

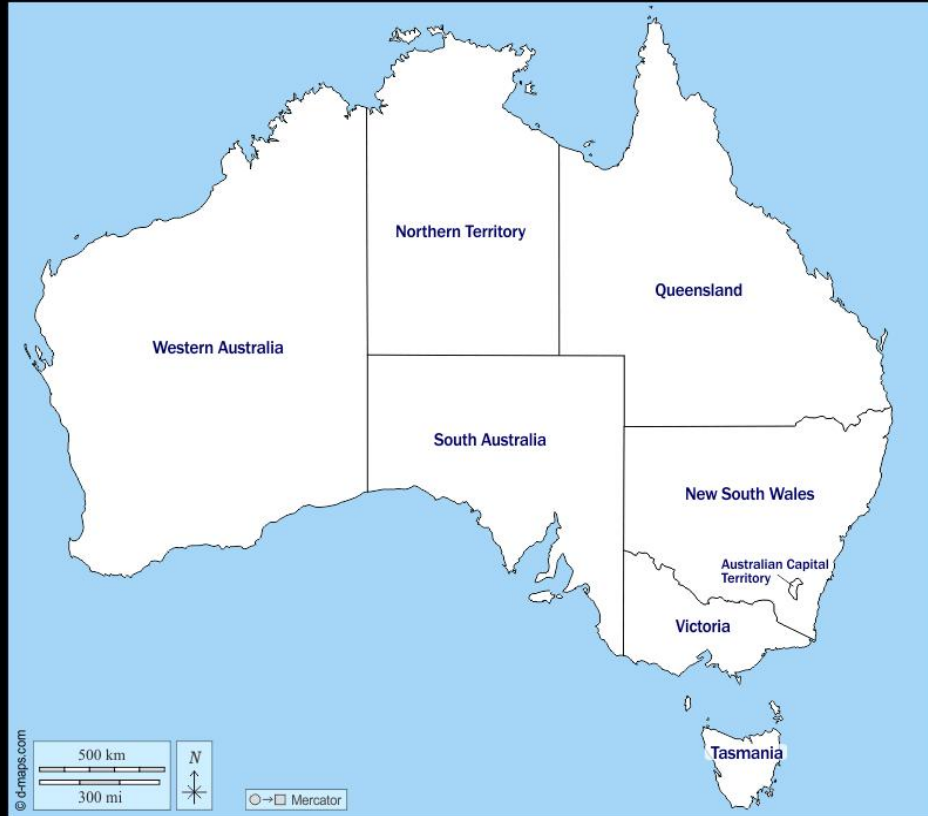
# Constraint Satisfaction Problems

# ACKNOWLEDGEMENTS

These slides are only for teaching purposes. The material has been taken from slides of Prof. Mausam (IITD), Prof. Rohan Paul (IITD), Prof. Brian Yu (Harvard University) and other researchers who are working in the field of AI.

# Motivating Example

Suppose you are given the map of australia and we want to color it in such a way that no two neighbours have the same color.





# Constraint Satisfaction Problems

- A constraint satisfaction problem consists of three components,  $X$ ,  $D$  and  $C$ :
  - $X$  is a set of variables  $\{X_1, X_2, \dots, X_n\}$
  - $D$  is set of domains  $\{D_1, D_2, \dots, D_n\}$ , one for each variable. Each domain  $D_i$  consists of a set of allowable values  $\{v_1, v_2, \dots, v_k\}$  for variable  $X_i$ .
  - $C$  is a set of constraints that specify combinations of values each variable can take. Each constraint  $C_i$  consists of a pair  $\langle \text{scope}, \text{rel} \rangle$  where **scope** is a tuple of variables that participate in the constraint and **rel** is a relation that defines the values that those variables can take on.
  - For example, if  $X_1$  and  $X_2$  both have the domain  $\{A, B\}$  then the constraint saying the two variables must have different values can be written as  $\langle (X_1, X_2), [(A, B), (B, A)] \rangle$  or as  $\langle (X_1, X_2), X_1 \neq X_2 \rangle$ .

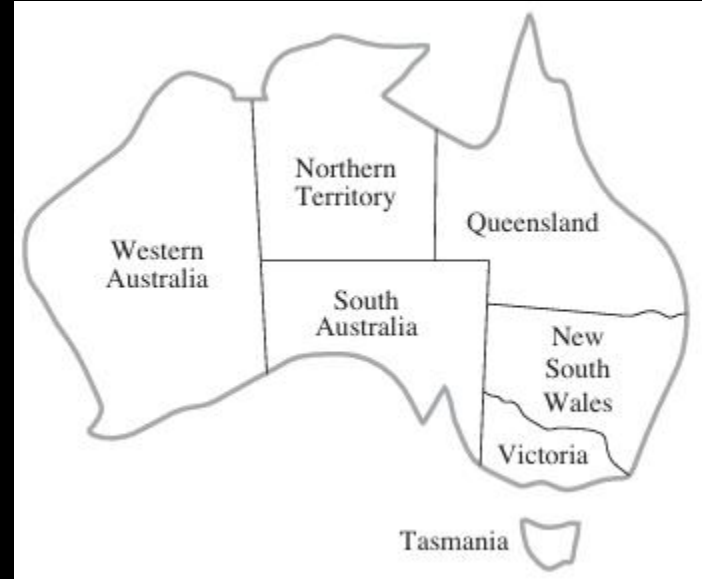
# Constraint Satisfaction Problems

- To solve a CSP, we need to define a state space and the notion of a solution.
- Each state in a CSP is defined by an assignment of values to some or all the variables,  $\{X_i = v_i, X_j = v_j, \dots\}$ .
- An assignment of values that does not violate any constraints is called a **consistent** or legal assignment.
- A **complete assignment** is one in which every variable is assigned a value.
- We call an assignment, a solution to the CSP if it's **consistent** and **complete**.
- A partial assignment is one that assigns values to only some of the variables.

## CSP Example: Map Coloring

We are given a map of Australia showing each of its states and territories. Our goal is color each region either red, green, or blue in such a way that no neighboring regions have the same color.

**How can we formulate this a CSP?**



# CSP Example: Map Coloring

To formulate this problem as a CSP, we can define variables to be the regions -

$$X = \{WA, NT, Q, NSW, V, SA, T\}$$

Domain for each variable would be  $D_i = \{red, green, blue\}$ .

Set of constraints would be -

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

Here  $SA \neq WA$  is short for  $\langle (SA, WA), SA \neq WA \rangle$  where  $SA \neq WA$  can be fully enumerated in turn as -

$$\{(red, green), (red, blue), (green, red), (green, blue), (blue, red), (blue, green)\}$$

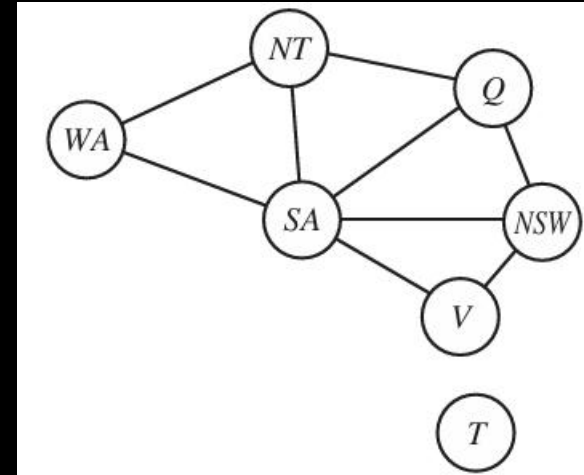


# CSP Example: Map Coloring

One possible solutions to map coloring problem is -

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

**Constraint Graph** - The nodes of the graph corresponds to variables of the problem, and a link connects any two variables that participate in a constraint.



Constraint Graph

# Why formulate a problem as a CSP?

- One reason is that the CSPs yield a natural representation for a wide variety of problems; if you already have a CSP-solving system, it is often easier to solve a problem using it than to design a custom solution using another search technique.
- In addition, CSP solvers can be faster than state-space searchers because the CSP solver can quickly eliminate large part of the search space.
- For example, once we have chosen {SA = blue} in the Australia problem, we can conclude that none of the five neighboring variables can take on the value blue. Without taking advantage of constraint propagation, a search procedure would have to consider  $3^5 = 243$  assignments for the five neighboring variables; with constraint propagation we never have to consider blue as a value, so we have only  $2^5 = 32$  assignments to look at, a reduction of 87%.

# Why formulate a problem as a CSP?

- In regular state-space search we can only ask: is this specific state a goal? No? What about this one?
- With CSPs, once we find out that a partial assignment is not a solution, we can immediately discard further refinements of the partial assignment.
- Furthermore, we can see why the assignment is not a solution—we see which variables violate a constraint—so we can focus attention on the variables that matter.
- As a result, many problems that are intractable (not solvable in finite amount of time) for regular state-space search can be solved quickly when formulated as a CSP.

