Artificial Intelligence

Lecture 6: Constraint Satisfaction Problems

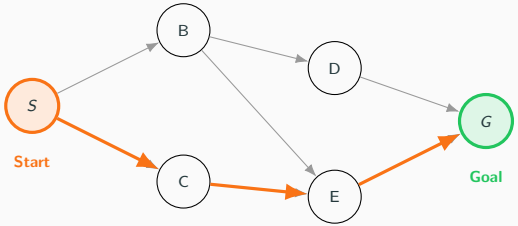
Prof. Antonio Khalil Moretti

Week 7

SCIS 432
Spelman College

Where We've Been: Search Paradigms

Search as Graph Exploration



Key idea: find a path from start to goal

What we've covered:

- **Uninformed search:** BFS, DFS
- **Informed search:** A* with heuristics
- **Adversarial:** Minimax, Alpha-Beta
- **Stochastic:** Expectimax, MCTS

Common thread:

- States = nodes in a graph
- Actions = edges between states
- Goal = reach a particular state
- Solution = sequence of actions

A Different Kind of Problem

When the goal isn't a destination, it's a configuration

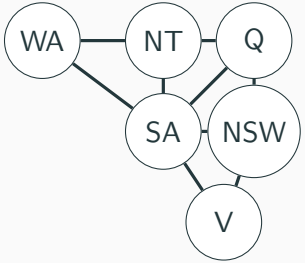
Sudoku:

5	3	?	?	7	?			
6			1	9	5			
	9	8					6	

Fill the grid so:

- Each row has 1–9
- Each column has 1–9
- Each 3×3 box has 1–9

Map Coloring:



Color each region so adjacent regions have different colors

Could We Use Regular Search?

Yes, but it's not the best idea

Let's try modeling Sudoku as a search problem:

- **State:** current assignment of numbers to cells
- **Action:** fill in one empty cell with a number
- **Goal test:** all cells filled and all constraints satisfied

Could We Use Regular Search?

Yes, but it's not the best idea

Let's try modeling Sudoku as a search problem:

- **State:** current assignment of numbers to cells
- **Action:** fill in one empty cell with a number
- **Goal test:** all cells filled and all constraints satisfied

Problems with this approach:

Branching Factor

- 81 cells in Sudoku
- Each can be 1–9
- Early branching factor: $\approx 81 \times 9 = 729$
- State space: potentially $9^{81} \approx 10^{77}$ states

Wasted Effort

- Standard search treats all states equally
- But most paths violate constraints
- We discover violations *late* (at goal test)
- No mechanism to detect conflicts *early*

CSPs: A Different Representation

Instead of states and actions, we think in terms of:

Variables Things we need to assign values to

Example (Sudoku): Each cell is a variable
Example (Map coloring): Each region is a variable

Domains Possible values for each variable

Sudoku: $\{1, 2, 3, 4, 5, 6, 7, 8, 9\}$
Map coloring: {Red, Green, Blue}

Constraints Rules that restrict which combinations of values are allowed

Sudoku: row/column/box must have distinct values
Map coloring: adjacent regions $\neq$ same color

Goal Find an assignment of values to variables that satisfies all constraints

This representation enables specialized algorithms that:

- Detect conflicts immediately
- Reduce domains incrementally
- Prune large parts of the search space early

Formal Definition of a CSP

A **Constraint Satisfaction Problem** consists of three components:

1. **Variables:** $X = \{X_1, X_2, \ldots, X_n\}$
   - The things we need to assign values to
2. **Domains:** $D = \{D_1, D_2, \ldots, D_n\}$
   - D_i is the set of allowable values for variable X_i
   - Can be finite (e.g., colors) or infinite (e.g., real numbers)
3. **Constraints:** $C = \{C_1, C_2, \ldots, C_m\}$
   - Each constraint C_k involves some subset of variables
   - Specifies allowable combinations of values for those variables

Solution: An assignment of values to all variables such that all constraints are satisfied.

(Sometimes we want *any* solution, sometimes we want the *best* solution according to some objective.)

