CMU 15-781
Lecture 3: Constraint Satisfaction Problems (CSPs)

Teacher: Gianni A. Di Caro
Overview

• Definitions, toy and real-world examples
• Basic algorithms for solving CSPs
• Pruning space through propagating information
 CONSTRAINT SATISFACTION PROBLEMS (CSP)

• Set of decision Variables: \( V = \{ V_1, \ldots, V_N \} \)
• Domains: Sets of \( D_i \) possible values for each variable \( V_i \)
• Set of Constraints: \( C = \{ C_1, \ldots, C_K \} \) restricting the values the variables can simultaneously take
• A constraint consists of:
  o variable tuple
  o list of possible values for tuple (ex. \( [(V_2, V_3), \{(R, B), (R, G)\}] \))
  o Or functional relation (ex. \( V_2 \neq V_3, V_1 > V_4 + 5 \))
• Allows useful general-purpose algorithms with more power than standard search algorithms
**Example: N-Queens**

- **Variables:**
  - \( Q_i \) position of queen in column \( i \)

- **Domains:**
  - \{1, ..., 8\}

- **Constraints:**
  - No queen attack each other
  - \( Q_i = k \Rightarrow Q_j \neq k, \ \forall \ j = 1,..8, \ j \neq i \)
  - Similar constraints for diagonals

**Alternative formulation?**
**Example: Map Coloring**

Given $n$ different colors, color a map so that adjacent areas are different colors.

![Map Coloring Diagram]

- Western Australia
- Northern Territory
- South Australia
- Queensland
- New South Wales
- Victoria
- Tasmania
Map Coloring: Match!

Constraints

\{\text{red, green, blue}\}

Variables

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

Domain

\{(WA, NT) \in \{(\text{red, green}), (\text{red, blue}), (\text{green, red}), \ldots\}\}

Solutions

WA, NT, Q, NSW, V, T, SA
Example: Sudoku

- Variables:
  - $X_{ij}$, each open square

- Domain:
  - \{1:9\}

- Constraints:
  - 9-way all diff col
  - 9-way all diff row
  - 9-way all diff box
Scheduling (Important Example)

• Many industries. Many multi-million $ decisions. Used extensively for space mission planning. Military uses.

• People really care about improving scheduling algorithms! Problems with phenomenally huge state spaces. But for which solutions are needed very quickly.

• Many kinds of scheduling problems e.g.:
  o *Job shop*: Discrete time; weird ordering of operations possible; set of separate jobs.
  o *Batch shop*: Discrete or continuous time; restricted operation of ordering; grouping is important.
**JOB SCHEDULING**

- A set of $J$ jobs, $J_1, \ldots, J_n$
- A set of $R$ resources, $R_1, R_2, \ldots, R_m$ to do the jobs
- Each job $j$ is a sequence of operations $O_{j1}, \ldots, O_{jL_j}$ to be scheduled according to process plans: $O_{j1} < O_{j2} < O_{j3} \ldots$
- Each operation has a fixed processing time and requires the use of resources $R_i$, a resource can have capacity constraints
- Each job has a *ready time* and a *due time*
- A resource can only be used by a single operation at a time.
- All jobs must be completed by a due time.

- Problem: assign a start time to each job such that all jobs are completed by their due times respecting all constraints
**Job Scheduling**

**Project job scheduling**

For each job, choose an execution mode and a start time.

1. **Design**
2. **Cover**
3. **Pages (400/book)**
4. **Assembly**

**Book 1**

1. **1 day**

**Book 2**

1. **1 day**

**Resources**

1. **1 day**

**10X10 Job Shop Scheduling Problem**

Constrained Schedule

<table>
<thead>
<tr>
<th>JOB</th>
<th>0</th>
<th>60</th>
<th>120</th>
<th>180</th>
<th>240</th>
<th>300</th>
<th>360</th>
<th>420</th>
<th>480</th>
<th>540</th>
<th>600</th>
<th>660</th>
<th>720</th>
<th>780</th>
<th>840</th>
<th>900</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Machine Required**

- M0
- M1
- M2
- M3
- M4
- M5
- M6
- M7
- M8
- M9

15781 Fall 2016: Lecture 3
CLASS SCHEDULING WOES

• 4 more required classes to graduate
  o A: Algorithms  B: Bayesian Learning
  o C: Computer Programming  D: Distributed Computing

