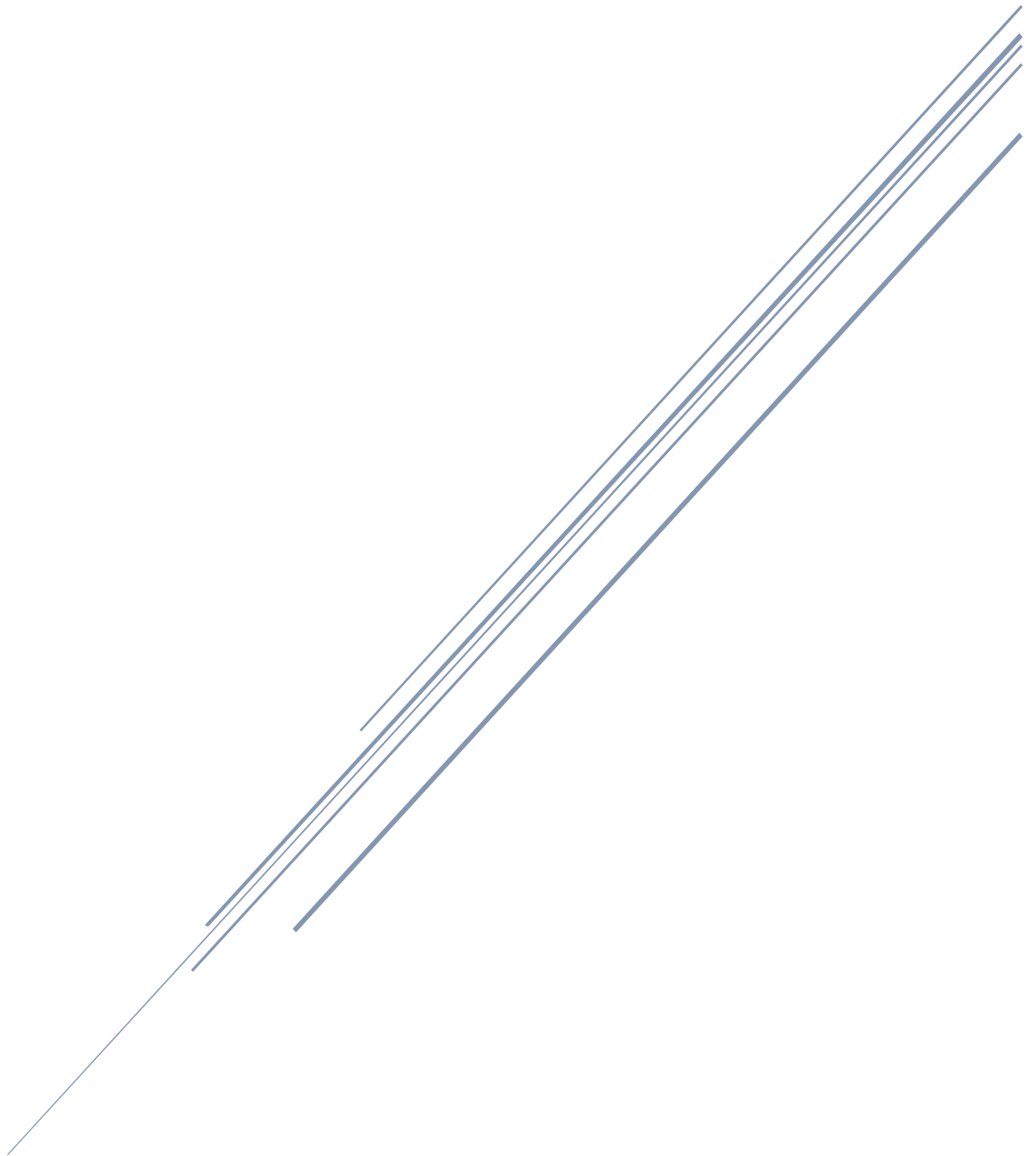


CHAPTER 3

Constraint Satisfaction Problems



3.1 Introduction

Chapters 1 and 2 delve into the notion that problems can be addressed through exploration within a space of states. These states are subject to assessment through an evaluation function and examination to determine if they qualify as goal or optimal states. However, from the perspective of the search algorithm, each state is atomic or indivisible—a black box with no internal structure.

This chapter describes a way to solve a wide variety of problems more efficiently. We use a factored representation for each state: a set of variables, each of which has a value. A problem is solved when each variable has a value that satisfies all the constraints on the variable. A problem described this way is called a constraint satisfaction problem, or CSP.

CSP search algorithms leverage the structure of states, employing general-purpose heuristics instead of problem-specific ones to address complex problems. The primary strategy involves efficiently eliminating substantial portions of the search space at once by identifying variable/value combinations that violate constraints.

3.2 Constraint Satisfaction Problems

3.2.1 Definition

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 a set of domains, $\{D_1, D_2, \dots, D_n\}$, one for each variable.

C is a set of constraints, $\{C_1, C_2, \dots, C_m\}$, that specifies allowable combinations of values.

Each domain D_i consists of a set of allowable values, $\{v_1, v_2, \dots, v_k\}$ for variable X_i . Each constraint C_i consists of a pair $\langle \text{scope}, \text{rel} \rangle$, where the *scope* is a tuple of variables participating in the constraint, and *rel* is a relation defining the values those variables can take on. A relation can be represented as an explicit list of all tuples of values that satisfy the constraint, or as an abstract relation. 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 $\langle (X_1, X_2), [(a, b), (b, a)] \rangle$ (which is the list of all tuples of values that satisfy the constraint) or $\langle (X_1, X_2), X_1 \neq X_2 \rangle$ (which is an abstract relation).

To solve a CSP, it is essential to define a state space and the notion of a solution:

- In a CSP, each **state** is determined by an assignment of values to some or all of the variables, denoted as $\{X_i = v_i, X_j = v_j, \dots\}$.
- A **partial assignment** is one that assigns values to only some of the variables.
- On the other hand, a **complete assignment** is one in which every variable is assigned.
- An assignment that adheres to all constraints is referred to as a **consistent assignment**.
- A **solution** to a CSP is a consistent, complete assignment.

3.2.2 Types of constraints

Constraints in Constraint Satisfaction Problems (CSPs) define the relationships and limitations between variables. There are several types of constraints in CSPs, and they can be broadly categorized into:

- a) **Unary Constraints:** The most basic type is the unary constraint, which restricts the possible values of a single variable. For example, $X > 5$, where X is a variable.

- b) **Binary Constraints:** Involve relationships between two variables. For example, in the constraint $X \neq Y$, where X and Y are variables. A binary CSP is one that consists only of binary constraints, and it can be represented as a constraint graph.
- c) **Global Constraints:** A constraint that involves an arbitrary number of variables is termed a global constraint. One of the most common global constraints is *Alldiff*, which stipulates that all variables involved in the constraint must have different values.
- d) **Preference Constraints:** The constraints we've discussed so far are absolute, meaning breaking them rules out a potential solution. In real-world CSPs, there are also preference constraints indicating preferred solutions. For instance, in a university class-scheduling problem, professors can't teach two classes simultaneously (absolute constraint), but some may prefer certain time slots. For example, Prof. X might prefer teaching in the morning. A schedule that has Prof. X teaching at 2 p.m. would still be an allowable solution but would not be an optimal one. These preferences can be represented as costs. Thus, assigning an afternoon slot for Prof. X may cost 2 points against the objective function, while a morning slot costs 1.

The combination and nature of constraints depend on the specific problem being modeled as a CSP. These types illustrate the flexibility of CSPs in representing a wide range of problems in various domains.

3.2.3 Examples

Example 1: Map coloring

We are examining a map of Australia that displays each of its states and territories (Figure 3.1(a)). Our goal is to color each region as red, green, or blue in a manner that ensures neighboring regions do not share the same color.

To formulate this as a Constraint Satisfaction Problem (CSP), we define the variables as the regions: $X = \{WA, NT, Q, NSW, V, SA, T\}$. The domain of each variable is the set $D_i = \{\text{red, green, blue}\}$. The constraints require neighboring regions to have distinct colors. Since there are nine places where regions border, there are nine constraints: $C = \{ \langle (SA, WA), SA \neq WA \rangle, \langle (SA, NT), SA \neq NT \rangle, \langle (SA, Q), SA \neq Q \rangle, \langle (SA, NSW), SA \neq NSW \rangle, \langle (SA, V), SA \neq V \rangle, \langle (WA, NT), WA \neq NT \rangle, \langle (NT, Q), NT \neq Q \rangle, \langle (Q, NSW), Q \neq NSW \rangle, \langle (NSW, V), NSW \neq V \rangle \}$.

