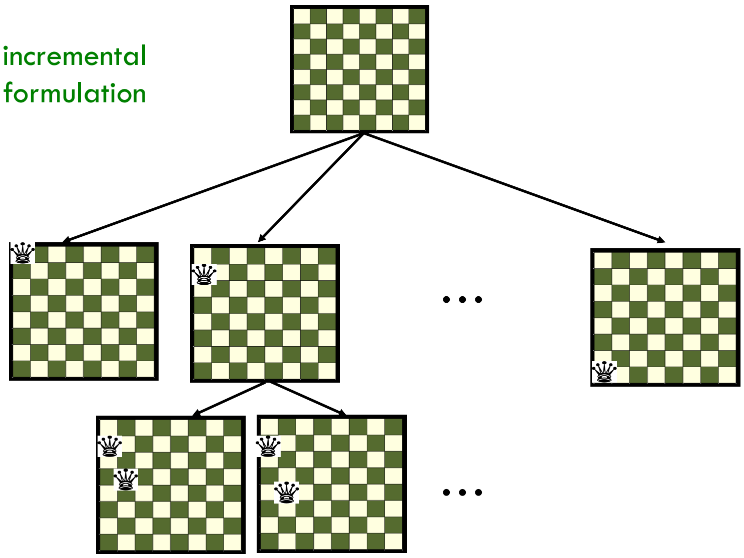
LOCAL SEARCH

Today

- Reading
  - AIMA Chapter 4.1-4.2, 5.1-5.2
- Goals
  - Local search algorithms
    - hill-climbing search
    - simulated annealing
    - local beam search
    - genetic algorithms
    - gradient descent and Newton-Rhapson
  - Introduce adversarial search

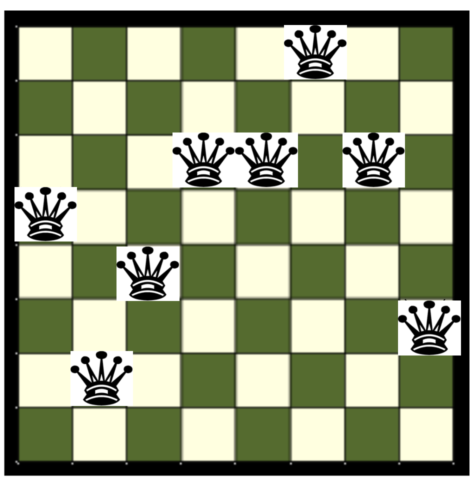
Recall the N-Queens problem

incremental formulation



N-Queens alternative approach

complete state formulation

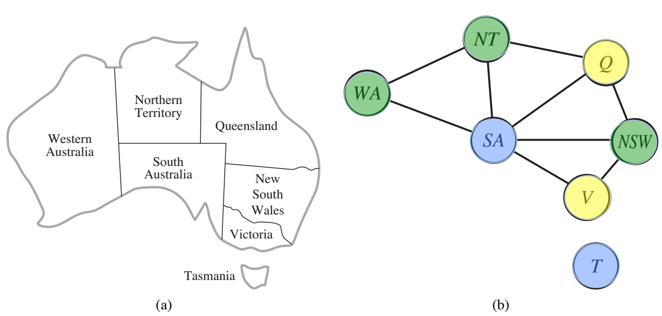


Local search

- The basic idea:
  1. Randomly initialize (complete) state
  2. If not goal state,
     a. make local modification to state to generate a neighbor state OR
     b. enumerate all neighbor states and choose the best
  3. Repeat step 2 until goal state is found (or out of time)
- Requires the ability to *quickly*:
  - Generate a random (probably-not-optimal) state
  - Evaluate the quality of a state
  - Move to other states (well-defined neighborhood function)

