost on transition → **just a solution, no path**



V1 V2 V3 V4 V5 V6
? ? ? ? ? ?
V1 V2 V3 V4 V5 V6
B ? ? ? ? ?
V1 V2 V3 V4 V5 V6
R ? ? ? ? ?
V1 V2 V3 V4 V5 V6
G ? ? ? ? ?
V1 V2 V3 V4 V5 V6
B B ? ? ? ?
V1
V2
V3
V4
V5
V6


9d

# Depth First Search




- Recursively:
  - For every possible value in *D:*
    - Set the next unassigned variable in the successor to that value
    - Evaluate the successor of the current state with this variable assignment
    - Stop as soon as a solution is found

# Standard search formulation (incremental)

- What is the depth of any solution (assuming ***n*** variables)?

  ***n*** (this is good)

- Given that there are ***m*** possible values for any variable, how many paths are there in the search tree?

  $n! \cdot m^n$ (this is bad)

- How can we reduce the branching factor?

# Solving CSPs

Solving CSPs involves some combination of:

1. Constraint propagation, to eliminate values that could not be part of any solution
2. Search, to explore valid assignments

# Backtracking DFS

For every possible value *x* in *D:*

– If assigning *x* to the next unassigned variable *V*k+1 does not violate any constraint with the *k* already assigned variables:

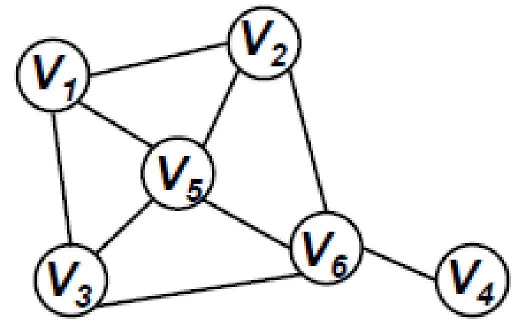
- Set the variable *V*k+1 to *x*
- Evaluate the successors of the current state with this variable assignment

- If no valid assignment is found:

Backtrack to previous state

- Stop as soon as a solution is found

# Backtracking DFS




| $V_1$ | $V_2$ | $V_3$ | $V_4$ | $V_5$ | $V_6$ |
|---|---|---|---|---|---|
| ? | ? | ? | ? | ? | ? |

| $V_1$ | $V_2$ | $V_3$ | $V_4$ | $V_5$ | $V_6$ |
|---|---|---|---|---|---|
| ***B*** | ? | ? | ? | ? | ? |

Order of values: (***B***,***R***,***G***)

| $V_1$ | $V_2$ | $V_3$ | $V_4$ | $V_5$ | $V_6$ |
|---|---|---|---|---|---|
| ***B*** | ***B*** | ? | ? | ? | ? |

Don't even consider that branch because $V_2$=***B*** is inconsistent with the parent state

| $V_1$ | $V_2$ | $V_3$ | $V_4$ | $V_5$ | $V_6$ |
|---|---|---|---|---|---|
| ***B*** | ***R*** | ? | ? | ? | ? |

| $V_1$ | $V_2$ | $V_3$ | $V_4$ | $V_5$ | $V_6$ |
|---|---|---|---|---|---|
| ***B*** | ***R*** | ***R*** | ***B*** | ? | ? |

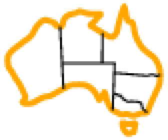
| $V_1$ | $V_2$ | $V_3$ | $V_4$ | $V_5$ | $V_6$ |
|---|---|---|---|---|---|
| ***B*** | ***R*** | ***R*** | ***B*** | ***G*** | ? |

Backtrack to the previous state because no valid assignment can be found for $V_6$

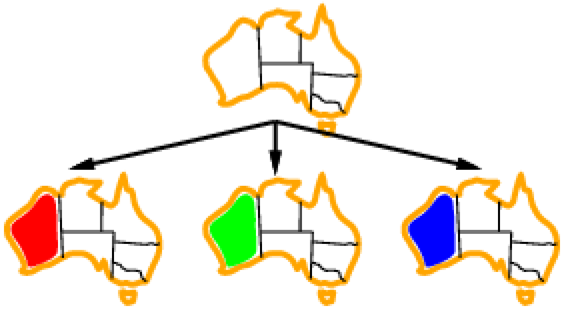
# Backtracking search algorithm

```
function RECURSIVE-BACKTRACKING(assignment, csp)
    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)
        if value is consistent with assignment given CONSTRAINTS[csp]
            add {var = value} to assignment
            result ← RECURSIVE-BACKTRACKING(assignment, csp)
            if result ≠ failure then return result
            remove {var = value} from assignment
    return failure
```

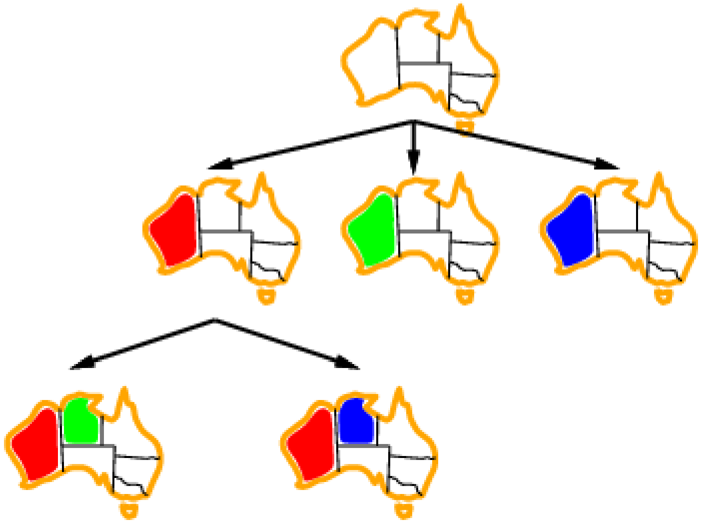
# Example



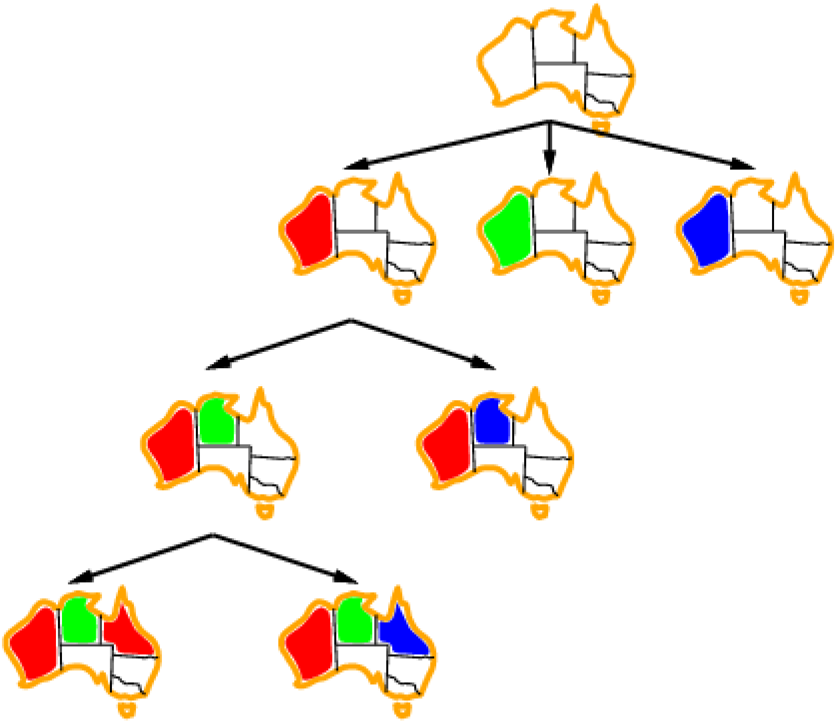
# Example



# Example



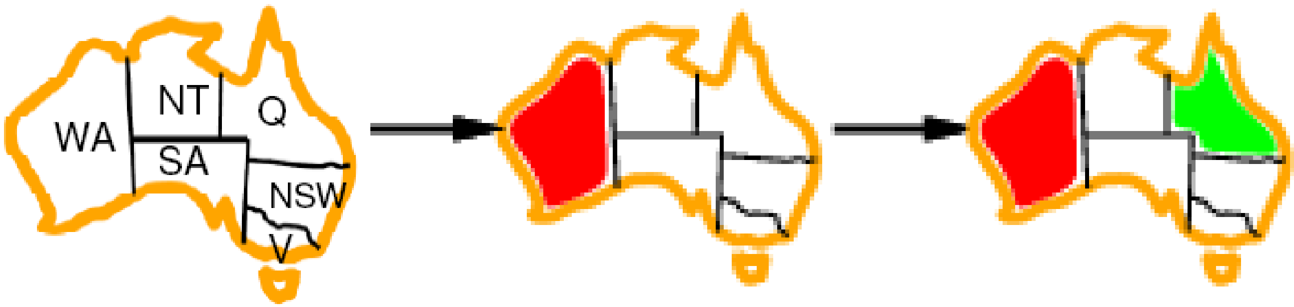
# Example



# Improving Backtracking Efficiency

- Making backtracking search efficient:
  - Can we detect inevitable failure early?
  - Which variable should be assigned next?
  - In what order should its values be tried?

# Early detection of failure




Apply *inference* to reduce the space of possible assignments and detect failure early

# Early detection of failure: Forward checking

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

# Early detection of failure

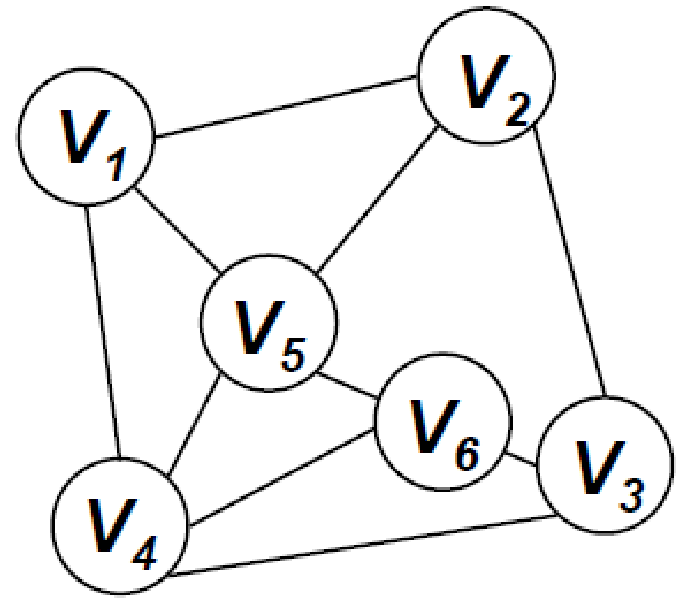
```
function RECURSIVE-BACKTRACKING(assignment, csp)
    if assignment is complete then return assignment
    var ← SELECT-UNASSIGNED-VARIABLE(VARIABLES[csp], assignment, csp)
    for each value in ORDER-DOMAIN-VALUES(var, assignment, csp)
        if value is consistent with assignment given CONSTRAINTS[csp]
            add {var = value} to assignment
            result ← RECURSIVE-BACKTRACKING(assignment, csp)
            if result ≠ failure then return result
            remove {var = value} from assignment
    return failure
```

Apply *inference* to reduce the space of possible assignments and detect failure early

# Forward Checking

- Keep track of remaining legal values for unassigned variables
- Backtrack when any variable has no legal values

| | $V_1$ | $V_2$ | $V_3$ | $V_4$ | $V_5$ | $V_6$ |
|---|---|---|---|---|---|---|
| R | ? | ? | ? | ? | ? | ? |
| B | ? | ? | ? | ? | ? | ? |
| G | ? | ? | ? | ? | ? | ? |

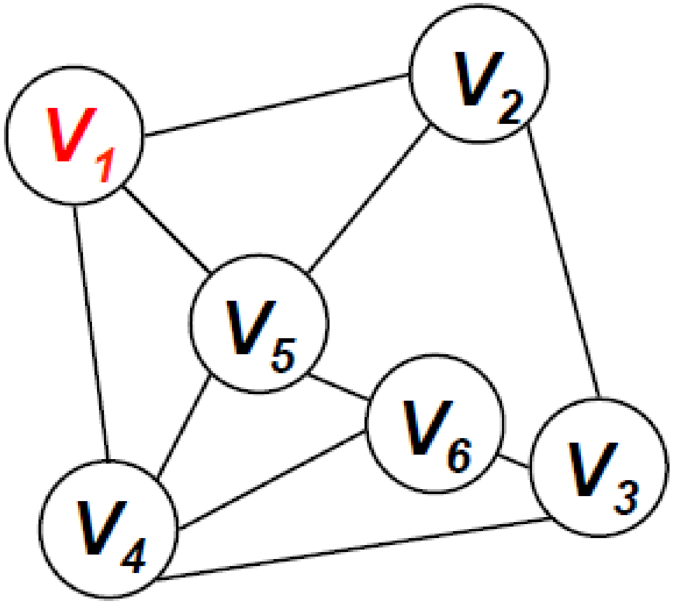



Warning: Different example with order (R,B,G)

# Forward Checking

- Keep track of remaining legal values for unassigned variables
- Backtrack when any variable has no legal values

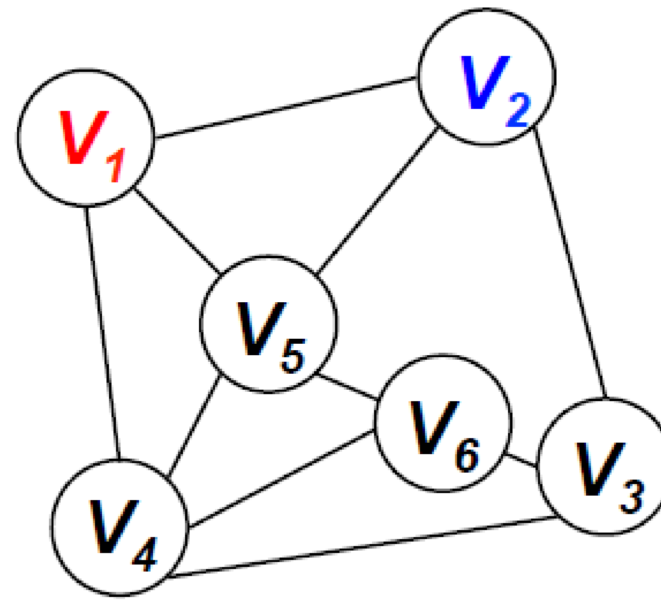
| | $V_1$ | $V_2$ | $V_3$ | $V_4$ | $V_5$ | $V_6$ |
|---|---|---|---|---|---|---|
| R | O | X | ? | X | X | ? |
| B | | ? | ? | ? | ? | ? |
| G | | ? | ? | ? | ? | ? |



# Forward Checking

- Keep track of remaining legal values for unassigned variables
- Backtrack when any variable has no legal values

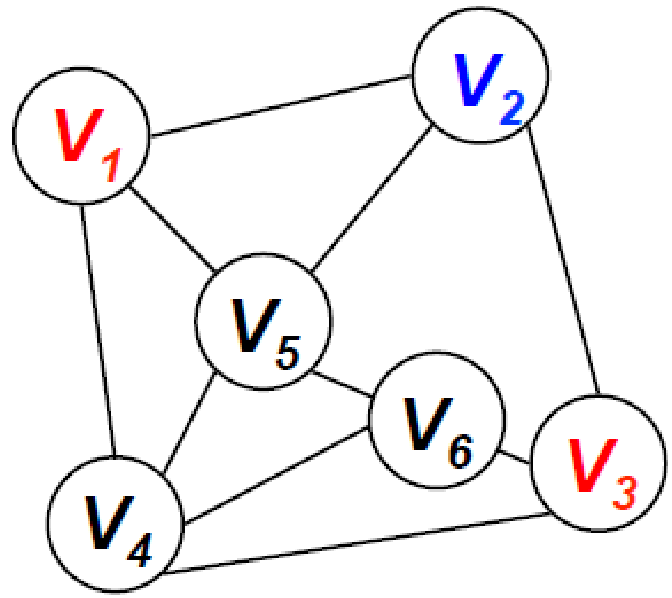
| | $V_1$ | $V_2$ | $V_3$ | $V_4$ | $V_5$ | $V_6$ |
|---|---|---|---|---|---|---|
| R | O | | ? | X | X | ? |
| B | | O | X | ? | X | ? |
| G | | | ? | ? | ? | ? |



# Forward Checking

- Keep track of remaining legal values for unassigned variables
- Backtrack when no variable has a legal value

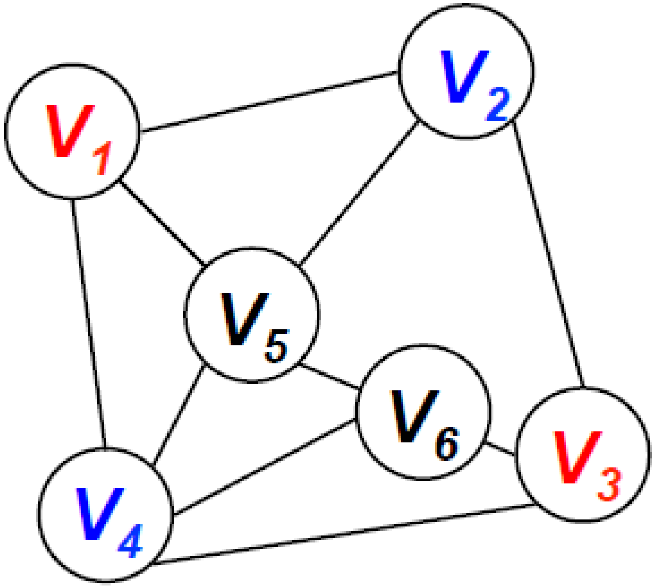
| | $V_1$ | $V_2$ | $V_3$ | $V_4$ | $V_5$ | $V_6$ |
|---|---|---|---|---|---|---|
| R | O | | O | X | X | X |
| B | | O | | ? | X | ? |
| G | | | | ? | ? | ? |



# Forward Checking

- Keep track of remaining legal values for unassigned variables
- Backtrack when any variable has no legal values

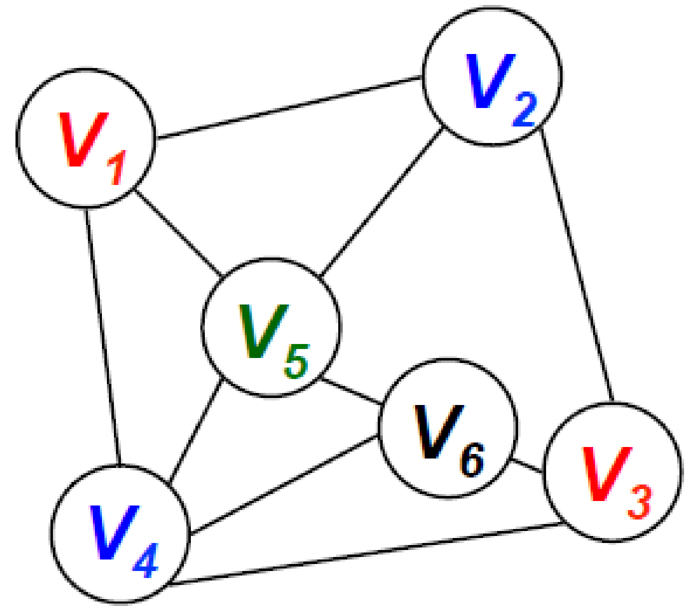
| | $V_1$ | $V_2$ | $V_3$ | $V_4$ | $V_5$ | $V_6$ |
|---|---|---|---|---|---|---|
| R | O | | O | | X | X |
| B | | O | | O | X | X |
| G | | | | | ? | ? |



# Forward Checking

- Keep track of remaining legal values for unassigned variables
- Backtrack when any variable has no legal values

| | $V_1$ | $V_2$ | $V_3$ | $V_4$ | $V_5$ | $V_6$ |
|---|---|---|---|---|---|---|
| *R* | *O* | | *O* | | | *X* |
| *B* | | *O* | | *O* | | *X* |
| *G* | | | | | *O* | *X* |




There are no valid assignments left for $V_6$ we need to backtrack

# Constraint Propagation (aka Arc Consistency)

Arc consistency eliminates values from domain of variable that can never be part of a consistent solution.

$$V_i \rightarrow V_j$$

Directed arc $(V_i, V_j)$ is arc consistent if
$\forall x \in D_i \; \exists y \in D_j$ such that (x,y) is allowed by the constraint on the arc

We can achieve consistency on arc by deleting values form $D_i$ (domain of variable at tail of constraint arc) that fail this condition.

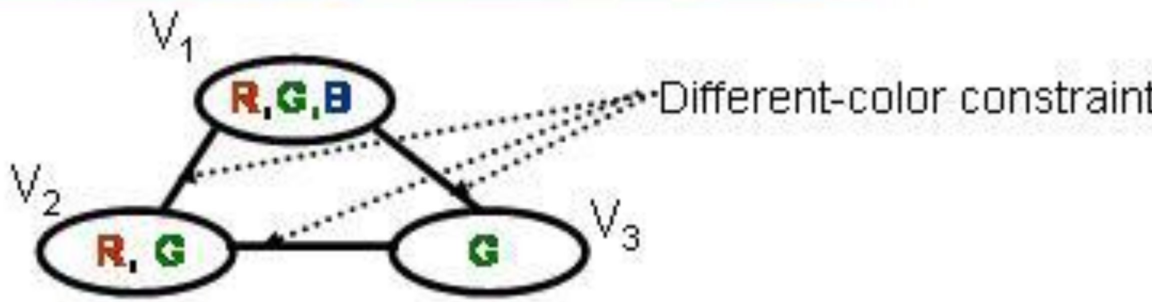
Assume domains are size at most d and there are e binary constraints.

A simple algorithm for arc consistency is $O(ed^3)$ – note that just verifying arc consistency takes $O(d^2)$ for each arc.

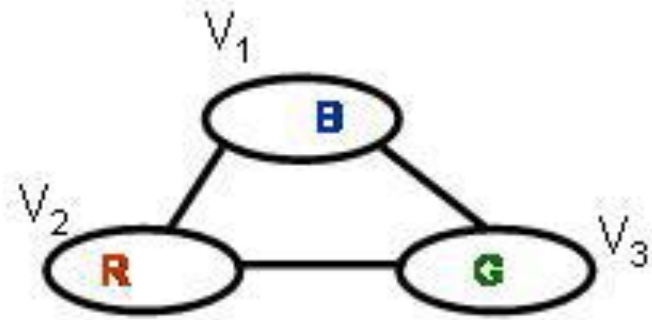
# Constraint Propagation Example

**Graph Coloring**

Initial Domains are indicated

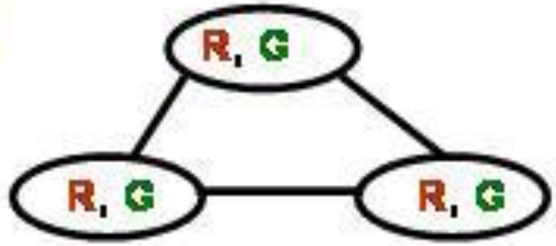



| Arc examined | Value deleted |
| --- | --- |
| $V_1 - V_2$ | none |
| $V_1 - V_3$ | $V_1$(**G**) |
| $V_2 - V_3$ | $V_2$(**G**) |
| $V_1 - V_2$ | $V_1$(**R**) |
| $V_1 - V_3$ | none |
| $V_2 - V_3$ | none |

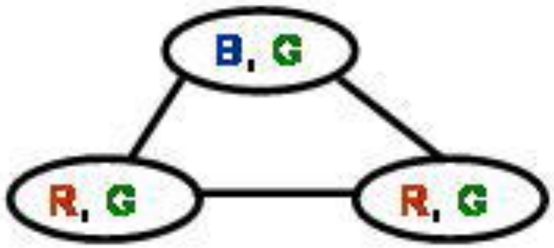


# But, arc consistency is not enough in general

## Graph Coloring

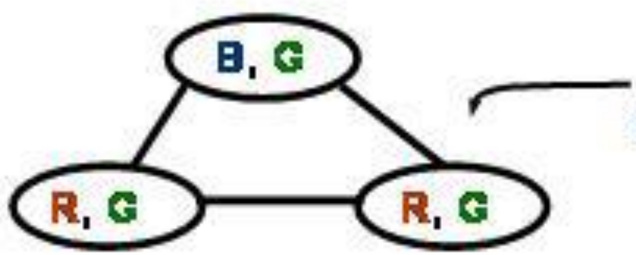



arc consistent but <u>no</u> solutions




arc consistent but <u>2</u> solutions B,R,G ; B,G,R .

arc consistent but <u>1</u> solution

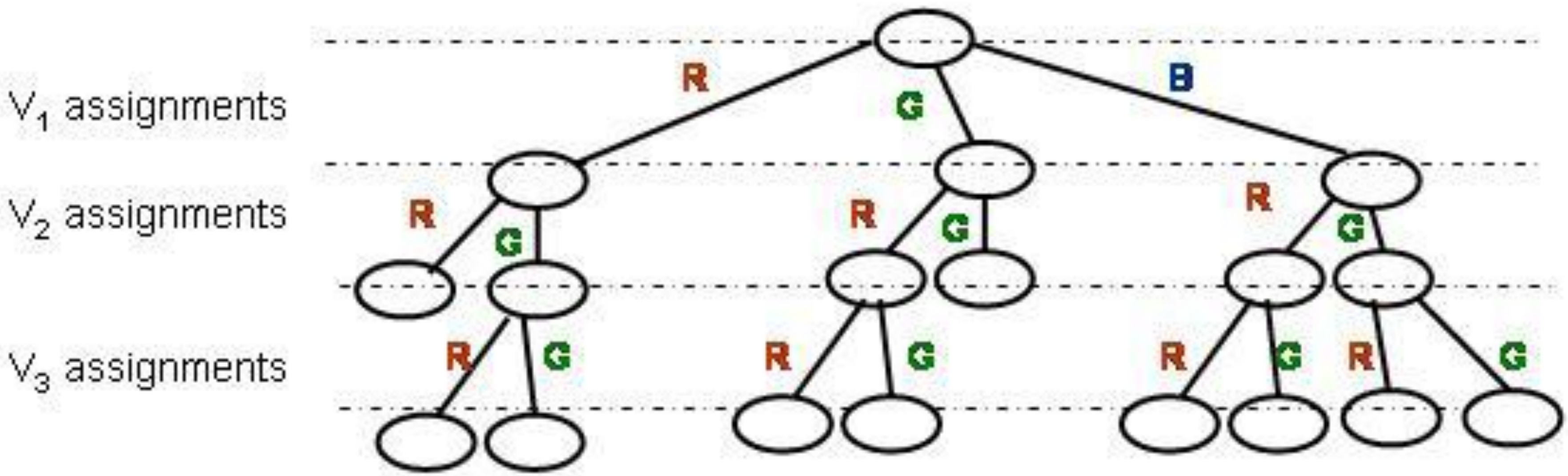


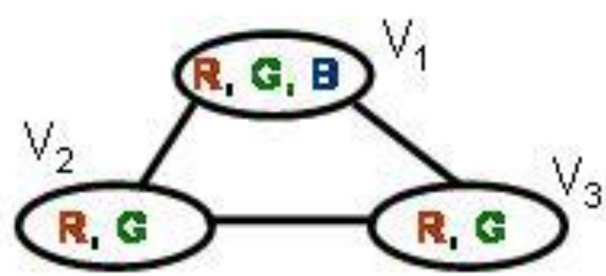

B, R not allowed

Need to do search to find solutions (if any)

# Searching for solutions – backtracking (BT)

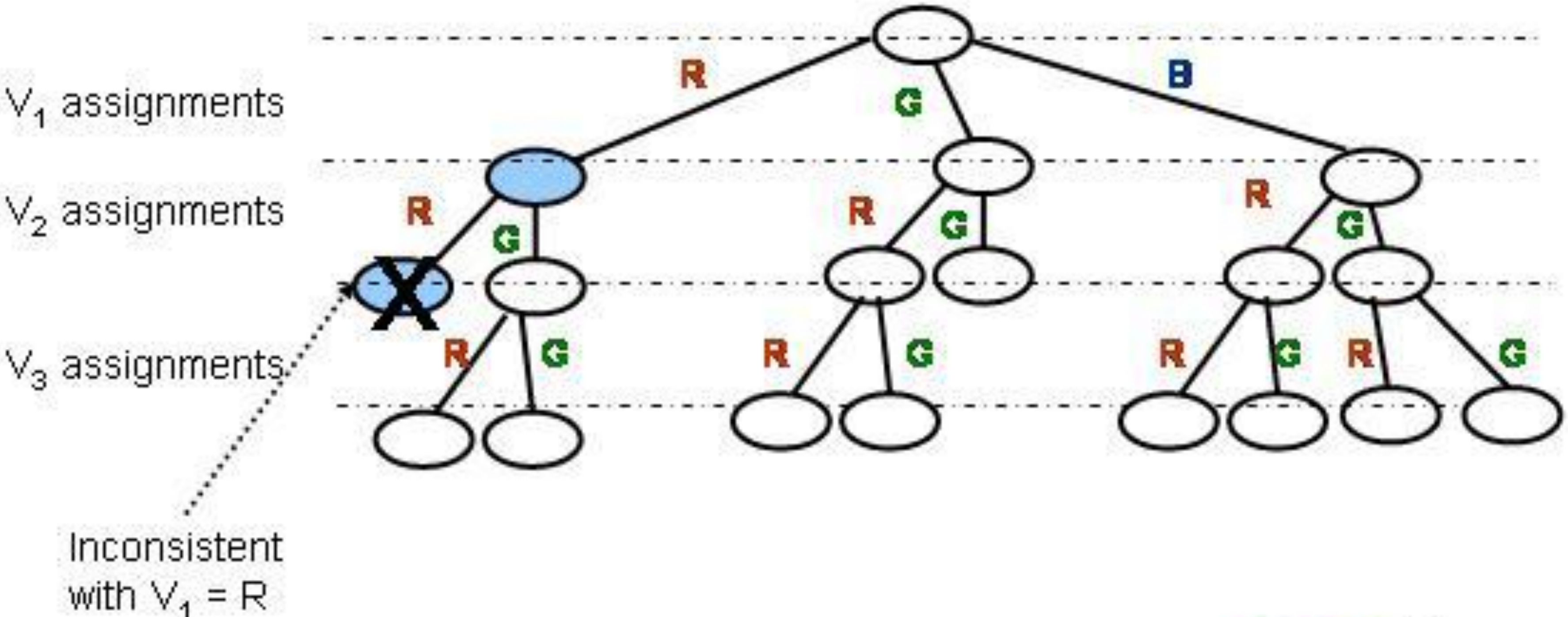
When we have too many values in domain (and/or constraints are weak) arc consistency doesn't do much, so we need to search. Simplest approach is pure backtracking (depth-first search).



