CS 188: Artificial Intelligence
Spring 2007

Lecture 5: Local Search and CSPs

1/30/2007

Srini Narayanan – UC Berkeley

Many slides over the course adapted from Dan klein, Stuart Russell and Andrew Moore

Announcements

- Assignment 1 due today 11:59 PM
- Assignment 2 out tonight,
  - due 2/12 11:59 PM
- Python Lab 3-5 PM Friday 2/2

Consistent Heuristic

A heuristic h is **consistent** if

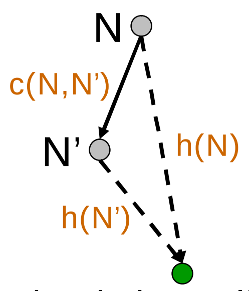
1) for each node N and each child N’ of N:

   h(N) £ c(N,N’) + h(N’)

   [Intuition: h gets more and more precise as we get deeper in the search tree]

2) for each goal node G:

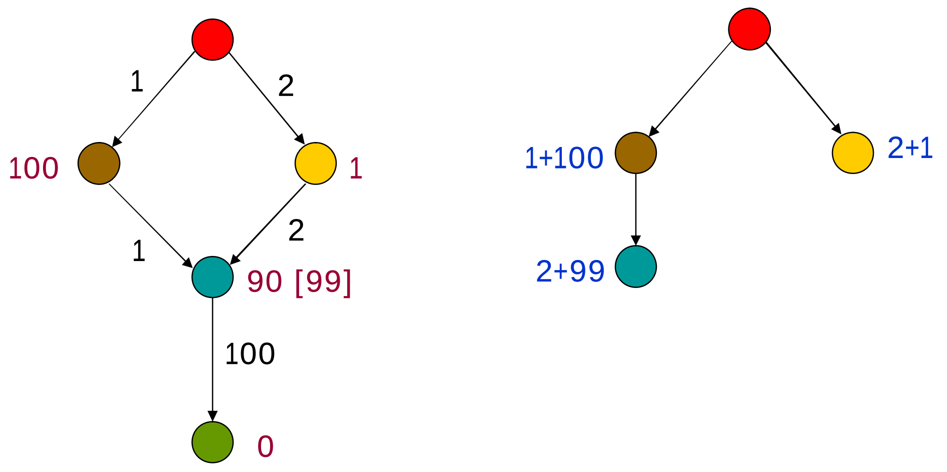
   h(G) = 0



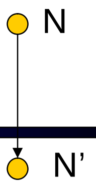
(triangle inequality)

The heuristic is also said to be **monotone**

What to do with revisited states?



Proof



1) Consider a node N and its child N'
   Since h is consistent: h(N) £ c(N,N')+h(N')

   f(N) = g(N)+h(N) ≤ g(N)+c(N,N')+h(N') = f(N')
   So, f is non-decreasing along any path

2) If K is selected for expansion, then any other node K' in the fringe verifies f(K') ≥ f(K)

   So, if one node K' lies on another path to the state of K, the cost of this other path is no smaller than the path to K (since h(K) = h(K'))

Result #2: If h is consistent, then whenever A* expands a node, it has already found an optimal path to this node's state

Trivial Heuristics, Dominance

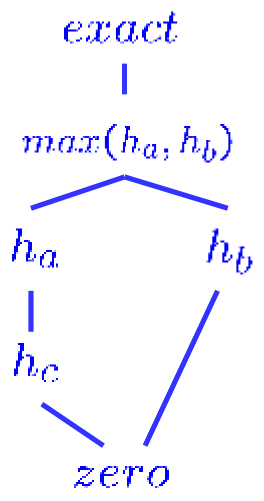
- Dominance:

$$\forall n : h_a(n) \geq h_c(n)$$

- Heuristics form a semi-lattice:
  - Max of admissible heuristics is admissible

$$h(n) = max(h_a(n), h_b(n))$$

- Trivial heuristics
  - Bottom of lattice is the zero heuristic (what does this give us?)
  - Top of lattice is the exact heuristic



Summary: A*

- A* uses both backward costs and (estimates of) forward costs
- A* is optimal with admissible and consistent heuristics
- Heuristic design is key: often use relaxed problems

A* Applications

- Pathing / routing problems
- Resource planning problems
- Robot motion planning
- Language analysis
- Machine translation
- Speech recognition
- …

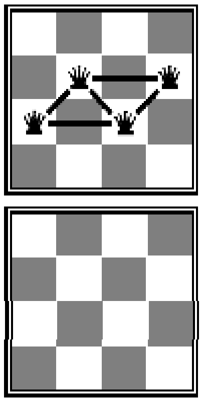
On Completeness and Optimality

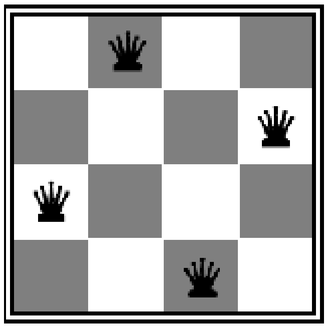
- A* with a consistent heuristic function has nice properties: completeness, optimality, no need to revisit states
- Theoretical completeness does not mean “practical” completeness if you must wait too long to get a solution (space/time limit)
- So, if one can’t design an accurate consistent heuristic, it may be better to settle for a non-admissible heuristic that “works well in practice”, even through completeness and optimality are no longer guaranteed

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)

Example: N-Queens

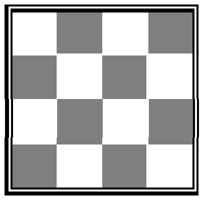




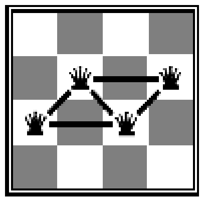
- What are the states?
- What is the start?
- What is the goal?
- What are the actions?
- What should the costs be?

