Constraint Satisfaction Problems

General class of Problems: **Binary CSP**

Unary constraint arc.

Variable $V_i$ with values in domain $D_i$

Binary constraint arc

Unary constraints just cut down domains

This diagram is called a constraint graph
Constraint Satisfaction Problems

General class of Problems: Binary CSP

Unary constraint arc.

Variable $V_i$ with values in domain $D_i$

Binary constraint arc

Unary constraints just cut down domains

This diagram is called a constraint graph

Basic problem:

Find a $d_j \in D_i$ for each $V_i$ s.t. all constraints satisfied (finding consistent labeling for variables)
N-Queens as CSP
Classic “benchmark” problem

Place N queens on an N\times N chessboard so that none can attack the other.

**Variables**  
are board positions in N\times N chessboard

**Domains**  
Queen or blank

**Constraints**  
Two positions on a line (vertical, horizontal, diagonal) cannot both be Q
Line labelings as CSP

Label lines in drawing as convex (+), concave (-), or boundary (>).

Variables are line junctions

Domains are set of legal labels for that junction type

Constraints shared lines between adjacent junctions must have same label.
Scheduling as CSP

Choose time for activities e.g. observations on Hubble telescope, or terms to take required classes.

**Variables**
- are activities

**Domains**
- sets of start times (or “chunks” of time)

**Constraints**
1. Activities that use same resource cannot overlap in time
2. Preconditions satisfied
Graph Coloring as CSP

Pick colors for map regions, avoiding coloring adjacent regions with the same color.

**Variables**
- regions

**Domains**
- colors allowed

**Constraints**
- adjacent regions must have different colors
3-SAT as CSP
The original NP-complete problem

Find values for boolean variables A, B, C, ... that satisfy the formula.

<table>
<thead>
<tr>
<th>Variables</th>
<th>clauses</th>
</tr>
</thead>
<tbody>
<tr>
<td>Domains</td>
<td>boolean variable assignments that make clause true</td>
</tr>
<tr>
<td>Constraints</td>
<td>clauses with shared boolean variables must agree on value of variable</td>
</tr>
</tbody>
</table>
Model-based recognition as CSP

Find given model in edge image, with rotation and translation allowed.

**Variables**  
edges in model

**Domains**  
set of edges in image

**Constraints**  
angle between model & image edges must match
Good News / Bad News

**Good News**
- very general & interesting class problems

**Bad News**
- includes NP-Hard (intractable) problems

So, *good* behavior is a function of domain not the formulation as CSP.
CSP Example

Given 40 courses (8.01, 8.02, . . . . 6.840) & 10 terms (Fall 1, Spring 1, . . . . , Spring 5). Find a legal schedule.
CSP Example

Given 40 courses (8.01, 8.02, . . . . 6.840) & 10 terms (Fall 1, Spring 1, . . . . , Spring 5). Find a legal schedule.

**Constraints**

Pre-requisites

Courses offered on limited terms

Limited number of courses per term

Avoid time conflicts
Given 40 courses (8.01, 8.02, . . . 6.840) & 10 terms (Fall 1, Spring 1, . . . . , Spring 5). Find a legal schedule.

**Constraints**

- **Pre-requisites**
- Courses offered on limited terms
- Limited number of courses per term
- Avoid time conflicts

Note, **CSPs** are not for expressing (soft) preferences e.g., minimize difficulty, balance subject areas, etc.
### Choice of variables & values

<table>
<thead>
<tr>
<th>VARIABLES</th>
<th>DOMAINS</th>
</tr>
</thead>
<tbody>
<tr>
<td>A. Terms?</td>
<td>Legal combinations of for example 4 courses (but this is huge set of values).</td>
</tr>
</tbody>
</table>
Choice of variables & values

<table>
<thead>
<tr>
<th>VARIABLES</th>
<th>DOMAINS</th>
</tr>
</thead>
<tbody>
<tr>
<td>A. Terms?</td>
<td>Legal combinations of for example 4 courses (but this is huge set of values).</td>
</tr>
<tr>
<td>B. Term Slots?</td>
<td>Courses offered during that term</td>
</tr>
</tbody>
</table>