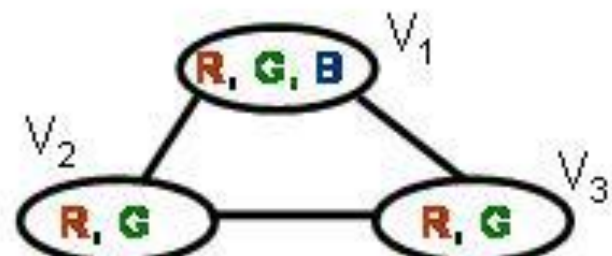



# Searching for solutions – backtracking (BT)

**When we have too many values in domain (and/or constraints are weak) arc consistency doesn't do much, so we need to search. Simplest approach is pure backtracking (depth-first search).**

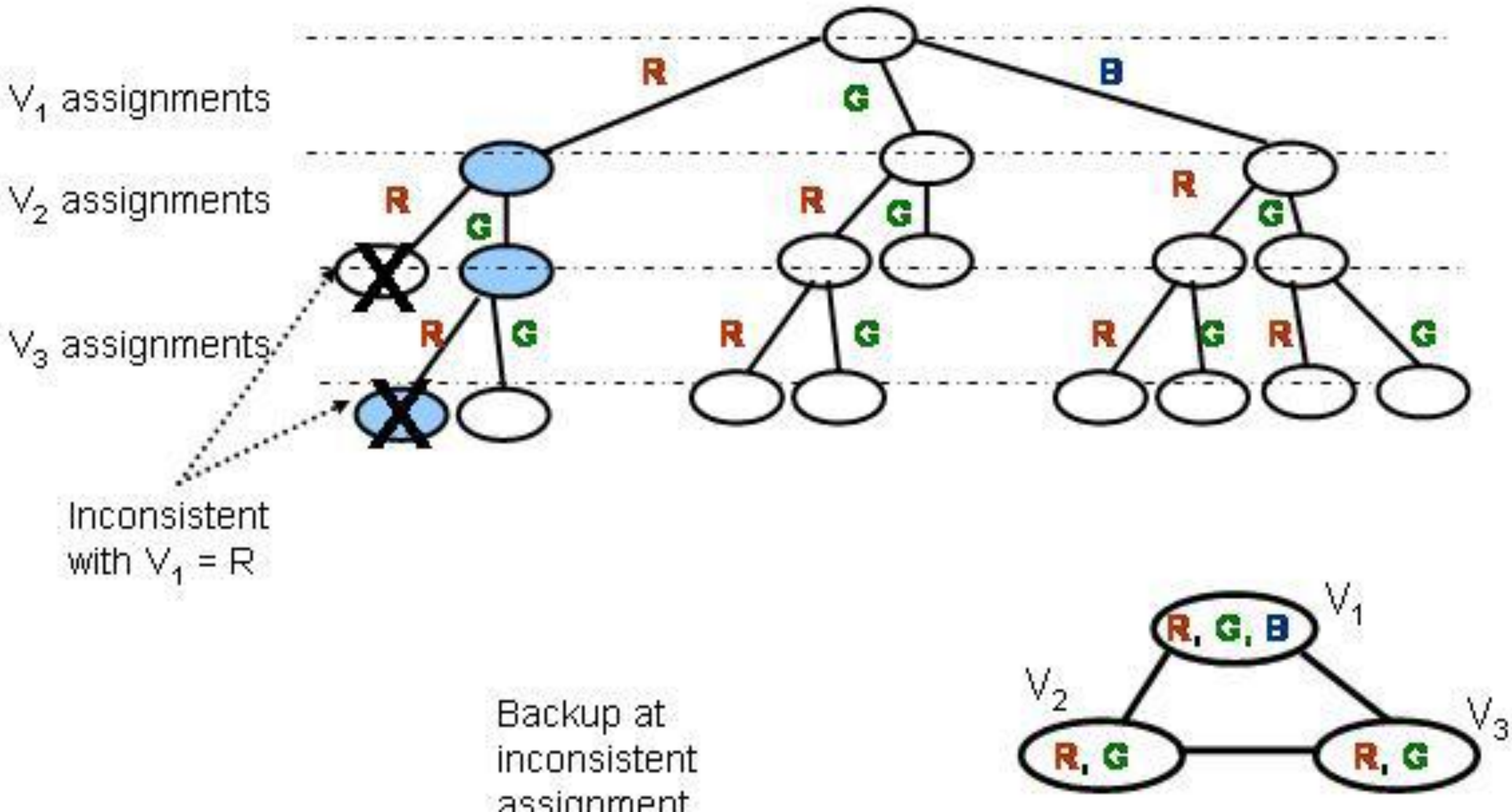



Backup at inconsistent assignment



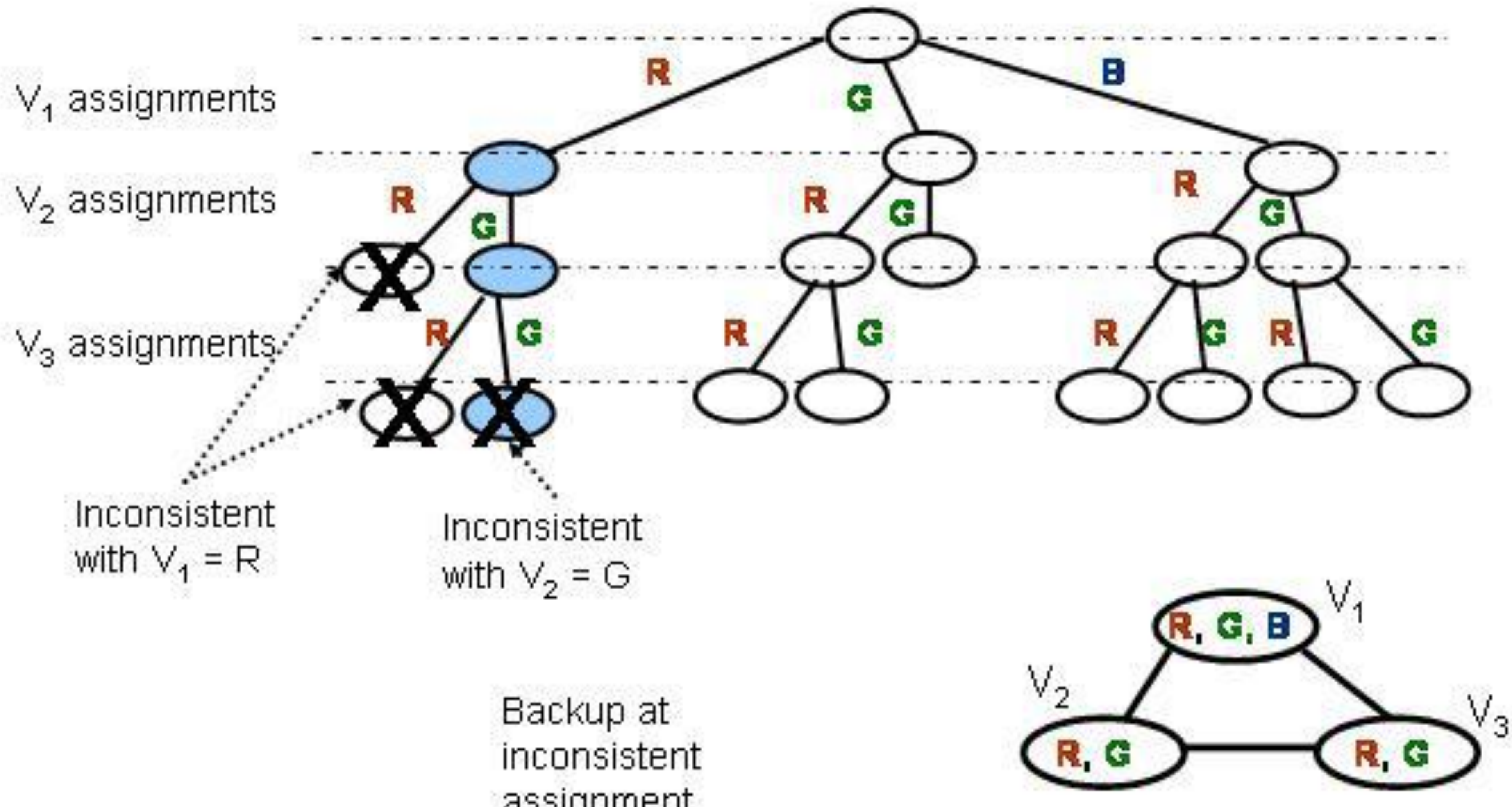
# Searching for solutions – backtracking (BT)

**When we have too many values in domain (and/or constraints are weak) arc consistency doesn't do much, so we need to search. Simplest approach is pure backtracking (depth-first search).**



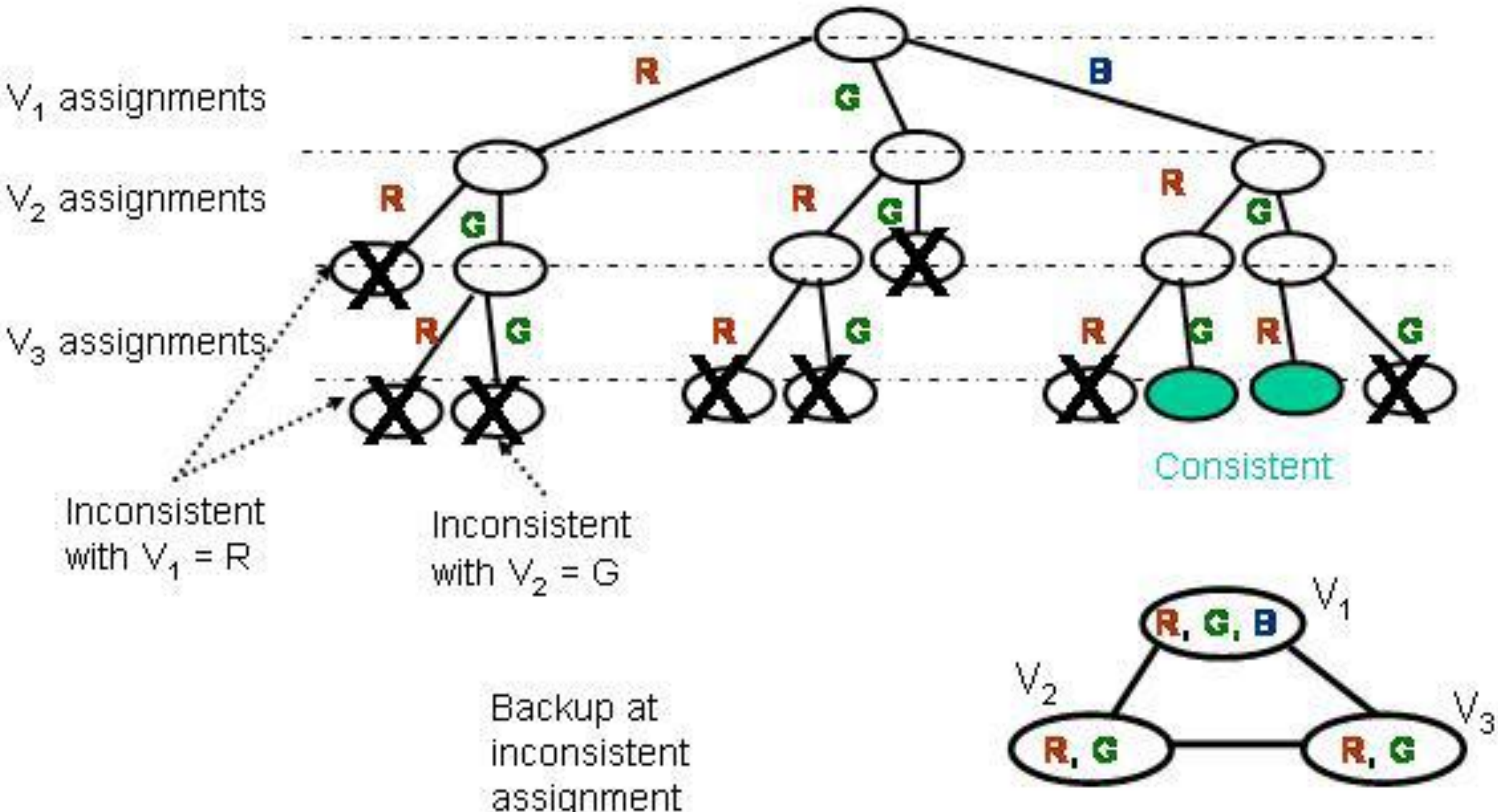
# Searching for solutions – backtracking (BT)

When we have too many values in domain (and/or constraints are weak) arc consistency doesn't do much, so we need to search. Simplest approach is pure backtracking (depth-first search).



# Searching for solutions – backtracking (BT)

**When we have too many values in domain (and/or constraints are weak) arc consistency doesn't do much, so we need to search. Simplest approach is pure backtracking (depth-first search).**



# Combine Backtracking & Constraint Propagation

A node in BT tree is partial assignment in which the domain of each variable has been set (tentatively) to singleton set.

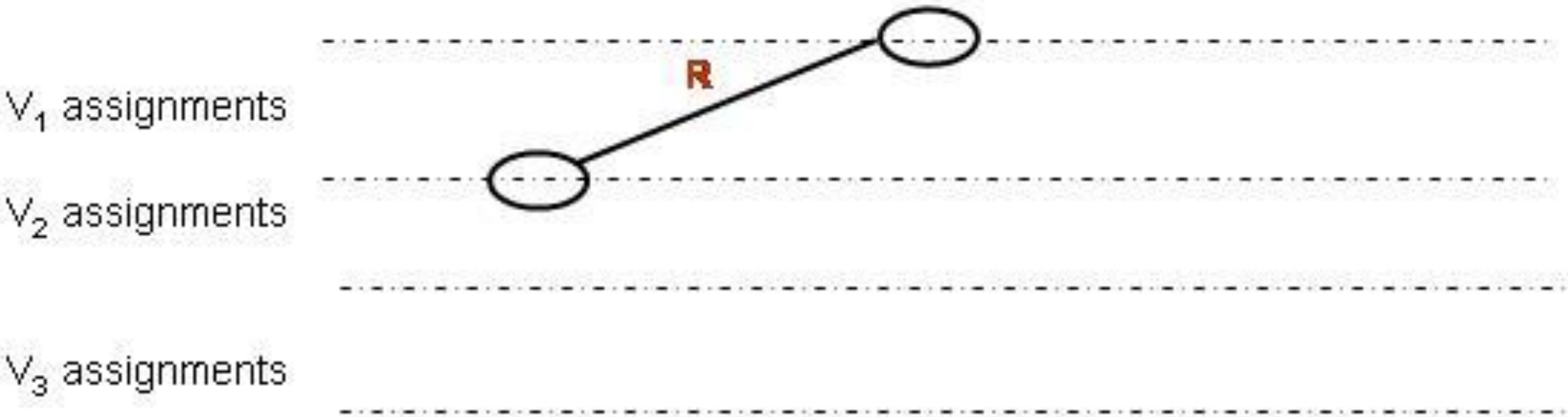
Use constraint propagation (arc-consistency) to propagate the effect of this tentative assignment, i.e., eliminate values inconsistent with current values.

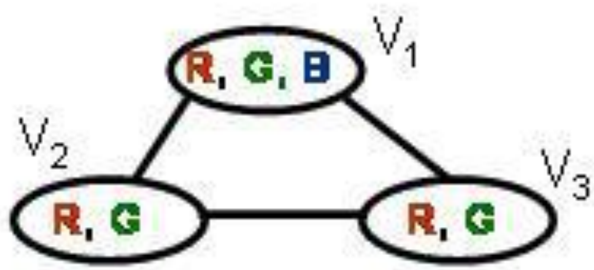
**Question:** How much propagation to do?

**Answer:** Not much, just local propagation from domains with unique assignments, which is called forward checking (FC). This conclusion is not necessarily obvious, but it generally holds in practice.

# Backtracking with Forward Checking (BT-FC)

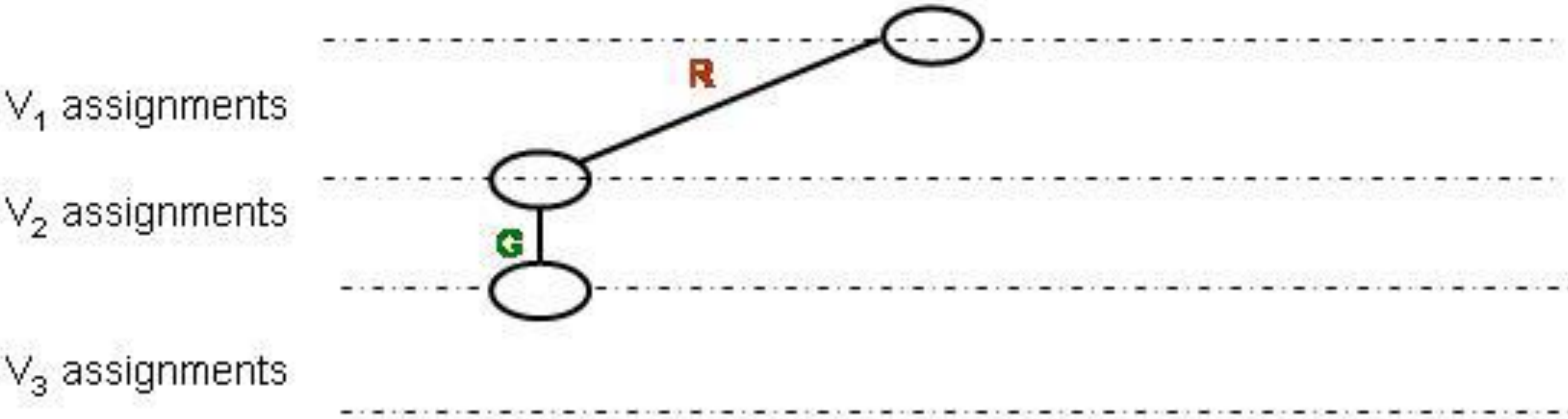
**When examining assignment $V_i = d_k$, remove any values inconsistent with that assignment from neighboring domains in constraint graph.**

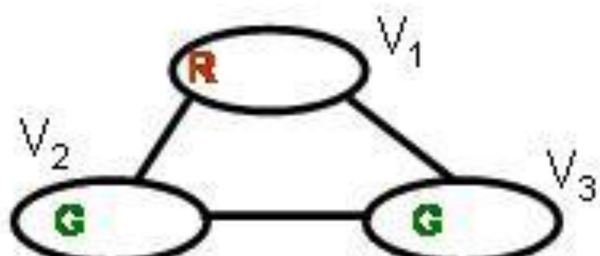


# Backtracking with Forward Checking (BT-FC)

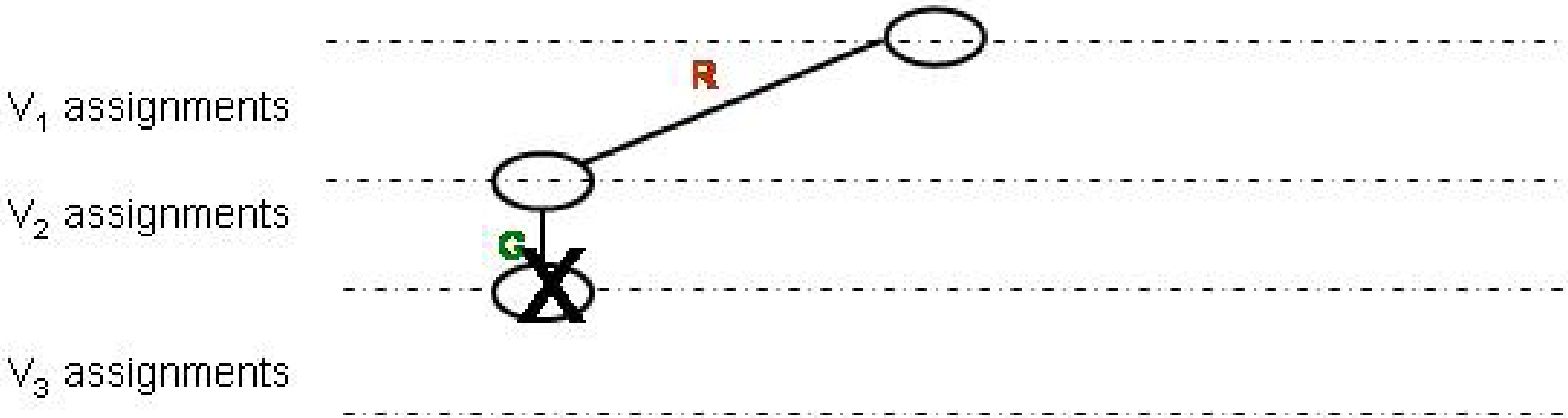
**When examining assignment $V_i = d_k$, remove any values inconsistent with that assignment from neighboring domains in constraint graph.**



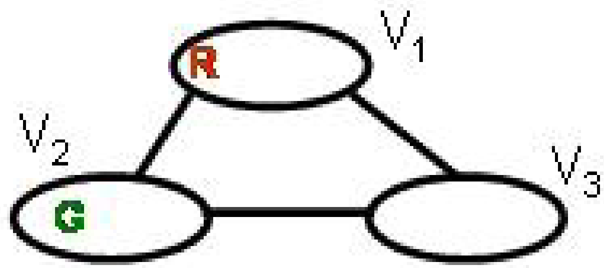



# Backtracking with Forward Checking (BT-FC)

**When examining assignment $V_i = d_k$, remove any values inconsistent with that assignment from neighboring domains in constraint graph.**

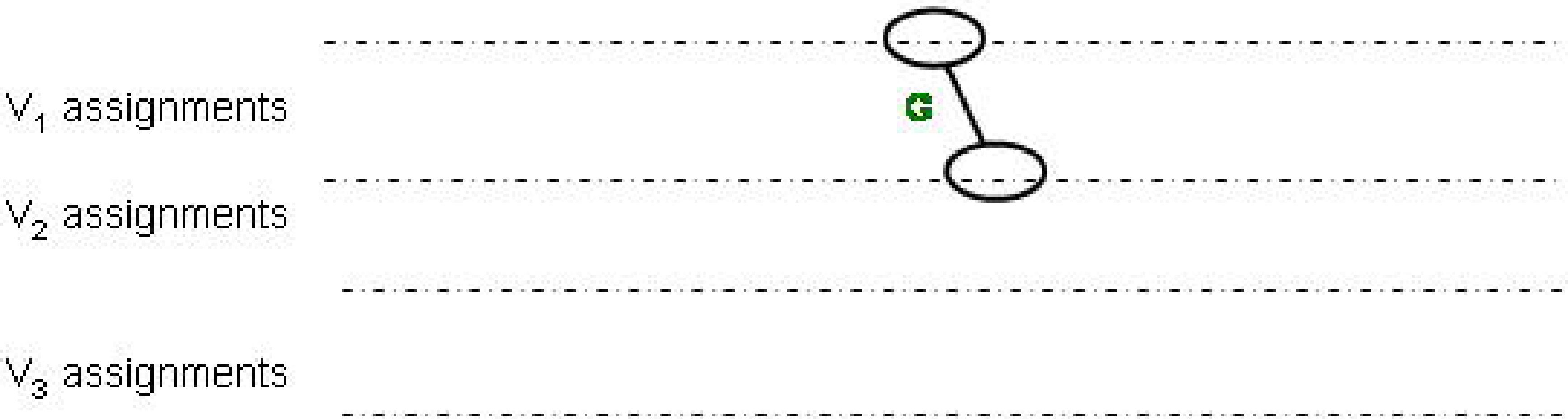



We have a conflict whenever a domain becomes empty.

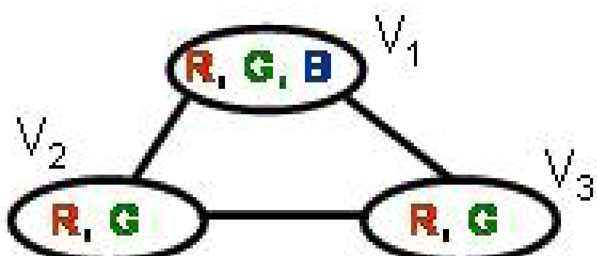


# Backtracking with Forward Checking (BT-FC)

**When examining assignment $V_i = d_k$, remove any values inconsistent with that assignment from neighboring domains in constraint graph.**

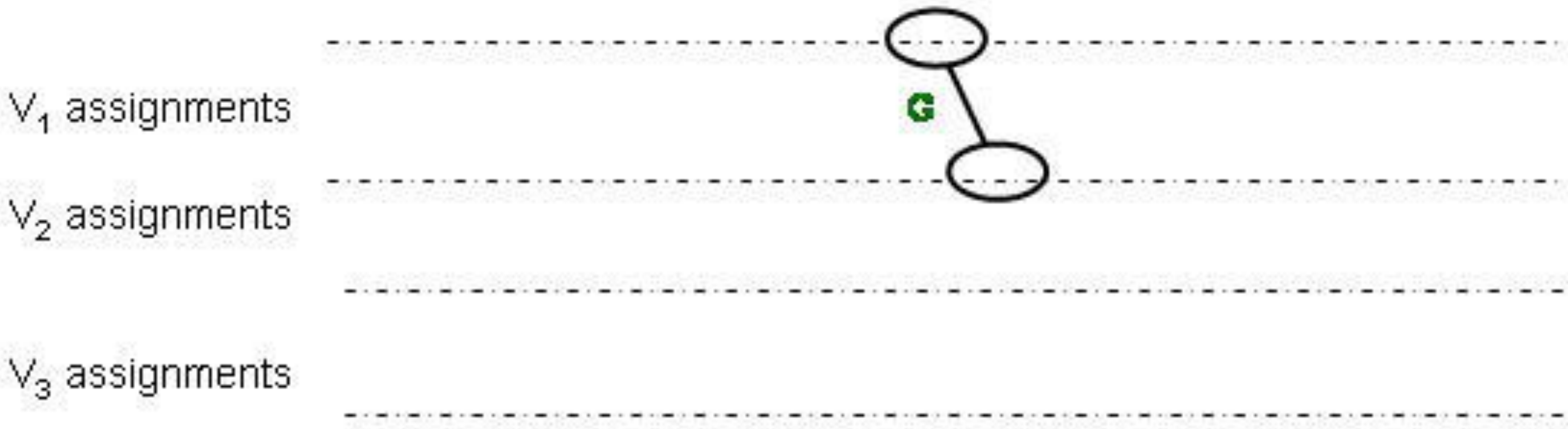


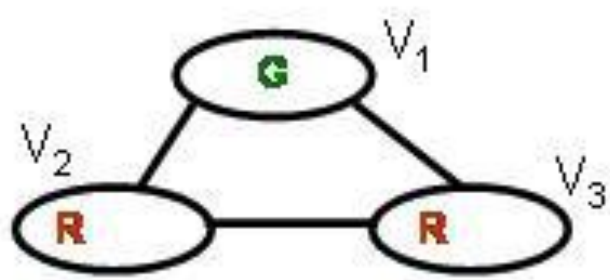

When backing up, need to restore domain values, since deletions were done to reach consistency with tentative assignments considered during search.



# Backtracking with Forward Checking (BT-FC)

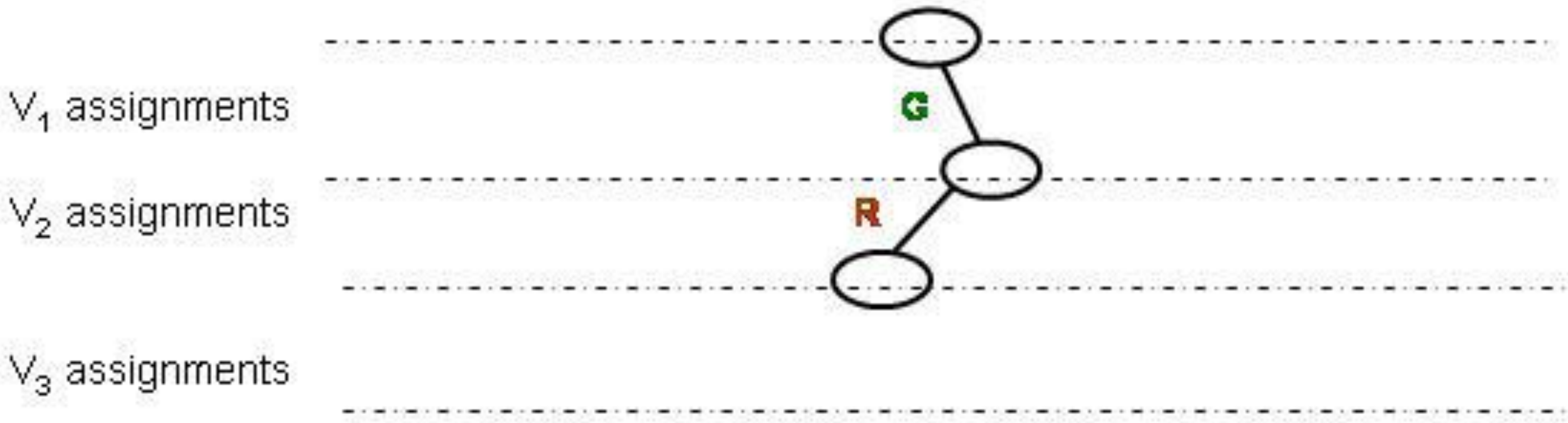
When examining assignment $V_i=d_k$, remove any values inconsistent with that assignment from neighboring domains in constraint graph.

