CS-602 - Design of Problem Solvers

Chapter 5 - Constraint Satisfaction Problems

Dr. Mahdi Khemakhem

Department of Computer Science

College of Computer Engineering and Science

Prince Sattam bin Abdulaziz University

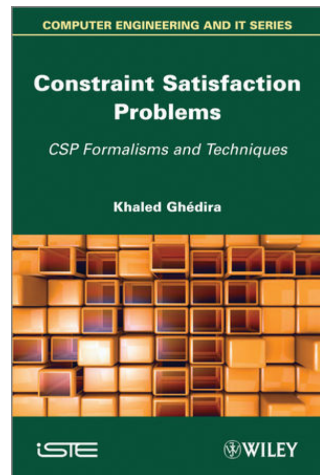
AY - 2025/2026

Outline

1. Preview
2. CSPs: Definition and Concepts
3. Classic CSP Examples
   3.1 N-Queens Problem
   3.2 Map Coloring Problem
   3.3 Sudoku Puzzle
4. CSP Solution Techniques
   4.1 Backtracking Search
   4.2 Improving Backtracking
   4.3 Constraint Propagation
   4.4 Intelligent Backtracking
5. Local Search for CSPs
6. CSP Problem Structure
7. Real-World Applications
8. Summary and Key Takeaways
9. Note on `python-constraint` Library
10. Homework

Materials

- **Textbook:** "Constraint Satisfaction Problems: CSP Formalisms and Techniques", 1st Edition, by *Khaled Ghedira*
- ⇒ **Read Chapters 1, 2 and 3 for this chapter's material**



Outline

1. Preview
2. CSPs: Definition and Concepts
3. Classic CSP Examples
4. CSP Solution Techniques
5. Local Search for CSPs
6. CSP Problem Structure
7. Real-World Applications
8. Summary and Key Takeaways

Preview

- **In previous chapters, we saw:**
  - various ways of **querying** and **optimizing** queries in logic programming.
  - techniques for **evaluation** and **optimization** of logical queries.
- **In this chapter, we will explore:**
  - the concept of **Constraint Satisfaction Problems (CSPs)** as a *structured approach* to problem-solving.
  - various techniques for *solving CSPs* including **backtracking** and **constraint propagation**.
  - *real-world applications* of CSPs in scheduling, planning, and resource allocation.

Outline

1. Preview
2. CSPs: Definition and Concepts
3. Classic CSP Examples
4. CSP Solution Techniques
5. Local Search for CSPs
6. CSP Problem Structure
7. Real-World Applications
8. Summary and Key Takeaways

What is a Constraint Satisfaction Problem?

A **Constraint Satisfaction Problem (CSP)** is a mathematical problem defined by:

- A set of **variables** that need to be assigned values
- A set of **domains** defining possible values for each variable
- A set of **constraints** that limit which combinations of values are allowed

The goal is to find an assignment of values to all variables such that all constraints are satisfied.

Formal Definition

Formally, a CSP is defined as a triple $\langle X, D, C \rangle$, where:

- $X = \{X_1, X_2, ..., X_n\}$ is a set of **variables**
- $D = \{D_1, D_2, ..., D_n\}$ is a set of **domains**
  - Each variable X_i can take values from domain D_i
  - D_i can be **finite** or **infinite**, **discrete** or **continuous**
- $C = \{C_1, C_2, ..., C_m\}$ is a set of **constraints**
  - Each constraint C_j is a **pair** $\langle scope_j, relation_j \rangle$
  - $scope_j$ is a **subset of variables involved in the constraint**
  - $relation_j$ is a **set of allowed value combinations** for the variables in $scope_j$

Key Concepts

Each CSP involves several key concepts:

- **Assignment:** A mapping of variables to values
- **Consistent Assignment:** An **assignment** that **does not violate any constraints**
- **Complete Assignment:** An **assignment** that **assigns values to all variables**
- **Solution:** **A complete and consistent assignment**

A CSP is satisfiable if at least one solution exists, unsatisfiable otherwise.

If a CSP is **unsatisfiable**, it means there is **no way to assign values to all variables without violating at least one constraint**.

Outline

1. Preview

2. CSPs: Definition and Concepts

3. Classic CSP Examples
3.1 N-Queens Problem
3.2 Map Coloring Problem
3.3 Sudoku Puzzle

4. CSP Solution Techniques

5. Local Search for CSPs

6. CSP Problem Structure

7. Real-World Applications

N-Queens Problem as a CSP

Problem: Place N queens on an $N \times N$ chessboard such that **no two queens threaten each other**.

CSP Formulation

- **Variables:** $Q_1, Q_2, ..., Q_N$ (one per column)
- **Domains:** $\forall i \in \{1, 2, ..., N\}, D_i = \{1, 2, ..., N\}$ (row positions)
- **Constraints:**
  - No two queens in same row: $Q_i \neq Q_j, \forall i, j \in \{1, 2, ..., N\}, i \neq j$
  - No diagonal attacks: $|Q_i - Q_j| \neq |i - j|, \forall i, j \in \{1, 2, ..., N\}, i \neq j$

	1	2	3	4
1	-	Q_2	-	-
2	-	-	-	Q_4
3	Q_1	-	-	-
4	-	-	Q_3	-

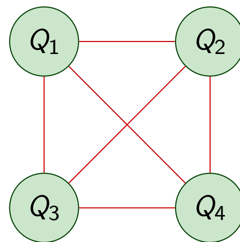
Example: 4-Queens Solution

N-Queens Constraint Graph

Constraint Graph Representation

In the constraint graph:

- Each **node** represents a variable (row position of a queen)
- Each **edge** represents a constraint between two variables
- For **N-Queens**, we have a **complete graph** since every pair of queens constrains each other



4-Queens Constraint Graph

Map Coloring Problem as a CSP

Problem: Color regions of a map such that **no two adjacent regions have the same color**.

CSP Formulation:

- **Variables:** One per region (WA, NT, SA, Q, NSW, V, T)
- **Domains:** $\{Red, Green, Blue\}$
- **Constraints:** Adjacent regions must have different colors
  - $WA \neq NT$, $WA \neq SA$
  - $NT \neq SA$, $NT \neq Q$
  - $SA \neq Q$, $SA \neq NSW$, $SA \neq V$
  - $Q \neq NSW$
  - $NSW \neq V$

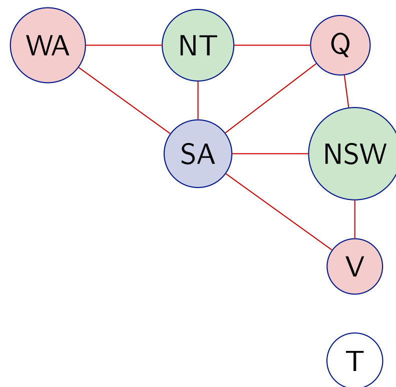
Example: Australia Map

Map Coloring Constraint Graph

Constraint Graph Representation

In the constraint graph:

- Each **node** represents a variable (color of a region)
- Each **edge** represents a constraint between two variables
- For **Map Coloring**, we have a **sparse graph** since not all regions are adjacent



Australia Map Constraint Graph

Sudoku Puzzle as a CSP

Problem: Fill a 9×9 grid so that **each row, column, and** 3×3 **subgrid contains all digits** $1 - 9$.