Graph Coloring

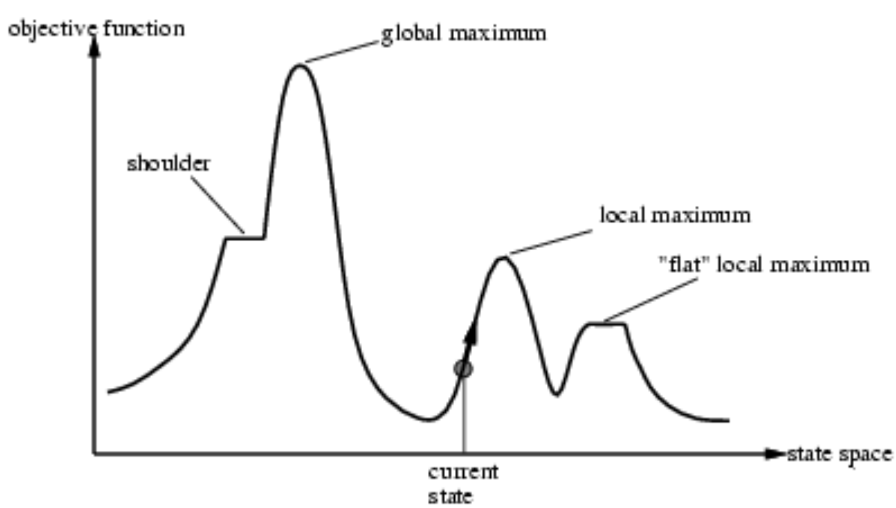
1. Start with random coloring of nodes
2. If not goal state, change color of one node
3. Repeat 2



Local Search Algorithms

- Useful when path to the goal state is irrelevant
- Keep track of "current" state only
- Explore nearby "neighbor" (successor) states
- Algorithms include:
  - Hill-climbing
  - Simulated annealing
  - Local beam search
  - Genetic algorithms
  - Gradient descent (Newton-Rhapson)

Local Search Algorithms

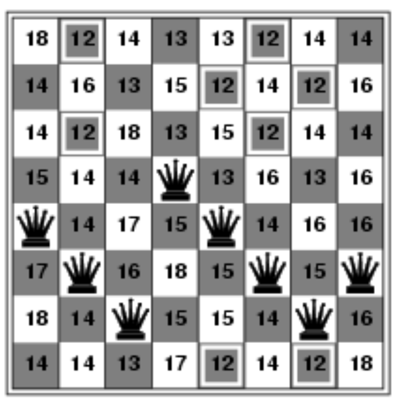


Hill-climbing Search

- "Like climbing Everest in thick fog with amnesia"

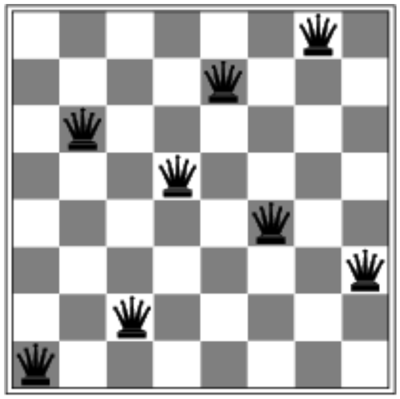
```
function HILL-CLIMBING(problem) returns a state that is a local maximum
   inputs: problem, a problem
   local variables: current, a node
                    neighbor, a node

   current ← MAKE-NODE(INITIAL-STATE[problem])
   loop do
      neighbor ← a highest-valued successor of current
      if VALUE[neighbor] ≤ VALUE[current] then return STATE[current]
      current ← neighbor
```

Hill-climbing Search: 8-queens problem



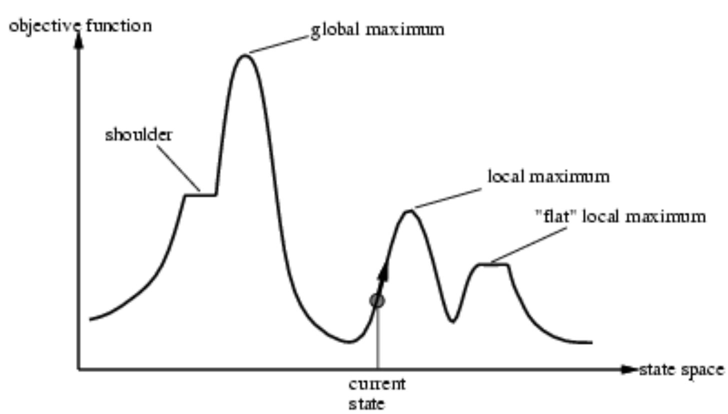
- *h* = number of pairs of queens that are attacking each other
- *h = 17* for the above state

Hill-climbing search: 8-queens problem



- A local minimum with $h = 1$

Problems with hill-climbing?