Types of Problems

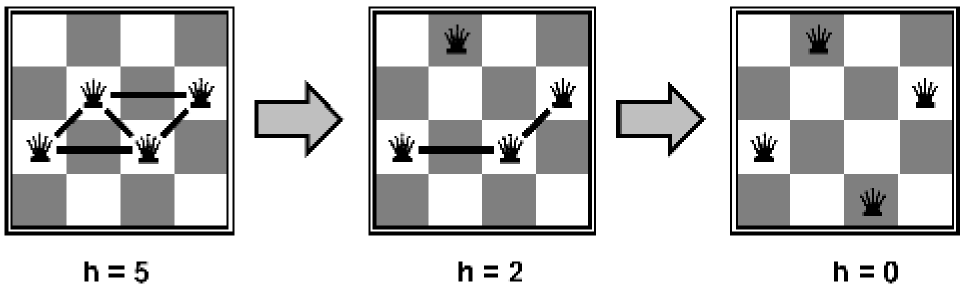
- Planning problems:
  - We want a path to a solution (examples?)
  - Usually want an optimal path
  - Incremental formulations



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

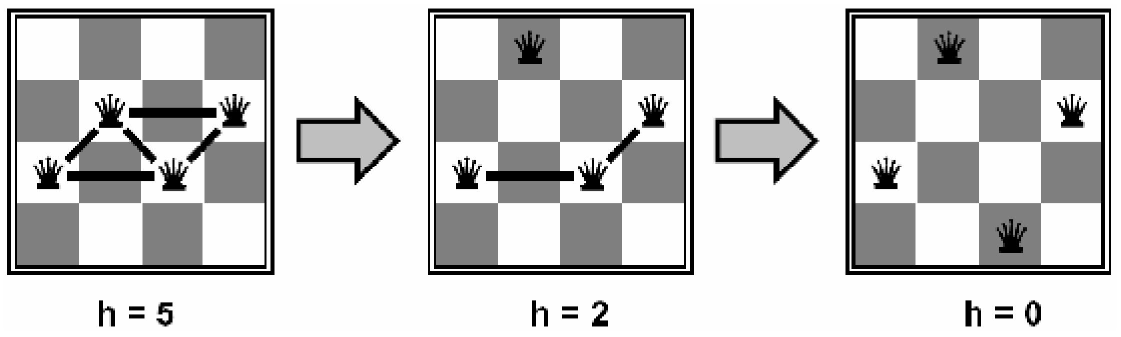


Example: 4-Queens



- 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

Example: N-Queens

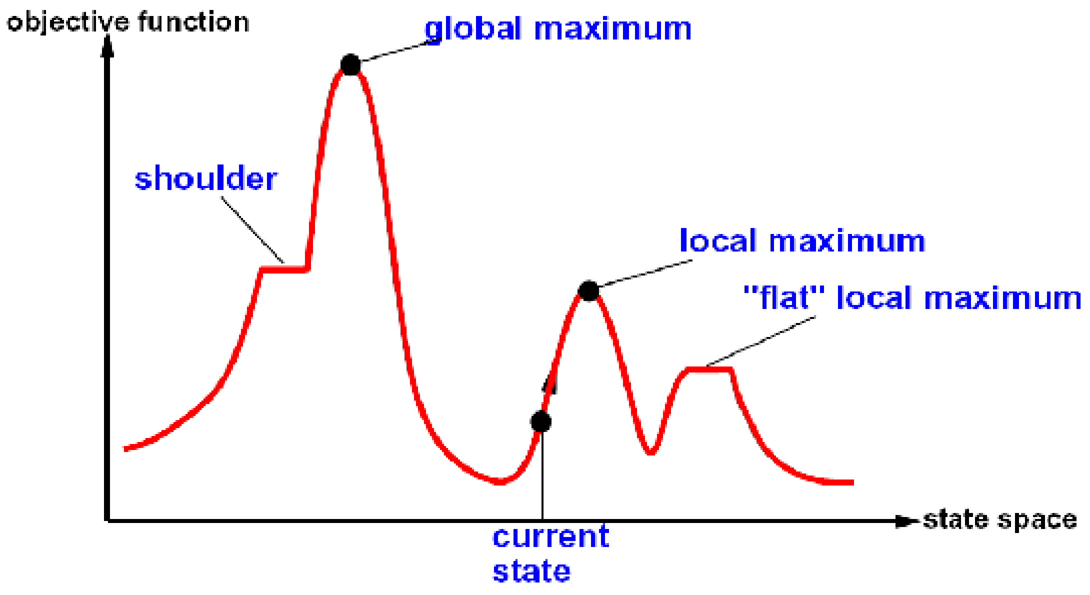


- Start wherever, move queens to reduce conflicts
- Almost always solves large n-queens nearly instantly

Hill Climbing

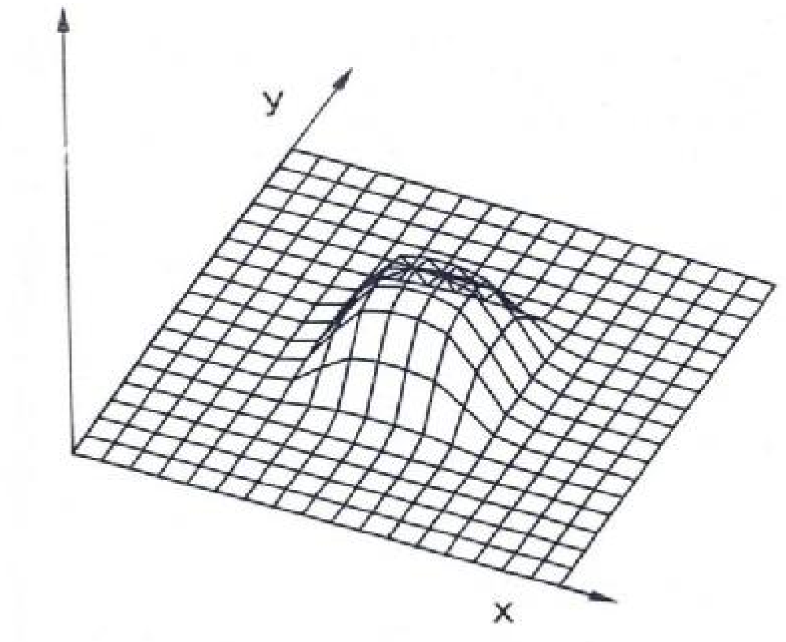
- Simple, general idea:
  - Start wherever
  - Always choose the best neighbor
  - If no neighbors have better scores than current, quit
- Why can this be a terrible idea?
  - Complete?
  - Optimal?
- What’s good about it?

