Today

- Efficient Solution of CSPs
- Local Search
Reminder: CSPs

- **CSPs:**
  - Variables
  - Domains
  - Constraints
    - Implicit (provide code to compute)
    - Explicit (provide a subset of the possible tuples)
    - Unary / Binary / N-ary

- **Goals:**
  - Usually: find any solution
  - Find all, find best, etc

---

Backtracking Search

```python
function BacktrackingSearch(esp) returns solution/failure
return Recursive-Backtracking(∅, esp)

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

- General-purpose ideas give huge gains in speed
  - … but it’s all still NP-hard

- Ordering (last class):
  - 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: 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: Constraint Propagation

- 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 propagates from constraint to constraint

Consistency of An 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

- What happens?
- Forward checking = Enforcing consistency of each arc pointing to the new assignment
Arc Consistency of a CSP

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

- If X loses a value, (incoming) neighbors of X need to be rechecked!
- Arc consistency detects failure earlier than forward checking
- What's the downside of enforcing arc consistency?
- Can be run as a preprocessor or after each assignment

Establishing Arc Consistency

```python
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 \leftrightarrow X_j
  then delete x from DOMAIN[X_i]; removed — 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]
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 $O((n/c)(d^c))$, 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
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

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 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

**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 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 one legal variable assigned per node
- Variable ordering and value selection heuristics help significantly
- Forward checking prevents assignments that guarantee later failure
- Constraint propagation (e.g., enforcing arc consistency) does additional work to constrain values and detect inconsistencies
- Constraint graphs allow for analysis of problem structure
- Tree-structured CSPs can be solved in linear time
- Iterative min-conflicts is usually effective in practice