For instance, consider the constraint $\langle (SA, WA), SA \neq WA \rangle$. The conditions $SA \neq WA$ can be exhaustively enumerated as [(red, green), (red, blue), (green, red), (green, blue), (blue, red), (blue, green)].

Figure 3.1(b) depicts the CSP as a constraint graph, where nodes correspond to variables in the problem, and links connect any two variables participating in a constraint.

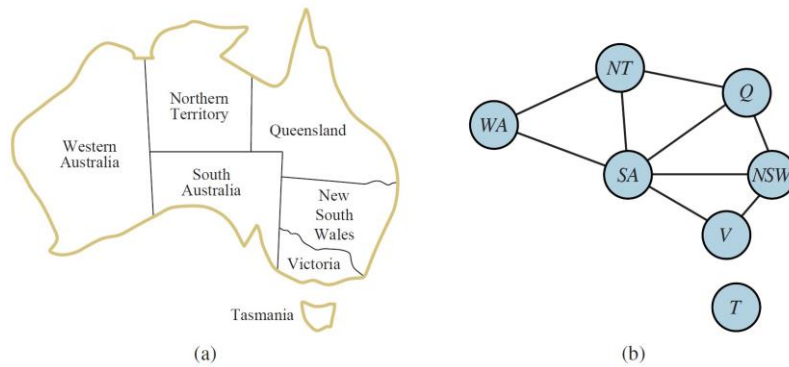


Figure 3.1 Australia map coloring

There are numerous potential solutions to this problem, for example: {WA = red, NT = green, Q = red, NSW = green, V = red, SA = blue, T = red}.



Figure 3.2 Potential solution for Australia map coloring

Example 2: Crossword puzzle

We are given the following crossword puzzle (see Figure 3.3):

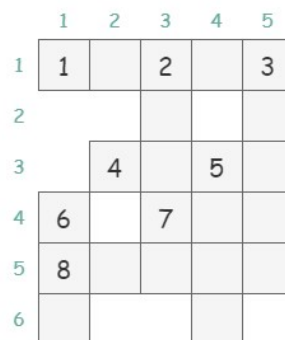


Figure 3.3 Crossword puzzle

The task is to solve this puzzle by incorporating the provided list of words: AFT, ALE, EEL, HEEL, HIKE, HOSES, KEEL, KNOT, LASER, LEE, LINE, SAILS, SHEET, STEER, TIE. The numbers 1, 2, 3, 4, 5, 6, 7, 8 in the crossword puzzle indicate the starting positions for the words.

The variables denote the position of each word in the puzzle, as outlined in Table 3.1. The domain of each variable is the list of words that could serve as its value. For instance, variable 1ACROSS necessitates words with five letters, 2DOWN requires words with five letters, 3DOWN requires words

with the same length, and so forth.

Table 3.1 Variables and domains for the crossword puzzle problem

Variable	Starting cell	Domain
1ACROSS	1	{HOSES, LASER, SAILS, SHEET, STEER}
4ACROSS	4	{HEEL, HIKE, KEEL, KNOT, LINE}
7ACROSS	7	{AFT, ALE, EEL, LEE, TIE}
8ACROSS	8	{HOSES, LASER, SAILS, SHEET, STEER}
2DOWN	2	{HOSES, LASER, SAILS, SHEET, STEER}
3DOWN	3	{HOSES, LASER, SAILS, SHEET, STEER}
5DOWN	5	{HEEL, HIKE, KEEL, KNOT, LINE}
6DOWN	6	{AFT, ALE, EEL, LEE, TIE}

There are 12 constraints. For example, in the first constraint the third letter of 1ACROSS must be equal to the first letter of 2DOWN.

$C = \{ \langle (1ACROSS, 2DOWN), 1ACROSS[3]=2DOWN[1] \rangle ,$
 $\langle (1ACROSS, 3DOWN), 1ACROSS[5]=3DOWN[1] \rangle ,$
 $\langle (4ACROSS, 2DOWN), 4ACROSS[2] = 2DOWN[3] \rangle ,$
 $\langle (4ACROSS, 5DOWN), 4ACROSS[3] = 5DOWN[1] \rangle ,$
 $\langle (4ACROSS, 3DOWN), 4ACROSS[4] = 3DOWN[3] \rangle ,$
 $\langle (7ACROSS, 2DOWN), 7ACROSS[1] = 2DOWN[4] \rangle ,$
 $\langle (7ACROSS, 5DOWN), 7ACROSS[2] = 5DOWN[2] \rangle ,$
 $\langle (7ACROSS, 3DOWN), 7ACROSS[3] = 3DOWN[4] \rangle ,$
 $\langle (8ACROSS, 6DOWN), 8ACROSS[1] = 6DOWN[2] \rangle ,$
 $\langle (8ACROSS, 2DOWN), 8ACROSS[3] = 2DOWN[5] \rangle ,$
 $\langle (8ACROSS, 5DOWN), 8ACROSS[4] = 5DOWN[3] \rangle ,$
 $\langle (8ACROSS, 3DOWN), 8ACROSS[5] = 3DOWN[5] \rangle \}$

The solution for the crossword puzzle problem is {1ACROSS = HOSES, 4ACROSS = HIKE, 7ACROSS = LEE, 8ACROSS = LASER, 2DOWN = SAILS, 3DOWN = STEER, 5DOWN = KEEL, 6DOWN = ALE}, as shown in Figure 3.4:

	1	2	3	4	5
1	H	O	S	E	S
2			A		T
3		H	I	K	E
4	A		L	E	E
5	L	A	S	E	R
6	E			L	

Figure 3.4 Solution for crossword puzzle

Example 3: Sudoku puzzle

A Sudoku board is comprised of 81 squares, with some squares initially filled with digits from 1 to 9. The objective is to fill in all the remaining squares in a way that ensures no digit appears twice in any row, column, or 3×3 box (see Figure 3.5). A row, column, or box is referred to as a unit.

A Sudoku puzzle can be considered a CSP with 81 variables, one for each square. We use the variable names A1 through A9 for the top row (left to right), down to I1 through I9 for the bottom row. The empty squares have the domain $\{1, 2, 3, 4, 5, 6, 7, 8, 9\}$, and the prefilled squares have a domain consisting of a single value. Additionally, there are 27 different constraints, representing the three types of units: rows, columns, and 3×3 boxes.

	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		

Figure 3.5 Sudoku puzzle

The Sudoku puzzle enforces *Alldiff* constraints, ensuring that each row, column, and 3×3 box of 9 squares contains unique values. For example:

- *Alldiff*(A1, A2, A3, A4, A5, A6, A7, A8, A9)
- *Alldiff*(B1, B2, B3, B4, B5, B6, B7, B8, B9)
- ...
- *Alldiff*(A1, B1, C1, D1, E1, F1, G1, H1, I1)
- *Alldiff*(A2, B2, C2, D2, E2, F2, G2, H2, I2)
- ...
- *Alldiff*(A1, A2, A3, B1, B2, B3, C1, C2, C3)
- *Alldiff*(A4, A5, A6, B4, B5, B6, C4, C5, C6)
- ...

A solution for the Sudoku puzzle is shown in Figure 3.6.

	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

Figure 3.6 Solution for Sudoku puzzle

3.3 Search algorithms for CSPs

3.3.1 Backtracking Search

The backtracking search refers to a Depth-First Search (DFS) that selects values for one variable at a time and backtracks when a variable has no legal values left for assignment. The algorithm, illustrated

in Figure 3.7, iteratively chooses an unassigned variable and then explores all values in the domain of that variable in sequence, aiming to find a solution. If an inconsistency is detected, the backtracking algorithm returns failure, prompting the previous call to attempt another value.

This can be achieved by a DFS on a special kind of state space, where states are defined by the values assigned so far:

- ✧ **Initial state:** the empty assignment.
- ✧ **Successor function:** assign a value to an unassigned variable that does not conflict with previously assigned values of other variables. (If no legal values remain, the successor function fails.)
- ✧ **Goal test:** all variables have been assigned a value, and no constraints have been violated.