## CSP Example: Job-shop scheduling

- Factories have the problem of scheduling a day's worth of jobs, subject to various constraints. In practice, many of these problems are solved with CSP techniques.
- Consider the problem of scheduling the assembly of a car. The whole job is composed of tasks, and we can model each task as a variable, where the value of each variable is the time that the task starts, expressed as an integer number of minutes.
- Constraints can assert that one task must occur before another—for example, a wheel must be installed before the hubcap is put on—and that only so many tasks can go on at once. Constraints can also specify that a task takes a certain amount of time to complete.

## CSP Example: Job-shop Scheduling

- Let's consider a small part of the car assembly, consisting of 15 tasks: install axles (front and back), affix all four wheels (right and left, front and back), tighten nuts for each wheel, affix hubcaps, and inspect final assembly.
- How can we represent above scheduling problem as CSP?
- We can represent the tasks with 15 variables  $X = \{\text{AxleF}, \text{AxleB}, \text{WheelRF}, \text{WheelLF}, \text{WheelRB}, \text{WheelLB}, \text{NutsRF}, \text{NutsLF}, \text{NutsRB}, \text{NutsLB}, \text{CapRF}, \text{CapLF}, \text{CapRB}, \text{CapLB}, \text{Inspect}\}$
- The value for each variable is the time that the task starts.

## CSP Example: Job-shop Scheduling

If a task  $T_1$  should be completed before task  $T_2$  and takes  $d_1$  unit of time, then we can write constraints in form of  $T_1 + d_1 \leq T_2$ . This type of constraints are called **precedence constraint**.

In our example, the axles have to be in place before the wheels are put on, and it takes 10 minutes to install an axle, so we write

- $AxleF + 10 \leq WheelRF$
- $AxleF + 10 \leq WheelLF$
- $AxleB + 10 \leq WheelRB$
- $AxleB + 10 \leq WheelLB$

## CSP Example: Job-shop Scheduling

Next we say that, for each wheel, we must affix the wheel (which takes 1 minute), then tighten the nuts (2 minutes), and finally attach the hubcap:

- $\text{WheelRF} + 1 \leq \text{NutsRF}$
- $\text{NutsRF} + 2 \leq \text{CapRF}$
- $\text{WheelLF} + 1 \leq \text{NutsLF}$
- $\text{NutsLF} + 2 \leq \text{CapLF}$
- $\text{WheelRB} + 1 \leq \text{NutsRB}$
- $\text{NutsRB} + 2 \leq \text{CapRB}$
- $\text{WheelLB} + 1 \leq \text{NutsLB}$
- $\text{NutsLB} + 2 \leq \text{CapLB}$

## CSP Example: Job-shop Scheduling

Suppose we have four workers to install wheels, but they have to share one tool that helps put the axle in place. We need a disjunctive constraint to say that AxleF and AxleB must not overlap in time; either one comes first or the other does:  $(\text{AxleF} + 10 \leq \text{AxleB})$  or  $(\text{AxleB} + 10 \leq \text{AxleF})$

Since inspection comes at the last and takes 3 minutes, we also need to add another constraint of form  $\mathbf{X} + \mathbf{d}_x \leq \mathbf{Inspect}$  for each variable.

Finally, suppose we have to complete all assembly in 30 mins then, we can limit the domain of all variables to -

$D_i = \{1, 2, 3, \dots, 27\}$  as last 3 minutes should be for inspection in worst case.



# Variations on the CSP Formalism

CSPs can be categorized based on types of domains -

- **CSPs with variables that have discrete, finite domains.**
  - Map Coloring problem
  - Job Scheduling problem
  - 8-queens problem
  - Sudoku
- **CSPs with variables that have discrete but infinite domains.**
  - Job Scheduling problem if we do not put any constraints on time limit of assembly
- **CSPs with variables that have continuous domains.**
  - Linear programming problems
  - The scheduling of experiments on the Hubble Space Telescope requires very precise timing of observations; the start and finish of each observation and maneuver are continuous-valued variables.

# Types of Constraints

- **Unary Constraints**

- Unary constraint involves only single variable. It is used to restrict taken by some variable.
- For example, In case of map coloring problem if South Australians won't tolerate the color green; then we can represent it using unary constraint  $\langle (SA), SA \neq \text{green} \rangle$

- **Binary Constraints**

- Binary constraint involves two variables. A binary CSP is one with only binary constraints.
- For example,  $SA \neq NSW$  is a binary constraint.

- Higher-order constraints involve 3 or more variables.

- **Global Constraints**

- A constraint involving an arbitrary number of variables is called a global constraint. Global constraints need not to involve all the variables.
- One of the most common global constraints is *Alldiff*, which says that all of the variables involved in the constraint must have different values.
- In Sudoku problems, all variables in a row or column must satisfy an *Alldiff* constraint.

# CSP Example: Exam Scheduling

Student:



Taking classes:

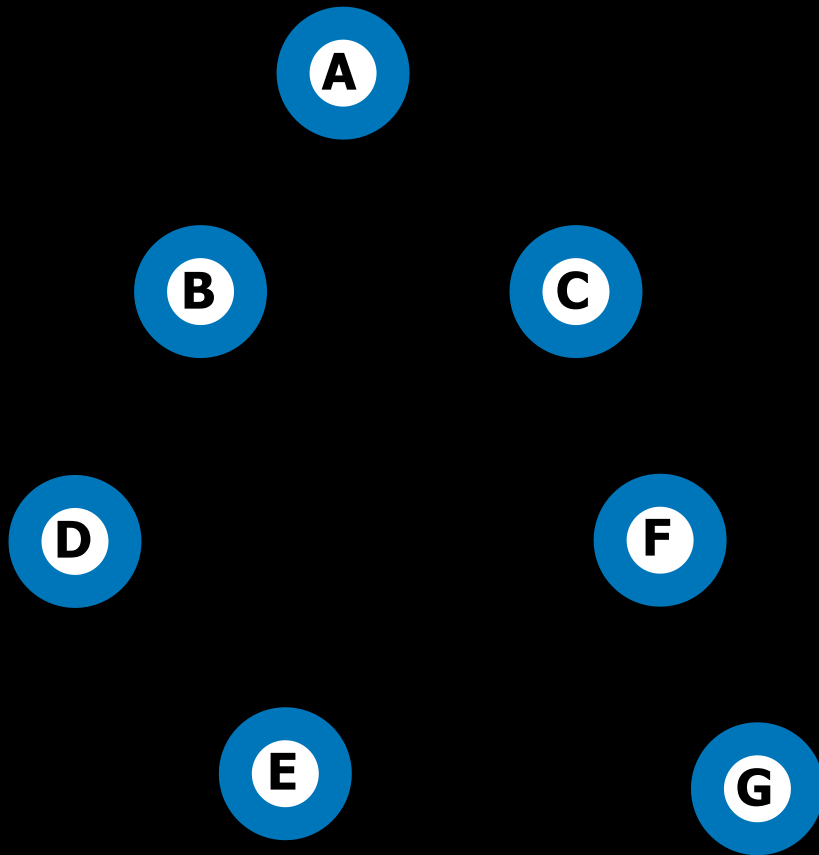
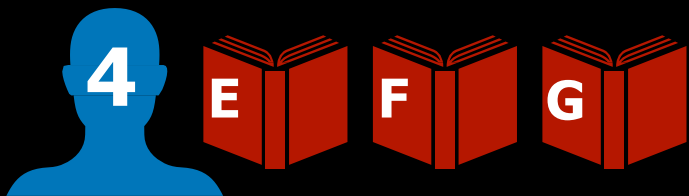
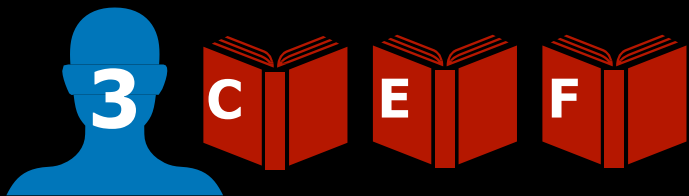
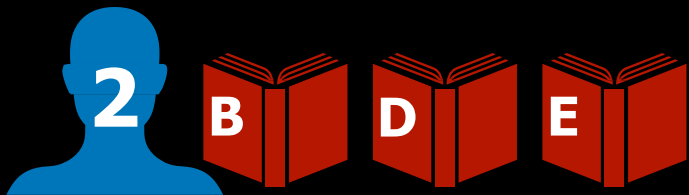
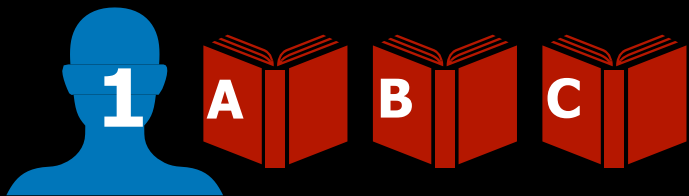


