Recap: Search

- Search problem:
  - States (configurations of the world)
  - Successor function: a function from states to lists of (state, action, cost) triples; drawn as a graph
  - Start state and goal test

- Search tree:
  - Nodes: represent plans for reaching states
  - Plans have costs (sum of action costs)

- Search Algorithm:
  - Systematically builds a search tree
  - Chooses an ordering of the fringe (unexplored nodes)
What is Search For?

- Models of the world: single agents, deterministic actions, fully observed state, discrete state space
- Planning: sequences of actions
  - The path to the goal is the important thing
  - Paths have various costs, depths
  - Heuristics to guide, fringe to keep backups
- Identification: assignments to variables
  - The goal itself is important, not the path
  - All paths at the same depth (for some formulations)
  - CSPs are specialized for identification problems

Constraint Satisfaction Problems

- Standard search problems:
  - State is a "black box": arbitrary data structure
  - Goal test: any function over states
  - Successor function can 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
  - Simple example of a formal representation language
- Allows useful general-purpose algorithms with more power than standard search algorithms
Example CSP: Map-Coloring

- Variables: \( WA, NT, Q, NSW, V, SA, T \)
- Domain: \( D = \{ \text{red, green, blue} \} \)
- Constraints: adjacent regions must have different colors
  \[ WA \neq NT \]
  \[ (WA, NT) \in \{ (\text{red, green}), (\text{red, blue}), (\text{green, red}), \ldots \} \]
- Solutions are assignments satisfying all constraints, e.g.:
  \[ \{ WA = \text{red}, NT = \text{green}, Q = \text{red}, NSW = \text{green}, V = \text{red}, SA = \text{blue}, T = \text{green} \} \]

Example CSP: N-Queens

- Formulation 1:
  - Variables: \( X_{ij} \)
  - Domains: \( \{0, 1\} \)
  - Constraints
    \[ \forall i, j, k \ (X_{ij}, X_{ik}) \in \{(0, 0), (0, 1), (1, 0)\} \]
    \[ \forall i, j, k \ (X_{ij}, X_{kj}) \in \{(0, 0), (0, 1), (1, 0)\} \]
    \[ \forall i, j, k \ (X_{ij}, X_{i+k,j+k}) \in \{(0, 0), (0, 1), (1, 0)\} \]
    \[ \forall i, j, k \ (X_{ij}, X_{i+k,j-k}) \in \{(0, 0), (0, 1), (1, 0)\} \]
    \[ \sum_{i,j} X_{ij} = N \]
Example CSP: N-Queens

- Formulation 2:
  - Variables: \( Q_k \)
  - Domains: \{1, 2, 3, \ldots N\}

- Constraints:
  Implicit: \( \forall i, j \) non-threatening\( (Q_i, Q_j) \)
  -or-
  Explicit: \( (Q_1, Q_2) \in \{(1, 3), (1, 4), \ldots\} \)

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 CSP: Cryptarithmetic**

- **Variables (circles):**
  
  \[ F \ T \ U \ W \ R \ O \ X_1 \ X_2 \ X_3 \]

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

- **Constraints (boxes):**
  
  \[ \text{alldiff}(F, T, U, W, R, O) \]
  
  \[ O + O = R + 10 \cdot X_1 \]
  
  \[ \ldots \]

**Example CSP: 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
Example CSP: The Waltz Algorithm

- The Waltz algorithm is for interpreting line drawings of solid polyhedra
- An early example of a computation posed as a CSP

- Look at all intersections
- Adjacent intersections impose constraints on each other

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

- **Continuous variables**
  - E.g., start-end state of a robot
  - Linear constraints solvable in polynomial time by LP methods (see cs170 for a bit of this theory)
Varieties of Constraints

- Varieties of Constraints
  - Unary constraints involve a single variable (equiv. to shrinking domains):
    \[ SA \neq \text{green} \]
  - Binary constraints involve pairs of variables:
    \[ SA \neq WA \]
  - 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)

Real-World 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
- Floorplanning
- Fault diagnosis
- … lots more!

- Many real-world problems involve real-valued variables…
Standard Search Formulation

- Standard search formulation of CSPs (incremental)
- Let's start with the straightforward, dumb approach, then fix it
- States are defined by the values assigned so far
  - 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
- Simplest CSP ever: two bits, constrained to be equal

Search Methods

- What does BFS do?
- What does DFS do?
  - [demo]
- What’s the obvious problem here?
- What’s the slightly-less-obvious problem?
Backtracking Search

- Idea 1: Only consider a single variable at each point
  - 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
  - How many leaves are there?