Input:

- assignment: Initially an empty state
- csp: Data structure containing all variables, domains, and constraints of the CSP

Output: Solution or None

Function BACKTRACKING(assignment, csp)

Begin

```
// Check if the assignment is complete
if (complete(assignment)) then
    return assignment // In backtracking, a complete assignment is a consistent one => solution found
end if
var <- selectUnassignedVariable(assignment, csp.variables) // Select an unassigned variable
// Test all possible values for the variable var
for val in orderDomainValues(var, csp.domains) do
    if (consistent(var, val, assignment, csp.constraints)) then
        // Assign the given value to the variable in the current assignment
        assignment.add(var, val)
        result <- BACKTRACKING(assignment, csp)
        if (result ≠ None) then
            return result
        end if
        // Remove the assignment of the variable in the current assignment
        assignment.remove(var, value)
    end if
end for
return None
```

End

Figure 3.7 Backtracking Search algorithm

The backtracking algorithm contains the function **selectUnassignedVariable(assignment)**. The simplest strategy for this step is to choose the next unassigned variable in order, typically denoted as $\{X_1, X_2, X_3, \dots, X_n\}$. Similarly, for the function **orderDomainValues(var)**, it returns all possible values that the variable can take according to its domain. For example, if the domain of X_1 is $\{v_1, v_2, v_3, \dots, v_m\}$, this function will return these values one by one.

Example 1: We are given variables A, B, and C, each with a domain of values {1, 2, 3}. What are the possible values for these variables while satisfying the following constraints:

C1: $A > B$, **C2:** $B \neq C$, **C3:** $A \neq C$.

Propose a solution for this CSP using backtracking search algorithm.

Solution: Table 3.2 and Figure 3.8 show the solution of example 1 using Backtracking Search algorithm.

Table 3.2 Solving the CSP in example 1 using Backtracking Search algorithm

Assignment	Constraint satisfaction			Action
	C1	C2	C3	
{ }				Select the variable A
{A=1}	Yes	Yes	Yes	Select the variable B
{A=1, B=1}	No	Yes	Yes	Select another value for B
{A=1, B=2}	No	Yes	Yes	Select another value for B
{A=1, B=3}	No	Yes	Yes	Back to A
{A=2}	Yes	Yes	Yes	Select the variable B
{A=2, B=1}	Yes	Yes	Yes	Select the variable C
{A=2, B=1, C=1}	Yes	No	Yes	Select another value for C
{A=2, B=1, C=2}	Yes	Yes	No	Select another value for C
{A=2, B=1, C=3}	Yes	Yes	Yes	Assignment complete

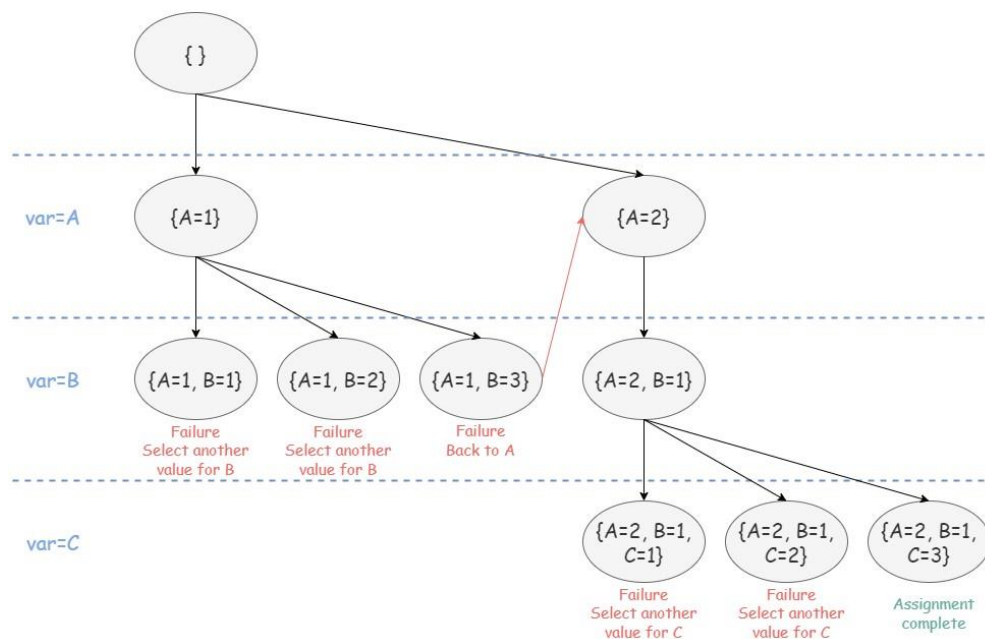


Figure 3.8 Backtracking Search tree for example 1

3.3.2 Improving the Backtracking Search

We can improve the efficiency of solving CSPs using the Backtracking Search by incorporating sophistication to address the following questions:

- Which variable should be assigned next **selectUnassignedVariable(assignment)**, and in what order should its values be tried **orderDomainValues(var)**?
- Can the search detect failure in advance?

3.3.2.1 Selecting unassigned variables

Heuristics for selecting the next unassigned variable include the following:

✧ Minimum-Remaining-Values (MRV) heuristic

This heuristic involves selecting the variable with the fewest legal remaining values in its domain. It is also known as the "fail-first" heuristic. This term reflects its tendency to choose a variable that is likely to cause a failure sooner, effectively pruning the search tree.

If a variable X_i has no legal values left, the MRV heuristic selects X_i , leading to the immediate detection of failure. This helps avoid unnecessary searches through other variables.

The MRV heuristic typically outperforms random or static orderings, sometimes by a factor of 1,000 or more. However, its effectiveness can vary depending on the specific problem at hand.

Example 2: The application of the MRV heuristic on the Australia Map Coloring problem is illustrated in Figure 3.9.

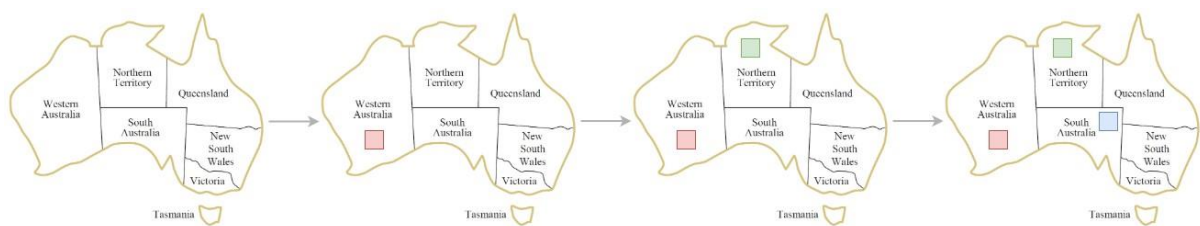


Figure 3.9 Example of application of MRV heuristic on the Australia Map Coloring problem

✧ Degree heuristic

The MRV heuristic is not effective in choosing the first variable when initially every variable has the same number of values. In such cases, the Degree heuristic becomes useful. It aims to select the variable involved in the largest number of constraints with other unassigned variables.

The Minimum-Remaining-Values heuristic is usually a more powerful guide, but the Degree heuristic can be useful as a tie-breaker.

Example 3: The application of the Degree heuristic on the Australia Map Coloring problem is illustrated in Figure 3.10.

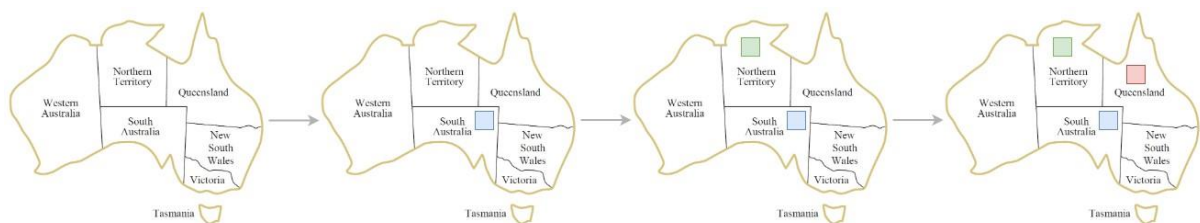


Figure 3.10 Example of application of Degree heuristic on the Australia Map Coloring problem

3.3.2.2 Ordering domain values

Once a variable has been selected, the algorithm must decide on the order in which to examine its values. Heuristics for ordering the values of a selected variable include the following:

✧ Least-Constraining-Value (LCV) heuristic

Select the value that excludes the fewest values (or leaves the greatest number of values) for neighboring unassigned variables. In essence, the heuristic aims to preserve maximum flexibility for subsequent variable assignments.

Example 4: The application of the LCV heuristic on the Australia Map Coloring problem is illustrated in Figure 3.11. In this example, the variables are selected in a static order.



Figure 3.11 Example of application of LCV heuristic on the Australia Map Coloring problem

3.3.2.3 Inference heuristic

The inference heuristic suggests that each time we make a choice for a variable's value, we have a brand-new opportunity to derive new domain reductions for the adjacent variables. One of the simplest forms of inference is called Forward-Checking. Whenever a variable X_i is assigned, a Forward-Checking process is established: for each unassigned variable X_j that is connected to X_i by a constraint, delete from X_j 's domain any value that is inconsistent with the value chosen for X_i .

