Warm-up as You Walk In

Assign Red, Green, or Blue to each node
Neighbors must be different

1) What is your brain doing to solve these?
2) How would you solve these with search (BFS, DFS, etc.)?
Plan

Last Time
- Adversarial search
  - Minimax
  - Evaluation functions
  - Pruning
  - Expectimax

Today
- Constraint Satisfaction Problems
AI: Representation and Problem Solving
Constraint Satisfaction Problems (CSPs)

Instructors: Tuomas Sandholm and Nihar Shah
Slide credits: CMU AI, http://ai.berkeley.edu
What is Search For?

• Planning: sequences of actions
  • The path to the goal is the important thing
  • Paths have various costs, depths
  • Heuristics give problem-specific guidance

• Identification: assignments to variables
  • The goal itself is important, not the path
  • All paths at the same depth (for some formulations)

Are the warm-up assignments (i.e., sudoku) planning or identification problems?
Constraint Satisfaction Problems

CSP is a special class of search problems

- Mostly identification problems
- Have specialized algorithms for them

Standard search problems:

- State is an arbitrary data structure
- Goal test can be any function over states

Constraint satisfaction problems (CSPs):

- State is defined by variables $X_i$ with values from a domain $D$ (sometimes $D$ depends on $i$)
- Goal test is a set of constraints specifying allowable combinations of values for subsets of variables
Why study CSPs?

Many real-world problems can be formulated as CSPs

- Assignment problems: e.g., who teaches what class
- Timetabling problems: e.g., which class is offered when and where?
- Hardware configuration
- Transportation scheduling
- Factory scheduling
- Circuit layout
- Fault diagnosis
- ... lots more!

- Sometimes involve real-valued variables...
Varieties of CSPs and Constraints
Example: Map Coloring

- Variables: WA, NT, Q, NSW, V, SA, T
- Domains: \( D = \{ \text{red, green, blue} \} \)
- Constraints: adjacent regions must have different colors
  - Implicit: WA \( \neq \) NT
  - Explicit: \((WA, NT) \in \{(\text{red, green}), (\text{red, blue}), \ldots \}\)  

- Solutions are assignments satisfying all constraints, e.g.:
  \{WA=\text{red}, NT=\text{green}, Q=\text{red}, NSW=\text{green}, V=\text{red}, SA=\text{blue}, T=\text{green} \}
Constraint Graphs

Map of Australia showing states and territories:
- Western Australia
- Northern Territory
- South Australia
- New South Wales
- Victoria
- Tasmania

Diagram on the right:
- NT (Northern Territory)
- Q (Queensland)
- WA (Western Australia)
- SA (South Australia)
- NSW (New South Wales)
- V (Victoria)
- T (Tasmania)
Constraint Graphs

- Binary CSP: each constraint relates (at most) two variables
- Binary constraint graph: nodes are variables, arcs show constraints
- General-purpose CSP algorithms use the graph structure to speed up search. E.g., Tasmania is an independent subproblem!
Example: N-Queens

• Formulation 1:
  • Variables: $X_{ij}$
  • Domains: $\{0, 1\}$
  • Constraints

\begin{align*}
\forall i, j, k \ (X_{ij}, X_{ik}) &\in \{(0, 0), (0, 1), (1, 0)\} \\
\forall i, j, k \ (X_{ij}, X_{kj}) &\in \{(0, 0), (0, 1), (1, 0)\} \\
\forall i, j, k \ (X_{ij}, X_{i+k,j+k}) &\in \{(0, 0), (0, 1), (1, 0)\} \\
\forall i, j, k \ (X_{ij}, X_{i+k,j-k}) &\in \{(0, 0), (0, 1), (1, 0)\} \\
\sum_{i,j} X_{ij} &= N
\end{align*}
Example: N-Queens

- **Formulation 2:**
  - **Variables:** $Q_k$
  - **Domains:** $\{1, 2, 3, \ldots N\}$
  - **Constraints:**
    
    **Implicit:** $\forall i, j \text{ non-threatening}(Q_i, Q_j)$

    **Explicit:** $(Q_1, Q_2) \in \{(1, 3), (1, 4), \ldots\}$
    
    \[\ldots\]
Example: Cryptarithmetic

• Variables:

\[ F \ T \ U \ W \ R \ O \ X_1 \ X_2 \ X_3 \]

• Domains:

\[ \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\} \]