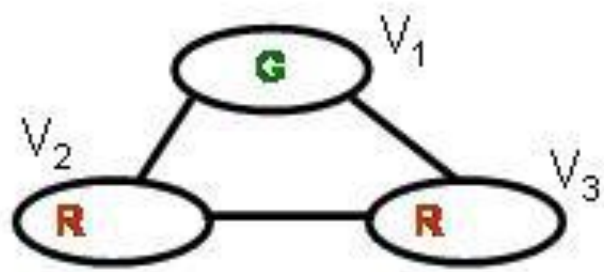


# Backtracking with Forward Checking (BT-FC)

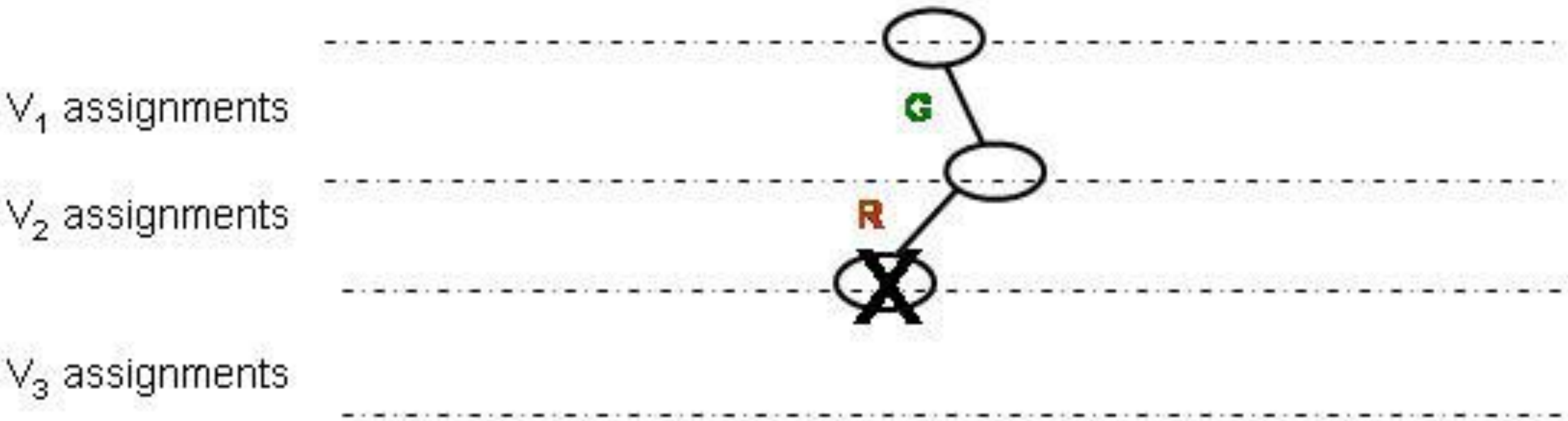
When examining assignment $V_i = d_k$, remove any values inconsistent with that assignment from neighboring domains in constraint graph.

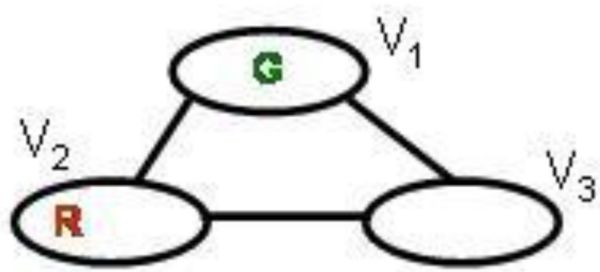


# Backtracking with Forward Checking (BT-FC)

**When examining assignment $V_i=d_k$, remove any values inconsistent with that assignment from neighboring domains in constraint graph.**



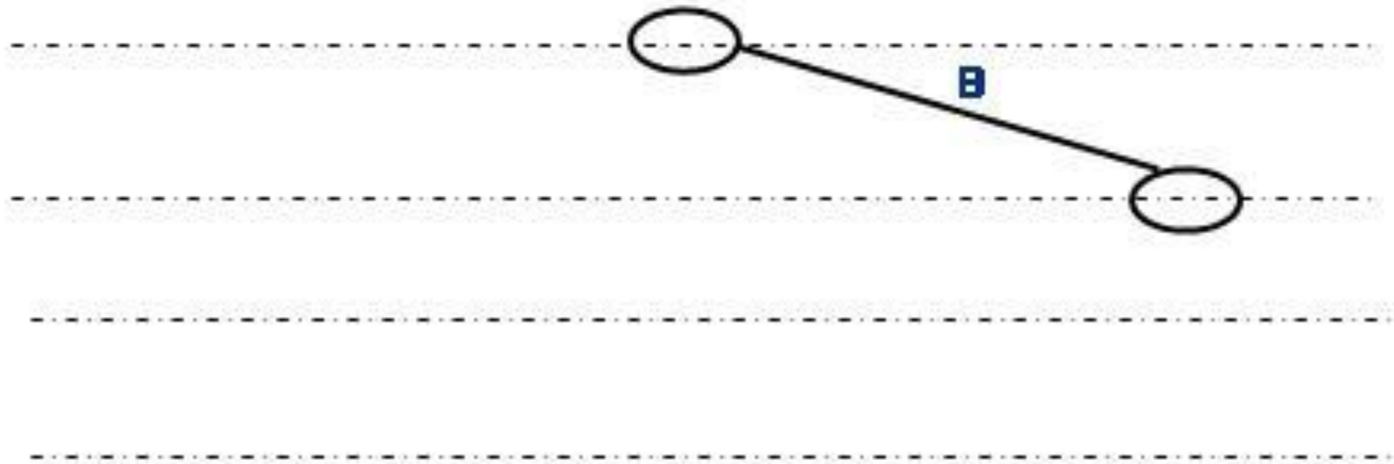
# Backtracking with Forward Checking (BT-FC)

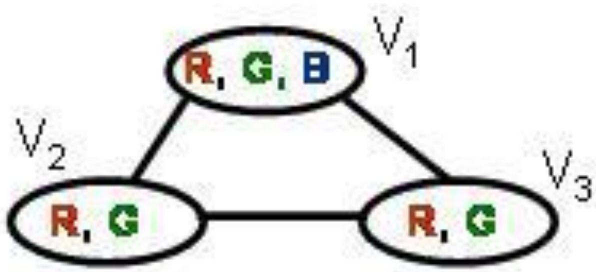
**When examining assignment $V_i=d_k$, remove any values inconsistent with that assignment from neighboring domains in constraint graph.**

$V_1$ assignments

$V_2$ assignments

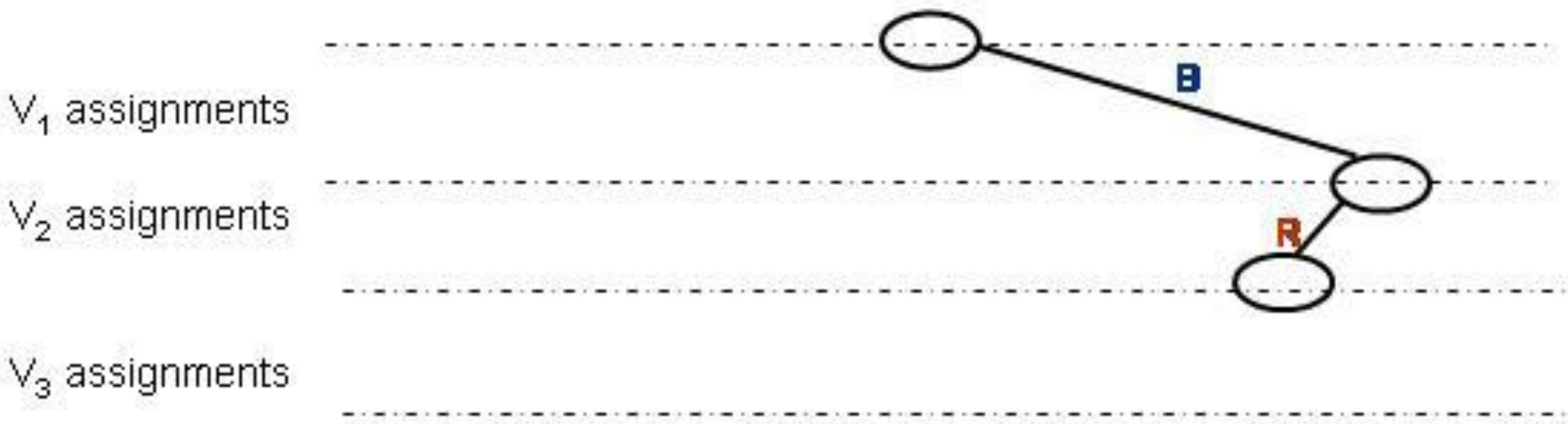
$V_3$ assignments

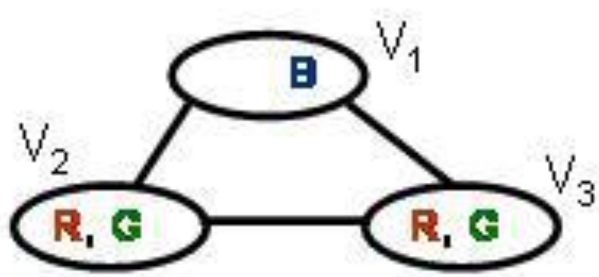


# Backtracking with Forward Checking (BT-FC)

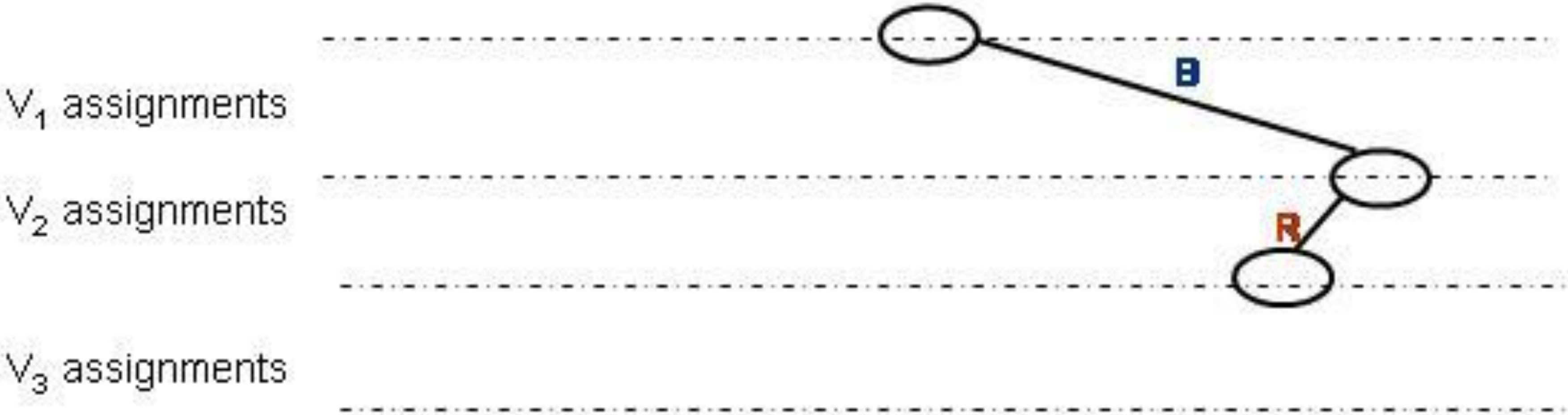
**When examining assignment $V_i = d_k$, remove any values inconsistent with that assignment from neighboring domains in constraint graph.**

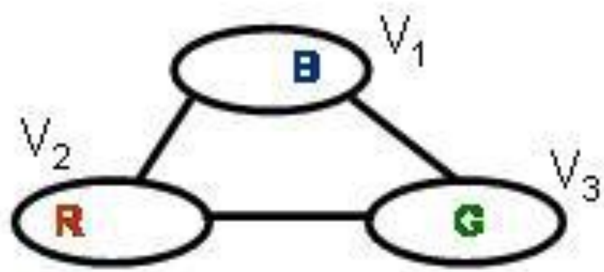


# Backtracking with Forward Checking (BT-FC)

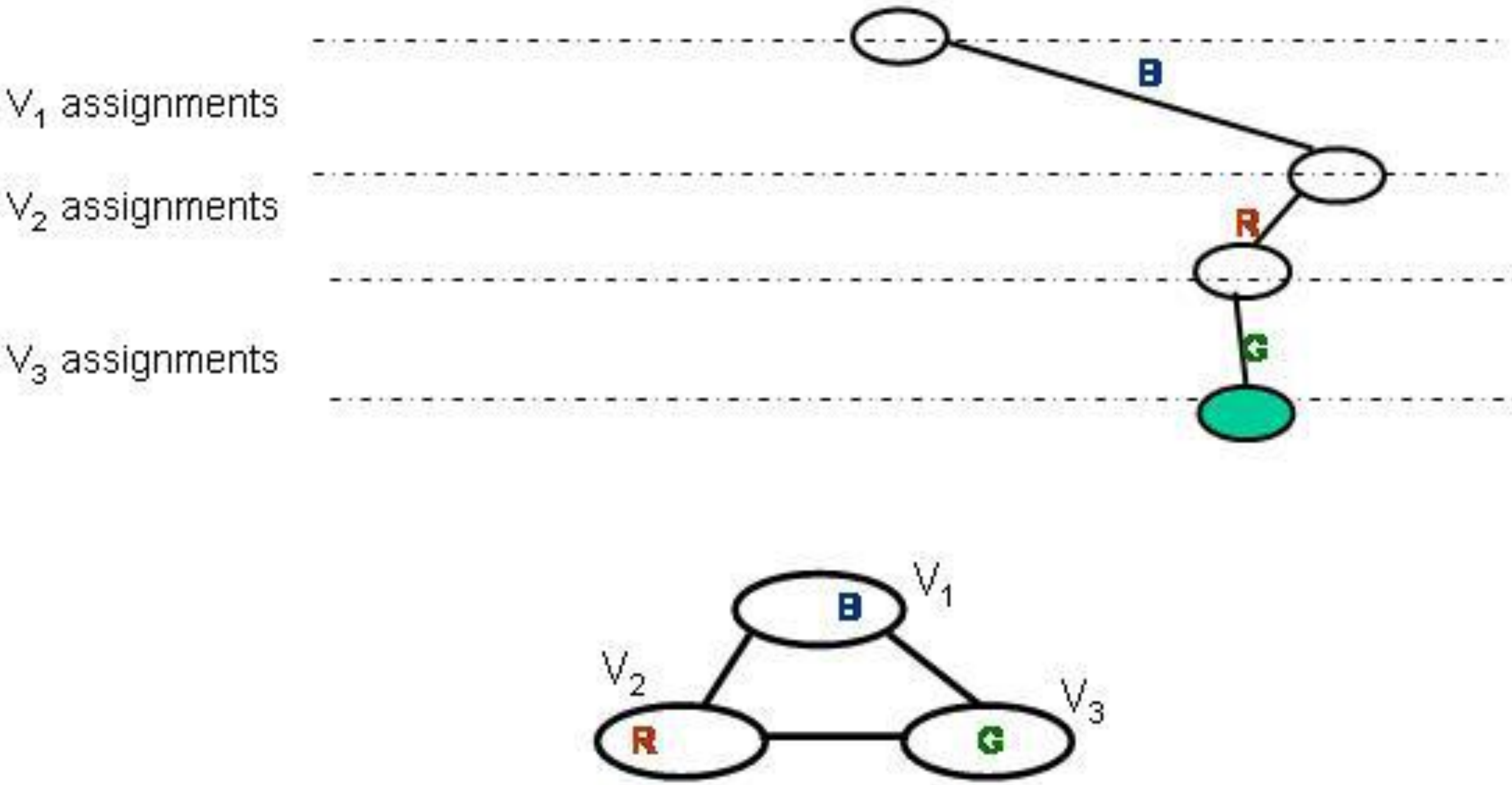
**When examining assignment $V_i = d_k$, remove any values inconsistent with that assignment from neighboring domains in constraint graph.**



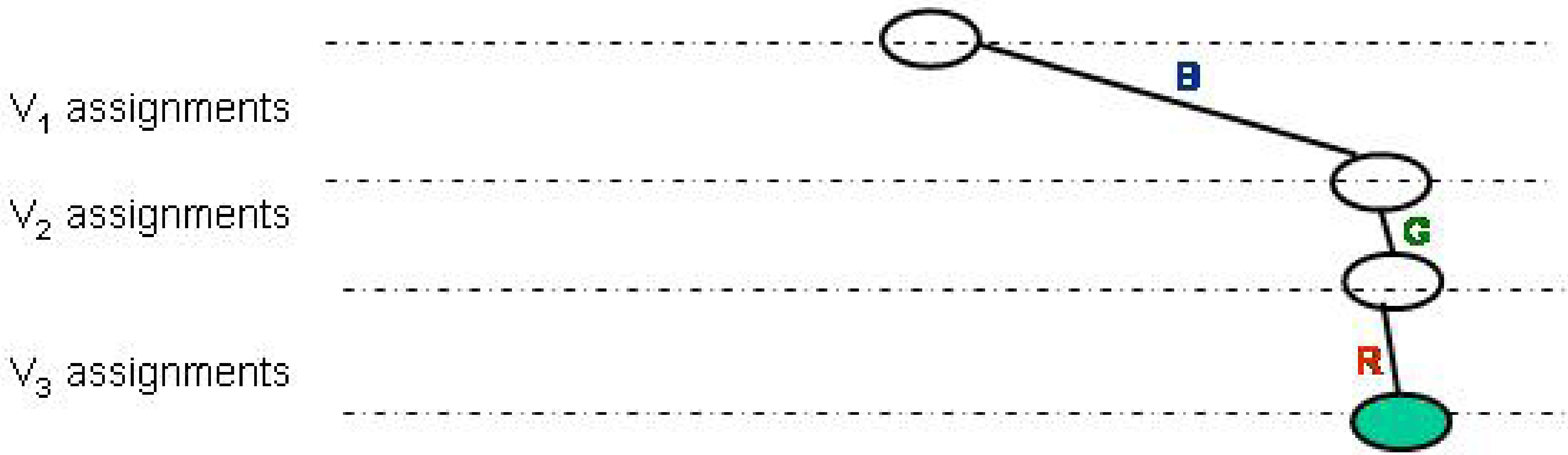
# Backtracking with Forward Checking (BT-FC)

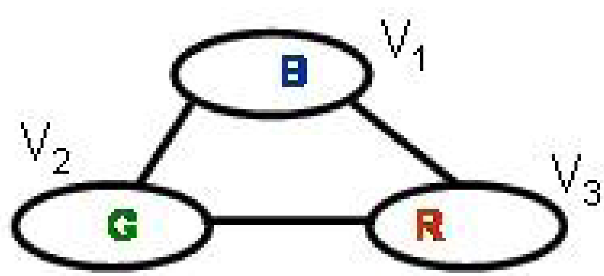
When examining assignment $V_i = d_k$, remove any values inconsistent with that assignment from neighboring domains in constraint graph.



# Backtracking with Forward Checking (BT-FC)

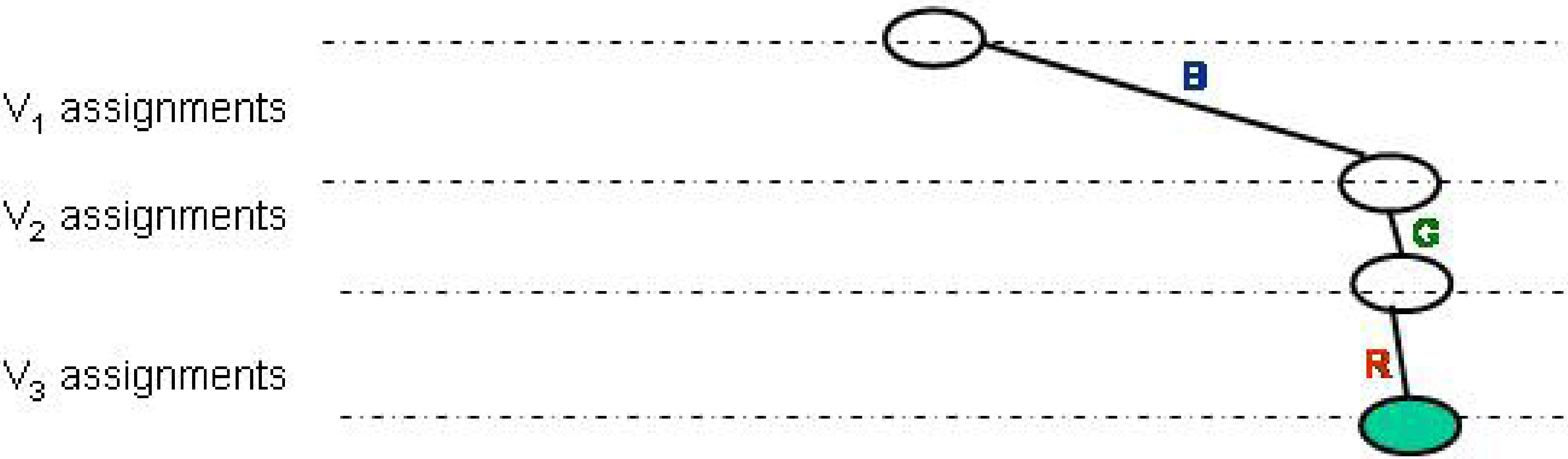
**When examining assignment $V_i=d_k$, remove any values inconsistent with that assignment from neighboring domains in constraint graph.**



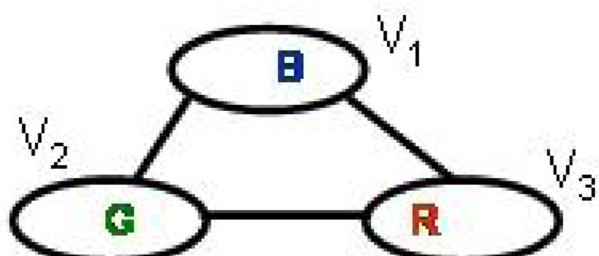



# Backtracking with Forward Checking (BT-FC)

**When examining assignment $V_i=d_k$, remove any values inconsistent with that assignment from neighboring domains in constraint graph.**




**No need to check previous assignments**

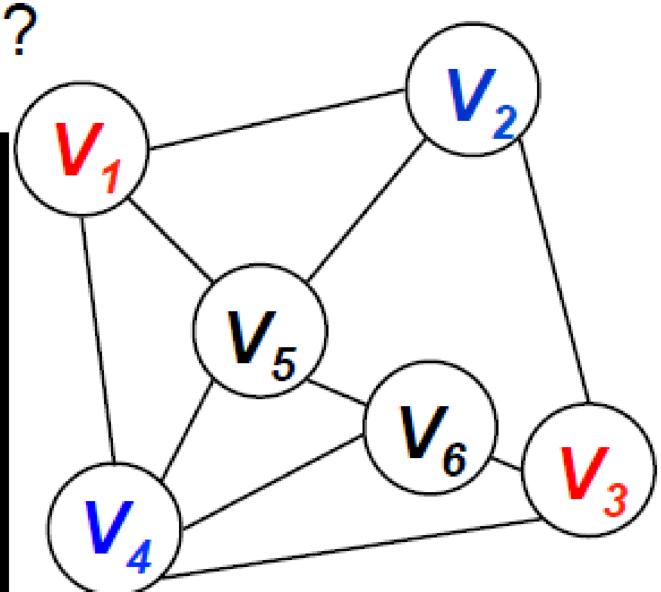



**Generally preferable to pure BT**

# Constraint Propagation

- Forward checking does not detect all the inconsistencies, only those that can be detected by looking at the constraints which contain the current variable.
- Can we look ahead further?

| | $V_1$ | $V_2$ | $V_3$ | $V_4$ | $V_5$ | $V_6$ |
|---|---|---|---|---|---|---|
| **R** | O | | O | | X | X |
| **B** | | O | | O | X | X |
| **G** | | | | | ? | ? |




At this point, it is already obvious that this branch will not lead to a solution because there are no consistent values in the remaining domain for $V_5$ and $V_6$.

# Constraint Propagation, not "just" checking

- ***V*** = variable being assigned at the current level of the search
- Set variable ***V*** to a value in *D*(***V***)
- For every variable ***V'*** connected to ***V***:
  - Remove the values in *D*(***V'***) that are inconsistent with the assigned variables
  - For every variable ***V''*** connected to ***V'***:
    - Remove the values in *D*(***V''***) that are no longer possible candidates
    - And do this again with the variables connected to ***V''***
      - ……..until no more values can be discarded

# Constraint Propagation

New: Constraint Propagation

Forward Checking as before

- ***V*** = variable being assig…
- … ariable ***V*** to a value in … )
- F… every variable ***V'*** conn…cted to ***V***:
  - … move the values in *D*(***V'***) that are inconsistent …th the assigned variables
  - …r every variable ***V''*** connected to ***V'***:
    - Remove the values in *D*(***V''***) that are no longer possible candidates
    - And do this again with the variables connected to ***V''***
      - ……..until no more values can be discarded

# CP for the graph coloring problem

Propagate (*node*, *color*)

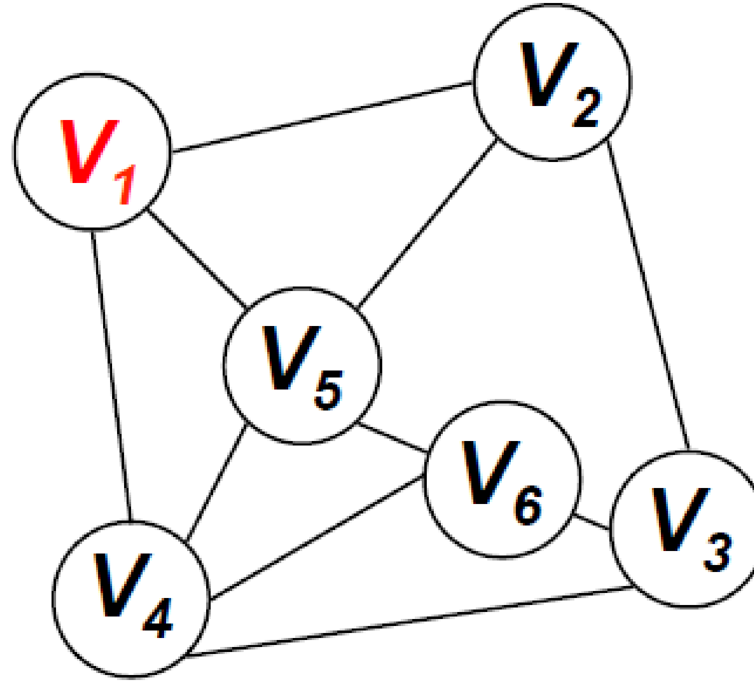
1. Remove color from the domain of all of the neighbors
2. For every neighbor *N*:

    If *D*(*N*) was reduced to only one color after step 1 (*D*(*N*) = {*c*}):

    Propagate (*N*,*c*)

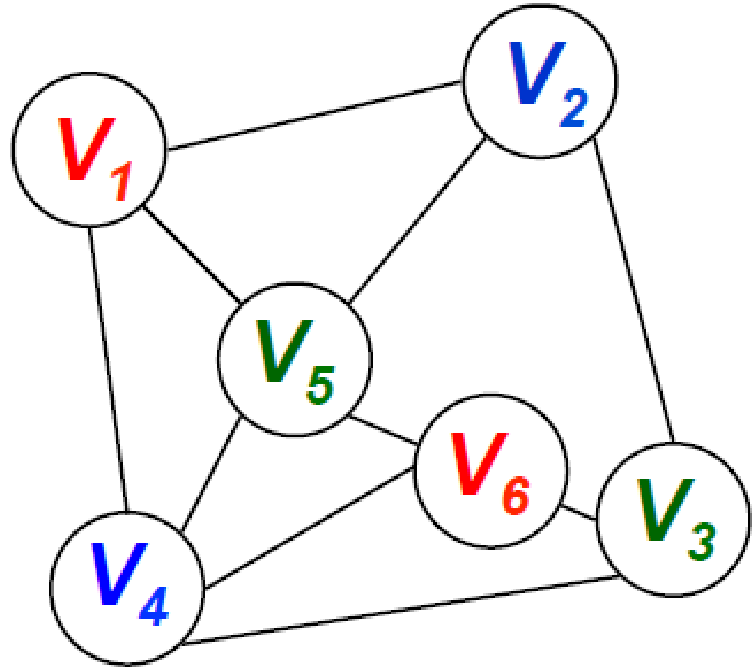
After Propagate ($V_1$, R):

| | $V_1$ | $V_2$ | $V_3$ | $V_4$ | $V_5$ | $V_6$ |
|---|---|---|---|---|---|---|
| **R** | O | X | ? | X | X | ? |
| **B** | | ? | ? | ? | ? | ? |
| **G** | | ? | ? | ? | ? | ? |



After Propagate ($V_2$, **B**):

| | $V_1$ | $V_2$ | $V_3$ | $V_4$ | $V_5$ | $V_6$ |
|---|---|---|---|---|---|---|
| **R** | O | | X | X | X | ? |
| **B** | | O | X | ? | X | X |
| **G** | | | ? | X | ? | X |

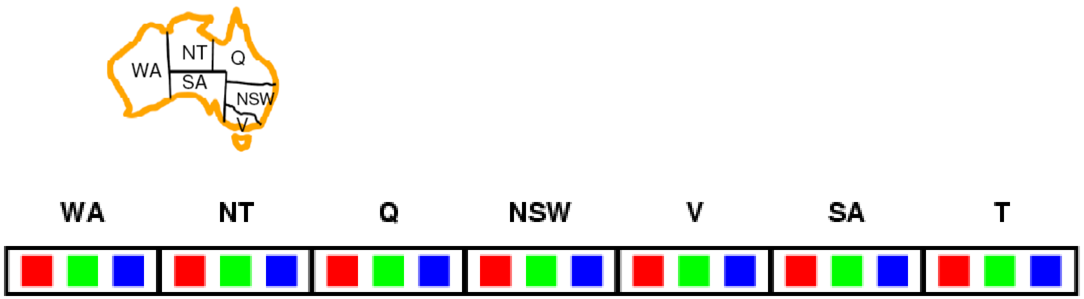



Note: We get directly to a solution in *one step of CP* after setting $V_2$ *without any additional search*

Some problems can even be solved by applying CP directly without search

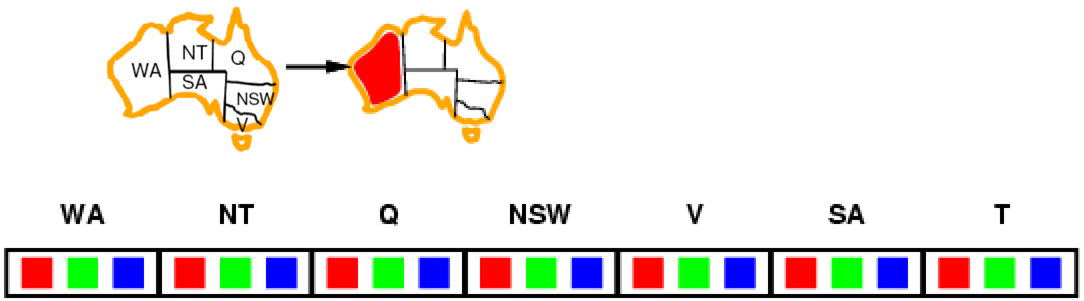
# Early detection of failure: Forward checking

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



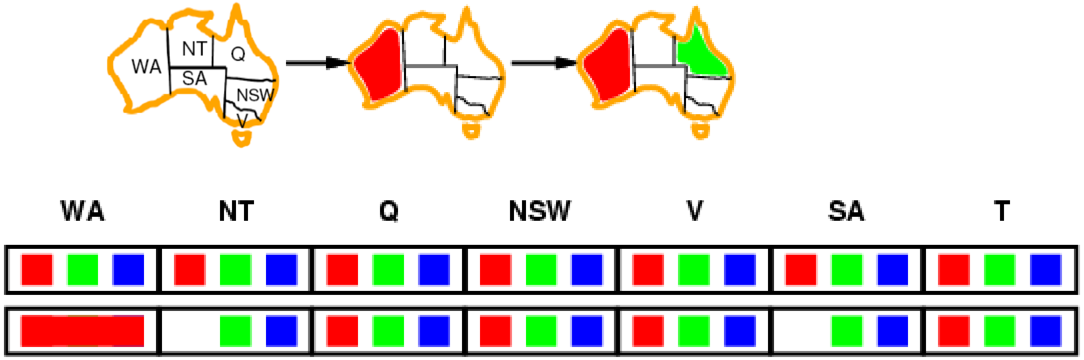
# Early detection of failure: Forward checking