Hill Climbing Diagram



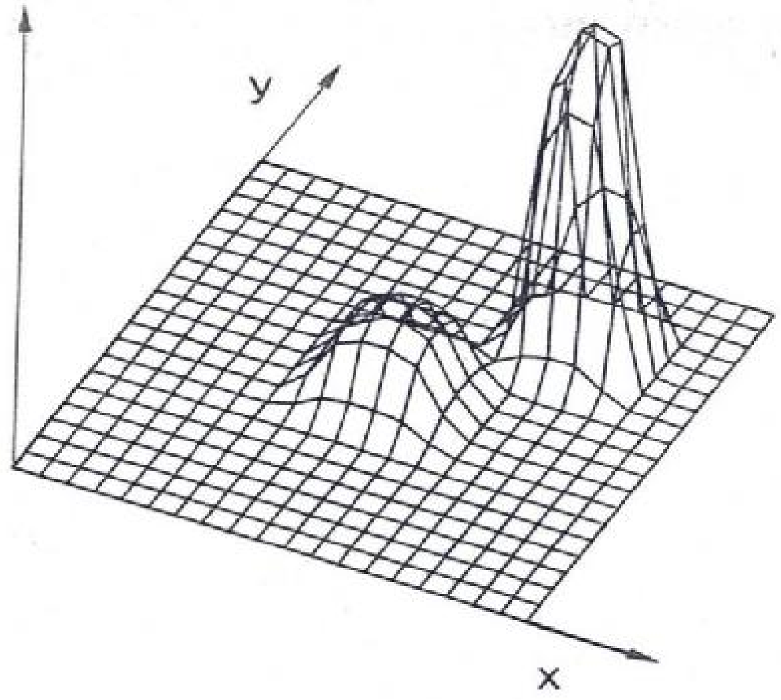
- Random restarts?
- Random sideways steps?

The Shape of an Easy Problem

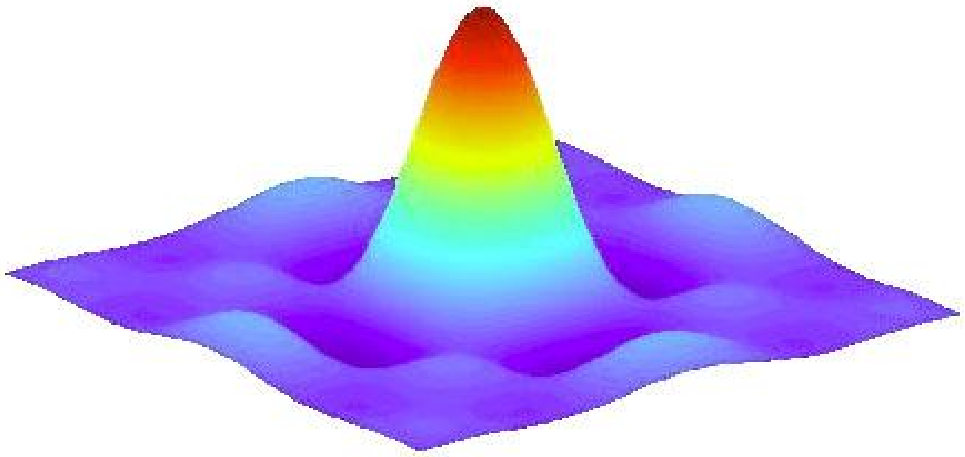


This and next several slides from Goldberg '89

The Shape of a Harder Problem



The Shape of a Yet Harder Problem



Remedies to drawbacks of hill climbing

- Random restart
- Problem reformulation
- In the end: Some problem spaces are great for hill climbing and others are terrible.

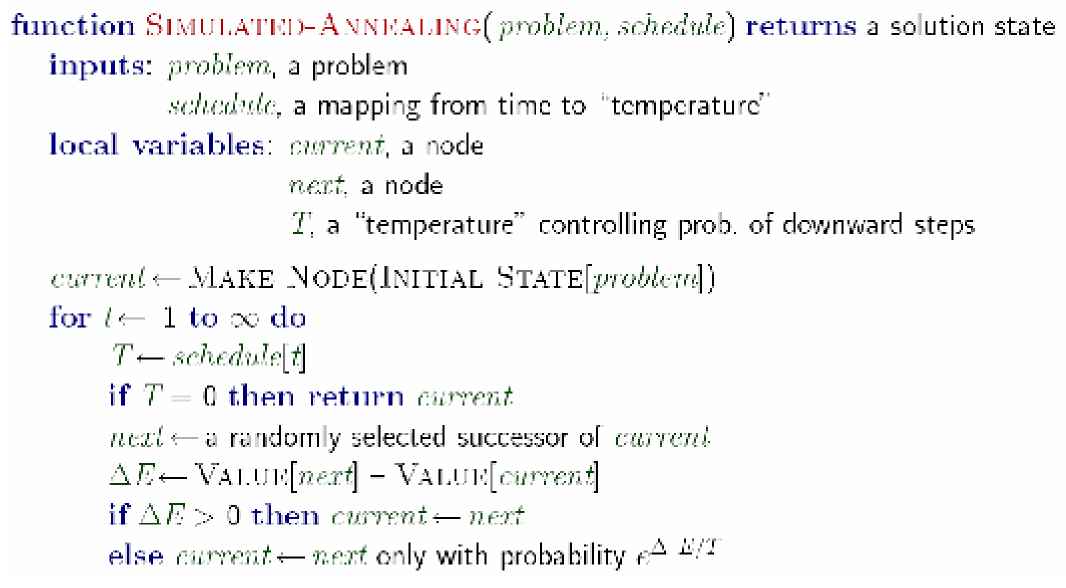
Monte Carlo Descent

```
1)  S ß initial state
2)  Repeat k times:
    a)  If GOAL?(S) then return S
    b)  S' ß successor of S picked at random
    c)  if h(S') ≤ h(S)  then S ß S'
    d)  else
        -  Δh = h(S')-h(S)
        -  with probability ~ exp(–Δh/T), where T is called the
           "temperature" S ß S'          [Metropolis criterion]
3)  Return failure
```