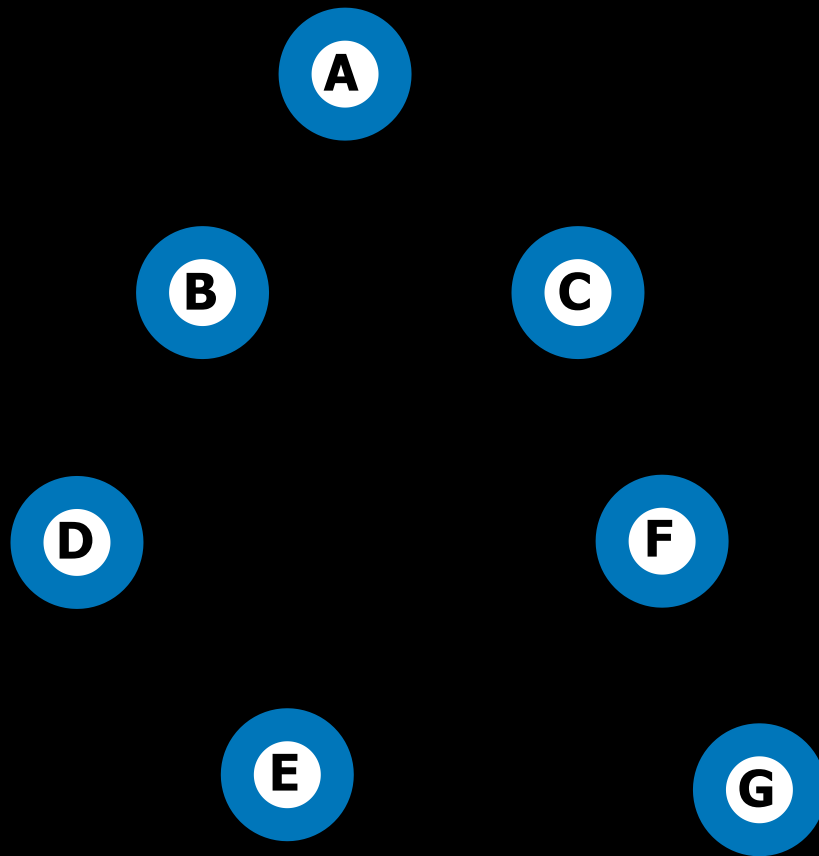
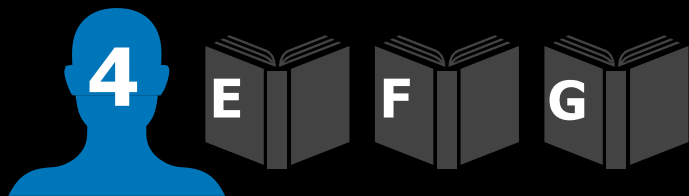
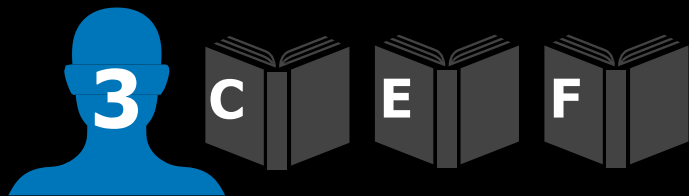
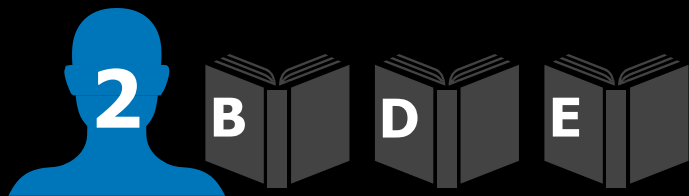
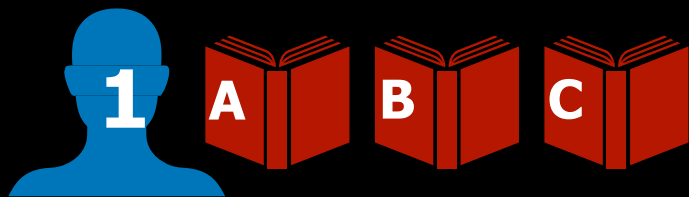
Exam slots:

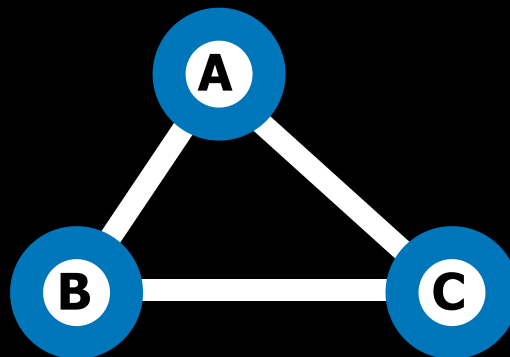
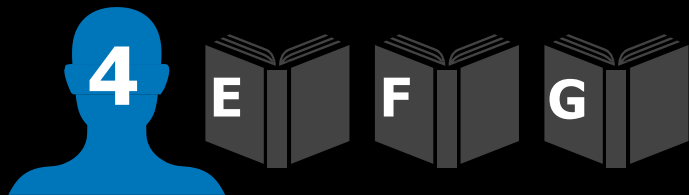
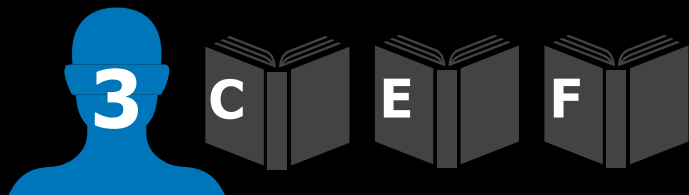
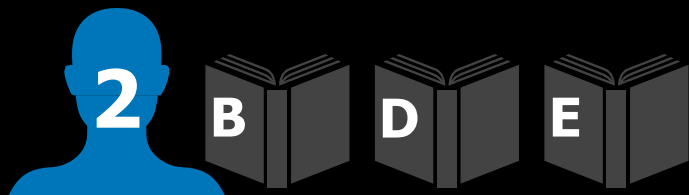
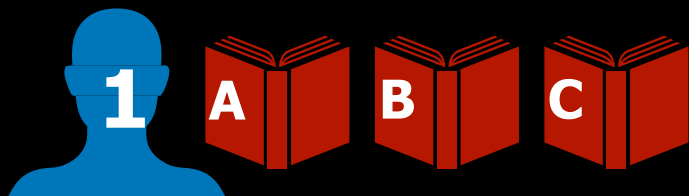
Monday

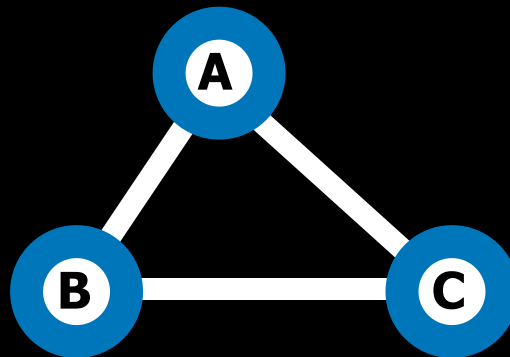
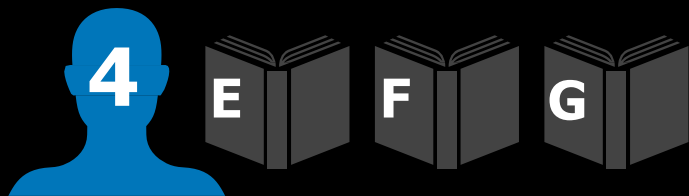
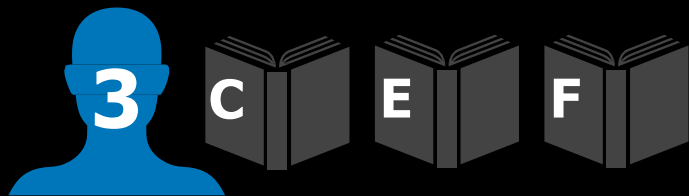
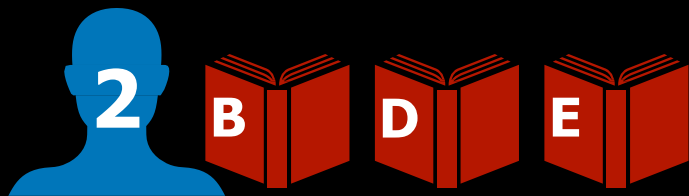
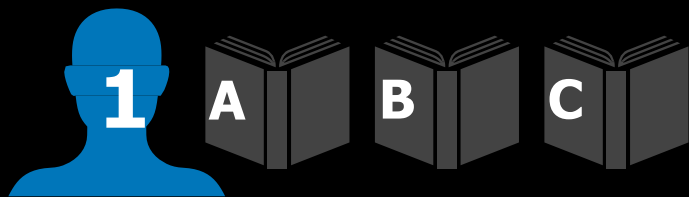
Tuesday