- Idea 2: Only allow legal assignments at each point
  - i.e. consider only values which do not conflict previous assignments
  - Might have to do some computation to figure out whether a value is ok
  - “Incremental goal test”

Depth-first search for CSPs with these two improvements is called backtracking search (useless name, really)
- [DEMO]

Backtracking search is the basic uninformed algorithm for CSPs
- Can solve n-queens for n ≈ 25

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 + var-ordering + fail-on-violation
- What are the choice points?
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?

Minimum Remaining Values

- Minimum remaining values (MRV):
  - Choose the variable with the fewest legal values

- Why min rather than max?
- Also called “most constrained variable”
- Also called “fail-fast” ordering
Degree Heuristic

- Tie-breaker among MRV variables
- Degree heuristic:
  - Choose the variable participating in the most constraints on remaining variables

- Why most rather than fewest constraints?

Least Constraining Value

- Given a choice of variable:
  - 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!

- Why least rather than most?

- Combining these heuristics makes 1000 queens feasible
Filtering: Forward Checking

- Idea: Keep track of remaining legal values for unassigned variables (using immediate constraints)
- Idea: Terminate when any variable has no legal values

Filtering: Forward Checking

- Forward checking propagates information from assigned to adjacent unassigned variables, but doesn’t detect more distant failures:
  - NT and SA cannot both be blue!
  - Why didn’t we detect this yet?
  - Constraint propagation repeatedly enforces constraints (locally)
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.

What happens?

- Forward checking = Enforcing consistency of each arc pointing to the new assignment

Arc Consistency of a CSP

- Simplest form of propagation makes each arc consistent
  - $X \rightarrow Y$ is consistent iff for every value $x$ there is some allowed $y$

- If $X$ loses a value, neighbors of $X$ need to be rechecked!
- Arc consistency detects failure earlier than forward checking
- What’s the downside of arc consistency?
- Can be run as a preprocessor or after each assignment
Establishing 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
  $(X_i, X_j) \leftarrow$ 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 $\leftarrow$ false
for each $x$ in Domain($X_i$) do
  if no value $y$ in Domain($X_j$) allows $(x,y)$ to satisfy the constraint $X_i \rightarrow X_j$
    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?

Limitations of Arc Consistency

- After running arc consistency:
  - Can have one solution left
  - Can have multiple solutions left
  - Can have no solutions left (and not know it)
K-Consistency

- Increasing degrees of consistency
  - 1-Consistency (Node Consistency): Each single node’s domain has a value which meets that node’s unary constraints
  - 2-Consistency (Arc Consistency): For each pair of nodes, any consistent assignment to one can be extended to the other
  - K-Consistency: For each k nodes, any consistent assignment to k-1 can be extended to the kth node.

- Higher k more expensive to compute
- (You need to know the k=2 algorithm)

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
  - ...
- Lots of middle ground between arc consistency and n-consistency! (e.g. path consistency)
Problem Structure

- Tasmania and mainland are independent subproblems
- Identifiable as connected components of constraint graph
- Suppose each subproblem has c variables out of n total
  - Worst-case solution cost is $O\left(\frac{n}{c}\right)^d$, linear in n
  - E.g., $n = 80$, $d = 2$, $c = 20$
  - $2^{20} = 4$ billion years at 10 million nodes/sec
  - $(4)(2^{20}) = 0.4$ seconds at 10 million nodes/sec

Backtracking with MRV, Degree, LCV, Filtering

```python
function RecursiveBacktracking(pa, fd, vars, constraints)
    if IsComplete(pa) then return pa
    next_var <= select_MRV_Degree(pa, fd, vars, constraints)
    for each value in fd[next_var] do
        new_fd[value] <= constraint_prop(pa, fd, vars, constraints)
        for each value in fd[next_var] in order of LCV do
            if any of the domains in new_fd[value] is empty
                continue;
            else // all domains in new_fd[value] have at least one value remaining
                add (var=value) to pa
                result <= recursive_backtracking(pa, new_fd[value], vars, constraints)
            if (result not equal to failure) then return result
    //if we get here none of the expansions led to a solution
    return failure

• select_MRV_degree: selects an unassigned variable based on MRV an degree heuristic
• constraint_prop: performs constraint propagation, this could be through forward propagation or through arc consistency
• pc: partial assignment
• fd: filtered domains
```
Tree-Structured CSPs

- Theorem: if the constraint graph has no loops, the CSP can be solved in $O(n d^2)$ time
  - Compare to general CSPs, where worst-case time is $O(d^n)$

- This property also applies to probabilistic reasoning (later): an important example of the relation between syntactic restrictions and the complexity of reasoning.