CSP Formulation:

- **Variables:** 81 cells ($X_{i,j}$) where $1 \leq i, j \leq 9$
- **Domains:** $\{1, 2, 3, 4, 5, 6, 7, 8, 9\}$ (**some cells are initially filled**)
- **Constraints:**
  - **Row constraints:** All different in each row
    $X_{i,1} \neq X_{i,2} \neq ... \neq X_{i,9}, \forall i \in \{1, 2, ..., 9\}$
  - **Column constraints:** All different in each column
    $X_{1,j} \neq X_{2,j} \neq ... \neq X_{9,j}, \forall j \in \{1, 2, ..., 9\}$
  - **Box constraints:** All different in each 3×3 box
    $X_{m,n} \neq X_{m,n+1} \neq X_{m,n+2} \neq X_{m+1,n} \neq X_{m+1,n+1} \neq X_{m+1,n+2} \neq X_{m+2,n} \neq X_{m+2,n+1} \neq X_{m+2,n+2}$
    $\forall m \in \{1, 4, 7\}, \forall n \in \{1, 4, 7\}$

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

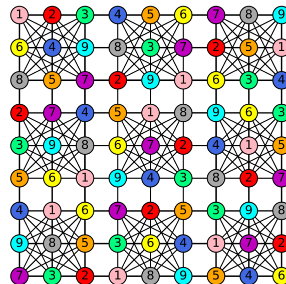
Example of a Sudoku Solution

Sudoku Constraint Graph

Constraint Graph Representation

In the constraint graph:

- Each **node** represents a variable (value of a cell)
- Each **edge** represents a constraint between two variables
- For **Sudoku**, we have a **highly connected graph** since many cells constrain each other



Sudoku Constraint Graph

Outline

1. Preview

2. CSPs: Definition and Concepts

3. Classic CSP Examples

4. CSP Solution Techniques
   - 4.1 Backtracking Search
   - 4.2 Improving Backtracking
   - 4.3 Constraint Propagation
   - 4.4 Intelligent Backtracking

5. Local Search for CSPs

6. CSP Problem Structure

Backtracking Algorithm

Backtracking is a **depth-first search** algorithm for solving CSPs by:

1. **Assigning** values to variables one at a time
2. **Checking** consistency after each assignment
3. **Backtracking** when no consistent value exists for a variable

Basic Algorithm Structure:

- If assignment is complete, return success
- Select an unassigned variable
- For each value in the variable's domain:
  - Assign the value
  - If consistent, recursively solve remaining variables
  - If successful, return success
  - Otherwise, remove assignment (backtrack)
- Return failure if no value works

Backtracking Pseudocode

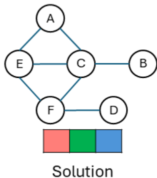
Algorithm 1: Backtracking Search for CSP

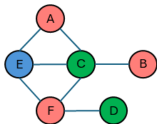
Input: CSP problem with variables, domains, and constraints
Output: Complete and consistent assignment or failure

```
1  Function: Backtrack(assignment, csp)
2      if assignment is complete then
3          return assignment ;                                   /* solution found */
4      end
5      var ← SelectUnassignedVariable(assignment, csp) ;         /* select unassigned variable */
6      for value in OrderDomainValues(var, assignment, csp) do
7          if IsConsistent(var, value, assignment, csp) then
8              assignment[var] ← value ;                         /* tentative assignment */
9              result ← Backtrack(assignment, csp) ;             /* recursive call */
10             if result ≠ failure then
11                 return result ;                               /* solution found */
12             end
13             remove {var = value} from assignment ;            /* backtrack */
14         end
15     end
16     return failure ;                                          /* no valid assignment found */
17 end
18 Main: return Backtrack({}, csp);
```

Backtracking Example: Map Coloring

	A	B	C	D	E	F
A			1		1	
B			1			
C	1	1			1	1
D						1
E	1		1			1
F			1	1	1	





A = Red ✅

A	B	C	D	E	F

Forward to B

B = Red ✅

A	B	C	D	E	F

Forward to C

C = Red ❌ - Conflict with A and B!
C = Green ✅

A	B	C	D	E	F

Forward to D

D = Red ✅

A	B	C	D	E	F

Forward to E

E = Red ❌ - Conflict with A!
E = Green ❌ - Conflict with C!
E = Blue ✅

A	B	C	D	E	F

Forward to F

F = Red ❌ - Conflict with D!
F = Green ❌ - Conflict with C!
F = Blue ❌ - Conflict with E!

A	B	C	D	E	F

Backtrack to E

E = No possible value → Discard Color

A	B	C	D	E	F

Backtrack to D

D = Green ✅

A	B	C	D	E	F

Forward to E

E = Red ❌ - Conflict with A!
E = Green ❌ - Conflict with C!
E = Blue ✅

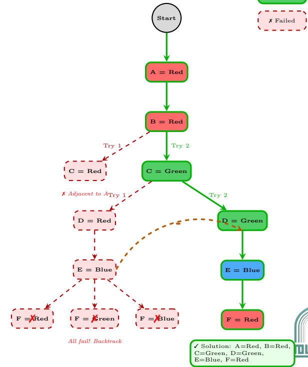
A	B	C	D	E	F

Forward to F

F = Red ✅

A	B	C	D	E	F

Complete and Consistent



Backtracking Example: 4-Queens

Step-by-Step Solution

Q1
Q1=1 ✓
Forward to Q2

Q1, Q2
Q2=1 ✗
Q2=2 ✗
Q2=3 ✓
Forward to Q3

Q1, Q2
Q3=1 ✗
Q3=2 ✗
Q3=3 ✗
Q3=4 ✗
Backtrack to Q2

Q1, ~~Q2~~, Q2
Q2=4 ✓
Forward to Q3

Q1, Q3, Q2
Q3=1 ✗
Q3=2 ✓
Forward to Q4

Q1, Q3, Q2
Q4=1 ✗
Q4=2 ✗
Q4=3 ✗
Q4=4 ✗
Backtrack to Q3

Q1, ~~Q3~~, Q2
Q3=3 ✗
Q3=4 ✗
Backtrack to Q2

Q1, ~~Q2~~
Backtrack to Q1

~~Q1~~, Q1
Q1=2 ✓
Forward to Q2

Q1, Q2
Q2=1 ✗
Q2=2 ✗
Q2=3 ✗
Q2=4 ✓
Forward to Q3

Q3, Q1, Q2
Q3=1 ✓
Forward to Q4

Q3, Q1, Q4, Q2
Q4=1 ✗
Q4=2 ✗
Q4=3 ✓
Complete and Consistent

Backtracking Example: 4-Queens

Backtracking Tree Visualization

Limitations of Basic Backtracking

Basic backtracking is **inefficient** for large CSPs:

- **Explores many dead ends** before finding a solution
- **No intelligence** in choosing **which variable to assign next**
- **No intelligence** in choosing **which value to try first**
- **Detects conflicts late** - only after full assignment

Key Takeaways

- **Variable ordering:** Choose which variable to assign next
- **Value ordering:** Choose which value to try first
- **Constraint propagation:** Detect conflicts early
- **Intelligent backtracking:** Avoid repeating failures

Variable Ordering Heuristics

Which variable should we assign next?

Minimum Remaining Values (MRV)

- Also called **"most constrained variable"** or **"fail-first"** heuristic
- Choose the **variable** with **fewest legal values remaining** in its domain
- **Rationale:** Detect failures early, prune search tree sooner

Degree Heuristic

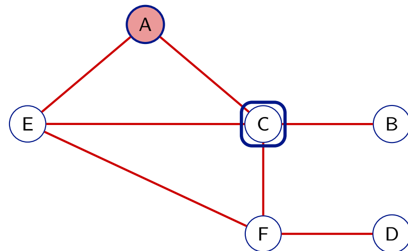
- Choose the variable involved in **most constraints** with other **unassigned variables**
- **Rationale:** Reduce branching factor for future choices
- Often used as a **tie-breaker** with MRV (i.e., when multiple variables have the same minimum remaining values, **choose the one with the highest degree**)

Variable Ordering Example: Map Coloring (Backtracking Impact)

6-Region Graph Coloring Example (3 colors: Red, Green, Blue)

Step 1: After assigning A = **Red**:

Variable	Remaining Values
A	**Red** (assigned)
B	Red, Green, Blue (3)
C	Green, Blue (2)
D	Red, Green, Blue (3)
E	Green, Blue (2)
F	Red, Green, Blue (3)



C (bold border) chosen next

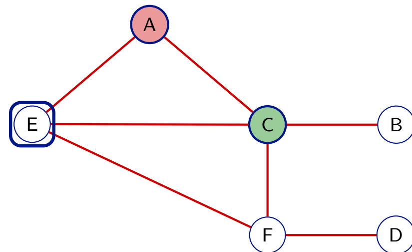
MRV heuristic: C and E have 2 remaining values (tie)
Decision (Degree heuristic for tie-breaker): (for tie-breaker) $deg(C) = 3$, $deg(E) = 3$ (tie) $\rightarrow$ **Choose C next (arbitrary choice) and assign C=Green**

Variable Ordering Example: Map Coloring (Step 2)

Step 2: After assigning A = **Red**, C = **Green**:

Variable	Remaining Values
A	**Red** (assigned)
B	Red, Blue (2)
C	**Green** (assigned)
D	Red, Green, Blue (3)
E	Blue (1)
F	Red, Blue (2)

MRV heuristic: E is most constrained (1 value only)
Decision: → **Choose E next and assign E=Blue**



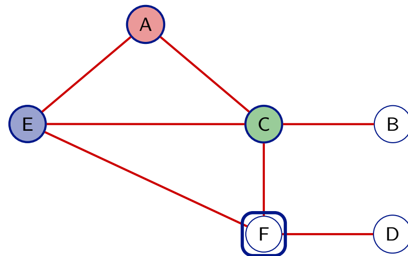
E (bold border) chosen next

Variable Ordering Example: Map Coloring (Step 3)

Step 3: After assigning A = **Red**, C = **Green**, E = **Blue**:

extbfVariable	**Remaining Values**
A	**Red** (assigned)
B	Red, Blue (2)
C	**Green** (assigned)
D	Red, Green, Blue (3)
E	Blue (assigned)
F	Red (1)

MRV heuristic: F is most constrained (1 value)
Decision: → **Choose F next and assign F=Red**

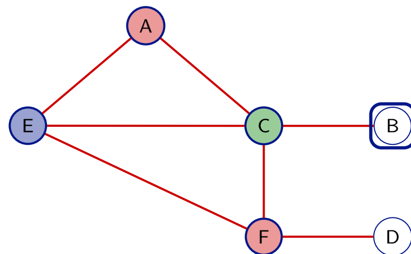


F (bold border) chosen next

Variable Ordering Example: Map Coloring (Step 4)

Step 4: After assigning A = **Red**, C = **Green**, E = **Blue**, F = **Red**:

Variable	Remaining Values
A	**Red** (assigned)
B	Red, Blue (2)
C	**Green** (assigned)
D	Green, Blue (2)
E	Blue (assigned)
F	**Red** (assigned)



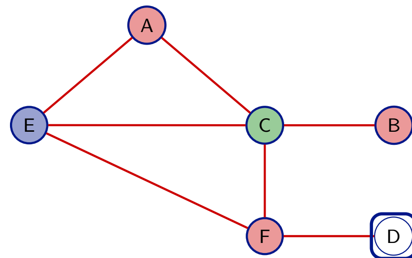
B (bold border) chosen next

MRV heuristic: B is most constrained (1 value)
Decision (Degree heuristic for tie-breaker): (for tie-breaker) $deg(B) = 0$, $deg(D) = 0$ (tie) $\rightarrow$ **Choose B next (arbitrary choice) and assign B=Green**

Variable Ordering Example: Map Coloring (Step 5)

Step 5: After assigning A = **Red**, C = **Green**, E = **Blue**, F = **Red**, B = **Red**:

Variable	Remaining Values
A	**Red** (assigned)
B	**Red** (assigned)
C	**Green** (assigned)
D	Green, Blue (2)
E	Blue (assigned)
F	**Red** (assigned)



D (bold border) chosen next

MRV heuristic: Only D remains (2 values)
Decision: → **Choose D=Green (or Blue) to complete**

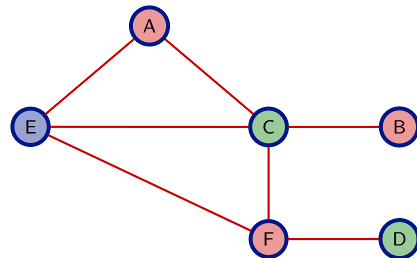
Only one variable left (D). Pick any non-Red color (e.g., Green). Solution completes without backtracking.