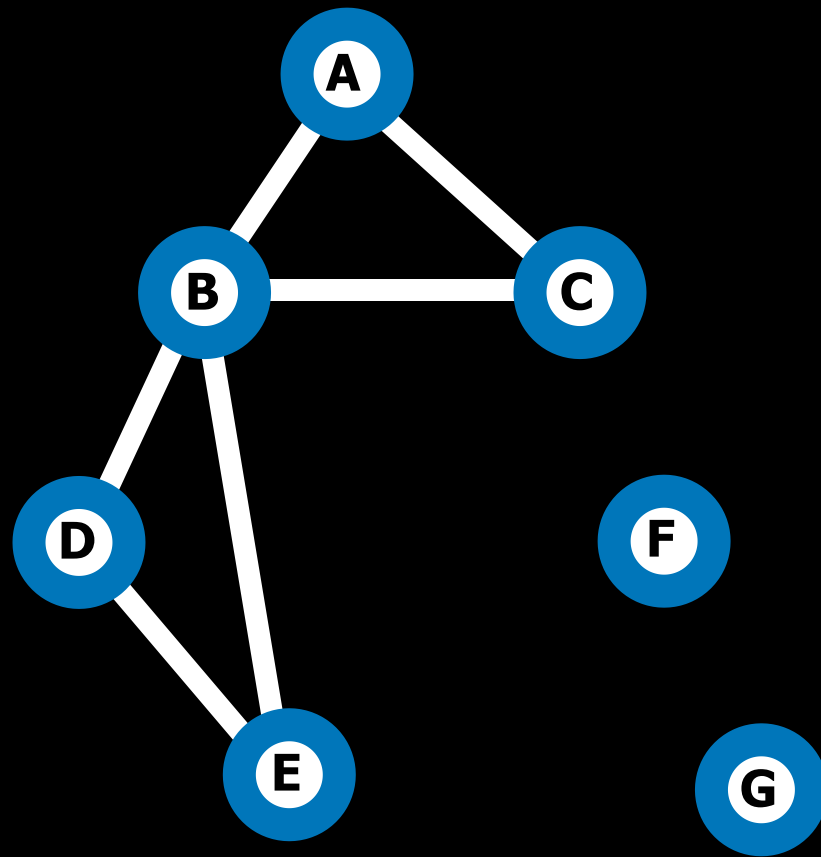
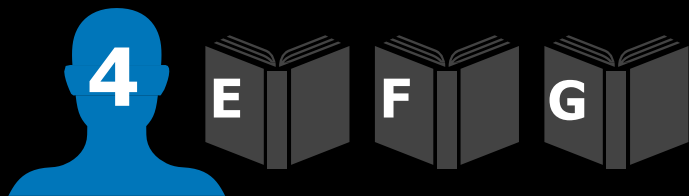
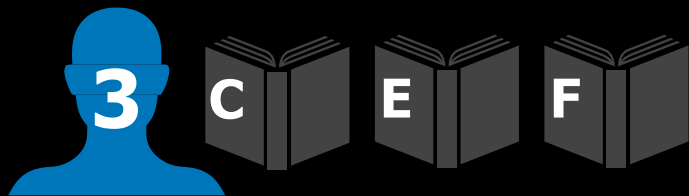
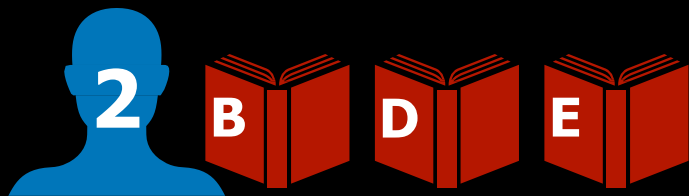
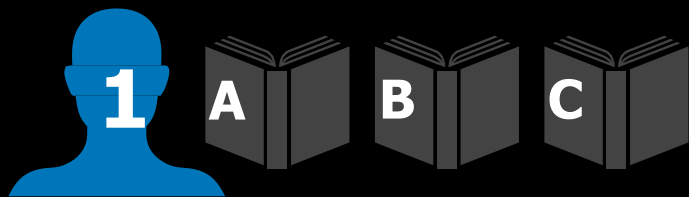
Wednesday



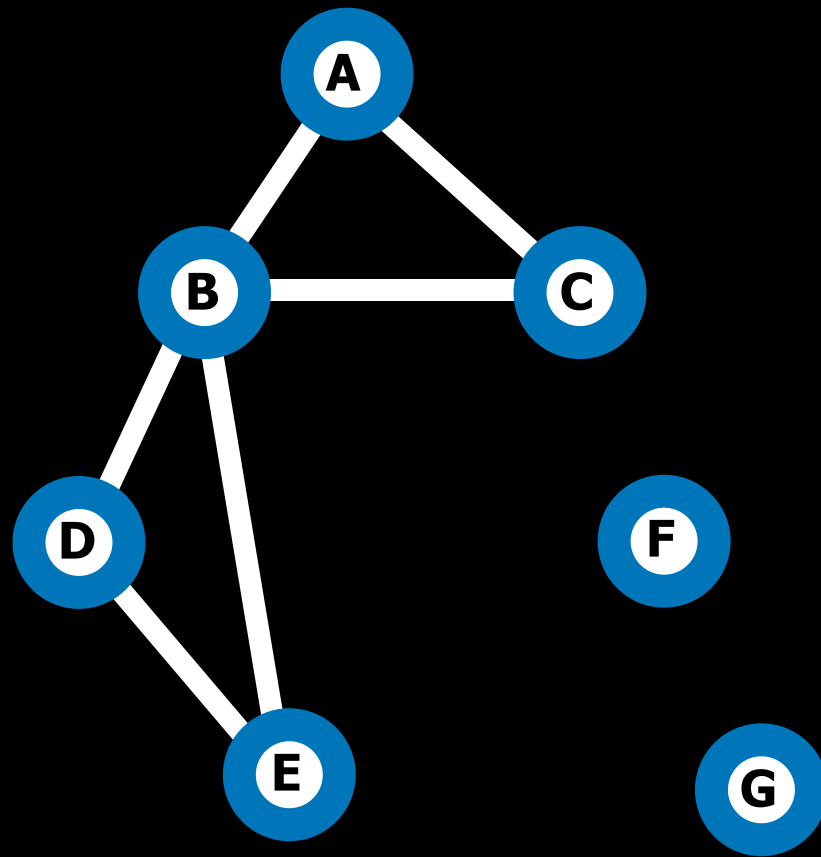
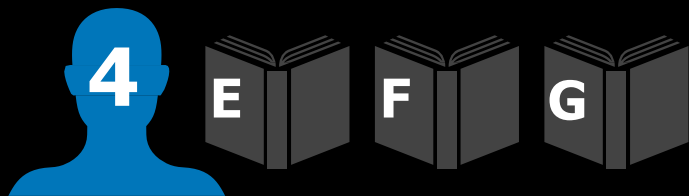
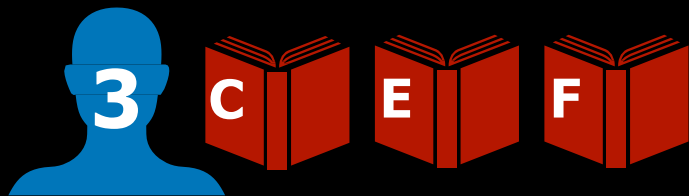
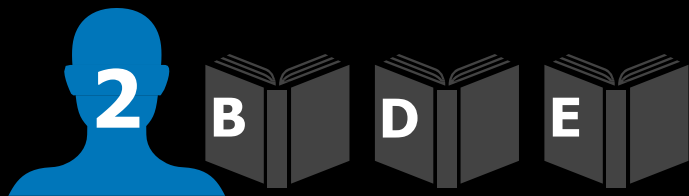
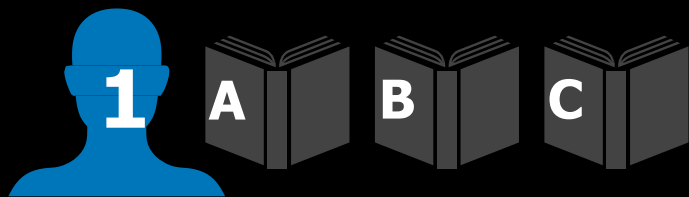


