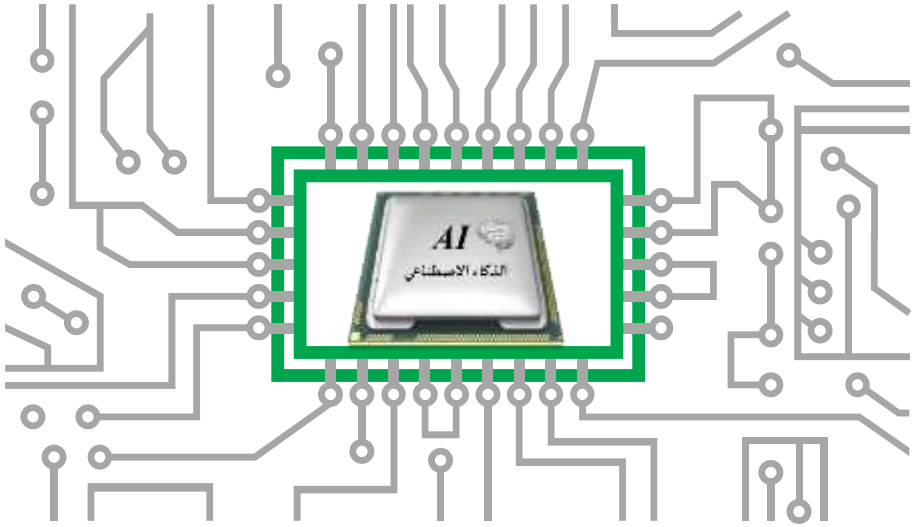
COE 292
Introduction to Artificial Intelligence



Constraint Satisfaction Problems

Outline

- Introduction
- Defining Constraint Satisfaction Problems
- Real-World Examples of CSPs
- Backtracking Search
- Improving Backtracking Search
- CSP Examples
- Conclusion

Introduction

- The search algorithms we discussed so far had no knowledge of the state representation (black box with no internal structure)
  - For each problem we had to design a new state representation
- In Constraint Satisfaction Problem (CSP) we use an internal representation for each state: a set of variables represent these states, each of which has a value
- We can build specialized search algorithms that operate efficiently on this general state representation

Defining Constraint Satisfaction Problems

- A Constraint Satisfaction Problem (CSP) consists of three components:
  - A set of variables that can take a finite value; $\boldsymbol{V}$={V_1, V_2, V_3,…, V_n}
  - A set of domains, one for each variable from which a value is picked $\boldsymbol{D}$={D_{v1}, D_{v2}, D_{v3},…, D_{vn}}.
  - A set of constraints $\boldsymbol{C}$={C_1, C_2, C_3,…, C_m} that specify allowable combination of values
- A solution to a CSP is an assignment of values to all variables such that all constraints are satisfied

Defining Constraint Satisfaction Problems

- Each variable in the variable set can belong to different domains
- The elements in the domain can be both continuous and discrete
- All sets should be finite except for the domain set
- Each constraint C:
  - Has a set of variables it is over, called scope of the constraint
    - e.g., the constraint $V_2 \neq 2$ has a scope $\{V_2\}$; while $V_1 \geq V_2$ has scope $\{V_1, V_2\}$
  - Has a restriction on the values of the variables in the scope
    - e.g., the constraint $V_2 \neq 2$ and $V_1 \geq V_2$ is restricting the values of V_1 and V_2
  - Is a Boolean function that maps assignments to the variables in its scope to true/false
    - e.g., $C(V_1 = a, V_2 = b, V_3 = c) = True$

Defining Constraint Satisfaction Problems

- Unary Constraints (over single variable)
  - e.g. $C_1(X)$: X=2; $C_2(Y)$: Y>5
- Binary Constraints (over two variables)
  - e.g. $C_3(X,Y)$: X+Y<6
- Higher-order constraints (over 3 or more variables)
  - We can convert any constraint into a set of binary constraints (may need some auxiliary variables)
- Can be represented by Constraint Graph that represents the relationship between the variables
  - Nodes are variables, edges show constraints

Real-World Examples of CSPs

- Assignment problems: e.g., who teaches what class
- Timetabling problems: e.g., which class is offered when and where?
- Transportation scheduling
- Factory scheduling
- Fault diagnosis
- … and many more

CSPs as Search Problems

- CSPs are a special subset of search problems in which:
  - States: assignments of values to a subset of the variables
  - Initial state: the empty assignment (no variables assigned values)
  - Goal state: complete assignment of values to all variables that satisfies all the constraints
  - Actions: Assign a value to an unassigned variable
  - Solution: The solution of a CSP is an assignment of a value to each variable such that all constraints are satisfied
    - Unlike previous search problems, knowing the path from the initial state to the goal state is not important

Backtracking Search

- Backtracking search with single-variable assignments for a CSP is the basic uninformed search for solving CSPs
- Backtracking search is a Depth-First Search (DFS) with two improvements:
  - Consider assignments to a single variable at each step
  - Check constraints as you go
    - consider only new assignments which do not conflict previous assignments
- Backtracking search incrementally builds candidates to the solutions and abandons a candidate ("backtracks") as soon as it determines that the candidate cannot possibly be completed to a valid solution

Backtracking Search Algorithm

Algorithm: Backtracking-Search(assignment)

```
1:  if assignment is complete then
2:      return assignment
3:  V = Select the next unassigned-variable(variables)
4:  for each value in Domain(V) do
5:      if {V = value} is consistent with all constraints then
6:          add {V = value} to assignment
7:          result = Backtracking-Search(assignment)
8:          if result ≠ failure then
9:              return result
10:         remove {V = value} from assignment
11:  end for
12: return failure
```

Solving CSPs using Backtracking Search

- To solve a CSP, we require the following steps:
  - Step 1: Create a variable set
  - Step 2: Create a domain set
  - Step 3: Create a constraint set with variables and domains (if possible) after considering the constraints
  - Step 4: Set the order of variable and domain sets
  - Step 5: Set the initial state with empty assignments for all variables
  - Step 6: Use backtracking search to assign values to all variables. A solution is found if variables assignments satisfy all constraints

CSP Example 1

- Example 1:
  - Suppose you have a set of variables $V1$, $V2$, $V3$. Each variable in the set V, has a non-empty domain of possible values, particularly $D_{V1} = \{1,2,3\}$, $D_{V2} = \{1,2\}$, and $D_{V3} = \{2,3\}$. A set of constraints $C = \{V1>V2, V2\neq V3, V1\neq V3\}$. Find the values that satisfy the constraints
- Solution:
  - The goal is to assign a value to each variable such that none of the constraints are violated
  - We will use backtracking search to solve the problem

CSP Example 1 - Solution

- Step 1: We need to create the variable set, from the problem
  - V={$V1$,$V2$,$V3$} is the variable set
- Step 2: We need to define the domain set; the domain is as follows:
  - D_{V1} = {1,2,3}
  - D_{V2} = {1,2}
  - D_{V3} = {2,3}
- Step 3: Constraints are:
  - $C1 = V1 > V2$
  - $C2 = V2 \neq V3$
  - $C3 = V1 \neq V3$

CSP Example 1 - Solution

- Step 4: The order of selecting variables and values:
  - The same order they appear in the variables and domain sets
- Step 5: Set the initial assignment of variables:
  - $V1 = \{\}, V2 = \{\}, V3 = \{\}$
- Step 6: Use backtracking search to find the solution by going over all possible values (one at a time) and make sure that all constraints are satisfied. If you reach an assignment of values that breaks the constraints, backtrack to the last decision made and change the value
  - Detailed steps are given in next slides

CSP Example 1 - Solution

- Backtracking search steps:
  1. Start with $V1$=1 (the first value in its domain). Since no values are assigned to $V2$ and $V3$, no constraints are dissatisfied → move to the next assignment
  2. Assign $V2$=1 (the first value in its domain). Check if $C1$=$V1$>$V2$ is satisfied. Since 1>1 is not satisfied, then the pair of values $V1$=1,$V2$=1 is not a possible assignment → backtrack

Domain:

$V1 = \{1,2,3\}$

$V2 = \{1,2\}$

$V3 = \{2,3\}$

Constraints:

$C1 = V1 > V2$

$C2 = V2 \neq V3$

$C3 = V1 \neq V3$

CSP Example 1 - Solution

3. Backtrack:
   - Change the value of $V2$ by selecting the next possible value in the domain of $V2$, i.e. set $V2$=2
   - However, $C1$ is not satisfied again, so we backtrack
   - Since we have exhausted the domain of $V2$, backtrack and change the value of $V1$, i.e., set $V1$=2 then move over to $V2$ and assign $V2$=1
   - Now constraint $C1$=$V1$>$V2$ is satisfied (i.e., 2>1)
   - Since no value is assigned for $V3$ (i.e., condition cannot be checked), no constraints are dissatisfied

Domain:

$V1 = \{1,2,3\}$

$V2 = \{1,2\}$

$V3 = \{2,3\}$

Constraints:

$C1 = V1 > V2$

$C2 = V2 \neq V3$

$C3 = V1 \neq V3$

CSP Example 1 - Solution

4. Assign $V3$=2 (the first value in its domain), so we get the tuple $\{V1=2, V2=1, V3=2\}$
   - Check the constraints
     - $C1$ is satisfied,
     - $C2$ is satisfied,
     - $C3$ is not satisfied due to C($V1 \neq V3$), so we backtrack
5. Assign $V3$=3 (the next value in the domain) so we get the tuple $\{V1=2, V2=1, V3=3\}$
   - $C1$, $C2$ and $C3$ are all satisfied

- Since we have satisfied all constrains $C=\{C1, C2, C3\}$ we stop as we **found a solution to the CSP**

Domain:

$V1 = \{1,2,3\}$

$V2 = \{1,2\}$

$V3 = \{2,3\}$

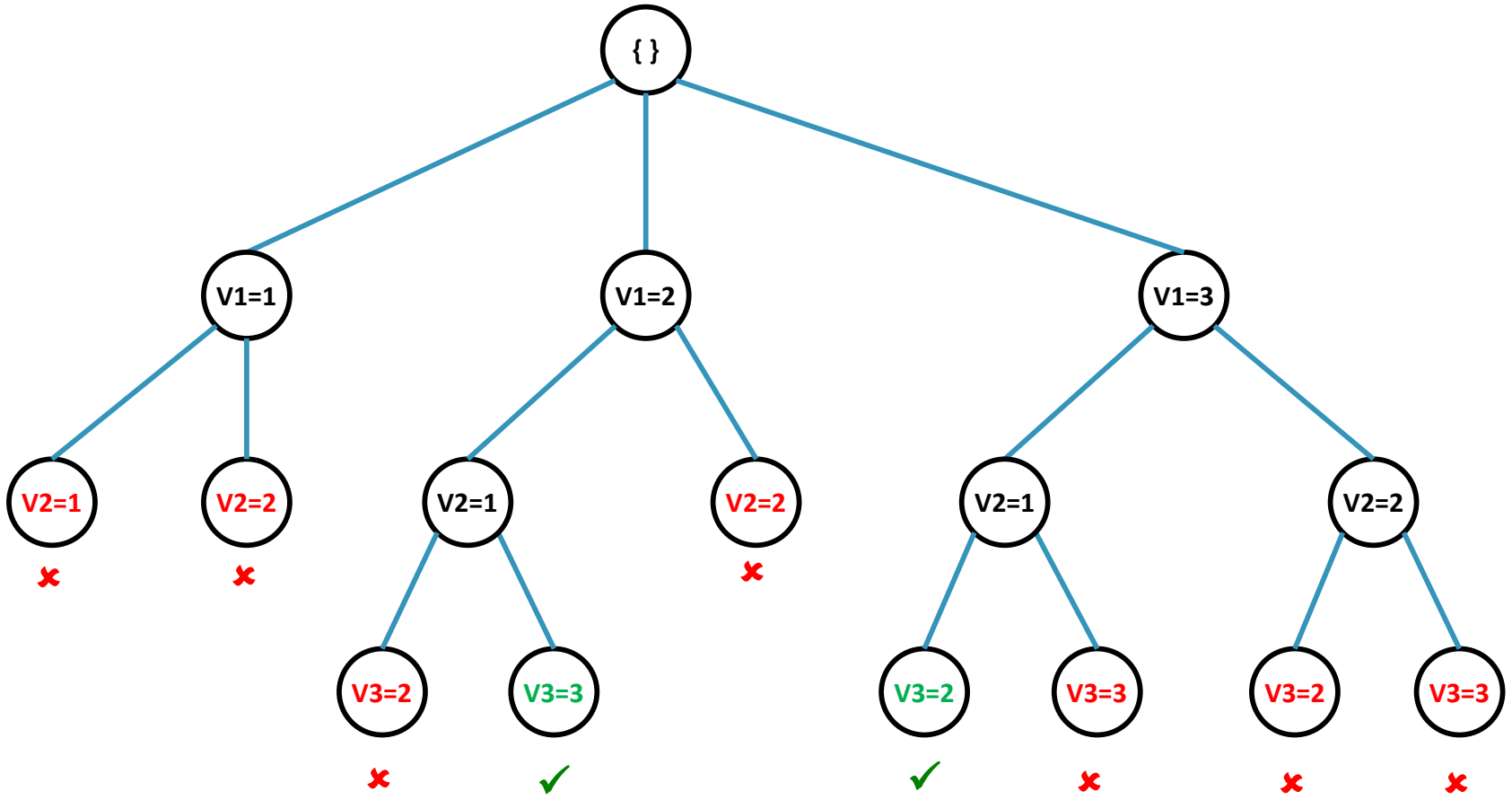
Constraints:

$C1 = V1 > V2$

$C2 = V2 \neq V3$

$C3 = V1 \neq V3$

CSP Examples 1: Solution's Search Tree



Domain:

$V1 = \{1,2,3\}$

$V2 = \{1,2\}$

$V3 = \{2,3\}$

Constraints:

$C1 = V1 > V2$

$C2 = V2 \neq V3$

$C3 = V1 \neq V3$

The search tree shows that **two** solutions to this CSP are possible depending on the order of selecting variables

Map Coloring

- The **map-coloring** CSP requires that you assign a color to each region of a map such that any two regions sharing a border have different colors
- The constraints for the **map-coloring** problem are:
  1. Each region is assigned one color only, of C possible colors
  2. The color assigned to one region cannot be assigned to adjacent regions

CSP Example 2: Map Coloring

- Given four colors, Red, Green, Yellow , Blue denoted by R, G, Y, B,

- A computer program must Color the regions *Tabuk, Hail, Madina, Al-Qasim, Makkah* and *Riyadh* such that no adjacent regions have the same color

CSP Example 2: Map Coloring

- Convert the problem to CSP
  - Variables:
    - Regions V={V1,V2,V3,V4,V5,V6}

V1 = *Tabuk*	V2 = *Ha'il*	V3 = *Qassim*
V4 = *Riyadh*	V5 = *Makkah*	V6 = *Medina*

  - Domains and order:
    - Colors allowed

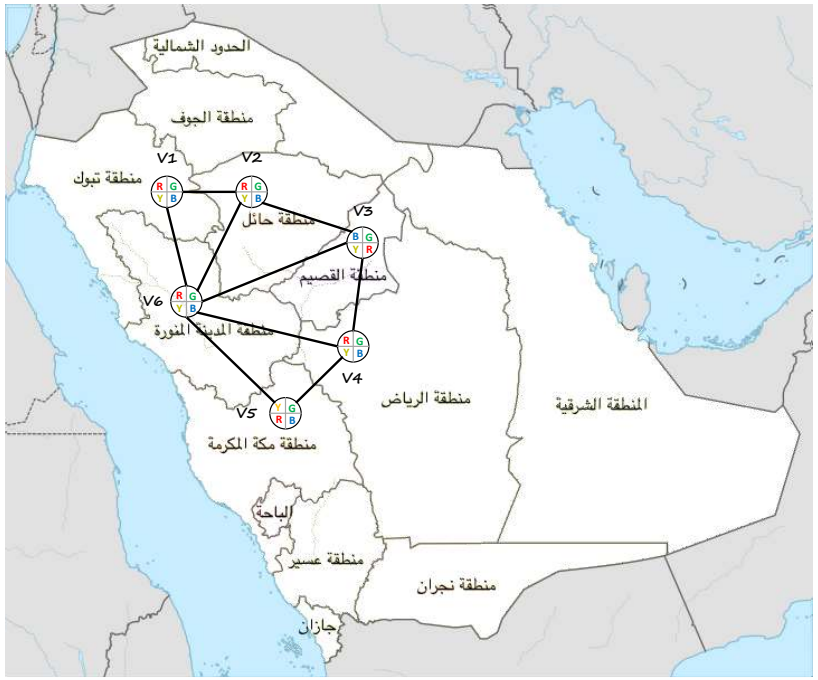
D1= {R, G , Y , B}	D2= {R, G , Y , B}	D3= {B, G, Y, R}
D4= {R, G , Y , B}	D5= {Y , G, R, B}	D6= {R, G , Y , B}

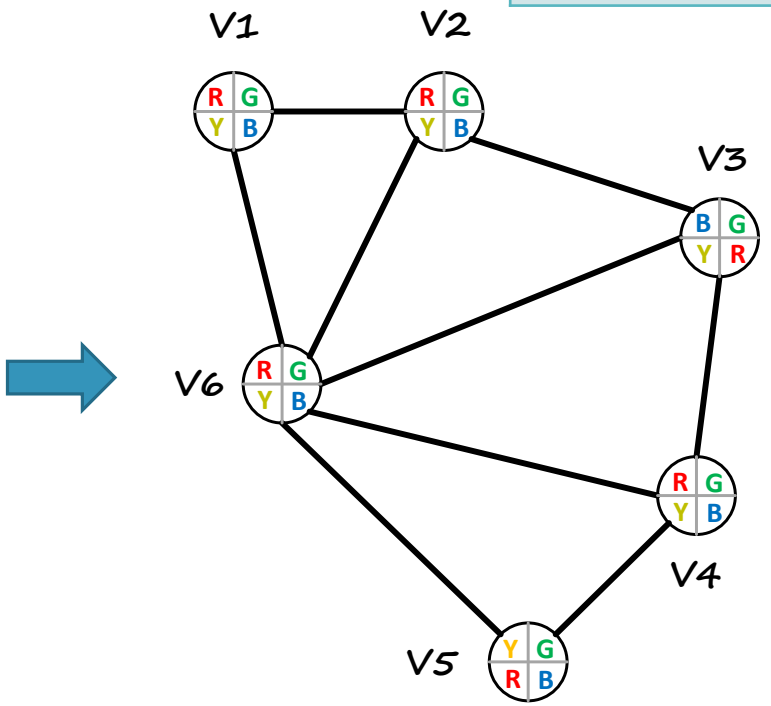
  - Constraints:
    - No two regions sharing a boundary have the same color

CSP Examples 2: Map-Coloring – Backtracking

- Convert the map into a graph:

V1 = *Tabuk*	V2 = *Ha'il*	V3 = *Qassim*
V4 = *Riyadh*	V5 = *Makkah*	V6 = *Medina*





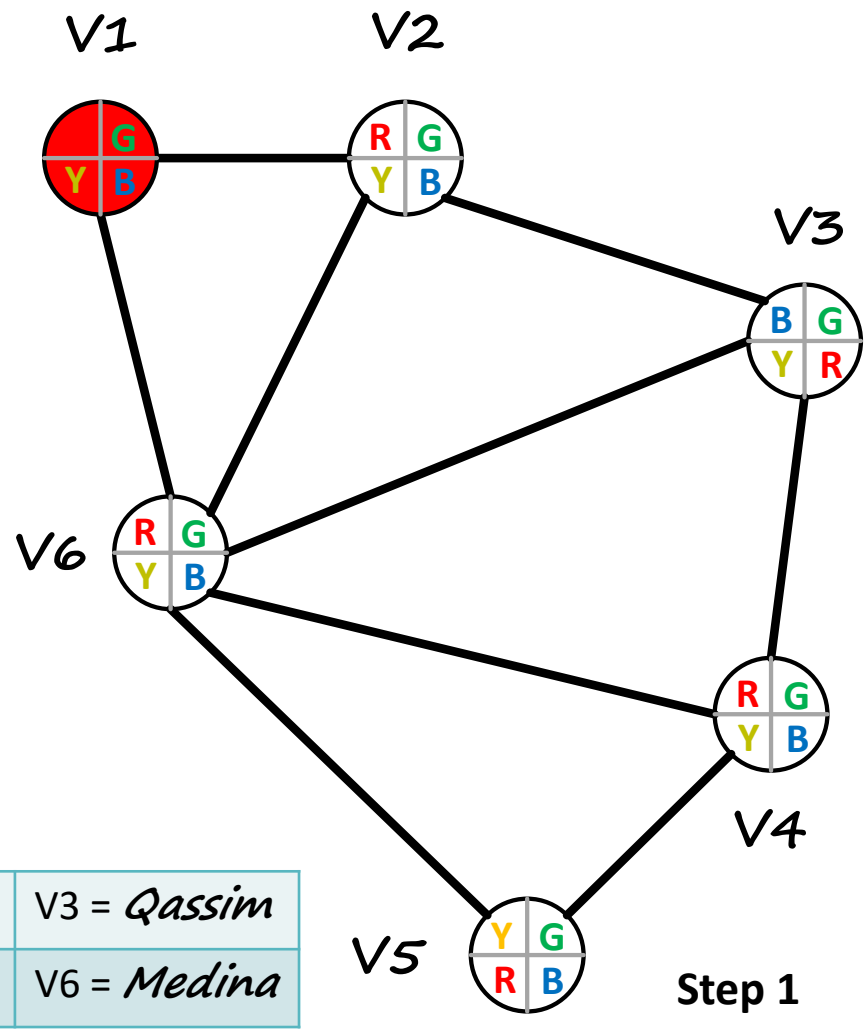
Step 0

A constraint is represented as an edge connecting two nodes in a graph

V1!=V2	V1!=V6	V2!=V3
V2!=V6	V3!=V4	V3!=V6
V4!=V5	V4!=V6	V5!=V6

CSP Examples 2: Map-Coloring – Backtracking

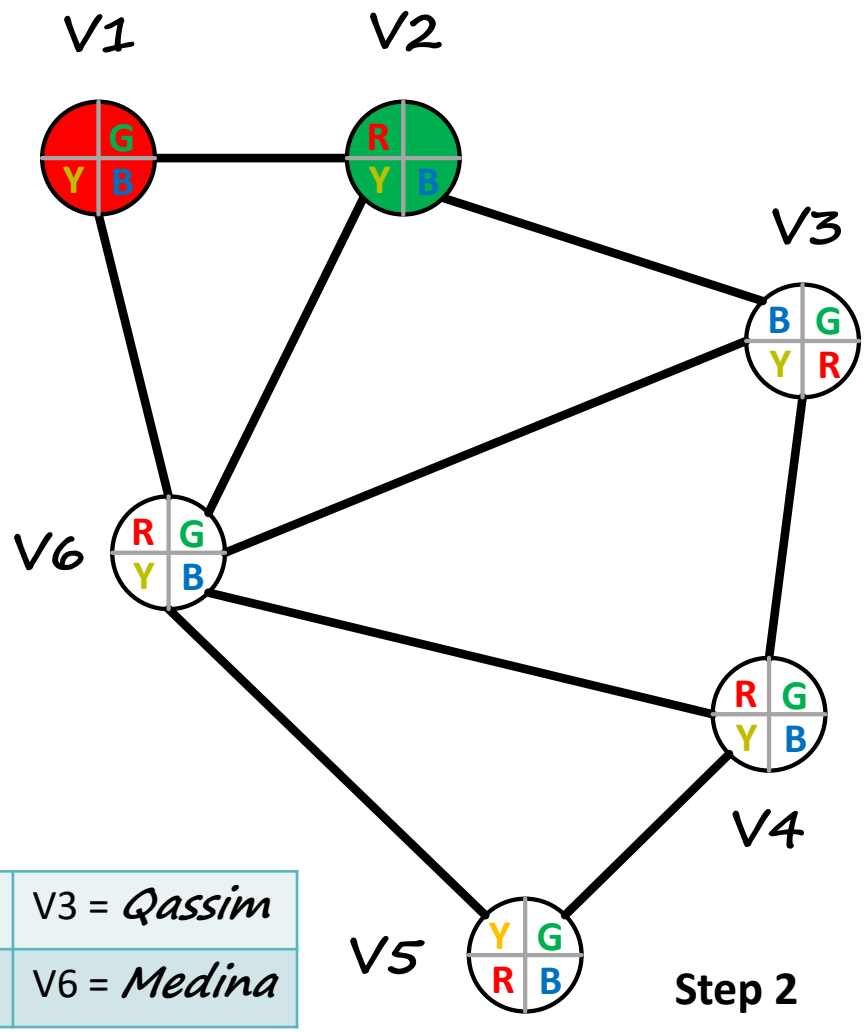
- Step 1: Select next unassigned variable in order from the variable set V={V1, V2, V3, V4, V5, V6}
  - V1 = *Tabuk*
- Step 2: Color the selected variable (region) with the first value in the domain; **R**
- Step 3: Check all constraints
  - All constraints are OK