• Constraints:

\[ \text{alldiff}(F, T, U, W, R, O) \]

\[ O + O = R + 10 \cdot X_1 \]

\[ \ldots \]
Example: Sudoku

• Variables: Each (open) square
• Domains: \{1,2,\ldots,9\}
• Constraints:
  9-way alldiff for each column
  9-way alldiff for each row
  9-way alldiff for each region
  (or can have a bunch of pairwise inequality constraints)
Varieties of CSPs

- **Discrete Variables**  
  - Finite domains
    - Size $d$ means $O(d^n)$ complete assignments
    - E.g., Boolean CSPs, including Boolean satisfiability (NP-complete)
  - Infinite domains (integers, strings, etc.)
    - E.g., job scheduling, variables are start/end times for each job
    - Linear constraints solvable, nonlinear undecidable

  **We will cover today**

- **Continuous variables**
  - E.g., start/end times for Hubble Telescope observations
  - Linear constraints solvable in polynomial time

  **We will cover in a later lecture (linear programming)**
Varieties of Constraints

• Varieties of Constraints
  • Unary constraints involve a single variable (equivalent to reducing domains), e.g.:
    \[ SA \neq \text{green} \]
  
  • Binary constraints involve pairs of variables, e.g.:
    \[ SA \neq WA \]
  
  • Higher-order constraints involve 3 or more variables:
    e.g., cryptarithmetic column constraints
    \[ O + O = R + 10 \cdot X_1 \]

• Preferences (soft constraints):
  • E.g., red is better than green
  • Often representable by a cost for each variable assignment
  • Gives constrained optimization problems
Solving CSPs
Standard Search Formulation

• Standard search formulation of CSPs

• States defined by the values assigned so far (partial assignments)
  • Initial state: the empty assignment, {}
  • Successor function: assign a value to an unassigned variable
  • Goal test: the current assignment is complete and satisfies all constraints

• We’ll start with the straightforward, naïve approach, then improve it
Question: Search for CSPs

Should we use BFS or DFS?
Depth First Search

• At each node, assign a value from the domain to the variable
• Check feasibility (constraints) when the assignment is complete
Demo – Naïve Search
Backtracking Search
Backtracking Search

- Backtracking search is the basic uninformed algorithm for solving CSPs
- Backtracking search = DFS + two improvements

Idea 1: One variable at a time
- Variable assignments are commutative
  - [WA = red then NT = green] same as [NT = green then WA = red]
  - Only need to consider assign value to a single variable at each step

Idea 2: Check constraints as you go
- Consider only values which do not conflict previous assignments
- May need some computation to check the constraints
- “Incremental goal test”

Can solve n-queens for n ≈ 25
Backtracking Example
Backtracking Search

function BACKTRACKING-SEARCH(csp) returns solution/failure
    return RECURSIVE-BACKTRACKING(\{ \}, csp)

function RECURSIVE-BACKTRACKING(assignment, csp) returns soln/failure
    if assignment is complete then return assignment
    var ← SELECT-UNASSIGNED-VARIABLE(VARIABLES[csp], assignment, csp)
    for each value in ORDER-DOMAIN-VALUES(var, assignment, csp) do
        if value is consistent with assignment given CONSTRAINTS[csp] then
            add \{ var = value \} to assignment
            result ← RECURSIVE-BACKTRACKING(assignment, csp)
            if result ≠ failure then return result
            remove \{ var = value \} from assignment
        return failure
Backtracking Search

function Backtracking-Search(csp) returns solution/failure
    return Recursive-Backtracking({}, csp)

function Recursive-Backtracking(assignment, csp) returns solution/failure
    if assignment is complete then return assignment
    var ← Select-Unassigned-Variable(VARIABLES[csp], assignment, csp)
    for each value in Order-Domain-Values(var, assignment, csp) do
        if value is consistent with assignment given CONSTRAINTS[csp] then
            add {var = value} to assignment
            result ← Recursive-Backtracking(assignment, csp)
            if result ≠ failure then return result
            remove {var = value} from assignment
    return failure.
Backtracking Search

```
function Backtracking-Search(csp) returns solution/failure
    return Recursive-Backtracking({}, csp)

function Recursive-Backtracking(assignment, csp) returns soln/failure
    if assignment is complete then return assignment
    var ← Select-Unassigned-Variable(VARIABLES[csp], assignment, csp)
    for each value in Order-Domain-Values(var, assignment, csp) do
        if value is consistent with assignment given CONSTRAINTS[csp] then
            add {var = value} to assignment
            result ← Recursive-Backtracking(assignment, csp)
            if result ≠ failure then return result
            remove {var = value} from assignment
    return failure
```
Backtracking Search

```
function BACKTRACKING-SEARCH(csp) returns solution/failure
    return RECURSIVE-BACKTRACKING({}, csp)

function RECURSIVE-BACKTRACKING(assignment, csp) returns soln/failure
    if assignment is complete then return assignment
    var ← SELECT-UNASSIGNED-VARIABLE(VARIABLES[csp], assignment, csp)
    for each value in ORDER-DOMAIN-VALUES(var, assignment, csp) do
        if value is consistent with assignment given CONSTRAINTS[csp] then
            add {var = value} to assignment
            result ← RECURSIVE-BACKTRACKING(assignment, csp)
            if result ≠ failure then return result
        remove {var = value} from assignment
    return failure
```