A. Terms?

B. Term Slots?

subdivide terms into slots e.g. 4 of them
(Fall 1,1) (Fall 1,2) (Fall 1,3) (Fall 1,4)
# Choice of variables & values

<table>
<thead>
<tr>
<th>VARIABLES</th>
<th>DOMAINS</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>A. Terms?</strong></td>
<td>Legal combinations of for example 4 courses (but this is huge set of values).</td>
</tr>
<tr>
<td><strong>B. Term Slots?</strong></td>
<td>Courses offered during that term</td>
</tr>
<tr>
<td>subdivide terms into</td>
<td></td>
</tr>
<tr>
<td>slots e.g. 4 of them</td>
<td></td>
</tr>
<tr>
<td>(Fall 1,1) (Fall 1,2)</td>
<td></td>
</tr>
<tr>
<td>(Fall 1,3) (Fall 1,4)</td>
<td></td>
</tr>
<tr>
<td><strong>C. Courses?</strong></td>
<td>Terms or term slots (Term slots allow expressing constraint on limited number of courses / term.)</td>
</tr>
</tbody>
</table>
Constraints

Use courses as variables and term slots as values.

Prerequisite ➔

For pairs of courses that must be ordered.

6.01 ➔ 6.034

term before

term after
Constraints

Use courses as variables and term slots as values.

Prerequisite

6.01 before 6.034

For pairs of courses that must be ordered.

Courses offered only in some terms

Filter domain
Constraints

Use courses as variables and term slots as values.

Prerequisite ➔ 6.01  ➔ 6.034 ➔ For pairs of courses that must be ordered.

Courses offered only in some terms ➔ Filter domain

Limit # courses ➔ slot not equal for all pairs of vars. ➔ Use term-slots only once
Constraints

Use courses as variables and term slots as values.

Prerequisite ➞ 6.01 ➔ 6.034 ➞ For pairs of courses that must be ordered.

Courses offered only in some terms ➞ Filter domain

Limit # courses ➞ for all pairs of vars.

Avoid time conflicts ➞ term not equal

Filter domain

Use term-slots only once

For pairs offered at same or overlapping times
Solving CSPs

Solving CSPs involves some combination of:

1. Constraint propagation, to eliminate values that could not be part of any solution
2. Search, to explore valid assignments
Constraint Propagation (aka Arc Consistency)

Arc consistency eliminates values from domain of variable that can never be part of a consistent solution.

\[ V_i \rightarrow V_j \]

Directed arc \((V_i, V_j)\) is arc consistent if \(\forall x \in D_i \exists y \in D_j\) such that \((x, y)\) is allowed by the constraint on the arc.
**Constraint Propagation (aka Arc Consistency)**

Arc consistency eliminates values from domain of variable that can never be part of a consistent solution.

\[ V_i \rightarrow V_j \]

Directed arc \((V_i, V_j)\) is arc consistent if
\[
\forall x \in D_i \ \exists y \in D_j \text{ such that } (x, y) \text{ is allowed by the constraint on the arc}
\]

We can achieve consistency on arc by deleting values from \(D_i\) (domain of variable at tail of constraint arc) that fail this condition.
Constraint Propagation (aka Arc Consistency)

Arc consistency eliminates values from domain of variable that can never be part of a consistent solution.

\[ V_i \rightarrow V_j \]

Directed arc \((V_i, V_j)\) is arc consistent if
\[ \forall x \in D_i \exists y \in D_j \text{ such that } (x,y) \text{ is allowed by the constraint on the arc} \]

We can achieve consistency on arc by deleting values form \(D_i\) (domain of variable at tail of constraint arc) that fail this condition.

Assume domains are size at most \(d\) and there are \(e\) binary constraints.

A simple algorithm for arc consistency is \(O(ed^3)\) – note that just verifying arc consistency takes \(O(d^2)\) for each arc.
Constraint Propagation Example

Graph Coloring
Initial Domains are indicated
Constraint Propagation Example

Graph Coloring
Initial Domains are indicated

<table>
<thead>
<tr>
<th>Arc examined</th>
<th>Value deleted</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Each undirected constraint arc is really two directed constraint arcs, the effects shown above are from examining BOTH arcs.