V1 = *Tabuk*	V2 = *Ha'il*	V3 = *Qassim*
V4 = *Riyadh*	V5 = *Makkah*	V6 = *Medina*

CSP Examples 2: Map-Coloring – Backtracking

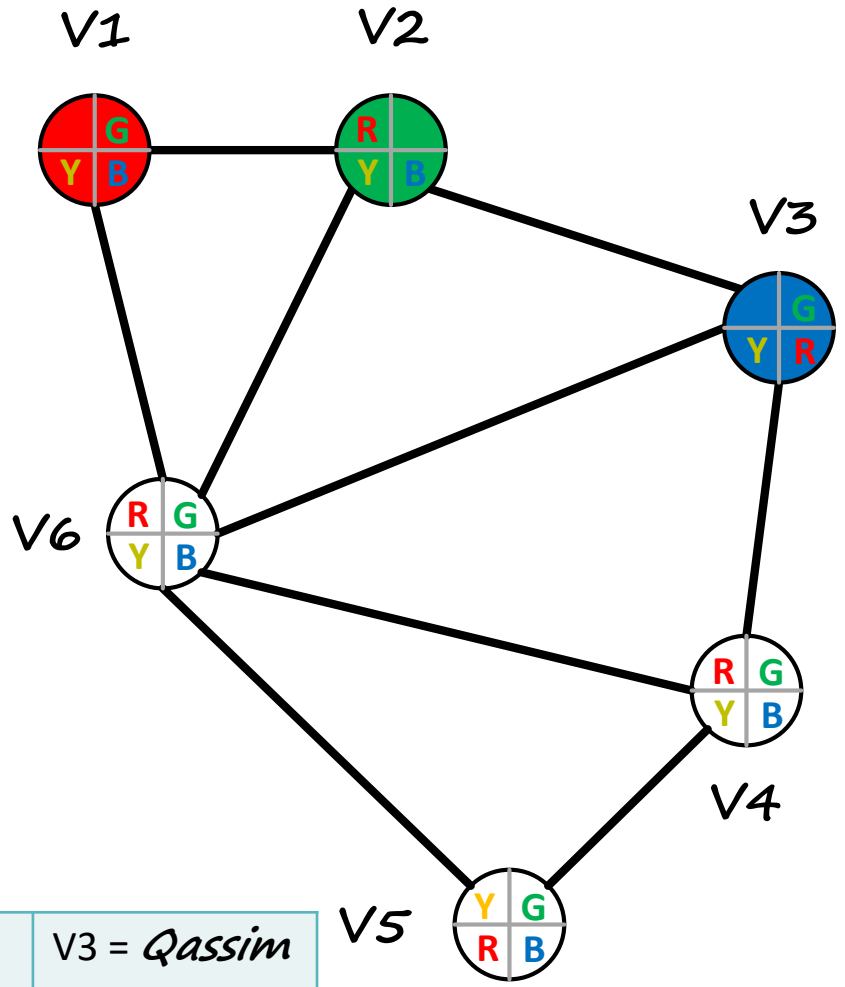
- Step 4: Select next unassigned variable in order
  - V2 = *Ha'il*
- Step 5: Color the selected variable (region) with the first value in the domain; **R**
- Step 6: Check constraints:
  - V1 = **R** and V2= **R**, constraint is not met ➔ backtrack
  - Assign next color in the domain **G**
  - all constraints are met



V1 = *Tabuk*	V2 = *Ha'il*	V3 = *Qassim*
V4 = *Riyadh*	V5 = *Makkah*	V6 = *Medina*

CSP Examples 2: Map-Coloring – Backtracking

- Step 7: Select next unassigned variable in order
  - V3 = *Qassim*
- Step 8: Color the selected variable (region) with the first value in the domain; **B**
- Step 9: Check constraints:
  - All constraints are OK

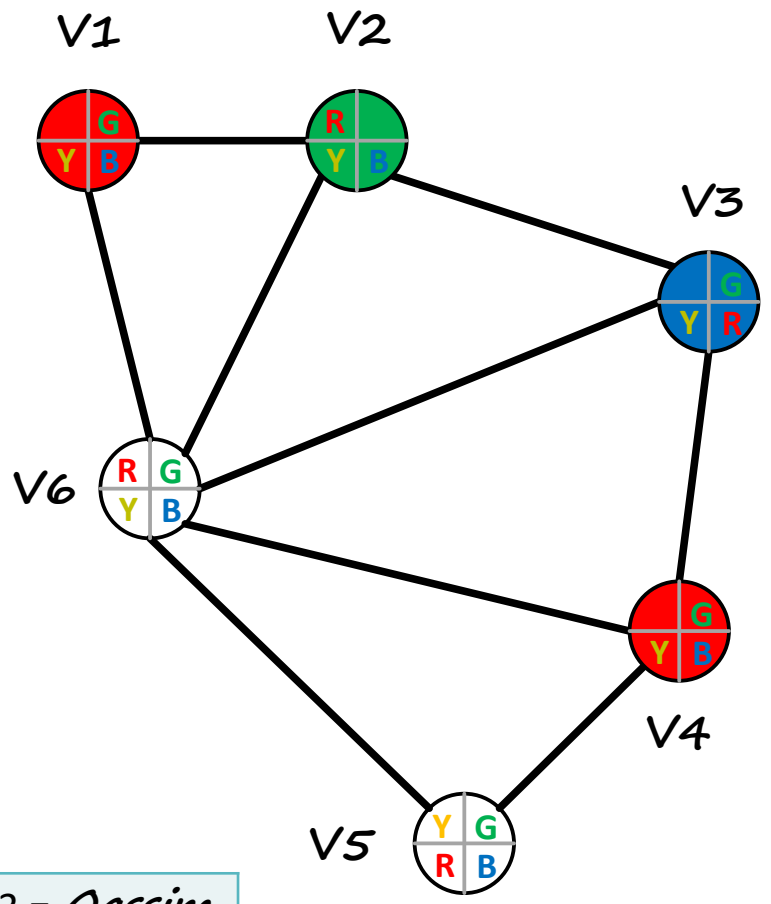


V1 = *Tabuk*	V2 = *Ha'il*	V3 = *Qassim*
V4 = *Riyadh*	V5 = *Makkah*	V6 = *Medina*

Step 3

CSP Examples 2: Map-Coloring – Backtracking

- Step 10: Select next unassigned variable in order
  - V4 = *Riyadh*
- Step 11: Color the selected variable (region) with the first value in the domain; **R**
- Step 12: Check constraints:
  - All constraints are OK

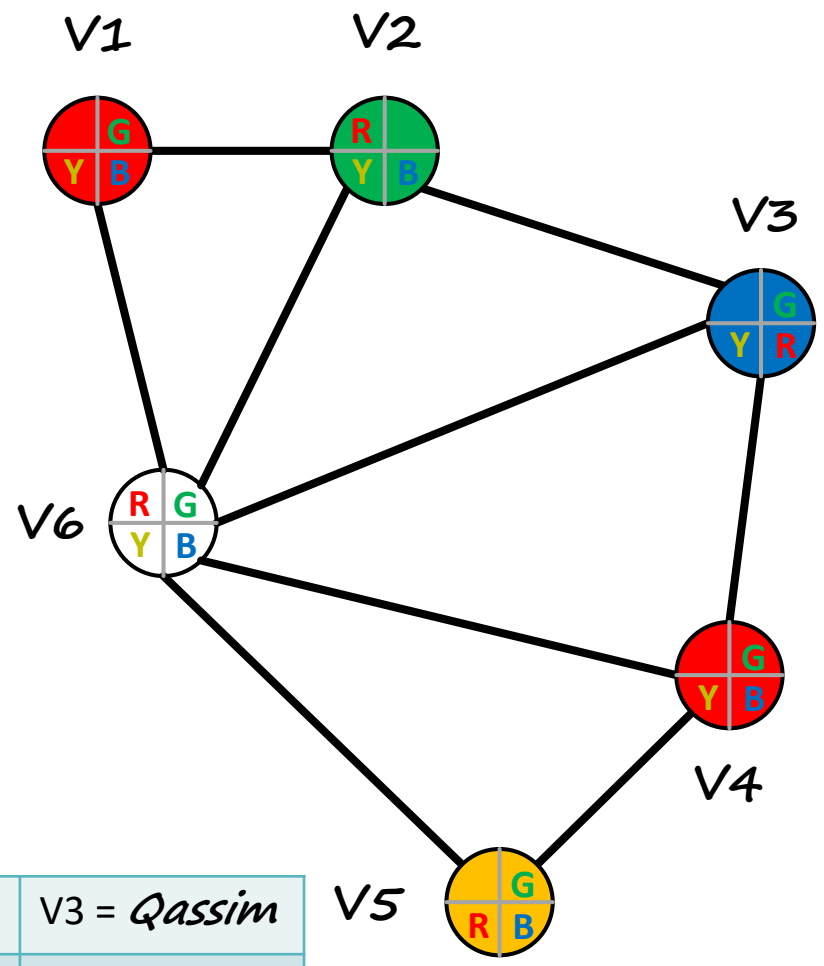


V1 = *Tabuk*	V2 = *Ha'il*	V3 = *Qassim*
V4 = *Riyadh*	V5 = *Makkah*	V6 = *Medina*

Step 4

CSP Examples 2: Map-Coloring – Backtracking

- Step 13: Select next unassigned variable in order
  - V5 = *Makkah*
- Step 14: Color the selected variable (region) with the first value in the domain; Y
- Step 15: Check constraints:
  - All constraints are OK

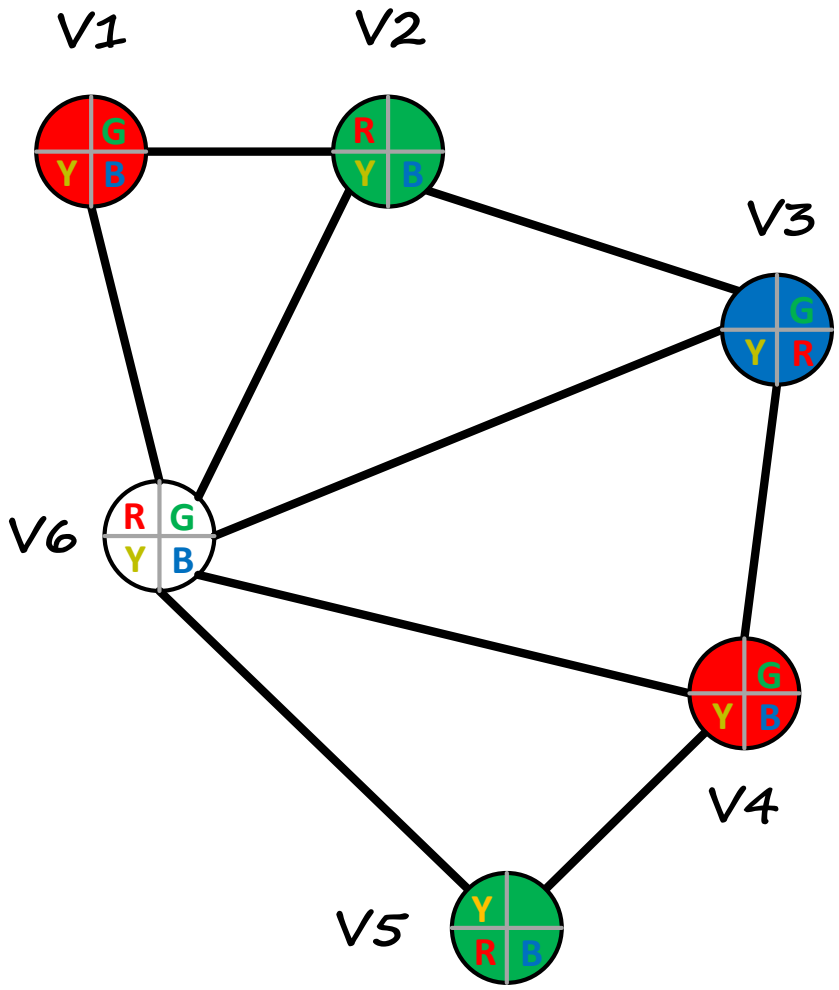


V1 = *Tabuk*	V2 = *Ha'il*	V3 = *Qassim*
V4 = *Riyadh*	V5 = *Makkah*	V6 = *Medina*

Step 5

CSP Examples 2: Map-Coloring – Backtracking

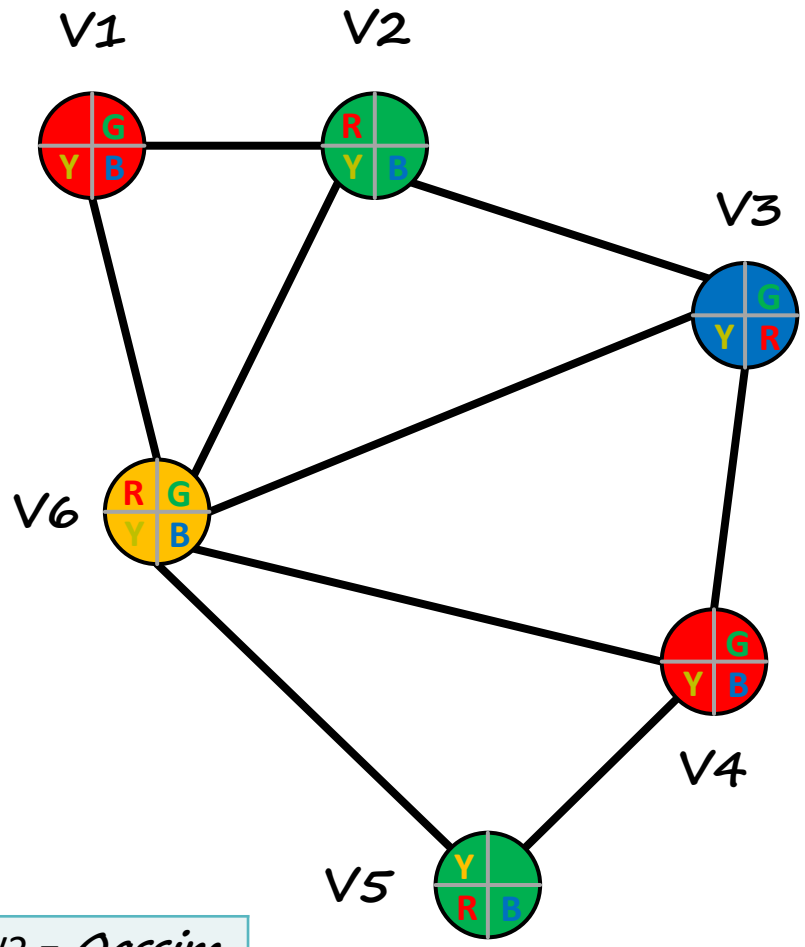
- Step 16: Select next unassigned variable in order
  - V6 = *Madina*
- Step 17: Color the selected variable (region) with the first value in the domain; **R**
- Step 18: Check constraints:
  - V6 = **R** constraint is not met ➔ backtrack
  - V6 = **G** constraint is not met ➔ backtrack
  - V6 = **Y** constraint is not met ➔ backtrack
  - V6 = **B** constraint is not met ➔ backtrack
  - ***V5*** = **G** all constraints are met



Step 6

CSP Examples 2: Map-Coloring – Backtracking

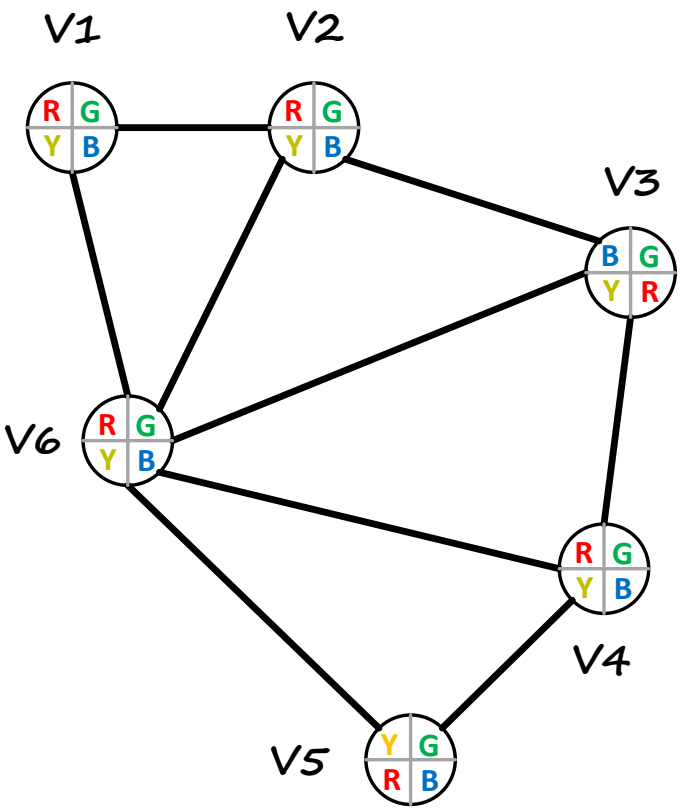
- Step 19: Color V6:
  - V6 = **R** constraint is not met ➔ backtrack
  - V6 = **G** constraint is not met ➔ backtrack
  - V6 = **Y** constraint is met
- Step 20: All variables are assigned ➔ End



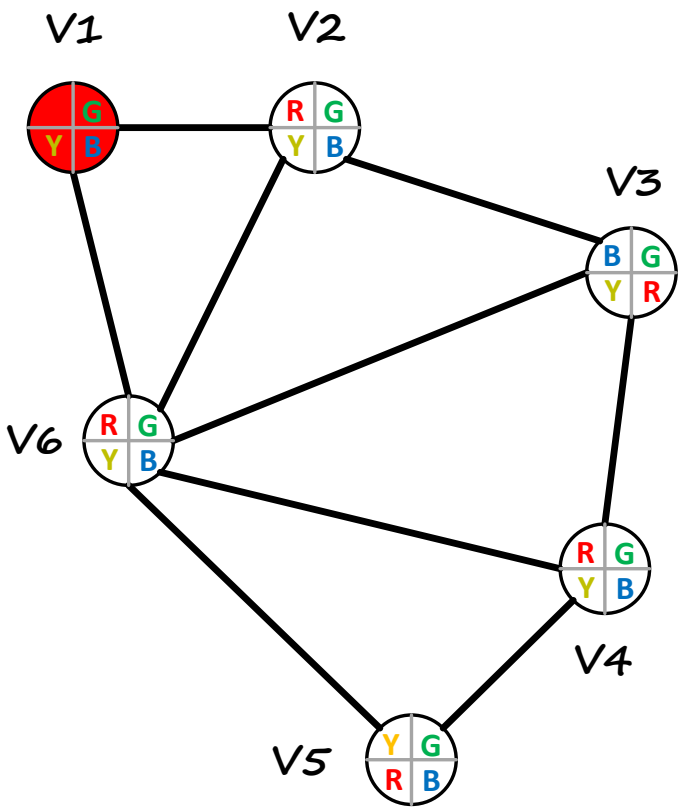
V1 = *Tabuk*	V2 = *Ha'il*	V3 = *Qassim*
V4 = *Riyadh*	V5 = *Makkah*	V6 = *Medina*