• A few restrictions
  o Algorithms must be taken same semester as Distributed computing
  o Computer programming is a prereq for Distributed computing and Bayesian learning, so it must be taken in an earlier semester
  o Advanced algorithms and Bayesian Learning are always offered at the same time, so they cannot be taken the same semester

• 3 semesters (semester 1,2,3) when can take classes
Exercise: Define CSP

- 4 more required classes to graduate: A, B, C, D
- A must be taken same semester as D
- C is a prereq for D and B so must take C earlier than D & B
- A & B are always offered at the same time, so they cannot be taken the same semester
- 3 semesters (semester 1, 2, 3) when can take classes
**Exercise: Define CSP**

- 4 more required classes to graduate: A, B, C, D
- A must be taken same semester as D
- C is a prereq for D and B so must take C earlier than D & B
- A & B are always offered at the same time, so they cannot be taken the same semester
- 3 semesters (semester 1, 2, 3) when can take classes
- Variables: A, B, C, D
- Domain: {1, 2, 3}
- Constraints: A ≠ B, A=D, C < B, C < D
Types of CSPs

• Discrete-domain variables
  o **Finite domains** (Map coloring, Sudoku, N-queens, SAT) ➔ Our focus!
  o **Infinite domains** (Integers or strings, deadline-free JSS)
    Constraint language is needed to understand relations $J_1 + d_1 \leq J_2$ without enumerating all tuples
    Integer programming methods deal effectively with (integer, binary) problems with linear constraints

• Continuous variables (planning, blending, positioning,...)
  o **Linear/convex programming** for linear/convex constraints
**Types of Constraints**

- **Unary**: involve a single variable
- **Binary**: involve two variables
- **n-ary**: involve $n$ variables
- **Soft constraints**: violation incurs a cost, the problem becomes a constraint optimization one
**Constraint Graph**

- Variables $\rightarrow$ Vertices
- Constraints $\rightarrow$ Edges
  - Unary: Self-edges
  - Binary: regular edges
  - $n$-ary: hyperedges (hypergraphs)

\[
\begin{align*}
A &< 2 \\
A > C \\
B \neq C
\end{align*}
\]
Cryptarithmetic puzzles

\[
\begin{align*}
\text{TWO} & \quad + \quad \text{TWO} \\
\hline
\text{FOUR} & \quad = \\
\end{align*}
\]

\[V = \{O,W,T,R,U,F\}\]

\[D = \{0, \ldots, 9\}\]

\[10^0(O+O) + 10^1(W+W) + 10^2(T+T) = 10^0R + 10^1U + 10^2O + 10^3F\]

\[\{O+O = R+10C_1, C_1+W+W=U+10C_2, C_2+T+T=O+10C_3, C_3 = F\}\]

\[V = \{O,W,T,R,U,F, C_1, C_2, C_3\} \quad \text{Auxiliary vars}\]
Binary constraint graphs

It’s always possible to reduce a hypergraph to a binary constraint graph!

But this is not always the best thing to do ....

If you want to know more ...

OVERVIEW

• Definitions, toy and real-world examples
• Basic algorithms for solving CSPs
• Pruning space through propagating information
WHY NOT JUST DO BASIC SEARCH ALGORITHMS FROM LAST TIME?

• **States:** Partial assignments to the $n$ variables
• **Initial state:** Empty state
• **Action:** Select an unassigned variable $i$ and assign a feasible value from its domain $D_i$ to it
• **Goal test:** Assignment consistent (no violations) and complete (all variables assigned)
• **Step cost:** Constant
• Solution is found at depth $n$, using **depth-limited DFS**
• Size of the search tree?
Why not just do basic search algorithms from last time?

\[ n = 4 \text{ variables each taking } d = 4 \text{ values} \]

\[ b = nd \]

\[ b = (n-1)d \]

Generate a search tree of \( n!d^n \) but there are only \( d^n \) possible assignments!
COMMUTATIVITY!

• The order of assigning the variables has no effect on the final outcome

• CSPs are commutative: Regardless of the assignment order, the same partial solution is reached for a defined set of assignment values

• Don’t care about path!