Variable Ordering Example: Map Coloring (Final Solution)

Final Assignment:

Variable	Assigned Color
A	**Red**
B	**Red**
C	**Green**
D	**Green**
E	**Blue**
F	**Red**

Success! Valid 3-coloring consistent with the new graph



Valid 3-coloring achieved!

All constraints satisfied: no adjacent regions share the same color.

Variable Ordering Heuristics: Key Takeaways

Key Takeaways

- **MRV heuristic** identified most constrained variables
- **Degree heuristic** broke ties effectively
- **Forward checking** eliminated values early
- **Smart ordering** = No backtracking needed!

Value Ordering Heuristics

Which value should we try first?

Least Constraining Value (LCV)

- Choose the value that **rules out the fewest values** for neighboring variables
- **Rationale:** Leave maximum flexibility for future assignments
- Opposite of "fail-first" - we want to **succeed first**

How to compute LCV?

1. For each possible value v of variable X
2. Count how many values it would eliminate from domains of **unassigned neighbors**
3. Choose the value with **minimum eliminations**

Value Ordering Example: Map Coloring (LCV Calculation - Step 1)

Scenario: After assigning $A =$ **Red**, $C =$ **Green**, which value should we choose for E?

Current State: Unassigned neighbors of E: $\{F\}$ (neighbors A, C already assigned)

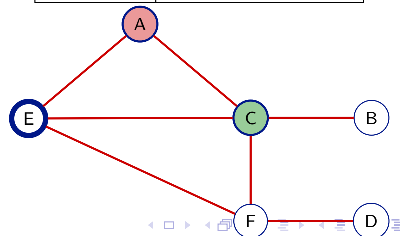
Evaluating each possible value for E:

- **Option 1: If $E =$ Red** $\Rightarrow$ Conflicts with A (adjacent). **Not valid!**
- **Option 2: If $E =$ Green** $\Rightarrow$ Conflicts with C (adjacent). **Not valid!**
- **Option 3: If $E =$ Blue** $\Rightarrow$ Eliminates Blue from: F (1) $\Rightarrow$ **Total eliminations: 1**

LCV Decision: Choose $E =$ Blue (only valid value)

Domain Analysis:

Variable	Domain
A	Red
B	Red, Green, Blue
C	Green
D	Red, Green, Blue
E	Blue (choosing)
F	Red, Green, Blue



Value Ordering Example: Map Coloring (LCV Calculation - Step 2)

Scenario: After assigning $A =$ **Red**, $C =$ **Green**, $E =$ Blue, which value for F?

Current State: Unassigned neighbors of F: $\{D\}$

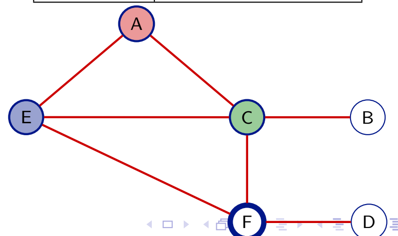
Evaluating each possible value for F:

- **Option 1: If** $F =$ **Red** $\Rightarrow$ Eliminates Red from: D (1) $\Rightarrow$ **Total eliminations: 1**
- **Option 2: If** $F =$ **Green** $\Rightarrow$ Conflicts with C (adjacent). **Not valid!**
- **Option 3: If** $F =$ **Blue** $\Rightarrow$ Conflicts with E (adjacent). **Not valid!**

LCV Decision: Choose $F =$ **Red** (only valid value, minimal eliminations)

Domain Analysis:

Variable	Domain
A	**Red**
B	Red, Blue
C	**Green**
D	Red, Green, Blue
E	Blue
F	**Red** (choosing)



Value Ordering Example: Map Coloring (LCV Calculation - Step 3)

Scenario: After assigning $A =$ **Red**, $C =$ **Green**, $E =$ Blue, $F =$ **Red**, which value for B?

Current State: Unassigned neighbors of B: none (C already assigned)

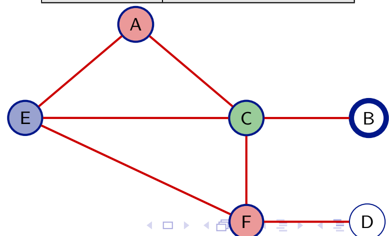
Evaluating each possible value for B:

- **Option 1:** If $B =$ **Red** ⇒ No eliminations (no unassigned neighbors) ⇒ **Total eliminations: 0**
- **Option 2:** If $B =$ **Green** ⇒ Conflicts with C. **Not valid!**
- **Option 3:** If $B =$ **Blue** ⇒ No eliminations ⇒ **Total eliminations: 0**

LCV Decision: Choose $B =$ **Red** (tie, pick first valid)

Domain Analysis:

Variable	Domain
A	Red
B	Red (choosing)
C	Green
D	Green, Blue
E	Blue
F	Red



Value Ordering Heuristics: Key Takeaways

Key Takeaways

- **LCV** chooses values that leave most options for neighbors
- **Complements MRV variable ordering**
- **Smart value ordering** = fewer dead-ends!

Constraint Propagation: Forward Checking

Forward Checking

Forward Checking is a simple form of constraint propagation

Idea:

- **After assigning a variable**, **eliminate inconsistent values** from domains of unassigned neighbors
- If any domain becomes **empty**, backtrack immediately
- Detects failures **earlier** than basic backtracking

Advantages:

- ✓ Prunes search space early
- ✓ Simple to implement
- ✓ Low overhead per assignment

Limitation: Detects conflicts only **one step ahead**

Forward Checking Example: Map Coloring (Step 1)

6-Region Graph: Forward Checking in Action

Step 1: Assign $A = $ **Red**

Variable	Domain
A	**Red** (assigned)
B	Red, Green, Blue
C	Red, Green, Blue
D	Red, Green, Blue
E	Red, Green, Blue
F	Red, Green, Blue

Forward Check: Remove **Red** from unassigned neighbors of A: $\{C, E\}$

After Forward Checking:

Variable	Domain
A	**Red** (assigned)
B	Red, Green, Blue
C	**Green, Blue**
D	Red, Green, Blue
E	**Green, Blue**
F	Red, Green, Blue

Result: Domains of C, E reduced from 3 to 2 values!

Forward Checking Example: Map Coloring (Step 2)

6-Region Graph: Forward Checking in Action

Step 2: Assign $C =$ **Green** (using MRV)

Variable	Domain
A	**Red** (assigned)
B	Red, Green, Blue
C	**Green** (assigned)
D	Red, Green, Blue
E	Green, Blue
F	Red, Green, Blue

Forward Check: Remove **Green** from unassigned neighbors of C: $\{B, E, F\}$

After Forward Checking:

Variable	Domain
A	**Red** (assigned)
B	Red, Blue
C	**Green** (assigned)
D	Red, Green, Blue
E	**Blue** (forced!)
F	Red, Blue

Key Benefit: E now has only **one choice** - automatically forced to Blue!