Example 1: Map Coloring (Australia)

Variables:

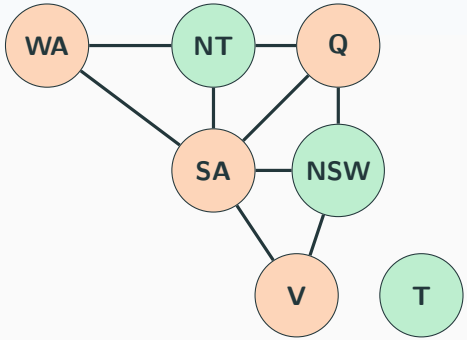
$$\{WA, NT, Q, NSW, V, SA, T\}$$

Domains: For each variable,

$$D_i = \{\text{Red}, \text{Green}, \text{Blue}\}$$

Constraints: Adjacent regions must have different colors

- $WA \neq NT$
- $WA \neq SA$
- $NT \neq SA$
- $NT \neq Q$
- $SA \neq Q$
- etc.



One solution shown above:
WA=Red, NT=Green, Q=Red, NSW=Green, V=Red, SA=Red, T=Green

Example 2: N-Queens

Place N queens on an N×N chessboard so none attack each other

Variables: $Q_1, Q_2, \ldots, Q_n$

- Each Q_i represents the queen in row i

Domains: For each Q_i,

$$D_i = \{1, 2, \ldots, n\}$$

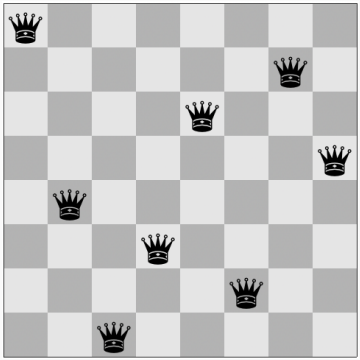
(column position of queen in row i)

Constraints:

- **No two queens in same column:** $Q_i \neq Q_j$
- **No two queens on same diagonal:**

$$|Q_i - Q_j| \neq |i - j|$$

8-Queens example:



Why CSP representation helps: Constraints eliminate illegal placements immediately.

Types of Constraints

Constraints can involve different numbers of variables:

Unary Constraints Restrict the value of a single variable

Example: "WA cannot be green"
$WA \neq$ Green

These can be handled by simply removing values from domains

Binary Constraints Relate two variables

Example: $WA \neq NT$ (most common type)

Can be represented as a **constraint graph**

Higher-Order Constraints Involve three or more variables

Example (Sudoku): "All cells in row 1 must be different"

Example (Cryptarithmetic):

$$\text{SEND} + \text{MORE} = \text{MONEY}$$

Global Constraints Special constraints with efficient propagation algorithms

Example: $\texttt{Alldifferent}(X_1, \ldots, X_n)$
Example: $\texttt{Atmost}(3, X_1, \ldots, X_n, \text{Red})$

Most CSPs can be reduced to binary constraints (though this may add variables).

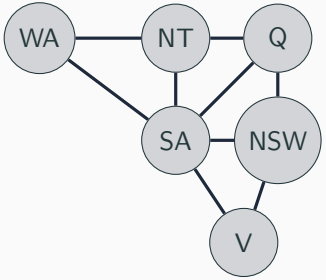
Constraint Graphs

Visualizing the structure of CSPs

For CSPs with **binary constraints**, we can draw a **constraint graph**:

- **Nodes** = variables
- **Edges** = constraints between variables

Map coloring constraint graph:



Graph structure reveals:

- **Connectivity:** SA is highly connected (hardest to color)
- **Independence:** Tasmania would be disconnected
- **Cycles:** The graph has many cycles

Tree-structured CSPs (no cycles) can be solved efficiently in $O(nd^2)$ time!

General CSPs are NP-complete.

Factor Graphs

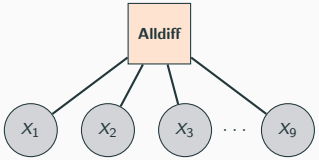
Representing higher-order constraints

Problem: Constraint graphs can only show binary constraints.

What about constraints involving 3+ variables?

Solution: Factor graphs — a bipartite graph with two types of nodes:

- **Variable nodes** (circles): represent variables
- **Factor nodes** (squares): represent constraints
- **Edges:** connect factors to the variables they constrain



The factor node represents the constraint "$X_1, X_2, \ldots, X_9$ all different"

Example: Sudoku row constraint

"All cells in row 1 must be different"

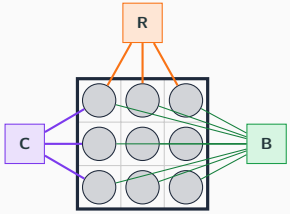
Sudoku as a Factor Graph

Partial illustration — full graph would be very dense

Variables: 81 cells (9×9 grid)

Factors: 27 Alldiff constraints

- 9 row constraints
- 9 column constraints
- 9 box (3×3) constraints



R row C col B box

Each factor connects 9 variables

Example:
Row 1 factor connects cells $(1,1), (1,2), \ldots, (1,9)$
Box 1 factor connects cells $(1,1), (1,2), (1,3),$
$(2,1), (2,2), (2,3), (3,1), (3,2), (3,3)$

Observation: Most variables participate in multiple factors.
This high connectivity is what makes Sudoku challenging!