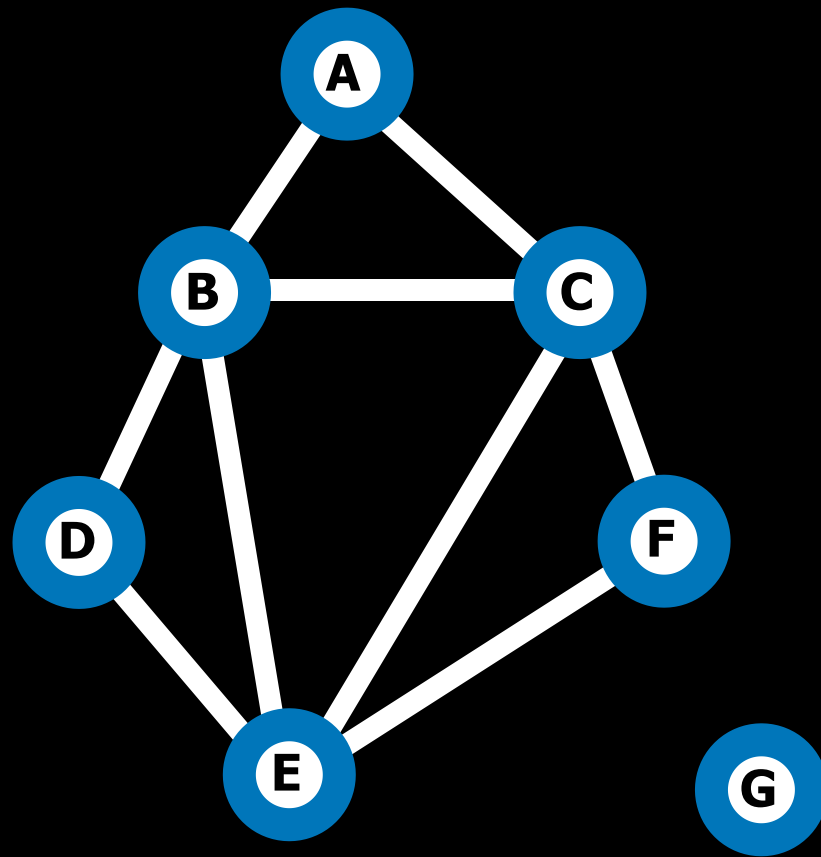
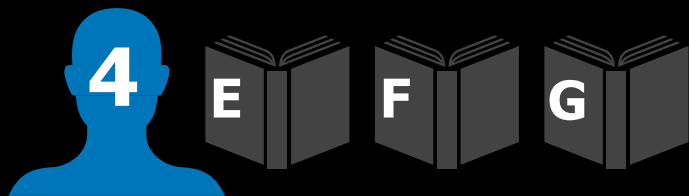
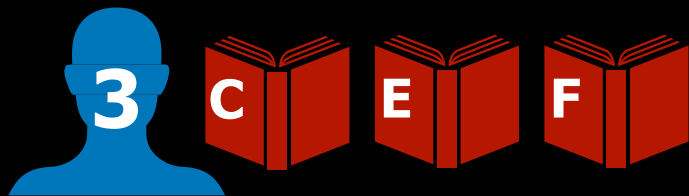
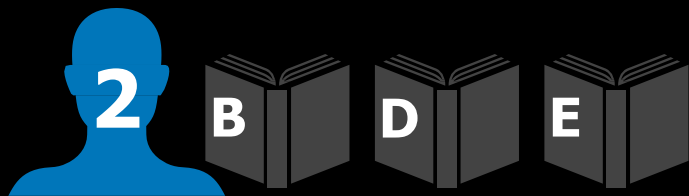
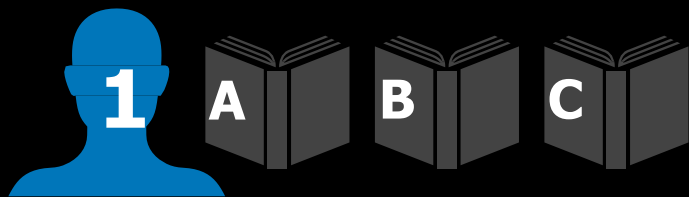


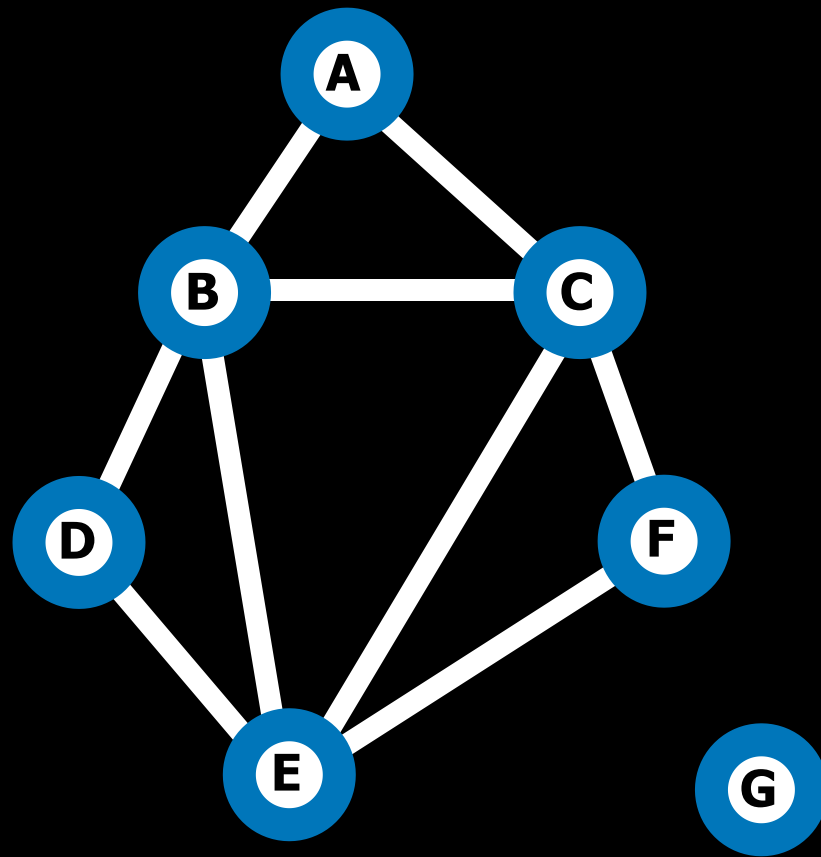
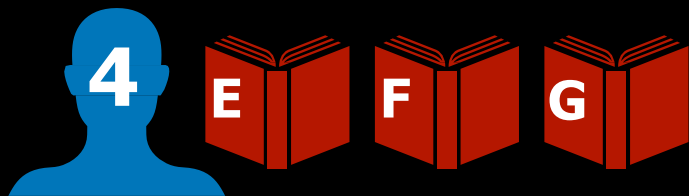
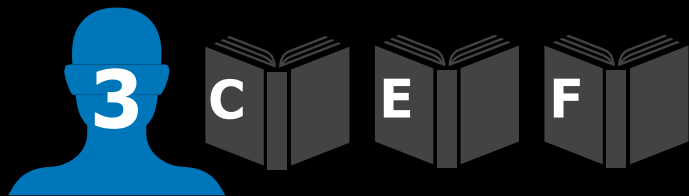
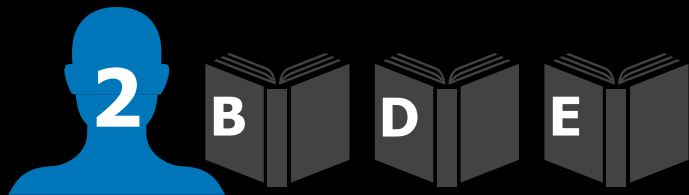
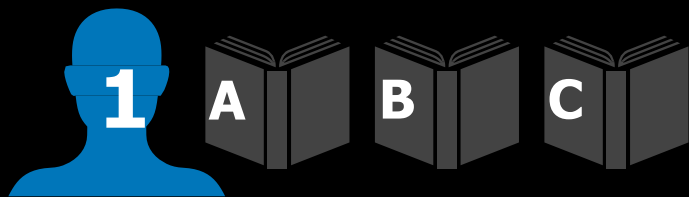