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



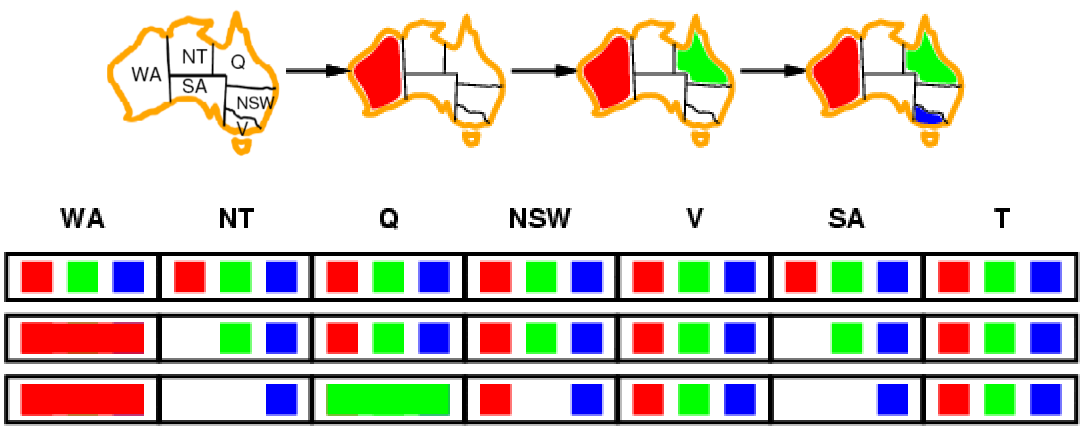
# Early detection of failure: Forward checking

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



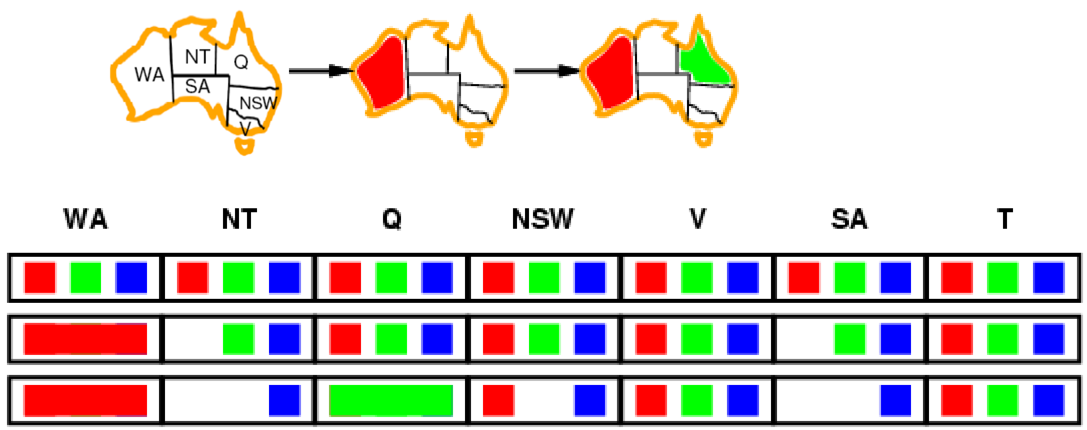
# Early detection of failure: Forward checking

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



# Constraint propagation

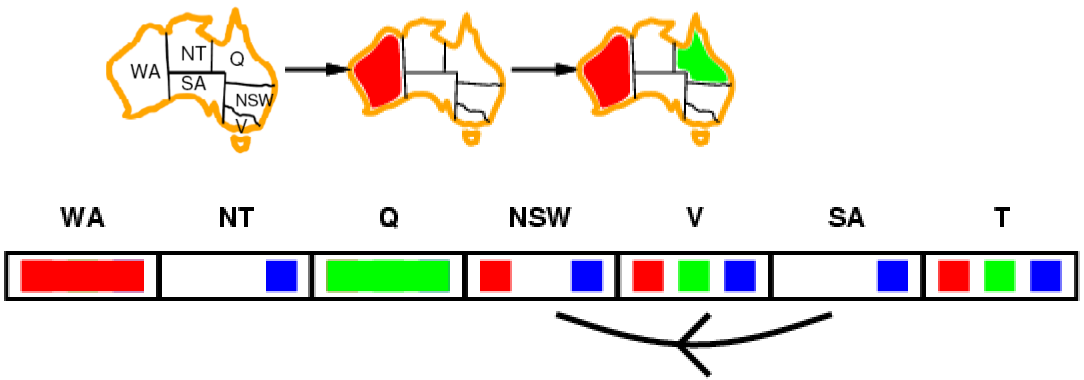
- Forward checking propagates information from assigned to unassigned variables, but doesn't provide early detection for all failures




- NT and SA cannot both be blue!
- **Constraint propagation** repeatedly enforces constraints *locally*

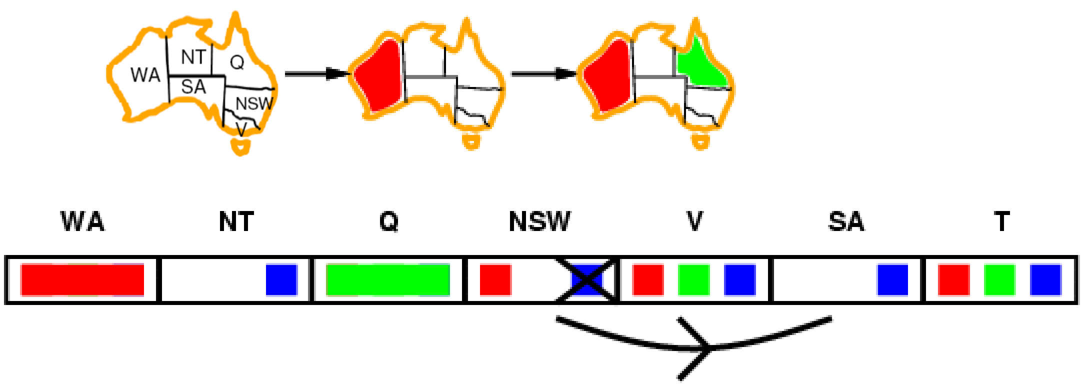
# Arc consistency

- Simplest form of propagation makes each pair of variables **consistent:**
  - $X \rightarrow Y$ is consistent iff for every value of $X$ there is some allowed value of $Y$



# Arc consistency

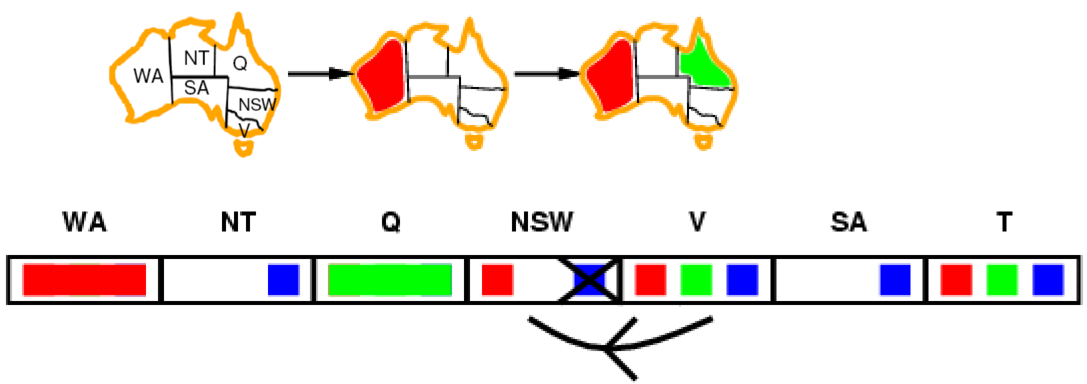
- Simplest form of propagation makes each pair of variables **consistent:**
  - *X* → *Y* is consistent iff for every value of *X* there is some allowed value of *Y*
  - When checking *X* → *Y*, throw out any values of X for which there isn't an allowed value of Y




- If *X* loses a value, all pairs *Z* → *X* need to be rechecked

# Arc consistency

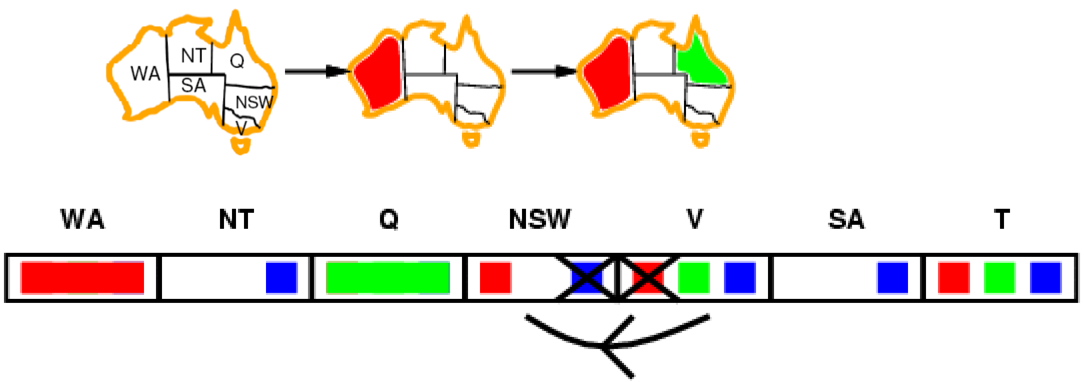
- Simplest form of propagation makes each pair of variables **consistent:**
  - *X* → *Y* is consistent iff for every value of *X* there is some allowed value of *Y*
  - When checking *X* → *Y*, throw out any values of X for which there isn't an allowed value of Y




- If *X* loses a value, all pairs *Z* → *X* need to be rechecked

# Arc consistency

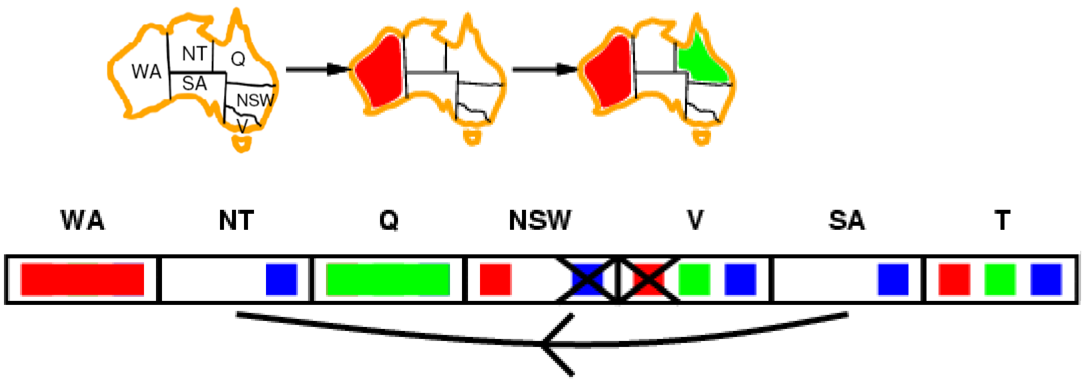
- Simplest form of propagation makes each pair of variables **consistent:**
  - *X* → *Y* is consistent iff for every value of *X* there is some allowed value of *Y*
  - When checking *X* → *Y*, throw out any values of X for which there isn't an allowed value of Y




- If *X* loses a value, all pairs *Z* → *X* need to be rechecked

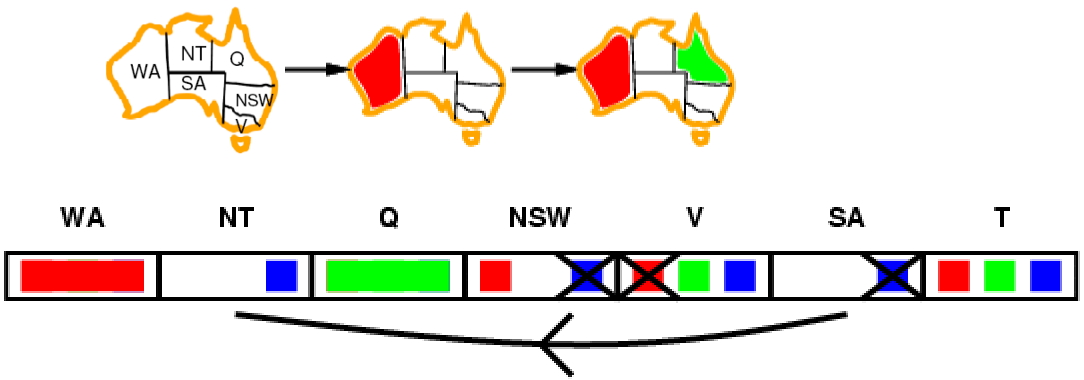
# Arc consistency

- Simplest form of propagation makes each pair of variables **consistent:**
  - $X \rightarrow Y$ is consistent iff for every value of $X$ there is some allowed value of $Y$
  - When checking $X \rightarrow Y$, throw out any values of X for which there isn't an allowed value of Y



# Arc consistency

- Simplest form of propagation makes each pair of variables **consistent:**
  - *X* → *Y* is consistent iff for every value of *X* there is some allowed value of *Y*
  - When checking *X* → *Y*, throw out any values of X for which there isn't an allowed value of Y




- Arc consistency detects failure earlier than forward checking
- Can be run before or after each assignment

# Arc consistency algorithm AC-3

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

  **while** *queue* is not empty
    $(X_i,\ X_j) \leftarrow$ 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* $\leftarrow$ *false*
  **for each** $x$ **in** Domain[$X_i$]
    **if** no value $y$ in Domain[$X_j$] allows $(x,y)$ to satisfy the constraint $X_i \leftrightarrow X_j$
      **then** delete $x$ from Domain[$X_i$]; *removed* $\leftarrow$ *true*
  **return** *removed*

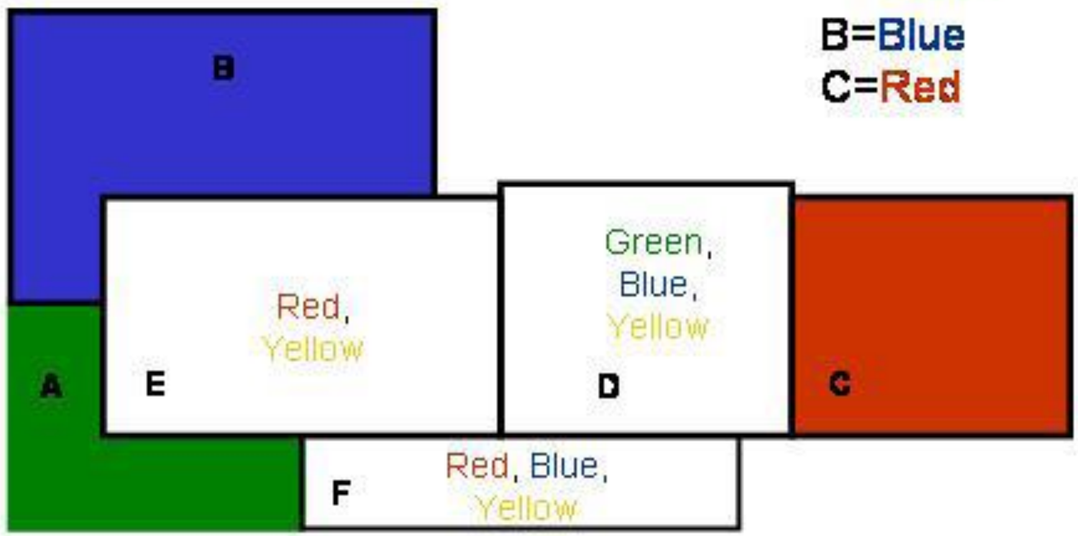
# Variable and Value Heuristics

- So far we have selected the next variable and the next value by using a fixed order

1. Is there a better way to pick the next variable?
2. Is there a better way to select the next value to assign to the current variable?

Colors: R, G, B, Y

A=Green
B=Blue
C=Red

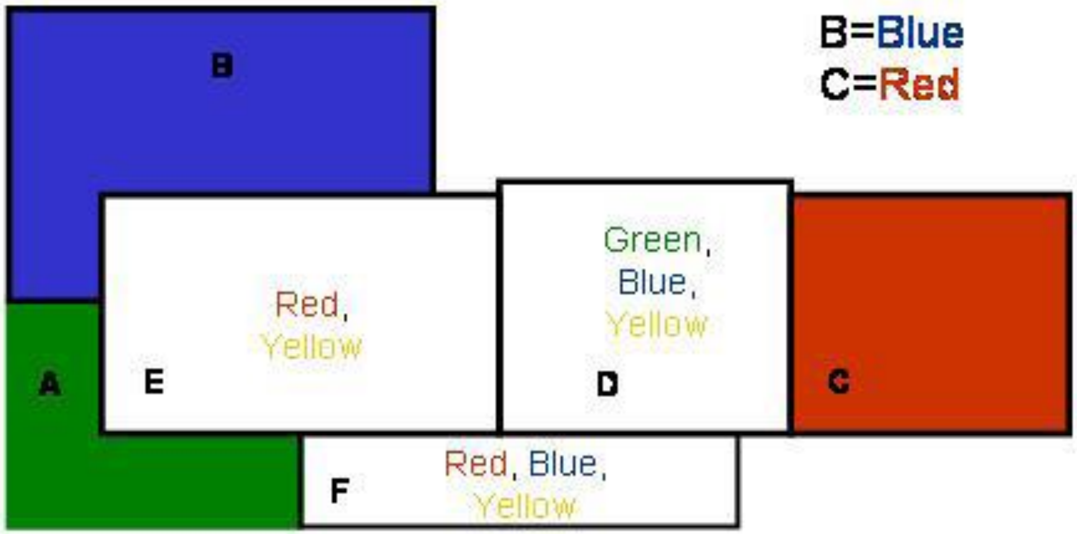



Which country should we color next →

What color should we pick for it? →

Colors: R, G, B, Y

A=Green
B=Blue
C=Red




Which country should we color next → E most-constrained variable (smallest domain)

What color should we pick for it? → RED least-constraining value (eliminates fewest values from neighboring domains)

# BT-FC with dynamic ordering

Traditional backtracking uses fixed ordering of variables & values, e.g., random order or place variables with many constraints first.

You can usually do better by choosing an order dynamically as the search proceeds.

- **Most constrained variable**

  when doing forward-checking, pick variable with fewest legal values to assign next (minimizes branching factor)

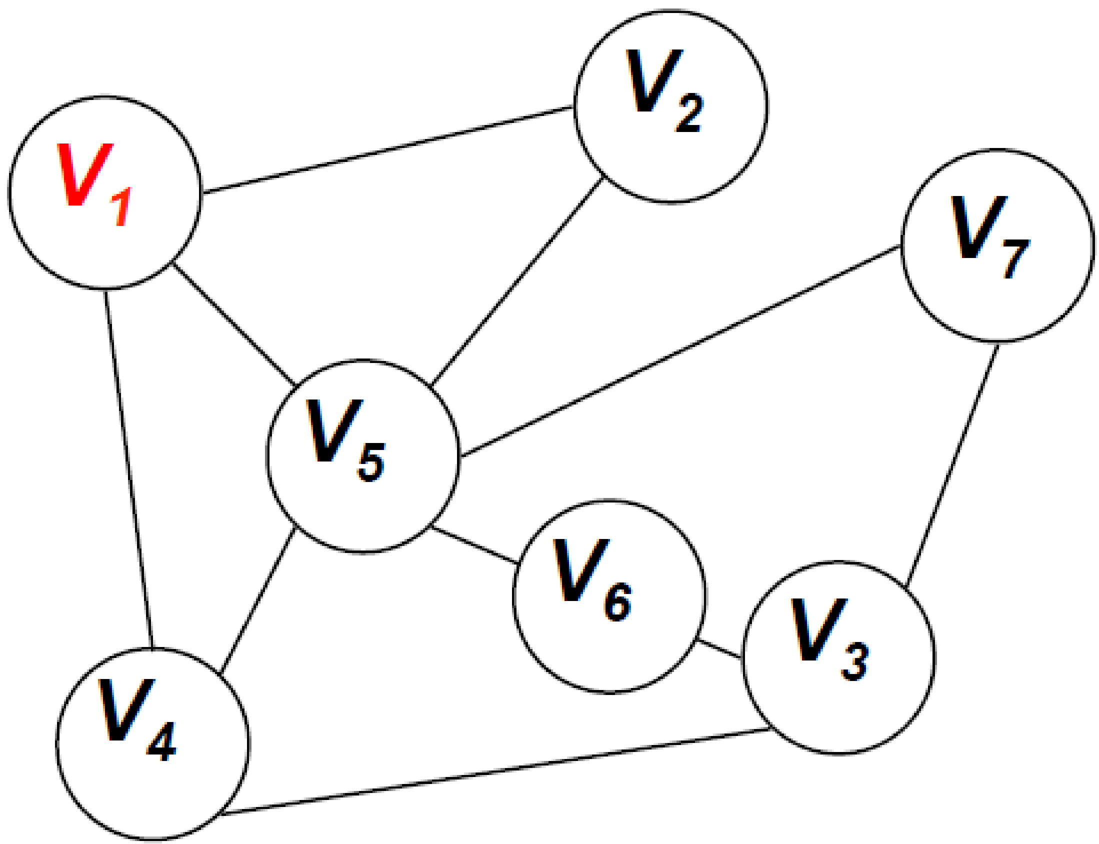
- **Least constraining value**

  choose value that rules out the fewest values from neighboring domains

E.g. this combination improves feasible n-queens performance from about n = 30 with just FC to about n = 1000 with FC & ordering.

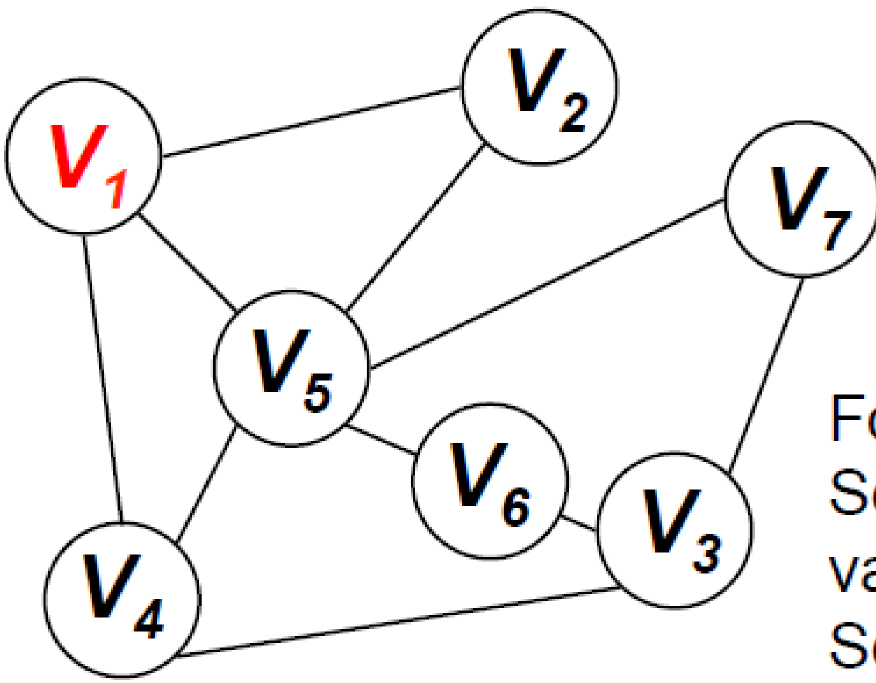
# CSP Heuristics: Variable Ordering I

| $V_1$ | $V_2$ | $V_3$ | $V_4$ | $V_5$ | $V_6$ | $V_7$ |
|---|---|---|---|---|---|---|
| R | ? | ? | ? | ? | ? | ? |



# CSP Heuristics: Variable Ordering I

- *Most Constraining Variable*
- Selecting a variable which contributes to the *largest* number of constraints will have the largest effect on the other variables → ***Hopefully will prune a larger part of the search***
- Equivalent to finding the variable that is connected to the largest number of variables in the constraint graph.

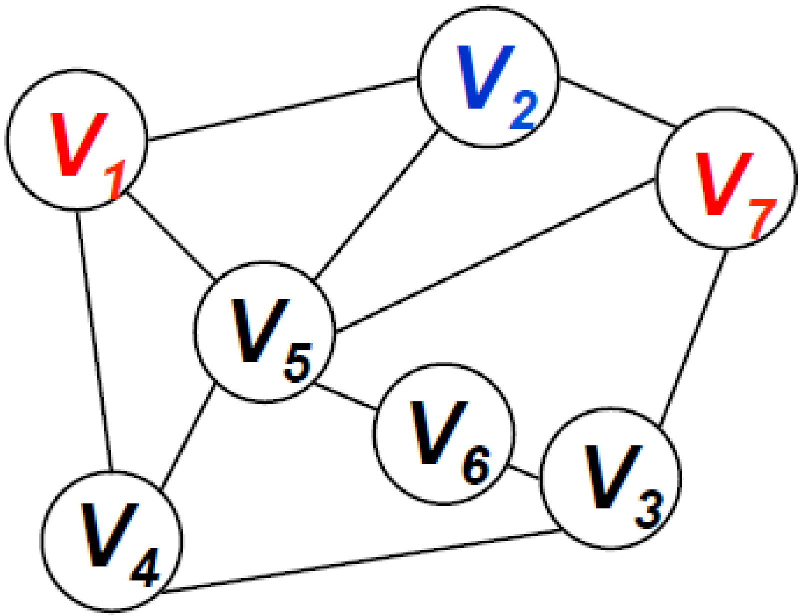



| $V_1$ | $V_2$ | $V_3$ | $V_4$ | $V_5$ | $V_6$ | $V_7$ |
|---|---|---|---|---|---|---|
| R | ? | ? | ? | ? | ? | ? |

For this state:
Setting variable $V_5$ affects *4* variables;
Setting any other variable affects fewer than 4 variables

# CSP Heuristics: Variable Ordering II

- *Minimum Remaining Values (MRV)*
- Selecting the variable that has the least number of candidate values is most likely to cause a failure early ("fail-first" heuristic)

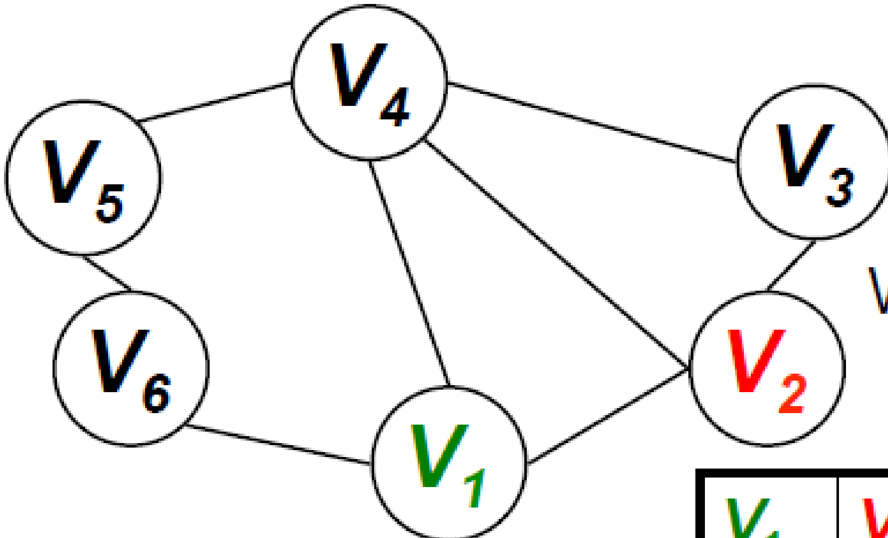



| | $V_1$ | $V_2$ | $V_3$ | $V_4$ | $V_5$ | $V_6$ | $V_7$ |
|---|---|---|---|---|---|---|---|
| R | O | | X | X | X | ? | O |
| B | | O | ? | ? | X | ? | |
| G | | | ? | ? | ? | ? | |

$V_5$ is the most constrained variable and is the most likely to prune the search tree

# CSP Heuristics: Value Ordering

- *Least Constraining Value*
- Choose the value which causes the smallest reduction in the number of available values for the neighboring variables




Four colors: $D = \{\boldsymbol{R}, \boldsymbol{G}, \boldsymbol{B}, \boldsymbol{Y}\}$

Which value to try next for $V_3$?

| $V_1$ | $V_2$ | $V_3$ | $V_4$ | $V_5$ | $V_6$ | $V_7$ |
|---|---|---|---|---|---|---|
| ***G*** | ***R*** | ? | ? | ? | ? | ? |

# Which variable should be assigned next?

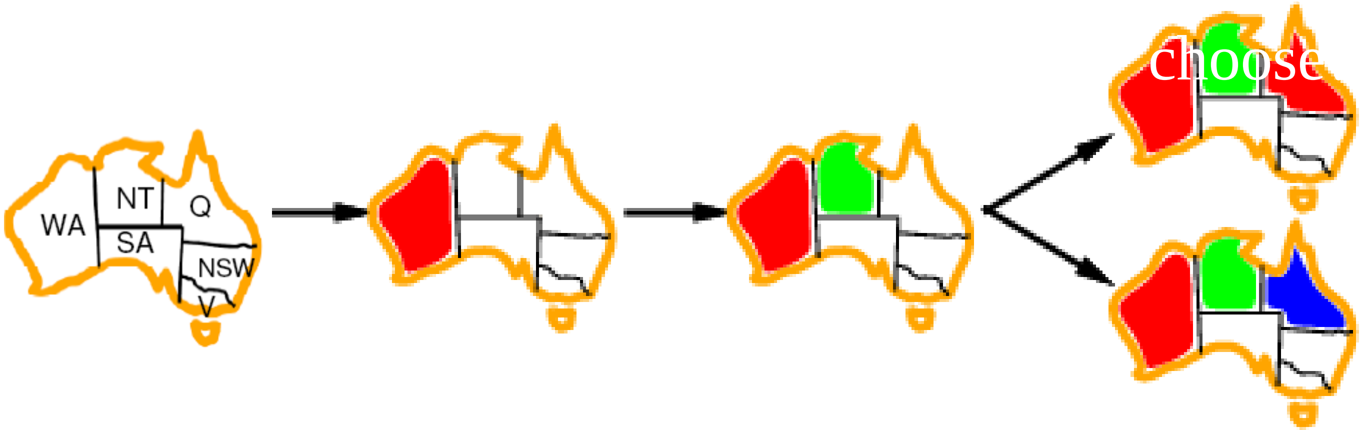
- **Most constrained variable:**
  - Choose the variable with the fewest legal values
  - A.k.a. **minimum remaining values** (MRV) heuristic

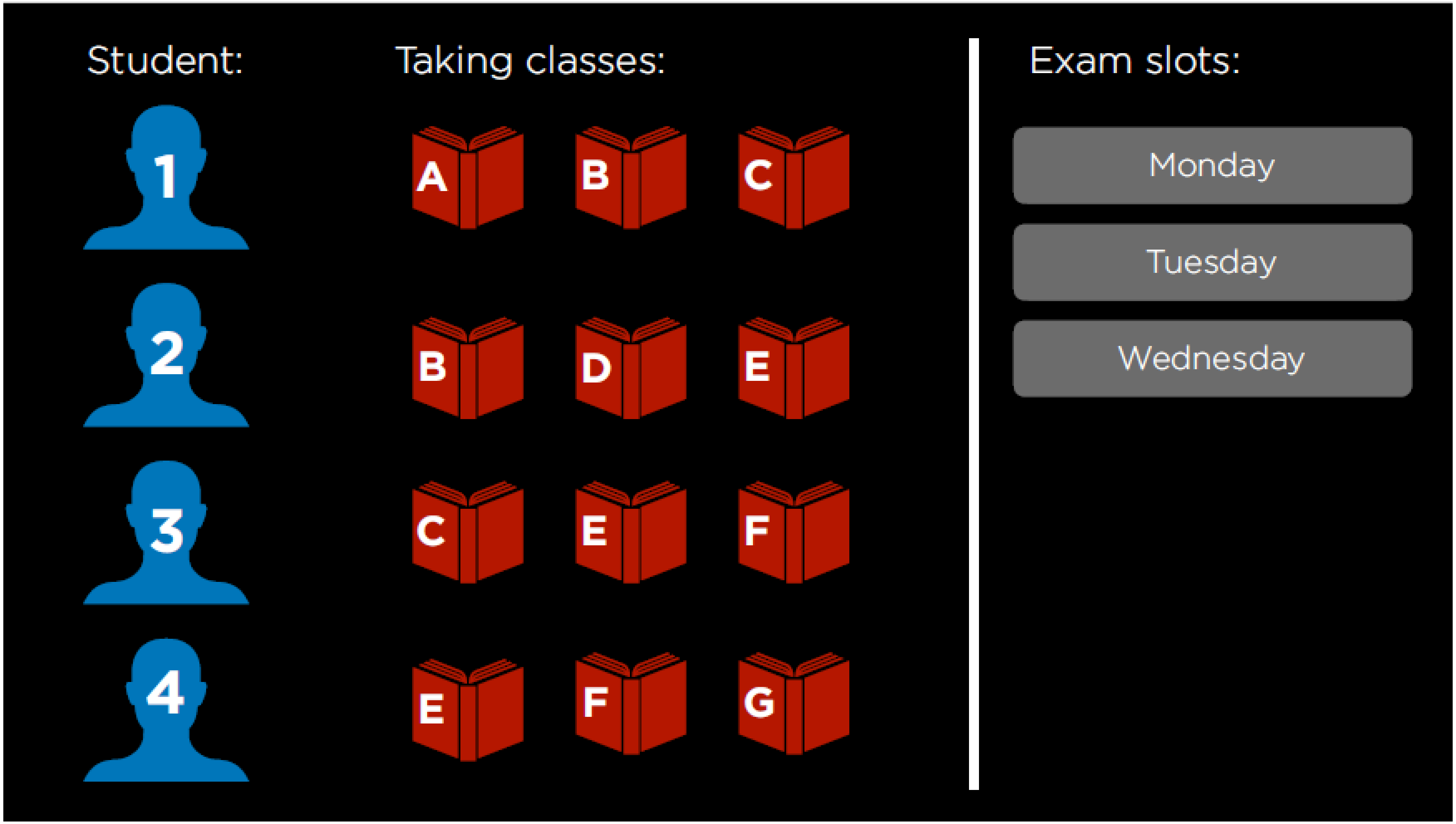
# Which variable should be assigned next?