Simulated annealing lowers T over the k iterations.
It starts with a large T and slowly decreases T

Simulated Annealing

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



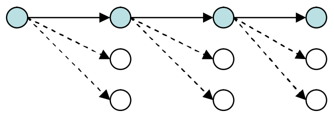
```
function SIMULATED-ANNEALING(problem, schedule) returns a solution state
   inputs: problem, a problem
           schedule, a mapping from time to "temperature"
   local variables: current, a node
                    next, a node
                    T, a "temperature" controlling prob. of downward 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
       $\Delta E$ ← VALUE[next] − VALUE[current]
       if $\Delta E > 0$ then current ← next
       else current ← next only with probability $e^{\Delta E/T}$
```

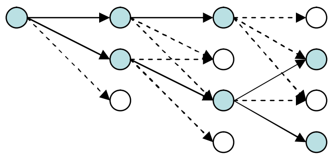
Simulated Annealing

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

Beam Search

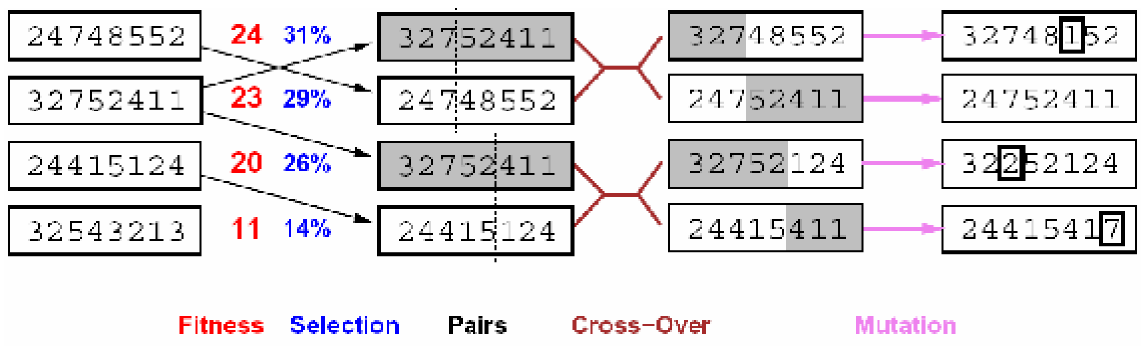
- Like greedy search, but keep K states at all times:


Greedy Search


Beam Search

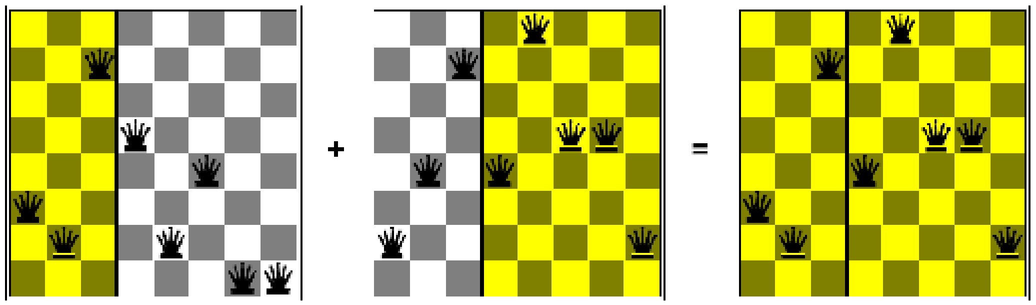
- Variables: beam size, encourage diversity?
- The best choice in MANY practical settings
- Complete? Optimal?
- Why do we still need optimal methods?