Forward Checking Example: Map Coloring (Completing)

6-Region Graph: Completing the Solution

Remaining assignments:

After A = **Red**, C = **Green**, forced assignment:

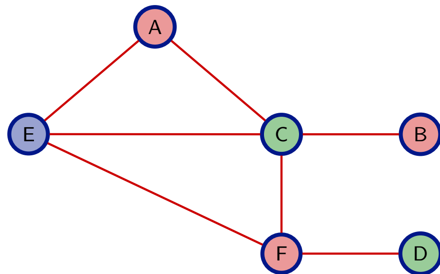
- E = Blue (forced)

Then continue:

- F: neighbors C (Green), E (Blue) → choose **Red**
- B: neighbor C (Green) → choose **Red**
- D: neighbor F (Red) → choose **Green**

Final Solution:

- A = **Red**, B = **Red**, C = **Green**
- D = **Green**, E = Blue, F = **Red**



Complete solution - no backtracking needed!

Success! Forward checking **detected forced assignment** early (E forced to Blue), simplifying the search.

Constraint Propagation: Arc Consistency

Arc Consistency

Arc Consistency is a stronger form of constraint propagation

Definition:

- A variable X is **arc-consistent** with variable Y if:
  - For every value x in domain of X
  - There exists some value y in domain of Y
  - Such that (x, y) satisfies the constraint between X and Y

AC-3 Algorithm:

- Maintains a queue of arcs (directed constraints) to check
- Removes values that violate arc consistency
- Propagates changes to related arcs
- Detects inconsistencies **before** search starts

Arc Consistency: AC-3 Algorithm Pseudocode (Part 1)

Algorithm 2: AC-3 Algorithm for Arc Consistency - Main Function

```
Input: CSP with variables, domains, and constraints
Output: Arc-consistent CSP or detection of inconsistency
1  Function: AC3(csp)
2    queue ← all arcs (X_i, X_j) in csp ;          /* initialize queue with all arcs */
3    while queue is not empty do
4      (X, Y) ← remove first arc from queue ;      /* dequeue an arc */
5      if Revise(csp, X, Y) then
6        if size of Domain[X] = 0 then
7          return false ;                          /* inconsistency detected */
8        end
9        for Z in neighbors of X except Y do
10         add (Z, X) to queue ;                   /* enqueue affected arcs */
11       end
12     end
13   end
14   return true ;                                 /* arc-consistent */
15 end
```

The AC-3 algorithm maintains a queue of arcs and iteratively enforces arc consistency by calling the Revise function.

Arc Consistency: AC-3 Algorithm Pseudocode (Part 2)

Algorithm 3: AC-3 Algorithm for Arc Consistency - Revise Function

Input: CSP, variables X and Y
Output: Boolean indicating if domain of X was revised

```
1  Function: Revise(csp, X, Y)
2      revised ← false ;                                   /* flag to track if domain was revised */
3      for x in Domain[X] do
4          if no value y in Domain[Y] satisfies constraint (X, Y) then
5              delete x from Domain[X] ;                   /* remove inconsistent value */
6              revised ← true ;                            /* mark that revision occurred */
7          end
8      end
9      return revised ;                                    /* return whether domain was revised */
10 end
```

The Revise function removes values from $Domain[X]$ that are inconsistent with all values in $Domain[Y]$. Returns true if any values were removed.

Arc Consistency (AC-3): Simple CSP Example (1/2)

Problem Definition:

- **Variables:** X, Y, Z
- **Domains:** $D_X = \{1,2,3\}, \quad D_Y = \{1,2\}, \quad D_Z = \{2,3\}$
- **Constraints:** $X \leq Y \quad \text{and} \quad Y \geq Z$

Step 1: Check arc (X, Y) — constraint $X \leq Y$

- $X = 1 \rightarrow$ supported by $Y = 1, 2$ ✔
- $X = 2 \rightarrow$ supported by $Y = 2$ ✔
- $X = 3 \rightarrow$ no $Y \geq 3$ ✗ $\rightarrow$ remove 3

$$D_X = \{1,2\}$$

Step 2: Check arc (Y, Z) — constraint $Y \geq Z$

- $Y = 1 \rightarrow$ needs $Z \leq 1$, but $D_Z = \{2,3\}$ ✗ $\rightarrow$ remove 1
- $Y = 2 \rightarrow$ supported by $Z = 2$ ✔

$$D_Y = \{2\}$$

Arc Consistency (AC-3): Simple CSP Example (2/2)

Step 3: Recheck arc (X, Y) — $X \leq Y$

- $D_Y = \{2\}$
- $X = 1$: $1 \leq 2$ ✔
- $X = 2$: $2 \leq 2$ ✔

$$D_X = \{1, 2\} \quad \text{(no change)}$$

Step 4: Check arc (Z, Y) — $Y \geq Z$ (from Z's side)

- $Z = 2$: $Y = 2$ satisfies $2 \geq 2$ ✔
- $Z = 3$: needs $Y \geq 3$ ✗ $\rightarrow$ remove 3

$$D_Z = \{2\}$$

Final Domains after AC-3: $D_X = \{1, 2\}, \quad D_Y = \{2\}, \quad D_Z = \{2\}$
Consistent Assignments: $(X, Y, Z) = (1, 2, 2)$ or $(2, 2, 2)$
Observation: Arc consistency reduced domains but did not force a single solution.

Backtracking: The Problem

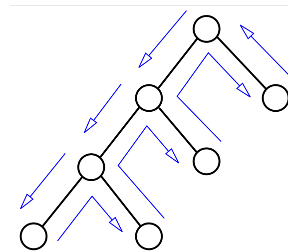
Chronological Backtracking - What Goes Wrong?

How it works

- Always backtracks to the **most recent** assignment that was made
- Undoes assignments one by one in reverse order
- Tries alternative values systematically

The problem

- The most recent variable may **not be the cause** of the failure
- Wastes time trying values for **irrelevant** variables
- Explores unnecessary parts of search tree



Chronological Backtracking

Intelligent Backtracking: The Solution

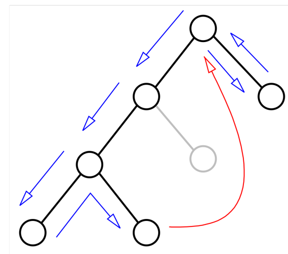
Backjumping - Skip the Irrelevant Variables

Key Idea:

- Track **why** each failure occurs
- Identify which variables **actually caused** the conflict
- Jump back directly to the **culprit variable**
- Skip all irrelevant variables in between

Conflict Set:

- For each variable, maintain its **conflict set**
- **Conflict set** = set of previously assigned variables that removed values from current variable's domain
- When domain becomes empty, backjump to most recent variable in conflict set



Backjumping

Conflict-Directed Backjumping (CBJ) - Part 1

CBJ: The Most Powerful Form of Intelligent Backtracking

Key Idea: Unlike chronological backtracking, CBJ remembers which variables caused the conflict and jumps directly to the culprit!