- **Most constraining variable:**
  - Choose the variable that imposes the most constraints on the remaining variables
  - Tie-breaker among most constrained variables

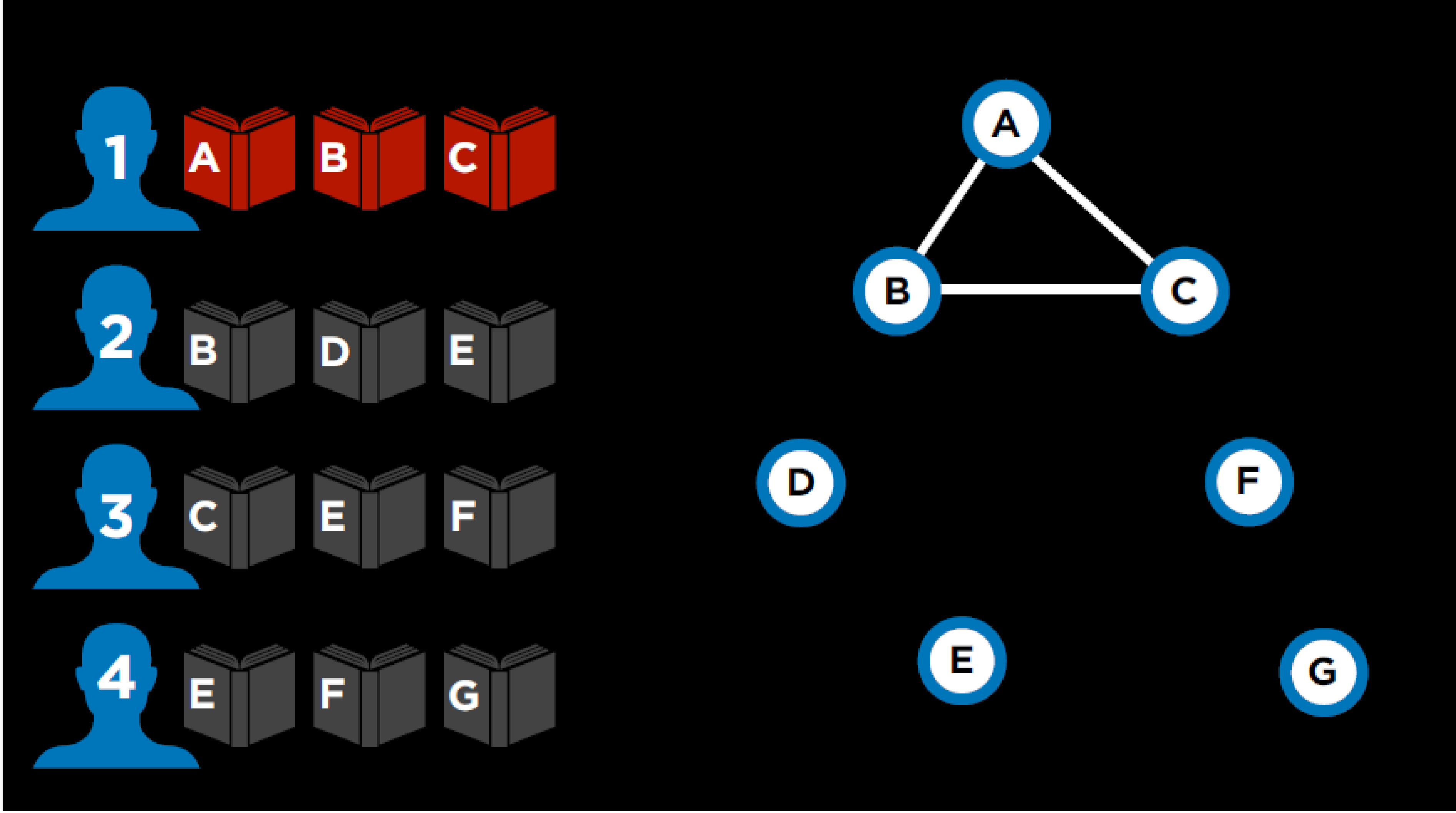
# Given a variable, what should be the order of values?

- Choose the **least constraining value**:
  - The value that rules out the fewest values in the remaining variables

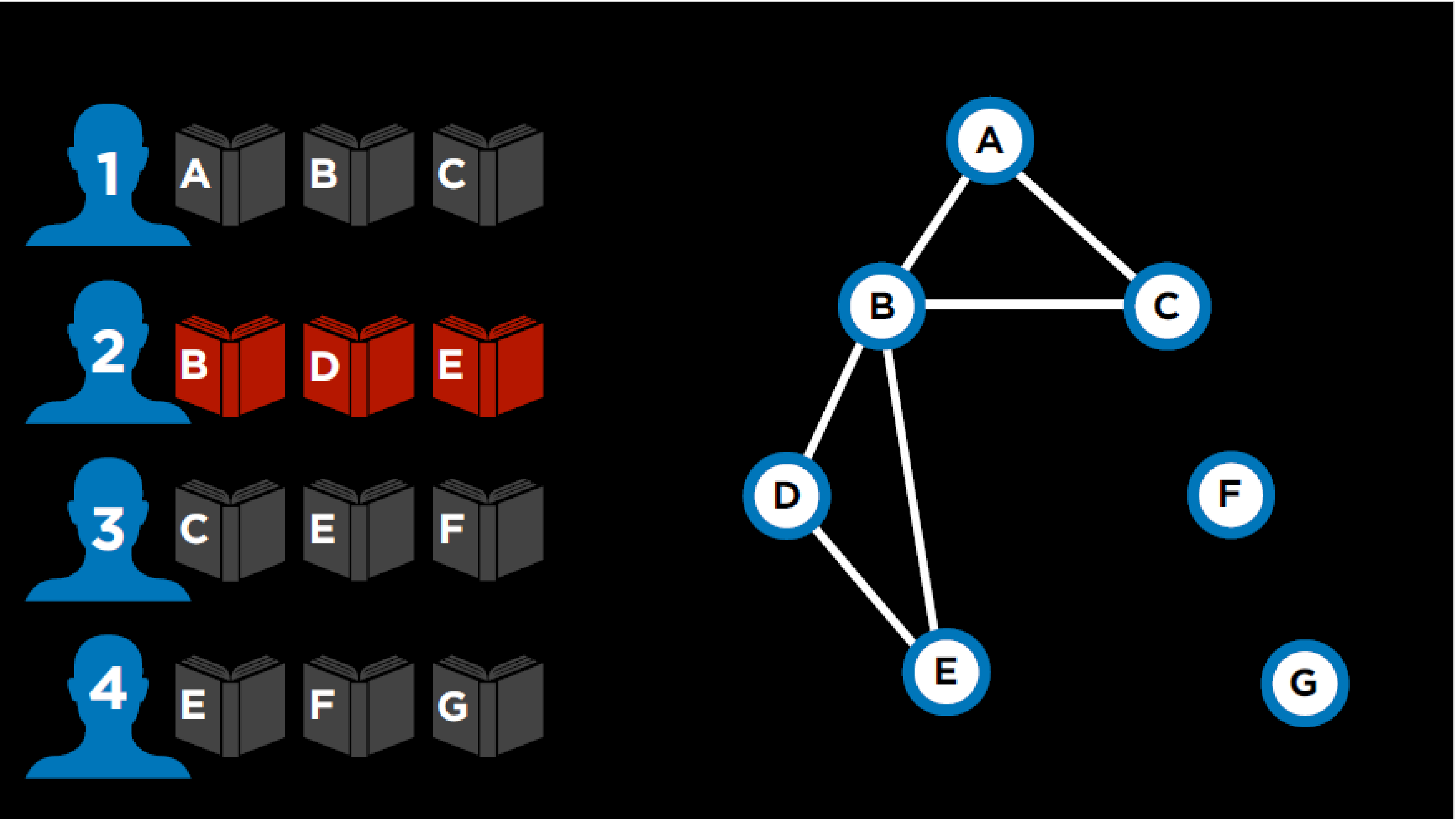




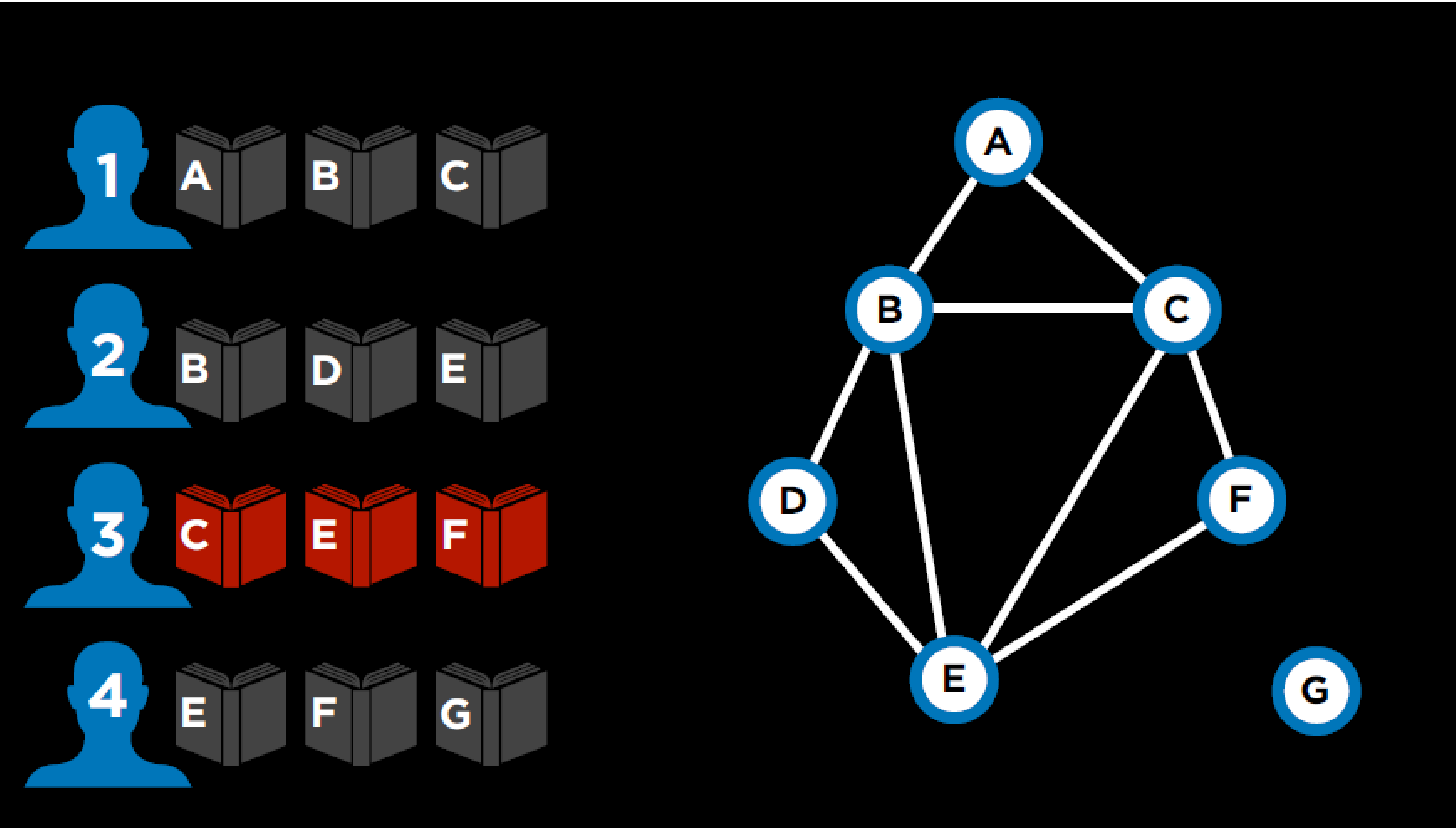

Slide credit : HarvardX CS50AICS50's Introduction to Artificial Intelligence with Python, David J. Malan and Brian Yu



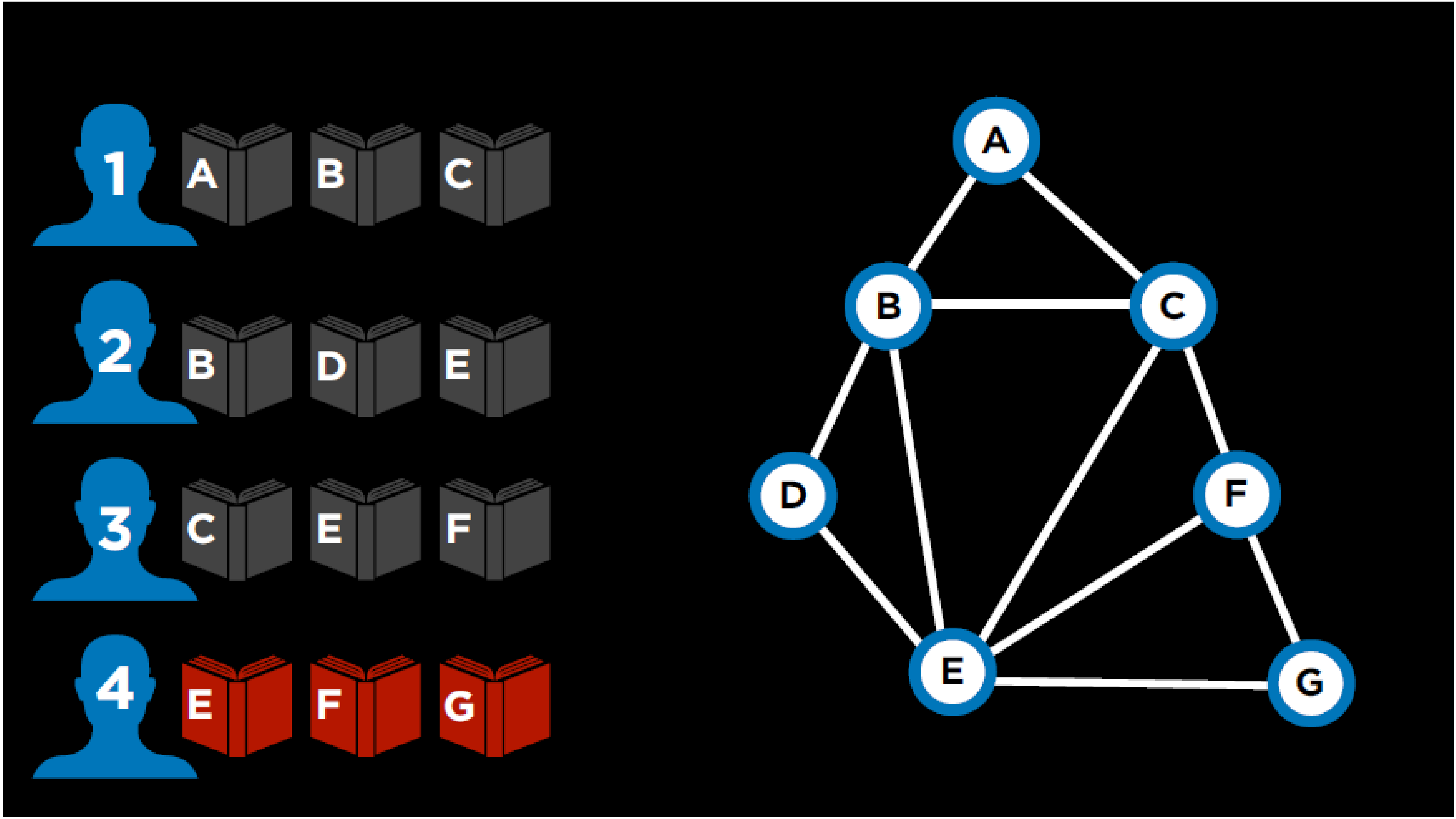

Slide credit : HarvardX CS50AICS50's Introduction to Artificial Intelligence with Python, David J. Malan and Brian Yu



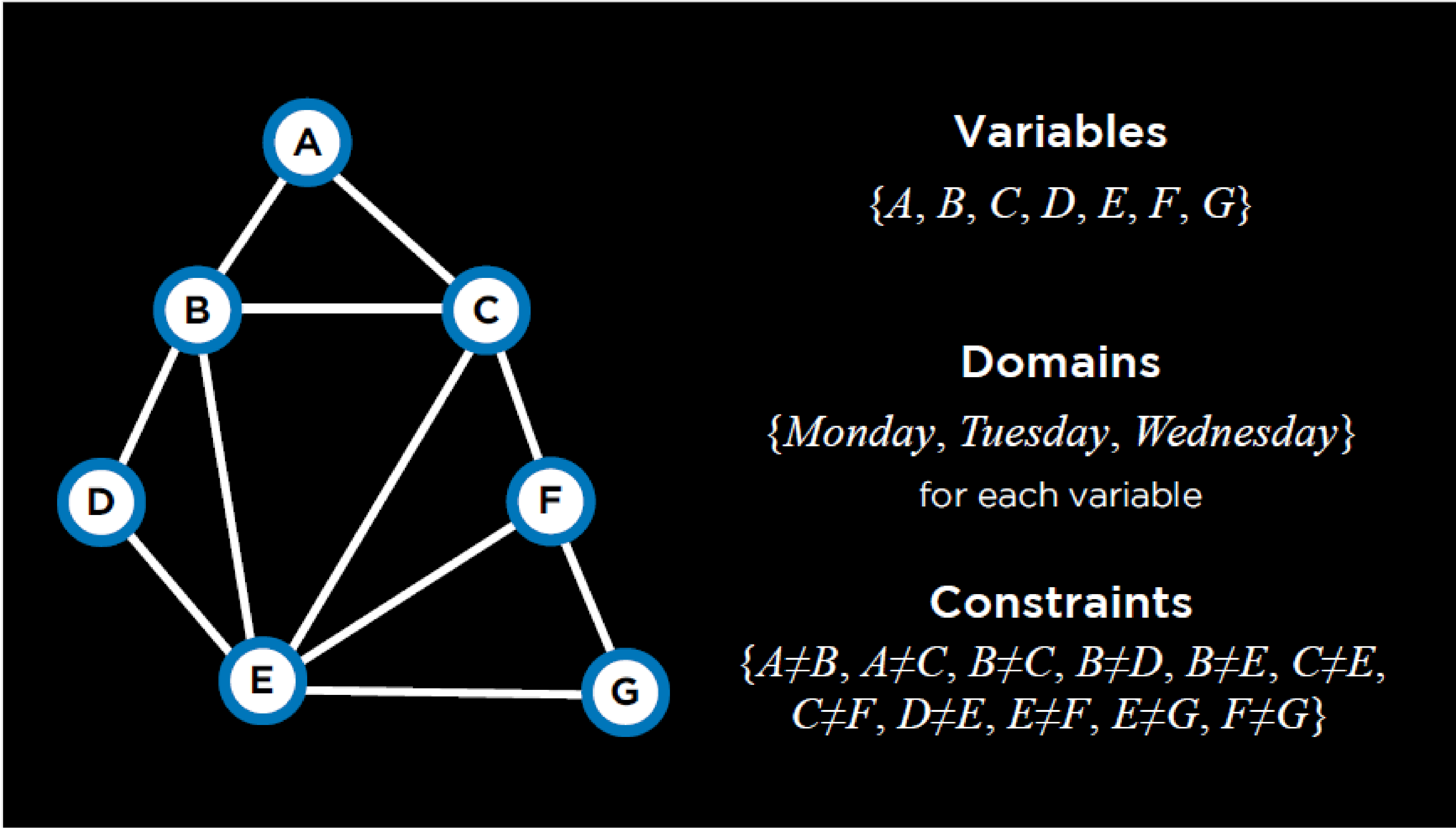

Slide credit : HarvardX CS50AICS50's Introduction to Artificial Intelligence with Python, David J. Malan and Brian Yu




Slide credit : HarvardX CS50AICS50's Introduction to Artificial Intelligence with Python, David J. Malan and Brian Yu



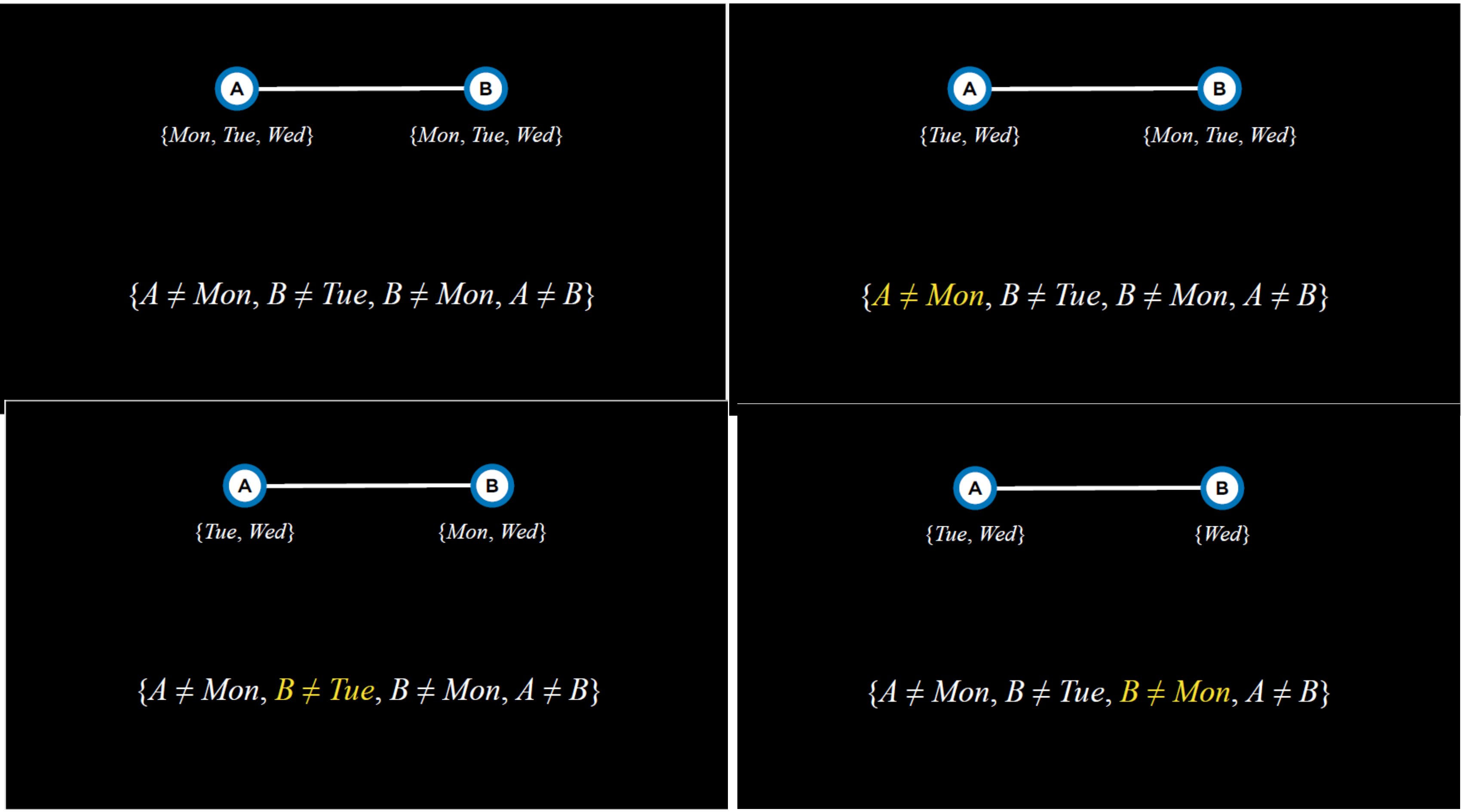

Slide credit : HarvardX CS50AICS50's Introduction to Artificial Intelligence with Python, David J. Malan and Brian Yu




Slide credit : HarvardX CS50AICS50's Introduction to Artificial Intelligence with Python, David J. Malan and Brian Yu

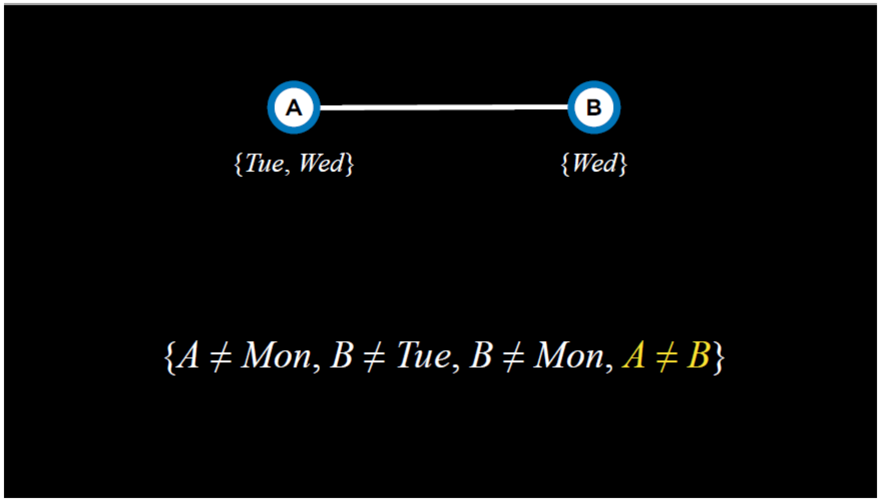
- unary constraint constraint involving only one variable
- {*A* ≠ *Monday*}
- binary constraint constraint involving two variables
- {*A* ≠ *B*}

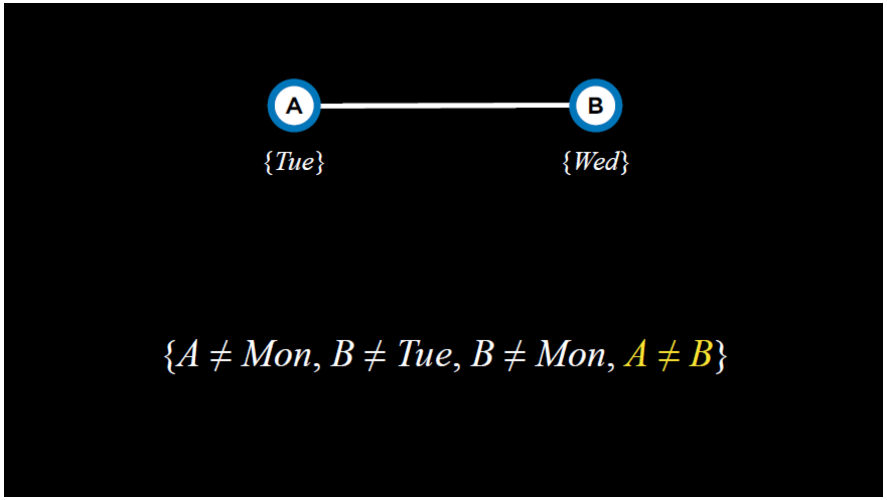
- node consistency when all the values in a variable's domain satisfy the variable's unary constraints




Slide credit : HarvardX CS50AICS50's Introduction to Artificial Intelligence with Python, David J. Malan and Brian Yu

- arc consistency when all the values in a variable's domain satisfy the variable's binary constraints
- To make *X* arc-consistent with respect to *Y*, remove elements from *X*'s domain until every choice for *X* has a possible choice for *Y*