Hill-climbing Performance

- Complete – No
- Optimal - No
- Time – Depend
- Space – O(1)

Hill-climbing Variants

- Stochastic Hill Climbing
  - Randomly chooses uphill successors
  - Probability of selection proportional to steepness
- First-choice hill climbing
  - Choose first generated uphill successor
- Random-restart hill climbing
  - Runs multiple hill-climbing searches from random initial states

Simulated annealing search

- Idea: escape local maxima by allowing some "bad" moves but gradually decrease their frequency

```
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 ΔE > 0 then current ← next
       else current ← next only with probability e^(ΔE/T)
```

Local Beam Search

- Idea: Keep as many states in memory as possible
  - Start with k randomly generated states
  - Generate all successors of all k states
  - If goal is found, stop. Else select the k best successors from the complete list of successors and repeat.

- What's one possible shortcoming of this approach?

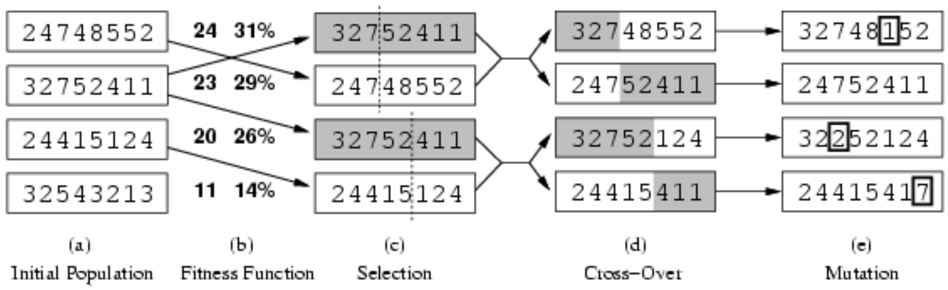
Local Beam Search

- Idea: Keep as many states in memory as possible
  - Start with k randomly generated states
  - Generate all successors of all k states
  - If goal is found, stop. Else select the k best successors from the complete list of successors and repeat.
- Possible problem: All k states can become concentrated in the same part of the search space
- Stochastic beam search
  - Choose k successors at random where the probability of selection is proportional to its objective function value

Genetic Algorithms

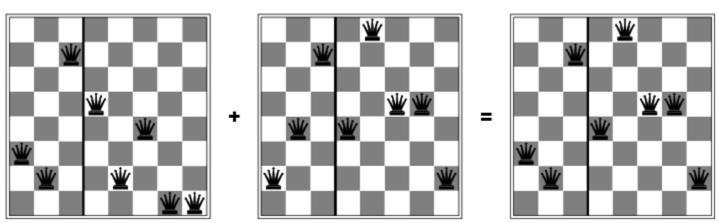
- Generate a successor state by combining two parent states
  - Begin with k randomly generated states (population) represented as strings over some finite alphabet
  - Evaluate the fitness of each state via fitness function (higher values = better state)
  - Repeat k times:
    - Randomly select 2 states proportional to their fitness
    - Randomly pick a crossover point and produce a new state
    - Randomly mutate each location of the new state

Genetic Algorithms



- Fitness function: number of non-attacking pairs of queens
- 24/(24+23+20+11) = 31%
- 23/(24+23+20+11) = 29% etc

