

CS 188: Artificial Intelligence Spring 2010

Lecture 4: A* wrap-up + Constraint Satisfaction 1/28/2010

Pieter Abbeel – UC Berkeley
Many slides from Dan Klein

Announcements

- Project 0 (Python tutorial) is due today ~~23:59~~
- If you don't have a class account yet, pick one up after lecture
- Written 1 (Search) is due today
~~drop off~~ 283 = soda
- Project 1 (Search) is out and due next week Thursday

→ Section/Lecture

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

General Tree Search

```
function TREE-SEARCH(problem, strategy) returns a solution, or failure
  initialize the search tree using the initial state of problem
  loop do
    if there are no candidates for expansion then return failure
    choose a leaf node for expansion according to strategy
    if the node contains a goal state then return the corresponding solution
    else expand the node and add the resulting nodes to the search tree
  end
```

Important ideas:

- Fringe
- Expansion
- Exploration strategy

*Detailed pseudocode
is in the book!*

- Main question: which fringe nodes to explore?

A* Review

- A* uses both backward costs g and forward estimate h : $f(n) = \underline{g(n)} + \underline{h(n)}$
- A* tree search is optimal with admissible heuristics (optimistic future cost estimates)
- Heuristic design is key: relaxed problems can help

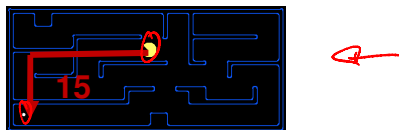
Admissible Heuristics

- A heuristic h is *admissible* (optimistic) if:

$$\underline{h(n) \leq h^*(n)}$$

where $h^*(n)$ is the true cost to a nearest goal

- Example:

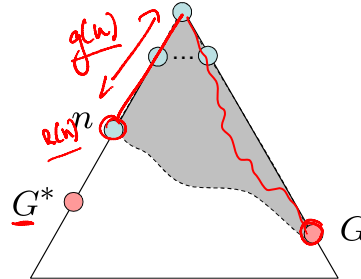


- Coming up with admissible heuristics is most of what's involved in using A* in practice.

Optimality of A*: Blocking

Notation:

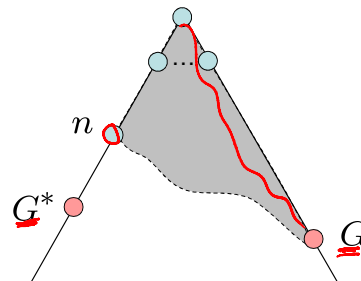
- $g(n)$ = cost to node n
- $h(n)$ = estimated cost from n to the nearest goal (heuristic)
- $f(n) = g(n) + h(n)$ = estimated total cost via n
- G^* : a lowest cost goal node
- G : another goal node



Optimality of A*: Blocking

Proof:

- What could go wrong?
- We'd have to have to pop a suboptimal goal G off the fringe before G^*
- This can't happen:
 - Imagine a suboptimal goal G is on the queue
 - Some node n which is a subpath of G^* must also be on the fringe (why?)
 - n will be popped before G

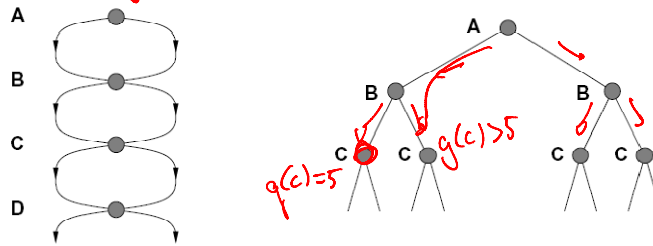


$$\begin{aligned}
 f(n) &= g(n) + h(n) \\
 g(n) + h(n) &\leq g(G^*) \\
 g(G^*) &< g(G) \\
 f(G) = g(G) &= f(n) \leq g(G^*)
 \end{aligned}$$

Tree Search: Extra Work!