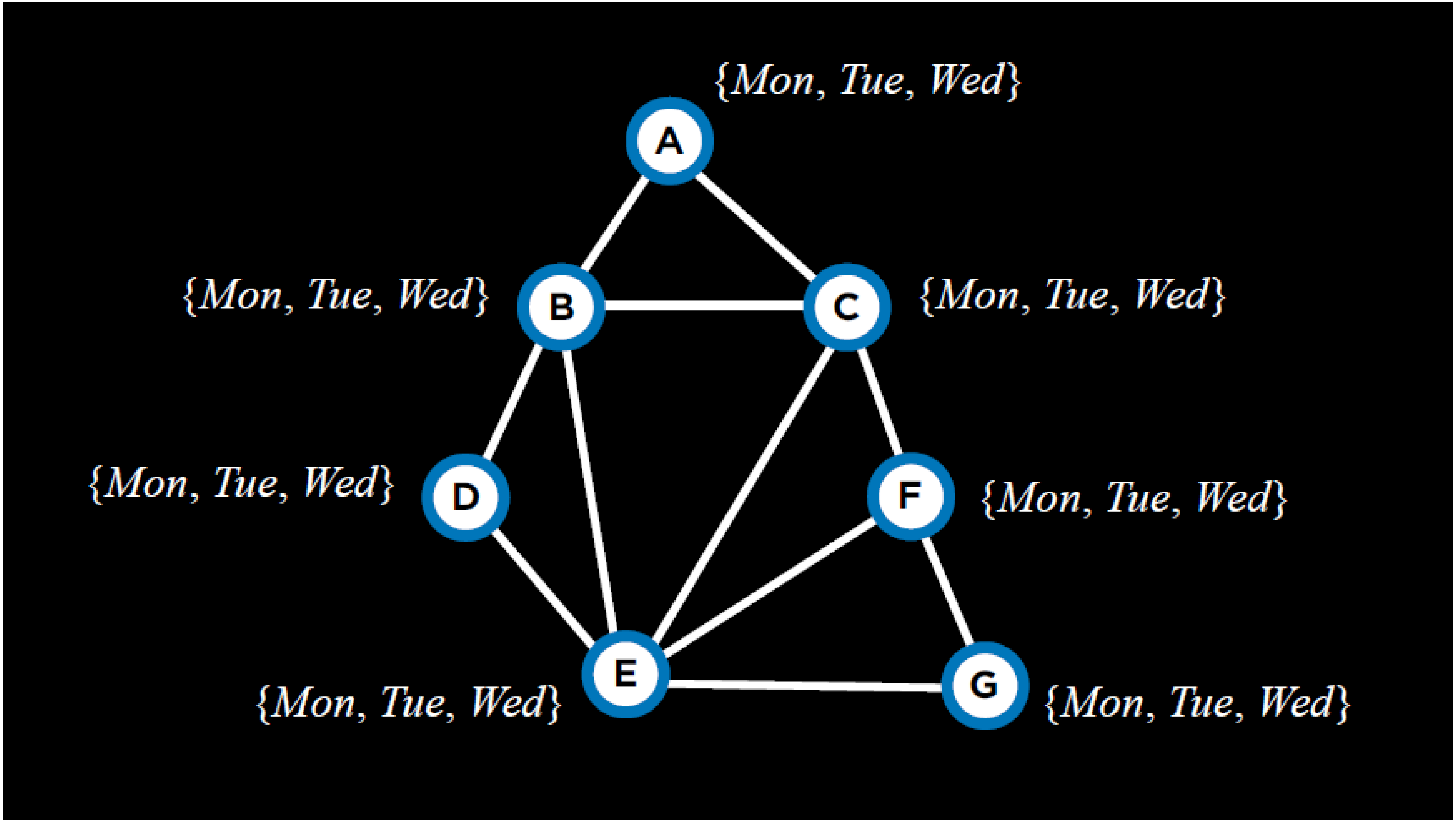




Slide credit : HarvardX CS50AICS50's Introduction to Artificial Intelligence with Python, David J. Malan and Brian Yu

# Arc Consistency

```
function AC-3(csp):
    queue = all arcs in csp
    while queue non-empty:
        (X, Y) = DEQUEUE(queue)
        if REVISE(csp, X, Y):
            if size of X.domain == 0:
                return false
            for each Z in X.neighbors - {Y}:
                ENQUEUE(queue, (Z, X))
    return true
```

Slide credit : HarvardX CS50AICS50's Introduction to Artificial Intelligence with Python, David J. Malan and Brian Yu




Slide credit : HarvardX CS50AICS50's Introduction to Artificial Intelligence with Python, David J. Malan and Brian Yu

# CSPs as Search Problems

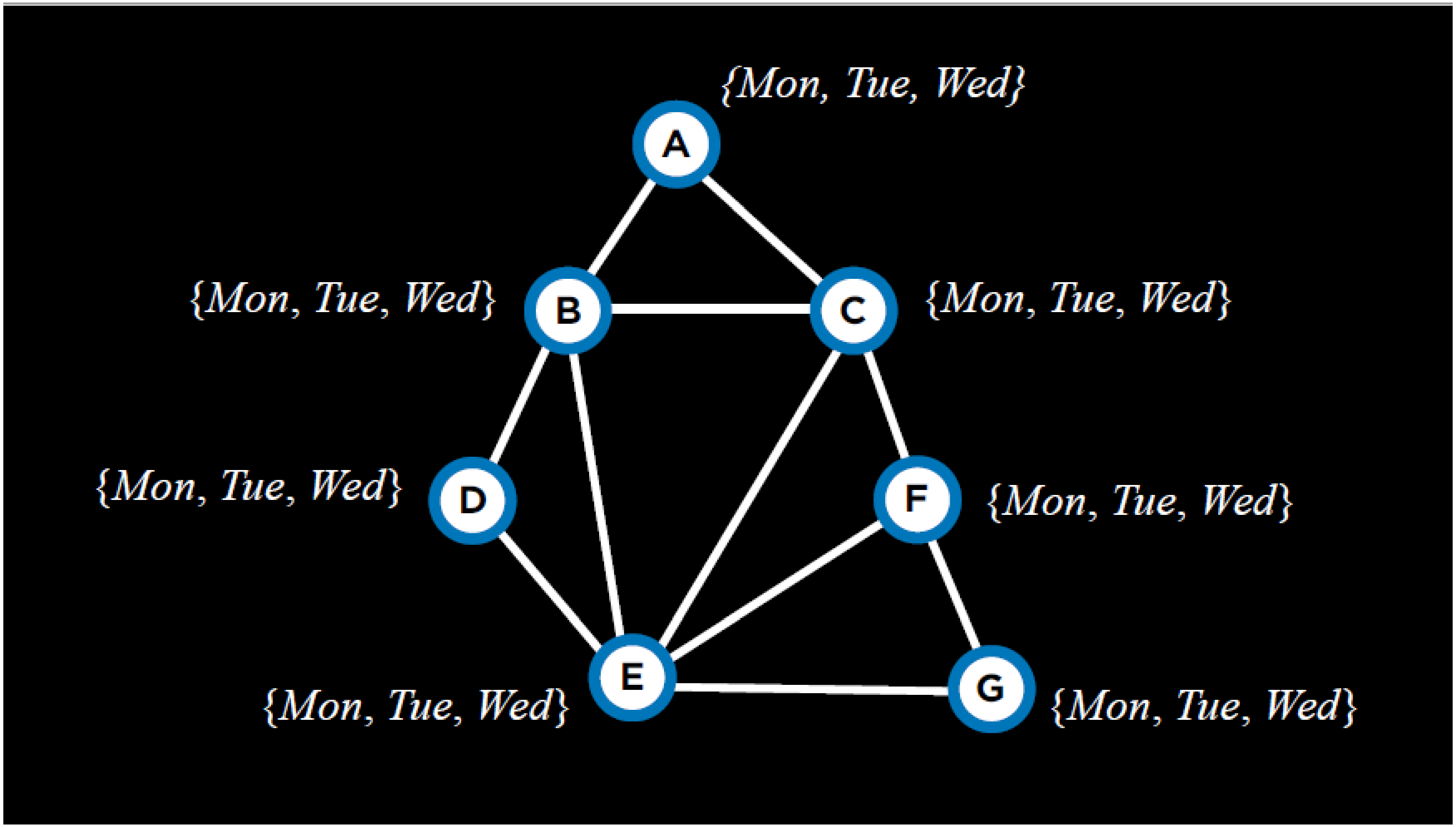
- initial state: empty assignment (no variables)
- actions: add a $\{variable = value\}$ to assignment
- transition model: shows how adding an assignment changes the assignment
- goal test: check if all variables assigned and constraints all satisfied
- path cost function: all paths have same cost

Slide credit : HarvardX CS50AICS50's Introduction to Artificial Intelligence with Python, David J. Malan and Brian Yu

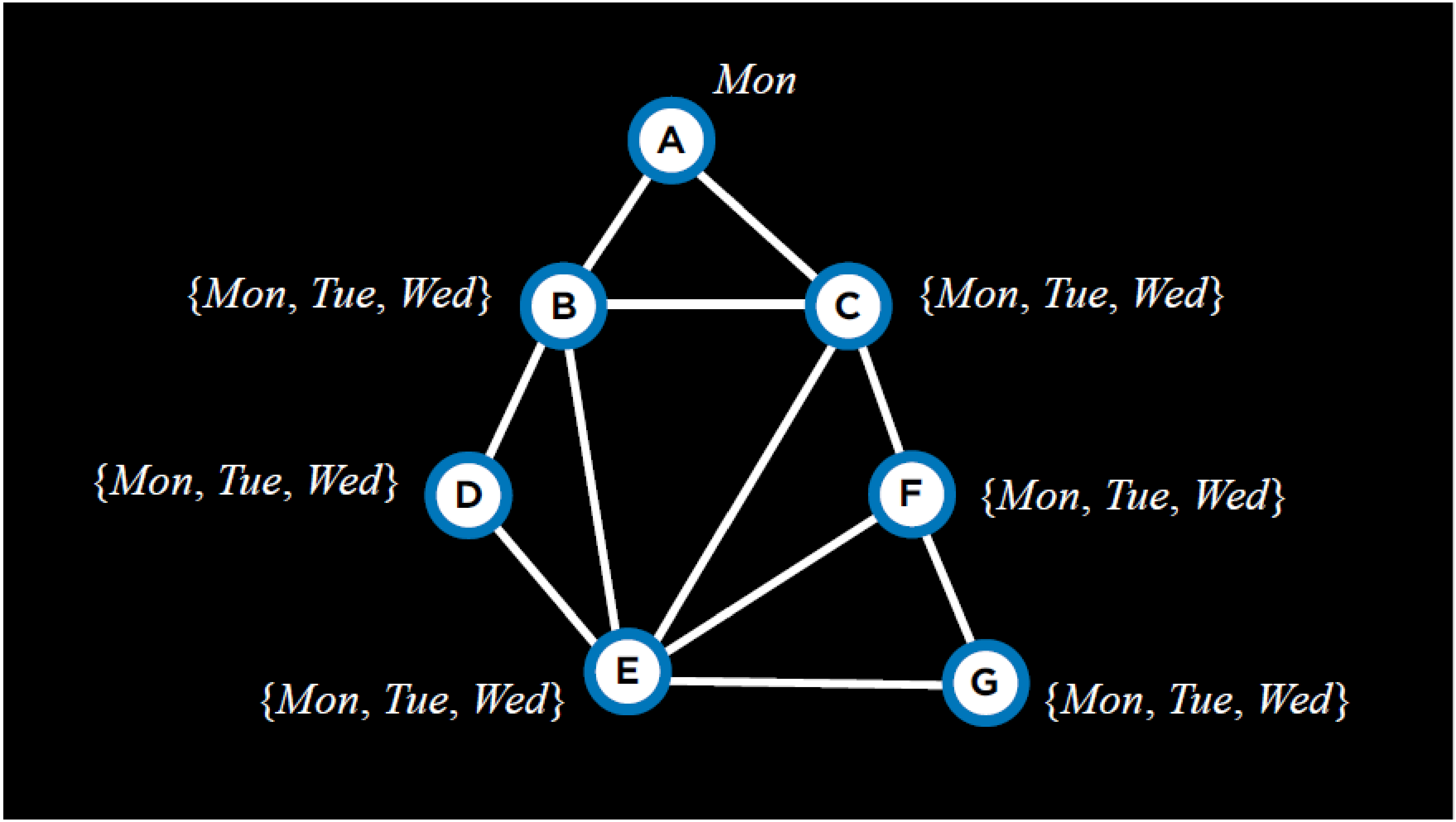
# Backtracking Search

```
function BACKTRACK(assignment, csp):
    if assignment complete: return assignment
    var = SELECT-UNASSIGNED-VAR(assignment, csp)
    for value in DOMAIN-VALUES(var, assignment, csp):
        if value consistent with assignment:
            add {var = value} to assignment
            result = BACKTRACK(assignment, csp)
            if result ≠ failure: return result
        remove {var = value} from assignment
    return failure
```

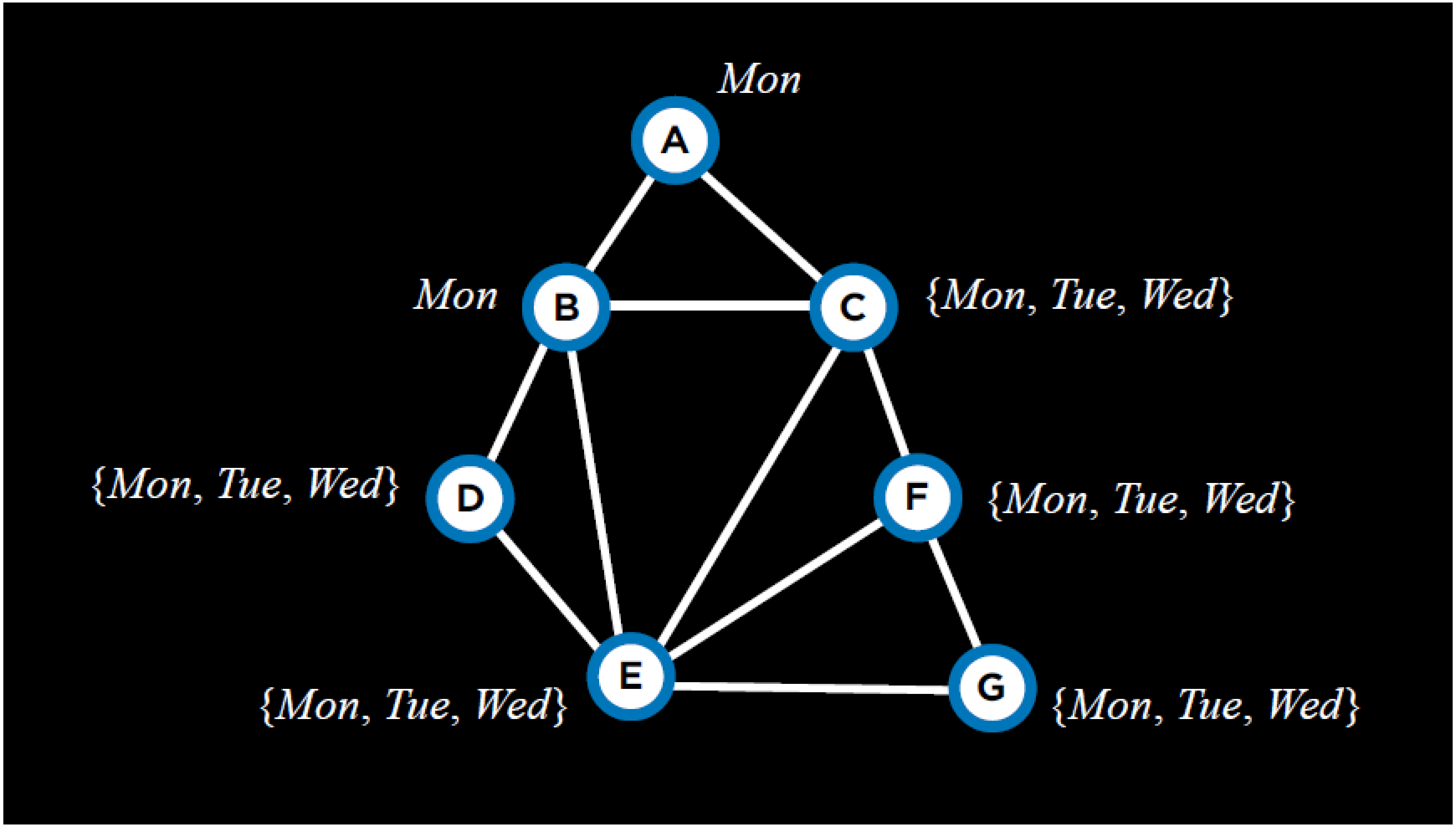
Slide credit : HarvardX CS50AICS50's Introduction to Artificial Intelligence with Python, David J. Malan and Brian Yu



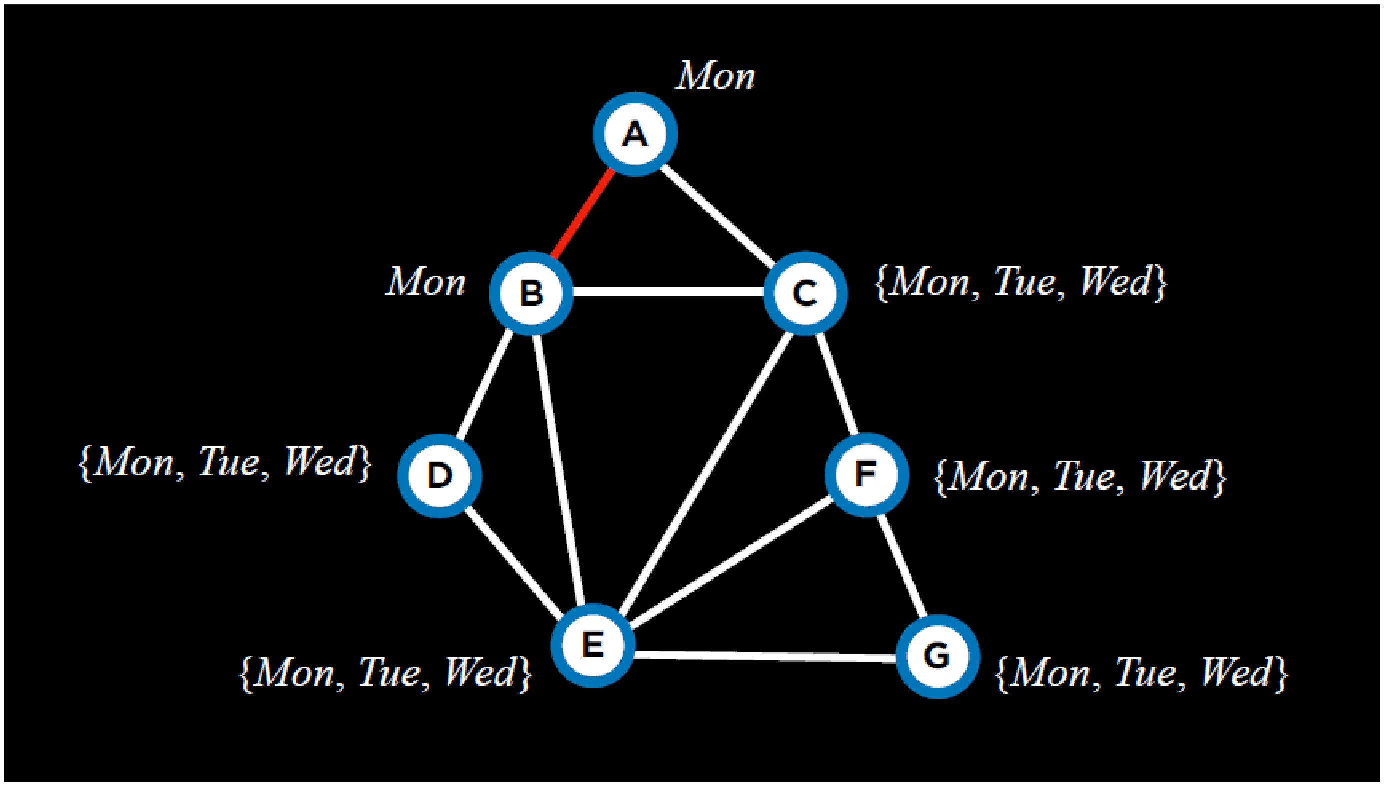

Slide credit : HarvardX CS50AICS50's Introduction to Artificial Intelligence with Python, David J. Malan and Brian Yu



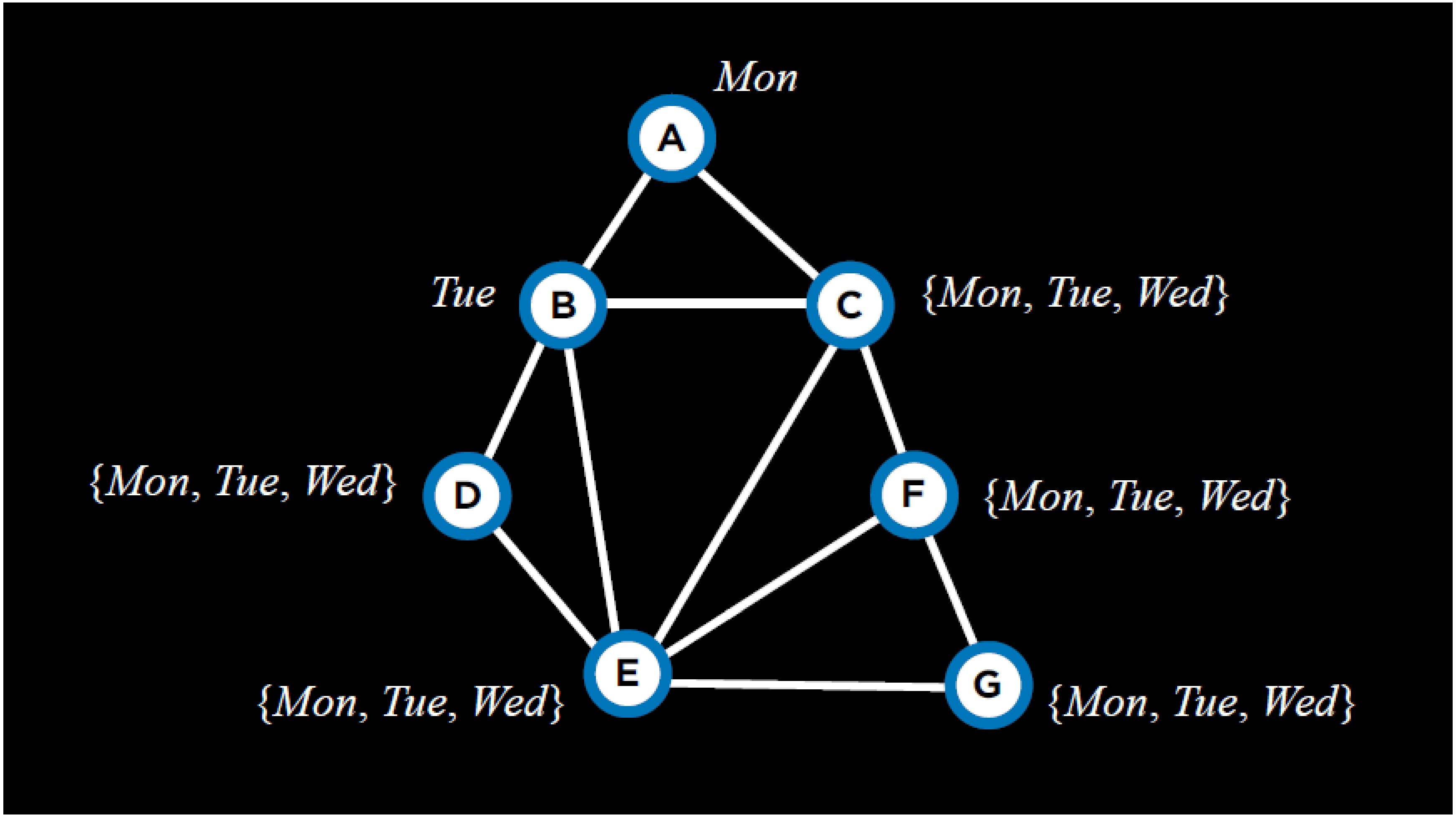

Slide credit : HarvardX CS50AICS50's Introduction to Artificial Intelligence with Python, David J. Malan and Brian Yu



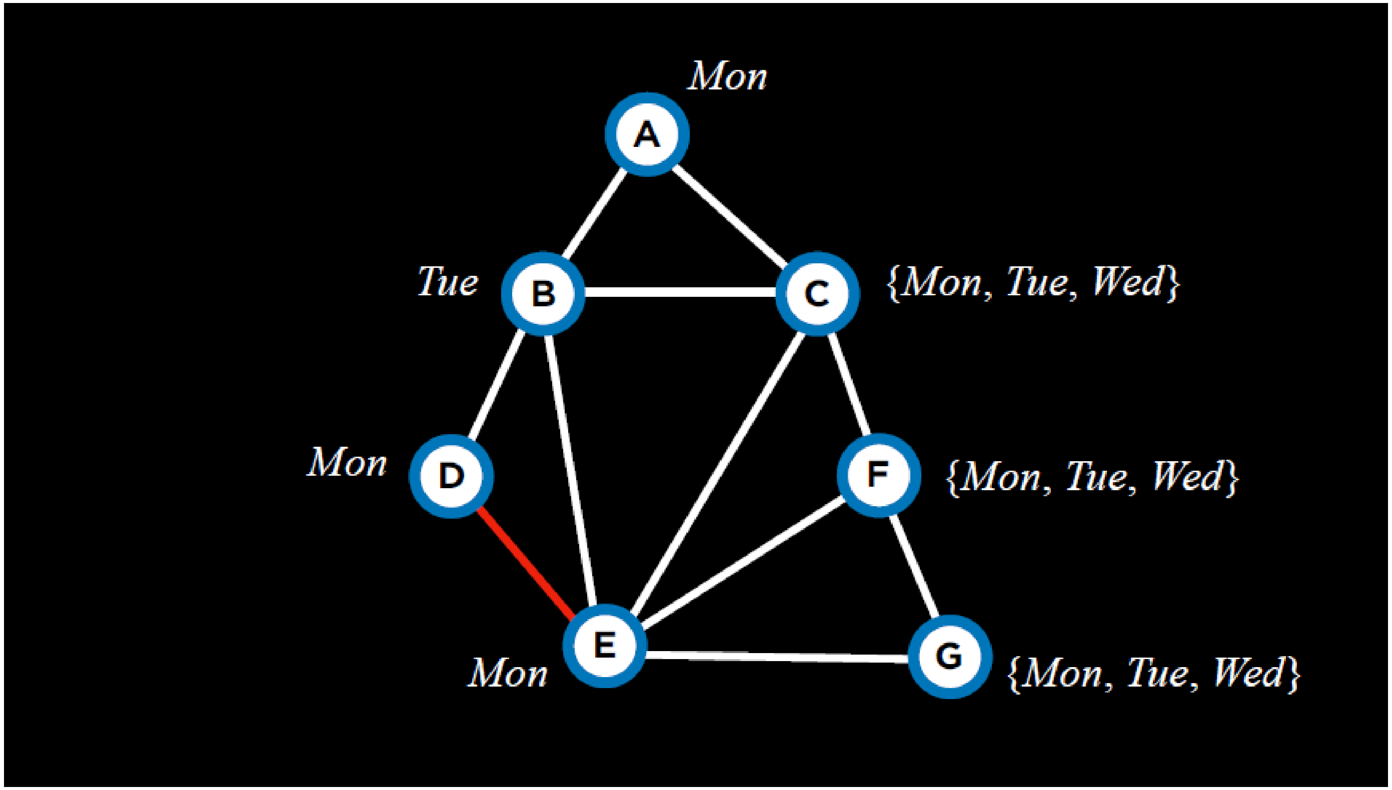

Slide credit : HarvardX CS50AICS50's Introduction to Artificial Intelligence with Python, David J. Malan and Brian Yu



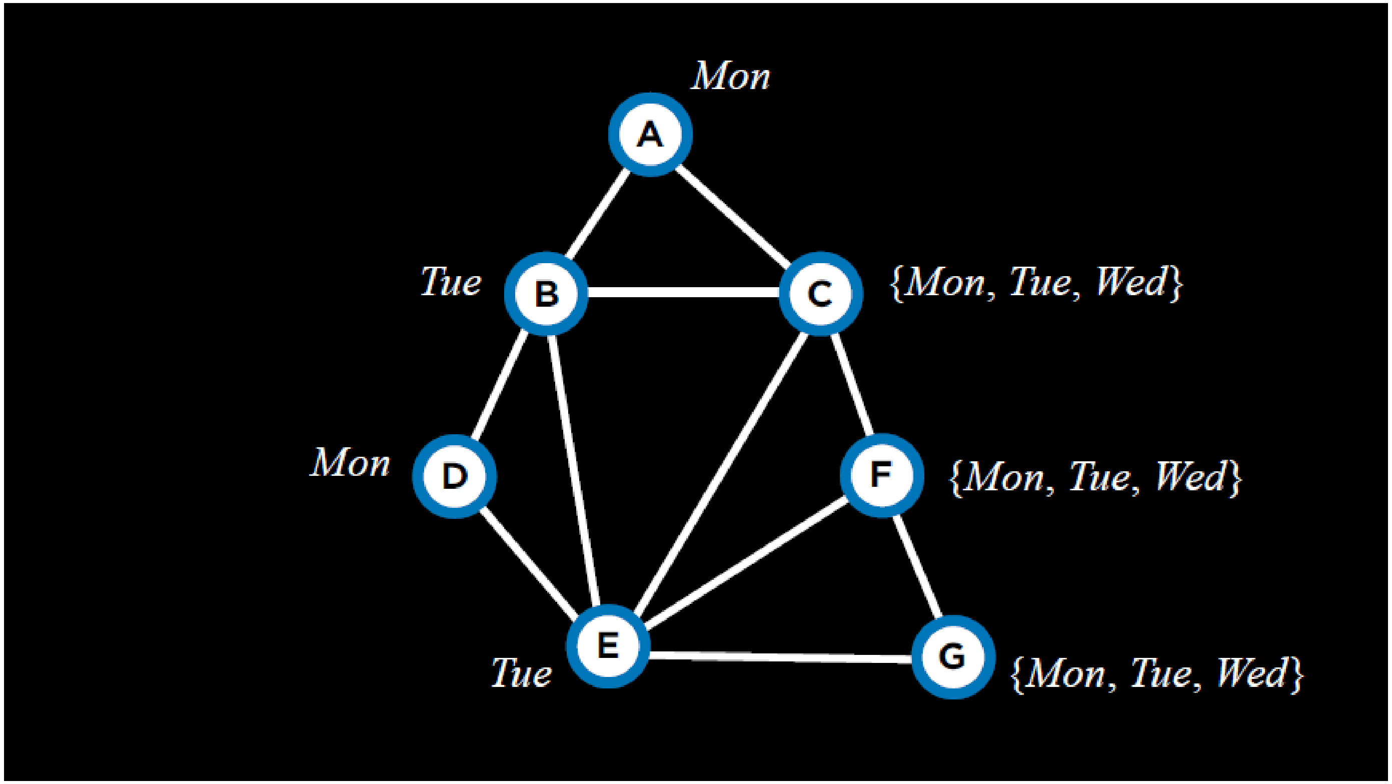

Slide credit : HarvardX CS50AICS50's Introduction to Artificial Intelligence with Python, David J. Malan and Brian Yu