Solving CSPs: Backtracking Search

Key idea: Assign variables one at a time; check constraints after each assignment; backtrack the moment a constraint is violated.

```
1:  function BACKTRACK(assignment)
2:      if assignment is complete then
3:          return assignment
4:      end if
5:      X ← SELECTUNASSIGNEDVAR
6:      for each value v in Domain(X) do
7:          if v consistent with assignment then
8:              add {X = v} to assignment
9:              result ← BACKTRACK(assignment)
10:             if result ≠ failure then
11:                 return result
12:             end if
13:             remove {X = v} from assignment
14:         end if
15:     end for
16:     return failure
```

The algorithm is systematic:

- Tries values left-to-right in the domain
- Checks every applicable constraint after each new assignment
- If a constraint is broken, immediately tries the next value
- If all values for a variable fail, go back to the *previous* variable and try its next value

We will walk through a complete example on the next slides.

Backtracking Example: Problem Setup

Three variables, three constraints, domain $\{1, 2, 3\}$

Variables & Domains

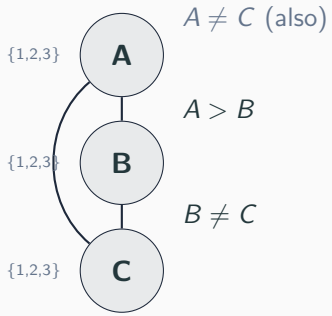
A, B, C — each can take values $\{1, 2, 3\}$

Constraints

1. $A > B$
2. $B \neq C$
3. $A \neq C$

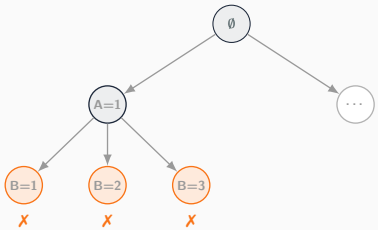
We will assign variables in order A, then B, then C, always trying values $1, 2, 3$ left-to-right.

After each new assignment we immediately check *all* constraints that involve only assigned variables.



Backtracking Walk-Through: A = 1 Branch

Every value of B fails immediately – backtrack to A



Try A = 1.
Partial assignment $\{A = 1\}$. No constraint broken yet (B and C unknown).

Now try B = 1.
Check $A > B$: $1 > 1$ is **false.** Backtrack.

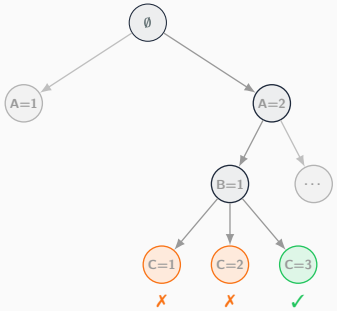
Try B = 2.
Check $A > B$: $1 > 2$ is **false.** Backtrack.

Try B = 3.
Check $A > B$: $1 > 3$ is **false.** Backtrack.

All values of B exhausted.
Go back to A and try the next value.

Backtracking Walk-Through: A = 2, B = 1 Branch

First partial assignment that survives – now try values for C



Try A = 2.
No constraints fire yet. Move to B.

Try B = 1.
$A > B$: $2 > 1$ **OK.** Move to C.

Try C = 1.
$B \neq C$: $1 \neq 1$ **false.** Backtrack.

Try C = 2.
$B \neq C$: $1 \neq 2$ OK.
$A \neq C$: $2 \neq 2$ **false.** Backtrack.