**Constraint Propagation Example**

**Graph Coloring**
Initial Domains are indicated

<table>
<thead>
<tr>
<th>Arc examined</th>
<th>Value deleted</th>
</tr>
</thead>
<tbody>
<tr>
<td>$V_1 - V_2$</td>
<td>none</td>
</tr>
</tbody>
</table>
Constraint Propagation Example

Graph Coloring
Initial Domains are indicated

<table>
<thead>
<tr>
<th>Arc examined</th>
<th>Value deleted</th>
</tr>
</thead>
<tbody>
<tr>
<td>$V_1 - V_2$</td>
<td>none</td>
</tr>
<tr>
<td>$V_1 - V_3$</td>
<td>$V_1(G)$</td>
</tr>
</tbody>
</table>
Constraint Propagation Example

Graph Coloring
Initial Domains are indicated

<table>
<thead>
<tr>
<th>Arc examined</th>
<th>Value deleted</th>
</tr>
</thead>
<tbody>
<tr>
<td>$V_1 - V_2$</td>
<td>none</td>
</tr>
<tr>
<td>$V_1 - V_3$</td>
<td>$V_1(G)$</td>
</tr>
<tr>
<td>$V_2 - V_3$</td>
<td>$V_2(G)$</td>
</tr>
</tbody>
</table>
Constraint Propagation Example

Graph Coloring
Initial Domains are indicated

<table>
<thead>
<tr>
<th>Arc examined</th>
<th>Value deleted</th>
</tr>
</thead>
<tbody>
<tr>
<td>$V_1 - V_2$</td>
<td>none</td>
</tr>
<tr>
<td>$V_1 - V_3$</td>
<td>$V_1(G)$</td>
</tr>
<tr>
<td>$V_2 - V_3$</td>
<td>$V_2(G)$</td>
</tr>
<tr>
<td>$V_1 - V_2$</td>
<td>$V_1(R)$</td>
</tr>
</tbody>
</table>
Constraint Propagation Example

Graph Coloring
Initial Domains are indicated

<table>
<thead>
<tr>
<th>Arc examined</th>
<th>Value deleted</th>
</tr>
</thead>
<tbody>
<tr>
<td>$V_1 - V_2$</td>
<td>none</td>
</tr>
<tr>
<td>$V_1 - V_3$</td>
<td>$V_1(G)$</td>
</tr>
<tr>
<td>$V_2 - V_3$</td>
<td>$V_2(G)$</td>
</tr>
<tr>
<td>$V_1 - V_2$</td>
<td>$V_1(R)$</td>
</tr>
<tr>
<td>$V_1 - V_3$</td>
<td>none</td>
</tr>
</tbody>
</table>
Constraint Propagation Example

Graph Coloring
Initial Domains are indicated

<table>
<thead>
<tr>
<th>Arc examined</th>
<th>Value deleted</th>
</tr>
</thead>
<tbody>
<tr>
<td>$V_1 - V_2$</td>
<td>none</td>
</tr>
<tr>
<td>$V_1 - V_3$</td>
<td>$V_1(G)$</td>
</tr>
<tr>
<td>$V_2 - V_3$</td>
<td>$V_2(G)$</td>
</tr>
<tr>
<td>$V_1 - V_2$</td>
<td>$V_1(R)$</td>
</tr>
<tr>
<td>$V_1 - V_3$</td>
<td>none</td>
</tr>
<tr>
<td>$V_2 - V_3$</td>
<td>none</td>
</tr>
</tbody>
</table>
But, arc consistency is not enough in general

Graph Coloring

arc consistent but no solutions
Arc consistency algorithm

**function AC-3( csp ) returns** the CSP, possibly with reduced domains

**inputs:** csp, a binary CSP with variables \{X_1, X_2, \ldots, X_n\}

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

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

\((X_i, X_j) \leftarrow \text{Remove-First}(queue)\)