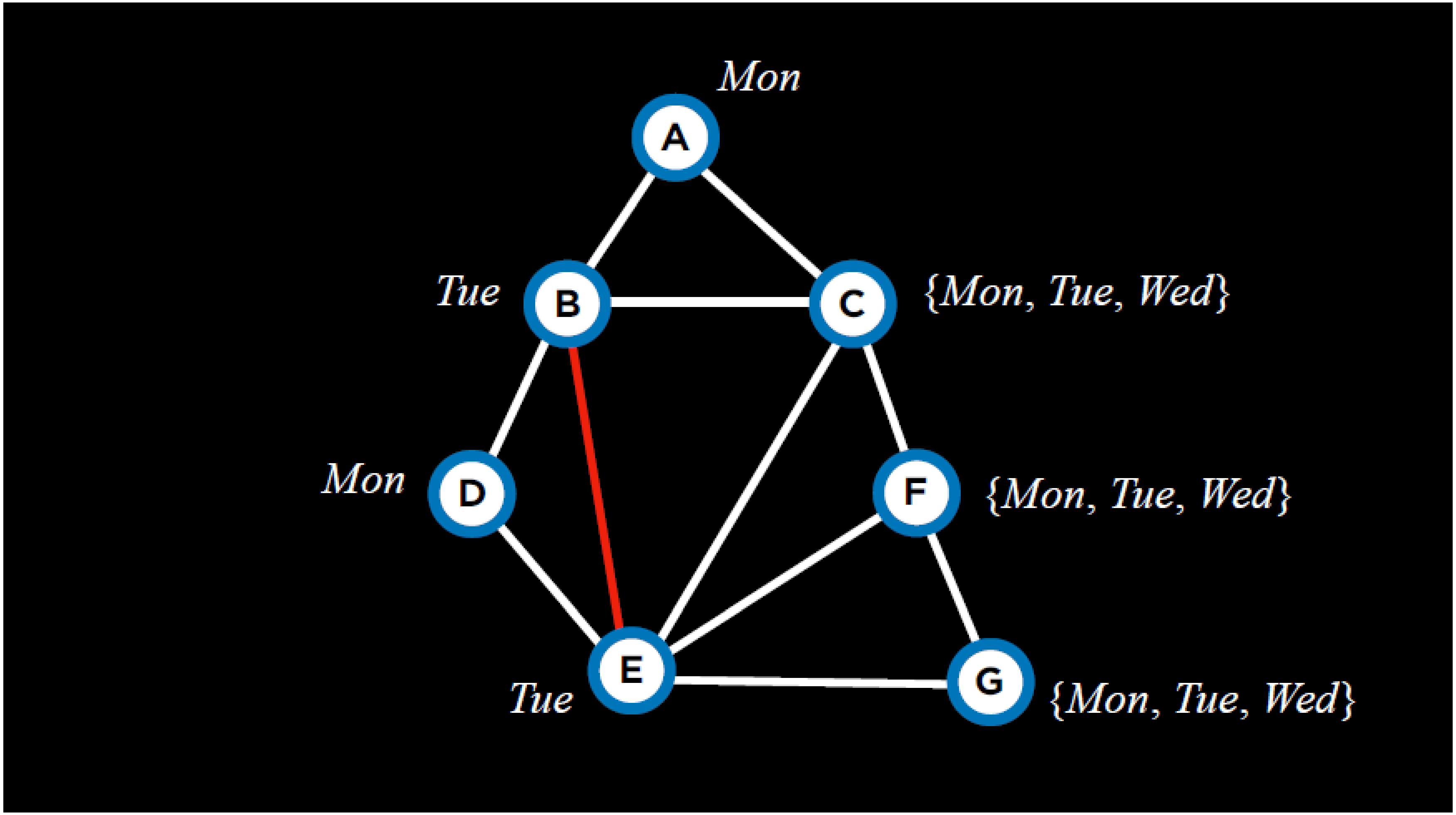



Slide credit : HarvardX CS50AICS50's Introduction to Artificial Intelligence with Python, David J. Malan and Brian Yu

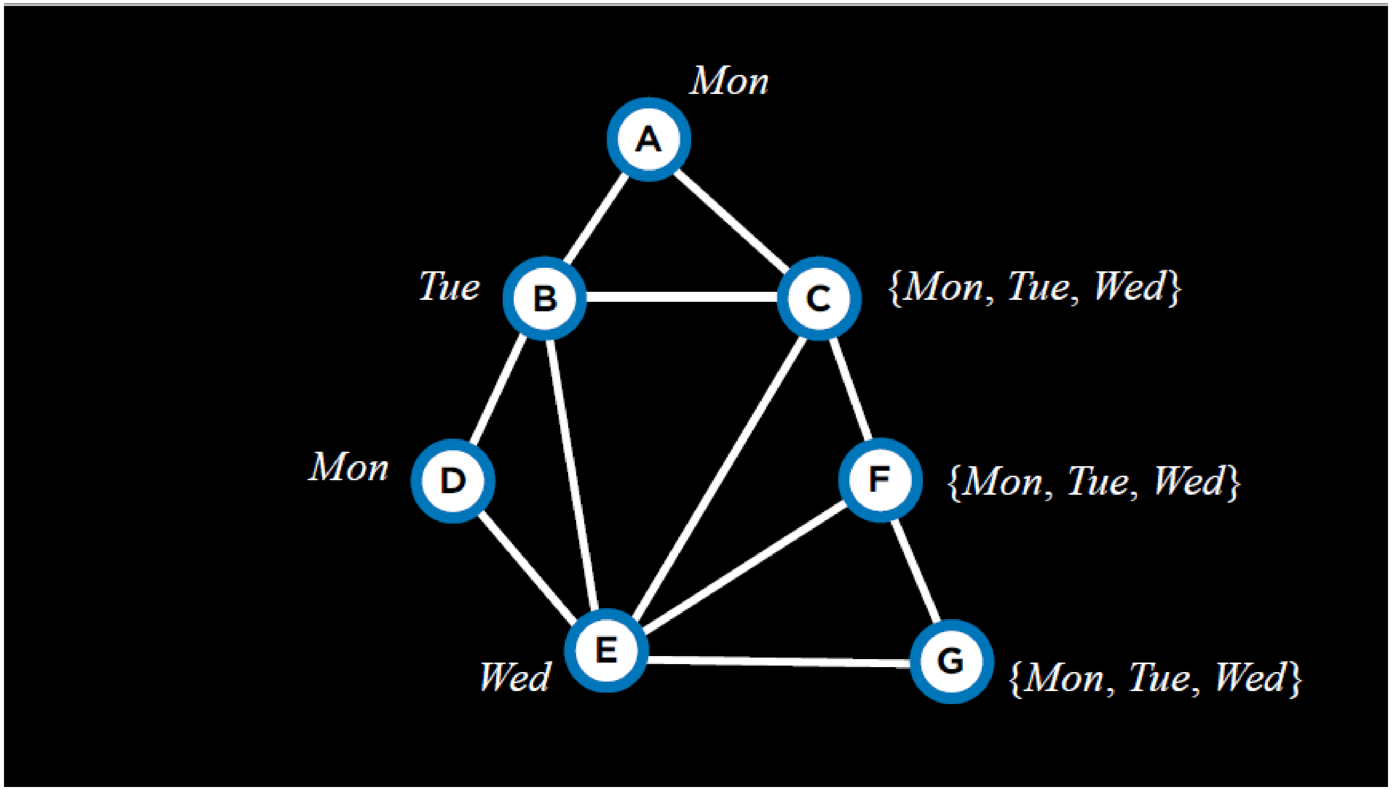



Slide credit : HarvardX CS50AICS50's Introduction to Artificial Intelligence with Python, David J. Malan and Brian Yu

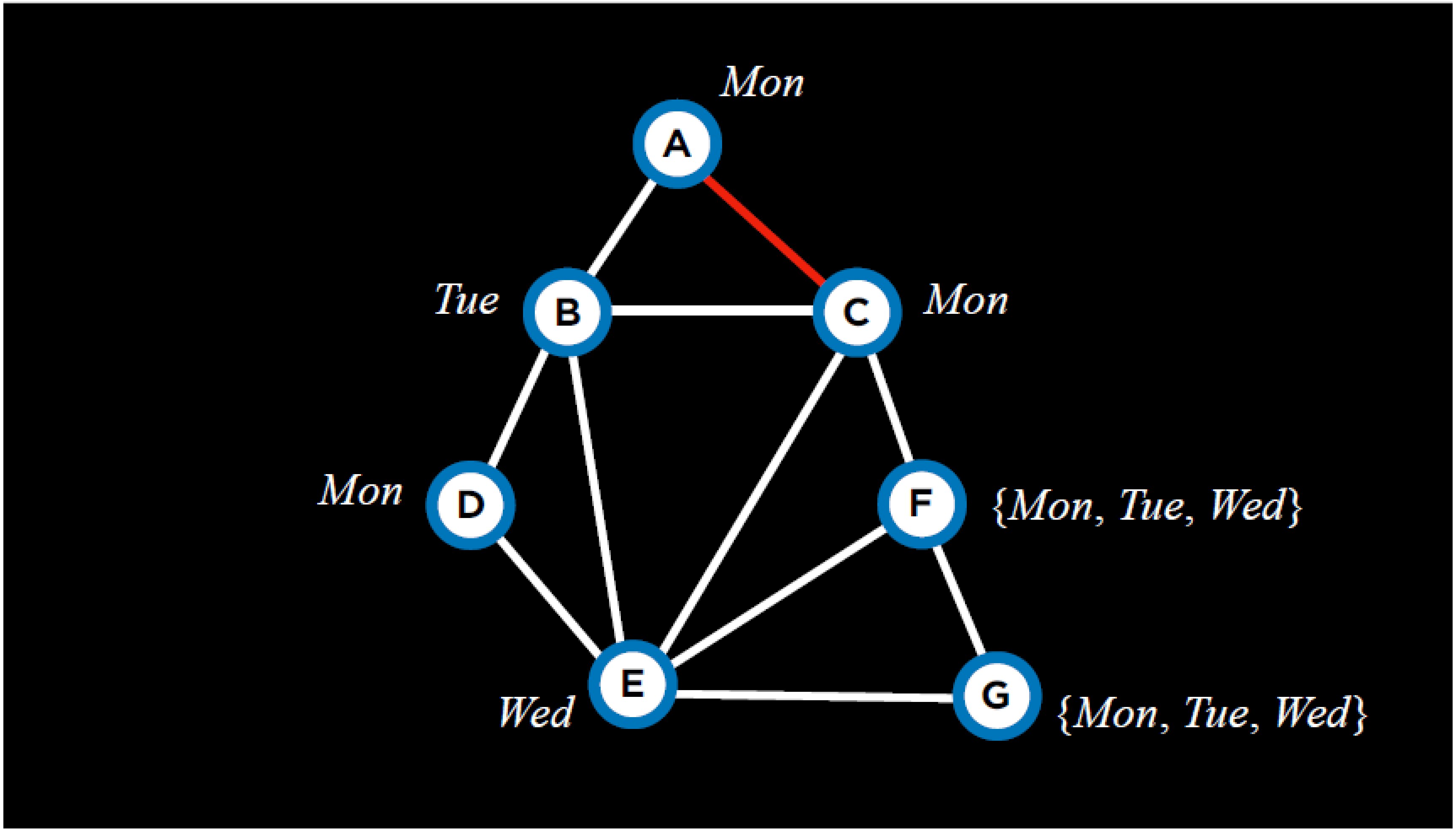

Mon
A
Tue
B
C
{Mon, Tue, Wed}
Mon
D
F
{Mon, Tue, Wed}
E
Tue
G
{Mon, Tue, Wed}



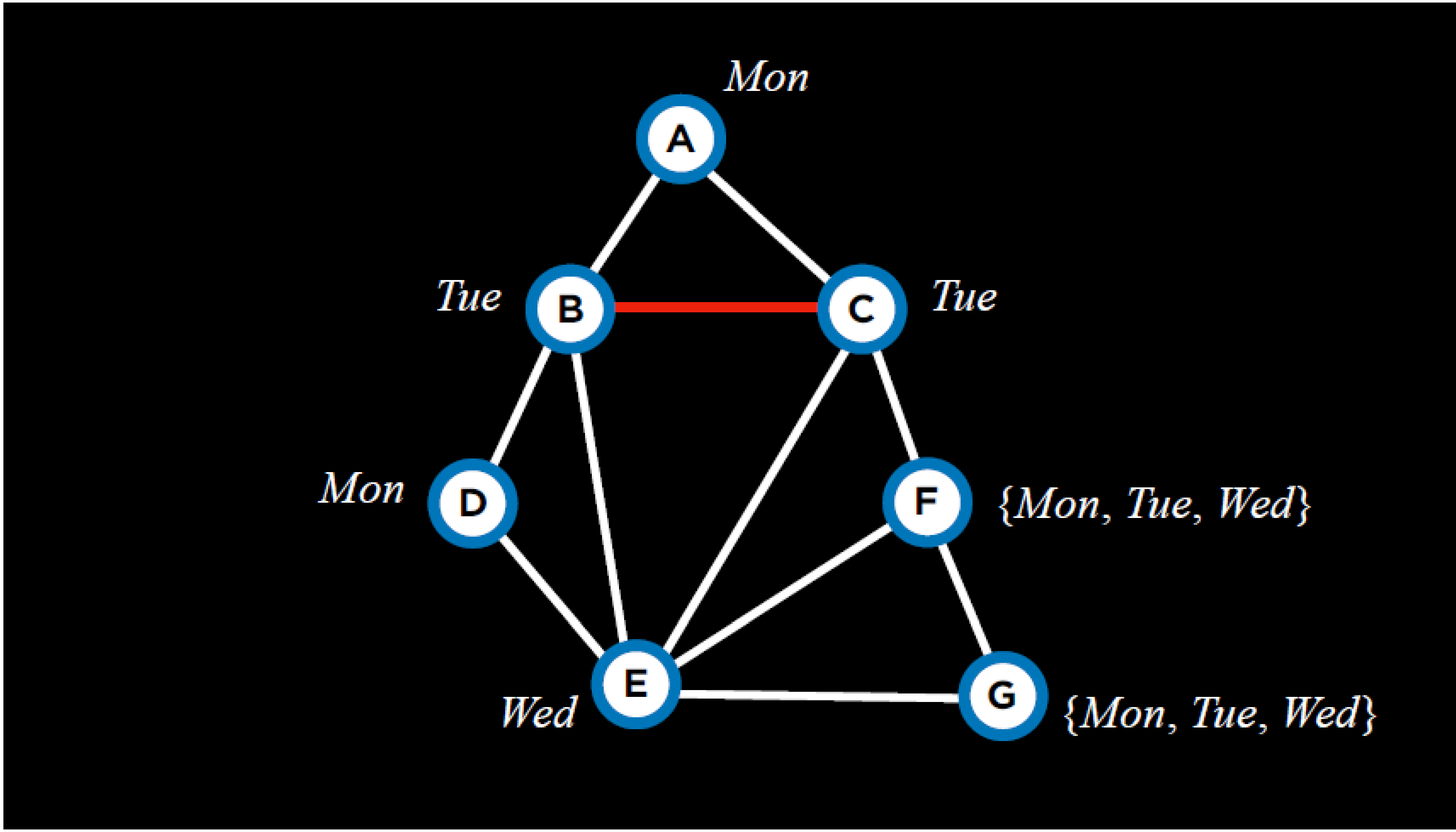

Slide credit : HarvardX CS50AICS50's Introduction to Artificial Intelligence with Python, David J. Malan and Brian Yu



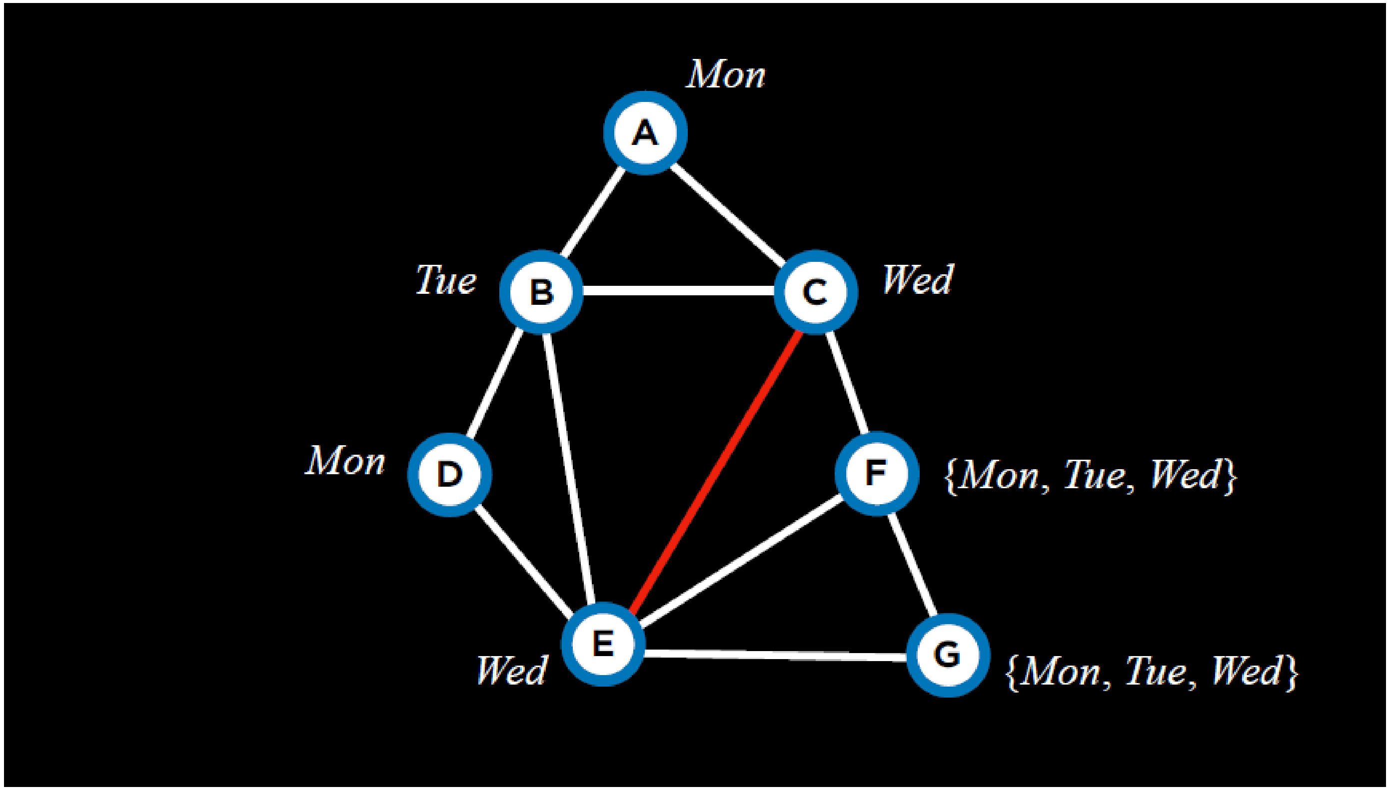

Slide credit : HarvardX CS50AICS50's Introduction to Artificial Intelligence with Python, David J. Malan and Brian Yu



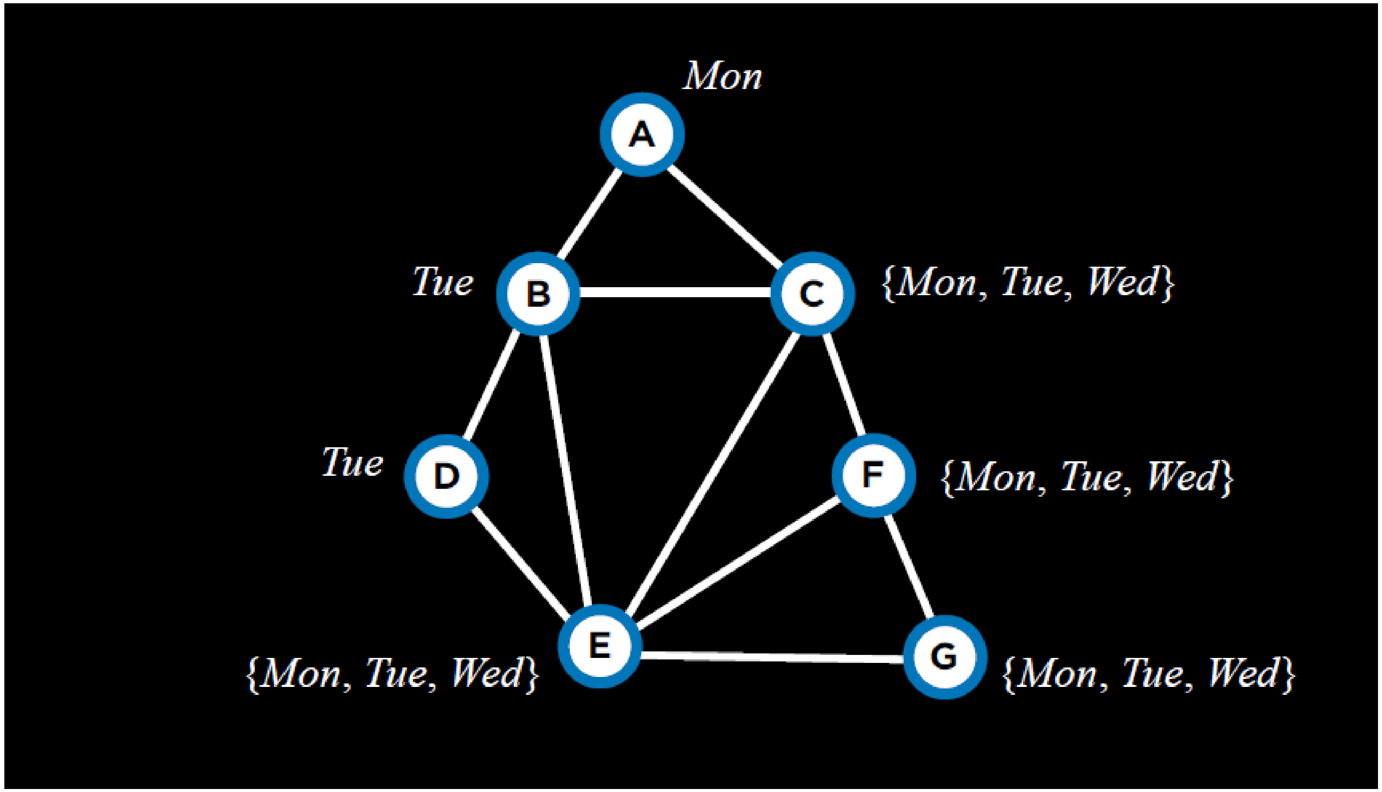

Slide credit : HarvardX CS50AICS50's Introduction to Artificial Intelligence with Python, David J. Malan and Brian Yu



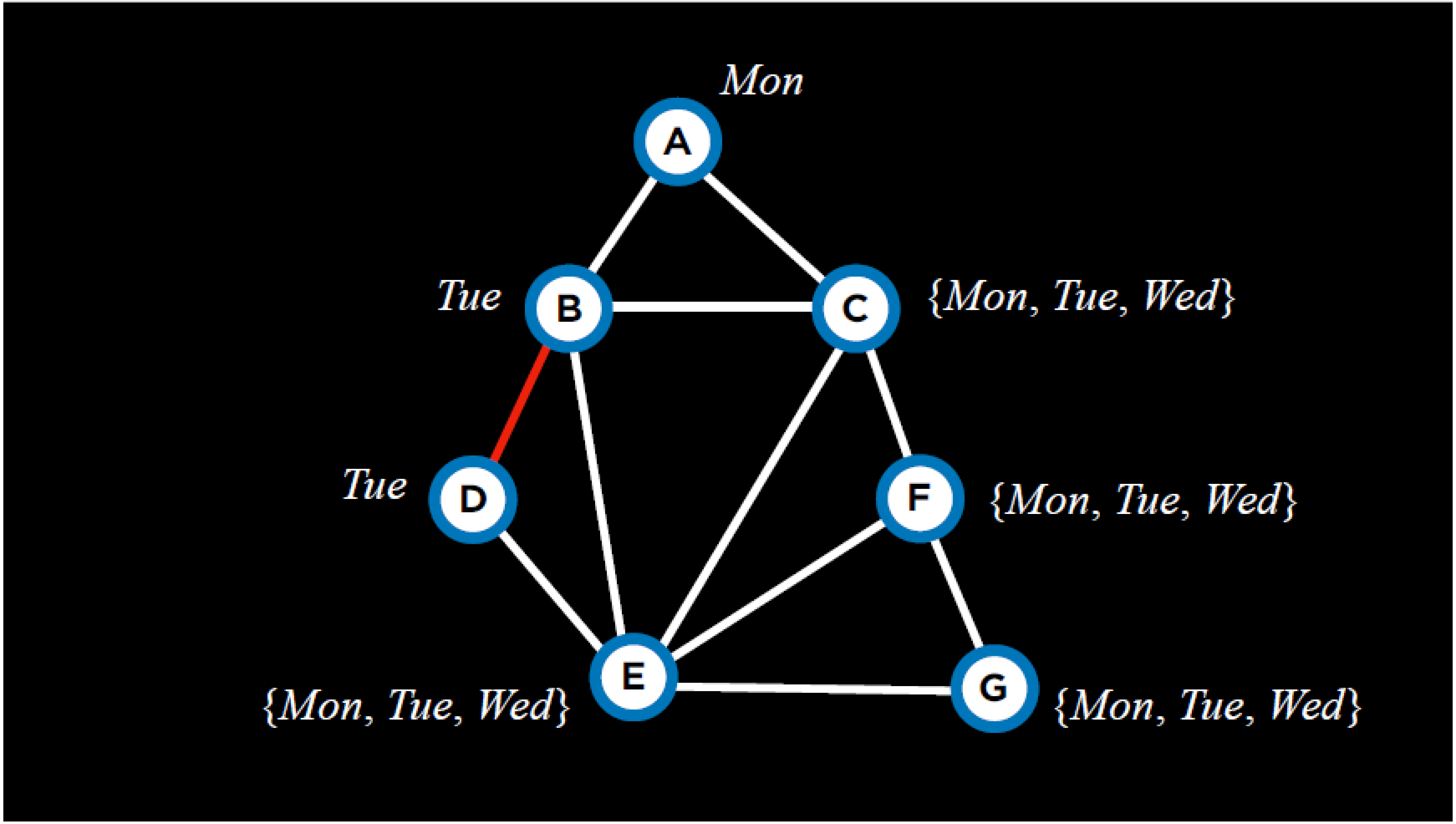

Slide credit : HarvardX CS50AICS50's Introduction to Artificial Intelligence with Python, David J. Malan and Brian Yu



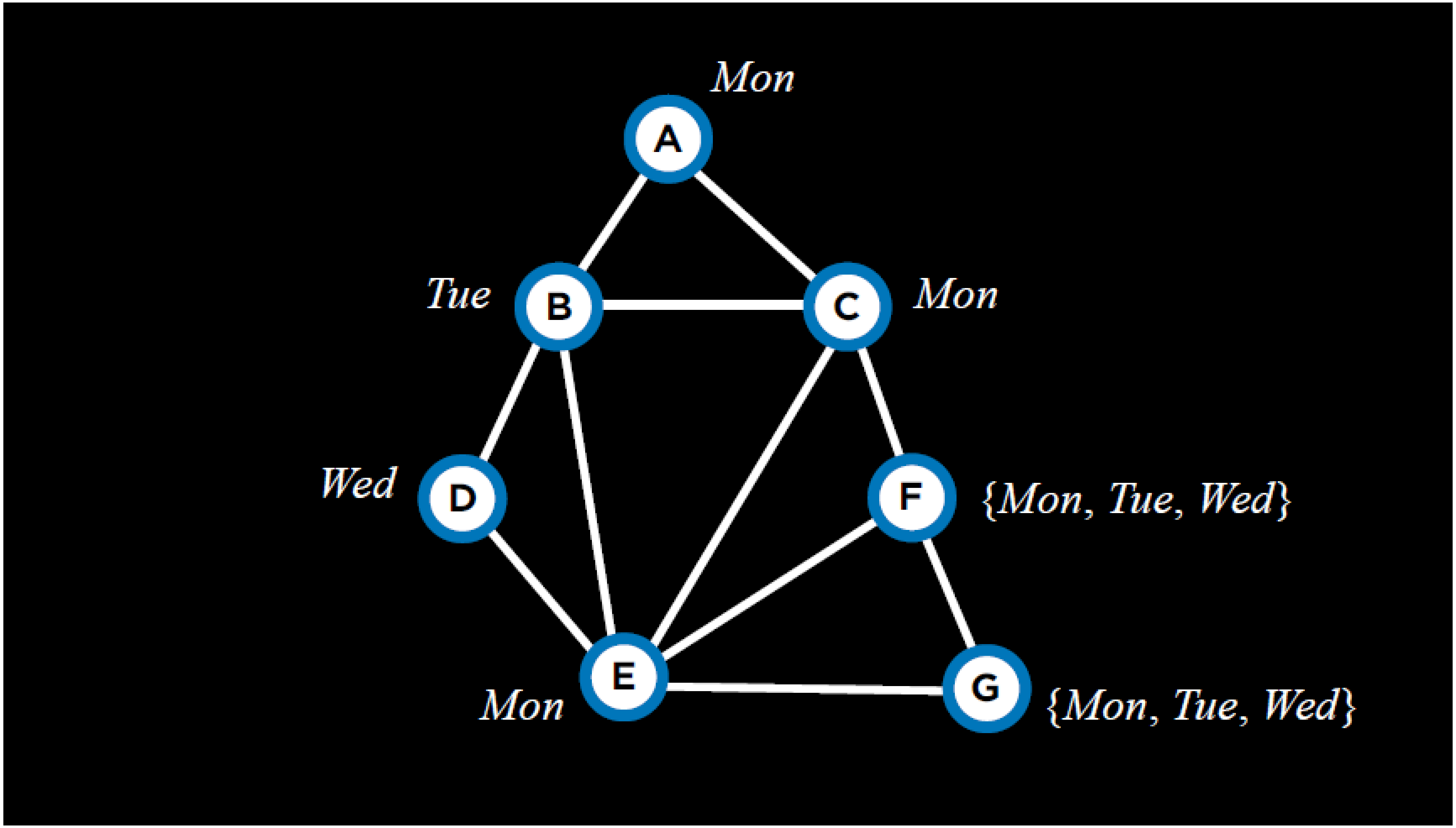

Slide credit : HarvardX CS50AICS50's Introduction to Artificial Intelligence with Python, David J. Malan and Brian Yu