No need to check constraints for a complete assignment
Backtracking Search

function `BACKTRACKING-SEARCH`\((csp)\) returns solution/failure

    return `RECURSIVE-BACKTRACKING`\(\{\}, csp\)

function `RECURSIVE-BACKTRACKING`\((assignment, csp)\) returns soln/failure

    if `assignment` is complete then return `assignment`

    \(var \leftarrow \text{SELECT-UNASSIGNED-VARIABLE}(\text{VAR}\{S\}, assignment, csp)\)

    for each \(value\) in \text{ORDER-DOMAIN-VALUES}(\text{var}, assignment, csp) do

        if `value` is consistent with `assignment` given \text{CONSTRAINTS}\(csp\) then

            add \(\{\text{var} = \text{value}\}\) to `assignment`

            `result` \(\leftarrow \text{RECURSIVE-BACKTRACKING}(\text{assignment}, csp)\)

            if `result` \(\neq\) failure then return `result`

            remove \(\{\text{var} = \text{value}\}\) from `assignment`

        return failure

Checks consistency at each assignment
Backtracking Search

function BACKTRACKING-Search(csp) returns solution/failure
    return RECURSIVE-BACKTRACKING(∅, csp)

function RECURSIVE-BACKTRACKING(assignment, csp) returns soln/failure
    if assignment is complete then return assignment

    var ← SELECT-Unassigned-Variable(Variables[csp], assignment, csp)
    for each value in ORDER-DOMAIN-VALUES(var, assignment, csp) do
        if value is consistent with assignment given Constraints[csp] then
            add {var = value} to assignment
            result ← RECURSIVE-BACKTRACKING(assignment, csp)
            if result ≠ failure then return result
            remove {var = value} from assignment
        return failure

• Backtracking = DFS + variable-ordering + fail-on-violation

• What are the decision points?
Improving Backtracking

• General-purpose ideas give huge gains in speed

• Filtering: Can we detect inevitable failure early?

• Ordering:
  • Which variable should be assigned next?
  • In what order should its values be tried?

• Structure: Can we exploit the problem structure?
  Not going to cover!
Filtering
Filtering: Forward Checking

Filtering: Keep track of domains for unassigned variables and cross off bad options

Forward checking: A simple way for filtering
  • After a variable is assigned a value, check related constraints and cross off values of unassigned variables which violate the constraints
  • Failure detected if some variables have no values remaining
Filtering: Forward Checking

- Filtering: Keep track of domains for unassigned variables and cross off bad options
- Forward checking: A simple way for filtering
  - After a variable is assigned a value, check related constraints and cross off values of unassigned variables which violate the constraints
  - Failure detected if some variables have no values remaining
Filtering: Forward Checking

- Filtering: Keep track of domains for unassigned variables and cross off bad options
- Forward checking: A simple way for filtering
  - After a variable is assigned a value, check related constraints and cross off values of unassigned variables which violate the constraints
  - Failure detected if some variables have no values remaining

Recall: Binary constraint graph for a binary CSP (i.e., each constraint has most two variables): nodes are variables, edges show constraints
Filtering: Forward Checking

• Filtering: Keep track of domains for unassigned variables and cross off bad options
• Forward checking: A simple way for filtering
  • After a variable is assigned a value, check related constraints and cross off values of unassigned variables which violate the constraints
  • Failure detected if some variables have no values remaining
Filtering: Forward Checking

- Filtering: Keep track of domains for unassigned variables and cross off bad options
- Forward checking: A simple way for filtering
  - After a variable is assigned a value, check related constraints and cross off values of unassigned variables which violate the constraints
  - Failure detected if some variables have no values remaining

FAIL – variable with no possible values
Demo – Backtracking with Forward Checking
Filtering: Constraint Propagation