Example 5: The application of the Forward-Checking on the map coloring problem is illustrated in Figure 3.12. In this example the variables are selected randomly.

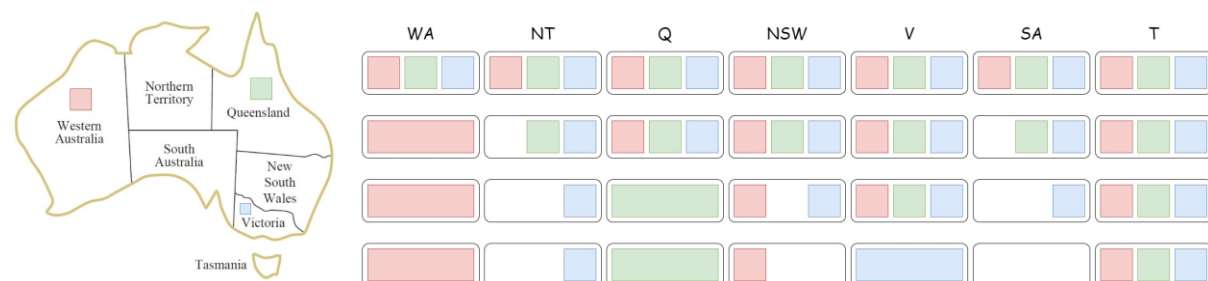


Figure 3.12 Example of the application of the Forward-Checking heuristic on the Australia Map Coloring problem

Figure 3.13 shows the Backtracking Search algorithm after integrating the Inference Heuristic.

Input:

- assignment: Initially an empty state
- csp: Data structure containing all variables, domains, and constraints of the CSP

Output: Solution or None**Function BACKTRACKING_INFERENCE(assignment, csp)****Begin**

// Check if the assignment is complete

if (complete(assignment)) **then**

return assignment // In backtracking, a complete assignment is a consistent one => solution found

end if

var <- **selectUnassignedVariable**(assignment, csp.variables) // Select an unassigned variable

// Test all possible values for the variable var

for val **in** **orderDomainValues**(var, csp.domains) **do**

if (consistent(var, val, assignment, csp.constraints)) **then**

 // Assign the given value to the variable in the current assignment

 assignment.add(var, val)

 // Update the csp.domains

 inference <- **inferenceHeuristic**(var, val, csp.domains)

 // If there is no empty domain after inference

if (inference is True) **then**

 result <- **BACKTRACKING_INFERENCE**(assignment, csp)

if (result ≠ None) **then**

return result

end if

end if

 // Remove the assignment of the variable in the current assignment

 assignment.remove(var, value)

 // Restore the domains of neighboring variables to their original values

 restore(csp.domains)

end if

end for

return None

End

Figure 3.13 Backtracking Search algorithm with Inference Heuristic

3.4 Inference in CSPs: Constraint Propagation

In regular state-space search, algorithms can only search. But in Constraint Satisfaction Problems (CSPs), algorithms have a choice: they can either search for a new variable assignment or perform a specific type of inference known as Constraint Propagation. Constraint propagation uses constraints to narrow down the possible values for a variable, and this reduction in legal values for one variable can subsequently impact the allowable values for another variable, creating a cascading effect. Constraint propagation can be part of the search or done before it starts as a preprocessing. Sometimes, just doing this preprocessing can solve the whole problem without needing to search at all.

The main idea behind constraint propagation is local consistency. If we consider each variable as a node

in a graph and each binary constraint as an arc, ensuring local consistency in each part of the graph results in the removal of inconsistent values throughout the entire graph. One of the most commonly employed forms of local consistency is arc consistency.

3.4.1 Arc Consistency

A variable in a Constraint Satisfaction Problem (CSP) is considered arc-consistent if, for every value in its current domain, there exists at least one value in the domain of another variable such that the binary constraint between them is satisfied. More formally, a variable X_i is arc-consistent with respect to another variable X_j if, for every value in the current domain D_i of X_i , there is at least one value in the domain D_j of X_j that satisfies the binary constraint on the arc (X_i, X_j) . Therefore, for arc consistency, we interpret each undirected edge, indicating the binary constraint in the constraint graph for a CSP as two directed edges, each pointing in opposite directions. Each of these directed edges is referred to as an arc (see Figure 3.14).

A CSP is considered arc-consistent if each variable in the network is arc-consistent with every other variable.

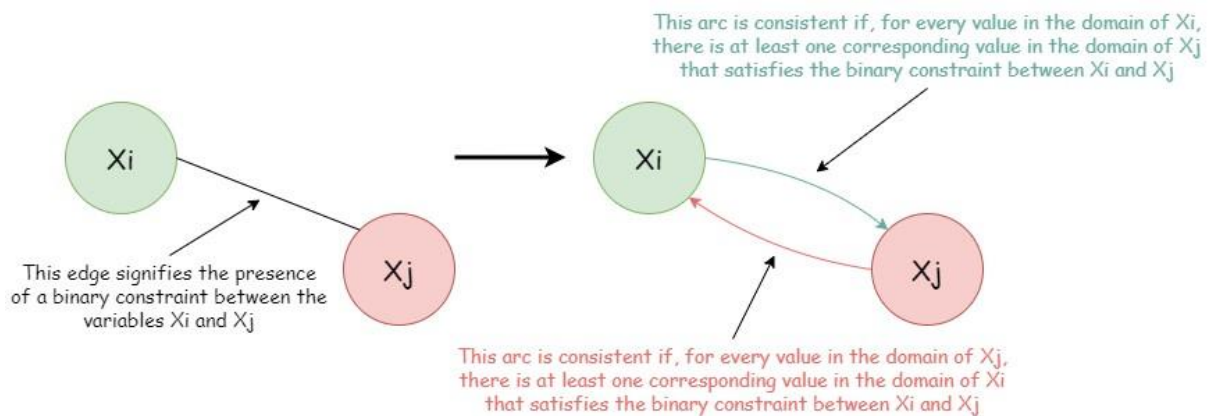


Figure 3.14 Arc Consistency

Example 6: Consider a CSP composed of the variables X_i and X_j , where the domain of both X_i and X_j is the set of digits, i.e., $D_i = D_j = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$, and the constraint $X_j = X_i^2$. We can express this constraint explicitly as: $\langle (X_i, X_j), \{(0, 0), (1, 1), (2, 4), (3, 9)\} \rangle$.

To make X_i arc-consistent with respect to X_j , we reduce X_i 's domain to $\{0, 1, 2, 3\}$. If we also ensure that X_j is arc-consistent with respect to X_i , then X_j 's domain becomes $\{0, 1, 4, 9\}$, and the entire CSP achieves arc-consistency.

3.4.1.1 AC-3 algorithm

The most popular algorithm for enforcing arc consistency is AC-3 (see Figure 3.15). To achieve arc consistency for every variable, the AC-3 algorithm maintains a queue of arcs to consider called *Agenda*. Initially, the queue contains all the arcs in the CSP. AC-3 then dequeues an arc (X_i, X_j) and makes X_i arc-consistent with respect to X_j , resulting in three possible outcomes:

- ✧ If this operation leaves D_i unchanged, the algorithm proceeds to the next arc.
- ✧ However, if D_i is revised (resulting in a smaller domain), the algorithm adds to the queue all arcs (X_k, X_i) where X_k is a neighbor of X_i (i.e., X_k is a variable that has a constraint with X_i) which is different than X_j . This step is necessary because the change in D_i might enable further reductions

in the domains of D_k , even if X_k has been considered before.

- ✧ If D_i is revised down to an empty set, then we know the entire CSP has no consistent solution, and AC-3 immediately returns failure.

The algorithm continues, attempting to remove values from the domains of variables until no more arcs are in the queue. At that point, we are left with a CSP that is equivalent to the original one (i.e., they both have the same solutions), but the arc-consistent CSP will, in most cases, be faster to search because its variables have smaller domains.

