CSE 573: Artificial Intelligence

Constraint Satisfaction Problems
Factored (aka Structured) Search

[With many slides by Dan Klein and Pieter Abbeel (UC Berkeley) available at http://ai.berkeley.edu.]
Final Presentations

- 21 groups / 40 people / 110 min
  - Minus transfers & tournament replay
- Presentations (with questions)
  - One person groups 2.5 min
  - Two person groups 4.5 min
  - Three person groups 6.5 min
- Everyone should speak (unless OOT)
- Rehearse
- Add URL for slides to g-doc
  - https://docs.google.com/spreadsheets/d/1Qt5BW0DkSAg6Q4MOM98jSSwjR2wTZpi5i01XdT0X-fs/edit#gid=0
Final report

- Default project ~2 pages
- Other projects ~6 pages
  - Experiments
  - Lessons learned
  - [http://courses.cs.washington.edu/courses/cse573/17wi/reports.html](http://courses.cs.washington.edu/courses/cse573/17wi/reports.html)
- Everyone
  - See note on appendices – dynamics & external code
AI Topics

- **Search**
  - Problem spaces
  - BFS, DFS, UCS, A* (tree and graph), local search
  - Completeness and Optimality
  - Heuristics: admissibility and consistency; pattern DBs

- **CSPs**
  - Constraint graphs, backtracking search
  - Forward checking, AC3 constraint propagation, ordering heuristics

- **Games**
  - Minimax, Alpha-beta pruning,
  - Expectimax
  - Evaluation Functions

- **MDPs**
  - Bellman equations
  - Value iteration, policy iteration

- **Reinforcement Learning**
  - Exploration vs Exploitation
  - Model-based vs. model-free
  - Q-learning
  - Linear value function approx.

- **Hidden Markov Models**
  - Markov chains, DBNs
  - Forward algorithm
  - Particle Filters

- **POMDPs**
  - Belief space
  - Piecewise linear approximation to value fun

- **Beneficial AI**

- **Bayesian Networks**
  - Basic definition, independence (d-sep)
  - Variable elimination
  - Sampling (rejection, importance)

- **Learning**
  - BN parameters with complete data
  - Search thru space of BN structures
  - Expectation maximization
What is intelligence?

- (bounded) Rationality
  - Agent has a performance measure to optimize
  - Given its state of knowledge
  - Choose optimal action
  - With limited computational resources

- Human-like intelligence/behavior
State-Space Search

- **X as a search problem**
  - states, actions, transitions, cost, goal-test

- **Types of search**
  - **uninformed systematic**: often slow
    - DFS, BFS, uniform-cost, iterative deepening
  - **Heuristic-guided**: better
    - Greedy best first, A*
    - Relaxation leads to heuristics
  - **Local**: fast, fewer guarantees; often local optimal
    - Hill climbing and variations
    - Simulated Annealing: global optimal
  - (Local) Beam Search
Which Algorithm?

- A*, Manhattan Heuristic:
Adversarial Search
Adversarial Search

- AND/OR search space (max, min)
- minimax objective function
- minimax algorithm (~dfs)
  - alpha-beta pruning
- Utility function for partial search
  - Learning utility functions by playing with itself
- Openings/Endgame databases
Policy Iteration

- Let $i = 0$
- Initialize $\pi_i(s)$ to random actions
- Repeat
  - Step 1: Policy evaluation:
    - Initialize $k=0$; For all $s$, $V_0^{\pi}(s) = 0$
    - Repeat until $V^\pi$ converges
      - For each state $s$, $V_{k+1}^{\pi_i}(s) \leftarrow \sum_{s'} T(s, \pi_i(s), s') \left[ R(s, \pi_i(s), s') + \gamma V_k^{\pi_i}(s') \right]$
      - Let $k += 1$
  - Step 2: Policy improvement:
    - For each state, $s$, $\pi_{i+1}(s) = \arg\max_a \sum_{s'} T(s, a, s') \left[ R(s, a, s') + \gamma V^{\pi_i}(s') \right]$
    - If $\pi_i = \pi_{i+1}$ then it’s optimal; return it.
    - Else let $i += 1$
Initialize $\pi_0$ to “always go right”

Perform policy evaluation

Perform policy improvement

Iterate through states

Has policy changed?