Step 7

CSP Example 2: Backtracking Algorithm Summary

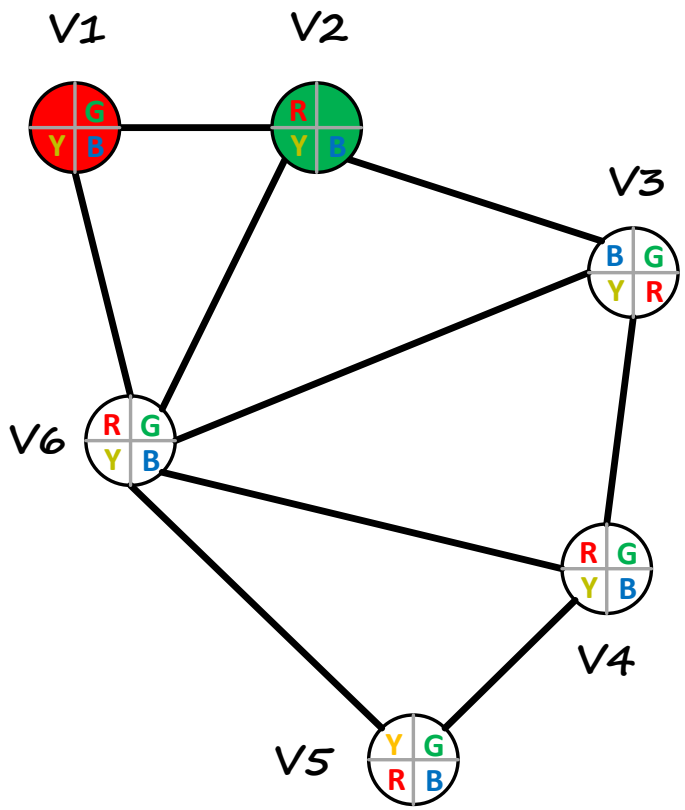


Step 0



Step 1

Select first color
in Domain, V1 = **R**

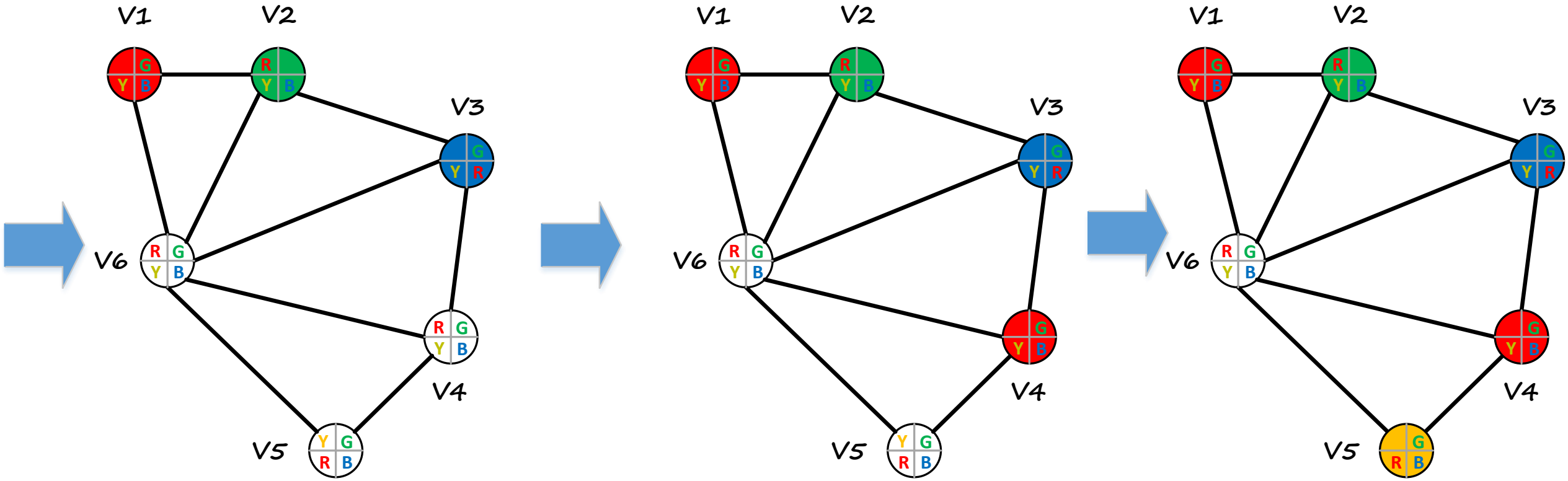


Step 2

Backtrack V2 once
to Select V2 = **G**

Note: Backtracking steps are not shown and are embedded into each step

CSP Example 2: Backtracking Algorithm Summary



Step 3

Select first color in Domain, V3 = **B**

Step 4

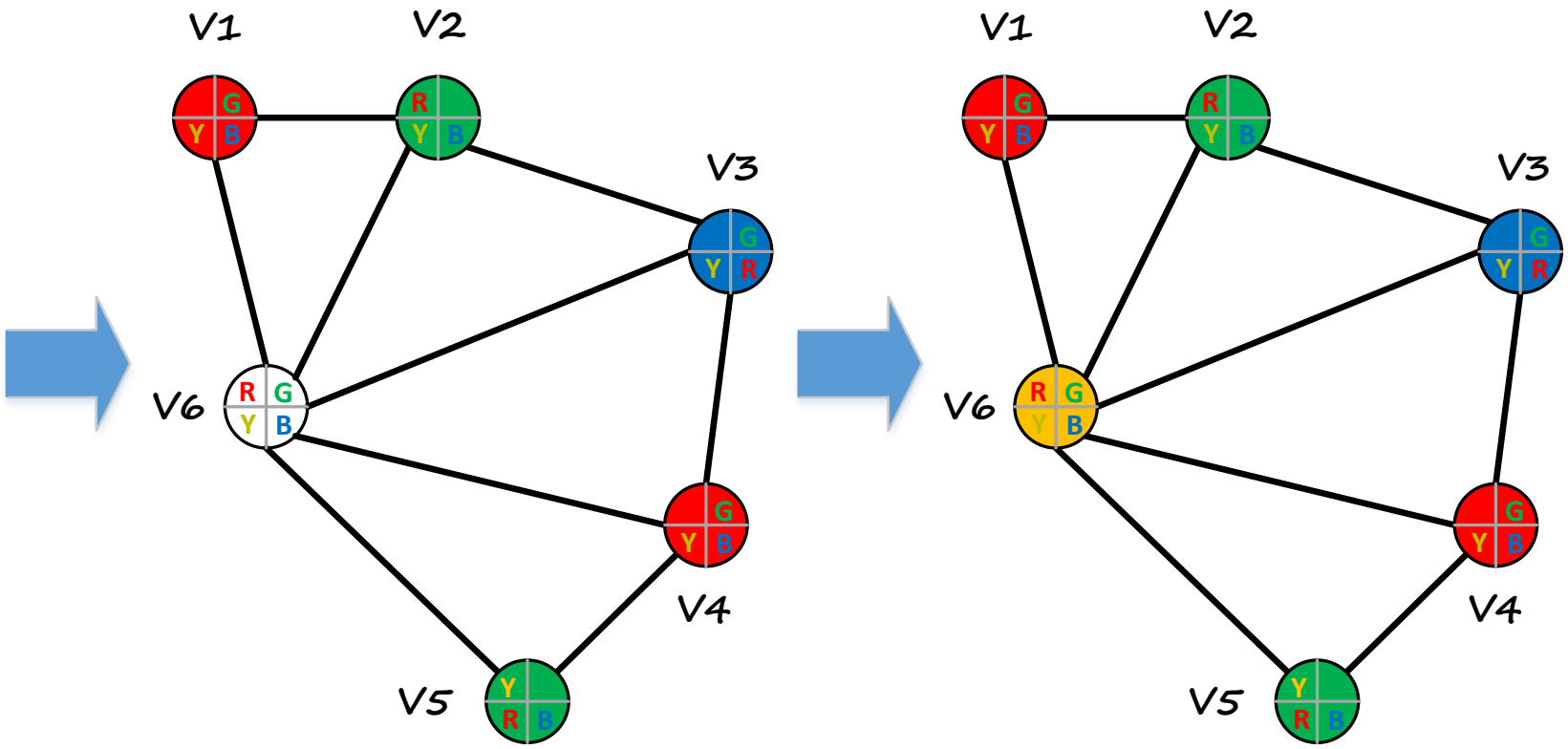
Select first color in Domain, V4 = **R**

Step 5

Select first color in Domain, V5 = **Y**

Note: Backtracking steps are not shown and are embedded into each step

CSP Example 2: Backtracking Algorithm Summary



Step 6

Backtrack 4 times for V6 then backtrack once for V5 and select V5 = **G**

Step 7

Backtrack 2 times for V6 and select V6 = **Y**

Note: Backtracking steps are not shown and are embedded into each step

Problems with Backtracking Search

- Keeps repeating the same failed variable assignments
- Does not detect inevitable failures earlier in the search
- May explore areas of the search space that aren’t likely to succeed

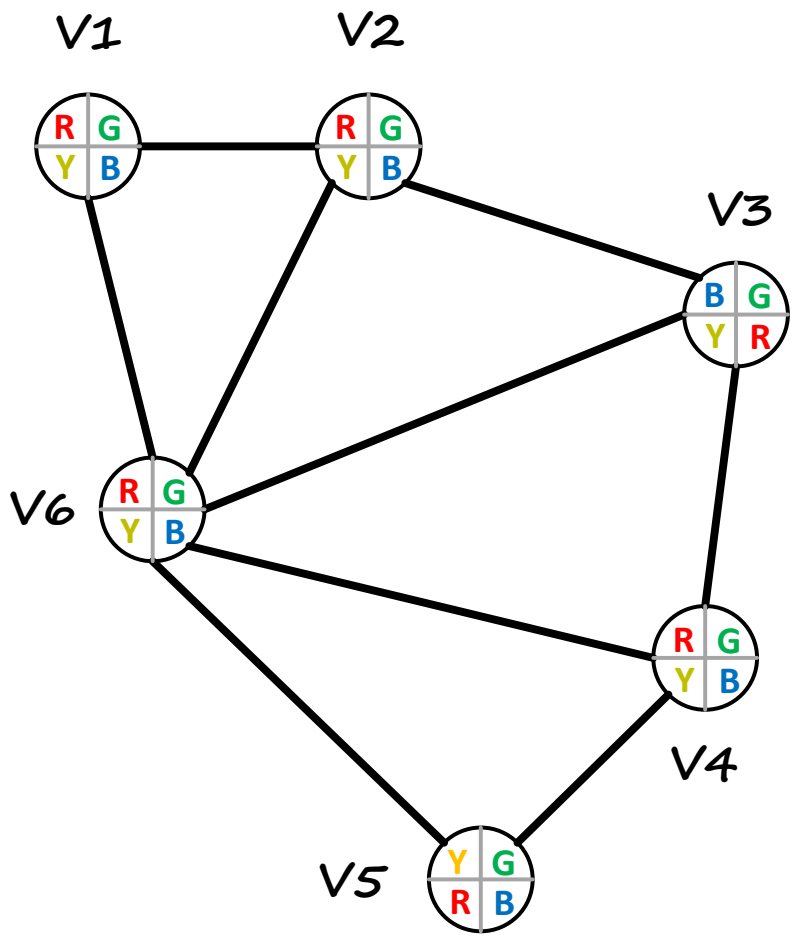
Improving Backtracking Search

- General-purpose ideas give huge gains in speed

1. Forward Checking: Can we detect an inevitable failure early?
2. Ordering:
   - Which variable should be assigned next?
   - In what order should its values be tried?
3. Consistency: Prune the domains as much as possible before selecting values for variables

Backtracking - Forward Checking

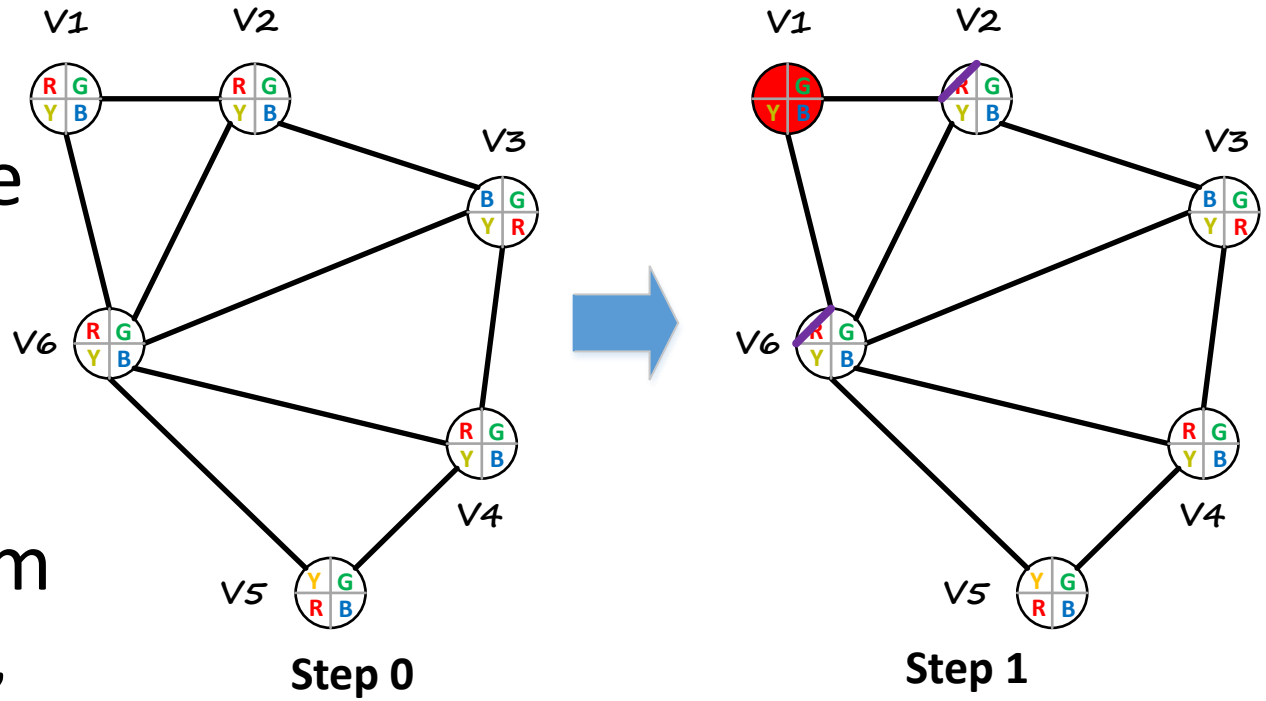
- Forward Checking propagates information from assigned to unassigned variables and crosses off values that violate a constraint when added to the existing assignment
  - In the map coloring example, the domain of each adjacent region that was not colored will lose the ability of choosing the same color to keep up with the constraint. For example, once *Tabuk* is assigned **R** both *Ha'il* and *Madina* will loose that color from their domains



V1 = *Tabuk*	V2 = *Ha'il*	V3 = *Qassim*
V4 = *Riyadh*	V5 = *Makkah*	V6 = *Medina*

CSP Example 2: Map Coloring - Forward Checking

- Step 1:
  - Select the first variable from the variable set, V1 = *Tabuk*, and assign it the first value in its domain; **R**
  - Cross off the value (color) **R** from domains of all adjacent regions, i.e., V2 = *Ha'il* and V6 = *Medina*



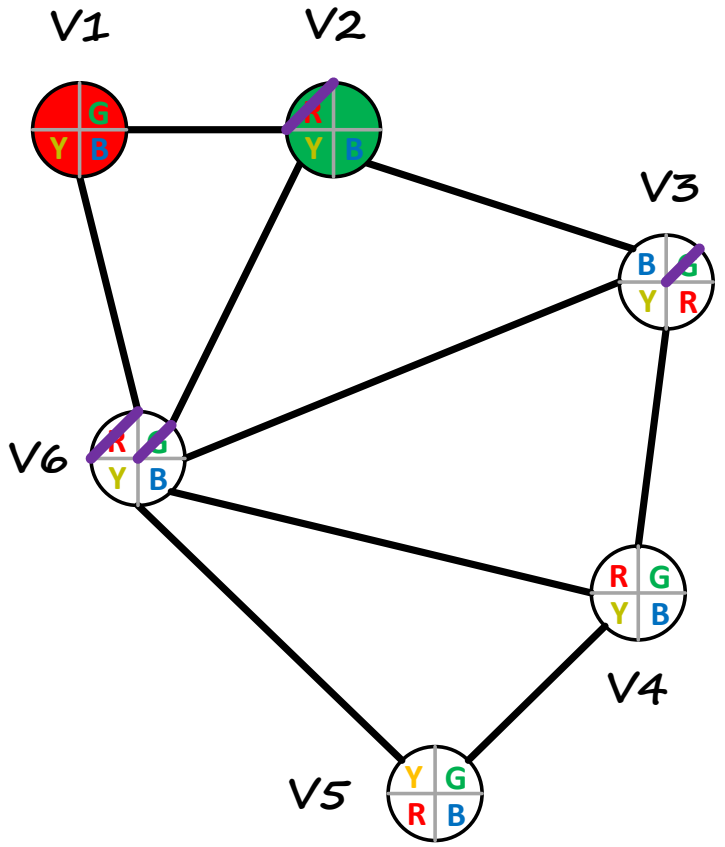
Select first color in Domain,
V1 = **R**
Note the domain reduction of V2 and V6

V1 = *Tabuk*	V2 = *Ha'il*	V3 = *Qassim*
V4 = *Riyadh*	V5 = *Makkah*	V6 = *Medina*

D1= {**R**, **G** , **Y** , **B**}	D2= {**R**, **G** , **Y** , **B**}	D3= {**B**, **G**, **Y**, **R**}
D4= {**R**, **G** , **Y** , **B**}	D5= {**Y** , **G**, **R**, **B**}	D6= {**R**, **G** , **Y** , **B**}

CSP Example 2: Map Coloring - Forward Checking

- Step 2:
  - Select the next unassigned variable from the variable set, V2 = *Ha'il*, and assign it the next available value in its domain; **G**
  - Cross off the value (color) **G** from domains of all adjacent unassigned variables, i.e., V3 and V6



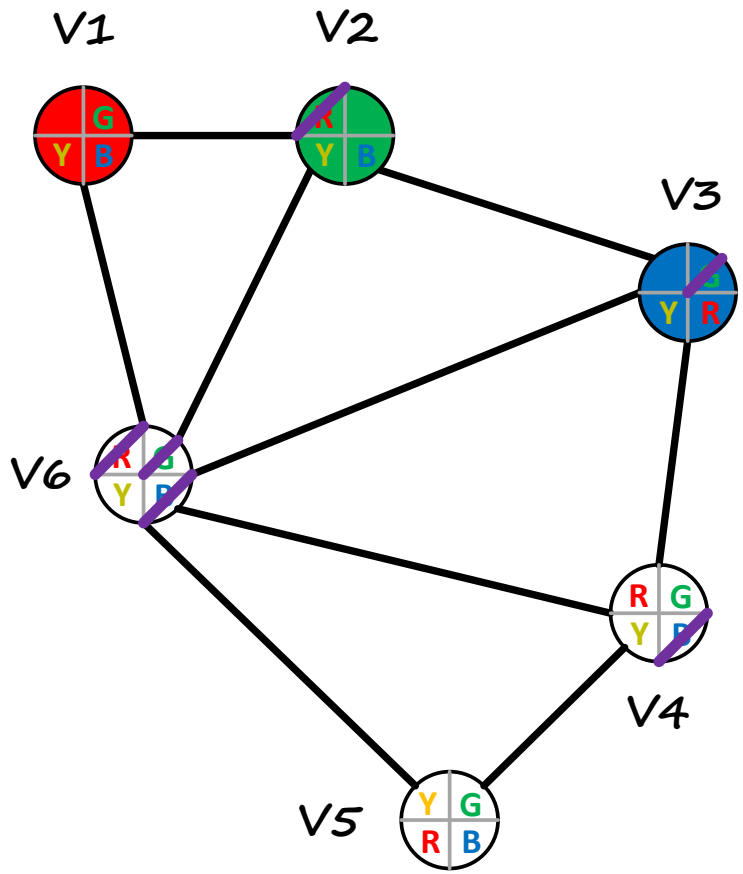
Step 2

Note the domain reduction of V3 and V6

V1 = *Tabuk*	V2 = *Ha'il*	V3 = *Qassim*
V4 = *Riyadh*	V5 = *Makkah*	V6 = *Medina*

CSP Example 2: Map Coloring - Forward Checking

- Step 3:
  - Select the next unassigned variable from the variable set, V3 = *Qasim*, and assign it the next available value in its domain; **B**
  - Cross off the value (color) **B** from domains of all adjacent unassigned variables, i.e., V4 and V6

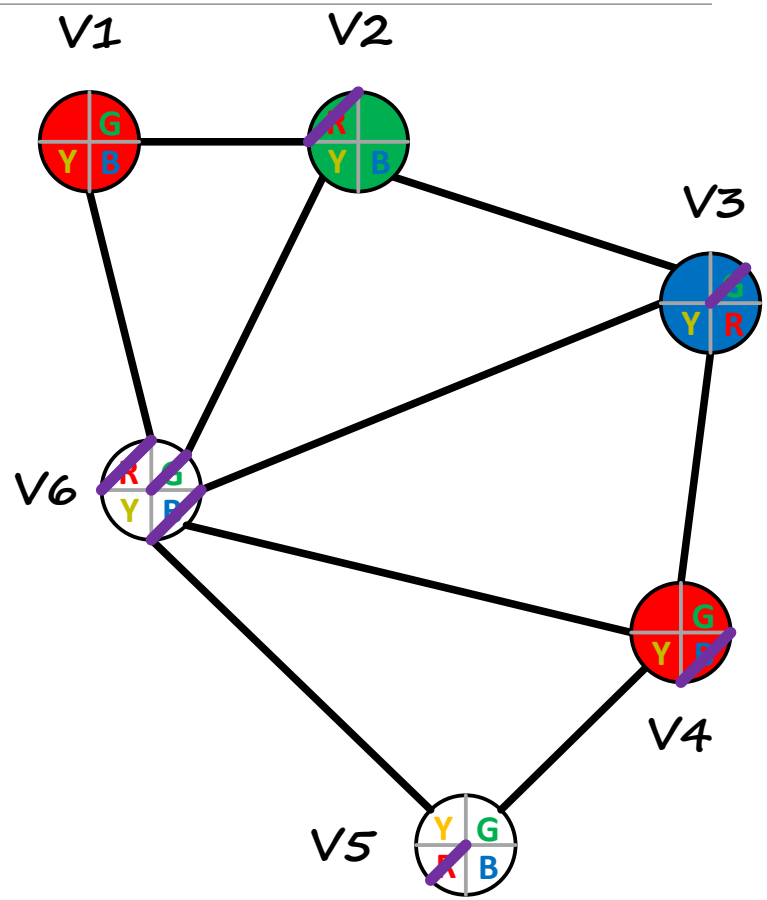


Step 3
Note the domain reduction of V4 and V6

V1 = *Tabuk*	V2 = *Ha'il*	V3 = *Qassim*
V4 = *Riyadh*	V5 = *Makkah*	V6 = *Medina*

CSP Example 2: Map Coloring - Forward Checking

- Step 4:
  - Select the next unassigned variable from the variable set, V4 = ***Riyadh***, and assign it the next available value in its domain; **R**
  - Cross off the value (color) **R** from domains of all adjacent unassigned variables, i.e., V5 and V6



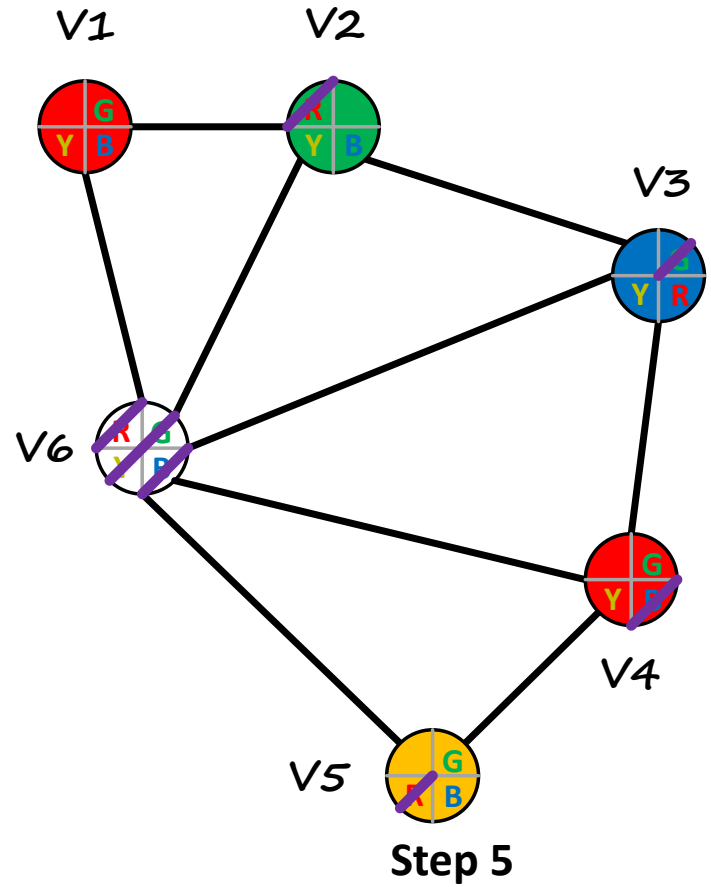
Step 4

Note the domain reduction of V5 and V6

V1 = *Tabuk*	V2 = *Ha'il*	V3 = *Qassim*
V4 = *Riyadh*	V5 = *Makkah*	V6 = *Medina*

CSP Example 2: Map Coloring - Forward Checking

- Step 5:
  - Select the next unassigned variable from the variable set, V5 = *Makkah*, and assign it the next available value in its domain; **Y**
  - Cross off the value (color) **Y** from domains of all adjacent regions, i.e., V6
  - Note that the domain of V6 becomes empty



Step 5

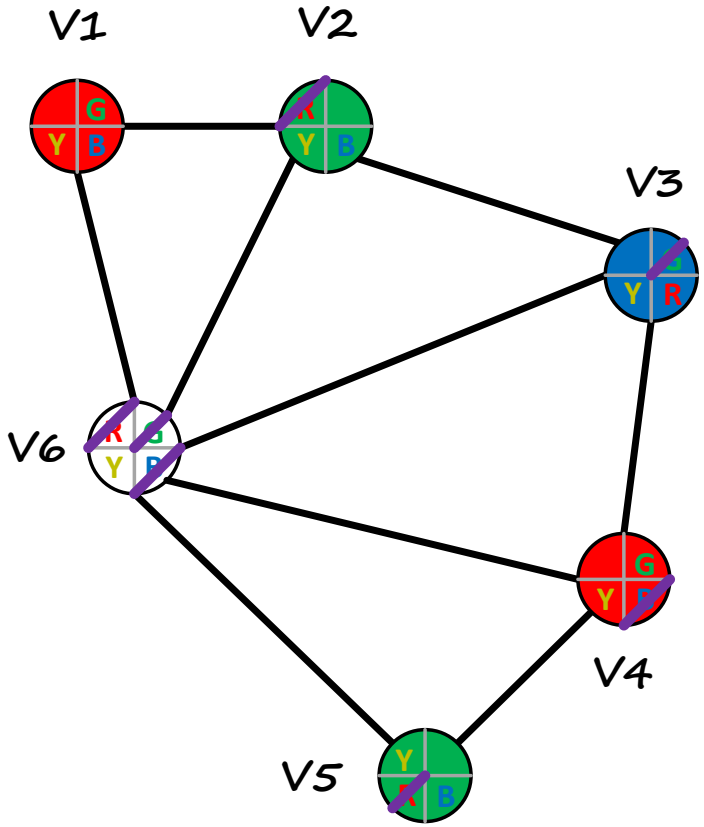
Note the domain reduction of V6

Domain of V6=ϕ

V1 = *Tabuk*	V2 = *Ha'il*	V3 = *Qassim*
V4 = *Riyadh*	V5 = *Makkah*	V6 = *Medina*

CSP Example 2: Map Coloring - Forward Checking

- Step 6:
  - Backtrack to V5 = *Makkah* and remove its color assignment
  - Choose the next available color in V5 domain; **G**.