• ➔ Only a single variable at each node in the search tree needs to be considered!! (can fix the order)

• ➔ $d^n$ number of leaves in the search tree!
Backtracking: DFS with Single Variable Assignments

- Only consider a single variable at each point
- Don’t care about path
- Order of variable assignment doesn’t matter, so fix ordering
- Only consider values which do not conflict with assignment made so far
- Depth-first search for CSPs with these two improvements is called backtracking search
BACKTRACKING

- Function **Backtracking**(csp) returns solution or fail
  - Return Backtrack({},csp)
- Function **Backtrack**(assignment,csp) returns solution or fail
  - If assignment is complete, return assignment
  - $V_i \leftarrow select\_unassigned\_var(csp)$
  - For each val in order-domain-values($var$, csp, assign)
    - If value is consistent with assignment
      - Add [$V_i = val$] to assignment
      - Result $\leftarrow$ Backtrack(assignment, csp)
      - If Result $\neq$ fail, return result
      - Remove [$V_i = val$] from assignments
  - Return fail
**Backtracking**

- Function **Backtracking**(csp) returns soln or fail
  - Return Backtrack({},csp)
- Function **Backtrack**(assignment,csp) returns soln or fail
  - If assignment is complete, return assignment
  - $V_i \leftarrow select\_unassigned\_var(csp)$
  - For each val in order-domain-values(var,csp,assign)
    - If value is consistent with assignment
      - Add [$V_i = \text{val}$] to assignment
      - Result $\leftarrow$ Backtrack(assignment,csp)
      - If Result $\neq$ fail, return result
      - Remove [$V_i = \text{val}$] from assignments
  - Return fail
THINK AND DISCUSS

• Does the variable/value order used affect how long backtracking takes to find a solution?
• Does the variable/value order used affect the solution found by backtracking?
**Example**

**Variables:** A, B, C, D  **Domain:** \{1, 2, 3\}

**Constraints:** A ≠ B, A = D, C < B, C < D

**Variable order:** alphabetical  **Value order:** Descending

- \((A = 3)\)
EXAMPLE

Variables: A, B, C, D
Domain: \{1, 2, 3\}
Constraints: A ≠ B, A = D, C < B, C < D

Variable order: alphabetical
Value order: Descending

• (A=3)
• (A=3, B=3) inconsistent with A ≠ B
• (A=3, B=2)
• (A=3, B=2, C=3) inconsistent with C < B
• (A=3, B=2, C=2) inconsistent with C < B
• (A=3, B=2, C=1)
• (A=3, B=2, C=1, D=3) VALID
EXAMPLE

 VARIABLES: A,B,C,D
 Domain: \{1,2,3\}
 Constraints: A \neq B, A=D, C < B, C < D

 Variable order: alphabetical
 Value order: ascending

• \(A=1\)
Example

Variables: A, B, C, D  Domain: \{1, 2, 3\}

Constraints: A \neq B, A=D, C < B, C < D

Variable order: alphabetical  Value order: ascending

- (A=1)
- (A=1, B=1) inconsistent with A \neq B
- (A=1, B=2)
- (A=1, B=2, C=1)
- (A=1, B=2, C=1, D=1) inconsistent with C < D
- (A=1, B=2, C=1, D=2) inconsistent with A=D
- (A=1, B=2, C=1, D=3) inconsistent with A=D
**Example**

**Variables:** A, B, C, D  **Domain:** \{1, 2, 3\}

**Constraints:** A ≠ B, A = D, C < B, C < D

**Variable order:** alphabetical  **Value order:** ascending

- (A=1, B=1) inconsistent with A ≠ B
- (A=1, B=2)
- (A=1, B=2, C=1)
- (A=1, B=2, C=1, D=1) inconsistent with C < D
- (A=1, B=2, C=1, D=2) inconsistent with A = D
- (A=1, B=2, C=1, D=3) inconsistent with A = D
- No valid assignment for D, return result = fail
  - Backtrack to (A=1, B=2, C=)
- Try (A=1, B=2, C=2) but inconsistent with C < B
- Try (A=1, B=2, C=3) but inconsistent with C < B
- No other assignments for C, return result = fail
  - Backtrack to (A=1, B=)
- (A=1, B=3)
- (A=1, B=3, C=1)
- (A=1, B=3, C=1, D=1) inconsistent with C < D
- (A=1, B=3, C=1, D=2) inconsistent with A = D
- (A=1, B=3, C=1, D=3) inconsistent with A = D
- Return result = fail
  - Backtrack to (A=1, B=3, C=)