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

- For $i = n : 2$, apply RemoveInconsistent(Parent($X_i$),$X_i$)
- For $i = 1 : n$, assign $X_i$ consistently with Parent($X_i$)
- Runtime: $O(n d^2)$ (why?)
Tree-Structured CSPs

- Why does this work?
  - Claim: After processing the right k nodes, given any satisfying assignment to the rest, the right k can be assigned (left to right) without backtracking.
  - Proof: Induction on position

- Why doesn’t this algorithm work with loops?
- Note: we’ll see this basic idea again with Bayes’ nets

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
Tree Decompositions*

- Create a tree-structured graph of overlapping subproblems, each is a mega-variable
- Solve each subproblem to enforce local constraints
- Solve the CSP over subproblem mega-variables using our efficient tree-structured CSP algorithm

CSPs: our status

- CSPs are a special kind of search problem:
  - States defined by values of a fixed set of variables
  - Goal test defined by constraints on variable values
- Backtracking = depth-first search with
  - Branching on only one variable per layer in search tree
  - Incremental constraint checks ("Fail fast")
- Heuristics at our points of choice to improve running time:
  - Ordering variables: Minimum Remaining Values and Degree Heuristic
  - Ordering of values: Least Constraining Value
  - Filtering: forward checking, arc consistency \(\rightarrow\) enable computation of these heuristics
- Structure: Disconnected and tree-structured CSPs are efficient
- Iterative improvement
Iterative Algorithms for CSPs

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

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

- Variable selection: randomly select any conflicted variable

- Value selection by min-conflicts heuristic:
  - Choose 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, i.e., no two queens on same row, same column or same diagonal
- Evaluation: $c(n) = \text{number of attacks}$
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

\[ R = \frac{\text{number of constraints}}{\text{number of variables}} \]

Hill Climbing

- Simple, general idea:
  - Start wherever
  - Always choose the best neighbor
  - If no neighbors have better scores than current, quit

- Why can this be a terrible idea?
  - Complete?
  - Optimal?

- What’s good about it?
Hill Climbing Diagram

- Random restarts?
- Random sideways steps?

Simulated Annealing*

- Idea: Escape local maxima by allowing downhill moves
  - But make them rarer as time goes on

```
function SIMULATED-ANNEALING (problem, schedule) returns a solution state
inputs: problem, a problem
        schedule, a mapping from time to "temperature"
local variables: current, a node
                next, a node
                $T$, a "temperature" controlling prob. of downward steps

current ← MAKE-NODE(INITIAL-STATE(problem))
for $t ← 1$ to $\infty$ do
    $T ← $ schedule[$t$]
    if $T = 0$ then return current
    next ← a randomly selected successor of current
    $\Delta E ← $ VALUE[next] − VALUE[current]
    if $\Delta E > 0$ then current ← next
    else current ← next only with probability $\exp(-\Delta E/T)$
```
Simulated Annealing*

- **Theoretical guarantee:**
  - Stationary distribution: \( p(x) \propto e^{\frac{E(x)}{kT}} \)
  - If \( T \) decreased slowly enough, will converge to optimal state!

- **Is this an interesting guarantee?**

- **Sounds like magic, but reality is reality:**
  - The more downhill steps you need to escape, the less likely you are to ever make them all in a row
  - People think hard about *ridge operators* which let you jump around the space in better ways

---

Recap CSPs

- CSPs are a special kind of search problem:
  - States defined by values of a fixed set of variables
  - Goal test defined by constraints on variable values

- Backtracking = depth-first search (why?, tree or graph search?) with
  - Branching on only one variable per layer in search tree
  - Incremental constraint checks (“Fail fast”)

- **Heuristics at our points of choice to improve running time:**
  - Ordering variables: Minimum Remaining Values and Degree Heuristic
  - Ordering of values: Least Constraining Value
  - Filtering: forward checking, arc consistency \( \rightarrow \) computation of heuristics

- **Structure: Disconnected and tree-structured CSPs are efficient**
  - Non-tree-structured CSP can become tree-structured after some variables have been assigned values

- **Iterative improvement: min-conflicts is usually effective in practice**