**if** \text{Remove-Inconsistent-Values}(X_i, X_j) **then**

**for each** \(X_k\) in \text{Neighbors}[X_i] **do**

add \((X_k, X_i)\) to queue

---

**function** \text{Remove-Inconsistent-Values}(X_i, X_j) **returns** true iff succeeds

\(removed \leftarrow false\)

**for each** \(x\) in \text{Domain}[X_i] **do**

**if** no value \(y\) in \text{Domain}[X_j] allows \((x, y)\) to satisfy the constraint \(X_i \leftrightarrow X_j\)

**then** delete \(x\) from \text{Domain}[X_i]; \(removed \leftarrow true\)

**return** removed

\(O(n^2d^3)\), can be reduced to \(O(n^2d^2)\) (but detecting all is NP-hard)
But, arc consistency is not enough in general.

Graph Coloring

Arc consistent but **no** solutions

Arc consistent but **2** solutions **B,R,G ; B,G,R**.
But, arc consistency is not enough in general

Graph Coloring

Arc consistent but **no** solutions

Arc consistent but **2** solutions $B, R, G$; $B, G, R$.

Arc consistent but **1** solution

Assume $B, R$ not allowed
But, arc consistency is not enough in general

Graph Coloring

- Arc consistent but **no** solutions
- Arc consistent but **2** solutions: B,R,G; B,G,R
- Arc consistent but **1** solution
- B, R not allowed

Need to do search to find solutions (if any)
Searching for solutions – backtracking (BT)

When we have too many values in domain (and/or constraints are weak) arc consistency doesn’t do much, so we need to search. Simplest approach is pure backtracking (depth-first search).
Searching for solutions – backtracking (BT)

When we have too many values in domain (and/or constraints are weak) arc consistency doesn’t do much, so we need to search. Simplest approach is pure backtracking (depth-first search).

V₁ assignments

V₂ assignments

V₃ assignments

Inconsistent with \( V₁ = \text{R} \)

Backup at inconsistent assignment
Searching for solutions – backtracking (BT)

When we have too many values in domain (and/or constraints are weak) arc consistency doesn’t do much, so we need to search. Simplest approach is pure backtracking (depth-first search).

![Diagram showing backtracking process]

Inconsistent with $V_1 = R$

Backup at inconsistent assignment
Searching for solutions – backtracking (BT)

When we have too many values in domain (and/or constraints are weak) arc consistency doesn’t do much, so we need to search. Simplest approach is pure backtracking (depth-first search).

V₁ assignments

V₂ assignments

V₃ assignments

Inconsistent with \( V₁ = R \)

Inconsistent with \( V₂ = G \)

Backup at inconsistent assignment

Inconsistent with \( V₂ = G \)
Searching for solutions – backtracking (BT)

When we have too many values in domain (and/or constraints are weak) arc consistency doesn’t do much, so we need to search. Simplest approach is pure backtracking (depth-first search).

V₁ assignments

V₂ assignments

V₃ assignments

Inconsistent with V₁ = R

Inconsistent with V₂ = G

Consistent

Backup at inconsistent assignment
Combine Backtracking & Constraint Propagation

A node in BT tree is **partial** assignment in which the domain of each variable has been set (tentatively) to singleton set.

Use constraint propagation (arc-consistency) to propagate the effect of this tentative assignment, i.e., eliminate values inconsistent with current values.
A node in BT tree is **partial** assignment in which the domain of each variable has been set (tentatively) to singleton set.

Use constraint propagation (arc-consistency) to propagate the effect of this tentative assignment, i.e., eliminate values inconsistent with current values.

**Question:** How much propagation to do?
Combine Backtracking & Constraint Propagation

A node in BT tree is partial assignment in which the domain of each variable has been set (tentatively) to singleton set.

Use constraint propagation (arc-consistency) to propagate the effect of this tentative assignment, i.e., eliminate values inconsistent with current values.

**Question:** How much propagation to do?

**Answer:** Not much, just local propagation from domains with unique assignments, which is called forward checking (FC). This conclusion is not necessarily obvious, but it generally holds in practice.
Backtracking with Forward Checking (BT-FC)

When examining assignment $V_i = d_k$, remove any values inconsistent with that assignment from neighboring domains in constraint graph.