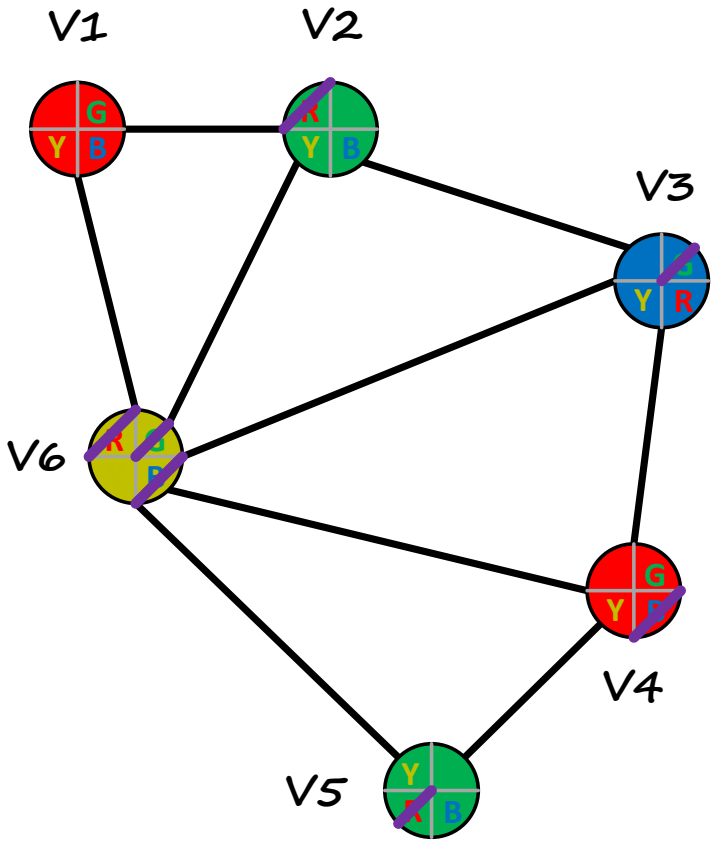
Step 6

Note the domain of V6

V1 = *Tabuk*	V2 = *Ha'il*	V3 = *Qassim*
V4 = *Riyadh*	V5 = *Makkah*	V6 = *Medina*

CSP Example 2: Map Coloring - Forward Checking

- Step 7:
  - Color V6 with available color (V6=Y)
  - Since all variables are assigned ➔ End

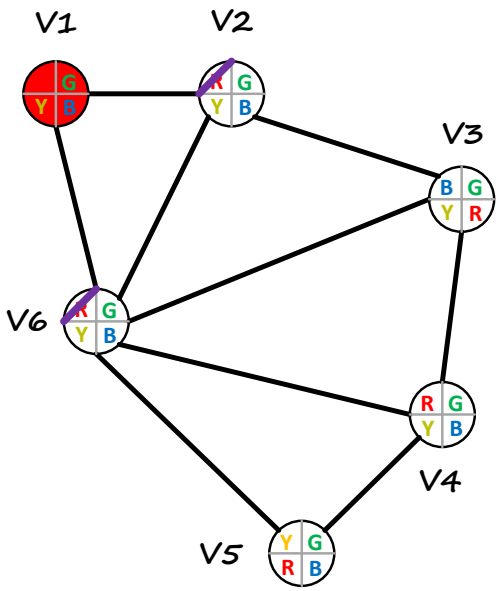


Step 7

Note the domain of V5

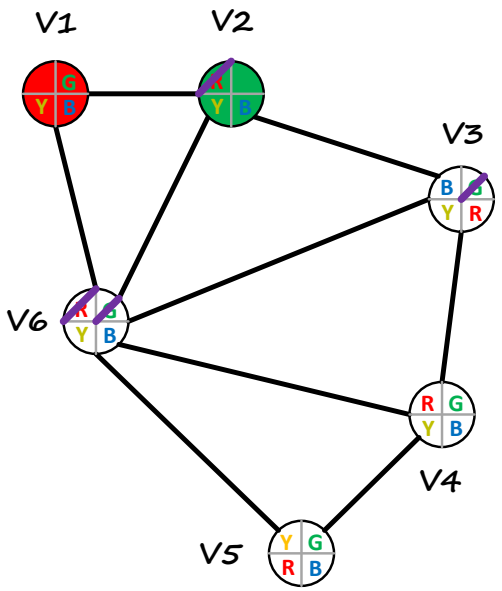
V1 = *Tabuk*	V2 = *Ha'il*	V3 = *Qassim*
V4 = *Riyadh*	V5 = *Makkah*	V6 = *Medina*

CSP Example 2: Map Coloring - Forward Checking - Summary



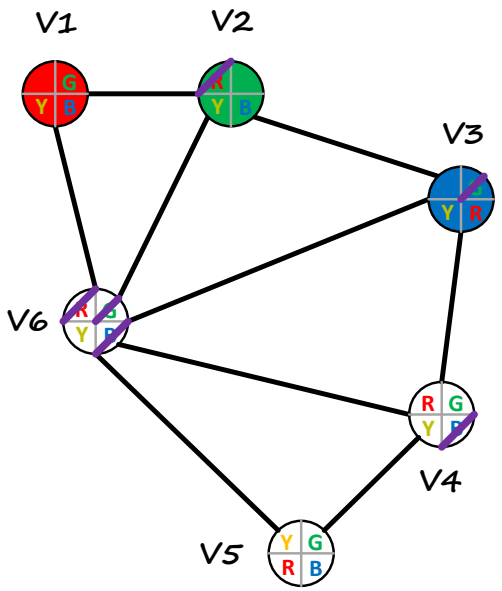
Step 1

Note the domain reduction of V2 and V6



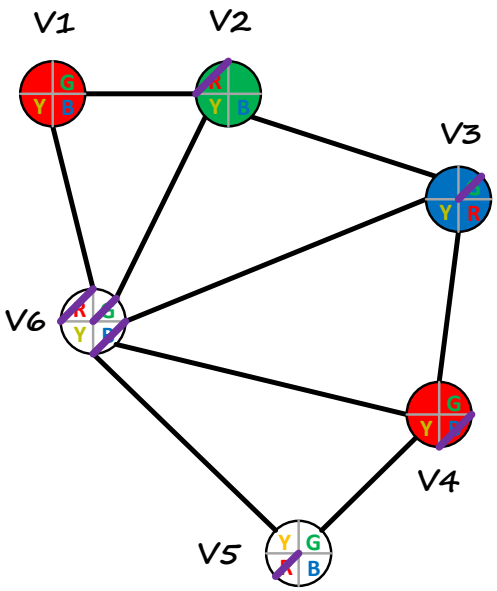
Step 2

Note the domain reduction of V3 and V6



Step 3

Note the domain reduction of V4 and V6

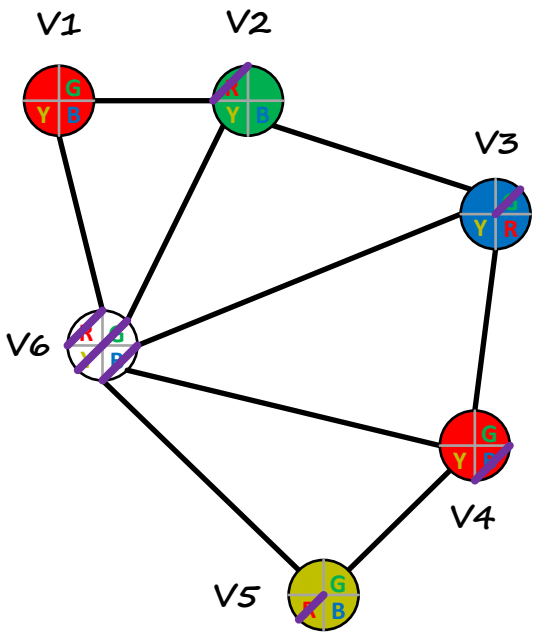


Step 4

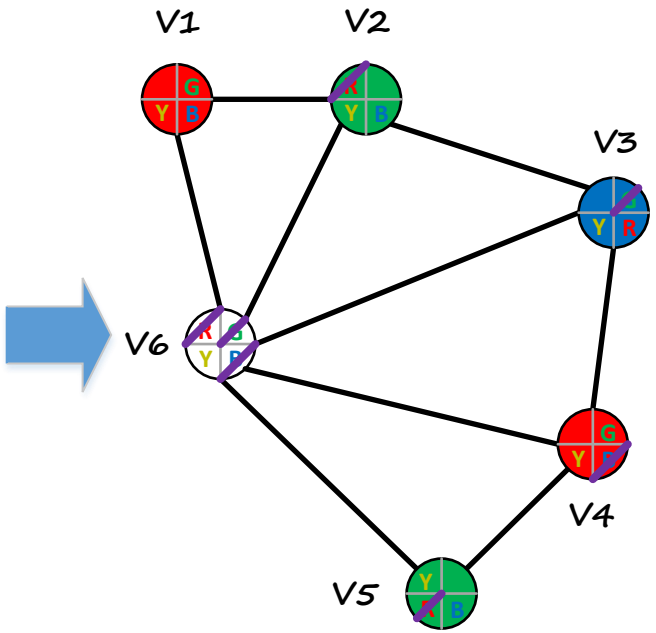
Note the domain reduction of V5

V1 = *Tabuk*	V2 = *Ha'il*	V3 = *Qassim*
V4 = *Riyadh*	V5 = *Makkah*	V6 = *Medina*

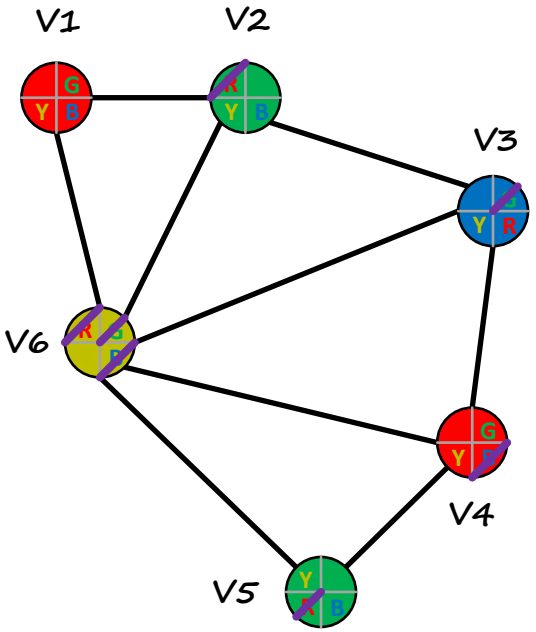
CSP Example 2: Map Coloring - Forward Checking - Summary



Step 5
Note the domain reduction of V6



Step 6
Backtracking and select next color in V5



Step 7
END

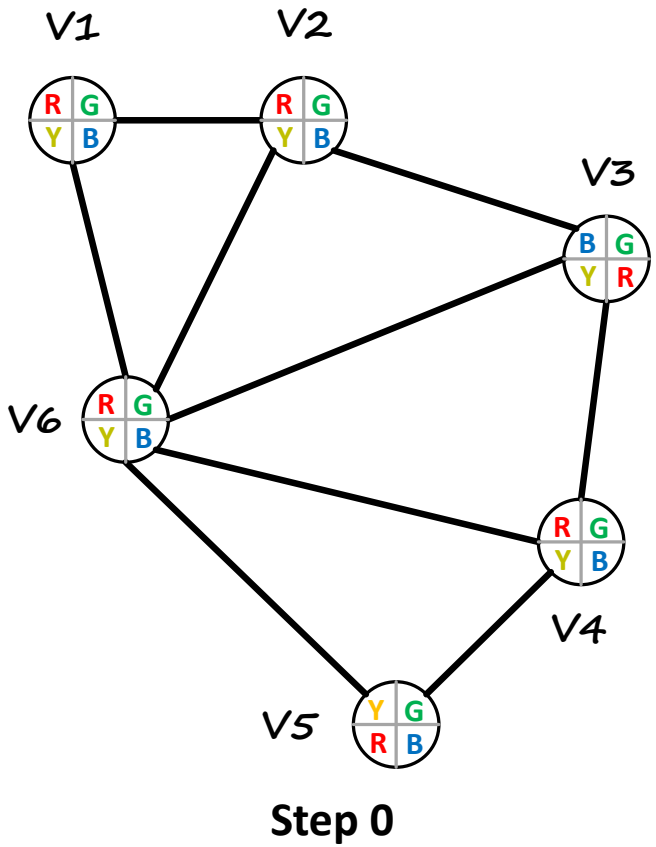
V1 = *Tabuk*	V2 = *Ha'il*	V3 = *Qassim*
V4 = *Riyadh*	V5 = *Makkah*	V6 = *Medina*

Backtracking with Minimum Remaining Values (MRV)

- Minimum Remaining Values (MRV) is a popular strategy used along with Backtracking algorithm
- Backtracking with MRV follows an ordering strategy that augments backtracking search
- The order of selection of variables follows the following rules:
  - Choose the variable with the least values in its domain
- MRV is also called "Most Constrained/Restricted Variable"
- Also called "Fail-Fast" ordering

CSP Examples 2: Map Coloring - MRV

- Follow the same steps as forward checking However, in MRV, the variable that is selected next is the variable which has the minimum number of values in its domain
  - If a tie occurs, lexically order variables then pick the first
  - MRV is implemented along with filtering to remove the assigned value from domains that can cause a conflict with this latest assignment



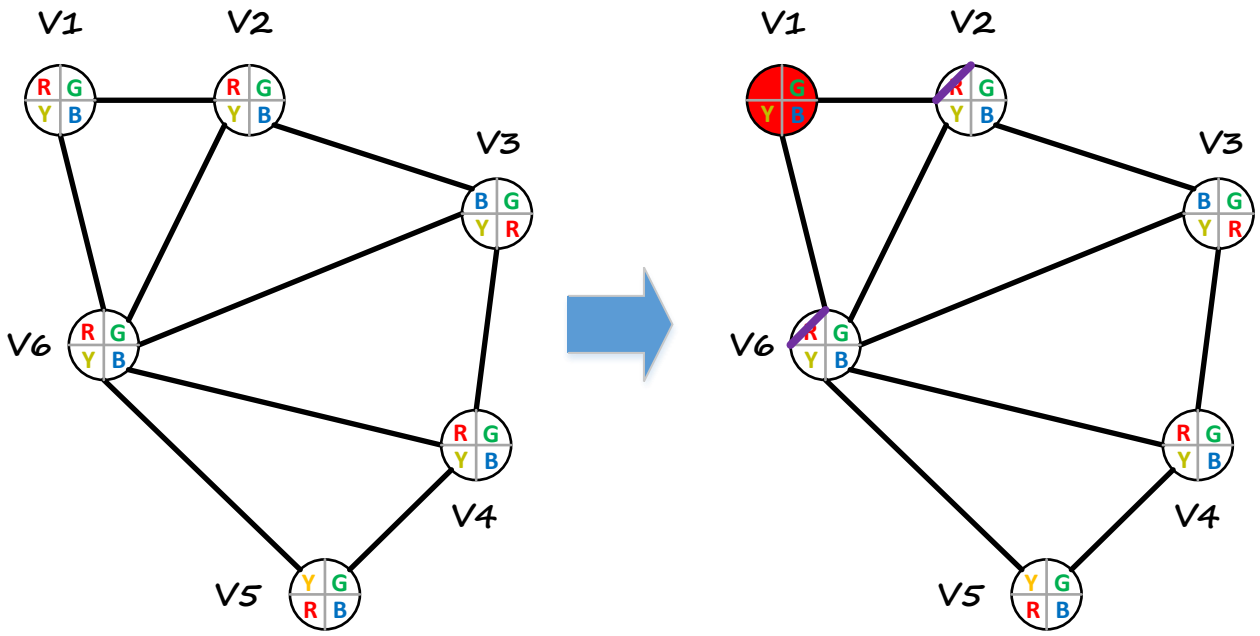
Step 0

V1 = *Tabuk*	V2 = *Ha'il*	V3 = *Qassim*
V4 = *Riyadh*	V5 = *Makkah*	V6 = *Medina*

D1= {R, G , Y , B}	D2= {R, G , Y , B}	D3= {B, G, Y, R}
D4= {R, G , Y , B}	D5= {Y , G, R, B}	D6= {R, G , Y , B}

CSP Examples 2: Map Coloring - MRV

- Step 1: Since all variables have equal number of values in their domains, pick the first one, i.e., V1= *Tabuk*
  - Color V1 = **R**; the first available color in its domain
  - Propagate the constraint to all neighbors, i.e. cross off the color **R** from their domains



Step 1

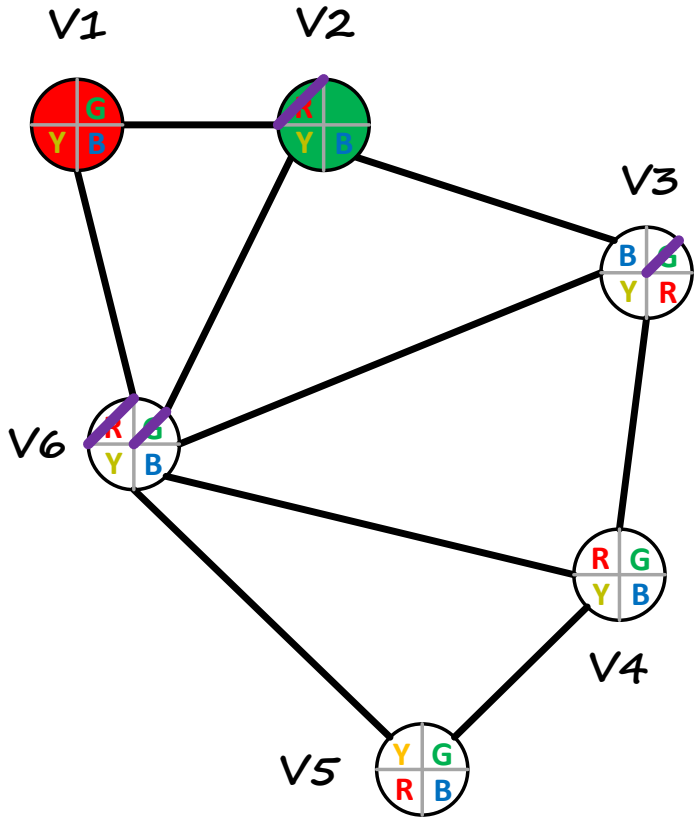
Note the domain reduction of V2 and V6

MRV = V2, V6

V1 = *Tabuk*	V2 = *Ha'il*	V3 = *Qassim*
V4 = *Riyadh*	V5 = *Makkah*	V6 = *Medina*

CSP Examples 2: Map Coloring - MRV

- Step 2: Since V2 and V6 have equal number of values in their domains, pick V2= *Ha'il*
  - Color it with the first available color in its domain; **G**
  - Propagate the constraint to all neighbors of unassigned variables

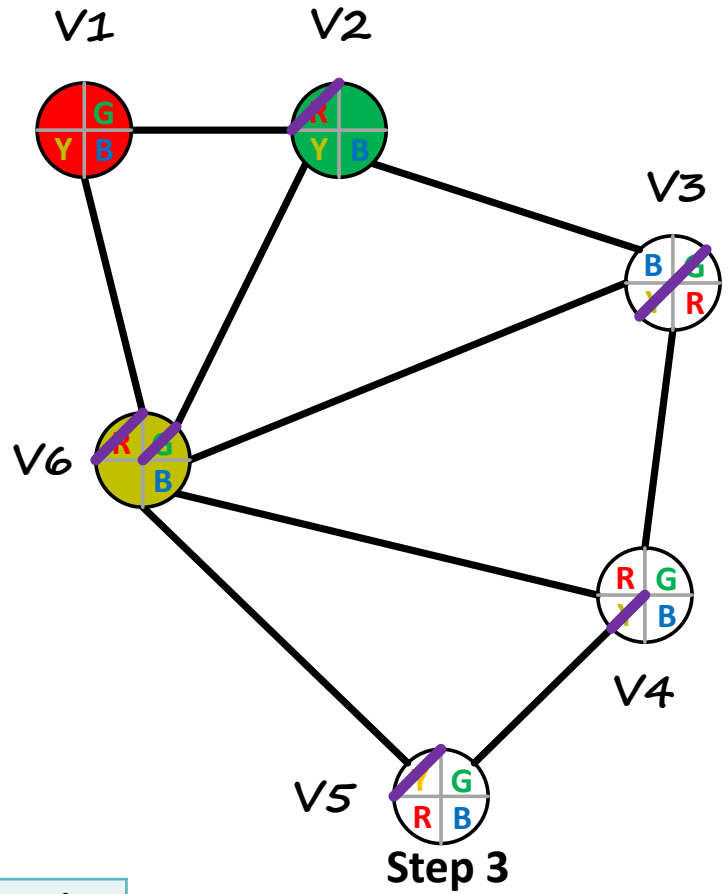


Step 2
Note the domain reduction of V3 and V6
MRV = V6

V1 = *Tabuk*	V2 = *Ha'il*	V3 = *Qassim*
V4 = *Riyadh*	V5 = *Makkah*	V6 = *Medina*

CSP Examples 2: Map Coloring - MRV

- Step 3: Since V6 has the least number of values in its domain, pick V6
  - Color it with the first available color in its domain; **Y**
  - Propagate the constraint to all neighbors of unassigned variables



Step 3

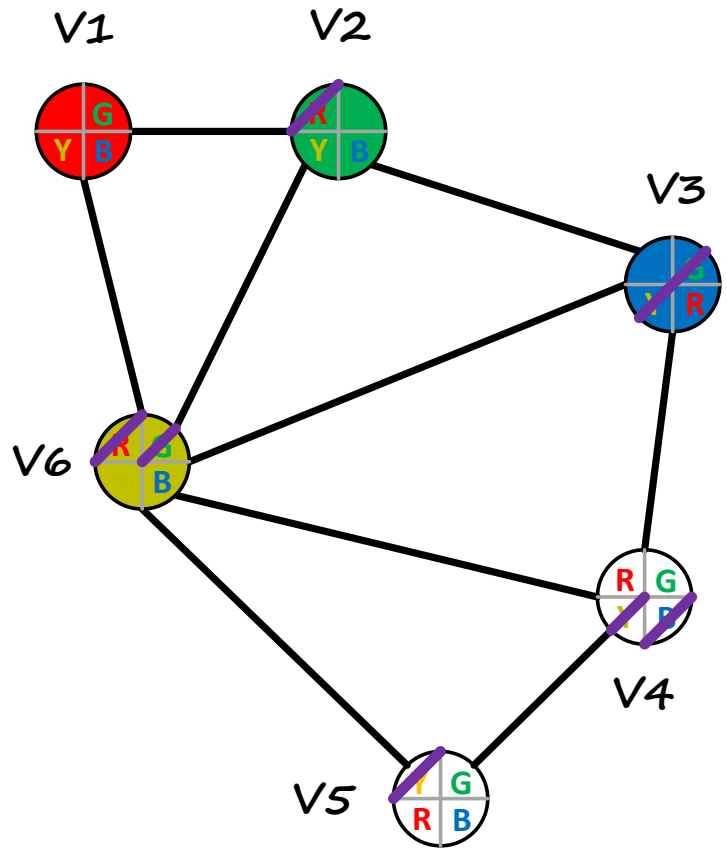
V1 = *Tabuk*	V2 = *Ha'il*	V3 = *Qassim*
V4 = *Riyadh*	V5 = *Makkah*	V6 = *Medina*

Note the domain reduction of V3, V4 and V5

MRV = V3

CSP Examples 2: Map Coloring - MRV

- Step 4: Since V3 has the least number of values in its domain, pick V3
  - Color it with the first available color in its domain; **B**
  - Propagate the constraint to all neighbors of unassigned variables



Step 4

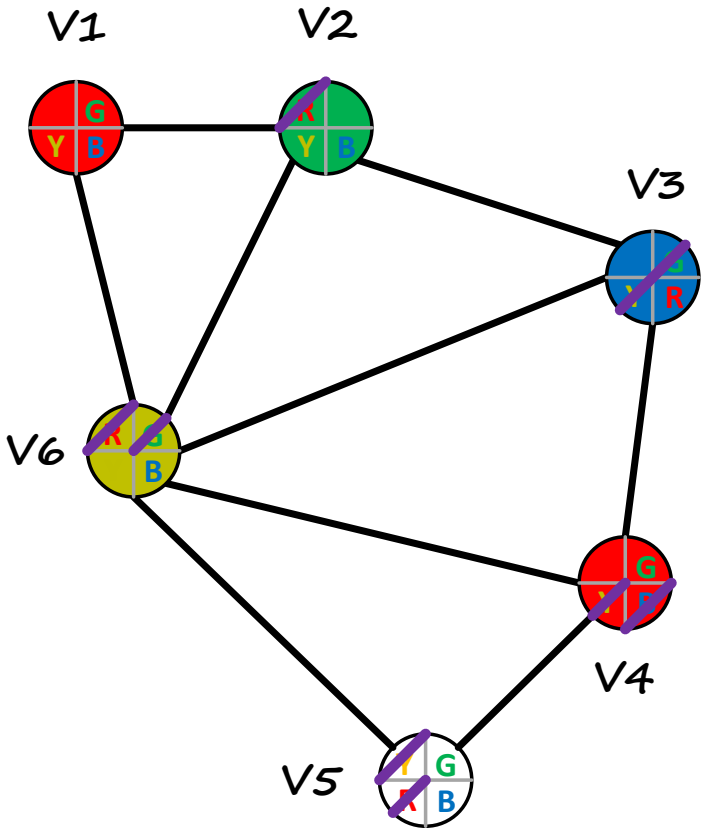
Note the domain reduction of V4

MRV = V4

V1 = *Tabuk*	V2 = *Ha'il*	V3 = *Qassim*
V4 = *Riyadh*	V5 = *Makkah*	V6 = *Medina*

