CMU 15-381
CSPs

Professor: Alex Waibel
Today: Joseph Giampapa

With thanks to Emma Brunskill, Ariel Procaccia and other prior instructions for slides
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
Constraint Satisfaction Problems (CSPs)

- **Variables**: \( V = \{V_1, \ldots, V_N\} \)
- **Domain**: Set of \( d \) possible values for each variable
- **Constraints**: \( C = \{C_1, \ldots, C_K\} \)
- A constraint consists of
  - variable tuple
  - list of possible values for tuple (ex. \([(V_2, V_3),((R,B),(R,G))]\))
  - Or function that describes possible values (ex. \( V_2 \neq V_3 \))
- Allows useful general-purpose algorithms with more power than standard search algorithms


Overview

• Real world CSPs
• Basic algorithms for solving CSPs
• Pruning space through propagating information
Overview

- Real world CSPs
- Basic algorithms for solving CSPs
- Pruning space through propagating information
Example: Map Coloring

Color a map so that adjacent areas are different colors
Map Coloring

Variables

Domain

Constraints

Solutions

\(WA, NT, Q, NSW\)

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

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

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

• Variables:

• Domain:

• Constraints:
Example: Sudoku

- Variables:
  - Each open sqr
- Domain:
  - \{1:9\}
- Constraints:
  - 9-way all diff col
  - 9-way all diff row
  - 9-way all diff box
Scheduling (Important Ex.)

- 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.:
  - *Job shop*: Discrete time; weird ordering of operations possible; set of separate jobs.
  - *Batch shop*: Discrete or continuous time; restricted operation of ordering; grouping is important.
Job Scheduling

• A set of N jobs, $J_1, \ldots, J_n$.
• Each job $j$ is a seq of operations $O_{j1}, \ldots, O_{Lj}$.
• Each operation may use resource $R$, and has a specific duration in time.
• A resource must 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.
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
Overview

• Real world CSPs
• Basic algorithms for solving CSPs
• Pruning space through propagating information
Why not just do basic search algorithms from last time?
Backtracking

• Only consider a single variable at each point
• Don’t care about path!
Backtracking

• Only consider a single variable at each point
• Don’t care about path!
• Order of variable assignment doesn’t matter, so fix ordering
Backtracking

- 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
Backtracking

• 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 soln or fail
  o  Return Backtrack({},csp)

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

- Variables: A, B, C, D
- Domain: \{1, 2, 3\}
- Constraints:
  - A ≠ B, A=D, C < B, C < D
- Variable order: ?
- Value order: ?
**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
  - $V_i \leftarrow \text{select\_unassigned\_var}(csp)$
  - For each val in order-domain-values(var,csp,assign)
    - If value is consistent with assignment
      - Add [$V_i = \text{val}$] to assignment
      - Result $\leftarrow$ Backtrack(assignment,csp)
      - If Result $\neq$ fail, return result
      - Remove [$V_i = \text{val}$] from assignments
    - Return fail


Think and discuss

• Does the value order used affect how long backtracking takes to find a solution?
• Does the 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 ≠ 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 ≠ B, A=D, C < B, C < D

Variable order: alphabetical  Value order: Ascending

• (A=1)
• (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
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)
- (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 A = D
- (A=2, B=3, C=1, D=3) inconsistent with A = D
- Return result = fail
  - Backtrack to (A=2, B=3, C=1, D=2) ALL VALID
Ordering Matters!

• Function Backtracking(csp) returns soln or fail
  o Return Backtrack({},csp)
• Function Backtrack(assignment,csp) returns soln or fail
  o If assignment is complete, return assignment
  o $V_i \leftarrow \text{select_unassigned_var}(csp)$
  o 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
    o Return fail
Min Remaining Values (MRV)

- Choose variable with minimum number of remaining values in its domain
- Why min rather than max?
Min Remaining Values (MRV)

• Choose variable with minimum number of remaining values in its domain
• Most constrained variable
• “Fail-fast” ordering
Least Constraining Value

• Given choice of variable:
  o Choose least constraining value
  o Aka value that rules out the least values in the remaining variables to be assigned
  o May take some computation to find this

• Why least rather than most?
Click! 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

- Function Backtracking(csp) returns soln or fail
  - Return Backtrack({},csp)
- Function Backtrack(assignment,csp) returns soln or fail
  - If assignment is complete, return assignment
  - $V_i \leftarrow \text{select}_\text{unassigned}_\text{var}(csp)$
  - 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
Constraint Graphs

- Nodes are variables
- Arcs show constraints
Propagate Information

• If we choose a value for one variable, that affects its neighbors
• And then potentially those neighbors...
• Prunes the space of
Arc Consistency

• Definition:
  o An “arc” (connection between two variables $X \rightarrow Y$ in constraint graph) is consistent if:
  o For every value could assign to $X$
    There exists some value of $Y$ that could be assigned without violating a constraint
AC-3 (Assume binary constraints)

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

Have to add in arc for \((X_i,X_j)\) and \((X_j,X_i)\) for \(i,j\) constraint
**AC-3 (Assume Binary Constraints)**

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

---

**Function Revise**(csp\(X_i, X_j\)) returns Domain\(X_i\) and any\text{ChangeToDomain}X_i

- any\text{ChangeToDomain}X_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\(X_i\)
    - any\text{ChangeToDomain}X_i = true

---

Have to add in arc for \((X_i, X_j)\) and \((X_j, X_i)\) for \(i, j\) constraint
AC-3 Computational Complexity?

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

---

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? \(D^2\)
AC-3 Computational Complexity?