- Failure to detect repeated states can cause exponentially more work. Why?

search graph



Graph Search

- Very simple fix: never expand a state twice

function GRAPH-SEARCH(*problem*, *fringe*) **returns** a solution, or failure

closed ← an empty set

→ *fringe* ← INSERT(MAKE-NODE(INITIAL-STATE[*problem*]), *fringe*)

loop do

→ **if** *fringe* is empty **then return** failure

node ← REMOVE-FRONT(*fringe*)

if GOAL-TEST(*problem*, STATE[*node*]) **then return** *node*

if STATE[*node*] is not in *closed* **then**

add STATE[*node*] to *closed*

fringe ← INSERTALL(EXPAND(*node*, *problem*), *fringe*)

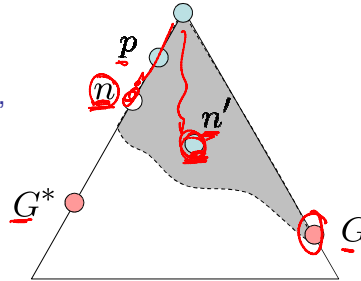
end

- Can this wreck completeness? Optimality?

Optimality of A* Graph Search

Proof:

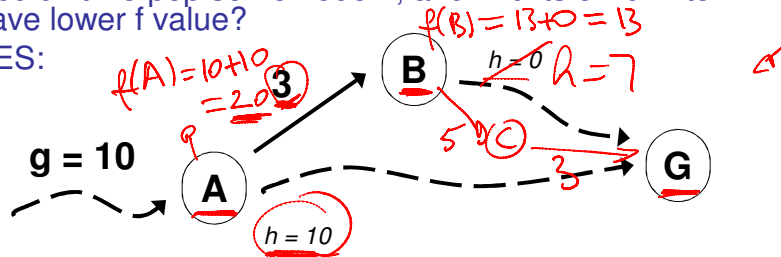
- New possible problem: nodes on path to G^* that would have been in queue aren't, because some worse n' for the same state as some n was dequeued and expanded first (disaster!)
- Take the highest such n in tree
- Let p be the ancestor which was on the queue when n' was expanded



- Assume $f(p) < f(n)$
 - $f(n) < f(n')$ because n' is suboptimal
 - p would have been expanded before n'
 - Contradiction!
- $f(p) = g(p) + h(p)$
 $f(p) < f(n) < f(n')$
 $\forall p \in \text{ancestors}(n): f(p) < f(n)$

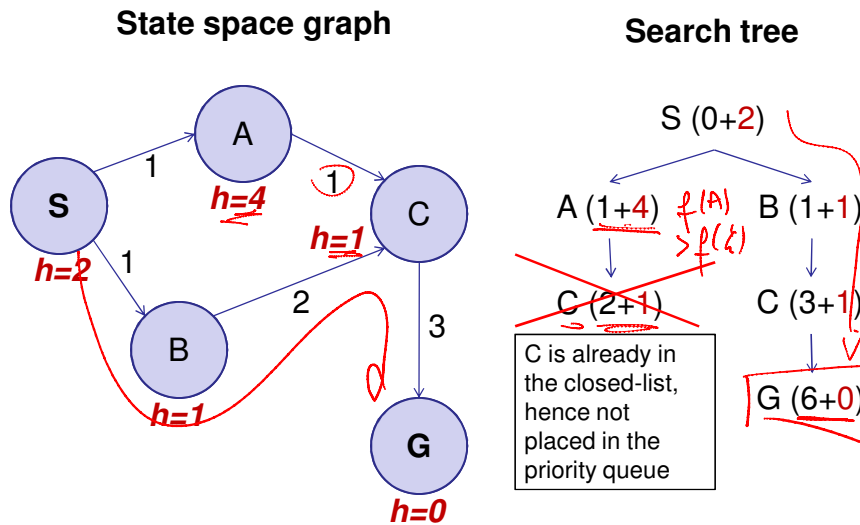
Consistency

- Wait, how do we know parents have better f -values than their successors?
- Couldn't we pop some node n , and find its child n' to have lower f value?
- YES:

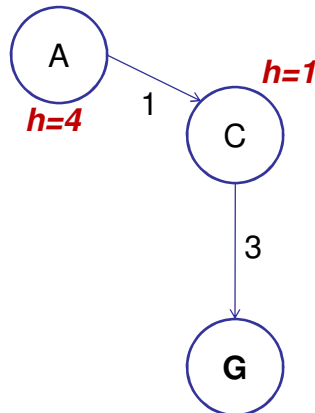


- What can we require to prevent these inversions?
- Consistency: $c(n, a, n') \geq h(n) - h(n')$
- Real cost must always exceed reduction in heuristic

A* Graph Search Gone Wrong



Consistency



The story on Consistency:

- Definition: $\text{cost}(A \text{ to } C) + h(C) \geq h(A)$ *check this one*
 - Consequence in search tree: Two nodes along a path: N_A, N_C
 $g(N_C) = g(N_A) + \text{cost}(A \text{ to } C)$
 $g(N_C) + h(C) \geq g(N_A) + h(A)$
 - The f value along a path never decreases
 - Non-decreasing f means you're optimal to every state (not just goals)
- $f(N_C) \geq f(N_A)$*

Optimality Summary

- **Tree search:**
 - A* optimal if heuristic is admissible (and non-negative)
 - Uniform Cost Search is a special case ($h = 0$)
- **Graph search:**
 - A* optimal if heuristic is consistent ↻
 - UCS optimal ($h = 0$ is consistent)
- **Consistency implies admissibility**
 - Challenge: Try to prove this. ↻
 - Hint: try to prove the equivalent statement not admissible implies not consistent ↻
- In general, natural admissible heuristics tend to be consistent ↻
- Remember, costs are always positive in search! ↻

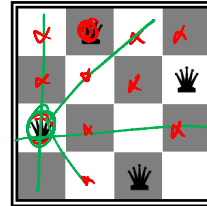
What is Search For?

- **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

19

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

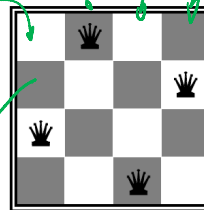


20

Example: N-Queens

Formulation 1:

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



- row $\rightarrow \forall i, j, k (X_{ij}, X_{ik}) \in \{(0, 0), (0, 1), (1, 0)\}$
- col $\rightarrow \forall i, j, k (X_{ij}, X_{kj}) \in \{(0, 0), (0, 1), (1, 0)\}$
- inc diag $\rightarrow \forall i, j, k (X_{ij}, X_{i+k, j+k}) \in \{(0, 0), (0, 1), (1, 0)\}$
- dec diag $\rightarrow \forall i, j, k (X_{ij}, X_{i+k, j-k}) \in \{(0, 0), (0, 1), (1, 0)\}$

$$\sum_{i,j} X_{ij} = N$$

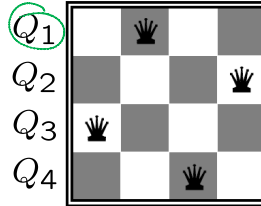
21

Example: N-Queens

- Formulation 2:

- Variables: Q_k

- Domains: $\{1, 2, 3, \dots, N\}$



- Constraints:

Implicit: $\forall i, j$ non-threatening(Q_i, Q_j)

-or-

Explicit: $(Q_1, Q_2) \in \{(1, 3), (1, 4), \dots\}$

...