Yes! $i += 1$
Example

$\pi_1$ says “always go up”

Perform policy evaluation

Perform policy improvement
Iterate through states

Has policy changed?
No! We have the optimal policy
Reinforcement Learning

- For all \( s, a \)
  - Initialize \( Q(s, a) = 0 \)

- Repeat Forever
  - Where are you? \( s \).
  - Choose some action \( a \)
  - Execute it in real world: transition \( = (s, a, r, s') \)
  - Do update:

\[
\text{difference} = \left[ r + \gamma \max_{a'} Q(s', a') \right] - Q(s, a)
\]

\[
Q(s, a) \leftarrow Q(s, a) + \alpha \text{[difference]}
\]
Approximate Q-Learning

\[ Q(s, a) = w_1 f_1(s, a) + w_2 f_2(s, a) + \ldots + w_n f_n(s, a) \]

- **For all** \( s, a \)
  - Initialize \( w_i = 0 \)
- **Repeat Forever**
  - Where are you? \( s \).
  - **Choose some action** \( a \)
  - Execute it in real world: transition = \((s, a, r, s')\)
  - Do updates:
    \[ \text{difference} = r + \gamma \max_{a'} Q(s', a') - Q(s, a) \]
    \[ w_i \leftarrow w_i + \alpha \text{[difference]} f_i(s, a) \]
- **Interpretation as search**
  - Adjust weights of active features
  - E.g., if something unexpectedly bad happens, blame the active features
Approximate Q-Learning

\[ Q(s, a) = w_1 f_1(s, a) + w_2 f_2(s, a) + \ldots + w_n f_n(s, a) \]

- Q-learning with linear Q-functions:
  
  transition \( = (s, a, r, s') \)

  \[ Q(s, a) \leftarrow Q(s, a) + \alpha \text{[difference]} \]

  Old way: Exact Q’s

  \[ w_i \leftarrow w_i + \alpha \text{[difference]} f_i(s, a) \]

  Now: Approximate Q’s

- Intuitive interpretation:
  - Adjust weights of active features
  - E.g., if something unexpectedly bad happens, blame the features that were **active**: disprefer all states with that state’s features
What is Search For?

- **Planning**: sequences of actions
  - The *path to the goal* is the important thing
  - Paths have various costs, depths
  - Assume little about problem structure

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

CSPs are *structured* (factored) identification problems
Constraint Satisfaction Problems

- **Standard search problems:**
  - State is a “black box”: arbitrary data structure
  - Goal test can be any function over states
  - Successor function can also be anything

- **Constraint satisfaction problems (CSPs):**
  - A special subset of search problems
  - 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

- **Making use of CSP formulation allows for optimized algorithms**
  - Typical example of trading generality for utility (in this case, speed)
Constraint Satisfaction Problems

- “Factoring” the state space
- Representing the state space in a knowledge representation

Constraint satisfaction problems (CSPs):
- A special subset of search problems
- 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
CSP Example: N-Queens

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

$\forall i, j, k \ (X_{ij}, X_{ik}) \in \{(0, 0), (0, 1), (1, 0)\}$
$\forall i, j, k \ (X_{ij}, X_{k,j}) \in \{(0, 0), (0, 1), (1, 0)\}$
$\forall i, j, k \ (X_{ij}, X_{i+k,j+k}) \in \{(0, 0), (0, 1), (1, 0)\}$
$\forall i, j, k \ (X_{ij}, X_{i+k,j-k}) \in \{(0, 0), (0, 1), (1, 0)\}$