CSP Examples 2: Map Coloring - MRV

- Step 5: Since V4 has the least number of values in its domain, pick V4
  - Color it with the first available color in its domain; **R**
  - Propagate the constraint to all neighbors of unassigned variables



Step 5

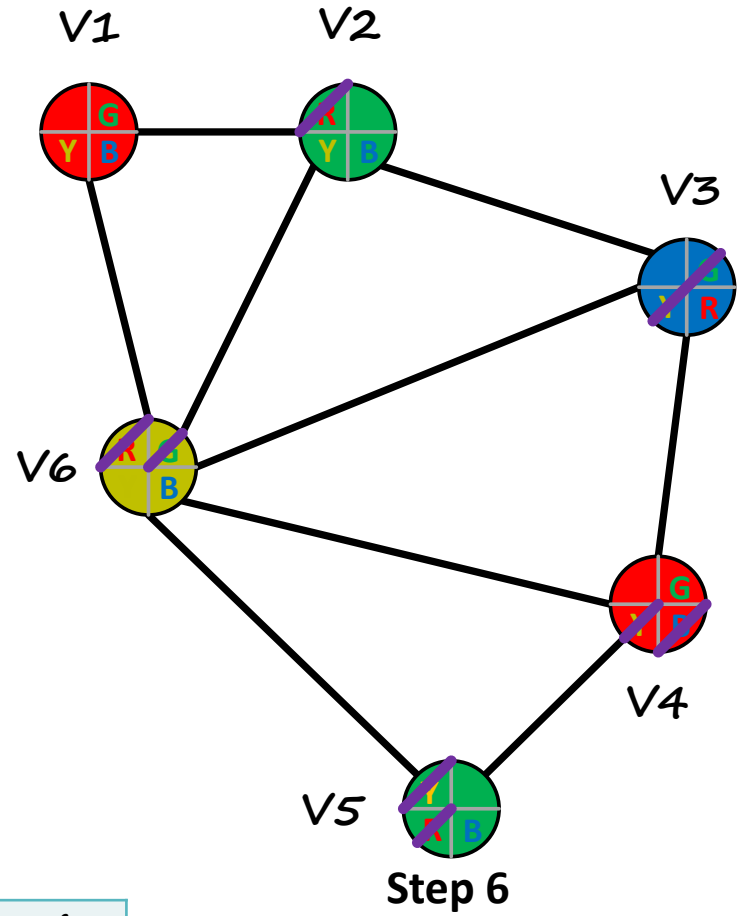
Note the domain reduction of V5

MRV = V5

V1 = *Tabuk*	V2 = *Ha'il*	V3 = *Qassim*
V4 = *Riyadh*	V5 = *Makkah*	V6 = *Medina*

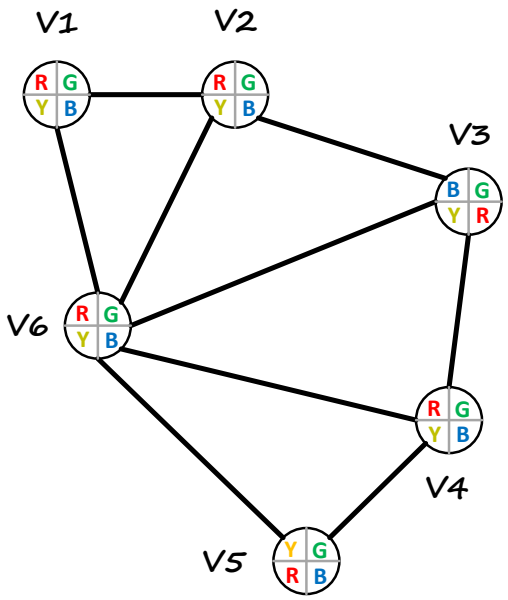
CSP Examples 2: Map Coloring - MRV

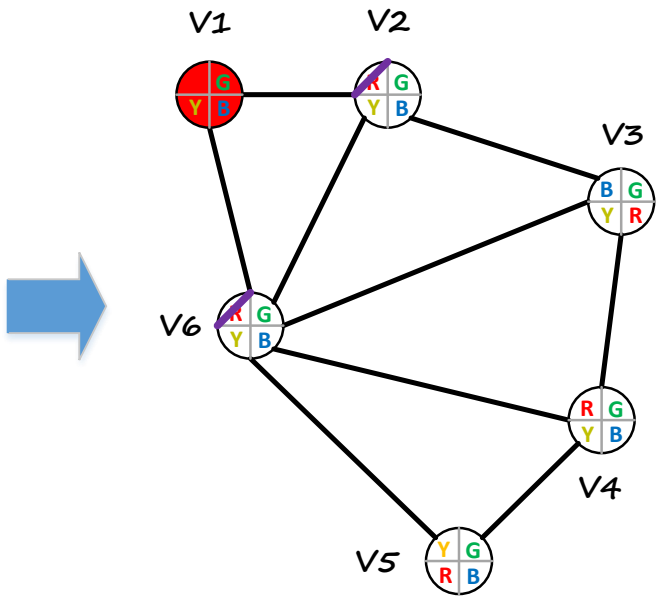
- Step 6: Since V5 has the least number of values in its domain, pick V5
  - Color it with the first available color in its domain; **G**
  - Propagate the constraint to all neighbors of unassigned variables
  - Since all variables are assigned ➔ End



V1 = *Tabuk*	V2 = *Ha'il*	V3 = *Qassim*
V4 = *Riyadh*	V5 = *Makkah*	V6 = *Medina*

CSP Examples 2: Map Coloring - MRV



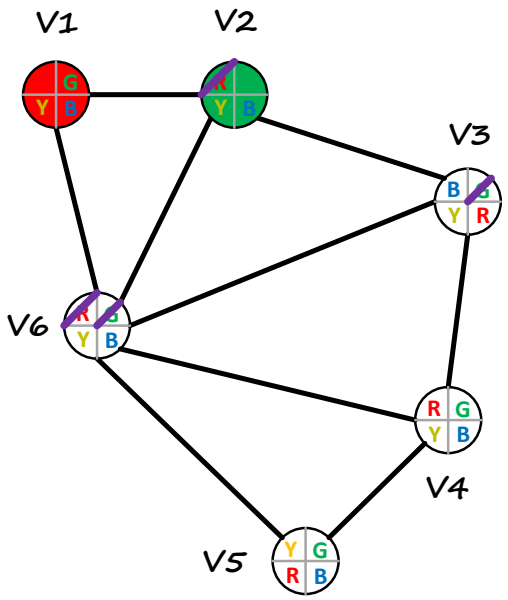


Step 1

Note the domain reduction of V2 and V6

MRV = V2, V6

[**Select V2** for next assignment]

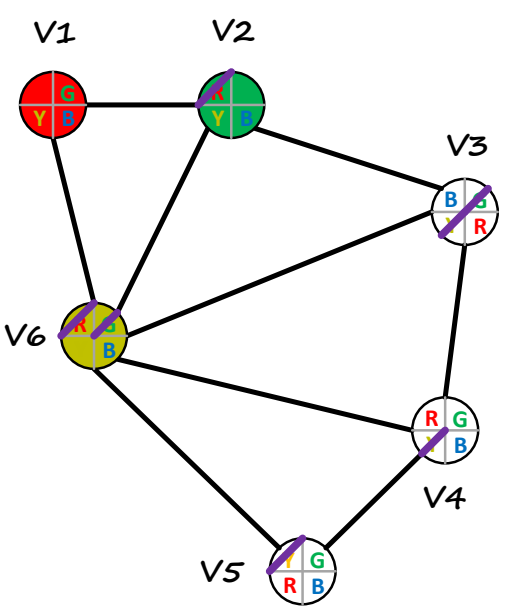


Step 2

Note the domain reduction of V3 and V6

MRV = V6

[**Select V6** for next assignment]



Step 3

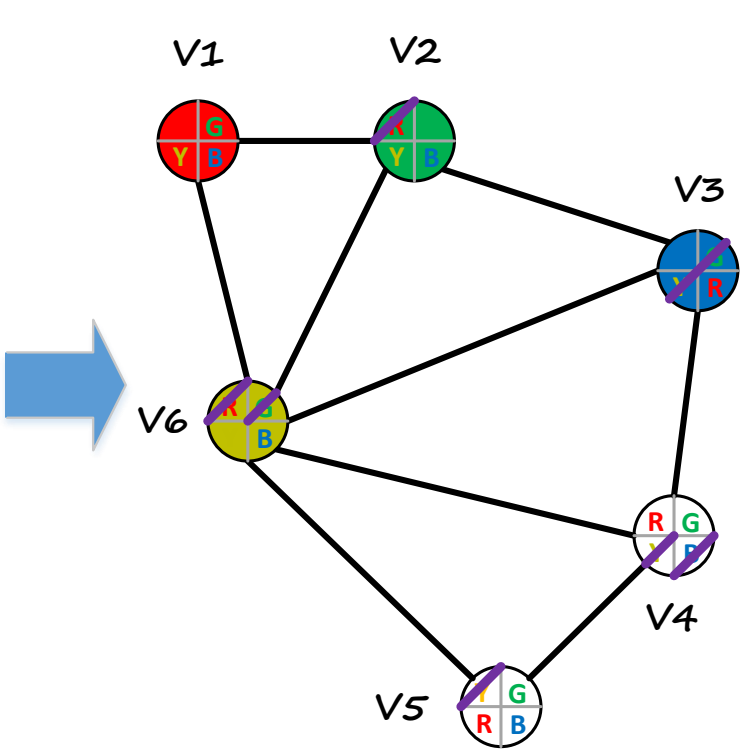
Note the domain reduction of V3, V4 and V5

MRV = V3

[**Select V3** for next assignment]

V1 = *Tabuk*	V2 = *Ha'il*	V3 = *Qassim*
V4 = *Riyadh*	V5 = *Makkah*	V6 = *Medina*

CSP Examples 2: Map Coloring - MRV

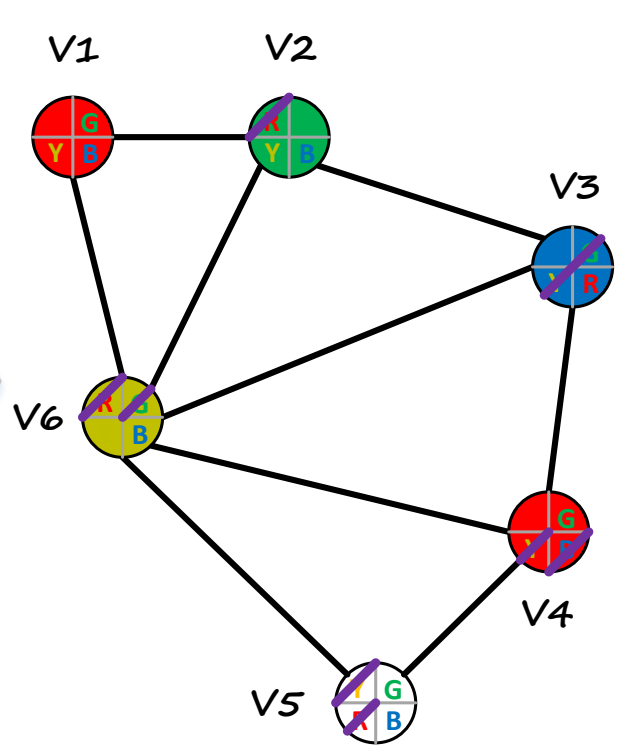


Step 4

Note the domain reduction of V4

MRV = V4

[**Select V4** for next assignment]

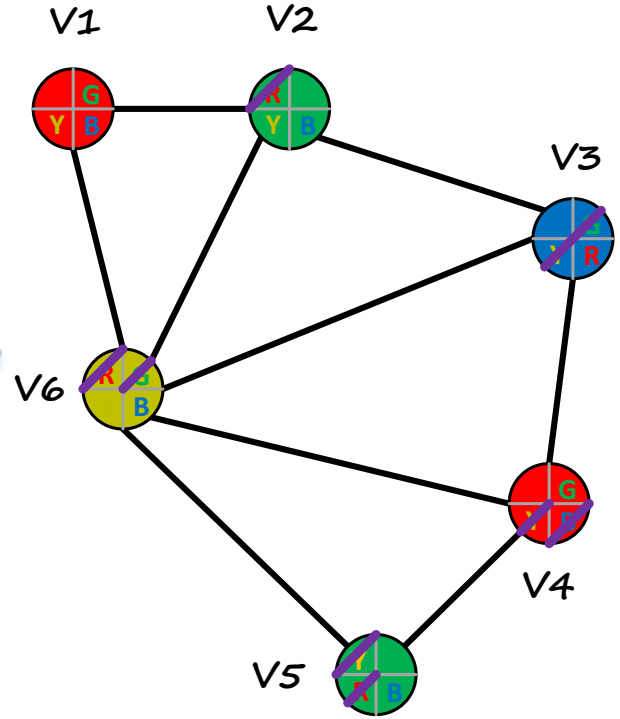


Step 5

Note the domain reduction of V5

MRV = V5

[**Select V5** for next assignment]



Step 6

We are done!

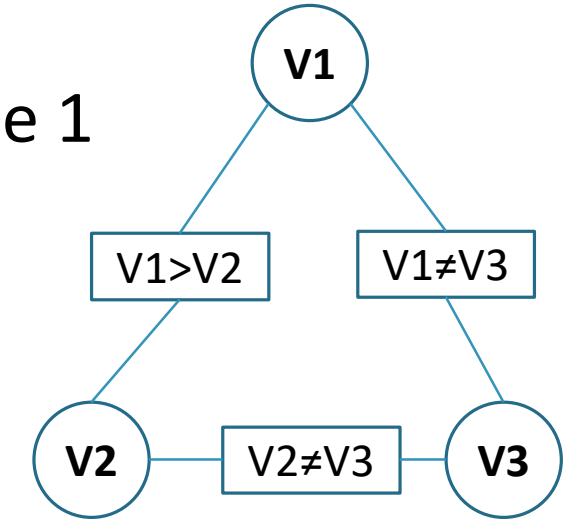
V1 = *Tabuk*	V2 = *Ha'il*	V3 = *Qassim*
V4 = *Riyadh*	V5 = *Makkah*	V6 = *Medina*

Consistency

- Consistency is an inference mechanism to rule out certain variable assignments, which in turn enhances the search
- A single variable is node consistent if no value of its domain is ruled impossible by any of the constraints
- For example, a variable V1 with D_{V1} = {1,2,3} is not node consistent if there is a constraint C_{V1} = {V1 ≥ 2}
  - V1 is node consistent if we remove 1 from its domain
- Node consistency involves unary constraints, i.e., over a single variable

Constraint Graph

- A constraint graph is defined by a graph, with
  - one node for every variable
  - one node for every constraint
  - and undirected edges running between variable nodes and constraint nodes whenever a given variable is involved in each constraint
- An arc is variable pair (X, Y) which share a common constraint in a constraint graph
  - The shown constraint graph represents the CSP of Example 1
    - Variables, V={$V1$,$V2$,$V3$}
    - Domains; D_{V1} = {1,2,3}, D_{V2} = {1,2} and D_{V3} = {2,3}
    - Constraints; $C1$ = $V1$>$V2$, $C2$ = $V2$≠$V3$ and $C3$ = $V1$≠$V3$
    - (V1, V2), (V2, V3) and (V1, V3) are three different arcs

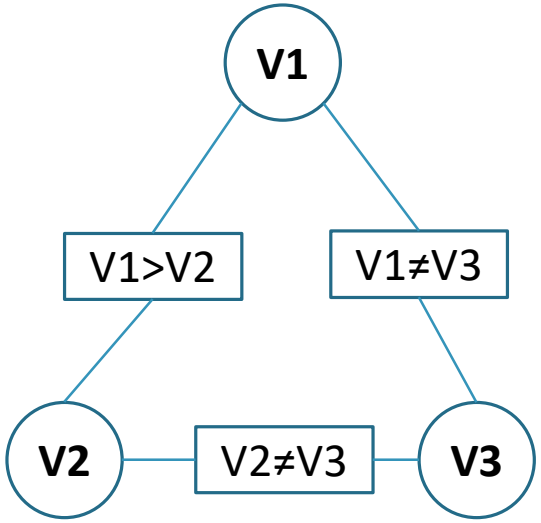


Consistency of a Single Arc

- A variable pair (X, Y) in a constraint graph is arc consistent if for each value $x \in D_X$ there exists a value $y \in D_Y$ such that the assignments $X = x$ and $Y = y$ satisfy all binary constraints between X and Y
- If a pair (X, Y) is not arc consistent, all values $x \in D_X$ for which there is no single value in $y \in D_Y$ that satisfies the constraint, may be deleted from D_X to make (X, Y) arc consistent
- Important: If X loses a value, neighbors of X need to be rechecked!

Consistency of a Single Arc

- Example:
  - Variables, V={$V1$,$V2$,$V3$}
  - Domains; D_{V1} = {1,2,3}, D_{V2} = {1,2} and D_{V3} = {2,3}
  - (V1, V2) is not arc consistent because when V1 = 1, V2 will not have any valid assignment
    - How can we make (V1, V2) arc consistent?
      - By removing 1 from D_{V1}, i.e. becomes D_{V1} = {2,3}
    - What about (V1, V3) (V2, V3)? Are they arc consistent?



Consistency of an Entire CSP

- A CSP is arc consistent if all variable pairs are arc consistent
- Arc consistency is a very efficient mechanism for optimizing the search space of a CSP. It helps prune domains of variables to the minimum possible
- Arc consistency can detect failure earlier than forward checking
- Three possible outcomes when CSP is made arc consistent :
  - One of the domains is empty ⟹ no solution
  - Each domain has a single value ⟹ unique solution
  - Some domains have more than one value ⟹ may or may not be a solution; we have to perform backtracking search!

Arc Consistency – AC-3 Algorithm

- The AC-3 can be implemented using a queue, named here as Arc_**Queue**
- The front node of the queue is represented by Arc_**Queue**.*Head*
- The algorithm repeatedly checks all arcs for consistency
- A function called **Remove-Inconsistent-Values** will remove values that prevent an arc from being consistent
- Neighbors list of a variable X_i is the set of all variables X_j such that X_i and X_j share a common constraint
- The inputs to the algorithm are:
  - The Constraint graph, C_Graph, and the Domains set

Arc Consistency – AC-3 Algorithm

Algorithm: AC-3 (Input: C_Graph, Domain)

1: **append** all arcs in **C_Graph** to Arc_**Queue**

2: **while** Arc_**Queue** is not empty **do**

3: arc (X_i, X_j) = Arc_**Queue**.Head; delete Arc_**Queue**.*Head*

4: **if Remove-Inconsistent-Values**(X_i, X_j) is **true then**

5: if size of D_i = 0 then **return false**

6: **for** each X_k in Neighbors(X_i)-$\{X_j\}$ **do**

7: **append** (X_k, X_i) to queue if not already in queue

8: **return true**

function: Remove-Inconsistent-Values (X_i, X_j)

1: removed **= false**

2: **for each** value x **in Domain(**X_i**) do**

3: **if** no value y in **Domain**(X_j) allows (x, y) to satisfy the constraint (X_i, X_j) **then**

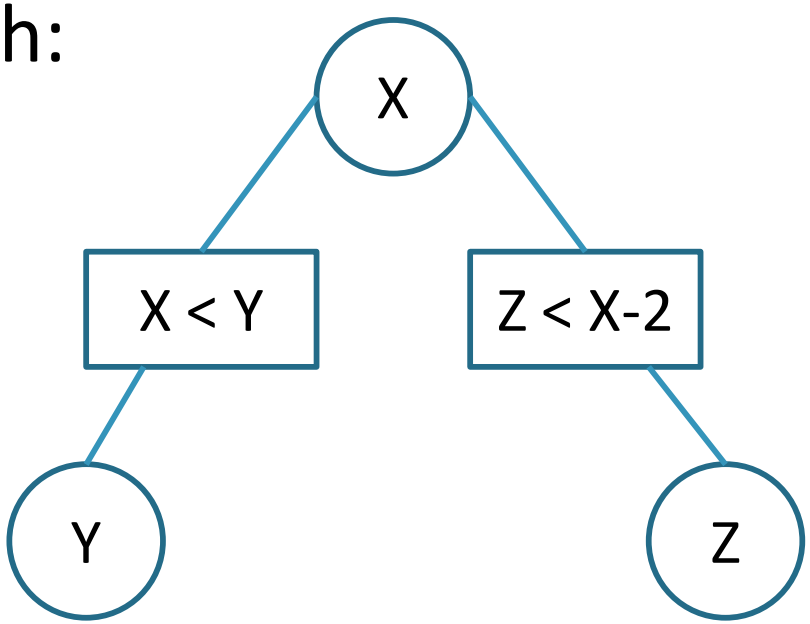
4: delete x from **Domain**(X_i)

5: removed **= true**

6: **return** removed

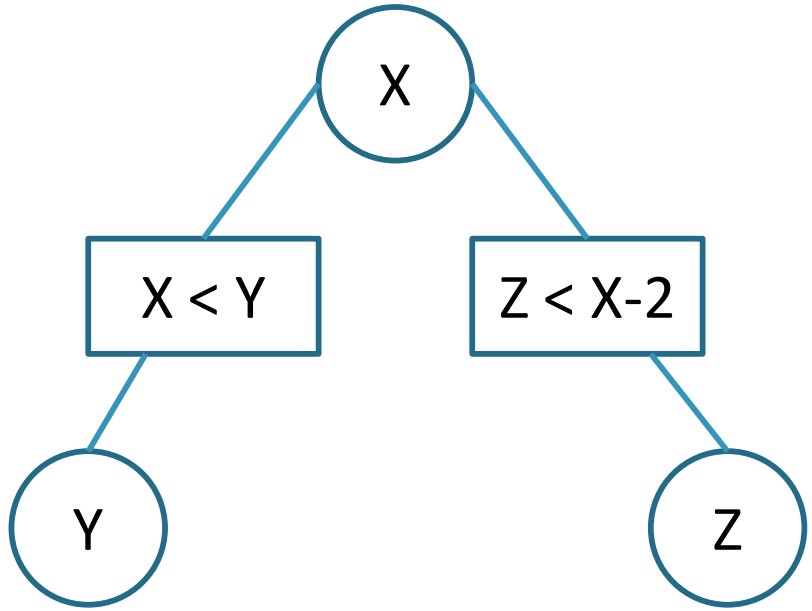
AC-3 Algorithm - Implementation

- Example: Consider the shown constraint graph:
  - Variables, V={X , Y, Z}
  - Domains, $D_X = D_Y = D_Z$ = {1,2,3,4,5,6}
  - Constraints, $C1$ = X < Y, $C2$ = Z < X-2
  - Neighbors list
    - Neighbors(X) = {Y, Z}
    - Neighbors(Y) = {X}
    - Neighbors(Z) = {X}