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

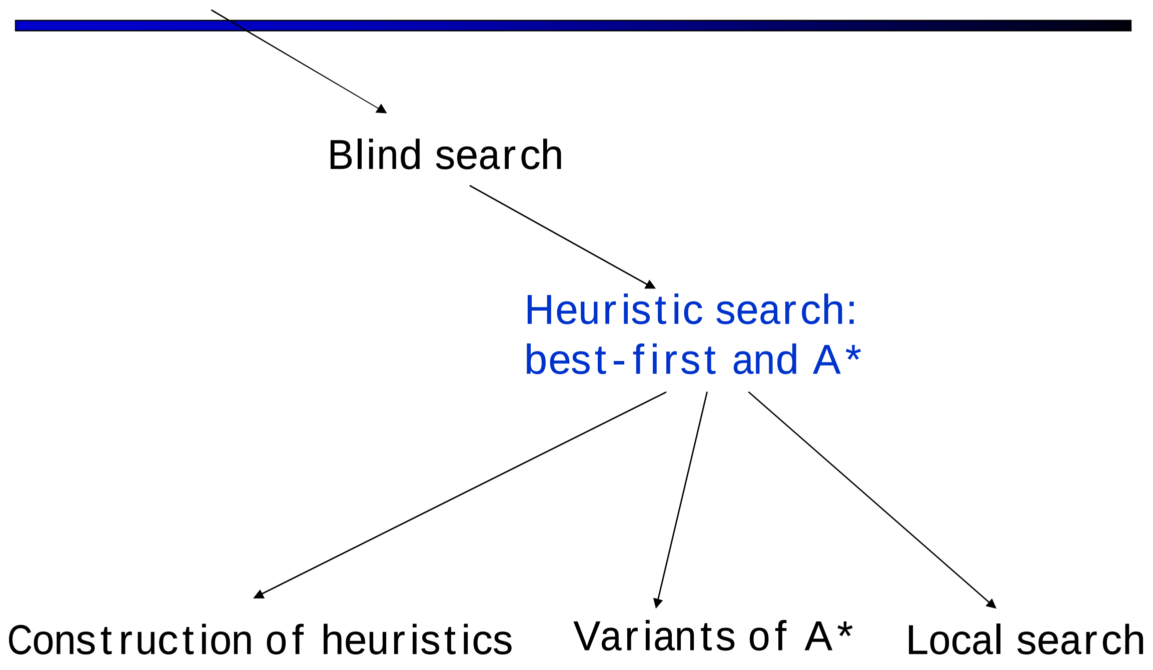


- Why does crossover make sense here?
- When wouldn’t it make sense?
- What would mutation be?
- What would a good fitness function be?

The Basic Genetic Algorithm

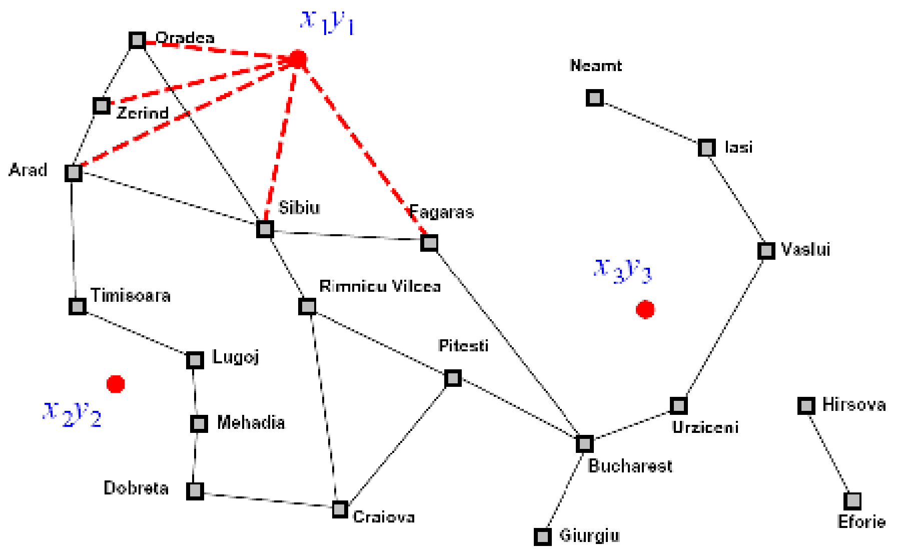
1. Generate random population of chromosomes
2. Until the end condition is met, create a new population by repeating following steps
    1. Evaluate the fitness of each chromosome
    2. Select two parent chromosomes from a population, weighed by their fitness
    3. With probability p_c cross over the parents to form a new offspring.
    4. With probability p_m mutate new offspring at each position on the chromosome.
    5. Place new offspring in the new population
3. **Return** the best solution in current population

Search problems


Blind search
Heuristic search: best-first and A*
Construction of heuristics
Variants of A*
Local search

Continuous Problems

- Placing airports in Romania
  - States: $(x_1,y_1,x_2,y_2,x_3,y_3)$
  - Cost: sum of squared distances to closest city

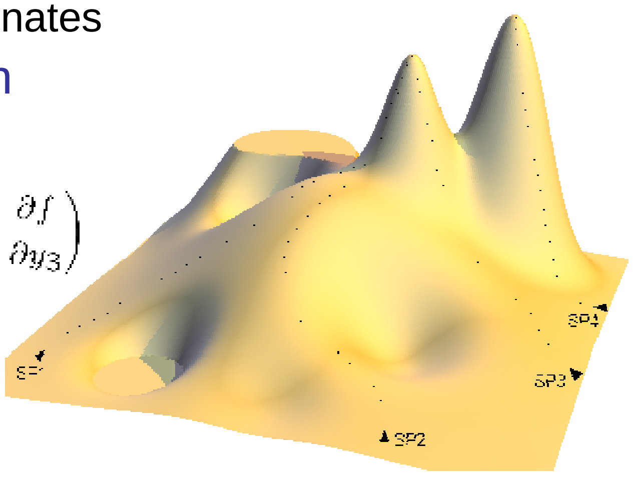


Gradient Methods

- How to deal with continous (therefore infinite) state spaces?
- Discretization: bucket ranges of values
  - E.g. force integral coordinates
- Continuous optimization
  - E.g. gradient ascent

$$\nabla f = \left(\frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial y_1}, \frac{\partial f}{\partial x_2}, \frac{\partial f}{\partial y_2}, \frac{\partial f}{\partial x_3}, \frac{\partial f}{\partial y_3}\right)$$

$$x \leftarrow x + \alpha \nabla f(x)$$

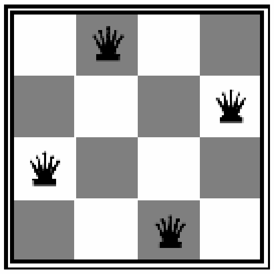


  - More later in the course

Image from vias.org

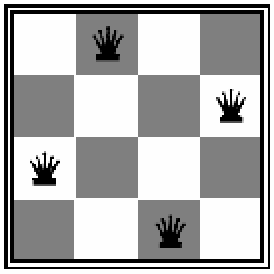
Constraint Satisfaction Problems

- Standard search problems:
  - State is a “black box”: any old data structure
  - Goal test: any function over states
  - Successors: any map from states to sets of states
- Constraint satisfaction problems (CSPs):
  - 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