Try C = 3.
$B \neq C$: $1 \neq 3$ **OK.**
$A \neq C$: $2 \neq 3$ **OK.**

All constraints satisfied. ✓

Backtracking Walk-Through: Solution Found

Solution: $A = 2,\ B = 1,\ C = 3$

Verify all constraints:

- $A > B$: $2 > 1$ ✓
- $B \neq C$: $1 \neq 3$ ✓
- $A \neq C$: $2 \neq 3$ ✓

Assignments tried:

Assignment	Result	Reason
$A = 1, B = 1$	✗	$A > B$ fails
$A = 1, B = 2$	✗	$A > B$ fails
$A = 1, B = 3$	✗	$A > B$ fails
$A = 2, B = 1, C = 1$	✗	$B \neq C$ fails
$A = 2, B = 1, C = 2$	✗	$A \neq C$ fails
$A = 2, B = 1, C = 3$	✓	**Solution!**

Compare to exhaustive search:

Total possible complete assignments: $3^3 = 27$
Backtracking explored only **6 partial assignments**.

Once $A = 1$ is shown to fail every B value, all completions of $A = 1$ are skipped entirely.

Key takeaway

Constraints prune the tree *early*, before we even reach a complete assignment. The earlier a violation is detected, the more work is saved.

Improving Backtracking: Variable Ordering

Which variable should we assign next?

Question: When choosing the next unassigned variable, does order matter?

Improving Backtracking: Variable Ordering

Which variable should we assign next?

Question: When choosing the next unassigned variable, does order matter?

Answer: Yes! Some orderings lead to many fewer backtracks.

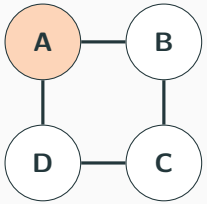
Minimum Remaining Values (MRV)

Choose the variable with the **fewest legal values** remaining

Intuition: "Fail fast"
If a variable has no legal values, we want to discover that as early as possible

Also called the "most constrained variable" heuristic

Example:



All unassigned variables have 2 legal values.
MRV doesn't help here — need another heuristic!

Variable Ordering: Degree Heuristic

Breaking ties in MRV

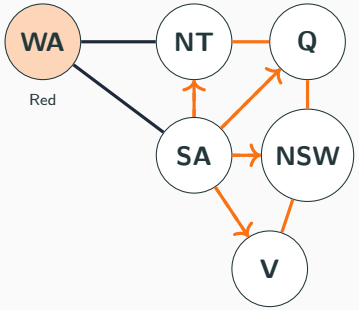
Degree Heuristic Among variables with the same MRV, choose the one with the **most constraints** on remaining variables

Intuition: This variable is most likely to reduce future choices for other variables

Example (Map Coloring):
After assigning WA=Red:

- SA has 5 neighbors
- NT has 3 neighbors
- Q has 3 neighbors

Choose SA next (highest degree)



SA (highlighted) has most unassigned neighbors

Combined strategy: Use MRV as the primary heuristic, degree heuristic to break ties.

Value Ordering: Least Constraining Value

Which value should we try first for a variable?

Question: We've chosen variable X to assign. Which value from its domain should we try first?

Least Constraining Value (LCV)

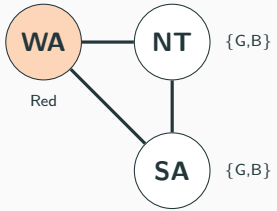
Choose the value that **rules out the fewest** values in the remaining variables

Intuition: Leave maximum flexibility for future assignments

Opposite philosophy to MRV!

- MRV: fail fast (choose hardest variable)
- LCV: succeed (choose least restrictive value)

Example:



Assigning WA (decide between G and B):

- Green: eliminates 2 values total from NT & SA
- Blue: eliminates 2 values total from NT & SA

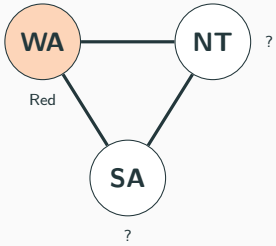
In this case, LCV doesn't help (tie).

Beyond Backtracking: Constraint Propagation

Use constraints to reduce domains before searching

Key idea: When we assign a value to a variable, we can immediately infer restrictions on other variables.

Example:



Before assignment:
NT: {Red, Green, Blue}
SA: {Red, Green, Blue}
After WA=Red:
NT: {Green, Blue}
SA: {Green, Blue}

This is called forward checking:

Whenever we assign $X = v$:

- Remove v from domains of all unassigned variables that have a constraint with X

Early failure detection:
If any variable's domain becomes empty, backtrack immediately — no need to continue down this branch!