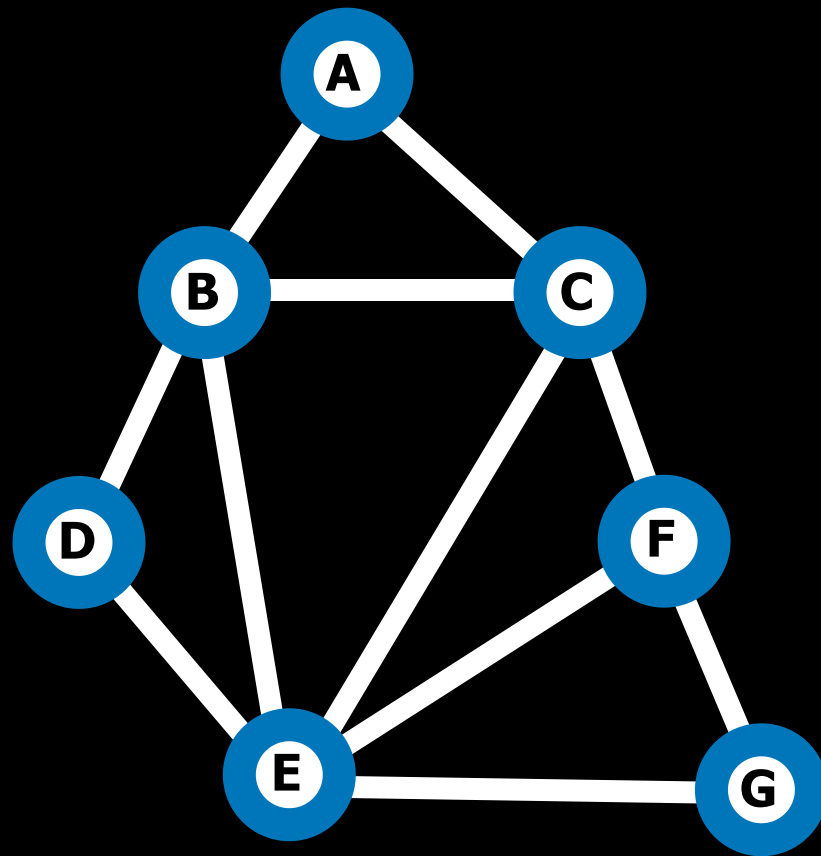
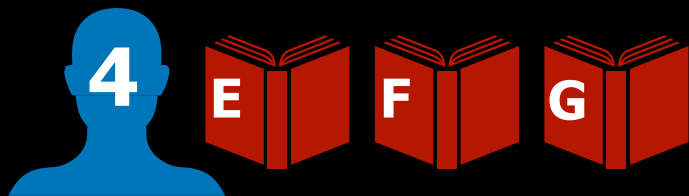
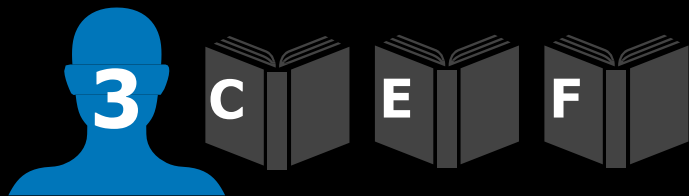
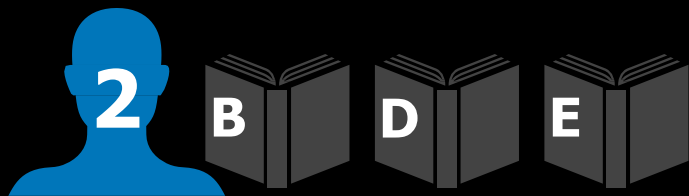
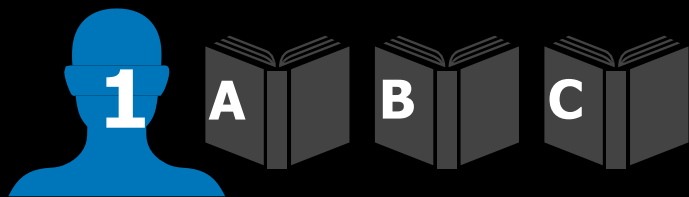


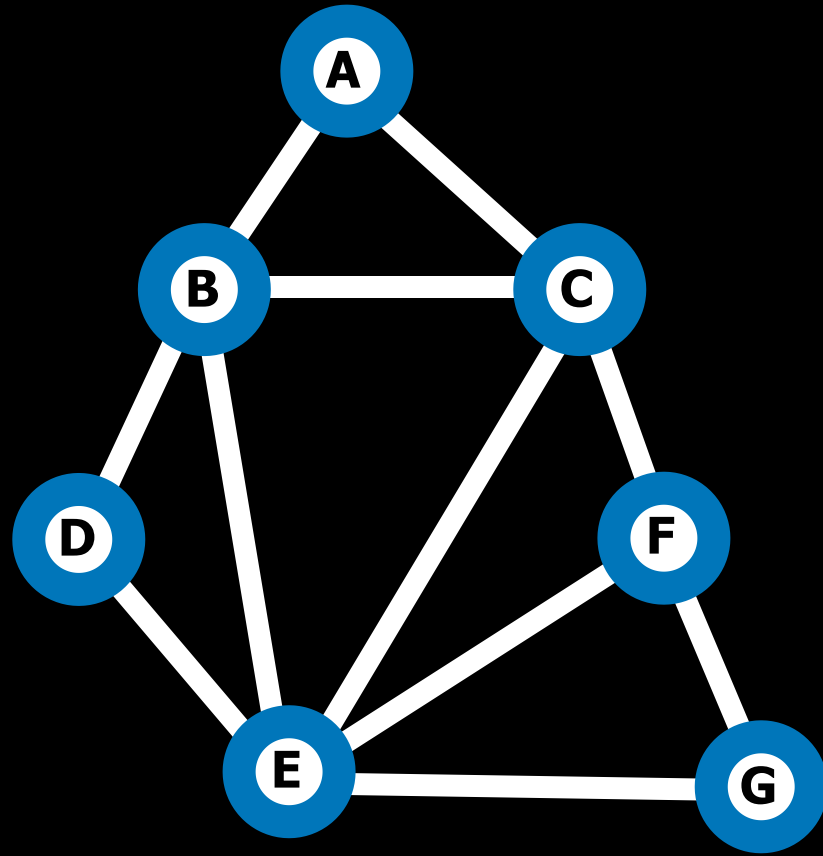


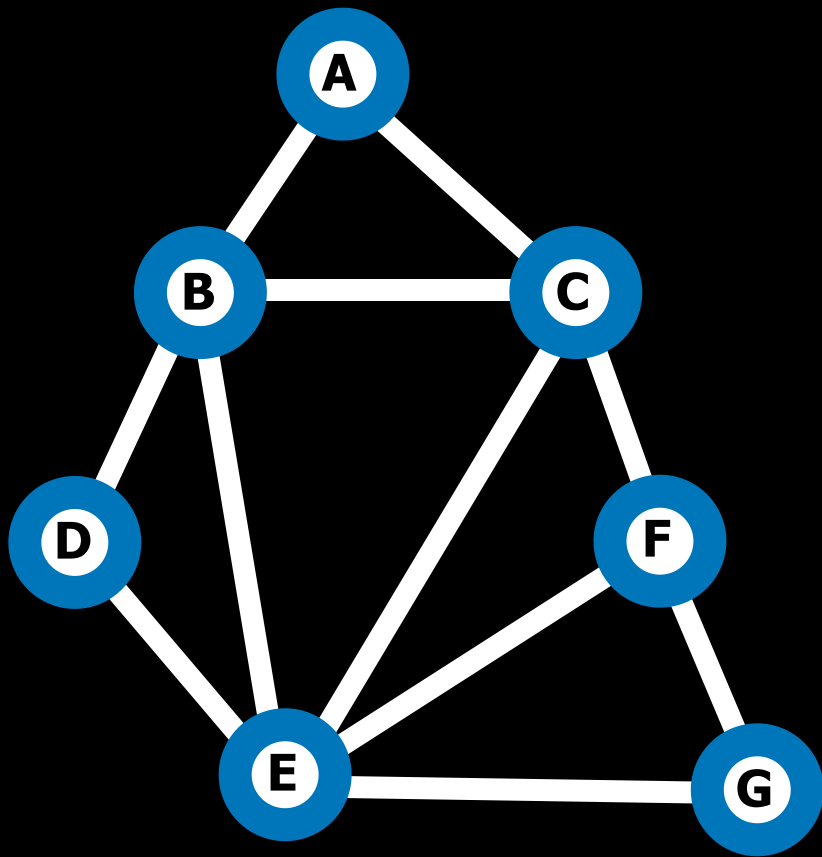












## Variables

$\{A, B, C, D, E, F, G\}$

## Domains

$\{Monday, Tuesday, Wednesday\}$

for each variable

## Constraints

$\{A \neq B, A \neq C, B \neq C, B \neq D, B \neq E,$   
 $C \neq E, C \neq F, D \neq E, E \neq F, E \neq G,$   
 $F \neq G\}$

# Cryptarithmic Puzzles

Each letter in a cryptarithmic puzzle represents a different digit.

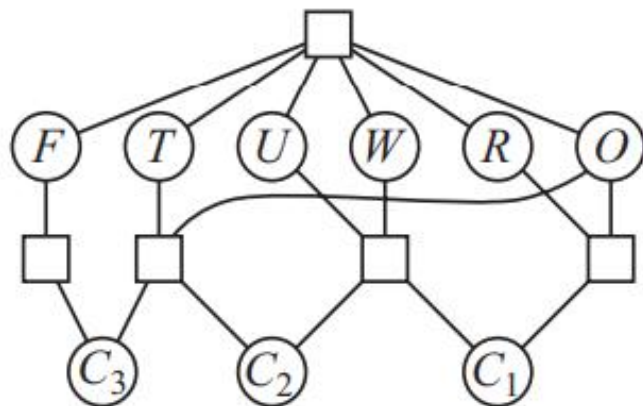
For this example, above constraint can be represented as the global constraint  $Alldiff(F, T, U, W, R, O)$ .

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

# Cryptarithmic Puzzles

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

(a)



(b)

**Figure 6.2** (a) A cryptarithmic problem. Each letter stands for a distinct digit; the aim is to find a substitution of digits for letters such that the resulting sum is arithmetically correct, with the added restriction that no leading zeroes are allowed. (b) The constraint hypergraph for the cryptarithmic problem, showing the *Alldiff* constraint (square box at the top) as well as the column addition constraints (four square boxes in the middle). The variables  $C_1$ ,  $C_2$ , and  $C_3$  represent the carry digits for the three columns.