How CBJ Works:

1. **Initialize:** Start with empty conflict sets: $conf(X_i) = \emptyset$ for all variables
2. **Track conflicts:** During constraint propagation (e.g., forward checking), when a value is removed from $Domain(X_i)$ due to an assignment to X_j, add X_j to $conf(X_i)$
3. **Assign variable:** Select next unassigned variable X_i and try to assign a value from its domain
4. **Detect domain wipeout:** If $Domain(X_i) = \emptyset$ (no values left), we have a dead-end
5. **Smart backjump:** Instead of going back to X_{i-1}, find the most recent variable in $conf(X_i)$, call it X_j, and backjump directly to X_j (this skips variables between X_j and X_i that didn't cause the conflict)

Conflict-Directed Backjumping (CBJ) - Part 2

How CBJ Works (continued):

6. **Update conflict information:** Before changing X_j, update its conflict set:

$$conf(X_j) \leftarrow conf(X_j) \cup (conf(X_i) \setminus \{X_j\})$$

   This merges all conflicts from X_i into X_j (except X_j itself) to preserve conflict information
7. **Try new value:** Try a different value for X_j and continue forward search
8. **Terminate:**
   - **Success:** If all variables are consistently assigned, return solution
   - **Failure:** If we backjump past X_1 (first variable), no solution exists

Advantages:

- Can jump over **multiple** irrelevant assignments at once
- Significantly reduces search tree size
- Works well with forward checking and arc consistency
- Nearly always better than chronological backtracking

CBJ Example: Mini Sudoku (4x4)

Problem: Fill a 4×4 grid with numbers 1-4 such that:

- Each row contains 1, 2, 3, 4 (no repeats)
- Each column contains 1, 2, 3, 4 (no repeats)
- Each 2 × 2 box contains 1, 2, 3, 4 (no repeats)

Initial Puzzle:

1		3	
	4		
		1	
			2

Blue cells = given values (fixed)

Given values | Our assignments | Not yet assigned

Variables to assign:

- X_1 = cell (1,2) - row 1, col 2
- X_2 = cell (1,4) - row 1, col 4
- X_3 = cell (2,1) - row 2, col 1
- X_4 = cell (2,3) - row 2, col 3
- X_5 = cell (2,4) - row 2, col 4
- ...
- X_{12} = cell (4,3) - row 4, col 3

Ordering: Fill left-to-right, top-to-bottom

Domain: Each variable can be {1, 2, 3, 4}

Making Assignments (1/2)

Let's start filling in values.
Assignment sequence:

1. X_1 (1,2): Try 2 ✔
   - Row 1 has: 1, 2, 3, ? → 2 is valid!
   - Col 2 has: 2, 4, ?, ? → 2 is valid!
   - Top-left box: 1, 2, ?, 4 → 2 is valid!
2. X_2 (1,4): Try 4 ✔
   - Row 1 has: 1, **2**, 3, 4 → 4 is valid!
   - Col 4 has: 4, ?, ?, 2 → 4 is valid!
   - Top-right box: 3, 4, ?, ? → 4 is valid!
3. X_3 (2,1): Try 3 ✔
   - Row 2 has: 3, 4, ?, ? → 3 is valid!
   - Col 1 has: 1, 3, ?, ? → 3 is valid!
   - Top-left box: 1, **2**, 3, 4 → 3 is valid!
4. X_4 (2,3): Try 2 ✔
   - Row 2 has: **3**, 4, 2, ? → 2 is valid!
   - Col 3 has: 3, 2, 1, ? → 2 is valid!
   - Top-right box: 3, **4**, 2, ? → 2 is valid!

1	2	3	4
3	4	2	
		1	
			2

Current State after 4 assignments

Making Assignments (2/2)

Assignment sequence (continued):

5. X_5 (2,4): Try 1 ✔
   - Row 2 has: **3**, 4, 2, 1 → 1 is valid!
   - Col 4 has: **4**, 1, ?, 2 → 1 is valid!
   - Top-right box: 3, **4**, **2**, 1 → 1 is valid!
6. X_6 (3,1): Try 2 ✔
   - Row 3 has: 2, ?, 1, ? → 2 is valid!
   - Col 1 has: 1, **3**, 2, ? → 2 is valid!
   - Bottom-left box: 2, ?, ?, ? → 2 is valid!
7. X_7 (3,2): Try 3 ✔
   - Row 3 has: **2**, 3, 1, ? → 3 is valid!
   - Col 2 has: **2**, 4, 3, ? → 3 is valid!
   - Bottom-left box: **2**, 3, ?, ? → 3 is valid!

1	2	3	4
3	4	2	1
2	3	1	
			2

Current State after 7 assignments

The Conflict Occurs!

Now let's try to assign X_8 (cell (3,4)).
Checking options for X_8:

- Try 1? ✗
  - Column 4 already has 1 (from X_5)
  - Conflict!
- Try 2? ✗
  - Column 4 already has 2 (given)
  - Conflict!
- Try 3? ✗
  - Row 3 already has 3 (from X_7)
  - Conflict!
- Try 4? ✗
  - Column 4 already has 4 (from X_2)
  - Conflict!

X_8 has NO valid values!
Conflict Detected:

1	2	3	4
3	4	2	1
2	3	1	?
			2

Dead-end at X_8
No valid values
left in domain

Need to backtrack!

Who Caused This Conflict?

Let's trace why X_8 failed:

Constraint Analysis: Why each value failed:

- **Value 1:**
  - Column 4 has 1 (from X_5) $\rightarrow$ X_5 **blocks value 1**.
  - Row 3 has 1 $\rightarrow$ **given value blocks 1**
- **Value 2:**
  - Row 3 has 2 (from X_6) $\rightarrow$ X_6 **blocks value 2**.
  - Column 4 has 2 $\rightarrow$ **given value blocks 2**
- **Value 3:**
  - Row 3 has 3 (from X_7) $\rightarrow$ X_7 **blocks value 3**.
- **Value 4:**
  - Column 4 has 4 (from X_2) $\rightarrow$ X_2 **blocks value 4**.