Arc Consistency (AC-3): More Powerful Constraint Propagation

Definition: An arc $X_i \to X_j$ is **arc-consistent** if, for every value x in the domain of X_i, there exists at least one value y in the domain of X_j that satisfies the constraint.

Algorithm AC-3:

```
1: function AC-3(csp)
2:     queue ← all arcs in csp
3:     while queue not empty do
4:         (X_i, X_j) ← RemoveFirst(queue)
5:         if Revise(X_i, X_j) then
6:             if Domain(X_i) is empty then
7:                 return false
8:             end if
9:             add all arcs (X_k, X_i) to queue,
10:        where X_k ≠ X_j
11:        end if
12:    end while
13:    return true
14: end function
```

Why two arcs per constraint?

Each binary constraint $X_i \neq X_j$ becomes **two directed arcs**:

- Arc $X_i \to X_j$: prune domain of X_i
- Arc $X_j \to X_i$: prune domain of X_j

The algorithm only modifies the *left-hand side* variable's domain. Both directions are needed to catch all pruning opportunities.

Time complexity: $O(cd^3)$
c = constraints, d = domain size

AC-3 Example: Problem Setup

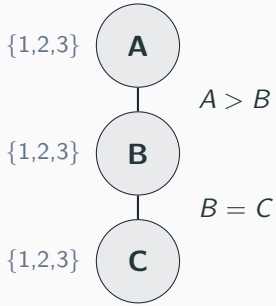
Variables A, B, C with domain $\{1, 2, 3\}$; constraints $A > B$ and $B = C$

Variables & Domains

A, B, C — each domain $= \{1, 2, 3\}$

Constraints

1. $A > B$
2. $B = C$



Step 1: Build the arc agenda.

Every binary constraint becomes **two directed arcs**:

From $A > B$: $(A > B)$ and $(B < A)$

From $B = C$: $(B = C)$ and $(C = B)$

Initial agenda:

$\{(A > B),\ (B < A),\ (B = C),\ (C = B)\}$

AC-3 Walk-Through: Arc (A > B)

For every value of A, is there a legal value in B?

Domains at start of this step:

A: $\{1, 2, 3\}$
B: $\{1, 2, 3\}$
C: $\{1, 2, 3\}$

Dequeue $(A > B)$. For each value of A, is there *any* B satisfying $A > B$?

$A =$	Support in B?	Action
1	Need $B < 1$: none	**Remove 1 from A**
2	$B = 1$: yes	Keep
3	$B \in \{1, 2\}$: yes	Keep

A revised: $\{1, 2, 3\} \rightarrow \{2, 3\}$

Domain of A changed $\Rightarrow$ re-queue all arcs with **A on the right-hand side**.

Arc $(B < A)$ has A on the RHS and is already in the agenda. No new additions needed.

Updated domains:

A: $\{2, 3\}$
B: $\{1, 2, 3\}$
C: $\{1, 2, 3\}$

Remaining agenda:
$\{(B < A),\ (B = C),\ (C = B)\}$

AC-3 Walk-Through: Arc (B $<$ A)

For every value of B, is there a legal value in A?

Domains at start of this step:

A: $\{2, 3\}$
B: $\{1, 2, 3\}$
C: $\{1, 2, 3\}$

Dequeue $(B < A)$. For each value of B, is there *any* A satisfying $B < A$?

$B =$	Support in A?	Action
1	$A = 2$ or $A = 3$: yes	Keep
2	$A = 3$: yes	Keep
3	Need $A > 3$: none!	**Remove 3 from B**

B revised: $\{1, 2, 3\} \rightarrow \{1, 2\}$

Domain of B changed $\Rightarrow$ re-queue all arcs with **B on the right-hand side**.

$(C = B)$ has B on the RHS — already in agenda.
$(A > B)$ has B on the RHS — already processed, but B changed since then, so **add** $(A > B)$ **back**.

Updated domains:

A: $\{2, 3\}$
B: $\{1, 2\}$
C: $\{1, 2, 3\}$

Remaining agenda:
$\{(B = C), (C = B), (A > B)\}$

AC-3 Walk-Through: Arc (B = C)

For every value of B, is there a legal value in C? Checking left-hand side only

Domains at start of this step:

A: $\{2, 3\}$
B: $\{1, 2\}$
C: $\{1, 2, 3\}$

Dequeue $(B = C)$. For each value of B, is there *any* C satisfying $B = C$?