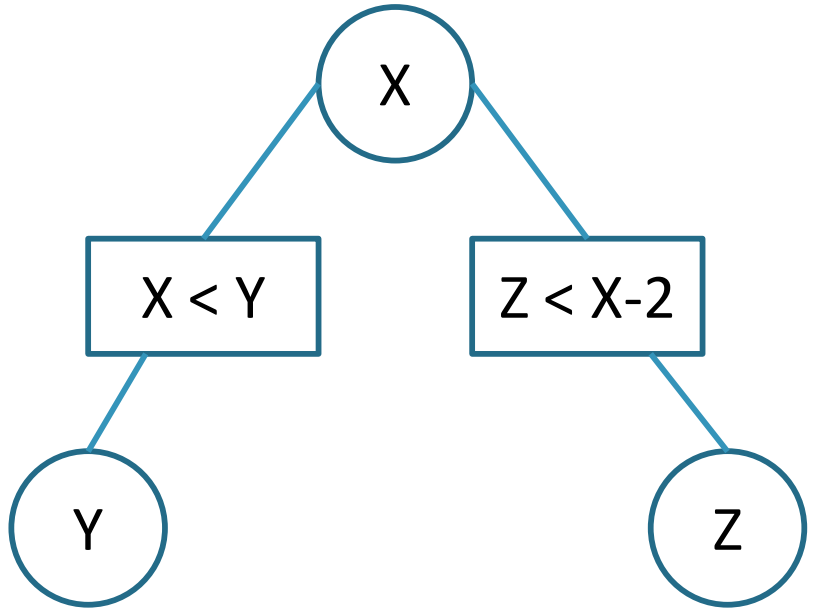
AC-3 Algorithm - Implementation

- Step 1:
  - Enqueue all arcs
  - Arc_**Queue =** (X,Y), (Y,X), (X,Z), (Z,X)
  - Domains: $D_X = D_Y = D_Z = \{1,2,3,4,5,6\}$



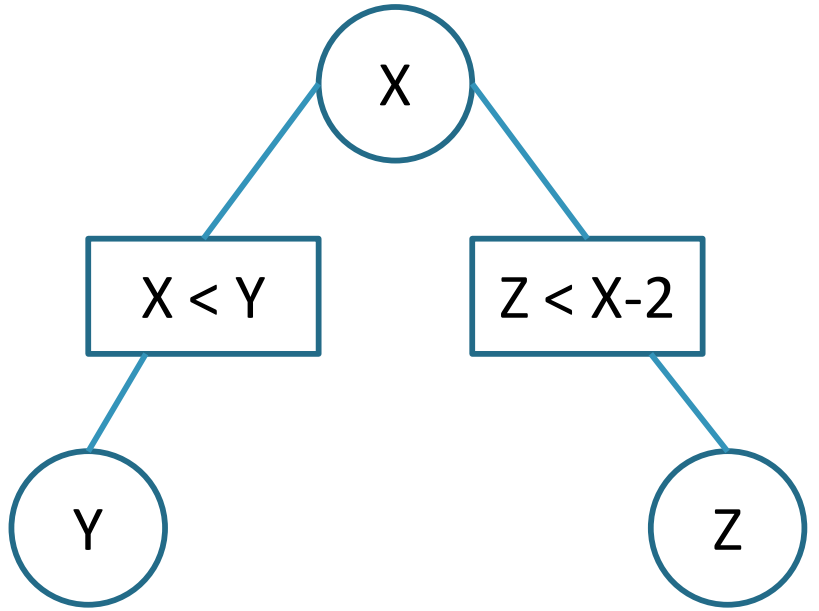
AC-3 Algorithm - Implementation

- Step 2:
  - Check arc (X,Y)
  - Remove 6 from D_X to make (X,Y) arc consistent
  - Domains: $D_X = \{1,2,3,4,5,\cancel{6}\}$, $D_Y = \{1,2,3,4,5,6\}$, $D_Z = \{1,2,3,4,5,6\}$
  - Since D_X has changed, you must re-enqueue all (i, X) arcs where $i \in$ Neighbors(X) and $i \neq$ Y, i.e. enqueue (Z,X) if it is not already enqueued. Since (Z,X) is already enqueued, no need to re-enqueue it
  - Remove arc (X,Y) from Arc_**Queue**
  - Arc_**Queue** = (Y,X), (X,Z), (Z,X)



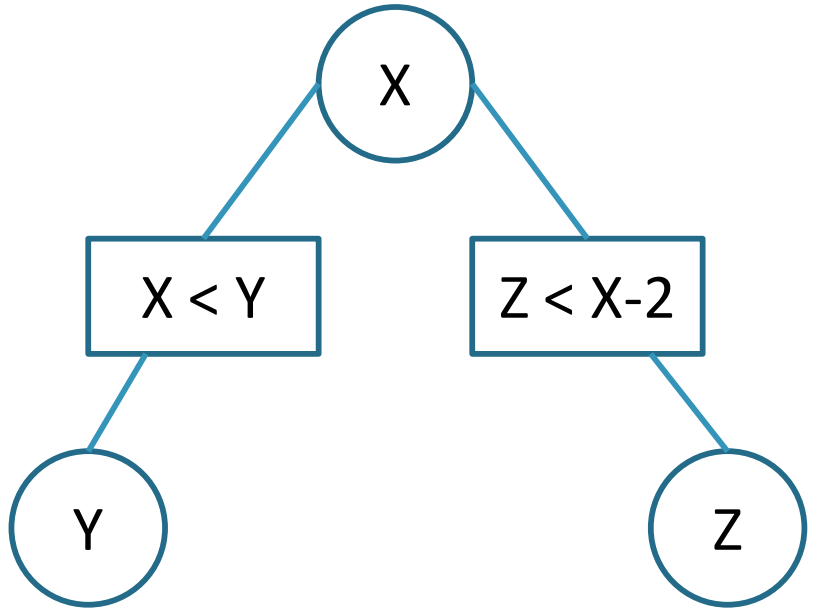
AC-3 Algorithm - Implementation

- Step 3:
  - Check arc (Y,X)
  - Remove 1 from D_Y to make (Y,X) arc consistent
  - Domains: $D_X = \{1,2,3,4,5,\cancel{6}\}$, $D_Y = \{\cancel{1},2,3,4,5,6\}$, $D_Z = \{1,2,3,4,5,6\}$
  - Since D_Y has changed, you must re-enqueue all (i, Y) arcs where $i \in$ Neighbors(Y) and $i \neq$ X. Since Y has only X in its neighbors list, no arc will be re-enqueued
  - Remove arc (Y,X) from Arc_**Queue**
  - Arc_**Queue =** (X,Z), (Z,X)



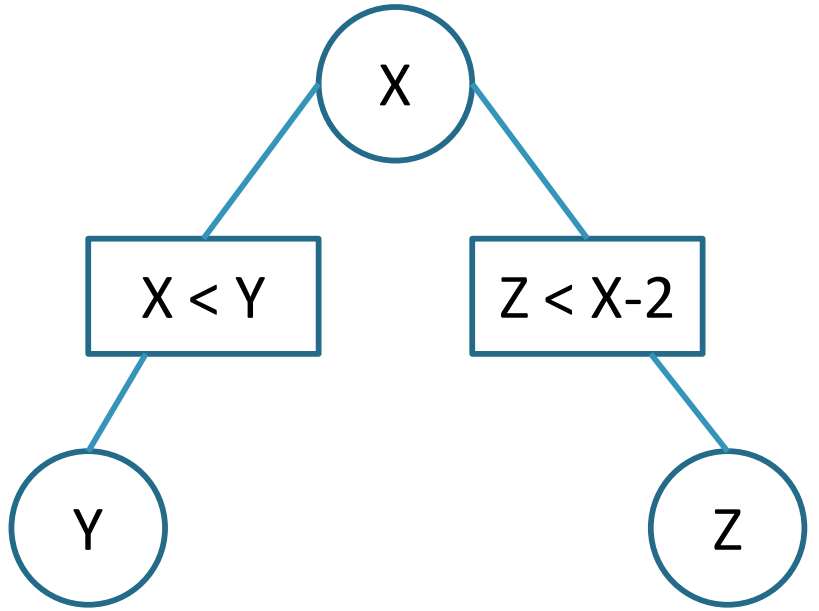
AC-3 Algorithm - Implementation

- Step 4:
  - Check arc (X,Z)
  - Remove 1, 2 and 3 from D_X to make (X,Z) arc consistent
  - Domains: D_X = {~~1~~,~~2~~,~~3~~,4,5,~~6~~}, D_Y = {~~1~~,2,3,4,5,6}, D_Z = {1,2,3,4,5,6}
  - Since D_X has changed, you must re-enqueue all (i, X) arcs where $i \in$ Neighbors(X) and $i \neq$ Z, i.e. enqueue (Y,X) if it is not already enqueued
    ➔ Add arc (Y,X) to Arc_**Queue**
  - Remove arc (X, Z) from Arc_**Queue**
  - Arc_**Queue =** (Z,X), (Y,X)



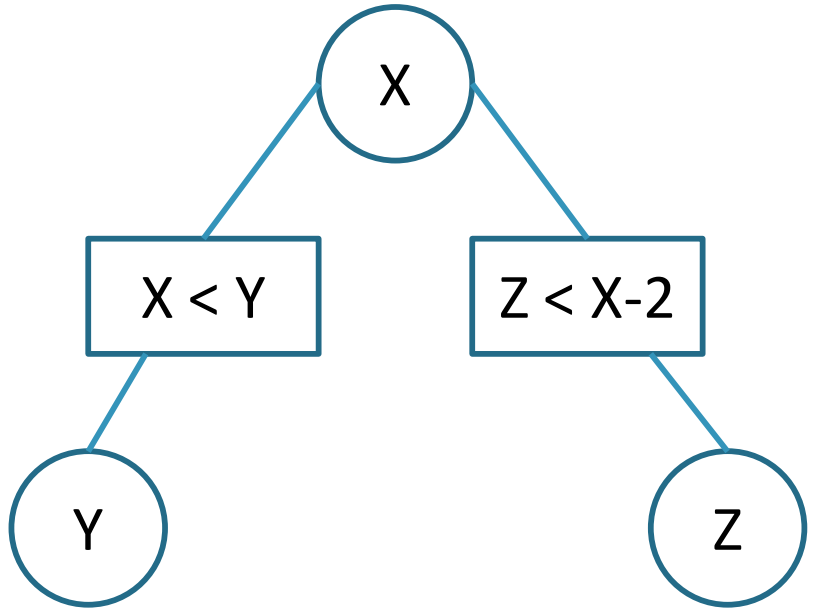
AC-3 Algorithm - Implementation

- Step 5:
  - Check arc (Z,X)
  - Remove 3, 4, 5 and 6 from D_Z to make (Z,X) arc consistent
  - Domains: D_X = {~~1~~,2,~~3~~,4,5,~~6~~}, D_Y = {~~1~~,2,3,4,5,6}, D_Z = {1,2,~~3~~,~~4~~,~~5~~,~~6~~}
  - Since D_Z has changed, you must re-enqueue all (i, Z) arcs where $i \in$ Neighbors(Z) and $i \neq$ X. Since Z has only X in its neighbors list, no arc will be re-enqueued
  - Remove arc (Z,X) from Arc_**Queue**
  - Arc_**Queue =** (Y,X)



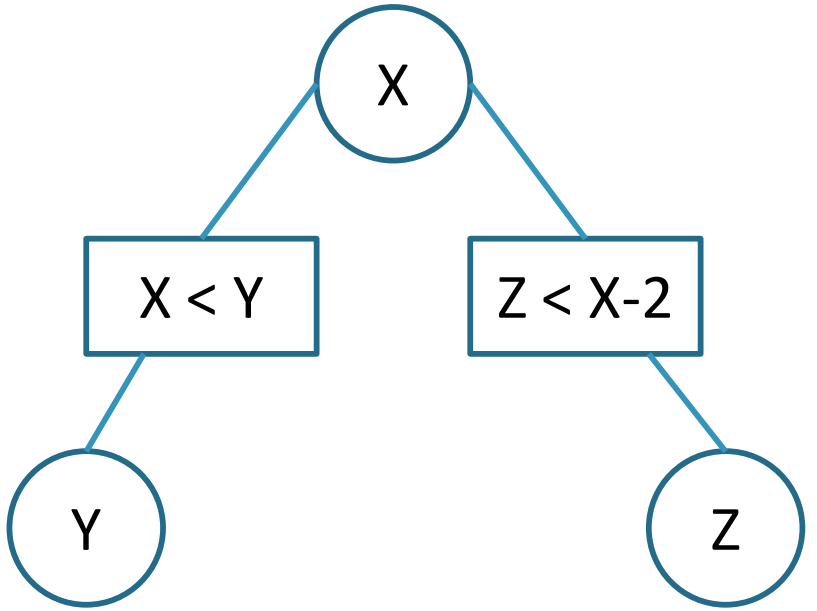
AC-3 Algorithm - Implementation

- Step 6:
  - Check arc (Y,X)
  - Remove 2, 3, 4 from D_Y to make (Y,X) arc consistent
  - Domains: D_X = {~~1~~,2,~~3~~,4,5,~~6~~}, D_Y = {~~1~~,2,~~3~~,~~4~~,5,6}, D_Z = {1,2,~~3~~,~~4~~,~~5~~,~~6~~}
  - Since D_Y has changed, you must re-enqueue all (i, Y) arcs where $i \in$ Neighbors(Y) and $i \neq$ X. Since Y has only X in its neighbors list, no arc will be re-enqueued
  - Remove arc (Y,X) from Arc_**Queue**



AC-3 Algorithm - Implementation

- Step 7:
  - Arc_**Queue** is empty
  - The CSP is now arc consistent
  - Domains: $D_X = \{4,5\}$, $D_Y = \{5,6\}$, $D_Z = \{1,2\}$
- Remarks
  - Reduced domains have more than one value
  - There may or may not be a solution
  - Backtracking search is needed to check if there is a solution
  - There could be more than one solution



Note: there are four solutions:
1. X = 4, Y = 5, and Z = 1
2. X = 4, Y = 6, and Z = 1
3. X = 5, Y = 6, and Z = 1
4. X = 5, Y = 6, and Z = 2

Draw the solution tree to verify!

AC-3 Algorithm - Implementation - Summary

#	Arc_Queue	Constraint	Updated Domains	Action
1.			$D_X = D_Y = D_Z = \{1,2,3,4,5,6\}$	• Start with an empty Arc_Queue • Enqueue all 4 arcs
2.	[**(X,Y)**, (Y,X), (X,Z), (Z,X)]	X < Y	$D_X = \{1,2,3,4,5\}$, $D_Y = \{1,2,3,4,5,6\}$ $D_Z = \{1,2,3,4,5,6\}$	• Check (X,Y). • Remove 6 from D_X to make (X,Y) arc consistent • Since D_X has changed, you must re-enqueue (Z,X) if it is not already enqueued. But it is already enqueued • Remove (X,Y) from Arc_Queue
3.	[**(Y,X)**, (X,Z), (Z,X)]	Y > X	$D_X = \{1,2,3,4,5\}$, $D_Y = \{2,3,4,5,6\}$ $D_Z = \{1,2,3,4,5,6\}$	• Check (Y,X). • Remove 1 from D_y to make (Y,X) arc consistent • Since Y has only X in its neighbors list. No arc need to be re-enqueued • Remove (Y,X) from Arc_Queue

AC-3 Algorithm - Implementation - Summary

#	Arc_Queue	Constraint	Updated Domains	Action
4.	[**(X,Z)**, (Z,X)]	X > Z+2	$D_X = \{4,5\}$, $D_Y = \{2,3,4,5,6\}$ $D_Z = \{1,2,3,4,5,6\}$	• Check (X,Z) • Remove 1, 2 and 3 from D_X to make (X,Z) arc consistent • Since D_X changed, you must enqueue (Y,X) if it is not already enqueued • Remove (X,Z) from Arc_Queue • Add (Y,X) to queue
5.	[**(Z,X)**, (Y,X)]	Z < X-2	$D_X = \{4,5\}$, $D_Y = \{2,3,4,5,6\}$ $D_Z = \{1,2\}$	• Check (Z,X) • Remove 3, 4, 5 and 6 from D_Z to make (Z,X) arc consistent • Since Z has only X in the neighbors list. No arc need to be re-enqueued • Remove (Z,X) from Arc_Queue

AC-3 algorithm - Implementation - Summary

#	Arc_Queue	Constraint	Updated Domains	Action
6.	[(Y,X)]	Y > X	$D_X = \{4,5\}$, $D_Y = \{5,6\}$ and $D_Z = \{1,2\}$	• Check (Y,X) • Remove 2, 3, 4 from D_y to make (Y,X) arc consistent • Since Y has only X in the neighbors list. No arc need to be re-enqueued • Remove (Y,X) from Arc_Queue
7.	[]			Arc_**Queue** is empty. The CSP is now arc consistent

CSP Example 3: Scheduling

- KFUPM decides to open a company called Fly-KFUPM. A study concluded that Fly-KFUPM must have the shown flights per day
- There is about 30 min cleaning time and boarding time for each flight
- Fly-KFUPM has its base in Dammam and Riyadh (Airplane can be parked and stay overnight in a base only)

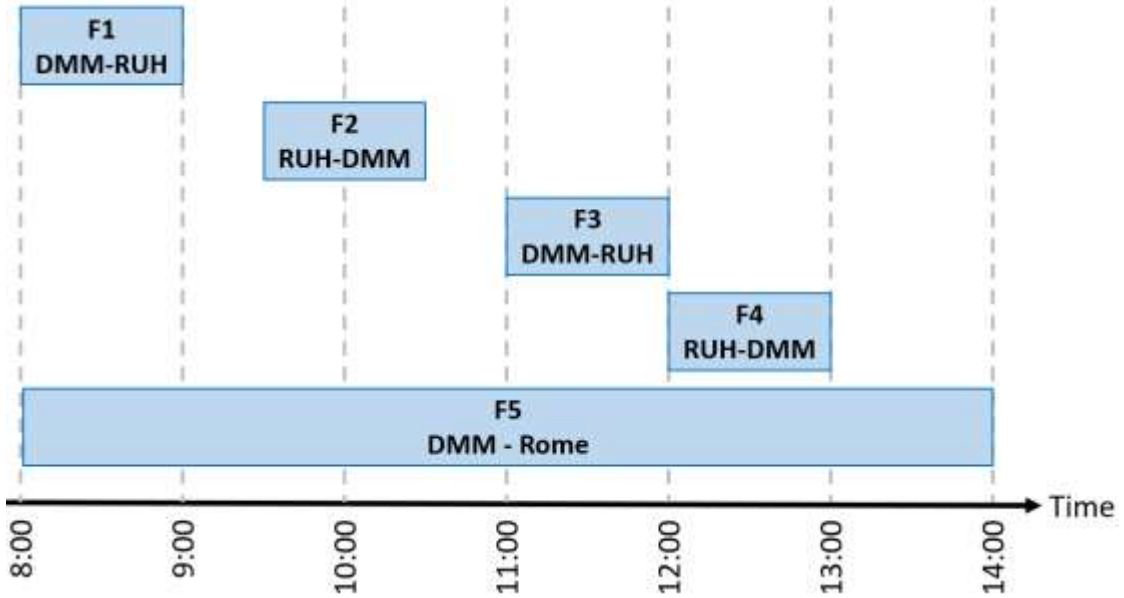
No.	From	To	Time of Departure	Duration in hours
F1	Dammam	Riyadh	08:00	1
F2	Riyadh	Dammam	09:30	1
F3	Dammam	Riyadh	11:00	1
F4	Riyadh	Dammam	12:00	1
F5	Dammam	Rome	8:00	6

CSP Example 3: Scheduling

- The business school is preparing a feasibility study but needs to know how many planes will be required to conduct all flights mentioned above
- You are working over the summer for KFUPM, and you are given the task to determine the minimum number of planes required so that the schedule is met
- **How many planes will you recommend?**

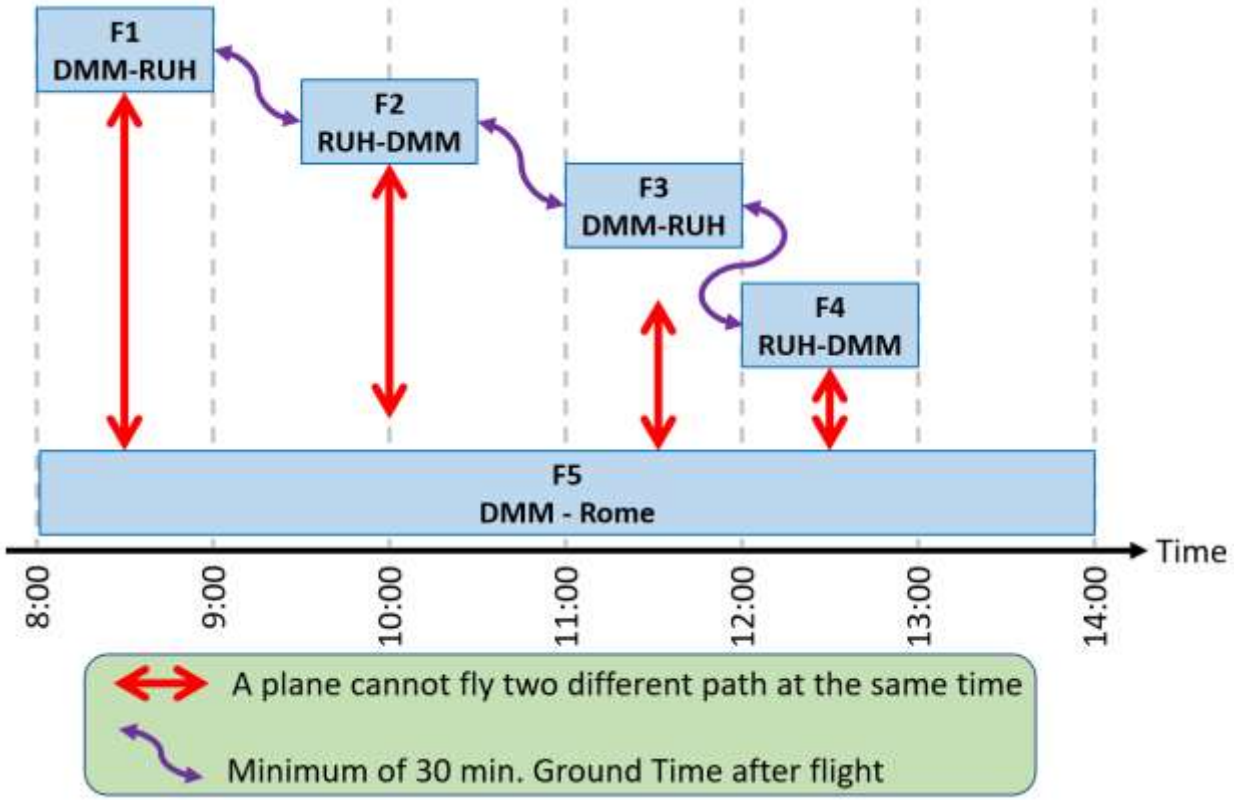
CSP Example 3: Scheduling

- The key to solving this problem is:
  1. Get the right representation
  2. Select an appropriate model to solve the problem
  3. Implement the model
- What is the best representation?
  - The best representation is one that captures all the data in the question and presents it in a way that is easy to perceive



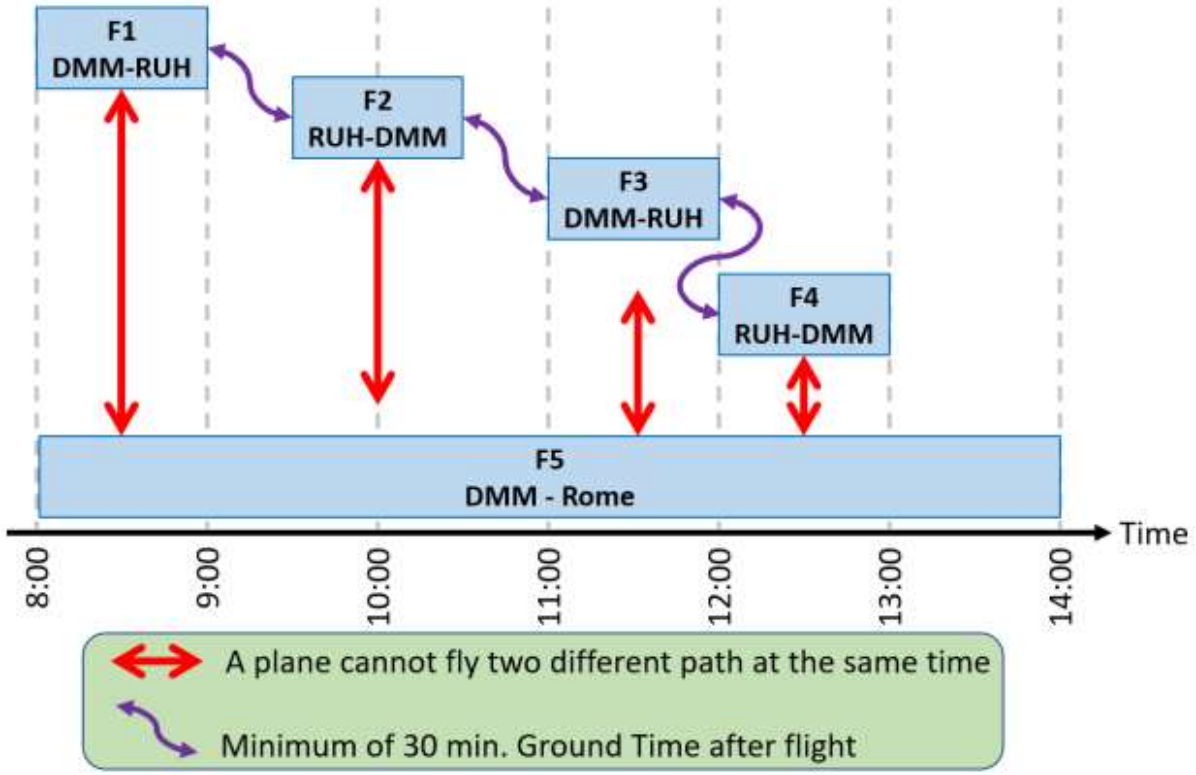
CSP Example 3: Scheduling – Representation