- (A=1, B=3, C=2) inconsistent with C < B
- (A=1, B=3, C=3) inconsistent with C < B
- No remaining assignments for C, return fail
  - Backtrack to (A=1, B=)
- No remaining assignments for B, return fail
  - Backtrack to A
- (A=2)
- (A=2, B=1)
- (A=2, B=1, C=1) inconsistent with C < B
- (A=2, B=1, C=2) inconsistent with C < B
- (A=2, B=1, C=3) inconsistent with C < B
- No remaining assignments for C, return fail
  - Backtrack to (A=2, B=?)
- (A=2, B=2) inconsistent with A ≠ B
- (A=2, B=3)
- (A=2, B=3, C=1)
- (A=2, B=3, C=1, D=1) inconsistent with C < D
- (A=2, B=3, C=1, D=2) inconsistent with C < D
- (A=2, B=3, C=1, D=3) ALL VALID
ORDERING MATTERS!

- Function Backtracking(csp) returns soln or fail
  - Return Backtrack({},csp)
- Function Backtrack(assignment,csp) returns soln or fail
  - If assignment is complete, return assignment
  - $V_i \leftarrow \text{select_unassigned_var}(csp)$
  - For each val in order-domain-values(var,csp,assign)
    - If value is consistent with assignment
      - Add $[V_i = \text{val}]$ to assignment
      - Result $\leftarrow \text{Backtrack}(assignment,csp)$
      - If Result $\neq$ fail, return result
      - Remove $[V_i = \text{val}]$ from assignments
  - Return fail

15781 Fall 2016: Lecture 3
ORDERING HEURISTICS

• Next variable?
  o Random or static
  o Variable with the fewest legal values: Minimum remaining values (MRV) heuristic (aka the most constrained var, the fail-first var)
  o Variable with the largest number of constraints on other unassigned variables, reduces b on future choices (Degree heuristic)

• Variable’s value?
  o Value that leaves most choices for the neighboring variables in the constraint graph, max flexibility (least-constraining-value heuristic), fail-last
(Test) Cost of Backtracking?

- d values per variable
- n variables
- Possible number of CSP assignments?

- A) $O(d^n)$
- B) $O(n^d)$
- C) $O(nd)$
OVERVIEW

- Real world CSPs
- Basic algorithms for solving CSPs
- Pruning space through propagating information
LIMITATIONS OF BACKTRACKING

• Can inevitable failure be detected earlier?
• Can problem structure can be exploited?
• Can the search space be reduced to speed up computation?
If we choose a value for one variable, that affects its neighbors. And then potentially those neighbors... We can use this inference to prune the search space.
**Arc Consistency**

- **Definition:**
  - An “arc” (connection between two variables $X \rightarrow Y$ in constraint graph) is **consistent** if:
  - For every value could assign to $X$ there exists some value of $Y$ that could be assigned without violating a constraint.

If a variable is not arc consistent with another one, it can be made so by removing some values from its domain. This can be done recursively → **Form of constraint propagation** that enforces arc consistency, maintains the problem solutions, and prunes the tree!
Arc Consistency in Practice

Example from Kevin Leyton-Brown

- $\text{dom}(A) = \{1, 2, 3, 4\}$; $\text{dom}(B) = \{1, 2, 3, 4\}$; $\text{dom}(C) = \{1, 2, 3, 4\}$
- Suppose you first select the arc $\langle A, A < B \rangle$.
  - Remove $A = 4$ from the domain of $A$.
  - Add nothing to $TDA$. (To-Do-Arcs)
- Suppose that $\langle B, B < C \rangle$ is selected next.
  - Prune the value 4 from the domain of $B$.
  - Add $\langle A, A < B \rangle$ back into the $TDA$ set (why?)
- Suppose that $\langle B, A < B \rangle$ is selected next.
  - Prune 1 from the domain of $B$.
  - Add no element to $TDA$ (why?)
- Suppose the arc $\langle A, A < B \rangle$ is selected next
  - The value $A = 3$ can be pruned from the domain of $A$.
  - Add no element to $TDA$ (why?)
- Select $\langle C, B < C \rangle$ next.
  - Remove 1 and 2 from the domain of $C$.
  - Add $\langle B, B < C \rangle$ back into the $TDA$ set

The other two edges are arc consistent, so the algorithm terminates with $\text{dom}(A) = \{1, 2\}$, $\text{dom}(B) = \{2, 3\}$, $\text{dom}(C) = \{3, 4\}$. 

\[ \square \]
AC-3 Computational Complexity?