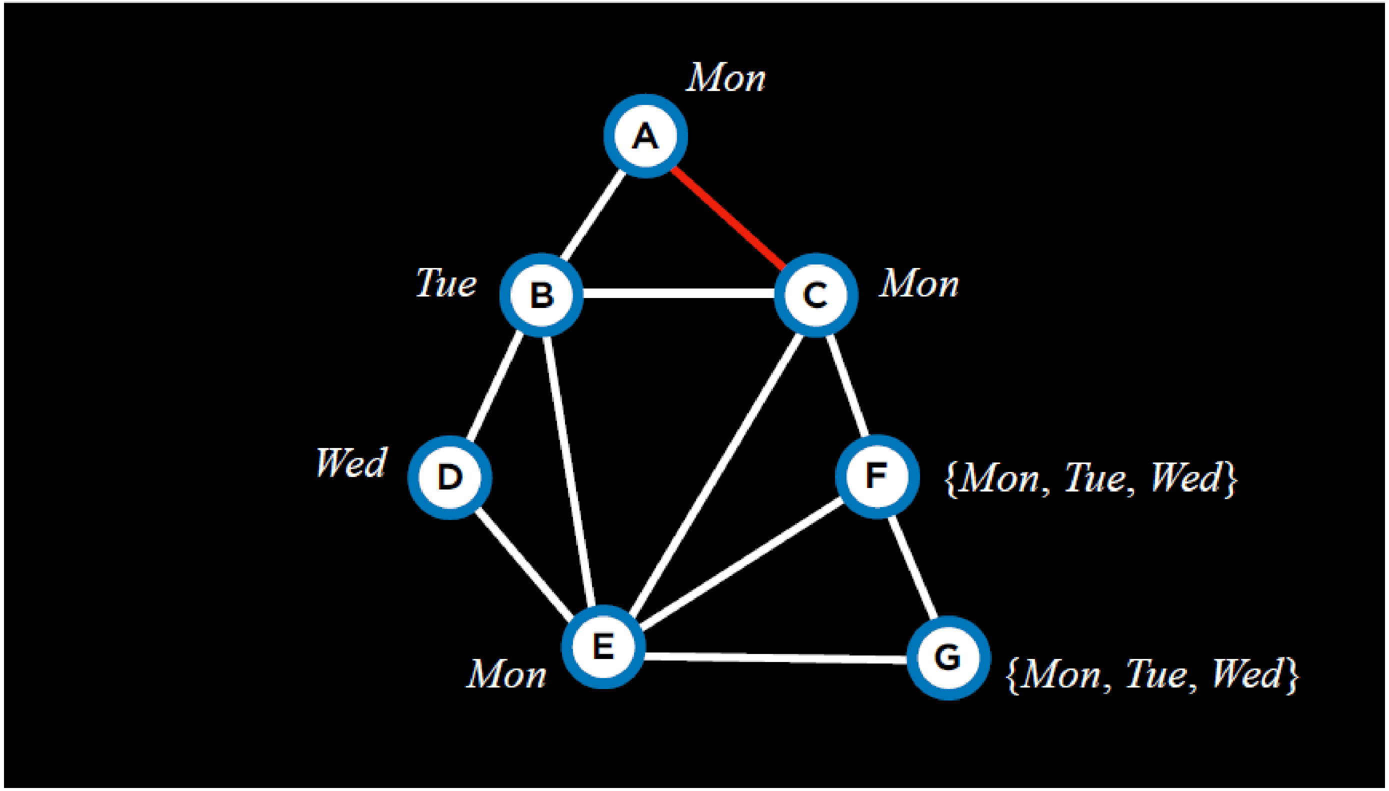



Slide credit : HarvardX CS50AICS50's Introduction to Artificial Intelligence with Python, David J. Malan and Brian Yu

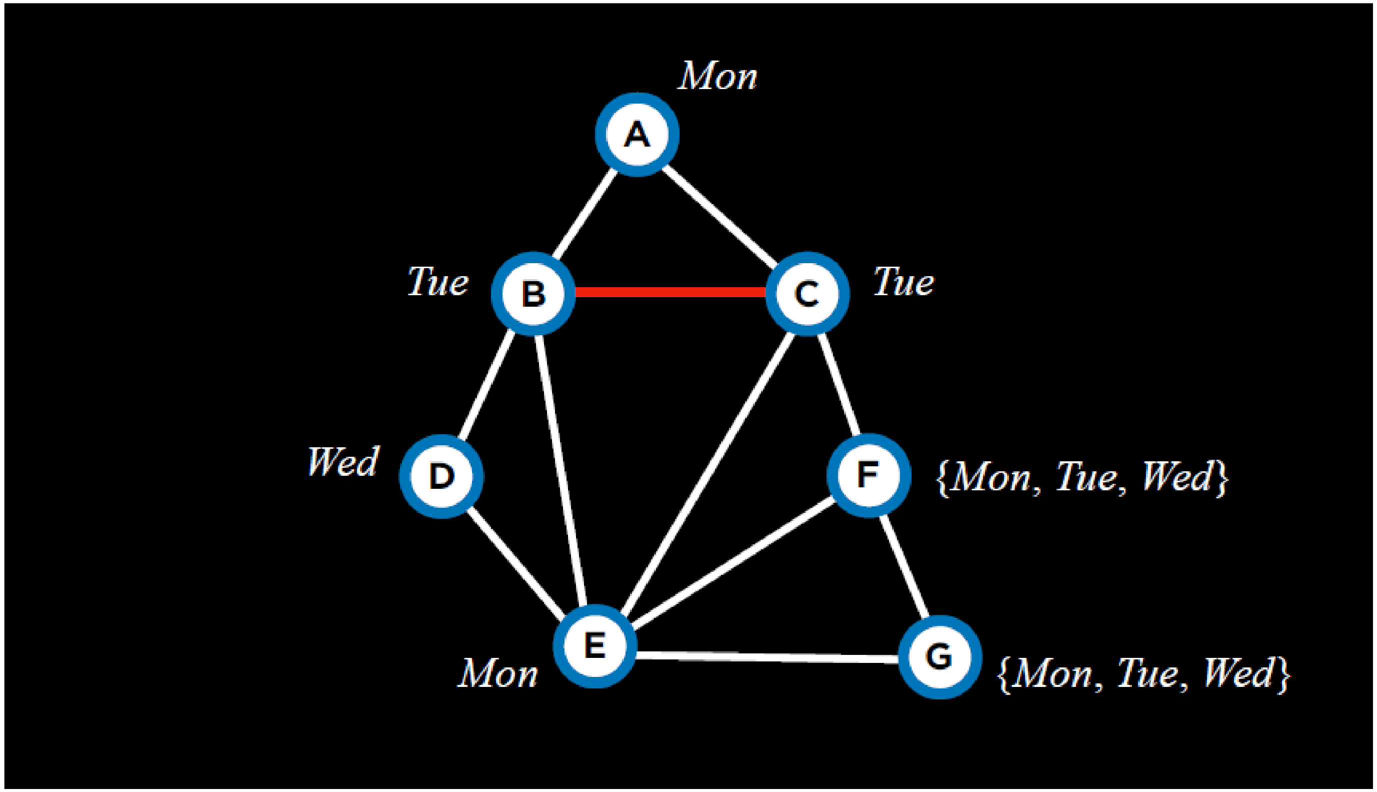


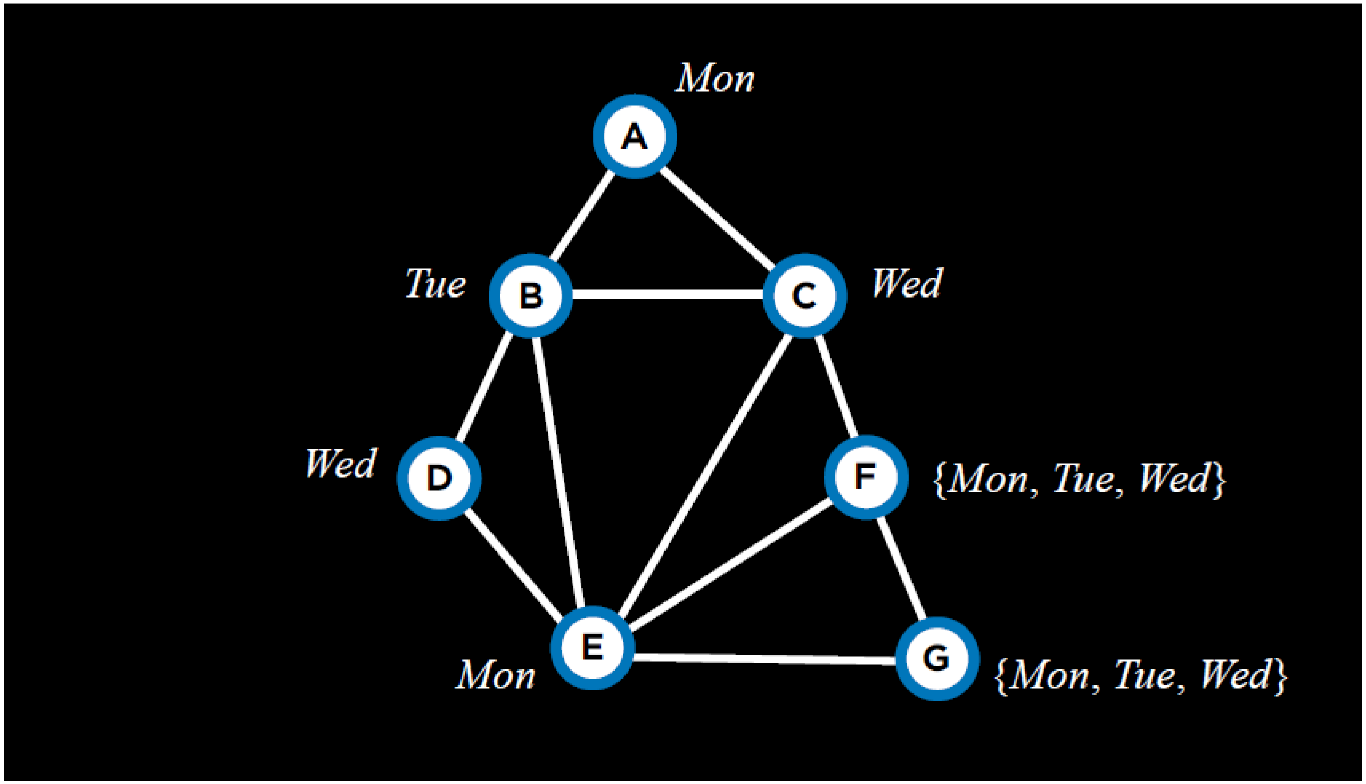

Slide credit : HarvardX CS50AICS50's Introduction to Artificial Intelligence with Python, David J. Malan and Brian Yu


Mon
A
Tue
B
C
Mon
Wed
D
F
{Mon, Tue, Wed}
E
Mon
G
{Mon, Tue, Wed}

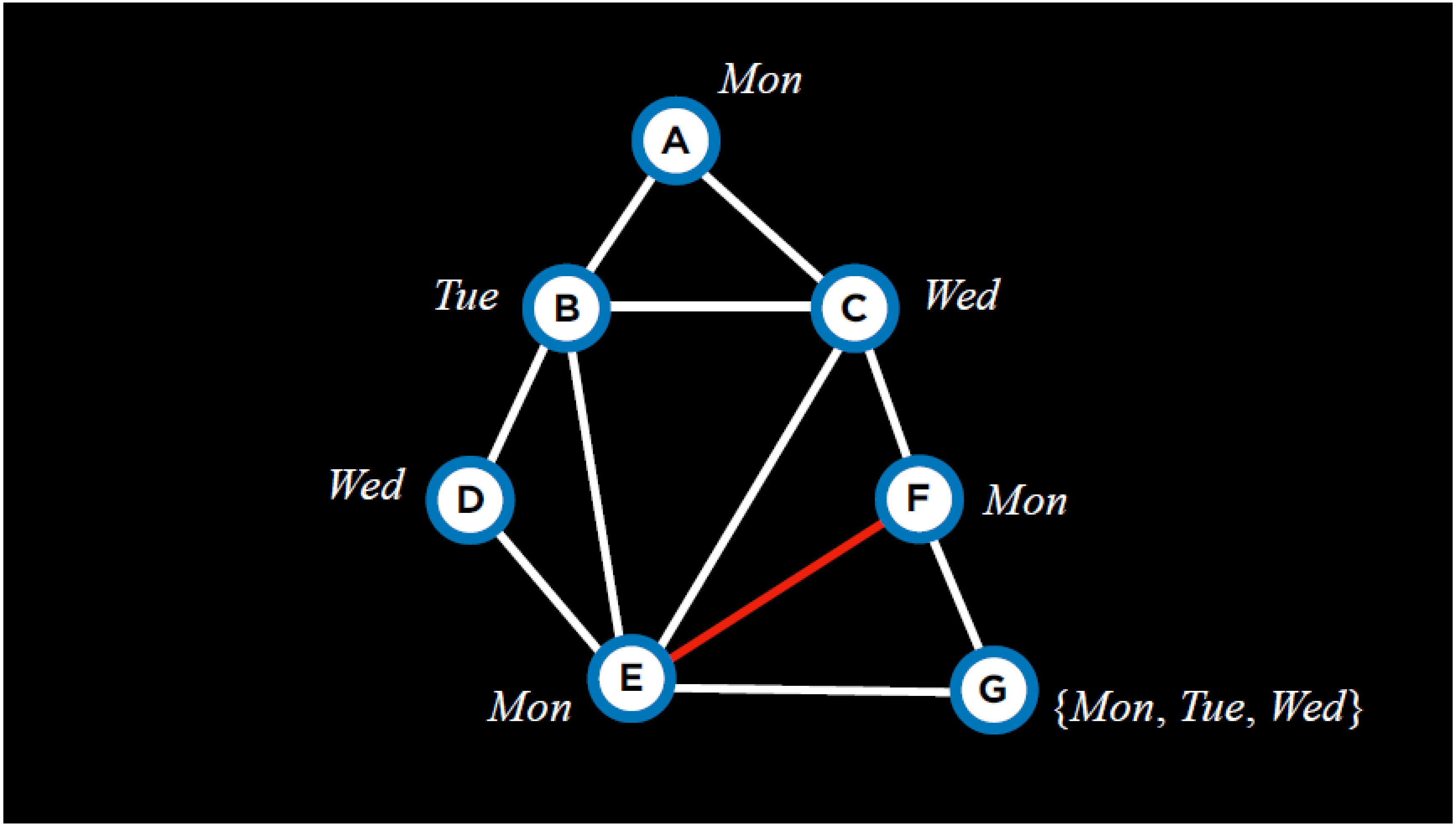



Slide credit : HarvardX CS50AICS50's Introduction to Artificial Intelligence with Python, David J. Malan and Brian Yu

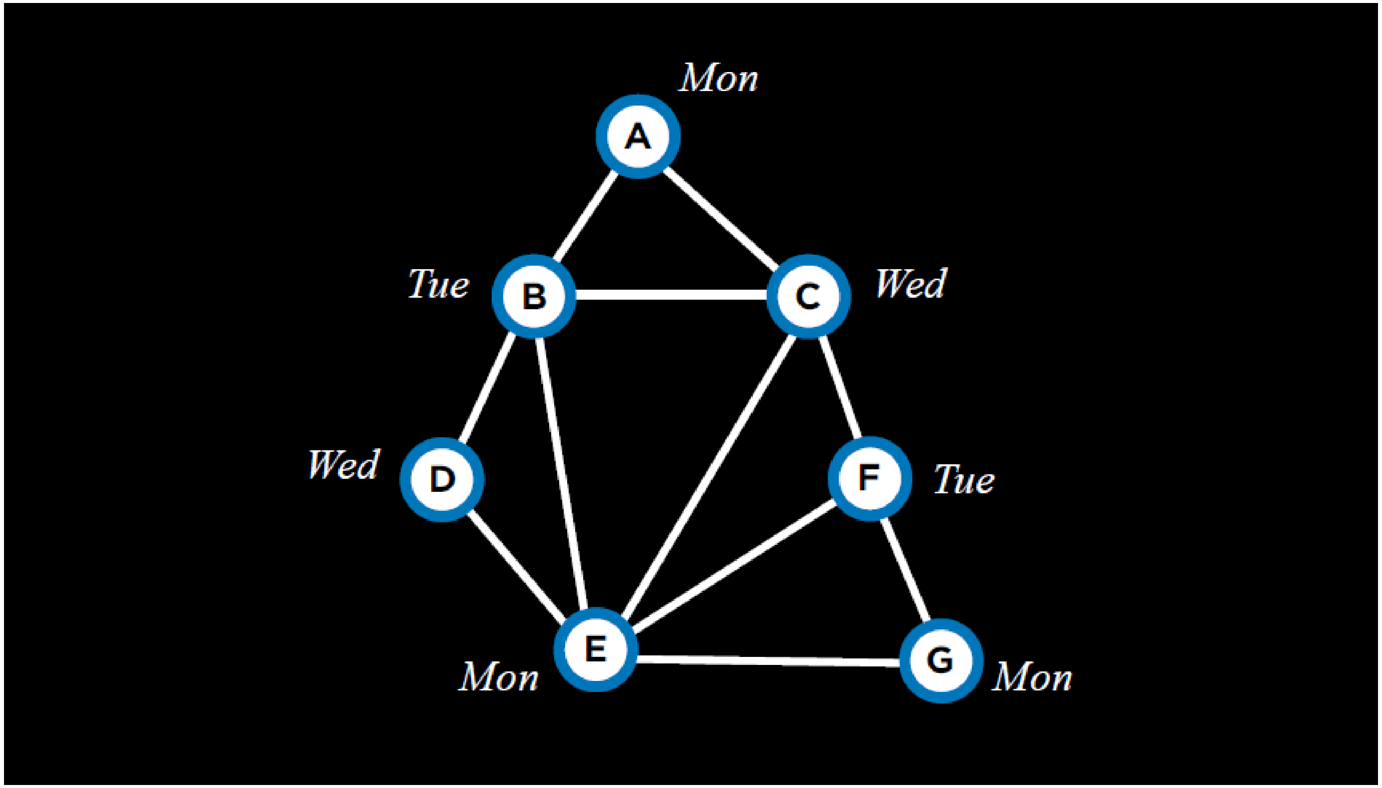

Mon
A
Tue
B
C
Tue
Wed
D
F
{Mon, Tue, Wed}
E
Mon
G
{Mon, Tue, Wed}



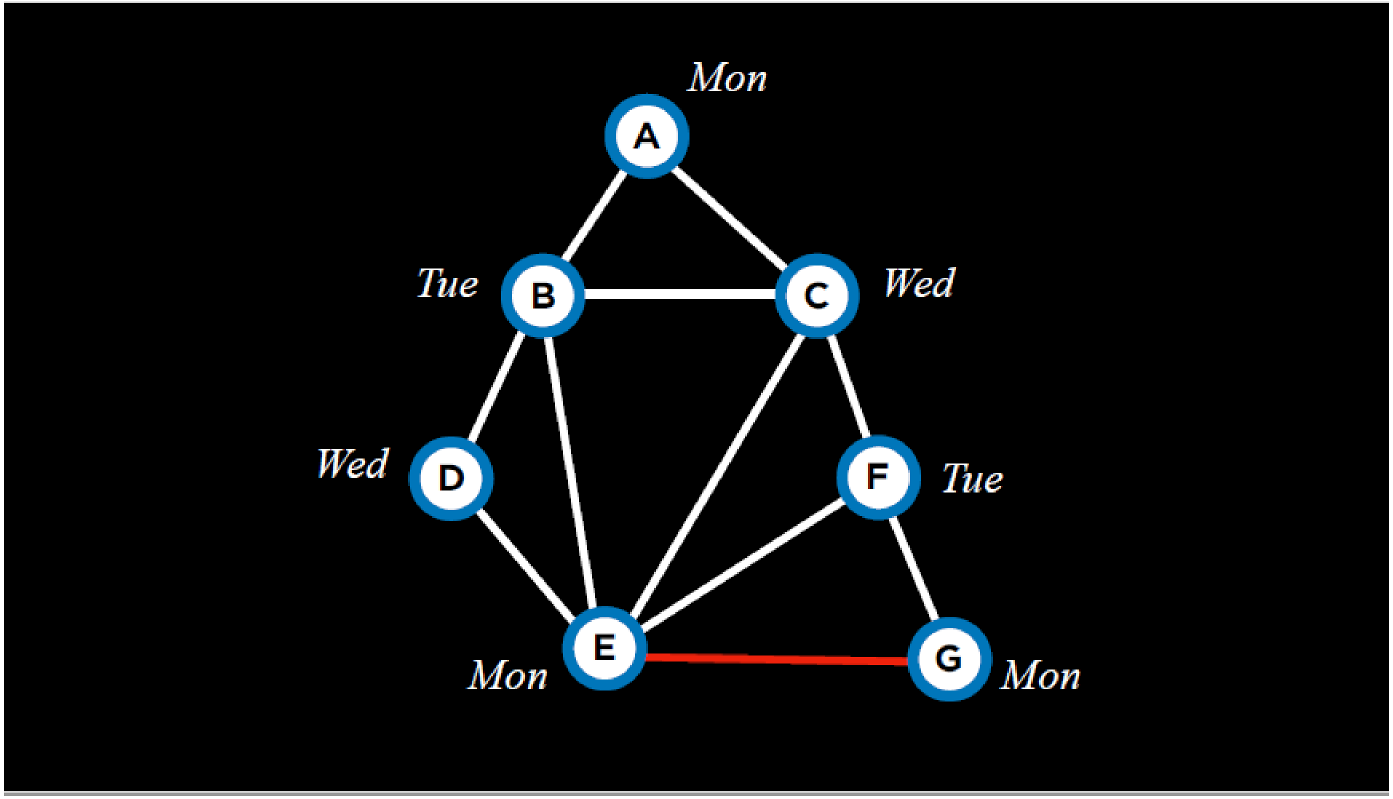

Slide credit : HarvardX CS50AICS50's Introduction to Artificial Intelligence with Python, David J. Malan and Brian Yu



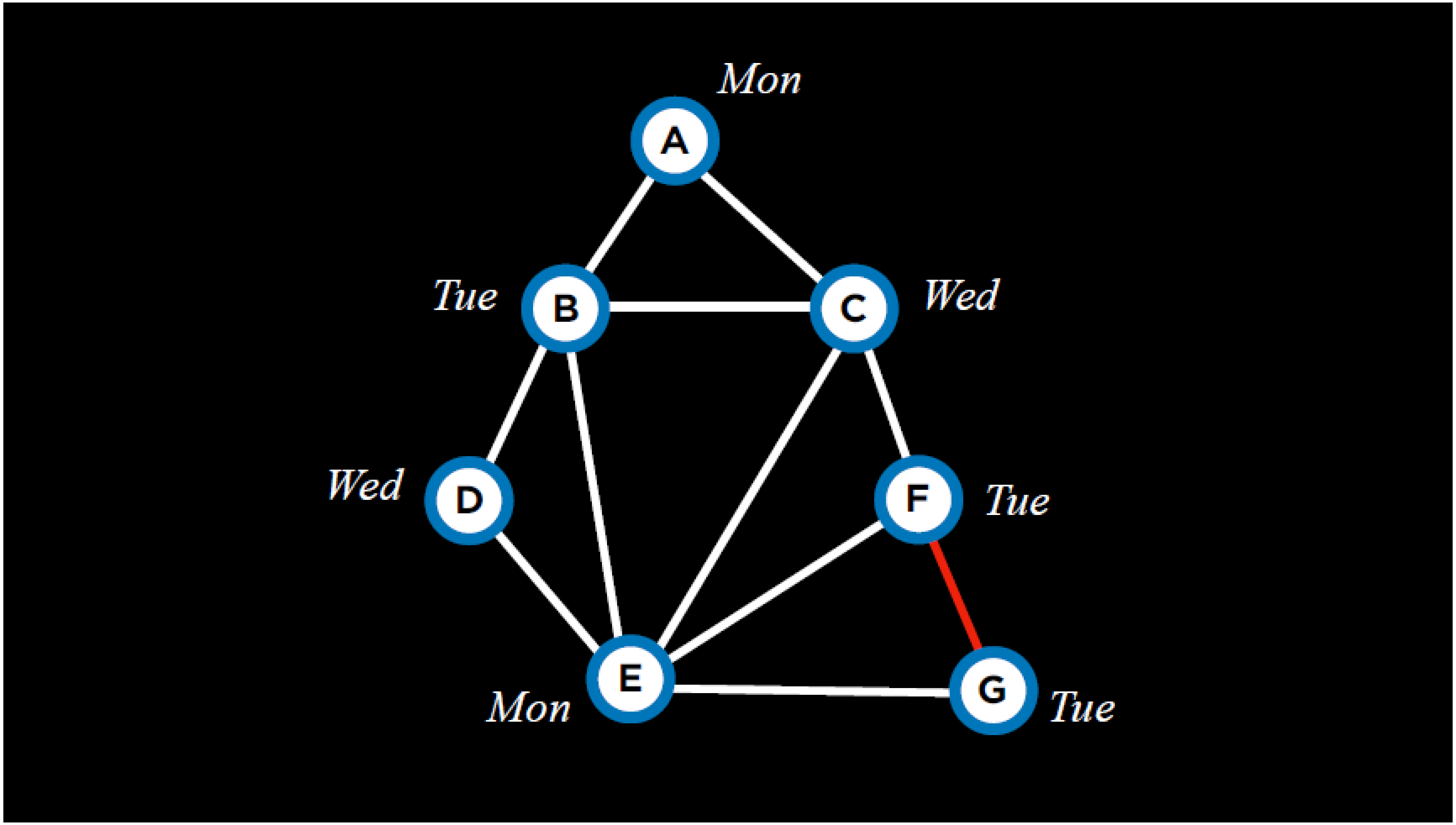

Slide credit : HarvardX CS50AICS50's Introduction to Artificial Intelligence with Python, David J. Malan and Brian Yu




Slide credit : HarvardX CS50AICS50's Introduction to Artificial Intelligence with Python, David J. Malan and Brian Yu

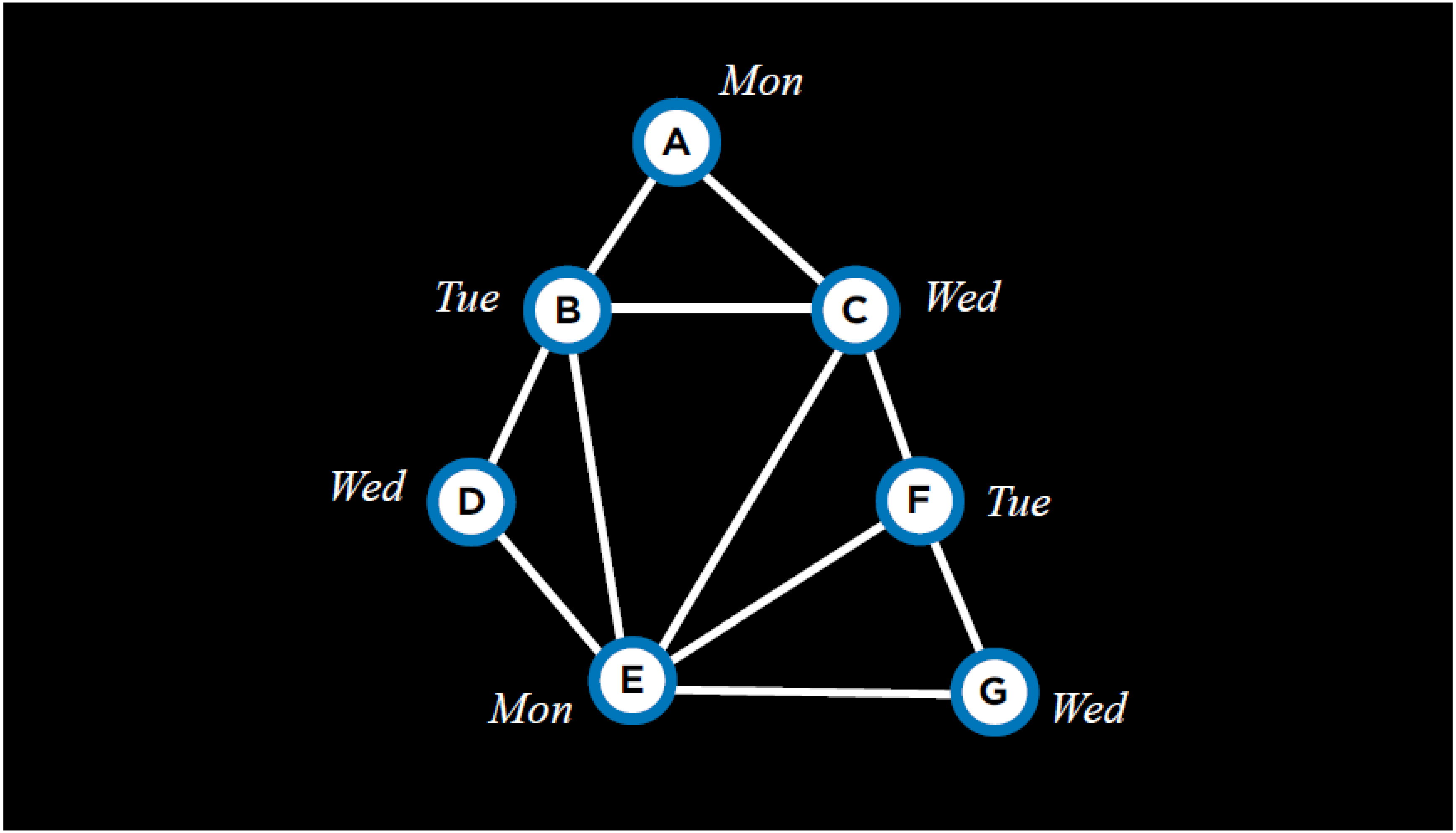



Slide credit : HarvardX CS50AICS50's Introduction to Artificial Intelligence with Python, David J. Malan and Brian Yu




Slide credit : HarvardX CS50AICS50's Introduction to Artificial Intelligence with Python,

David J. Malan and Brian Yu




Slide credit : HarvardX CS50AICS50's Introduction to Artificial Intelligence with Python, David J. Malan and Brian Yu

Slide credit : HarvardX CS50AICS50's Introduction to Artificial Intelligence with Python,
David J. Malan and Brian Yu