$B =$	Support in C?	Action
1	$C = 1$: yes	Keep
2	$C = 2$: yes	Keep

No change to B's domain.

B's domain did *not* change, so no arcs need to be re-queued.

Important: C has an extra value (3) that would not be matched. But we only check values of the *left-hand side* (B). The extra value in C is not a problem *from this arc's perspective* — it will be caught by arc $(C = B)$.

Updated domains: (unchanged)

A: $\{2, 3\}$
B: $\{1, 2\}$
C: $\{1, 2, 3\}$

Remaining agenda:
$\{(C = B), (A > B)\}$

AC-3 Walk-Through: Arc (C = B)

This is why we need both directions — C's domain gets pruned here

Domains at start of this step:

A: $\{2, 3\}$

B: $\{1, 2\}$

C: $\{1, 2, 3\}$

Dequeue $(C = B)$. For each value of C, is there *any* B satisfying $C = B$?

$C =$	Support in B?	Action
1	$B = 1$: yes	Keep
2	$B = 2$: yes	Keep
3	Need $B = 3$: none!	**Remove 3 from C**

C revised: $\{1, 2, 3\} \rightarrow \{1, 2\}$

This is the key asymmetry!

Arc $(B = C)$ found no problem: every value of B had support in C.
But arc $(C = B)$ reveals that $C = 3$ has *no* support in B.

Domain of C changed ⇒ re-queue arcs with C on the RHS. That is arc $(B = C)$ — add it back.

Updated domains:

A: $\{2, 3\}$

B: $\{1, 2\}$

C: $\{1, 2\}$

Remaining agenda:
$\{(A > B),\ (B = C)\}$

AC-3 Walk-Through: Final Two Arcs

Arc $(A > B)$ re-check:

Domains: A $= \{2, 3\}$, B $= \{1, 2\}$

$A =$	Support in B?	Action
2	$B = 1$: yes	Keep
3	$B \in \{1, 2\}$: yes	Keep

No change. No new arcs added.

Arc $(B = C)$ re-check:

Domains: B $= \{1, 2\}$, C $= \{1, 2\}$

$B =$	Support in C?	Action
1	$C = 1$: yes	Keep
2	$C = 2$: yes	Keep

No change.

Agenda is empty.

Final arc-consistent domains

A: {2, 3}
B: {1, 2}
C: {1, 2}

Started: $3^3 = 27$ possible assignments.
After AC-3: $2 \times 2 \times 2 = 8$.
Any backtracking search now has a much smaller space to explore.

What AC-3 detected without any search:

- $A = 1$ is impossible (no B could satisfy $A > B$)
- $B = 3$ is impossible (no A could satisfy $B < A$)
- $C = 3$ is impossible (no B could satisfy

Alternative Approach: Local Search

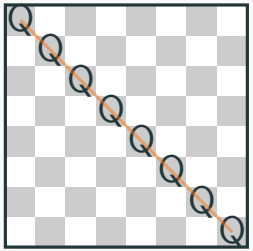
For problems where we just need any solution

Idea: Start with a complete (but possibly inconsistent) assignment, then iteratively improve it.

Min-Conflicts Algorithm:

```
1: function MinConflicts(csp, max_steps)
2:     current ← random complete assignment
3:     for i = 1 to max_steps do
4:         if current is a solution then
5:             return current
6:         end if
7:         X ← random conflicted variable
8:         v ← value that minimizes conflicts
9:         set X = v in current
10:    end for
11:    return failure
12: end function
```

Example: 8-Queens



Initial: all queens on diagonal (many conflicts)

Real-World CSP Applications

CSPs are everywhere in practice:

Scheduling Variables: Tasks, time slots, resources

Constraints:

- Precedence (task A before task B)
- Resource limits (one room, one time)
- Availability (professor teaches 9-5)

Examples:

- Course timetabling
- Airport gate assignment
- Sports league scheduling
- Manufacturing job shop scheduling

Configuration Variables: Components, parameters

Constraints:

- Compatibility (CPU socket matches motherboard)
- Requirements (GPU needs PSU $\geq$ 600W)
- Mutual exclusion (can't have both options)

Examples:

- Computer system configuration
- Car manufacturing options
- Network design
- Circuit board layout

Other Applications **Puzzles:** Sudoku, crosswords, logic puzzles, **Planning:** resource allocation, logistics, **Vision:** scene labeling, object recognition, **Biology:** protein folding, DNA sequencing

Example: Course Scheduling CSP

Problem: Schedule courses for a semester.