- Input: CSP
- Output: CSP, possible with reduced domains for variables, or inconsistent
- Local variables: queue, initially queue of all arcs (binary constraints in csp)
- While queue is not empty
  - \((X_i,X_j) = \text{Remove-First(queue)}\)
  - \([\text{domain}X_i, \text{anyChangeToDomain}X_i] = \text{Revise(csp,X_i,X_j)}\)
  - if anyChangeToDomainX_i == true
    - if size(domainX_i) = 0, return inconsistent
    - else
      - for each \(X_k\) in Neighbors(X_i) except \(X_j\)
        - add \((X_k,X_i)\) to queue
  - Return csp

Function \(\text{Revise(csp,X_i,X_j)}\) returns DomainXi and anyChangeToDomainX_i
- anyChangeToDomainX_i = false
- for each \(x\) in Domain(X_i)
  - if no value \(y\) in Domain(X_j) allows \((x,y)\) to satisfy constraint between \((X_i,X_j)\)
    - delete \(x\) from Domain(X_i)
    - anyChangeToDomainX_i = true

Have to add in arc for \((X_i,X_j)\) and \((X_j,X_i)\) for \(i,j\) constraint

\(D\) domain values
\(C\) binary constraints

Complexity of revise function? \(D^2\)

Number of times can put a constraint in queue? \(D\)

Total: \(CD^3\)
(Test) Sufficient?

• After we run AC-3 have we always found a solution? (aka only 1 value left for each variable)

• A) Yes
• B) No
AC-3 Example

- Variables: A, B, C, D
- Domain: \{1, 2, 3\}
- Constraints: A ≠ B, C < B, C < D (subset of constraints from before)
AC-3 Example

- Variables: A, B, C, D
- Domain: \{1, 2, 3\}
- Constraints: A \neq B, C < B, C < D (subset of constraints from before)
- Constraints both ways: A \neq B, B \neq A, C < B, B > C, C < D, D > C
AC-3 Example

- Variables: A, B, C, D
- Domain: \{1,2,3\}
- Constraints: A \neq B, C < B, C < D (subset of constraints from before)
- Constraints both ways: A \neq B, B \neq A, C < B, B > C, C < D, D > C
- Queue: AB, BA, BC, CB, CD, DC
AC-3 EXAMPLE

- Variables: A, B, C, D
- Domain: \{1, 2, 3\}
- Constraints: A \neq B, C < B, C < D (subset of constraints from before)
- Constraints both ways: A \neq B, B \neq A, C < B, B > C, C < D, D > C
- Queue: AB, BA, BC, CB, CD, DC
- Pop AB:
  - for each x in Domain(A)
    - if no value y in Domain(B) that allows (x, y) to satisfy constraint between (A, B)
      - delete x from Domain(A)
  - No change to domain of A
AC-3 Example

- Variables: A, B, C, D
- Domain: \{1, 2, 3\}
- Constraints: A \neq B, C < B, C < D (subset of constraints from before)
- Constraints both ways: A \neq B, B \neq A, C < B, B > C, C < D, D > C
- Queue: AB, BA, BC, CB, CD, DC
- Pop AB
- Queue: BA, BC, CB, CD, DC
AC-3 Example

- Variables: A, B, C, D
- Domain: \{1, 2, 3\}
- Constraints: A ≠ B, C < B, C < D (subset of constraints from before)
- Constraints both ways: A ≠ B, B ≠ A, C < B, B > C, C < D, D > C
- Queue: AB, BA, BC, CB, CD, DC
- Pop AB
- Queue: BA, BC, CB, CD, DC
- Pop BA
- for each x in Domain(B)
  if no value y in Domain(A) that allows (x, y) to satisfy constraint between (B, A)
  delete x from Domain(B)
- No change to domain of B
AC-3 EXAMPLE

• Variables: A, B, C, D
• Domain: \{1, 2, 3\}
• Constraints: A \neq B, C < B, C < D (subset of constraints from before)
• Constraints both ways: A \neq B, B \neq A, C < B, B > C, C < D, D > C
• Queue: AB, BA, BC, CB, CD, DC
• Queue: BA, BC, CB, CD, DC
• Queue: BC, CB, CD, DC
• Pop BC
• for each x in Domain(B)
  • if no value y in Domain(C) that allows (x, y) to satisfy constraint between (B, C)
  • delete x from Domain(B)
• If B is 1, constraint B > C cannot be satisfied. So delete 1 from B’s domain, B = \{2, 3\}
• **Also have to add neighbors of B (except C) back to queue: AB**
• Queue: AB, CB, CD, DC
AC-3 EXAMPLE