- Limitations of simple forward checking: propagates information from assigned to unassigned variables, but doesn't provide early detection for all failures
  - NT and SA cannot both be blue! Why didn’t we detect this yet?
- Constraint propagation: reason from constraint to constraint
Consistency of A Single Arc

- An arc \( X \rightarrow Y \) is consistent if for every \( x \) in the tail there is some \( y \) in the head which could be assigned without violating a constraint.
- Enforce arc consistency: Remove values in domain of \( X \) if no corresponding legal \( Y \) exists.
- Forward checking: Only enforce \( X \rightarrow Y \), \( \forall (X, Y) \in E \) and \( Y \) newly assigned.

(Recall: Binary constraint graph for a binary CSP (i.e., each constraint has most two variables): nodes are variables, edges show constraints.)
Consistency of A Single Arc

• An arc $X \rightarrow Y$ is consistent if for every $x$ in the tail there is some $y$ in the head which could be assigned without violating a constraint.

• Enforce arc consistency: Remove values in domain of $X$ if no corresponding legal $Y$ exists.

• Forward checking: Only enforce $X \rightarrow Y$, $\forall (X, Y) \in E$ and $Y$ newly assigned.
How to Enforce Arc Consistency of Entire CSP

- A simplistic algorithm: Cycle over the pairs of variables, enforcing arc-consistency, repeating the cycle until no domains change for a whole cycle.
- AC-3 (short for Arc Consistency Algorithm #3): A more efficient algorithm ignoring constraints that have not been modified since they were last analyzed.
AC-3: Enforce Arc Consistency of Entire CSP

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) ← REMOVE-FIRST(queue)
    if REMOVE-INCONSISTENT-VALUES(X_i, X_j) then
        for each X_k in Neighbors[X_i] do
            add (X_k, X_i) to queue

function REMOVE-INCONSISTENT-VALUES(X_i, X_j) returns true iff succeeds
removed ← false
for each x in DOMAIN[X_i] do
    if no value y in DOMAIN[X_j] allows (x, y) to satisfy the constraint X_i ← X_j
    then delete x from DOMAIN[X_i]; removed ← true
return removed

Constraint Propagation!
AC-3: Enforce Arc Consistency of Entire CSP

Queue:
SA->WA
NT->WA

Remember: Delete from the tail!
AC-3: Enforce Arc Consistency of Entire CSP

Remember: Delete from the tail!
AC-3: Enforce Arc Consistency of Entire CSP

Queue:
WA->SA
NT->SA
Q->SA
NSW->SA
V->SA
WA->NT
SA->NT
Q->NT

Remember: Delete from the tail!
AC-3: Enforce Arc Consistency of Entire CSP

Queue:
- WA->SA
- NT->SA
- Q->SA
- NSW->SA
- V->SA
- WA->NT
- SA->NT
- Q->NT

Remember: Delete from the tail!
AC-3: Enforce Arc Consistency of Entire CSP

Remember: Delete from the top!
AC-3: Enforce Arc Consistency of Entire CSP

Queue:
Q->SA
NSW->SA
V->SA
WA->NT
SA->NT
Q->NT

Remember: Delete from the top!
AC-3: Enforce Arc Consistency of Entire CSP

Queue:
NSW→SA
V→SA
WA→NT
SA→NT
Q→NT

Remember: Delete from the tail!
AC-3: Enforce Arc Consistency of Entire CSP

Remember: Delete from the tail!
Poll 1: After assigning Q to Green, what gets added to the Queue?

A: NSW->Q, SA->Q, NT->Q
B: Q->NSW, Q->SA, Q->NT
AC-3: Enforce Arc Consistency of Entire CSP

Queue:  
NT->Q  
SA->Q  
NSW->Q

Remember: Delete from the tail!
AC-3: Enforce Arc Consistency of Entire CSP

Queue:
SA->Q
NSW->Q
WA->NT
SA->NT
Q->NT

Remember: Delete from the tail!
AC-3: Enforce Arc Consistency of Entire CSP

Queue: NSW->Q, WA->NT, SA->NT, Q->NT, WA->SA, NT->SA, Q->SA, NSW->SA, V->SA

Remember: Delete from the tail!
AC-3: Enforce Arc Consistency of Entire CSP

Queue:
- WA->NT
- SA->NT
- Q->NT
- WA->SA
- NT->SA
- Q->SA
- NSW->SA
- V->SA
- V->NSW
- Q->NSW
- SA->NSW

Remember: Delete from the tail!
AC-3: Enforce Arc Consistency of Entire CSP

Queue:
- WA→NT
- SA→NT
- Q→NT
- WA→SA
- NT→SA
- Q→SA
- NSW→SA
- V→SA
- V→NSW
- Q→NSW
- SA→NSW

Remember: Delete from the tail!
AC-3: Enforce Arc Consistency of Entire CSP
AC-3: Enforce Arc Consistency of Entire CSP

- Backtrack on the assignment of Q
- Arc consistency detects failure earlier than forward checking
- Can be run as a preprocessor or after each assignment
- What’s the downside of enforcing arc consistency?