$\sum_{i,j} X_{ij} = N$
CSP Example: N-Queens

- **Formulation 2:**
  - **Variables:** $Q_k$
  - **Domains:** $\{1, 2, 3, \ldots N\}$

- **Constraints:**
  - **Implicit:** $\forall i, j$ non-threatening($Q_i, Q_j$)
  - **Explicit:**
    - $(Q_1, Q_2) \in \{(1, 3), (1, 4), \ldots\}$
    - $\cdots$
CSP Example: Sudoku

- Variables:
  - Each (open) square
- Domains:
  - \{1,2,...,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)
Propositional Logic

\[( (p \leftrightarrow q) \land r ) \lor ( p \land q \land \sim r ) \]

- Variables: propositional variables
- Domains: \{T, F\}
- Constraints: logical formula
CSP 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: \( \text{WA} \neq \text{NT} \)
  - Explicit: \( (\text{WA}, \text{NT}) \in \{(\text{red, green}), (\text{red, blue}), \ldots\} \)
- **Solutions** are assignments satisfying all constraints, e.g.:
  \[ \{\text{WA=red, NT=green, Q=red, NSW=green, V=red, SA=blue, T=green}\} \]
Constraint Graphs
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: 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:**
  \[ \text{alldiff}(F, T, U, W, R, O) \]
  \[ O + O = R + 10 \cdot X_1 \]
  \[ \ldots \]
Chinese Constraint Network

- Soup
  - Total Cost < $40
  - Must be Hot&Sour
- Chicken Dish
  - No Peanuts
- Vegetable
  - No Peanuts
- Seafood
  - Not Both Spicy
- Pork Dish
- Appetizer
- Rice
  - Not Chow Mein
Real-World CSPs

- Assignment problems: e.g., who teaches what class
- Timetabling problems: e.g., which class is offered when and where?
- Hardware configuration
- Gate assignment in airports
- Space Shuttle Repair
- Transportation scheduling
- Factory scheduling
- ... lots more!
Example: The Waltz Algorithm

- The Waltz algorithm is for interpreting line drawings of solid polyhedra as 3D objects
- An early example of an AI computation posed as a CSP
Waltz on Simple Scenes

- **Assume all objects:**
  - Have no shadows or cracks
  - Three-faced vertices
  - “General position”: no junctions change with small movements of the eye.

- **Then each line on image is one of the following:**
  - Boundary line (edge of an object) (>) with right hand of arrow denoting “solid” and left hand denoting “space”
  - Interior convex edge (+)
  - Interior concave edge (-)
Legal Junctions

- Only certain junctions are physically possible
- How can we formulate a CSP to label an image?
- **Variables:** edges
- **Domains:** $>, <, +, -$ 
- **Constraints:** legal junction types
Slight Problem: Local vs Global Consistency
Varieties of CSPs
Varieties of CSP Variables

- **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

- **Continuous variables**
  - E.g., start/end times for Hubble Telescope observations
  - Linear constraints solvable in polynomial time by linear program methods (see CSE 521 for a bit of LP theory)
Varieties of CSP 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

- **Preferences (soft constraints):**
  - E.g., red is better than green
  - Often representable by a cost for each variable assignment
  - Gives constrained optimization problems
  - (We’ll ignore these until we get to Bayes’ nets)
Solving CSPs
CSP as Search

- States
- Operators
- Initial State
- Goal State
Standard Depth First Search
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
Backtracking Search
### Backtracking Search

- **Backtracking search** is the basic uninformed algorithm for solving CSPs.

  - **Idea 1:** One variable at a time
    - Variable assignments are commutative, so fix ordering
    - I.e., [WA = red then NT = green] same as [NT = green then WA = red]
    - Only need to consider assignments to a single variable at each step

  - **Idea 2:** Check constraints as you go
    - I.e. consider only values which do not conflict previous assignments
    - Might have to do some computation to check the constraints
    - “Incremental goal test”

- Depth-first search with these two improvements is called **backtracking search**.

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

```plaintext
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
        end if
    end for
    return failure
```

- What are the choice points?