- Input: CSP
- Output: CSP, possible with reduced domains for variables, or inconsistent
- Local variables: stack, initially stack of all arcs (binary constraints in csp)
- While stack is not empty
  - \( (X_i,X_j) = \text{Remove-First}(\text{stack}) \)
  - \[ \text{domain}X_i, \text{anyChangeToDomain}X_i \] = \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 stack
  - Return csp

---

Have to add an 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 stack?

---

**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
AC-3 Computational Complexity?

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

-----------------------------------------------

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

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

\[D \text{ domain values} \]
\[C \text{ binary constraints} \]

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

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

Total: \(CD^3\)
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)
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
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
• stack: 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: \(A \neq B, B \neq A, C < B, B > C, C < D, D > C\)
- stack: 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 ≠ 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
- Stack: AB, BA, BC, CB, CD, DC
- Pop AB
- Stack: 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: \(A \neq B, B \neq A, C < B, B > C, C < D, D > C\)
- stack: AB, BA, BC, CB, CD, DC
- Pop AB
- stack: 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 ≠ 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
- stack: AB, BA, BC, CB, CD, DC
- stack: BA, BC, CB, CD, DC
- stack: 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 stack: AB**
- stack: AB, CB, CD, DC
AC-3 Example

- stack: AB, BA, BC, CB, CD, DC  A-D = \{1,2,3\}
- stack: BA, BC, CB, CD, DC  A-D = \{1,2,3\}
- stack: BC, CB, CD, DC  A-D = \{1,2,3\}
- stack: AB, CB, CD, DC  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 ≠ B?
  - Yes, so no change
AC-3 Example

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

- stack: AB, BA, BC, CB, CD, DC, A-D = \{1, 2, 3\}
- stack: BA, BC, CB, CD, DC A-D = \{1, 2, 3\}
- stack: BC, CB, CD, DC A-D = \{1, 2, 3\}
- stack: AB, CB, CD, DC B = \{2, 3\}, A/C/D = \{1, 2, 3\}
- stack: 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 stack: no change because already in
AC-3 Example

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

- stack: AB, BA, BC, CB, CD, DC, A-D = \{1, 2, 3\}
- stack: BA, BC, CB, CD, DC, A-D = \{1, 2, 3\}
- stack: BC, CB, CD, DC, A-D = \{1, 2, 3\}
- stack: AB, CB, CD, DC, B=\{2, 3\}, A/C/D = \{1, 2, 3\}
- stack: CB, CD, DC, B=\{2, 3\}, A/C/D = \{1, 2, 3\}
- stack: 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: \( A \neq B \), \( C < B \), \( C < D \)

- stack: AB, BA, BC, CB, CD, DC \( A-D = \{1, 2, 3\} \)
- stack: BA, BC, CB, CD, DC \( A-D = \{1, 2, 3\} \)
- stack: BC, CB, CD, DC \( A-D = \{1, 2, 3\} \)
- stack: AB, CB, CD, DC \( B = \{2, 3\}, A/C/D = \{1, 2, 3\} \)
- stack: CB, CD, DC \( B = \{2, 3\}, A/C/D = \{1, 2, 3\} \)
- stack: CD, DC \( B = \{2, 3\}, C = \{1, 2\} \ A, D = \{1, 2, 3\} \)
- stack: DC \( B = \{2, 3\}, C = \{1, 2\} \ A, D = \{1, 2, 3\} \)
- 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

- stack: AB, BA, BC, CB, CD, DC  \(A-D = \{1, 2, 3\}\)
- stack: BA, BC, CB, CD, DC  \(A-D = \{1, 2, 3\}\)
- stack: BC, CB, CD, DC  \(A-D = \{1, 2, 3\}\)
- stack: AB, CB, CD, DC  \(B=\{2, 3\}, A/C/D = \{1, 2, 3\}\)
- stack: CB, CD, DC  \(B=\{2, 3\}, A/C/D = \{1, 2, 3\}\)
- stack: CD, DC  \(B=\{2, 3\}, C = \{1, 2\} A,D = \{1, 2, 3\}\)
- stack: DC  \(B=\{2, 3\}, C = \{1, 2\} A,D = \{1, 2, 3\}\)
- \(A = \{1, 2, 3\} B=\{2, 3\}, C = \{1, 2\} D = \{2, 3\}\)
Forward Checking

- When assign a variable, make all of its neighbors arc-consistent
Backtracking + Forward Checking

- 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 `order-domain-values(var, csp, 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 \( \leftarrow \text{Backtrack}(assignment, csp) \)
      - If Result \( \neq \text{fail} \), return result
    - Remove \([V_i = \text{val}]\) from assignments
  - Return fail
Maintaining Arc Consistency

• Forward checking doesn’t ensure all arcs are consistent
• AC-3 detects failure faster than forward checking
• 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 ← 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
      - Run AC-3 to make all variables arc-consistent with `[V_i = val]`.
        - Initial stack is arcs `(X_j, V_i)` of neighbors of `V_i` that are unassigned, but add other arcs if these vars change domains.
      - Result ← `Backtrack(assignment, csp)`
      - If Result ≠ fail, return result
      - Remove `[V_i = val]` from assignments
  - Return fail
Sufficient to Avoid Backtracking?

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

- A) True
- B) False
AC-3 Limitations

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

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

What went wrong here?
Complexity

• CSP in general are NP-hard
• Some structured domains are easier
**Constraint Trees**

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

---

Figure from Russell & Norvig
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 $X_i$

Figure from Russell & Norvig
Computational Complexity?

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

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

Figure from Russell & Norvig
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 (LCV, etc)
• Pruning space through propagating information
  o Arc consistency
  o Tradeoffs: + reduces search space, - costs computation
• Computational complexity and special cases (tree)
• Relevant reading: R&N Chapter 6