Queue:
- SA->NT
- Q->NT
- WA->SA
- NT->SA
- Q->SA
- NSW->SA
- V->SA
- V->NSW
- Q->NSW
- SA->NSW

Remember: Delete from the tail!
Limitations of Arc Consistency

• After enforcing arc consistency:
  • Can have one solution left
  • Can have multiple solutions left
  • Can have no solutions left (and not know it)

• Arc consistency only checks local consistency conditions

• Arc consistency still runs inside a backtracking search!

What went wrong here?
Backtracking Search with AC-3

```
function Backtracking-Search(csp) returns solution/failure
    return Recursive-Backtracking({}, csp)

function Recursive-Backtracking(assignment, csp) returns soln/failure
    if assignment is complete then return assignment
    var ← Select-Unassigned-Variable(VARIABLES[csp], assignment, csp)
    for each value in Order-Domain-Values(var, assignment, csp) do
        if value is consistent with assignment given CONSTRAINTS[csp] then
            add {var = value} to assignment
            AC-3(csp)
            result ← Recursive-Backtracking(assignment, csp)
            if result ≠ failure then return result
            remove {var = value} from assignment
    return failure
```

- Where do you run AC-3?
Demo – Backtracking with AC-3
Complexity of a single run of AC-3

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
    \((X_i, X_j) \leftarrow\) REMOVE-FIRST(queue)
    if REMOVE-INCONSISTENT-VALUES(X_i, X_j) then
        for each \(X_k\) in Neighbors[X_i] do
            add \((X_k, X_i)\) to queue

function REMOVE-INCONSISTENT-VALUES(X_i, X_j) returns true iff succeeds
removed ← false
for each \(x\) in Domain[X_i] do
    if no value \(y\) in Domain[X_j] allows \((x, y)\) to satisfy the constraint \(X_i \leftarrow X_j\)
    then delete \(x\) from Domain[X_i]; removed ← true
return removed

Recall that the whole backtracking algorithm with AC-3 will call AC-3 many times.
Complexity of a single run of AC-3

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) ← REMOVE-FIRST(queue)
if REMOVE-INCONSISTENT-VALUES(X_i, X_j) then
    for each X_k in Neighbors[X_i] do
        add (X_k, X_i) to queue

function REMOVE-INCONSISTENT-VALUES(X_i, X_j) returns true iff succeeds
removed ← false
for each x in DOMAIN[X_i] do
    if no value y in DOMAIN[X_j] allows (x, y) to satisfy the constraint X_i ← X_j
    then delete x from DOMAIN[X_i]; removed ← true
return removed

• An arc is added after a removal of value at a node
• \(n\) nodes in total, each has \(\leq d\) values
• Total times of removal: \(O(n^d)\)
Complexity of a single run of AC-3

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 \(queue\)

end if

end while

function \(\text{REMOVE-INCONSISTENT-VALUES}(X_i, X_j)\) returns true iff succeeds

\(removed \leftarrow \text{false}\)

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 \(\text{DOMAIN}[X_i]\); \(removed \leftarrow \text{true}\)

end if

end for

return \(removed\)

- An arc is added after a removal of value at a node
- \(n\) nodes in total, each has \(\leq d\) values
- Total times of removal: \(O(nd)\)
- After a removal, \(\leq n\) arcs added
- Total times of adding arcs: \(O(n^2d)\)
Complexity of a single run of AC-3

function $\text{AC-3}(\text{csp})$ returns the CSP, possibly with reduced domains
inputs: $\text{csp}$, a binary CSP with variables $\{X_1, X_2, \ldots, X_n\}$
local variables: $\text{queue}$, a queue of arcs, initially all the arcs in $\text{csp}$

while $\text{queue}$ is not empty do
    $(X_i, X_j) \leftarrow \text{REMOVE-FIRST}(\text{queue})$
    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 $\text{queue}$

function $\text{REMOVE-INCONSISTENT-VALUES}(X_i, X_j)$ returns true iff succeeds
$\text{removed} \leftarrow \text{false}$
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 $X_i \leftarrow X_j$
        then delete $x$ from $\text{DOMAIN}[X_i]$; $\text{removed} \leftarrow \text{true}$
return $\text{removed}$