[Demo: coloring -- backtracking]
Backtracking Search

- Kind of depth first search
- Is it complete?
Improving Backtracking

- General-purpose ideas give huge gains in speed

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

- Filtering: Can we detect inevitable failure early?

- Structure: Can we exploit the problem structure?
Filtering: Keep track of domains for unassigned variables and cross off bad options

Forward checking: Cross off values that violate a constraint when added to the existing assignment

[Demo: coloring -- forward checking]
Filtering: Constraint Propagation

- Forward checking only propagates information from assigned to unassigned
- It doesn't catch when two unassigned variables have no consistent assignment:

  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** iff for *every* $x$ in the tail there is *some* $y$ in the head which could be assigned without violating a constraint.

- **Forward checking**: Enforcing consistency of arcs pointing to each new assignment.

Delete from the tail!
Arc Consistency of an **Entire CSP**

- A simple form of propagation makes sure **all** arcs are consistent:

  ![Map of Australia with states WA, NT, Q, NSW, V, SA]

- Important: If X loses a value, neighbors of X need to be rechecked!
- 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?

Remember: Delete from the tail!
AC-3 algorithm for Arc Consistency

**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** REMOVE-INCONSISTENT-VALUES\((X_i, X_j)\) **then**

**for each** \( X_k \) in Neighbors[\( X_j \)] **do**

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

**function** REMOVE-INCONSISTENT-VALUES\((X_i, X_j)\) **returns** true iff succeeds

removed \( \leftarrow \) 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 \( \leftarrow \) true

**return** removed

- Runtime: \( O(n^2d^3) \), can be reduced to \( O(n^2d^2) \)
- ... but detecting *all* possible future problems is NP-hard – why?

[Demo: CSP applet (made available by aispace.org) -- n-queens]
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)

- Even with Arc Consistency you still need backtracking search!
  - Could run at even step of that search
  - Usually better to run it once, before search

What went wrong here?
Video of Demo Coloring – Backtracking with Forward Checking – Complex Graph
Video of Demo Coloring – Backtracking with Arc Consistency – Complex Graph
K-Consistency
K-Consistency

- Increasing degrees of consistency
  - 1-Consistency (Node Consistency): Each single variable’s domain has a value which meets that variables unary constraints
  - 2-Consistency (Arc Consistency): For each pair of variables, any consistent assignment to one can be extended to the other
  - 3-Consistency (Path Consistency): For every set of 3 vars, any consistent assignment to 2 of the variables can be extended to the third var
  - K-Consistency: For each k nodes, any consistent assignment to k-1 can be extended to the k\textsuperscript{th} node.

- Higher k more expensive to compute

- (You need to know the algorithm for k=2 case: arc consistency)
Strong K-Consistency

- Strong k-consistency: also k-1, k-2, ... 1 consistent
- Claim: strong n-consistency means we can solve without backtracking!
- Why?
  - Choose any assignment to any variable
  - Choose a new variable
  - By 2-consistency, there is a choice consistent with the first
  - Choose a new variable
  - By 3-consistency, there is a choice consistent with the first 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
Ordering: Minimum Remaining Values

- Variable Ordering: Minimum remaining values (MRV):
  - Choose the variable with the fewest legal left values in its domain

- Why min rather than max?
- Also called “most constrained variable”
- “Fail-fast” ordering
Ordering: Maximum Degree

- **Tie-breaker among MRV variables**
  - What is the very first state to color? (All have 3 values remaining.)