![Diagram of constraint graph showing $V_1$, $V_2$, and $V_3$ assignments with inconsistencies indicated.](image-url)
Backtracking with Forward Checking (BT-FC)

When examining assignment $V_i = d_k$, remove any values inconsistent with that assignment from neighboring domains in constraint graph.
Backtracking with Forward Checking (BT-FC)

When examining assignment $V_i=d_k$, remove any values inconsistent with that assignment from neighboring domains in constraint graph.

We have a conflict whenever a domain becomes empty.
**Backtracking with Forward Checking (BT-FC)**

When examining assignment $V_i = d_k$, remove any values inconsistent with that assignment from neighboring domains in constraint graph.

When backing up, need to restore domain values, since deletions were done to reach consistency with tentative assignments considered during search.
Backtracking with Forward Checking (BT-FC)

When examining assignment $V_i = d_k$, remove any values inconsistent with that assignment from neighboring domains in constraint graph.
Backtracking with Forward Checking (BT-FC)

When examining assignment $V_i = d_k$, remove any values inconsistent with that assignment from neighboring domains in constraint graph.
Backtracking with Forward Checking (BT-FC)

When examining assignment $V_i = d_k$, remove any values inconsistent with that assignment from neighboring domains in constraint graph.
Backtracking with Forward Checking (BT-FC)

When examining assignment $V_i=d_k$, remove any values inconsistent with that assignment from neighboring domains in constraint graph.

- $V_1$ assignments
- $V_2$ assignments
- $V_3$ assignments
Backtracking with Forward Checking (BT-FC)

When examining assignment $V_i = d_k$, remove any values inconsistent with that assignment from neighboring domains in constraint graph.
Backtracking with Forward Checking (BT-FC)

When examining assignment $V_i = d_k$, remove any values inconsistent with that assignment from neighboring domains in constraint graph.
Backtracking with Forward Checking (BT-FC)

When examining assignment $V_i = d_k$, remove any values inconsistent with that assignment from neighboring domains in constraint graph.
Backtracking with Forward Checking (BT-FC)

When examining assignment $V_i = d_k$, remove any values inconsistent with that assignment from neighboring domains in constraint graph.
Backtracking with Forward Checking (BT-FC)

When examining assignment $V_i = d_k$, remove any values inconsistent with that assignment from neighboring domains in constraint graph.

No need to check previous assignments

Generally preferable to pure BT
BT-FC with dynamic ordering

Traditional backtracking uses fixed ordering of variables & values, e.g. random order or place variables with many constraints first.

You can usually do better by choosing an order dynamically as the search proceeds.
BT-FC with dynamic ordering

Traditional backtracking uses fixed ordering of variables & values, e.g., random order or place variables with many constraints first.

You can usually do better by choosing an order dynamically as the search proceeds.

• Most constrained variable
  when doing forward-checking, pick variable with fewest legal values to assign next (minimizes branching factor)
BT-FC with dynamic ordering

Traditional backtracking uses fixed ordering of variables & values, e.g., random order or place variables with many constraints first.

You can usually do better by choosing an order dynamically as the search proceeds.

• **Most constrained variable**
  when doing forward-checking, pick variable with fewest legal values to assign next (minimizes branching factor)

• **Least constraining value**
  choose value that rules out the fewest values from neighboring domains
BT-FC with dynamic ordering

Traditional backtracking uses fixed ordering of variables & values, e.g., random order or place variables with many constraints first.

You can usually do better by choosing an order dynamically as the search proceeds.

• **Most constrained variable**
  when doing forward-checking, pick variable with fewest legal values to assign next (minimizes branching factor)

• **Least constraining value**
  choose value that rules out the fewest values from neighboring domains

E.g. this combination improves feasible $n$-queens performance from about $n = 30$ with just FC to about $n = 1000$ with FC & ordering.
Which country should we color next

What color should we pick for it?

Colors: R, G, B, Y

A = Green
B = Blue
C = Red

A
B
C
D

Red, Yellow
Red, Blue, Yellow
Green, Blue, Yellow

E
F
Which country should we color next?

What color should we pick for it?