Input:

- `csp`: Data structure containing all variables, domains, and constraints of a binary CSP

Output: False if an inconsistency is found and True otherwise

Function AC_3(`csp`)

Begin

`agenda`: queue

`listArcs` <- `csp.getArcs()` // Get the set of all arcs from the CSP constraints

`agenda.enqueue(listArcs)` // Add all arcs in the queue

while (not `agenda.empty()`) **do**

 (X_i, X_j) <- `agenda.dequeue()` // Pop the first arc

if (`REVISE`(`csp`, X_i , X_j)) **then** // Revise the domain of X_i which is D_i

if (`size`(`csp.Di`) = 0) **then**

return False

end if

for each X_k **in** $X_i.getNeighbors(csp) - \{X_j\}$ **do**

`agenda.enqueue((X_k, X_i))`

end for

end if

end while

return True

End

function REVISE(`csp`, X_i , X_j)

Begin

`revised` <- False

for each v_i **in** D_i **do**

v_j <- get a value in D_j that allows (v_i, v_j) to satisfy the constraint between X_i and X_j

if ($v_j = \text{None}$) **then**

`delete`(v_i, D_i)

`revised` <- True

end if

return `revised`

End

Figure 3.15 AC-3 algorithm

Example 7: Apply the AC-3 algorithm for the CSP defined by the variables A, B, C, D, each with a domain of values {1, 2, 3} and the following constraints: **C1:** $A \neq B$, **C2:** $C < B$, **C3:** $C < D$. The

constraint graph is shown in Figure 3.14.

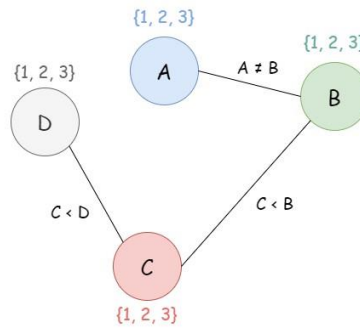


Figure 3.16 Constraint graph for example

Solution: Table 3.3 shows the application of AC-3 on example 7. The arc-consistent CSP is shown in Figure 3.17.

Table 3.3 Application of AC-3 on example 7

Agenda	Selected arc (X_i, X_j)	Domains
$\{(A, B), (B, A), (B, C), (C, B), (C, D), (D, C)\}$	(A, B)	$D_A = \{1, 2, 3\}$ $D_B = \{1, 2, 3\}$ $D_C = \{1, 2, 3\}$ $D_D = \{1, 2, 3\}$
$\{(B, A), (B, C), (C, B), (C, D), (D, C)\}$	(B, A)	$D_A = \{1, 2, 3\}$ $D_B = \{1, 2, 3\}$ $D_C = \{1, 2, 3\}$ $D_D = \{1, 2, 3\}$
$\{(B, C), (C, B), (C, D), (D, C), (A, B)\}$	(B, C)	$D_A = \{1, 2, 3\}$ $D_B = \{2, 3\}$ $D_C = \{1, 2, 3\}$ $D_D = \{1, 2, 3\}$
$\{(C, B), (C, D), (D, C), (A, B)\}$	(C, B)	$D_A = \{1, 2, 3\}$ $D_B = \{2, 3\}$ $D_C = \{1, 2\}$ $D_D = \{1, 2, 3\}$
$\{(C, D), (D, C), (A, B)\}$	(C, D)	$D_A = \{1, 2, 3\}$ $D_B = \{2, 3\}$ $D_C = \{1, 2\}$ $D_D = \{1, 2, 3\}$
$\{(D, C), (A, B)\}$	(D, C)	$D_A = \{1, 2, 3\}$ $D_B = \{2, 3\}$ $D_C = \{1, 2\}$ $D_D = \{2, 3\}$
$\{(A, B)\}$	(A, B)	$D_A = \{1, 2, 3\}$ $D_B = \{2, 3\}$ $D_C = \{1, 2\}$ $D_D = \{2, 3\}$

{}		$D_A = \{1, 2, 3\}$ $D_B = \{2, 3\}$ $D_C = \{1, 2\}$ $D_D = \{2, 3\}$
----	--	---

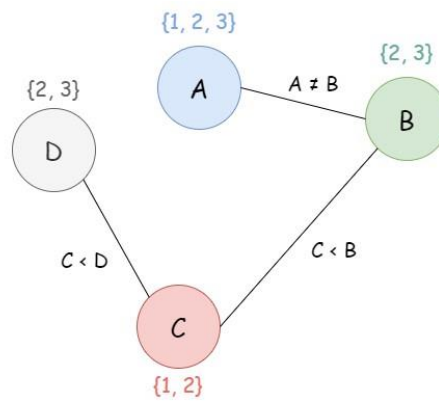


Figure 3.17 Result of application of AC-3 on example 7

Exercise 1: n-Queens problem

The n-Queens problem involves placing four queens on a $n \times n$ chessboard in such a way that no two queens attack each other. Specifically, no two queens are allowed to be positioned in the same row, column, or diagonal. We will be seeking a solution for $n = 4$ on a 4×4 chessboard.

1. Propose a CSP formulation for the 4-Queens problem (define the set of variables with their domains and the set of constraints)
2. Propose a solution for this CSP using the Backtracking Search Algorithm (without using any heuristic or inference).
3. Propose a solution for this CSP using the Backtracking Search Algorithm with Forward-Checking. Do not use any heuristic for variable and value selection.

Solution:

1. The CSP formulation for the 4-Queens problem is defined as following:

✧ **Variables and domains:**

Q_1, Q_2, Q_3, Q_4 : Variables representing the four queens, where the indices represent the columns where the queens are placed. The domain for each variable is $\{1, 2, 3, 4\}$, corresponding to the four rows of the 4×4 chessboard.

✧ **Constraints:**

C1: This constraint ensures that no two queens are placed in the same row.

$$Q_i \neq Q_j \quad \text{for } i, j \in \{1, 2, 3, 4\}, \quad i \neq j$$

C2: This constraint ensures that no two queens are placed on the same diagonal.

$$|Q_i - Q_j| \neq |i - j| \quad \text{for } i, j \in \{1, 2, 3, 4\}, \quad i \neq j$$

2. Finding a valid assignment for the variables Q_1, Q_2, Q_3, Q_4 that satisfies all the constraints using the Backtracking Search algorithm (without using any heuristic or inference).

Table 3.4 Solving the 4-Queens problem using Backtracking Search algorithm

Assignment	Constraint satisfaction		Action
	C1	C2	
			Select the variable Q_1
$Q_1=1$	Yes	Yes	Select the variable Q_2
$Q_1=1, Q_2=1$	No (between Q_1 and Q_2)	Yes	Select another value for Q_2
$Q_1=1, Q_2=2$	Yes	No (between Q_1 and Q_2)	Select another value for Q_2
$Q_1=1, Q_2=3$	Yes	Yes	Select the variable Q_3
$Q_1=1, Q_2=3, Q_3=1$	No (between Q_1 and Q_3)	Yes	Select another value for Q_3
$Q_1=1, Q_2=3, Q_3=2$	Yes	No (between Q_2 and Q_3)	Select another value for Q_3
$Q_1=1, Q_2=3, Q_3=3$	No (between Q_2 and Q_3)	No (between Q_1 and Q_3)	Select another value for Q_3
$Q_1=1, Q_2=3, Q_3=4$	Yes	No (between Q_2 and Q_3)	Back to Q_2 and select another value for Q_2
$Q_1=1, Q_2=4$	Yes	Yes	Select the variable Q_3
$Q_1=1, Q_2=4, Q_3=1$	No (between Q_1 and Q_3)	Yes	Select another value for Q_3

$Q_3=1$			
$Q_1=1, Q_2=4, Q_3=2$	Yes	Yes	Select the variable Q_4
$Q_1=1, Q_2=4, Q_3=2, Q_4=1$	No (between Q_1 and Q_4)	No (between Q_3 and Q_4)	Select another value for Q_4
$Q_1=1, Q_2=4, Q_3=2, Q_4=2$	No (between Q_3 and Q_4)	No (between Q_2 and Q_4)	Select another value for Q_4
$Q_1=1, Q_2=4, Q_3=2, Q_4=3$	Yes	No (between Q_3 and Q_4)	Select another value for Q_4
$Q_1=1, Q_2=4, Q_3=2, Q_4=4$	No (between Q_2 and Q_4)	No (between Q_1 and Q_4)	Back to Q_3 and select another value for Q_3
$Q_1=1, Q_2=4, Q_3=3$	Yes	No (between Q_1 and Q_3 and between Q_2 and Q_3)	Select another value for Q_3
$Q_1=1, Q_2=4, Q_3=4$	No (between Q_2 and Q_3)	Yes	Back to Q_2 , then back to Q_1 and select another value for Q_1
$Q_1=2$	Yes	Yes	Select the variable Q_2
$Q_1=2, Q_2=1$	Yes	No (between Q_1 and Q_2)	Select another value for Q_2
$Q_1=2, Q_2=2$	No (between Q_1 and Q_2)	Yes	Select another value for Q_2
$Q_1=2, Q_2=3$	Yes	No (between Q_1 and Q_2)	Select another value for Q_2
$Q_1=2, Q_2=4$	Yes	Yes	Select the variable Q_3
$Q_1=2, Q_2=4, Q_3=1$	Yes	Yes	Select the variable Q_4
$Q_1=2, Q_2=4, Q_3=1, Q_4=1$	No (between Q_3 and Q_4)	Yes	Select another value for Q_4
$Q_1=2, Q_2=4, Q_3=1, Q_4=2$	No (between Q_1 and Q_4)	No (between Q_2 and Q_4 , and between Q_3 and Q_4)	Select another value for Q_4
$Q_1=2, Q_2=4, Q_3=1, Q_4=3$	Yes	Yes	Assignment complete