- **Maximum degree heuristic:**
  - Choose the variable participating in the most constraints on remaining variables

- Why most rather than fewest constraints?
Ordering: Least Constraining Value

- **Value Ordering: Least Constraining Value**
  - Given a choice of variable, choose the *least constraining value*
  - i.e., 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 ordering ideas makes 1000 queens feasible
Rationale for MRV, MD, LCV

- We want to enter the most promising branch, but we also want to detect failure quickly

- **MRV+MD:**
  - Choose the variable that is most likely to cause failure
  - It must be assigned at some point, so if it is doomed to fail, better to find out soon

- **LCV:**
  - We hope our early value choices do not doom us to failure
  - Choose the value that is most likely to succeed
Pacman is trapped! He is surrounded by mysterious corridors, each of which leads to either a pit (P), a ghost (G), or an exit (E). In order to escape, he needs to figure out which corridors, if any, lead to an exit and freedom, rather than the certain doom of a pit or a ghost.

The one sign of what lies behind the corridors is the wind: a pit produces a strong breeze (S) and an exit produces a weak breeze (W), while a ghost doesn’t produce any breeze at all. Unfortunately, Pacman cannot measure the strength of the breeze at a specific corridor. Instead, he can stand between two adjacent corridors and feel the max of the two breezes. For example, if he stands between a pit and an exit he will sense a strong (S) breeze, while if he stands between an exit and a ghost, he will sense a weak (W) breeze. The measurements for all intersections are shown in the figure below.

Also, while the total number of exits might be zero, one, or more, Pacman knows that two neighboring squares will not both be exits.
Pacman is trapped! He is surrounded by mysterious corridors, each of which leads to either a pit (P), a ghost (G), or an exit (E). In order to escape, he needs to figure out which corridors, if any, lead to an exit and freedom, rather than the certain doom of a pit or a ghost.

The one sign of what lies behind the corridors is the wind: a pit produces a strong breeze (S) and an exit produces a weak breeze (W), while a ghost doesn’t produce any breeze at all. Unfortunately, Pacman cannot measure the strength of the breeze at a specific corridor. Instead, he can stand between two adjacent corridors and feel the max of the two breezes. For example, if he stands between a pit and an exit he will sense a strong (S) breeze, while if he stands between an exit and a ghost, he will sense a weak (W) breeze. The measurements for all intersections are shown in the figure below.

Also, while the total number of exits might be zero, one, or more, Pacman knows that two neighboring squares will not both be exits.
Trapped

- A pit produces a strong breeze (S) and an exit produces a weak breeze (W), while a ghost doesn’t produce any breeze at all.
- Pacman feels the max of the two breezes.
- The total number of exits might be zero, one, or more,
- Two neighboring squares will not both be exits.

Constraints?

Variables? X₁, … X₆
Domains {P, G, E}
Trapped

- A pit produces a strong breeze (S) and an exit produces a weak breeze (W), while a ghost doesn’t produce any breeze at all.
- Pacman feels the max of the two breezes.
- the total number of exits might be zero, one, or more,
- two neighboring squares will not both be exits.

Constraints?

\[
\begin{align*}
X_1 &= P \text{ or } X_2 = P & X_4 &= P \text{ or } X_5 = P \\
X_2 &= E \text{ or } X_3 = E & X_5 &= P \text{ or } X_6 = P \\
X_3 &= E \text{ or } X_4 = E & X_6 &= P \text{ or } X_1 = P
\end{align*}
\]

\[
X_i = E \text{ nand } X_{i+1} = E
\]

Also!

\[
X_2 \neq P \\
X_3 \neq P \\
X_4 \neq P
\]
Trapped

- A pit produces a strong breeze (S) and an exit produces a weak breeze (W), while a ghost doesn’t produce any breeze at all.
- Pacman feels the max of the two breezes.
- The total number of exits might be zero, one, or more,
- Two neighboring squares will not both be exits.

Constraints?

\[ X_1 = P \text{ or } X_2 = P \quad X_4 = P \text{ or } X_5 = P \]
\[ X_2 = E \text{ or } X_3 = E \quad X_5 = P \text{ or } X_6 = P \]
\[ X_3 = E \text{ or } X_4 = E \quad X_6 = P \text{ or } X_1 = P \]

\[ X_i = E \text{ nand } X_{i+1} = E \]

Also! \[ X_2 \neq P \]
\[ X_3 \neq P \]
\[ X_4 \neq P \]
Trapped

- A pit produces a strong breeze (S) and an exit produces a weak breeze (W), while a ghost doesn’t produce any breeze at all.
- Pacman feels the max of the two breezes.
- The total number of exits might be zero, one, or more,
- Two neighboring squares will not both be exits.