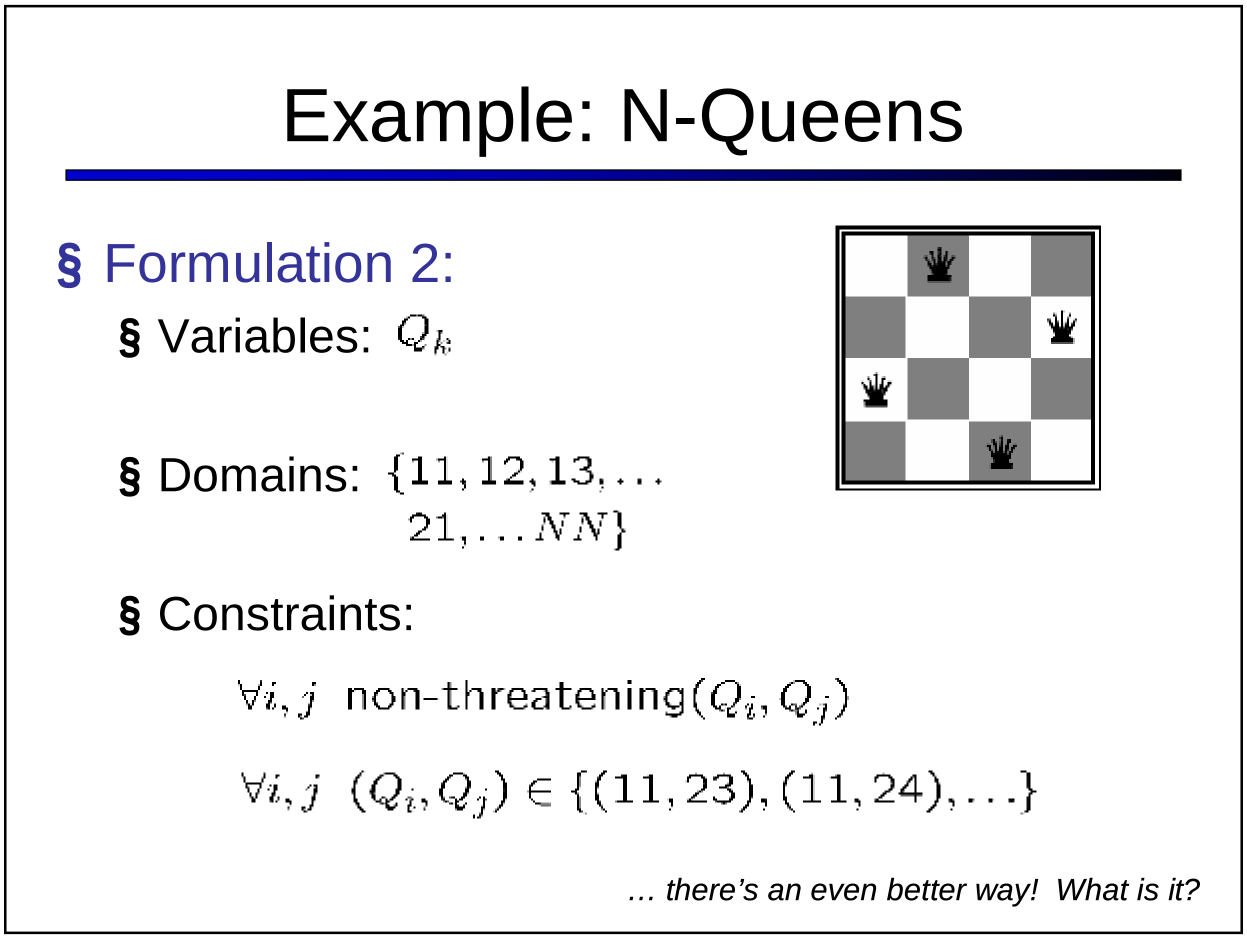
Example: N-Queens

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



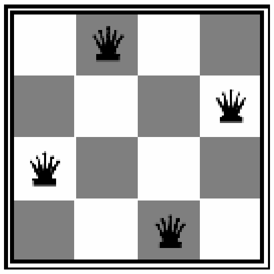
$$\forall i,j,k \ \ (X_{ij}, X_{ik}) \in \{(0,0),(0,1),(1,0)\}$$
$$\forall i,j,k \ \ (X_{ij}, X_{kj}) \in \{(0,0),(0,1),(1,0)\}$$
$$\forall i,j,k \ \ (X_{ij}, X_{i+k,j+k}) \in \{(0,0),(0,1),(1,0)\}$$
$$\forall i,j,k \ \ (X_{ij}, X_{i+k,j-k}) \in \{(0,0),(0,1),(1,0)\}$$

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



Example: N-Queens

- Formulation 2:
  - Variables: Q_k
  - Domains: $\{11, 12, 13, \ldots$
    $21, \ldots NN\}$
  - Constraints:



$$\forall i, j \;\; \text{non-threatening}(Q_i, Q_j)$$

$$\forall i, j \;\; (Q_i, Q_j) \in \{(11, 23), (11, 24), \ldots\}$$

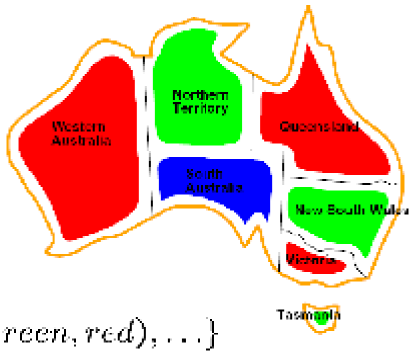
… there's an even better way! What is it?