Variables: A,B,C,D
Domain: \{1,2,3\}
Constraints: A \neq B, C < B, C < D

- Queue: AB, BA, BC, CB, CD, DC  \quad A-D = \{1,2,3\}
- Queue: BA, BC, CB, CD, DC  \quad A-D = \{1,2,3\}
- Queue: BC, CB, CD, DC  \quad A-D = \{1,2,3\}
- Queue: AB, CB, CD, DC  \quad B=\{2,3\}, A/C/D = \{1,2,3\}
- Pop AB
  - For every value of A is there a value of B such that A \neq B?
  - Yes, so no change
AC-3 Example

- Queue: AB, BA, BC, CB, CD, DC, A-D = {1,2,3}
- Queue: BA, BC, CB, CD, DC, A-D = {1,2,3}
- Queue: BC, CB, CD, DC, A-D = {1,2,3}
- Queue: AB, CB, CD, DC, B={2,3}, A/C/D = {1,2,3}
- Queue: CB, CD, DC, B={2,3}, A/C/D = {1,2,3}
- Pop CB
  - For every value of C is there a value of B such that C < B
  - If C = 3, no value of B that fits
  - So delete 3 from C’s domain, C = {1,2}
  - Also have to add neighbors of C (except B) back to queue: no change because already in

Variables: A,B,C,D
Domain: {1,2,3}
Constraints: A ≠ B, C < B, C < D
AC-3 EXAMPLE

Variables: A,B,C,D
Domain: \{1,2,3\}
Constraints: A ≠ B, C < B, C < D

- Queue: AB, BA, BC, CB, CD, DC, A-D = \{1,2,3\}
- Queue: BA, BC, CB, CD, DC, A-D = \{1,2,3\}
- Queue: BC, CB, CD, DC, A-D = \{1,2,3\}
- Queue: AB, CB, CD, DC, B={2,3}, A/C/D = \{1,2,3\}
- Queue: CB, CD, DC, B={2,3}, A/C/D = \{1,2,3\}
- Queue: CD, DC, B={2,3}, C = \{1,2\} A,D = \{1,2,3\}
- Pop CD
  - For every value of C, is there a value of D such that C < D?
  - Yes, so no change
AC-3 EXAMPLE

Variables: A, B, C, D
Domain: \{1, 2, 3\}
Constraints: A \neq B, C < B, C < D

- Queue: AB, BA, BC, CB, CD, DC  \quad A-D = \{1, 2, 3\}
- Queue: BA, BC, CB, CD, DC  \quad A-D = \{1, 2, 3\}
- Queue: BC, CB, CD, DC  \quad A-D = \{1, 2, 3\}
- Queue: AB, CB, CD, DC  \quad B=\{2, 3\}, A/C/D = \{1, 2, 3\}
- Queue: CB, CD, DC  \quad B=\{2, 3\}, A/C/D = \{1, 2, 3\}
- Queue: CD, DC  \quad B=\{2, 3\}, C = \{1, 2\} A,D = \{1, 2, 3\}
- Queue: DC  \quad B=\{2, 3\}, C = \{1, 2\} A,D = \{1, 2, 3\}
- For every value of D is there a value of C such that D > C?
  - Not if D = 1
  - So D = \{2, 3\}
**AC-3 EXAMPLE**

Variables: A, B, C, D  
Domain: \{1,2,3\}  
Constraints: A ≠ B, C < B, C < D

- Queue: AB, BA, BC, CB, CD, DC  \hspace{1cm} A-D = \{1,2,3\}
- Queue: BA, BC, CB, CD, DC  \hspace{1cm} A-D = \{1,2,3\}
- Queue: BC, CB, CD, DC  \hspace{1cm} A-D = \{1,2,3\}
- Queue: AB, CB, CD, DC  \hspace{1cm} B={2,3}, A/C/D = \{1,2,3\}
- Queue: CB, CD, DC  \hspace{1cm} B={2,3}, A/C/D = \{1,2,3\}
- Queue: CD, DC  \hspace{1cm} B={2,3}, C = \{1,2\} A,D = \{1,2,3\}
- Queue: DC  \hspace{1cm} B={2,3}, C = \{1,2\} A,D = \{1,2,3\}
- A = \{1,2,3\}  \hspace{1cm} B={2,3}, C = \{1,2\} D = \{2,3\}
**Forward Checking**