Variables:

- One variable per course
- Value = (time slot, room) pair

Domains:

- Time slots: MWF 9-10, MWF 10-11, TTh 9:30-11, etc.
- Rooms: CSB 201, CSB 204, MSC 100, etc.
- Domain size: ≈ 30 time slots $\times$ 20 rooms $= 600$

Example variable:

$CS\_432 \in \{$(MWF 9-10, CSB 201),
(MWF 10-11, CSB 201), $\ldots\}$

Constraints:

1. **Room capacity:**
   Enrollment $\leq$ room size
2. **No room conflicts:**
   Two courses can't use same room at same time
3. **Professor availability:**
   Prof. Smith doesn't teach TTh
4. **No student conflicts:**
   Required courses for same cohort at different times
5. **Course requirements:**
   CS 432 needs computer lab
6. **Preferences:**
   Spread courses evenly across week (soft constraint)

Note: Some constraints are *hard* (must satisfy), others are *soft* (preferences). This becomes a **Constraint Optimization Problem** (COP).

Summary: CSPs vs. Standard Search

	Standard Search	**CSPs**
Representation	States & actions	Variables & constraints
Goal	Reach a specific state	Any assignment satisfying constraints
Path	Matters (want shortest/cheapest)	Irrelevant
Search strategy	BFS, A*, etc.	Backtracking + inference
Pruning	Based on costs/heuristics	Based on constraint violations
Key optimization	Better heuristics	Earlier constraint propagation
When to use	Navigation, planning	Configuration, scheduling

Key insight: CSPs are a *specialized representation* that enables:

- More efficient search (early pruning via constraints)
- Domain-specific inference (AC-3, etc.)
- Clear separation of problem structure from search algorithm

CSPs and the $\exists/\forall$ View

Recall from Minimax: We used quantifiers to describe game search.

CSPs fit this framework too:

$$\exists X_1 \; \exists X_2 \; \cdots \; \exists X_n \quad : \quad \text{all constraints satisfied}$$

"There exist values for all variables such that all constraints hold."

This is a pure $\exists$ problem — no adversary (no $\forall$).

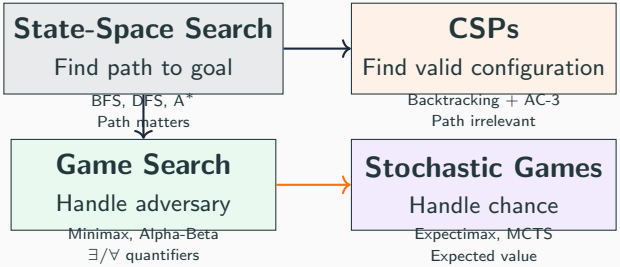
Backtracking = existential search

- Try to find *some* assignment that works
- If current choice fails, try another
- Stop when we find one that satisfies all constraints

Constraint propagation = inference

- Use logical deduction to narrow possibilities
- "If $X = v$, then Y cannot be w"
- Reduces search space before exploring

The Big Picture: Problem-Solving Paradigms



Each paradigm is suited to different problem types.

Recognizing which to use is a key AI skill!

What We Learned Today

CSP Basics

- Variables, domains, constraints
- Different from state-space search
- Constraint graphs & factor graphs
- Binary vs. higher-order constraints

Constraint Propagation

- Forward checking
- Arc consistency (AC-3)
- Detecting unsatisfiability early
- Reducing search space

Search Techniques

- Backtracking search
- Variable ordering: MRV, degree heuristic
- Value ordering: least constraining value
- Why these heuristics matter

Applications

- Scheduling (courses, airports)
- Configuration (systems, manufacturing)
- Puzzles (Sudoku, N-Queens)
- Many real-world problems

Key takeaway: CSPs give us a powerful way to represent and solve problems where the solution is a configuration, not a path.

Next Week

Coming up: Probabilistic Reasoning

- Reasoning under uncertainty
- Conditional probability and Bayes' rule
- Bayesian networks: representing joint distributions compactly
- Inference algorithms
- Applications: diagnosis, filtering, prediction

Reading: Russell & Norvig, Chapter 13 (Quantifying Uncertainty) and Chapter 14 (Probabilistic Reasoning)

Practice: Try solving a Sudoku puzzle by hand using CSP techniques: forward checking, MRV, and arc consistency!