1	2	3	4
3	4	2	1
2	3	1	?
			2

Conflict Set:
$conf(X_8) = \{X_2, X_5, X_6, X_7\}$

- Most recent variable that caused conflict: X_7

Backjump to the Culprit

Backjumping Steps

Backjump to X_7:

- From conflict set $conf(X_8) = \{X_2, X_5, X_6, X_7\}$, backjump to most recent variable: X_7
- Update conflict set for X_7:

$$conf(X_7) \leftarrow conf(X_7) \cup \{conf(X_8) \setminus \{X_7\}\} = \emptyset \cup \{X_2, X_5, X_6\}$$
$$= \{X_2, X_5, X_6\}$$

- Now try a different value for X_7
  - $X_7 = 1$? ✗ (Row 3 has 1)
  - $X_7 = 2$? ✗ (Column 2 has 2)
  - $X_7 = 4$? ✗ (Column 2 has 4)
- X_7 has no valid values! ✗
- Backjump to most recent in conflict set:

1	2	3	4
3	4	2	1
2	?	1	
			2

Backjumped to X_7

Continuing the Backjump

Backjump to X_6:

- From conflict set $conf(X_7) = \{X_2, X_5, X_6\}$, backjump to most recent variable: X_6
- Update conflict set for X_6:

$$conf(X_6) \leftarrow conf(X_6) \cup \{conf(X_7) \setminus \{X_6\}\} = \emptyset \cup \{X_2, X_5\} = \{X_2, X_5\}$$

- Now try a different value for X_6
  - $X_6 = 1$? ✗ (Column 1 has 1)
  - $X_6 = 3$? ✗ (Column 1 has 3)
  - $X_6 = 4$? ✔ Try this value!
- Assignment $X_6 = 4$ is valid, continue search from here

1	2	3	4
3	4	2	1
?		1	
			2

Backjumped to X_6

Outline

1. Preview
2. CSPs: Definition and Concepts
3. Classic CSP Examples
4. CSP Solution Techniques
5. Local Search for CSPs
6. CSP Problem Structure
7. Real-World Applications
8. Summary and Key Takeaways

Local Search for CSPs

Idea: Use local search instead of systematic search

Approach:

- Start with a **complete assignment** (possibly inconsistent)
- Iteratively **improve** the assignment by changing variable values
- Goal: **minimize** the number of constraint violations

Advantages:

- ✓ Can handle **very large** CSPs
- ✓ Finds solutions **quickly** in practice
- ✓ Works well for **optimization** CSPs

Disadvantages:

- ✗ Not **complete** - may not find solution
- ✗ Can get stuck in **local minima**

Local Search for CSPs: Min-Conflicts Heuristic

Min-Conflicts Algorithm

1. Start with a random complete assignment
2. Repeat until solution found or max iterations:
   - Select a **conflicted variable** (random or most conflicts)
   - Choose value that **minimizes conflicts** with other variables
   - Assign that value

Example: N-Queens

- Initial: Place all queens randomly
- Count conflicts for each queen
- Pick queen with most conflicts
- Move it to row with fewest conflicts in its column
- Repeat until no conflicts

Remarkably effective: Solves million-queens in under a minute!

Min-Conflicts Pseudocode

Algorithm 4: Min-Conflicts Algorithm

Input: CSP, maximum number of iterations
Output: Solution or failure

```
1  Function: MinConflicts(csp, max_iterations)
2      current ← random complete assignment for csp ;                    /* initial state */
3      for i ← 1 to max_iterations do
4          if Conflicts(current) = 0 then
5              return current ;                                          /* solution found */
6          end
7          var ← randomly select conflicted variable from current ;      /* select variable to change */
8          value ← MinConflictValue(var, current, csp) ;                 /* select value minimizing conflicts */
9          current[var] ← value ;                                        /* update assignment */
10     end
11     return failure ;                                                  /* max iterations exceeded */
12 end
13 Function: MinConflictValue(var, current, csp)
14     return value that minimizes conflicts for var given current ;
15 end
```

Min-Conflicts Example: 4-Queens

Step 0 — Initial Random State (corrected)
We use the convention $(Q_1, Q_2, Q_3, Q_4) =$ (row of queen in column 1,2,3,4).

Initial placement:

$$(Q_1, Q_2, Q_3, Q_4) = (1, 4, 2, 3)$$

Q_1	-	-	-
-	-	Q_3	-
-	-	-	Q_4
-	Q_2	-	-

Conflicts: One attacking pair: Q_3 attacks Q_4 (diagonal). (Per-queen conflicts: $Q_3 : 1,\ Q_4 : 1,\ Q_1 : 0,\ Q_2 : 0$.)

Min-Conflicts Example: 4-Queens

Iteration 1 — Move a conflicted queen
Pick a conflicted queen (either Q_3 or Q_4). Move Q_3 to a row minimizing conflicts. We choose to move Q_3 from row 2 to row 1.
New placement:

$$(Q_1, Q_2, Q_3, Q_4) = (1, 4, 1, 3)$$

Q_1	-	Q_3	-
-	-	-	Q_4
-	-	-	-
-	Q_2	-	-

Conflicts now: One attacking pair: Q_1 attacks Q_3 (same row). (Per-queen conflicts: $Q_1 : 1,\ Q_3 : 1,\ Q_2 : 0,\ Q_4 : 0$.)

Min-Conflicts Example: 4-Queens

Iteration 2 — Resolve the remaining conflict

Choose the conflicted queen (here Q_1) and move it to the row that minimizes conflicts.
Move Q_1 from row 1 to row 2.
New placement (solution):

$$(Q_1, Q_2, Q_3, Q_4) = (2, 4, 1, 3)$$

-	-	Q_3	-
Q_1	-	-	Q_4
-	-	-	-
-	Q_2	-	-

Conflicts: 0 — valid solution reached.

Final configuration: $(Q_1, Q_2, Q_3, Q_4) = (2, 4, 1, 3)$, which is one of the 4-Queens solutions.

Outline

1. Preview
2. CSPs: Definition and Concepts
3. Classic CSP Examples
4. CSP Solution Techniques
5. Local Search for CSPs
6. CSP Problem Structure
7. Real-World Applications
8. Summary and Key Takeaways

Exploiting Problem Structure

Many CSPs have special structure that can be exploited:

1. Independent Subproblems

- CSP breaks into **disconnected components**
- Solve each component **independently**
- Combine solutions
- Time complexity: linear instead of exponential!

2. Tree-Structured CSPs

- Constraint graph is a **tree** (no loops)
- Can be solved in **linear time** $O(n \cdot d^2)$ instead of $O(d^n)$ where n = number of variables, d = domain size
- Algorithm: Arc consistency + topological ordering

3. Nearly Tree-Structured CSPs

- Remove small set of variables to get tree
- Try all assignments to removed variables
- Solve resulting tree-structured CSP

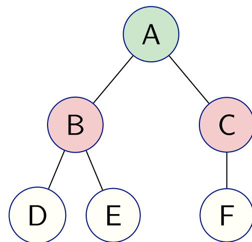
Tree-Structured CSPs

Tree CSP Algorithm:

1. Choose a root variable
2. Order variables from root to leaves
3. **Backward pass:** For $i = n$ down to 2:
   - Make arc $(Parent(X_i), X_i)$ consistent
4. **Forward pass:** For $i = 1$ to n:
   - Assign X_i consistently with $Parent(X_i)$

Complexity:

- $O(n \cdot d^2)$ where n = number of variables, d = domain size
- Compare to general CSP: $O(d^n)$!