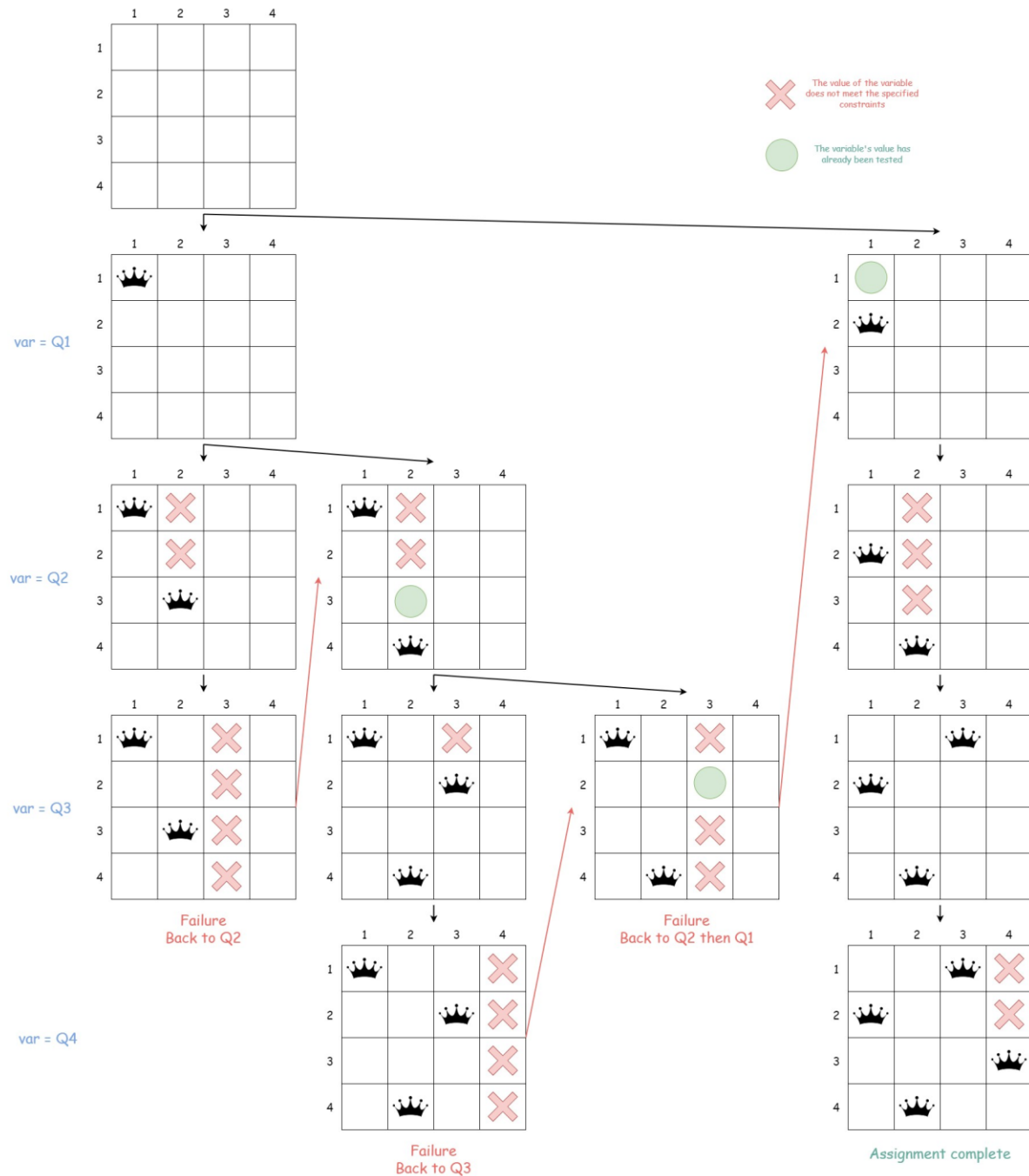


Figure 3.18 Solving the 4-Queens problem using the Backtracking search algorithm

- Finding a valid assignment for the variables Q_1, Q_2, Q_3, Q_4 that satisfies all the constraints using the Backtracking Search Algorithm with Forward Checking.

Assignment	Domains	Action
	$D_{Q1} = \{1, 2, 3, 4\}$ $D_{Q2} = \{1, 2, 3, 4\}$ $D_{Q3} = \{1, 2, 3, 4\}$ $D_{Q4} = \{1, 2, 3, 4\}$	Select the variable Q_1
$Q_1=1$	$D_{Q2} = \{3, 4\}$ $D_{Q3} = \{2, 4\}$	Select the variable Q_2

	$D_{Q_4}=\{2, 3\}$	
$Q_1=1, Q_2=3$	$D_{Q_3}=\{ \}$ $D_{Q_4}=\{2\}$	Inference returns a failure => Restore the previous domain (remove the inference applied for $Q_2=3$)
$Q_1=1$	$D_{Q_2}=\{3, 4\}$ $D_{Q_3}=\{2, 4\}$ $D_{Q_4}=\{2, 3\}$	Select another value for Q_2
$Q_1=1, Q_2=4$	$D_{Q_3}=\{2\}$ $D_{Q_4}=\{3\}$	Select the variable Q_3
$Q_1=1, Q_2=4, Q_3=2$	$D_{Q_4}=\{ \}$	Inference returns a failure => Restore the previous domain (remove the inference applied for $Q_3=2$)
$Q_1=1, Q_2=4$	$D_{Q_3}=\{2\}$ $D_{Q_4}=\{3\}$	No other value for Q_3 Back to Q_2 Restore the previous domain (remove the inference applied for $Q_2=4$)
$Q_1=1$	$D_{Q_2}=\{3, 4\}$ $D_{Q_3}=\{2, 4\}$ $D_{Q_4}=\{2, 3\}$	No other value for Q_2 Back to Q_1 Restore the previous domain (remove the inference applied for $Q_1=1$)
	$D_{Q_1}=\{1, 2, 3, 4\}$ $D_{Q_2}=\{1, 2, 3, 4\}$ $D_{Q_3}=\{1, 2, 3, 4\}$ $D_{Q_4}=\{1, 2, 3, 4\}$	Select another value for Q_1
$Q_1=2$	$D_{Q_2}=\{4\}$ $D_{Q_3}=\{1, 3\}$ $D_{Q_4}=\{1, 3, 4\}$	Select the variable Q_2
$Q_1=2, Q_2=4$	$D_{Q_3}=\{1\}$ $D_{Q_4}=\{1, 3\}$	Select the variable Q_3
$Q_1=2, Q_2=4, Q_3=1$	$D_{Q_4}=\{3\}$	Select the variable Q_4
$Q_1=2, Q_2=4, Q_3=1, Q_4=3$		Assignment complete

F

F

U
a
S
S

NSW) , $\langle (\text{NSW}, \text{V}), \text{NSW} \neq \text{V} \rangle$ }.