Example: Map-Coloring

- Variables: WA, NT, Q, NSW, V, SA, T

- Domain: $D = \{red, green, blue\}$

- Constraints: adjacent regions must have different colors

$$WA \neq NT$$

$$(WA, NT) \in \{(red, green), (red, blue), (green, red), \dots\}$$



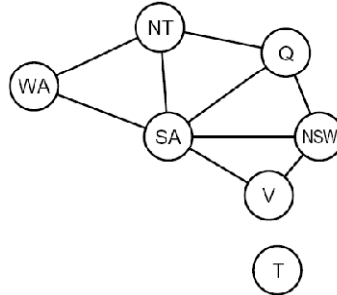
- Solutions are assignments satisfying all constraints, e.g.:

$$\{WA = red, NT = green, Q = red, NSW = green, V = red, SA = blue, T = green\}$$

24

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!

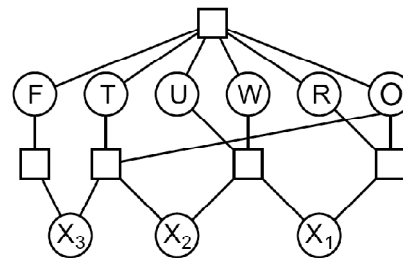


25

Example: Cryptarithmic

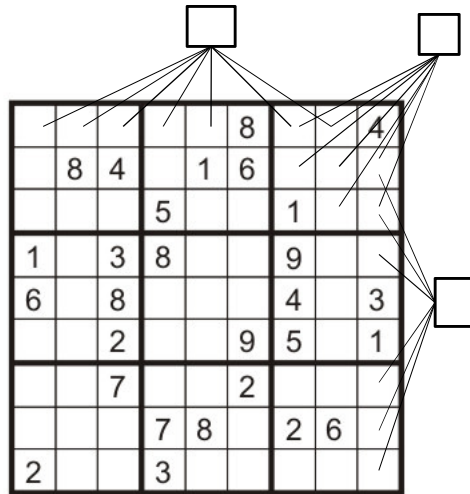
- Variables (circles):
 $F T U W R O X_1 X_2 X_3$
- Domains:
 $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$
- Constraints (boxes):
 $\text{alldiff}(F, T, U, W, R, O)$
 $O + O = R + 10 \cdot X_1$
...

$$\begin{array}{r} T W O \\ + T W O \\ \hline F O U R \end{array}$$



26

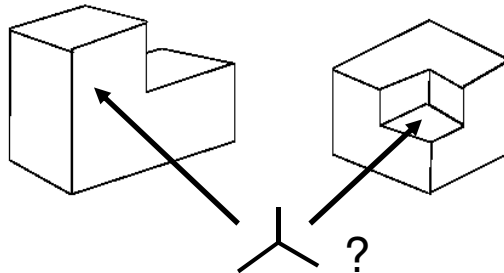
Example: Sudoku



- 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

Example: 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

28

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)

32

Varieties of Constraints

- **Varieties of Constraints**
 - Unary constraints involve a single variable (equiv. to shrinking domains):
$$SA \neq green$$
 - Binary constraints involve pairs of variables:
$$SA \neq WA$$
 - Higher-order constraints involve 3 or more variables:
e.g., cryptarithmic 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)

33

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...

34

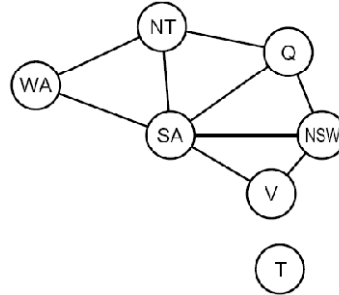
Standard Search Formulation

- 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

35

Search Methods

- What does BFS do?



- What does DFS do?
- What's the obvious problem here?
- What's the slightly-less-obvious problem?