Complexity of a single run of AC-3 is at most $O(n^2d^3)$
(Not required) Zhang&Yap (2001) show that its complexity is $O(n^2d^2)$

• An arc is added after a removal of value at a node
• $n$ nodes in total, each has $\leq d$ values
• Total times of removal: $O(nd)$
• After a removal, $\leq n$ arcs added
• Total times of adding arcs: $O(n^2d)$
• Check arc consistency per arc: $O(d^2)$
Ordering
Backtracking Search

```
function BACKTRACKING-SEARCH(csp) returns solution/failure
    return RECURSIVE-BACKTRACKING({}, csp)

function RECURSIVE-BACKTRACKING(assignment, csp) returns soln/failure
    if assignment is complete then return assignment
    var ← SELECT-UNASSIGNED-VARIABLE(VARIABLES[csp], assignment, csp)
    for each value in ORDER-DOMAIN-VALUES(var, assignment, csp) do
        if value is consistent with assignment given CONSTRAINTS[csp] then
            add \{var = value\} to assignment
            result ← RECURSIVE-BACKTRACKING(assignment, csp)
            if result ≠ failure then return result
        remove \{var = value\} from assignment
    return failure
```

- Backtracking = DFS + variable-ordering + fail-on-violation
- What are the decision points?
Question for the class

• Would it be better to branch on the most constrained or the least constrained variable next?
Most constrained variable heuristic

• Choose the variable with the fewest legal values

• a.k.a. minimum remaining values (MRV) heuristic
Most constraining variable heuristic

• Choose the variable with the most constraints on remaining variables
• A good idea is to use it as a tie-breaker among most constrained variables:
Least constraining value heuristic

- Given a variable to assign, choose the least constraining value:
  - the one that rules out the fewest values in the remaining variables
  - Note that it may take some computation to determine this! (E.g., rerunning filtering)

- Why least rather than most?

- Combining these heuristics makes 1000 queens feasible
Demo – Coloring with a Complex Graph

Compare
• Backtracking with Forward Checking
• Backtracking with AC-3
• Backtracking + Forward Checking + Minimum Remaining Values (MRV)
• Backtracking + AC-3 + MRV + LCV
How to deal with non-binary CSPs?

• Variables:
  \[ F \ T \ U \ W \ R \ O \ X_1 \ X_2 \ X_3 \]

• Domains:
  \( \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\} \)

• Constraints:
  \[ \text{alldiff}(F, T, U, W, R, O) \]
  \[ O + O = R + 10 \cdot X_1 \]
  \[ \ldots \]
Constraint graph for non-binary CSPs

- Variable nodes: nodes to represent the variables
- Constraint nodes: auxiliary nodes to represent the constraints
- Edges: connects a constraint node and its corresponding variables

Constraints:

\[
\text{alldiff}(F, T, U, W, R, O)
\]
\[
O + O = R + 10 \cdot X_1
\]
\[
\cdots
\]
Solve non-binary CSPs

• Naïve search?
  • Yes!

• Backtracking?
  • Yes!

• Forward Checking?
  • Need to generalize the original FC operation
  • \((nFC0)\) After a variable is assigned a value, find all constraints with only one unassigned variable and cross off values of that unassigned variable which violate the constraint
  • There exist other ways to do generalized forward checking
Solve non-binary CSPs

• (Bonus material, not required)
• AC-3? Need to generalize the definition of AC and enforcement of AC
• Generalized arc-consistency (GAC)
  • A non-binary constraint is GAC if for every value for a variable there exist consistent value combinations for all other variables in the constraint
  • Reduced to AC for binary constraints
• Enforcing GAC
  • Simple schema: enumerate value combination for all other variables
  • $O(d^k)$ on $k$-ary constraint on variables with domains of size $d$

• There are other algorithms for non-binary constraint propagation, e.g., (i,j)-consistency [Freuder, JACM 85]
Summary: CSPs

- CSPs are a special kind of search problem:
  - States are partial assignments
  - Goal test defined by constraints

- Basic solution: backtracking search

- Speed-ups:
  - Ordering
  - Filtering
  - Structure
Additional Resources (Not required)

• References