E most-constrained variable (smallest domain)

Colors: R, G, B, Y

A = Green
B = Blue
C = Red
Which country should we color next

What color should we pick for it?

E most-constrained variable
(smallest domain)

RED least-constraining value
(eliminates fewest values from neighboring domains)

Colors: R, G, B, Y

A=Green
B=Blue
C=Red
Problem structure

Tasmania and mainland are independent subproblems

Identifiable as connected components of constraint graph
Problem structure contd.

Suppose each subproblem has $c$ variables out of $n$ total

Worst-case solution cost is $n/c \cdot d^c$, linear in $n$

E.g., $n = 80$, $d = 2$, $c = 20$

$2^{80} = 4$ billion years at 10 million nodes/sec

$4 \cdot 2^{20} = 0.4$ seconds at 10 million nodes/sec
Tree-structured CSPs

Theorem: if the constraint graph has no loops, the CSP can be solved in $O(n d^2)$ time

Compare to general CSPs, where worst-case time is $O(d^n)$

This property also applies to logical and probabilistic reasoning: an important example of the relation between syntactic restrictions and the complexity of reasoning.
Algorithm for tree-structured CSPs

1. Choose a variable as root, order variables from root to leaves such that every node’s parent precedes it in the ordering

2. For $j$ from $n$ down to 2, apply $\text{REMOVEINCONSISTENT}(\text{Parent}(X_j), X_j)$

3. For $j$ from 1 to $n$, assign $X_j$ consistently with $\text{Parent}(X_j)$
Nearly tree-structured CSPs

**Conditioning:** instantiate a variable, prune its neighbors’ domains

![Diagram showing conditioning](image)

**Cutset conditioning:** instantiate (in all ways) a set of variables such that the remaining constraint graph is a tree

Cutset size $c \Rightarrow$ runtime $O(d^c \cdot (n - c)d^2)$, very fast for small $c$
Incremental Repair (min-conflict heuristic)

1. Initialize a candidate solution using “greedy” heuristic – get solution “near” correct one.

2. Select a variable in conflict and assign it a value that minimizes the number of conflicts (break ties randomly).

Can use this heuristic as part of systematic backtracker that uses heuristics to do value ordering or in a local hill-climber (without backup).

Performance on n-queens. (with good initial guesses)
The pure hill climber (without backtracking) can get stuck in local minima. Can add random moves to attempt getting out of minima – generally quite effective. Can also use weights on violated constraints & increase weight every cycle it remains violated.

**GSAT**

Randomized hill climber used to solve SAT problems. One of the most effective methods ever found for this problem
GSAT as Heuristic Search

• State space: Space of all full assignments to variables
• Initial state: A random full assignment
• Goal state: A satisfying assignment
• Actions: Flip value of one variable in current assignment
• Heuristic: The number of satisfied clauses (constraints); we want to maximize this. Alternatively, minimize the number of unsatisfied clauses (constraints).
GSAT(F)

- For i=1 to Maxtries
  - Select a complete random assignment A
  - Score = number of satisfied clauses
  - For j=1 to Maxflips
    - If (A satisfies all clauses in F) return A
    - Else flip a variable that maximizes score
    - Flip a randomly chosen variable if no variable flip increases the score.
**WALKSAT(F)**

- For i=1 to Maxtries
  - Select a complete random assignment A
  - Score = number of satisfied clauses
- For j=1 to Maxflips
  - If (A satisfies all clauses in F) return A
  - Else
    - With probability p /* GSAT */
      » flip a variable that maximizes score
      » Flip a randomly chosen variable if no variable flip increases the score.
    - With probability 1-p /* Random Walk */
      » Pick a random unsatisfied clause C
      » Flip a randomly chosen variable in C
CSPs are a special kind of problem:
states defined by values of a fixed set of variables
goal test defined by constraints on variable values

Backtracking = depth-first search with one variable assigned per node

Variable ordering and value selection heuristics help significantly

Forward checking prevents assignments that guarantee later failure

Constraint propagation (e.g., arc consistency) does additional work
to constrain values and detect inconsistencies

The CSP representation allows analysis of problem structure

Tree-structured CSPs can be solved in linear time

Iterative min-conflicts is usually effective in practice