Example: Map-Coloring

- Variables: $WA,\ NT,\ Q,\ NSW,\ V,\ SA,\ T$
- Domain: $D = \{red, green, blue\}$
- Constraints: adjacent regions must have different colors

  $WA \neq NT$

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

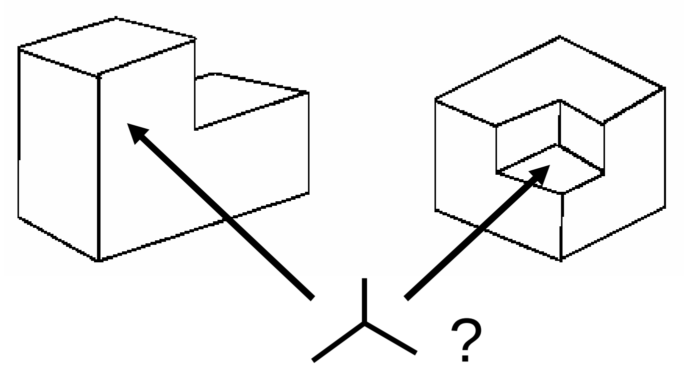


- Solutions are assignments satisfying all constraints, e.g.:

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

Example: The Waltz Algorithm

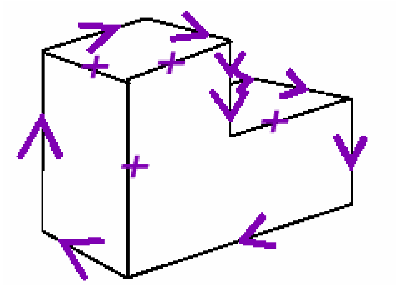
- The Waltz algorithm is for interpreting line drawings of solid polyhedra
- An early example of a computation posed as a CSP



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

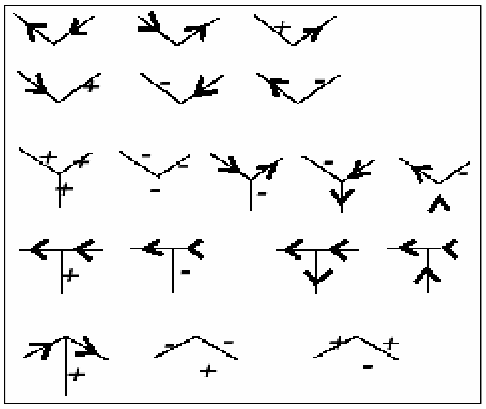
Waltz on Simple Scenes

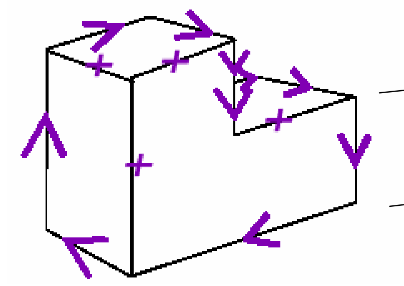
- Assume all objects:
  - Have no shadows or cracks
  - Three-faced vertices
  - “General position”: no junctions change with small movements of the eye.
- Then each line on image is one of the following:
  - Boundary line (edge of an object) (→) with right hand of arrow denoting “solid” and left hand denoting “space”
  - Interior convex edge (+)
  - Interior concave edge (-)

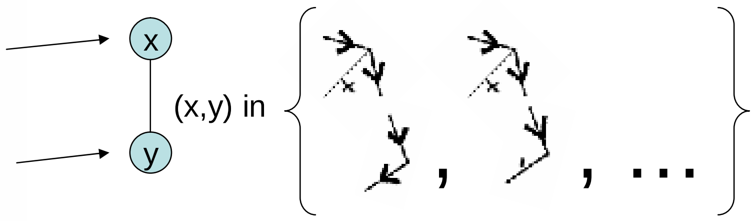


Legal Junctions

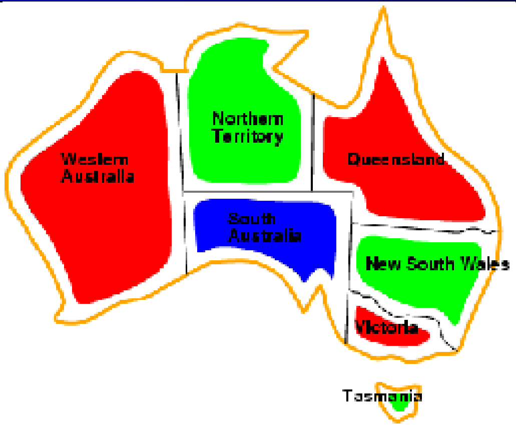
- Only certain junctions are physically possible
- How can we formulate a CSP to label an image?
- Variables: vertices
- Domains: junction labels
- Constraints: both ends of a line should have the same label







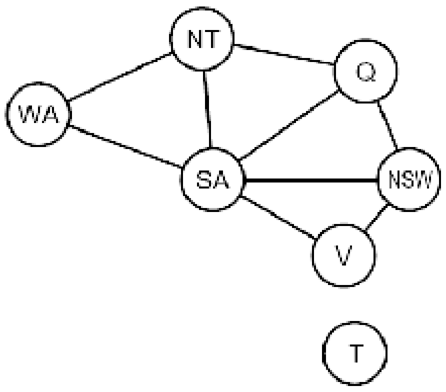
Example: Map-Coloring



- Solutions are complete and consistent assignments, e.g., WA = red, NT = green,Q = red,NSW = green,V = red,SA = blue,T = green

Constraint Graphs

- Binary CSP: each constraint relates (at most) two variables
- 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:

  $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$

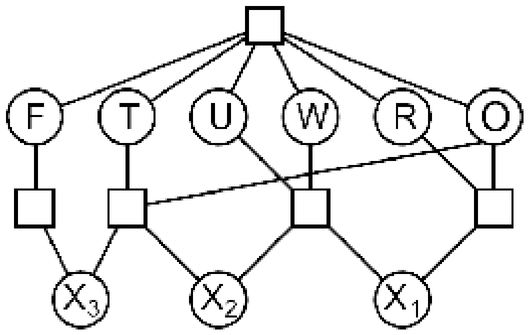
- Constraints:

  $\text{alldiff}(F, T, U, W, R, O)$

  $O + O = R + 10 \cdot X_1$

  $\ldots$

```
    T W O
+   T W O
---------
  F O U R
```



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
    - Need a *constraint language*, e.g., $\text{StartJob}_1 + 5 < \text{StartJob}_3$
    - Linear constraints solvable, nonlinear undecidable
- Continuous variables
  - E.g., start/end times for Hubble Telescope observations
  - 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):

$$SA \neq green$$

  - 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
- Spreadsheets
- Transportation scheduling
- Factory scheduling
- Floorplanning

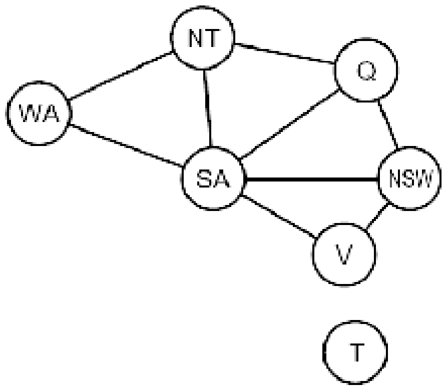
- Many real-world problems involve real-valued variables…

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
    - fail if no legal assignment
  - Goal test: the current assignment is complete and satisfies all constraints

Search Methods

- What does DFS do?



- What's the obvious problem here?
- What's the slightly-less-obvious problem?

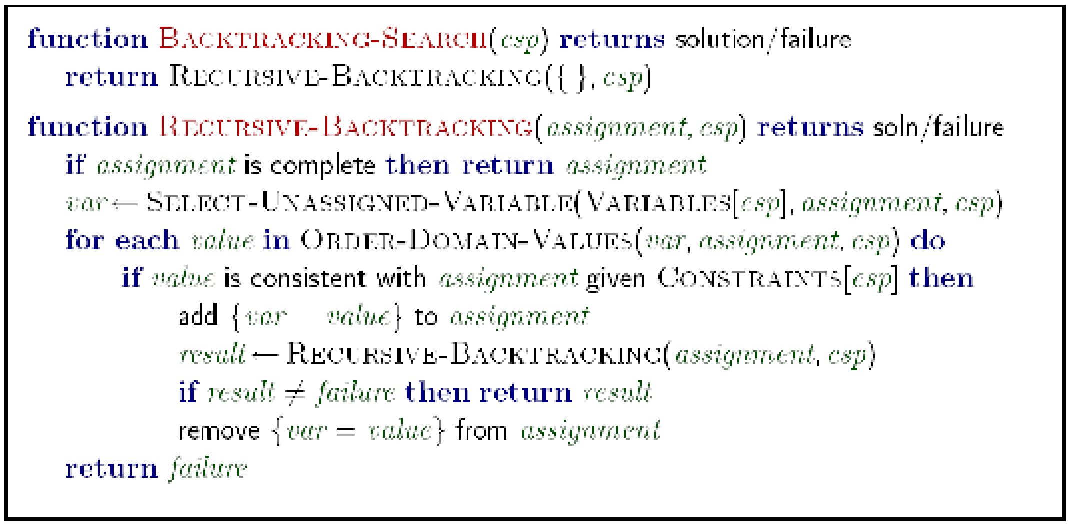
CSP formulation as search

1. This is the same for all CSPs
2. Every solution appears at depth *n* with *n* variables
   à use depth-first search
3. Path is irrelevant, so can also use complete-state formulation
4. $b = (n - l)d$ at depth l, hence $n! \cdot d^n$ leaves

Backtracking Search

- Idea 1: Only consider a single variable at each point:
  - Variable assignments are commutative
  - 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
- Depth-first search for CSPs with these two improvements is called *backtracking search*
- Backtracking search is the basic uninformed algorithm for CSPs
- Can solve n-queens for $n \approx 25$

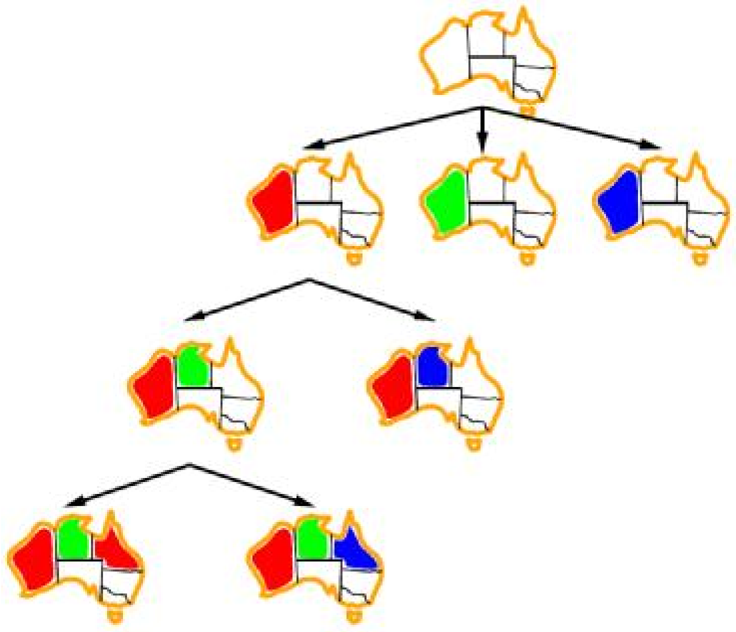
Backtracking Search



```
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
```

- What are the choice points?

Backtracking Example



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?