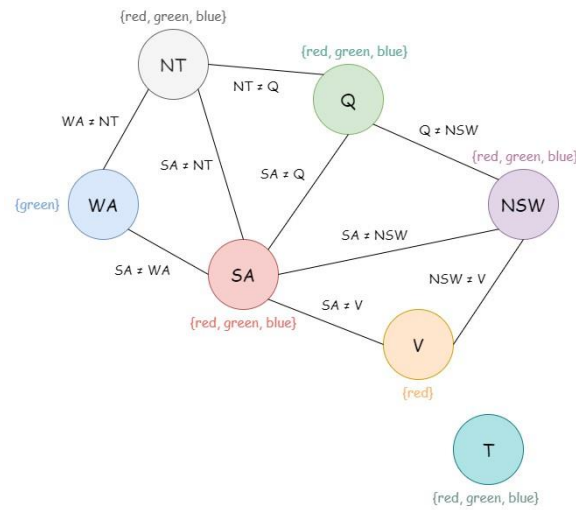


Figure 3. 20 Constraint graph for Australia map coloring problem

Solution:

Agenda	Selected arc (X_i, X_j)	Domains
		$D_{WA} = \{\text{green}\}$ $D_{NT} = \{\text{red, green, blue}\}$ $D_Q = \{\text{red, green, blue}\}$ $D_{NSW} = \{\text{red, green, blue}\}$ $D_V = \{\text{red}\}$ $D_{SA} = \{\text{red, green, blue}\}$ $D_T = \{\text{red, green, blue}\}$
$\{(\text{SA}, \text{WA}), (\text{WA}, \text{SA}), (\text{SA}, \text{NT}), (\text{NT}, \text{SA}), (\text{SA}, \text{Q}), (\text{Q}, \text{SA}), (\text{SA}, \text{NSW}), (\text{NSW}, \text{SA}), (\text{SA}, \text{V}), (\text{V}, \text{SA}), (\text{WA}, \text{NT}), (\text{NT}, \text{WA}), (\text{NT}, \text{Q}), (\text{Q}, \text{NT}), (\text{Q}, \text{NSW}), (\text{NSW}, \text{Q}), (\text{NSW}, \text{V}), (\text{V}, \text{NSW})\}$	(SA, WA)	$D_{WA} = \{\text{green}\}$ $D_{NT} = \{\text{red, green, blue}\}$ $D_Q = \{\text{red, green, blue}\}$ $D_{NSW} = \{\text{red, green, blue}\}$ $D_V = \{\text{red}\}$ $D_{SA} = \{\text{red, blue}\}$ $D_T = \{\text{red, green, blue}\}$
$\{(\text{WA}, \text{SA}), (\text{SA}, \text{NT}), (\text{NT}, \text{SA}), (\text{SA}, \text{Q}), (\text{Q}, \text{SA}), (\text{SA}, \text{NSW}), (\text{NSW}, \text{SA}), (\text{SA}, \text{V}), (\text{V}, \text{SA}), (\text{WA}, \text{NT}), (\text{NT}, \text{WA}), (\text{NT}, \text{Q}), (\text{Q}, \text{NT}), (\text{Q}, \text{NSW}), (\text{NSW}, \text{Q}), (\text{NSW}, \text{V}), (\text{V}, \text{NSW})\}$	(WA, SA), (SA, NT), (NT, SA), (SA, Q), (Q, SA), (SA, NSW), (NSW, SA)	$D_{WA} = \{\text{green}\}$ $D_{NT} = \{\text{red, green, blue}\}$ $D_Q = \{\text{red, green, blue}\}$ $D_{NSW} = \{\text{red, green, blue}\}$ $D_V = \{\text{red}\}$ $D_{SA} = \{\text{red, blue}\}$ $D_T = \{\text{red, green, blue}\}$
$\{(\text{SA}, \text{V}), (\text{V}, \text{SA}), (\text{WA}, \text{NT}), (\text{NT}, \text{WA}), (\text{NT}, \text{Q}), (\text{Q}, \text{NT}), (\text{Q}, \text{NSW}), (\text{NSW}, \text{Q}), (\text{NSW}, \text{V}), (\text{V}, \text{NSW}), (\text{WA}, \text{SA}), (\text{NT}, \text{SA}), (\text{Q}, \text{SA}), (\text{NSW}, \text{SA})\}$	(SA, V)	$D_{WA} = \{\text{green}\}$ $D_{NT} = \{\text{red, green, blue}\}$ $D_Q = \{\text{red, green, blue}\}$ $D_{NSW} = \{\text{red, green, blue}\}$

		$D_V = \{\text{red}\}$ $D_{SA} = \{\text{blue}\}$ $D_T = \{\text{red, green, blue}\}$
$\{(V, SA), (WA, NT), (NT, WA), (NT, Q), (Q, NT), (Q, NSW), (NSW, Q), (NSW, V), (V, NSW), (WA, SA), (NT, SA), (Q, SA), (NSW, SA)\}$	$(V, SA), (WA, NT)$	$D_{WA} = \{\text{green}\}$ $D_{NT} = \{\text{red, green, blue}\}$ $D_Q = \{\text{red, green, blue}\}$ $D_{NSW} = \{\text{red, green, blue}\}$ $D_V = \{\text{red}\}$ $D_{SA} = \{\text{blue}\}$ $D_T = \{\text{red, green, blue}\}$
$\{(NT, WA), (NT, Q), (Q, NT), (Q, NSW), (NSW, Q), (NSW, V), (V, NSW), (WA, SA), (NT, SA), (Q, SA), (NSW, SA), (SA, NT)\}$	(NT, WA)	$D_{WA} = \{\text{green}\}$ $D_{NT} = \{\text{red, blue}\}$ $D_Q = \{\text{red, green, blue}\}$ $D_{NSW} = \{\text{red, green, blue}\}$ $D_V = \{\text{red}\}$ $D_{SA} = \{\text{blue}\}$ $D_T = \{\text{red, green, blue}\}$
$\{(NT, Q), (Q, NT), (Q, NSW), (NSW, Q), (NSW, V), (V, NSW), (WA, SA), (NT, SA), (Q, SA), (NSW, SA), (SA, NT)\}$	$(NT, Q), (Q, NT), (Q, NSW), (NSW, Q)$	$D_{WA} = \{\text{green}\}$ $D_{NT} = \{\text{red, blue}\}$ $D_Q = \{\text{red, green, blue}\}$ $D_{NSW} = \{\text{red, green, blue}\}$ $D_V = \{\text{red}\}$ $D_{SA} = \{\text{blue}\}$ $D_T = \{\text{red, green, blue}\}$
$\{(NSW, V), (V, NSW), (WA, SA), (NT, SA), (Q, SA), (NSW, SA), (SA, NT), (SA, NSW), (Q, NSW)\}$	(NSW, V)	$D_{WA} = \{\text{green}\}$ $D_{NT} = \{\text{red, blue}\}$ $D_Q = \{\text{red, green, blue}\}$ $D_{NSW} = \{\text{green, blue}\}$ $D_V = \{\text{red}\}$ $D_{SA} = \{\text{blue}\}$ $D_T = \{\text{red, green, blue}\}$
$\{(V, NSW), (WA, SA), (NT, SA), (Q, SA), (NSW, SA), (SA, NT), (SA, NSW), (Q, NSW)\}$	$(V, NSW), (WA, SA)$	$D_{WA} = \{\text{green}\}$ $D_{NT} = \{\text{red, blue}\}$ $D_Q = \{\text{red, green, blue}\}$ $D_{NSW} = \{\text{green, blue}\}$ $D_V = \{\text{red}\}$ $D_{SA} = \{\text{blue}\}$ $D_T = \{\text{red, green, blue}\}$
$\{(NT, SA), (Q, SA), (NSW, SA), (SA, NT), (SA, NSW), (Q, NSW), (WA, NT), (Q, NT)\}$	(NT, SA)	$D_{WA} = \{\text{green}\}$ $D_{NT} = \{\text{red}\}$ $D_Q = \{\text{red, green, blue}\}$ $D_{NSW} = \{\text{green, blue}\}$ $D_V = \{\text{red}\}$ $D_{SA} = \{\text{blue}\}$