37

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)
- Backtracking search is the basic uninformed algorithm for CSPs
- Can solve n-queens for $n \approx 25$

39

Backtracking Search

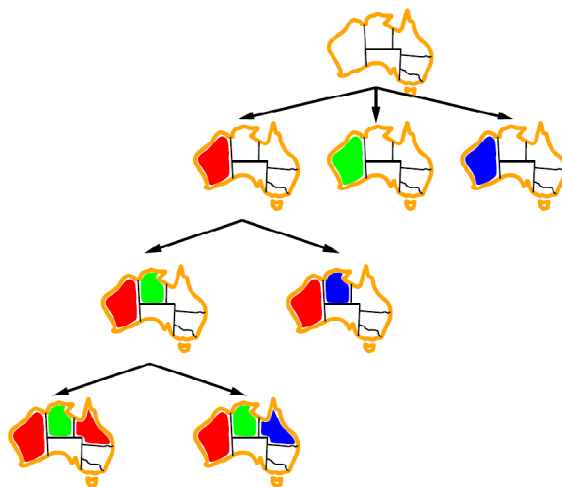
```
function BACKTRACKING-SEARCH(csp) returns solution/failure
  return RECURSIVE-BACKTRACKING({}, csp)

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
```

- What are the choice points?

41

Backtracking Example



42

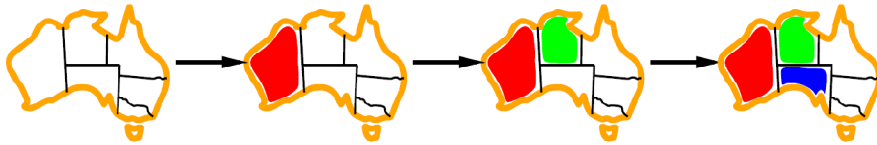
Improving Backtracking

- General-purpose ideas can give huge gains in speed:
 - Which variable should be assigned next?
 - In what order should its values be tried?
 - Can we detect inevitable failure early?
 - Can we take advantage of problem structure?

43

Minimum Remaining Values

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

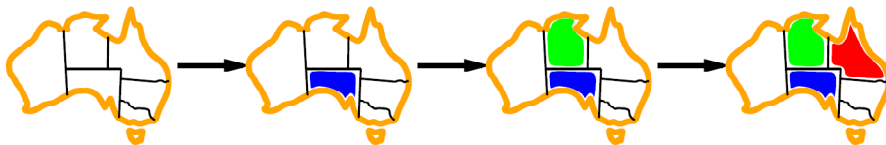


- Why min rather than max?
- Also called “most constrained variable”
- “Fail-fast” ordering

45

Degree Heuristic

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

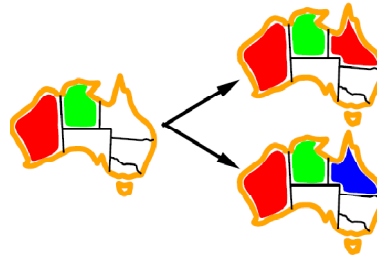


- Why most rather than fewest constraints?

46

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



47

Forward Checking



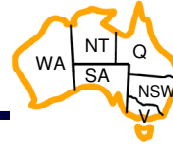
- Idea: Keep track of remaining legal values for unassigned variables (using immediate constraints)
- Idea: Terminate when any variable has no legal values



48

[demo: forward checking animation]

Constraint Propagation



- Forward checking propagates information from assigned to adjacent unassigned variables, but doesn't detect more distant failures:



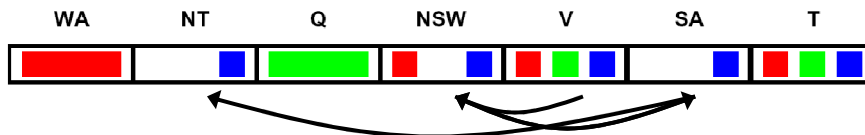
- NT and SA cannot both be blue!
- Why didn't we detect this yet?
- Constraint propagation repeatedly enforces constraints (locally)