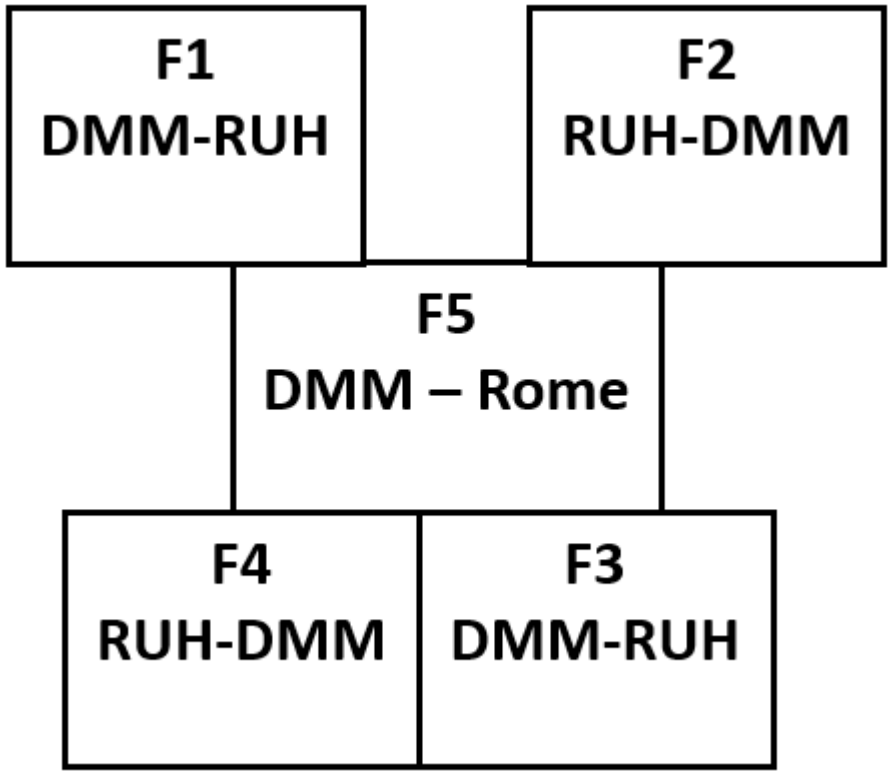
- Add the constraints:
  - Minimum of 30 min Ground Time after flight
  - A plane cannot fly two different paths at the same time
  - A plane can start from either Dammam or Riyadh since Fly-KFUPM has a base there
- Implementation constraint
  - A plane that lands in a city cannot take off from another city unless it flies to that city



CSP Examples 3: Scheduling – Select a Model

- Model = Map Coloring
- Shift the representation to map coloring such that it satisfies all the constraints





CSP Example 3: Scheduling – Implement

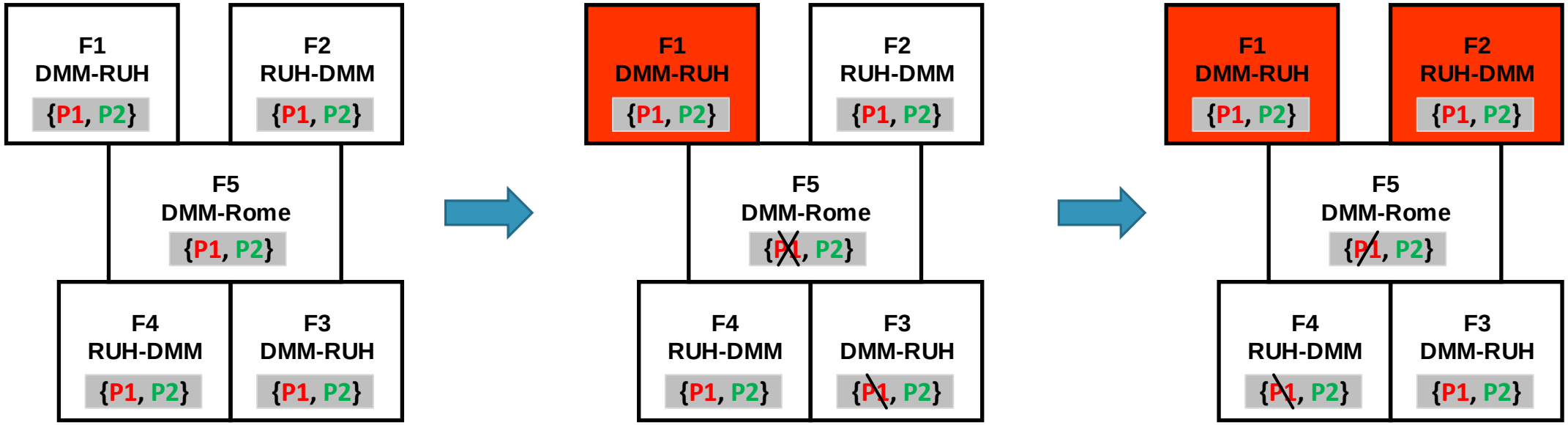
- To determine the minimum numbers of planes required, lets bound the number with the absolute minimum and absolute maximum
  - The absolute minimum number of planes is 1
  - The absolute maximum number of planes is 5 (since there are 5 different paths and each one can be assigned to one)
- To find the number of planes using the map coloring problem use the generate and test (lecture 1), i.e., test starting from 1 plane and keep increasing the number by 1 until you get a success
  - Obviously 1 Plane will not work

CSP Example 3: Scheduling – Implement 2 Planes

- 2 Planes with sequential assignment of planes where P1 is the first plane and P2 is the second one
  - **Variables**:
    - V={V1 = F1, V2 = F2, V3 = F3, V4 = F4, V5 = F5}
  - **Domain** of each variable is {P1, P2}
  - Constraints:
    - Minimum of 30 min Ground Time after flight
    - A plane cannot fly two different paths at the same time
    - A plane can start from either Dammam or Riyadh since Fly-KFUPM has a base there
    - A plane that lands in a city cannot take off from another city unless it flies to that city

CSP Example 3: Scheduling – Implement 2 Planes

- CSP Using backtracking algorithm

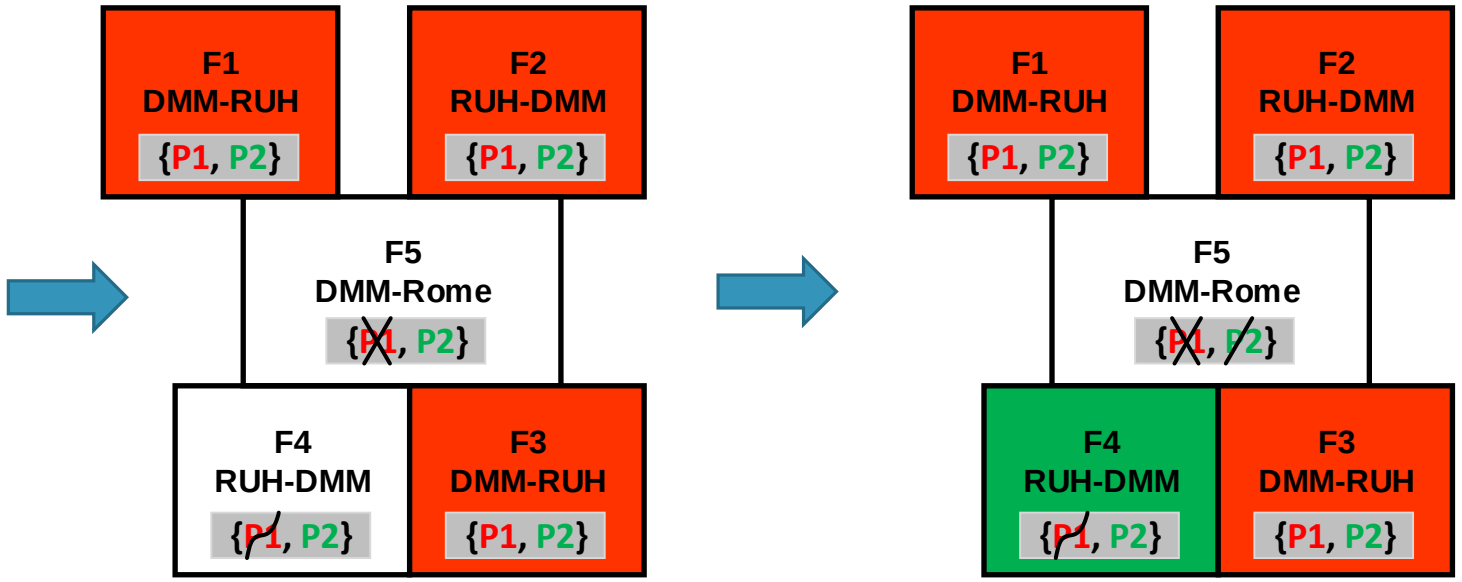


Constraint due to a plane cannot fly two different paths

Constraint due to a plane is in another city

Constraint not enough ground time

CSP Example 3: Scheduling – Implement 2 Planes



Constraint due to a plane cannot fly two different paths

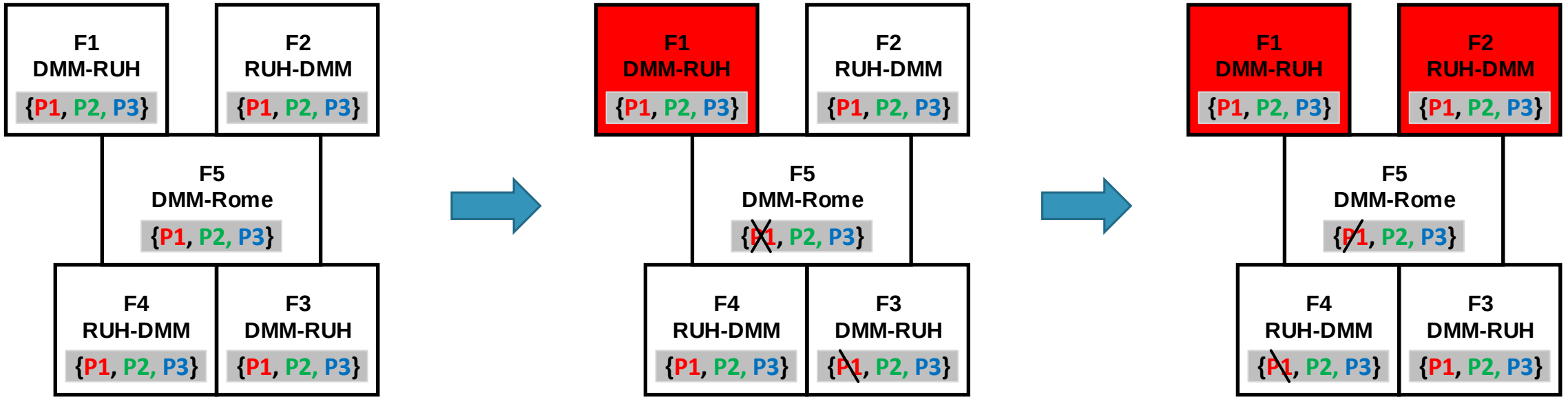
Constraint due to a plane is in another city

Constraint not enough ground time

- Planes are not enough → increase the number of planes to 3 **Domain** {P1, P2, P3}

CSP Example 3: Scheduling – Implement 3 Planes

- CSP Using backtracking algorithm:

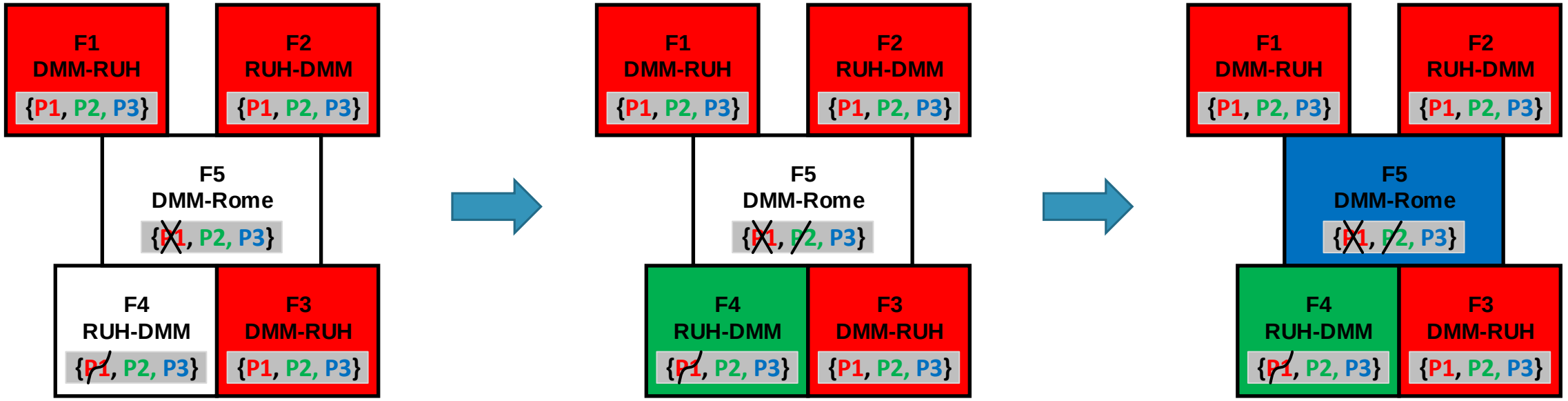


Constraint due to a plane cannot fly two different paths

Constraint due to a plane is in another city

Constraint not enough ground time

CSP Example 3: Scheduling – Implement 3 Planes



Constraint due to a plane cannot fly two different paths

Constraint due to a plane is in another city

Constraint not enough ground time

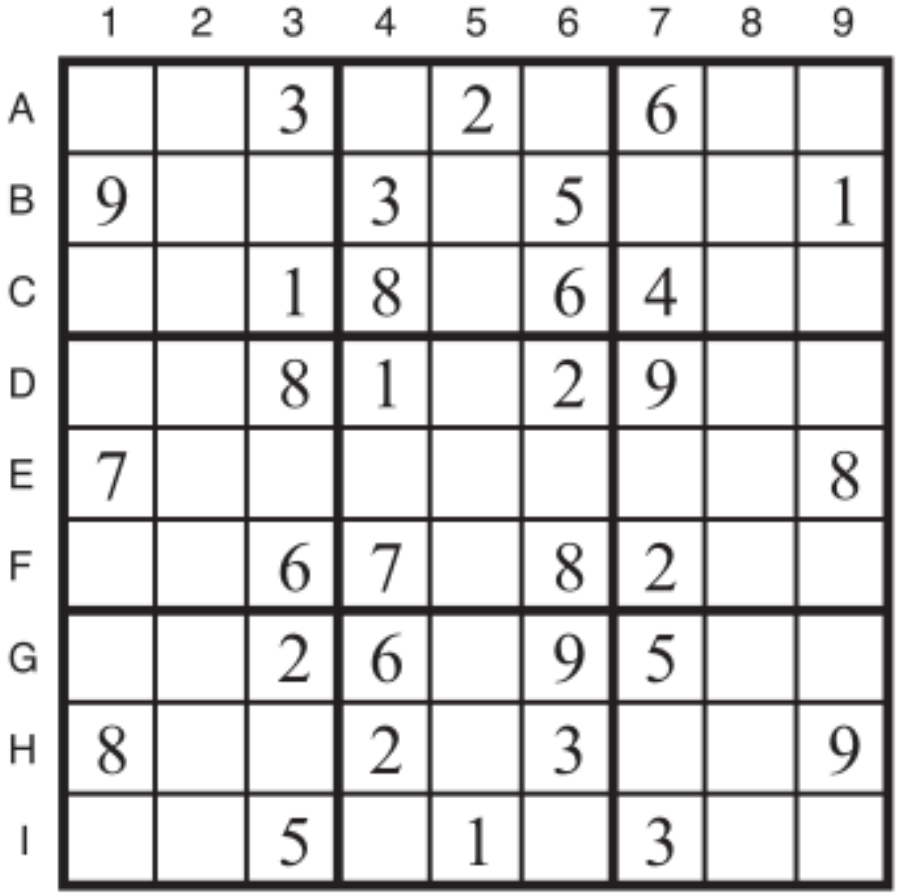
CSP Example 4: Sudoku Puzzle

- The Sudoku problem objective is to fill a 9 x 9 grid with numerical digits so that each column, each row and each of the nine 3 x 3 sub-grids (also called boxes) contains one of all the digits 1 through 9
- A typical Sudoku puzzle is shown:

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

CSP Example 4: Sudoku Puzzle - Model

- It has 81 variables: {A1, A2, …, A9, …, I1, I2 , …, I9}
- Empty squares domain : {1, 2, 3, 4, 5, 6, 7, 8, 9}
- 27 All different (Alldiff) constraints:
  - Rows:
    - Row A: Alldiff (A1, A2, A3, A4, A5, A6, A7, A8, A9)
    - Row B: Alldiff (B1, B2, B3, B4, B5, B6, B7, B8, B9)
    - …
  - Columns:
    - Column 1: Alldiff (A1, B1, C1, D1, E1, F1, G1, H1, I1)
    - Column 2: Alldiff (A2, B2, C2, D2, E2, F2, G2, H2, I2)
    - …
  - 3x3 sub-grids
    - Alldiff (A1, A2, A3, B1, B2, B3, C1, C2, C3)
    - Alldiff (A4, A5, A6, B4, B5, B6, C4, C5, C6)
    - …



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

CSP Example 4: Sudoku Puzzle - Model

- We can solve the problem by using generate-and-test method, but it will take 9^n, where n is the number of unfilled boxes
  - In this example, it is $9^{49}=5.7x10^{46}$
- Using constraint satisfaction problem, we can reduce the time

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

CSP Example 4: Sudoku Puzzle - Model

- For CSP we define the following:
  - The variables are the empty squares
  - The domain of each variable is {1,2,3,4,5,6,7,8,9}
- The constraints are:
  - Each numerical digit in row X is unique or we write formally
    - $\forall X \in \{A, B, \dots, I\}$ Alldifferent $\{X1, X2, \dots, X9\}$
  - Each numerical digit in j^{th} column is unique or we write formally
    - $\forall j \in \{1, \dots, 9\}$ Alldifferent $\{Aj, Bj, \dots, Ij\}$
  - Each numerical digit in a sub-grid is unique or we write formally
    - For $m = 0, 1, 2$:
      - Alldifferent $\{A(1+3m), A(2+3m), A(3+3m), B(1+3m), B(2+3m), B(3+3m), C(1+3m), C(2+3m), C(3+3m)\}$
      - Alldifferent $\{D(1+3m), D(2+3m), D(3+3m), E(1+3m), E(2+3m), E(3+3m), F(1+3m), F(2+3m), F(3+3m)\}$
      - Alldifferent $\{G(1+3m), G(2+3m), G(3+3m), H(1+3m), H(2+3m), H(3+3m), I(1+3m), I(2+3m), I(3+3m)\}$

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

CSP Example 4: Sudoku Puzzle - Model

- We can define the domain of $A1$={1,2,3,4,5,6,7,8,9} and apply the following constraints:
  - Row constraint: $A1$≠{3,2,6}
  - Column constraint: $A1$≠{9,7,8}
  - Sub-grid constraint: $A1$≠{3,9,1}
- This reduces the domain of A1 to {4,5}
- We keep doing this for the remaining cells and then solve the CSP problem using backtracking

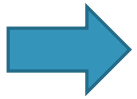
	1	2	3	4	5	6	7	8	9
A			3		2		6		
B	9			3		5			1
C			1	8		6	4		
D			8	1		2	9		
E	7								8
F			6	7		8	2		
G			2	6		9	5		
H	8			2		3			9
I			5		1		3		

CSP Example 4: Sudoku Puzzle – Implementation

- The solution is then:

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

Solved by AC-3 alone

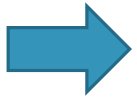


	1	2	3	4	5	6	7	8	9
A	4	8	3	9	2	1	6	5	7
B	9	6	7	3	4	5	8	2	1
C	2	5	1	8	7	6	4	9	3
D	5	4	8	1	3	2	9	7	6
E	7	2	9	5	6	4	1	3	8
F	1	3	6	7	9	8	2	4	5
G	3	7	2	6	8	9	5	1	4
H	8	1	4	2	5	3	7	6	9
I	6	9	5	4	1	7	3	8	2

CSP: Sudoku Puzzle – Backtracking Example

			1			7		2
	3		9	5				
		1			2			3
5	9					3		1
	2						7	
7		3					9	8
8			2			1		
				8	5		6	
6		5			9			

After running AC-3, no variable was assigned

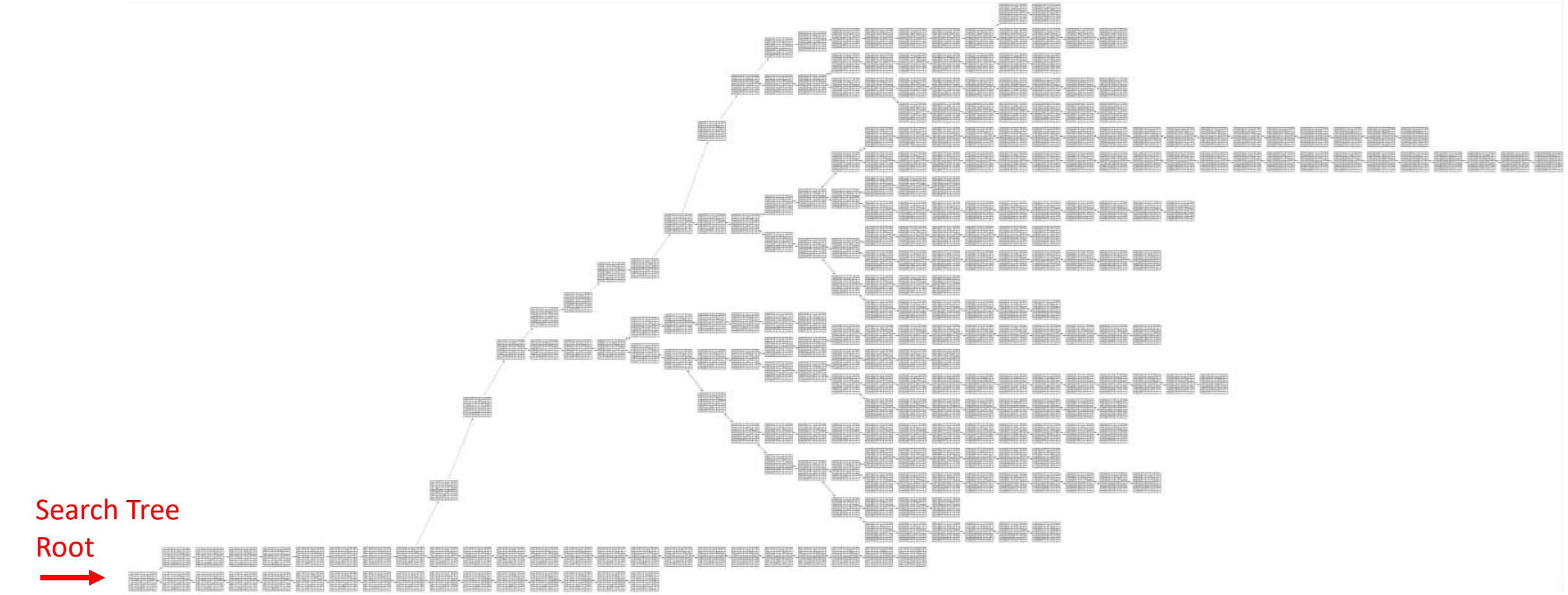


			1			7		2
	3		9	5				
		1			2			3
5	9					3		1
	2						7	
7		3					9	8
8			2			1		
				8	5		6	
6		5			9			

https://sandipanweb.wordpress.com/2017/03/17/solving-sudoku-as-a-constraint-satisfaction-problem-using-constraint-propagation-with-arc-consistency-checking-and-then-backtracking-with-minimum-remaning-value-heuristic-and-forward-checking/