		$D_T = \{\text{red, green, blue}\}$
$\{(\text{Q, SA}), (\text{NSW, SA}), (\text{SA, NT}), (\text{SA, NSW}), (\text{Q, NSW}), (\text{WA, NT}), (\text{Q, NT}), (\text{NT, Q}), (\text{NSW, Q})\}$	(Q, SA)	$D_{WA} = \{\text{green}\}$ $D_{NT} = \{\text{red}\}$ $D_Q = \{\text{red, green}\}$ $D_{NSW} = \{\text{green, blue}\}$ $D_V = \{\text{red}\}$ $D_{SA} = \{\text{blue}\}$ $D_T = \{\text{red, green, blue}\}$
$\{(\text{NSW, SA}), (\text{SA, NT}), (\text{SA, NSW}), (\text{Q, NSW}), (\text{WA, NT}), (\text{Q, NT}), (\text{NT, Q}), (\text{NSW, Q}), (\text{V, NSW})\}$	(NSW, SA)	$D_{WA} = \{\text{green}\}$ $D_{NT} = \{\text{red}\}$ $D_Q = \{\text{red, green}\}$ $D_{NSW} = \{\text{green}\}$ $D_V = \{\text{red}\}$ $D_{SA} = \{\text{blue}\}$ $D_T = \{\text{red, green, blue}\}$
$\{(\text{SA, NT}), (\text{SA, NSW}), (\text{Q, NSW}), (\text{WA, NT}), (\text{Q, NT}), (\text{NT, Q}), (\text{NSW, Q}), (\text{V, NSW})\}$	$(\text{SA, NT}), (\text{SA, NSW})$	$D_{WA} = \{\text{green}\}$ $D_{NT} = \{\text{red}\}$ $D_Q = \{\text{red, green}\}$ $D_{NSW} = \{\text{green}\}$ $D_V = \{\text{red}\}$ $D_{SA} = \{\text{blue}\}$ $D_T = \{\text{red, green, blue}\}$
$\{(\text{Q, NSW}), (\text{WA, NT}), (\text{Q, NT}), (\text{NT, Q}), (\text{NSW, Q}), (\text{V, NSW}), (\text{SA, Q})\}$	(Q, NSW)	$D_{WA} = \{\text{green}\}$ $D_{NT} = \{\text{red}\}$ $D_Q = \{\text{red}\}$ $D_{NSW} = \{\text{green}\}$ $D_V = \{\text{red}\}$ $D_{SA} = \{\text{blue}\}$ $D_T = \{\text{red, green, blue}\}$
$\{(\text{WA, NT}), (\text{Q, NT}), (\text{NT, Q}), (\text{NSW, Q}), (\text{V, NSW}), (\text{SA, Q})\}$	(WA, NT)	$D_{WA} = \{\text{green}\}$ $D_{NT} = \{\text{red}\}$ $D_Q = \{\text{red}\}$ $D_{NSW} = \{\text{green}\}$ $D_V = \{\text{red}\}$ $D_{SA} = \{\text{blue}\}$ $D_T = \{\text{red, green, blue}\}$
$\{(\text{Q, NT}), (\text{NT, Q}), (\text{NSW, Q}), (\text{V, NSW}), (\text{SA, Q})\}$	(Q, NT)	$D_{WA} = \{\text{green}\}$ $D_{NT} = \{\text{red}\}$ $D_Q = \{\}: \text{Inconsistency detected}$ $D_{NSW} = \{\text{green}\}$ $D_V = \{\text{red}\}$ $D_{SA} = \{\text{blue}\}$ $D_T = \{\text{red, green, blue}\}$

Exercise 3: exam-scheduling problem

Students 1-4 are each enrolled in three courses from A, B, ..., G. Every course requires an exam, and exams can be scheduled on Monday, Tuesday, or Wednesday. However, a student cannot have two exams on the same day. This can be illustrated as follows (Figure 3.21):

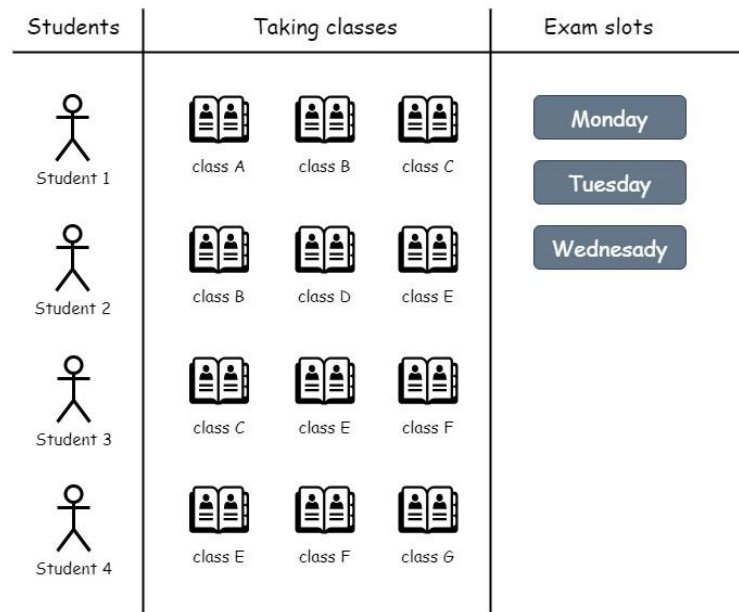


Figure 3.21 Exam scheduling problem

- Propose a CSP formulation for the exam scheduling problem: define the variables with their domains and the constraints (give a graph representation for the constraints).
- Propose a solution for this CSP using the Backtracking Search Algorithm with Forward-Checking. Use the MRV and the Degree heuristics for variable selection and the LCV heuristic for value selection.

Solution:

In this scenario, the variables represent the courses, the domain corresponds to the days, and the constraints dictate which courses cannot be scheduled for exams on the same day due to a common student enrollment.

This problem can be resolved by employing constraints represented as a graph. In this graph, each node corresponds to a course, and an edge connects two courses if they cannot be scheduled on the same day. In this instance, the graph will appear as follows (see Figure 3.22):

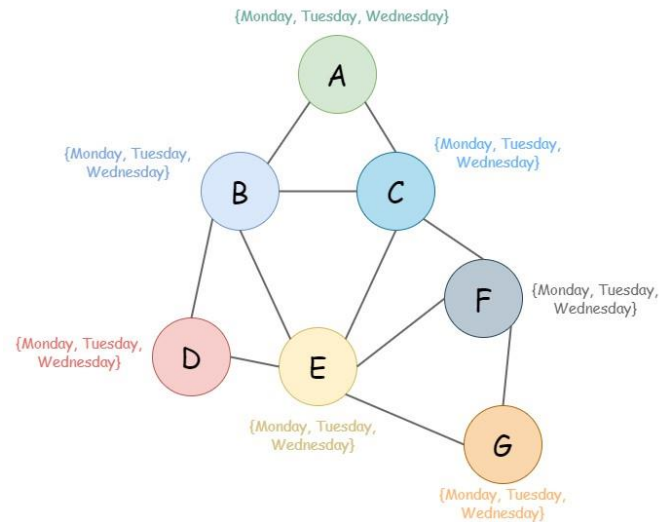


Figure 3.22 Constraint graph for the exam scheduling problem

Assignment	Domains	Number of constraints with unassigned variables	Action
	A={Monday, Tuesday, Wednesday} B={Monday, Tuesday, Wednesday} C={Monday, Tuesday, Wednesday} D={Monday, Tuesday, Wednesday} E={Monday, Tuesday, Wednesday} F={Monday, Tuesday, Wednesday} G={Monday, Tuesday, Wednesday}	2 constraints on A 4 constraints on B 4 constraints on C 2 constraints on D 5 constraints on E 3 constraints on F 2 constraints on G	Selecting the variable E
E=Monday	A={Monday, Tuesday, Wednesday} B={Tuesday, Wednesday} C={Tuesday, Wednesday} D={Tuesday, Wednesday} F={Tuesday, Wednesday} G={Tuesday, Wednesday}	3 constraints on B 3 constraints on C 1 constraint on D 2 constraints on F 1 constraint on G	Selecting the variable B
E=Monday, B=Tuesday	A={Monday, Wednesday} C={Wednesday} D={Wednesday} F={Tuesday, Wednesday} G={Tuesday, Wednesday}	2 constraints on C no constraints on D	Selecting the variable C
E=Monday, B=Tuesday, C=Wednesday	A={Monday} D={Wednesday} F={Tuesday} G={Tuesday, Wednesday}	no constraints on A no constraint on D 1 constraint on F	Selecting the variable F
E=Monday, B=Tuesday, C=Wednesday, F=Tuesday	A={Monday} D={Wednesday} G={Wednesday}	no constraints on A no constraints on D no constraints on G	Selecting the variable A
E=Monday,	D={Wednesday}	no constraints on D	Selecting the

B=Tuesday, C=Wednesday, F= Tuesday, A= Monday	G={Wednesday}	no constraints on G	variable D
E=Monday, B=Tuesday, C=Wednesday, F= Tuesday, A= Monday D=Wednesday	G={Wednesday}	no constraints on G	Selecting the variable G
E=Monday, B=Tuesday, C=Wednesday, F= Tuesday, A= Monday D=Wednesday G=Wednesday			Assignment complete

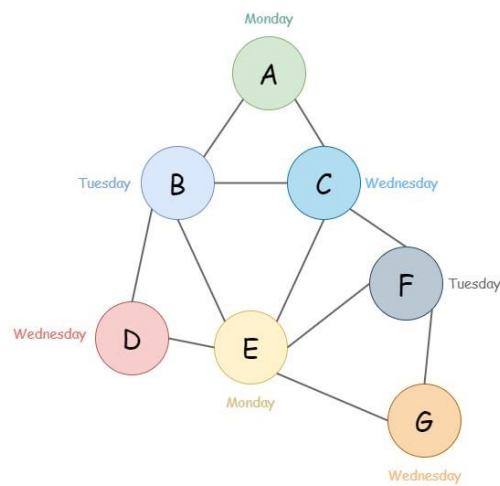


Figure 3.23 Possible solution for the exam scheduling problem