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 Constraints
- Binary Constraints
- N-ary Constraints

Backtracking Search

- 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?

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

Arc Consistency of a CSP

- A simple form of propagation makes sure all arcs are consistent:
- 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 enforcing arc consistency?
- Can be run as a preprocessor or after each assignment

Establishing Arc Consistency

- Function \( AC(X, \text{copy}) \) returns the CSP, possibly with reduced domains
- If \( X \rightarrow Y \) is inconsistent, remove \( X \) from the CSP

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 \( k^{th} \) 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., \( n = 80, d = 2, c = 20 \)
  - \( 2^{20} = 4 \) billion years at 10 million nodes/sec
- \( 4 \times 2^{20} = 0.4 \) seconds at 10 million nodes/sec

**Tree-Structured CSPs**

- 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?)

**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 \( O((\phi) (n-c) d^c) \), 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

Example: 4-Queens

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

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

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

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

Local Search Methods

- Tree search keeps unexplored alternatives on the fringe (ensures completeness)
- Local search: improve what you have until you can’t make it better
- Generally much faster and more memory efficient (but incomplete)
Types of Search Problems

- Planning problems:
  - We want a path to a solution (examples?)
  - Usually want an optimal path
  - Incremental formulations

- Identification problems:
  - We actually just want to know what the goal is (examples?)
  - Usually want an optimal goal
  - Complete-state formulations
  - Iterative improvement algorithms

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

Theoretical guarantee:
- Stationary distribution: \( p(x) \propto e^{-E(x)/T} \)
- 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

Genetic Algorithms

- Genetic algorithms use a natural selection metaphor
- Like beam search (selection), but also have pairwise crossover operators, with optional mutation
- Probably the most misunderstood, misapplied (and even maligned) technique around!
Example: N-Queens

- Why does crossover make sense here?
- When wouldn’t it make sense?
- What would mutation be?
- What would a good fitness function be?