49

Arc Consistency



- 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

50

Arc Consistency

```

function AC-3(csp) returns the CSP, possibly with reduced domains
inputs: csp, a binary CSP with variables  $\{X_1, X_2, \dots, X_n\}$ 
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  $X_i \leftrightarrow X_j$ 
      then delete  $x$  from DOMAIN[ $X_i$ ]; removed  $\leftarrow$  true
  return removed

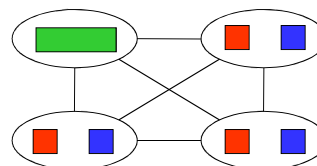
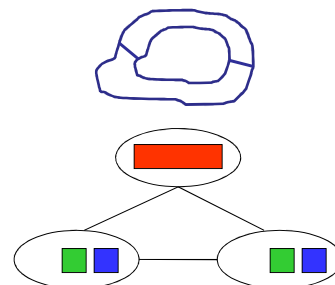
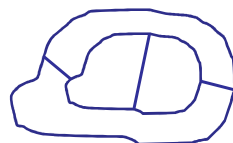
```

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

51
[demo: arc consistency animation]

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)

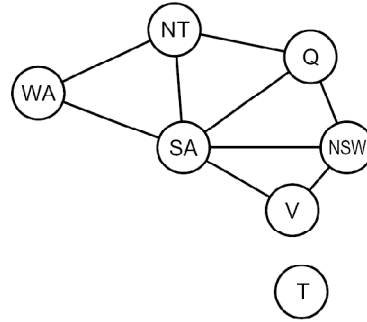


Demo: Backtracking + AC



Problem Structure

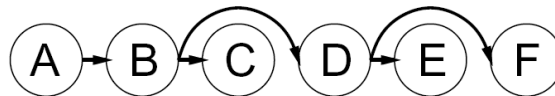
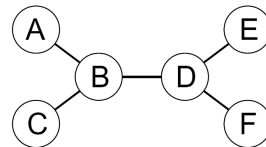
- Tasmania and mainland are independent subproblems
- Identifiable as connected components of constraint graph
- Suppose each subproblem has c variables out of n total
- Worst-case solution cost is $O((n/c)(d^c))$, linear in n
 - E.g., $n = 80$, $d = 2$, $c = 20$
 - $2^{80} = 4$ billion years at 10 million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



55

Tree-Structured CSPs

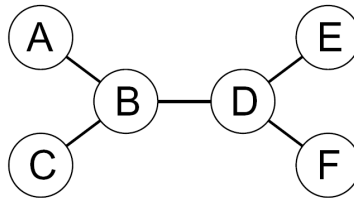
- Choose a variable as root, order variables from root to leaves such that every node's parent precedes it in the ordering



- For $i = n : 2$, apply $\text{RemoveInconsistent}(\text{Parent}(X_i), X_i)$
- For $i = 1 : n$, assign X_i consistently with $\text{Parent}(X_i)$
- Runtime: $O(n d^2)$

56

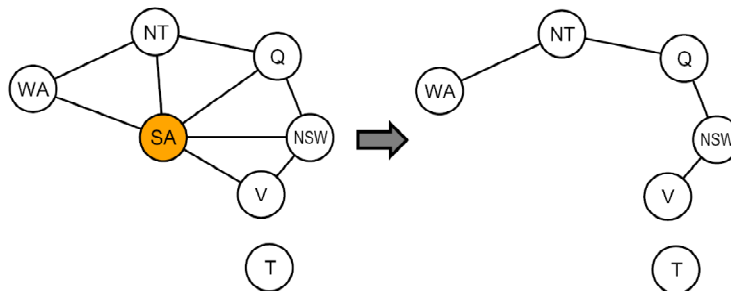
Tree-Structured CSPs



- 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 logical and probabilistic reasoning: an important example of the relation between syntactic restrictions and the complexity of reasoning.