Tree-structured CSP example

Cutset Conditioning

For nearly tree-structured CSPs:

Cycle Cutset

- A set of variables whose removal makes the constraint graph a tree
- **Smaller cutset** = **better performance**

Algorithm:

1. Find a small cycle cutset S
2. For each assignment to variables in S:
   - Remove S from CSP
   - Solve resulting tree CSP in $O(nd^2)$ time
   - If solution found, combine with S assignment

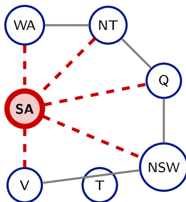
Complexity: $O(d^c \cdot (n-c)d^2)$ where c = cutset size

- Much better than $O(d^n)$ if $c \ll n$

Cutset Conditioning: Australia Map Example

Key Idea: Remove a variable (cutset) to break cycles and create a tree structure

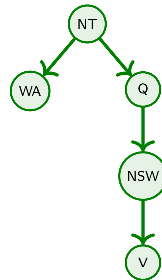
Original Constraint Graph



SA is the cutset

Dashed red = edges to SA

After Removing SA



Tree structure!

No cycles → efficient solving

Outline

1. Preview
2. CSPs: Definition and Concepts
3. Classic CSP Examples
4. CSP Solution Techniques
5. Local Search for CSPs
6. CSP Problem Structure
7. Real-World Applications
8. Summary and Key Takeaways

CSP Applications

CSPs are used in many **real-world domains**

1. Scheduling

- Course timetabling, employee scheduling
- Variables: tasks/courses, Domains: time slots
- Constraints: no conflicts, resource limits

2. Resource Allocation

- Frequency assignment in mobile networks
- Variables: transmitters, Domains: frequencies
- Constraints: avoid interference

3. Configuration

- Hardware/software configuration
- Variables: components, Domains: compatible options
- Constraints: compatibility, dependencies

Application Example: Course Timetabling

Problem: Schedule courses to time slots and rooms

CSP Formulation:

- **Variables:** Courses (CS101, CS201, MATH101, ...)
- **Domains:** (Time slot, Room) pairs
  - Times: Mon-9am, Mon-10am, ...
  - Rooms: A101, A102, B201, ...
- **Constraints:**
  - Same room, different times
  - Same instructor, different times
  - Prerequisites properly ordered
  - Room capacity $\geq$ class size

Example Solution:

Course	Time	Room
CS101	Mon 9am	A101
CS201	Mon 10am	A101
MATH101	Mon 9am	A102
PHYS101	Mon 11am	B201

Techniques used:

- Forward checking
- MRV variable ordering
- Arc consistency

Outline

1. Preview
2. CSPs: Definition and Concepts
3. Classic CSP Examples
4. CSP Solution Techniques
5. Local Search for CSPs
6. CSP Problem Structure
7. Real-World Applications
8. Summary and Key Takeaways

CSP Summary — Conceptual Foundations

Key Takeaways

- **Constraint Satisfaction Problems (CSPs)** provide a **unified and powerful framework** for modeling **combinatorial problems**.
- Each CSP is defined by three core components:
  - **Variables** – unknowns to assign values to
  - **Domains** – possible values for each variable
  - **Constraints** – relations restricting valid combinations
- Examples include: **map coloring**, **scheduling**, **puzzle solving**, and **resource allocation**.

CSP Summary — Solution Techniques

Key Takeaways

- **Systematic Search Methods:**
  - **Backtracking search** – explores assignments recursively
  - **Constraint propagation** – prunes inconsistent values early
- **Heuristic Enhancements:**
  - **Variable ordering:** choose most constrained variable first (MRV)
  - **Value ordering:** least constraining value heuristic
- Together, these techniques **reduce the search space** and **increase efficiency** of finding consistent assignments.

CSP Summary — Structure and Applications

Key Takeaways

- **Exploiting structure** in the constraint graph (e.g., **independent subproblems**, **tree-structured CSPs**) can drastically reduce computation.
- **Local search methods** such as **Min-Conflicts** are effective for large-scale or real-time CSPs.
- **Applications:** **scheduling**, **planning**, **configuration**, **map coloring**, and **resource allocation**.

Outline

1. Preview
2. CSPs: Definition and Concepts
3. Classic CSP Examples
4. CSP Solution Techniques
5. Local Search for CSPs
6. CSP Problem Structure
7. Real-World Applications
8. Summary and Key Takeaways

Note on `python-constraint` Library

`python-constraint` **Library:** A Python library for solving CSPs

- **Installation:**
  - Install via pip: `pip install python-constraint`
- **Key Features:**
  - Define variables and their domains
  - Add constraints (binary, n-ary, custom)
  - Solve CSPs using backtracking search
- **Basic Usage Example:**
  - Create a problem instance
  - Add variables and domains
  - Add constraints
  - Call the solver to find solutions
- **Official Documentation:** https://github.com/python-constraint/python-constraint

Outline

1. Preview
2. CSPs: Definition and Concepts
3. Classic CSP Examples
4. CSP Solution Techniques
5. Local Search for CSPs
6. CSP Problem Structure
7. Real-World Applications
8. Summary and Key Takeaways

Warning

Homeworks 5.1 and 5.2 should be submitted on Blackboard before the next course session!

Homework 5.1

Solving Small Constraint Satisfaction Problems

Using the library `python-constraint`, model and solve the following small CSPs:

1. **Map Coloring:** Color the map of Australia using 3 colors such that no adjacent regions share the same color. Submit your well-commented code and ready to run named `small_map_coloring_yourUniversityID.py`.
2. **Sudoku Puzzle:** Solve a given 4x4 Sudoku puzzle using CSP techniques. Submit your well-commented code and ready to run named `small_sudoku_yourUniversityID.py`.
3. **N-Queens Problem:** Place 4 queens on a 4x4 chessboard such that no two queens threaten each other. Submit your well-commented code and ready to run named `small_nqueens_yourUniversityID.py`.

Homework 5.2

Solving Medium Constraint Satisfaction Problem

Using the library `python-constraint`, model and solve the following Scheduling CSP: Schedule 20 courses into 5 time slots and 4 rooms such that no two courses with overlapping students are scheduled at the same time and room capacities are not exceeded.

Data for courses, students, time slots, and room capacities are provided via following link course_data.json.

Submit your well-commented code and ready to run named `big_course_scheduling_yourUniversityID.py`.

End of Chapter 5