# Cryptarithmic Puzzles

The addition constraints on the four columns of the puzzle can be written as the following n-ary constraints -

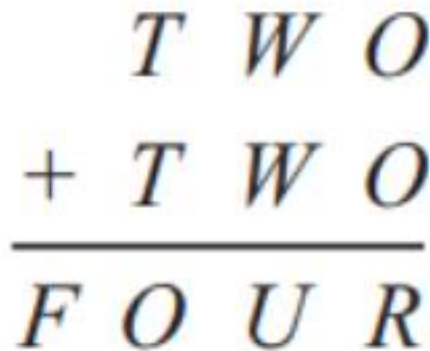
$$\mathbf{O} + \mathbf{O} = \mathbf{R} + 10 \cdot \mathbf{C}_{10}$$

$$\mathbf{C}_{10} + \mathbf{W} + \mathbf{W} = \mathbf{U} + 10 \cdot \mathbf{C}_{100}$$

$$\mathbf{C}_{100} + \mathbf{T} + \mathbf{T} = \mathbf{O} + 10 \cdot \mathbf{C}_{1000}$$

$$\mathbf{C}_{1000} = \mathbf{F}$$

where  $\mathbf{C}_{10}$ ,  $\mathbf{C}_{100}$ , and  $\mathbf{C}_{1000}$  are auxiliary variables representing the digit carried over into ten, hundreds or thousands column.


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

# Constraint Propagation: Inference in CSPs

In state-space graph search, search algorithm can only explore the graph and find a solution.

But in CSPs, search algorithm can perform following tasks -

- Search for variable assignment which satisfies all the constraints.
- Do a special type of inference called constraint propagation.

**Constraint Propagation** - The process of constraints to reduce the no of legal values for a variable, which in turn can reduce the legal values for another variable.

Constraint propagation may be intertwined with search, or it may be done as a preprocessing step, before search starts.

Sometimes this preprocessing can solve the whole problem, so no search is required at all.

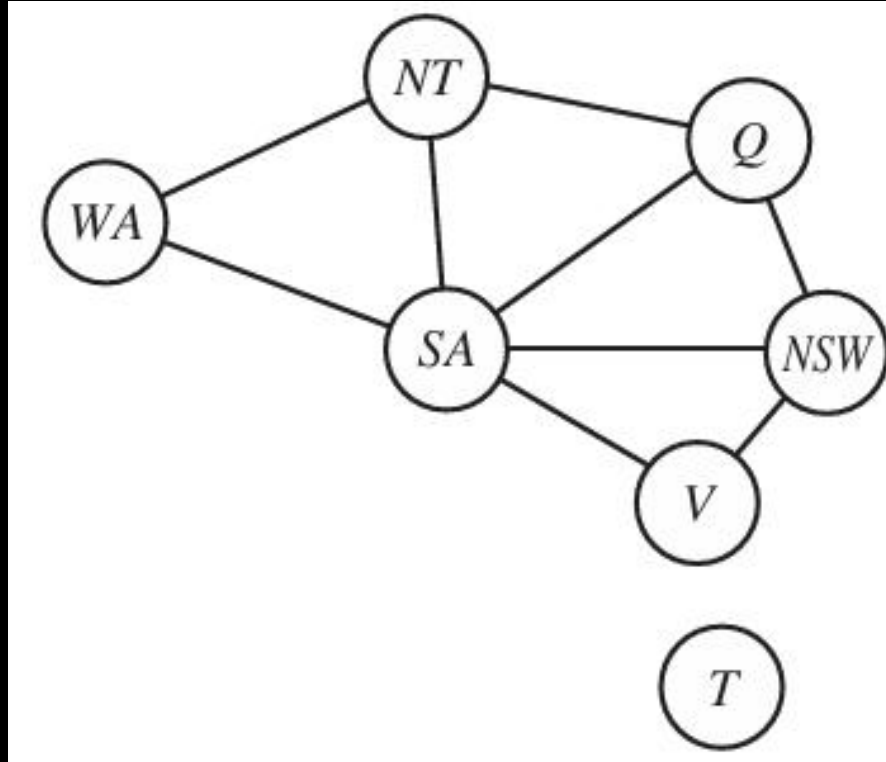
# Local Consistency

The key idea behind constraint progression is that if we represent each variable as a node in a graph and each binary constraint as an arc then the process of enforcing local consistency in each part of the graph causes inconsistent values to be eliminated throughout the graph.

There are different types of consistency -

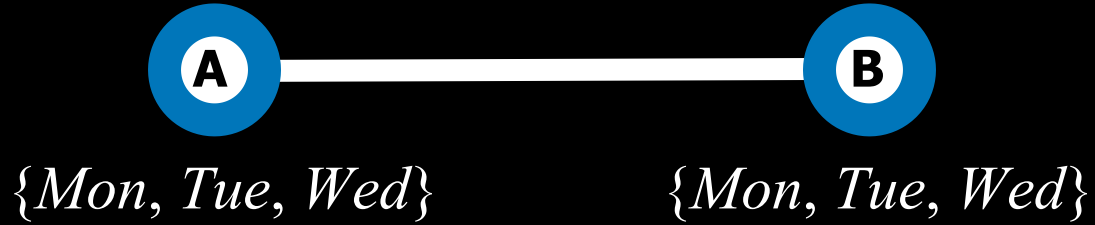
- Node Consistency
- Arc Consistency
- Path Consistency
- k-consistency

# Local Consistency

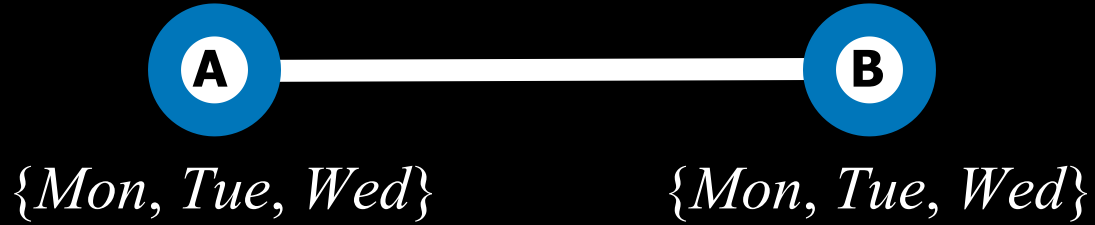


# node consistency

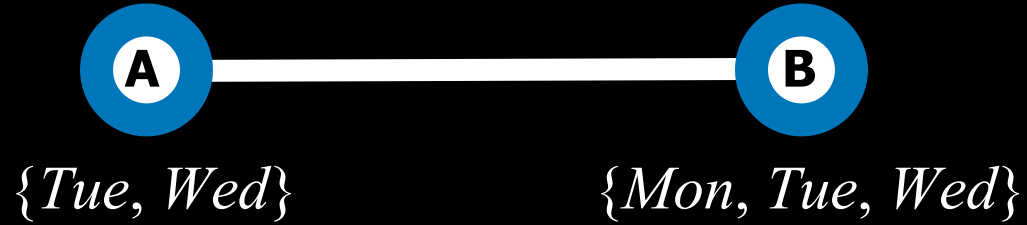
when all the values in a variable's domain satisfy the variable's unary constraints then the variable is node-consistent.



*{A ≠ Mon, B ≠ Tue, B ≠ Mon, A ≠ B}*

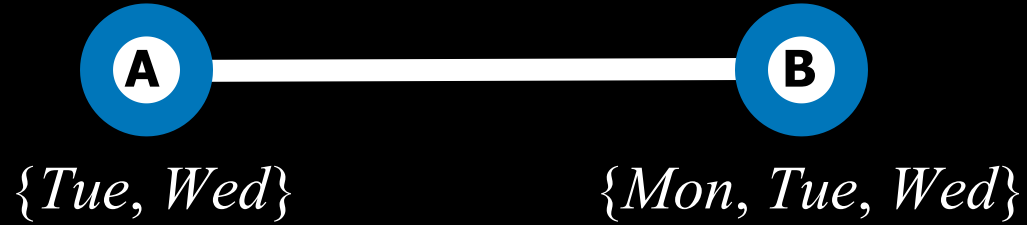


$\{A \neq Mon, B \neq Tue, B \neq Mon, A \neq B\}$



$$\{A \neq Mon, B \neq Tue, B \neq Mon, A \neq B\}$$

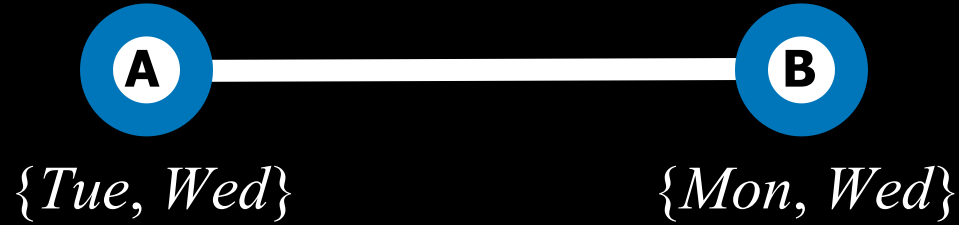




$$\{A \neq Mon, B \neq Tue, B \neq Mon, A \neq B\}$$



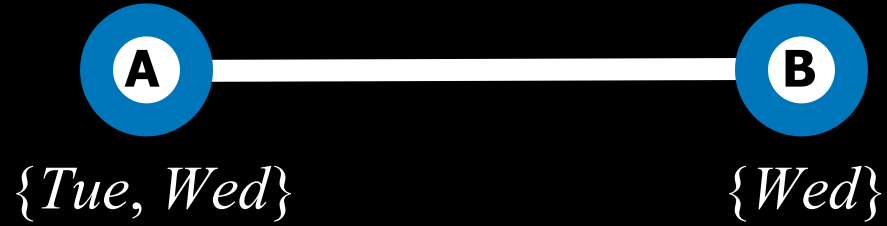
$$\{A \neq Mon, B \neq Tue, B \neq Mon, A \neq B\}$$