57

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

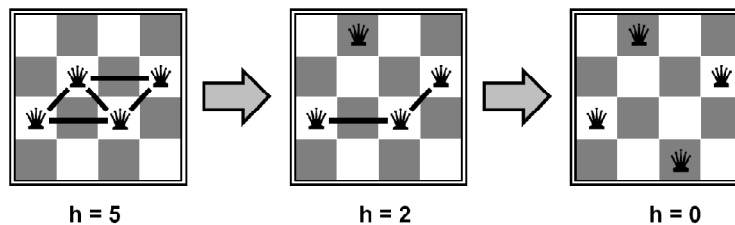
58

Iterative Algorithms for CSPs

- Greedy and local methods typically work with “complete” states, i.e., all variables assigned
- To apply to CSPs:
 - Allow states with unsatisfied constraints
 - Operators *reassign* variable values
- 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 $h(n)$ = total number of violated constraints

59

Example: 4-Queens



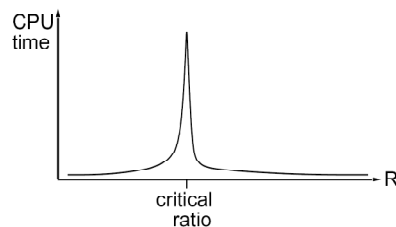
- States: 4 queens in 4 columns ($4^4 = 256$ states)
- Operators: move queen in column
- Goal test: no attacks
- Evaluation: $h(n)$ = number of attacks

60

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

$$R = \frac{\text{number of constraints}}{\text{number of variables}}$$



61

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., arc consistency) does additional work to constrain values and detect inconsistencies
- The constraint graph representation allows analysis of problem structure
- Tree-structured CSPs can be solved in linear time
- Iterative min-conflicts is usually effective in practice

62

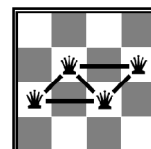
Local Search Methods

- Queue-based algorithms keep fallback options (backtracking)
- Local search: improve what you have until you can't make it better
- Generally much more efficient (but incomplete)

63

Types of 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



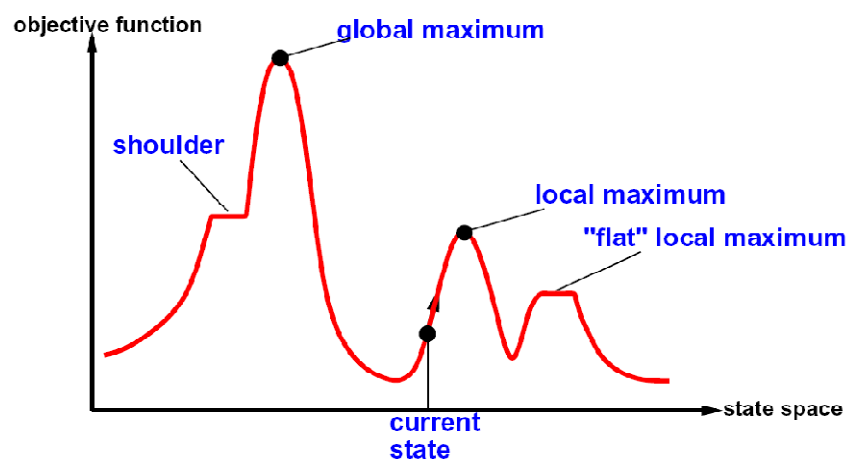
64

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?

65

Hill Climbing Diagram



- Random restarts?
- Random sideways steps?

66

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** ∞ **do**

T ← *schedule*[*t*]

if *T* = 0 **then return** *current*

next ← a randomly selected successor of *current*

ΔE ← VALUE[*next*] − VALUE[*current*]

if $\Delta E > 0$ **then** *current* ← *next*

else *current* ← *next* only with probability $e^{\Delta E/T}$

67

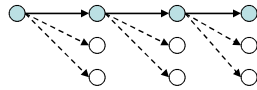
Simulated Annealing

- Theoretical guarantee:
 - 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 every make them all in a row
 - People think hard about *ridge operators* which let you jump around the space in better ways

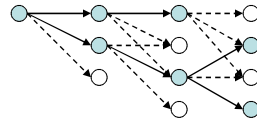
68

Beam Search

- Like greedy search, but keep K states at all times:



Greedy Search



Beam Search

- Variables: beam size, encourage diversity?
- The best choice in MANY practical settings
- Complete? Optimal?
- What criteria to order nodes by?

69