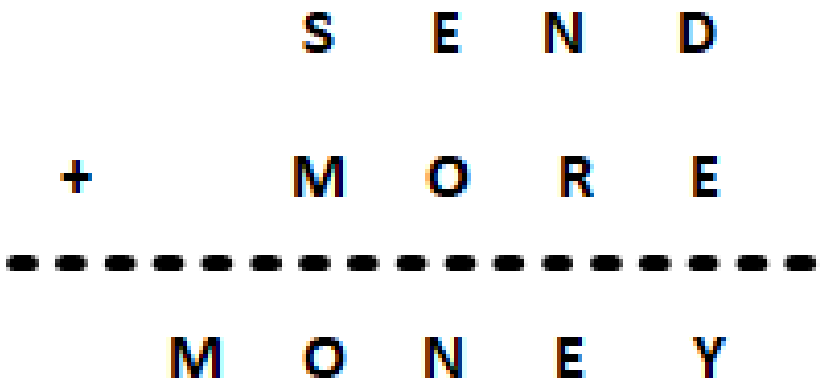
CSP: Sudoku – Tree

- A typical tree constructed by a computer is shown below:



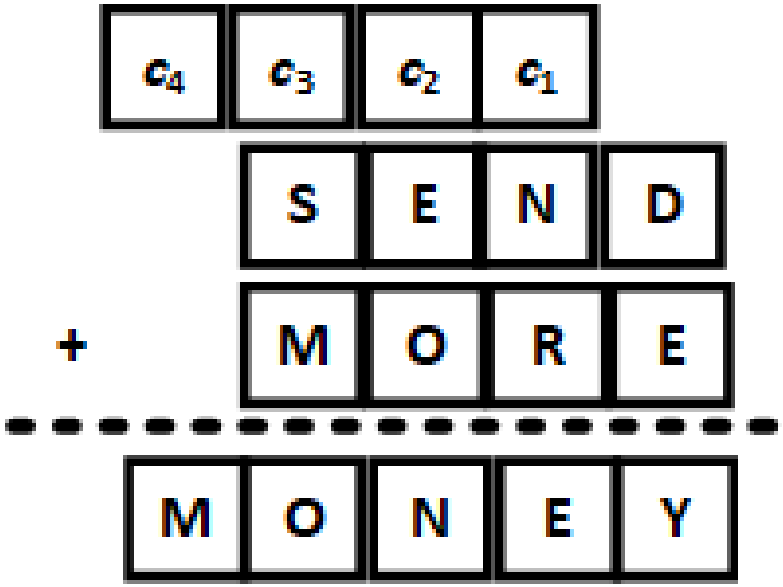
CSP: Cryptarithmetic

- Given the shown expression, Find the value of each letter given the following:
  1. Each letter should have a ***unique*** and distinct value
  2. Each letter represents ***only one-digit*** throughout the problem
  3. Numbers must ***not begin with zero*** i.e., 0937 (wrong), 937 (correct).
  4. After replacing letters by their digits, the resulting ***arithmetic operations must be correct***



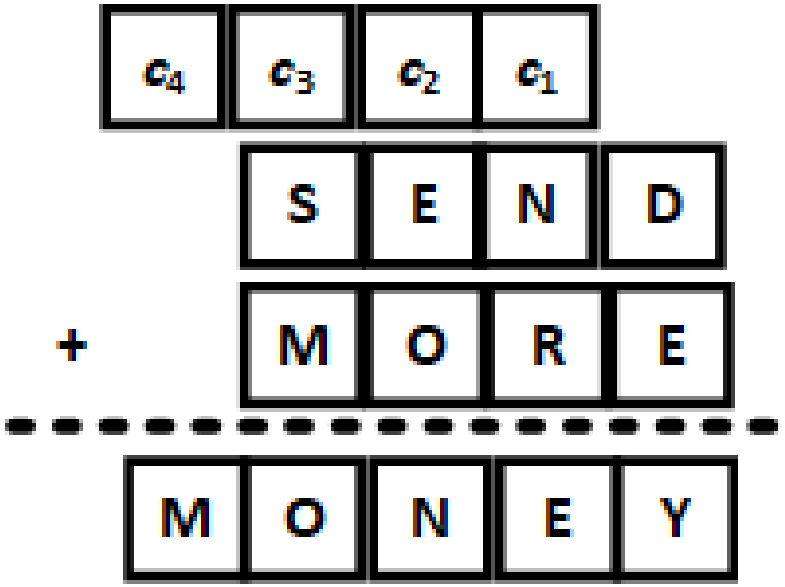
CSP: Cryptarithmetic

- Start by understanding addition and identify
  - Variables V= {S, E, N, D, M, O, R, Y, c_1, c_2, c_3, c_4}
  - The domains:
    - S={0,1,2,3,4,5,6,7,8,9}; domain of variable S
    - $S = E = N = D = M = O = R = Y$
    - c_1 = {0,1}; domain of variable c_1
    - $c_1 = c_2 = c_3 = c_4$
- Note: We need to introduce the carry to enable solving the problem



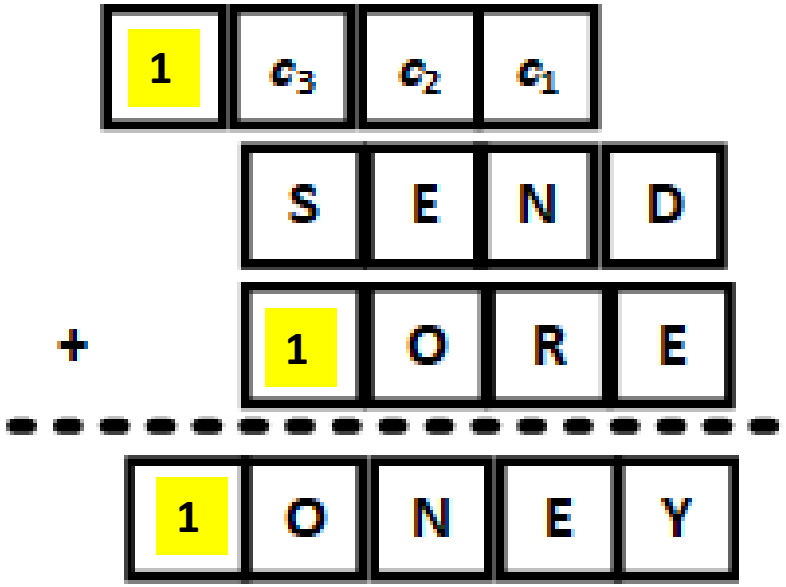
CSP: Cryptarithmetic

- Constraints:
  1. Each letter should have a unique and distinct value
  2. Each letter represents only one-digit throughout the problem
  3. Numbers must not begin with zero, i.e., 0937 (wrong), 937 (correct)
  4. After replacing letters by their digits, the resulting arithmetic operations must be correct



CSP: Cryptarithmetic

- Step 1: Using constraint (3), i.e. numbers must not begin with zero, we can say that the domain of S, M is reduced to {1,2,3,4,5,6,7,8,9}
- Step 2: Since M=c_4 , c_4 cannot be 0 because this will make M=0. Therefore, c_4=1 and M=1 and the problem reduces to

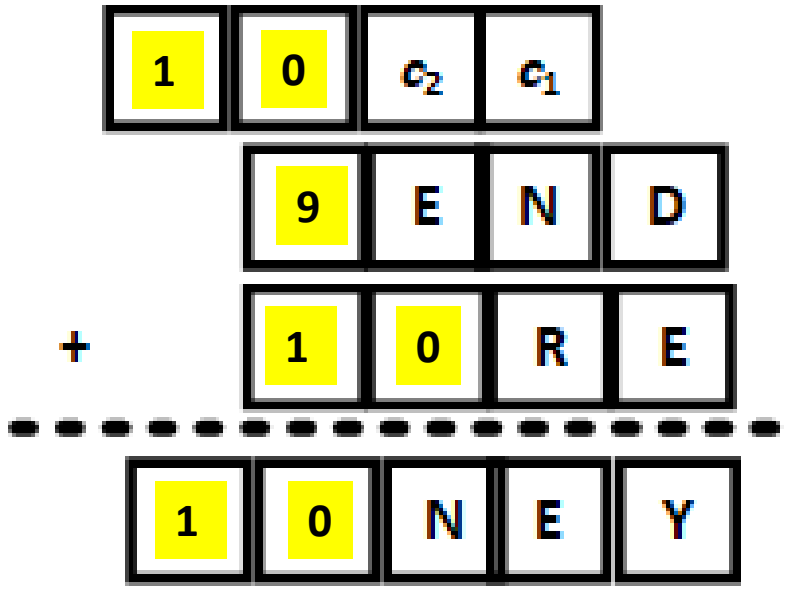


S={2,3,4,5,6,7,8,9} *M*={1}
E={0,2,3,4,5,6,7,8,9} *O*={0,2,3,4,5,6,7,8,9}
N={0,2,3,4,5,6,7,8,9} *R*={0,2,3,4,5,6,7,8,9}
D={0,2,3,4,5,6,7,8,9} *Y*={0,2,3,4,5,6,7,8,9}

Updated domains of variables

CSP: Cryptarithmetic

- Step 3: consider c_3+S+1=10+O (Note that the 10 is due to c4=1). From this equation, the domain of S is reduced to {8,9}
  Therefore, all possible cases for c_3, S, O are:
  - if c3=0 and S=9 then O=0
  - if c3=1 and S=8 then O=0
  - ~~if c3=1 and S=9 then O=1,~~
  - A contradiction is observed with constraint (1) where M=O=1 ⇒ discard this option
  - We will select c3=0, S=9 and O=0 and the problem reduces to:

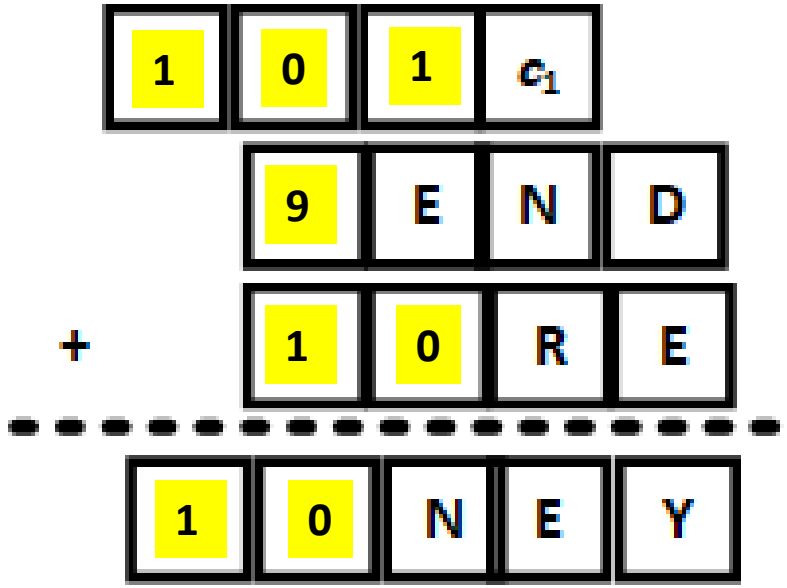


S={9} M={1}
E={2,3,4,5,6,7,8} O={0}
N={2,3,4,5,6,7,8} R={2,3,4,5,6,7,8}
D={2,3,4,5,6,7,8} Y={2,3,4,5,6,7,8}

Updated domains of variables

CSP: Cryptarithmetic

- Step 4: consider the equation $c2+E=N$. It has two possible solutions:
  - if c_2=0 then we have $E=N$, this contradicts with constraint (1), i.e. Each letter should have a unique and distinct value → discard this option
  - c_2=1 then we have $1+E=N$

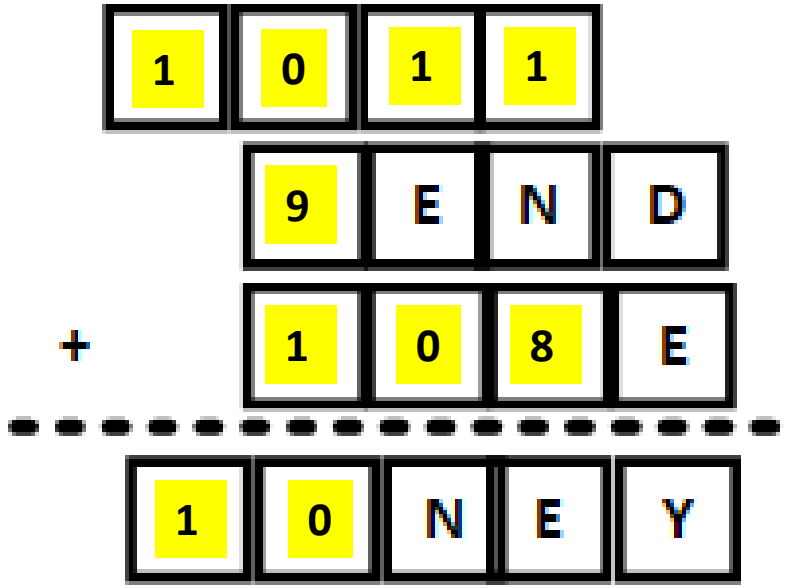


S={9} *M*={1}
E={2,3,4,5,6,7,8} *O*={0}
N={2,3,4,5,6,7,8} *R*={2,3,4,5,6,7,8}
D={2,3,4,5,6,7,8} *Y*={2,3,4,5,6,7,8}

Updated domains of variables

CSP: Cryptarithmetic

- Step 5: Consider c_1+N+R=10+E. Using Step 4, (1+E=N), then substituting above we get $c1$+R=9
  - Selecting c_1=0 will result with R=9, contradicts with constraint (1) since we have already chosen S=9, discard the option
  - Thus, selecting c_1=1 will result with R=8 and the problem reduces to:



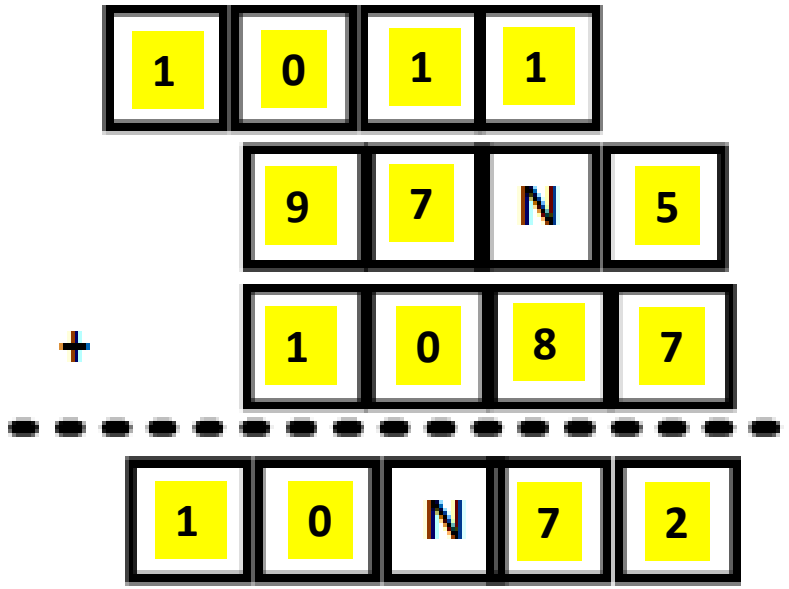
S={**9**}
E={2,3,4,5,6,7}
N={2,3,4,5,6,7}
D={2,3,4,5,6,7}

M={**1**}
O={**0**}
R={**8**}
Y={2,3,4,5,6,7}

Updated domains of variables

CSP: Cryptarithmetic

- Step 6: Consider the equation $D+E=10+Y$, since the numbers 0,1,8 and 9 are taken, we need two numbers that generate a sum greater than 9. The remaining options are:
  - D=3 and E=7 that makes $D+E$=10 and Y=0, but since O=0 has already been selected, discard this option
  - D=4 and E=7 that makes $D+E$= 11 and Y=1, but since M=1 has already been selected, discard this option
  - D=5 and E=7 that makes Y=2

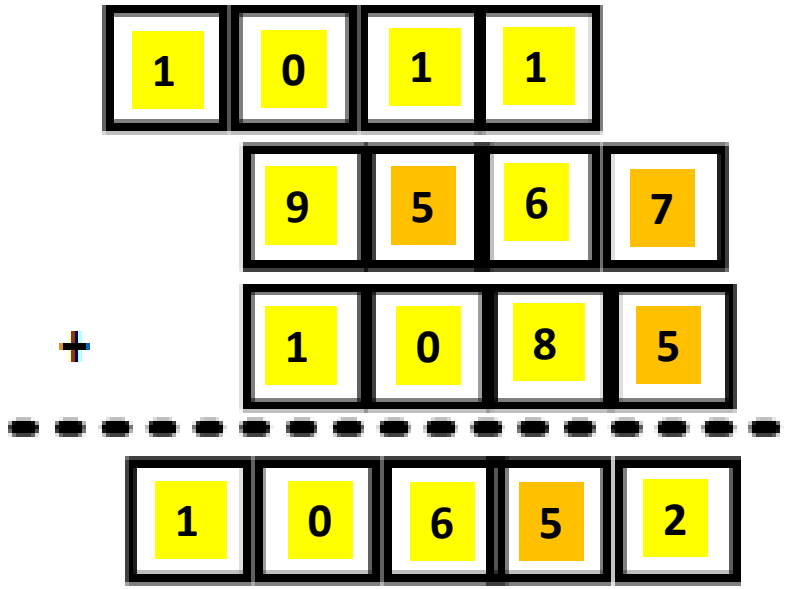


S={9}
E={7}
N={3,4,6}
D={5}
M={1}
O={0}
R={8}
Y={2}

Updated domains of variables

CSP: Cryptarithmetic

- Step 7: Going back to Step 4, (1+E=N), and substituting E=7 makes N=8, this contradicts with constraint (1)→ backtrack
- Select E=5 that makes
  - N= 1+E=6,
  - D= 7
  - Y=2 (no change)
- The final solution then becomes:



S={9}
E={5}
N={6}
D={7}

M={1}
O={0}
R={8}
Y={2}

Updated domains of variables

Cryptarithmetic Puzzles Program

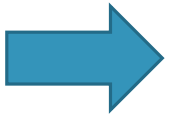
- Summary of the problem: A cryptarithmetic puzzle is a mathematical exercise where the digits of some numbers are represented by letters (or symbols). Each letter represents a unique digit. The goal is to find the digits such that a given mathematical equation is verified
- Example:

```
      CP
+     IS
+    FUN
--------
=   TRUE
```

Cryptarithmetic Puzzles Program

- One assignment (Solution) of letters to digits yields the following equation:

```
      CP                 23
+     IS           +     74
+    FUN           +    968
--------           --------
=   TRUE           =   1065
```



- There are other answers to this problem. We'll show how to find all solutions next

Cryptarithmetic Puzzles Library

- To run the code we will be discussing, you need to install a library to run the code given below. To install, open the command prompt and type the following:

  `python -m pip install --upgrade --user ortools`

  Press Enter and wait until it finishes

  OR (from a notebook) **`!pip install ortools`**

- Note: you may need to load the notebook again for the tool to work or run the server again depending on your machine

Cryptarithmetic Puzzles Modeling

- As with any CSP, we start by identifying variables and constraints.
  - The variables are the letters, which can take on any single digit value
    - V={C, P, I, S, F, U, N, T, R, E}
  - Note that there will be no repeating variable. Also, the values of the variables are single digits, therefore the domain is 0 to 9
- Follow the steps to program your machine to solve the cryptarithmetic problems

Cryptarithmetic Puzzles Code

- Step 1: Import the required libraries.

```
from ortools.sat.python import cp_model
model = cp_model.CpModel()
```

Step 2: Set up the variables

```
base = 10
c = model.NewIntVar(1, 9, 'C')
p = model.NewIntVar(0, 9, 'P')
i = model.NewIntVar(1, 9, 'I')
s = model.NewIntVar(0, 9, 'S')
f = model.NewIntVar(1, 9, 'F')
u = model.NewIntVar(0, 9, 'U')
n = model.NewIntVar(0, 9, 'N')
t = model.NewIntVar(1, 9, 'T')
r = model.NewIntVar(0, 9, 'R')
e = model.NewIntVar(0, 9, 'E')
# We need to group variables in a list to
use the constraint AllDifferent.
letters = [c, p, i, s, f, u, n, t, r, e]
# Verify that we have enough digits.
assert base >= len(letters)
```

Cryptarithmetic Puzzles Code

- Step 3: Identify the constraints
  - The variables are letters and can take on a single digit value
  - Therefore, we can establish the following constraints
    - CP + IS + FUN = TRUE
    - Each of the letters must be a different digit
    - C, I, F, T ≠ 0 (since leading digit in a number is not zero)
  - The code below defines the constraints and the objective function

```
# Define the constraints
model.AddAllDifferent(letters)

# CP + IS + FUN = TRUE
model.Add(c * 10 + p + i * 10 + s + f * 10**2 + u * 10 + n == t * 10**3
+ r * 10**2 + u * 10 + e)
```

Cryptarithmetic Puzzles Code

- Step 4:Printing all Solutions
  - We want the solver to print solutions as it finds them. The code below does that:

```
# Solution Printer Class
class VarArraySolutionPrinter(cp_model.CpSolverSolutionCallback):
    def __init__(self, variables):
        cp_model.CpSolverSolutionCallback.__init__(self)
        self.__variables = variables
        self.__solution_count = 0

    def on_solution_callback(self):
        self.__solution_count += 1
        for v in self.__variables:
            print('%s=%i ' %(v, self.Value(v)), end='')
        print()

    def solution_count(self):
        return self.__solution_count
```

Cryptarithmetic Puzzles Code

- Step 5: Invoke the Solver and call the printer.

```
# Call the solver
solver = cp_model.CpSolver()
solution_printer = VarArraySolutionPrinter(letters)
status =  solver.SearchForAllSolutions(model, solution_printer)
```

- Sample Run

```
C=6 P=4 I=3 S=5 F=9 U=2 N=8 T=1 R=0 E=7
C=3 P=4 I=6 S=5 F=9 U=2 N=8 T=1 R=0 E=7
C=3 P=4 I=6 S=8 F=9 U=2 N=5 T=1 R=0 E=7
C=3 P=2 I=6 S=7 F=9 U=8 N=5 T=1 R=0 E=4
C=3 P=2 I=6 S=5 F=9 U=8 N=7 T=1 R=0 E=4
C=2 P=3 I=7 S=6 F=9 U=8 N=5 T=1 R=0 E=4
C=2 P=3 I=7 S=4 F=9 U=6 N=8 T=1 R=0 E=5
C=2 P=3 I=7 S=5 F=9 U=4 N=8 T=1 R=0 E=6
C=2 P=3 I=7 S=8 F=9 U=4 N=5 T=1 R=0 E=6
C=2 P=3 I=7 S=8 F=9 U=6 N=4 T=1 R=0 E=5
C=2 P=3 I=7 S=5 F=9 U=8 N=6 T=1 R=0 E=4
C=7 P=3 I=2 S=5 F=9 U=8 N=6 T=1 R=0 E=4
```

Cryptarithmetic Puzzles Code

- Step 6: Analyze Data:

```
print()
print('Statistics')
print('  - status          : %s' % solver.StatusName(status))
print('  - conflicts       : %i' % solver.NumConflicts())
print('  - branches        : %i' % solver.NumBranches())
print('  - wall time       : %f s' % solver.WallTime())
print('  - solutions found : %i' % solution_printer.solution_count())
```

- Result

```
Statistics
  - status          : OPTIMAL
  - conflicts       : 179
  - branches        : 828
  - wall time       : 0.071651 s
  - solutions found : 72
```

Conclusion

- Constraint Satisfaction Problems consist of variables, associated with finite domains, and constraints which are binary relations specifying permissible value pairs
- In CSP, the objective is finding a solution for a problem where you need to satisfy some constraints. This has numerous real life applications including, e.g., scheduling and timetabling
- Backtracking search assigns values to variables one-by-one, and backtracks when there is an inconsistent assignment
- Improvements such as forward checking, MRV and arc consistency reduce the size of the CSP search space