- AC-3 can run *before* the search begins to prune search tree; it operates on the entire search tree (expensive!)
- It’s a form of inference (inferring reductions)
- What if, instead, we make inference at run-time?
- **Forward checking:** When assign a variable, make all of its neighbors arc-consistent (*purely local*)
Backtracking + Forward Checking

- Function $\text{Backtrack}(\text{assignment},\text{csp})$ returns soln or fail
  - If assignment is complete, return assignment
  - $V_i \leftarrow \text{select\_unassigned\_var}(\text{csp})$
  - For each val in $\text{order\_domain\_values}(\text{var},\text{csp},\text{assign})$
    - If value is consistent with assignment
      - Add $[V_i = \text{val}]$ to assignment
      - Make domains of all neighbors of $V_i$ arc-consistent with $[V_i = \text{val}]$
      - Result $\leftarrow \text{Backtrack}(\text{assignment},\text{csp})$
      - If Result $\neq$ fail, return result
        - Remove $[V_i = \text{val}]$ from assignments
    - Return fail

- Note: When backtracking, domains must be restored
**Maintaining Arc Consistency**

- Forward checking doesn’t ensure all arcs are consistent, only the local ones, no look-ahead
- AC-3 can detect failure faster than forward checking
- The MAC algorithm includes AC-3 in the search, executing it from the arcs of the locally unassigned variables
- What’s the downside? **Computation**
**Maintaining Arc Consistency (MAC)**

- Function `Backtrack(assignment,csp)` returns soln or fail
  - If assignment is complete, return assignment
  - \( V_i \leftarrow \text{select\_unassigned\_var}(csp) \)
  - For each \( \text{val} \) in \( \text{order\_domain\_values}(\text{var},csp,\text{assign}) \)
    - If value is consistent with assignment
      - Add \([V_i = \text{val}]\) to assignment
      - Run AC-3 to make all variables arc-consistent with \([V_i = \text{val}]\).
        - Initial queue is arcs \((X_j,V_i)\) of neighbors of \(V_i\) that are unassigned, but add other arcs if these vars change domains.
      - \( \text{Result} \leftarrow \text{Backtrack}(\text{assignment},csp) \)
  - If Result \(\neq\) fail, return result
    - Remove \([V_i = \text{val}]\) from assignments
  - Return fail
(Test) Sufficient to Avoid Backtracking?

- If we maintain arc consistency, we will never have to backtrack while solving a CSP

  - A) True
  - B) False
AC LIMITATIONS

- After running AC-3
  - Can have one solution left
  - Can have multiple solutions left
  - Can have no solutions left (and not know it)

Arc-consistent but no feasible assignment

What went wrong here?
COMPLEXITY

• CSPs in general are NP-complete
• Valued, optimization version of CSPs are usually NP-hard
• Some structured domains, like those with a constraint tree, are easier and can be solved in polynomial time
**Constraint Trees**

- Constraint tree
  - Any 2 variables in constraint graph connected by $\leq 1$ path
- Can be solved in time *linear in # of variables*

![Constraint Tree Diagram](image)
Algorithm for CSP Trees

1) Choose any var as root and order vars such that every var’s parents in constraint graph precede it in ordering

2) Let $X_i$ be the parent of $X_j$ in the new ordering

3) For $j=n:2$, run arc consistency to arc $(X_i, X_j)$

4) For $j=1:n$, assign val for $X_j$ consistent w/val assigned for $X_i$
1) Choose any var as root and order vars such that every var’s parents in constraint graph precede it in ordering

2) Let $X_i$ be the parent of $X_j$ in the new ordering
3) For $j=n:2$, run arc consistency to arc $(X_i, X_j)$
4) For $j=1:n$, assign val for $X_j$ consistent w/val assigned for $X_i$
SUMMARY

• Be able to define real world CSPs
• Understand basic algorithm (backtracking)
  o Complexity relative to basic search algorithms
  o Doesn’t require problem specific heuristics
  o Ideas shaping search (ordering heuristics)
• Pruning space through propagating information
  o Arc consistency
  o Tradeoffs: + reduces search space, - computation costs
• Computational complexity and special cases (tree)
• Relevant reading: R&N Chapter 6