Constraints?

\[
\begin{align*}
X_1 &= P \text{ or } X_2 = P \quad & X_4 &= P \text{ or } X_5 = P \\
X_2 &= E \text{ or } X_3 = E \quad & X_5 &= P \text{ or } X_6 = P \\
X_3 &= E \text{ or } X_4 = E \quad & X_6 &= P \text{ or } X_1 = P \\
X_i = E & \text{ nand } X_{i+1} = E \\
\end{align*}
\]

Also! \( X_2 \neq P \)

MRV heuristic?

Arc consistent?

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>P</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>P</td>
<td>G</td>
<td>E</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>P</td>
<td>G</td>
<td>E</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>P</td>
<td>G</td>
<td>E</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>P</td>
<td>G</td>
<td>E</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>P</td>
<td>G</td>
<td>E</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Structure
Problem Structure

- Extreme case: independent subproblems
  - Example: Tasmania and mainland do not interact

- Independent subproblems are identifiable as connected components of constraint graph

- Suppose a graph of n variables can be broken into subproblems of only c variables:
  - Worst-case solution cost is $O((n/c)(d^c))$, linear in n
  - E.g., $n = 80$, $d = 2$, $c = 20$
  - $2^{80} = 4$ billion years at 10 million nodes/sec
  - $(4)(2^{20}) = 0.4$ seconds at 10 million nodes/sec
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 probabilistic reasoning (later): an example of the relation between syntactic restrictions and the complexity of reasoning
Tree-Structured CSPs

- Algorithm for tree-structured CSPs:
  - Order: Choose a root variable, order variables so that parents precede children
  - Remove backward: For $i = n : 2$, apply RemoveInconsistent(Parent($X_i$),$X_i$)
  - Assign forward: For $i = 1 : n$, assign $X_i$ consistently with Parent($X_i$)

- Runtime: $O(n \cdot d^2)$ (why?)
Claim 1: After backward pass, all root-to-leaf arcs are consistent
Proof: Each \( X \rightarrow Y \) was made consistent at one point and \( Y \)'s domain could not have been reduced thereafter (because \( Y \)'s children were processed before \( Y \))

Claim 2: If root-to-leaf arcs are consistent, forward assignment will not backtrack
Proof: Induction on position

Why doesn’t this algorithm work with cycles in the constraint graph?

Note: we’ll see this basic idea again with Bayes’ nets
Connection to Bayes Nets
Bayes Net Example: Alarm Network

<table>
<thead>
<tr>
<th>B</th>
<th>P(B)</th>
</tr>
</thead>
<tbody>
<tr>
<td>+b</td>
<td>0.001</td>
</tr>
<tr>
<td>-b</td>
<td>0.999</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>E</th>
<th>P(E)</th>
</tr>
</thead>
<tbody>
<tr>
<td>+e</td>
<td>0.002</td>
</tr>
<tr>
<td>-e</td>
<td>0.998</td>
</tr>
</tbody>
</table>

| A  | J    | P(J|A) |
|----|------|------|
| +a | +j   | 0.9  |
| +a | -j   | 0.1  |
| -a | +j   | 0.05 |
| -a | -j   | 0.95 |

| A  | M    | P(M|A) |
|----|------|-------|
| +a | +m   | 0.7   |
| +a | -m   | 0.3   |
| -a | +m   | 0.01  |
| -a | -m   | 0.99  |

| B  | E    | A    | P(A|B,E) |
|----|------|------|---------|
| +b | +e   | +a   | 0.95    |
| +b | +e   | -a   | 0.05    |
| +b | -e   | +a   | 0.94    |
| +b | -e   | -a   | 0.06    |
| -b | +e   | +a   | 0.29    |
| -b | +e   | -a   | 0.71    |
| -b | -e   | +a   | 0.001   |
| -b | -e   | -a   | 0.999   |
More Complex Bayes’ Net: Auto Diagnosis
An HMM is defined by:

- Initial distribution: $P(X_1)$
- Transitions: $P(X_t | X_{t-1})$
- Emissions: $P(E | X)$
Forward Algorithm

\[ B'(x_1=r) = 0.7 \]
\[ B'(x_2=r) = P(x_2=r \mid x_1=r) \times 0.875 + P(x_2=r \mid x_1=s) \times 0.125 \]
\[ = 0.8 \times 0.875 + 0.6 \times 0.125 \]
\[ = 0.775 \]

\[ B(x_0=r) = 0.5 \]
\[ B(x_1=r) = 0.875 \]
\[ B(x_1=s) \propto 0.3 \times 0.225 = 0.0675 \]
\[ \text{Divide by 0.765 to normalize} \]
\[ B(x_1=r) = 0.912 \]

\[ B'(X_{t+1}) = \sum_{x_t} P(X'_{t+1} \mid X_t) B(x_t) \]
\[ B(X_{t+1}) \propto x_{t+1} \]
\[ P(e_{t+1} \mid X_{t+1}) B'(X_{t+1}) \]

<table>
<thead>
<tr>
<th>( R_{t+1} )</th>
<th>( P(R_{t+1} \mid R_t) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.8</td>
<td>0.8</td>
</tr>
<tr>
<td>0.6</td>
<td>0.6</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>( R_t )</th>
<th>( P(U_t \mid R_t) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.9</td>
<td>0.9</td>
</tr>
<tr>
<td>0.3</td>
<td>0.3</td>
</tr>
</tbody>
</table>
More Complex HMM Inference

- **Forward Backward**
  - Computes marginal probabilities of *all* hidden states given sequence of observations
More Complex HMM Inference

- **Forward Backward**
  - Computes marginal *probabilities* of all hidden states given sequence of observations

- **Viterbi**
  - Computes most likely *sequence of states*
Improving Structure
Nearly Tree-Structured CSPs

- Conditioning: instantiate a variable, prune its neighbors' domains
- Cutset conditioning: instantiate (in all ways) a set of variables such that the remaining constraint graph is a tree
- Cutset size c gives runtime $O( (d^c) (n-c) d^2 )$, very fast for small c
Cutset Conditioning

- Choose a cutset
- Instantiate the cutset (all possible ways)
- Compute residual CSP for each assignment
- Solve the residual CSPs (tree structured)
Cutset Quiz

- Find the smallest cutset for the graph below.
Local Search for CSPs
Iterative Algorithms for CSPs

- Local search methods typically work with “complete” states, i.e., all variables assigned

- To apply to CSPs:
  - Take an assignment with unsatisfied constraints
  - Operators *reassign* variable values
  - No fringe! Live on the edge.

- Algorithm: While not solved,
  - Variable selection: randomly select any conflicted variable
  - Value selection: min-conflicts heuristic:
    - Choose a value that violates the fewest constraints
    - I.e., hill climb with \( h(n) = \text{total number of violated constraints} \)
Example: 4-Queens

- States: 4 queens in 4 columns ($4^4 = 256$ states)
- Operators: move queen in column
- Goal test: no attacks
- Evaluation: $c(n) =$ number of attacks

[Demo: n-queens – iterative improvement (L5D1)]
[Demo: coloring – iterative improvement]
Performance of Min-Conflicts

- Given random initial state, can solve n-queens in almost constant time for arbitrary $n$ with high probability (e.g., $n = 10,000,000$)!

- The same appears to be true for any *randomly-generated* CSP *except* in a narrow range of the ratio

$$R = \frac{\text{number of constraints}}{\text{number of variables}}$$
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 (cutset conditioning)

- Iterative min-conflicts is often effective in practice