$\{A \neq Mon, B \neq Tue, B \neq Mon, A \neq B\}$



$\{A \neq Mon, B \neq Tue, B \neq Mon, A \neq B\}$



$\{A \neq Mon, B \neq Tue, B \neq Mon, A \neq B\}$

# arc consistency

when all the values in a variable's domain  
satisfy the variable's binary constraints

# arc consistency

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  $Y$



$\{A \neq Mon, B \neq Tue, B \neq Mon, A \neq B\}$

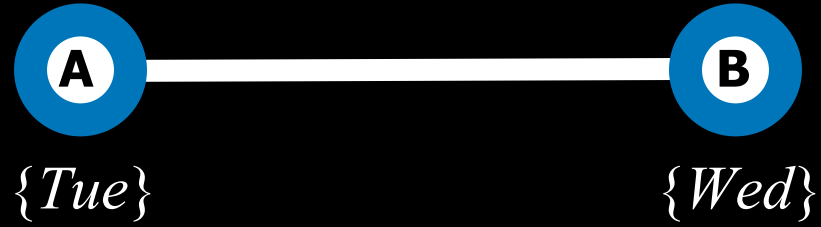




$\{A \neq Mon, B \neq Tue, B \neq Mon, A \neq B\}$



$\{A \neq Mon, B \neq Tue, B \neq Mon, A \neq B\}$



$\{A \neq Mon, B \neq Tue, B \neq Mon, A \neq B\}$

## Arc Consistency: Example

- Consider the constraint  $Y = X^2$  where the domain of both  $X$  and  $Y$  is the set of digits.
- Do the following -
  - Write the constraint in detailed form.
  - Make variable  $X$  arc-consistent with  $Y$ .
  - Make  $Y$  arc-consistent with  $X$ .

# Arc Consistency

**function** AC-3(*csp*) **returns** false if an inconsistency is found and true otherwise

**inputs:** *csp*, a binary CSP with components  $(X, D, C)$

**local variables:** *queue*, a queue of arcs, initially all the arcs in *csp*

**while** *queue* is not empty **do**

$(X_i, X_j) \leftarrow \text{REMOVE-FIRST}(\text{queue})$

**if** REVISE(*csp*,  $X_i, X_j$ ) **then**

**if** size of  $D_i = 0$  **then return** false

**for each**  $X_k$  **in**  $X_i.\text{NEIGHBORS} - \{X_j\}$  **do**

            add  $(X_k, X_i)$  to *queue*

**return** true

---

**function** REVISE(*csp*,  $X_i, X_j$ ) **returns** true iff we revise the domain of  $X_i$

*revised*  $\leftarrow$  false

**for each**  $x$  **in**  $D_i$  **do**

**if** no value  $y$  in  $D_j$  allows  $(x, y)$  to satisfy the constraint between  $X_i$  and  $X_j$  **then**

            delete  $x$  from  $D_i$

*revised*  $\leftarrow$  true

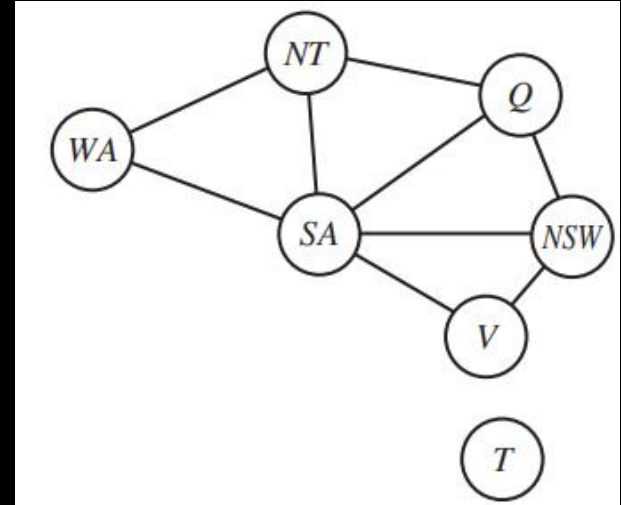
**return** *revised*

**Figure 6.3** The arc-consistency algorithm AC-3. After applying AC-3, either every arc is arc-consistent, or some variable has an empty domain, indicating that the CSP cannot be solved. The name “AC-3” was used by the algorithm’s inventor (Mackworth, 1977) because it’s the third version developed in the paper.

## Arc Consistency: Limitation

Arc consistency can go a long way toward reducing the domains of variables, sometimes finding a solution (by reducing every domain to size 1) and sometimes finding that the CSP cannot be solved (by reducing some domain to size 0). But for other networks, arc consistency fails to make enough inferences.

For example, for the map-coloring problem on Australia but only with two colors allowed red and blue. Arc consistency can do nothing because every variable is already arc consistent.



# Path Consistency

- Path consistency tightens the binary constraints by using implicit constraints that are inferred by looking at triples of variables.
- A two-variable set  $\{X_i, X_j\}$  is path-consistent with respect to a third variable  $X_m$  if, for every assignment  $\{X_i = a, X_j = b\}$  consistent with the constraints on  $\{X_i, X_j\}$ , there is an assignment to  $X_m$  that satisfies the constraints on  $\{X_i, X_m\}$  and  $\{X_m, X_j\}$ . This is called path consistency because one can think of it as looking at a path from  $X_i$  to  $X_j$  with  $X_m$  in the middle.

# K-consistency

Stronger forms of propagation can be defined with the notion  $k$ -consistency. A CSP is *k-consistent* if, for any set of  $k-1$  variables and for any consistent assignment to those variables, a consistent value can always be assigned to any  $k$ th variable.

1-consistency  $\rightarrow$  node consistency

2-consistency  $\rightarrow$  arc consistency

3-consistency  $\rightarrow$  path consistency

...



## CSP Example: Sudoku

	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		

(a)

	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

(b)

**Figure 6.4** (a) A Sudoku puzzle and (b) its solution.

# CSP Example: Sudoku

Set of variables - Each cell of sudoku can be considered as a variable

$$X = \{A_1, A_2, \dots, I_8, I_9\}$$

$$\text{Domain } D = \{1, 2, \dots, 9\}$$

Constraints - Sudoku constraints can be defined using global constraint *Alldiff*

Alldiff constraints: one for each row, column, and box of 9 squares.

$$\text{Alldiff}(A_1, A_2, A_3, \dots, A_9) \quad \text{Alldiff}(A_1, B_1, C_1, \dots, I_1)$$

$$\text{Alldiff}(B_1, B_2, \dots, B_9) \quad \text{Alldiff}(A_2, B_2, C_2, \dots, I_2)$$

# 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

# Backtracking Search

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

function BACKTRACK(assignment, csp) returns a solution, or failure
  if assignment is complete then return assignment
  var  $\leftarrow$  SELECT-UNASSIGNED-VARIABLE(csp)
  for each value in ORDER-DOMAIN-VALUES(var, assignment, csp) do
    if value is consistent with assignment then
      add {var = value} to assignment
      inferences  $\leftarrow$  INFERENCE(csp, var, value)
      if inferences  $\neq$  failure then
        add inferences to assignment
        result  $\leftarrow$  BACKTRACK(assignment, csp)
        if result  $\neq$  failure then
          return result
      remove {var = value} and inferences from assignment
  return failure
```

**Figure 6.5** A simple backtracking algorithm for constraint satisfaction problems. The algorithm is modeled on the recursive depth-first search of Chapter 3. By varying the functions SELECT-UNASSIGNED-VARIABLE and ORDER-DOMAIN-VALUES, we can implement the general-purpose heuristics discussed in the text. The function INFERENCE can optionally be used to impose arc-, path-, or  $k$ -consistency, as desired. If a value choice leads to failure (noticed either by INFERENCE or by BACKTRACK), then value assignments (including those made by INFERENCE) are removed from the current assignment and a new value is tried.