Genetic Algorithms

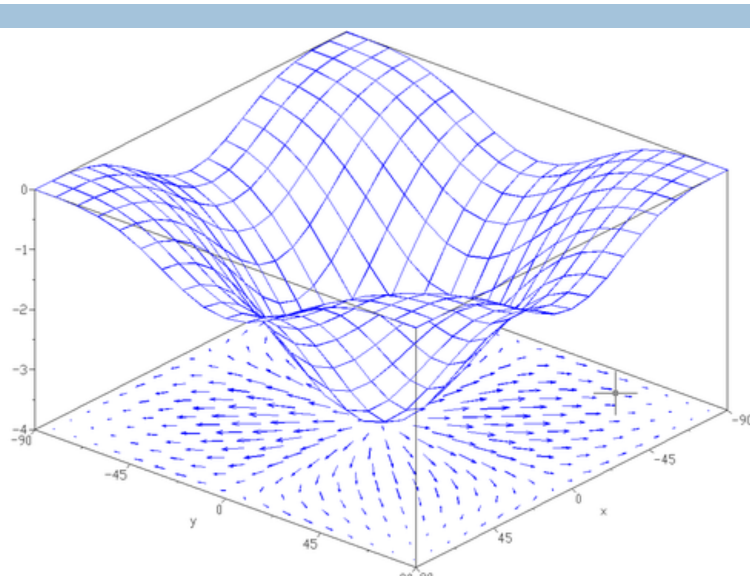


3275411 24748552 32748552

Genetic Algorithms

- Crossover can produce an offspring that is in an entirely different area of the search space than either parent
  - Sometimes offspring is outside of the “feasible” or “evaluable” region
- Either replace entire population at each step (generational GA) or replace just a few (low fitness) members of the population (steady-state GA)
- The benefit comes from having a representation where contiguous blocks are actually meaningful

Gradient-based methods



Gradient-based methods

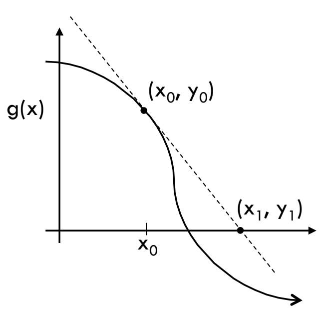
- Gradient-based methods are similar to hill-climbing
  - Find the best direction and take it
- Nevertheless, they are widely used

"Their operation is similar to a blind man walking up a hill, whose only knowledge of of the hill comes from what passes under his feet. If the hill is predictable in some fashion, he will reach the top, but it is easy to imagine confounding terrain"

- Goffe et al., 1994

Newton-Rhapson Method

- Newton-Rhapson is a method for finding roots of a function, i.e. finding x such that g(x) = 0



Step 1: Make an initial guess x_0
Step 2: Compute new point x_1 using the update rule:

$$x_{n+1} = x_n - \frac{g(x_n)}{g'(x_n)}$$

Intuition behind the update rule:
The tangent line at x_0 is a linear approximation of g(x) at the point x_0. Since the tangent line is an approximation of g(x), the root of the tangent line is probably a better estimate of the root of g(x) than our initial guess x_0. So, to find the root of the tangent line we use the formula for slope and then solve for x_1:

$$g'(x_0) = \frac{y_1 - y_0}{x_1 - x_0} \quad \longleftarrow \text{remember me?!}$$

Using a bit of algebra, we solve for x_1 which gives us the above update rule. We then iterate this process until we converge to the root of g(x).

Newton-Rhapson applied to optimization

- When we're minimizing a function we want to find the point x^* such that $f(x^*) < f(x)$ for all x
- Recall from calculus that the slope at such a point x* is zero, i.e. $f'(x^*) = 0$
- So we can restate the problem as follows: we want to find the point x^* such that $f'(x^*) = 0$
- Now we can use the Newton-Rhapson method to find the root of the first derivative f'(x). The update rule in this case is:

$$x_{n+1} = x_n - \frac{f'(x_n)}{f''(x_n)}$$

- The function f''(x) is the second derivative.
- Ask yourself: Why does the second derivative appear in this formula?

Newton-Rhapson applied to optimization

- In the multivariate case, the update rule looks like this:

$$x_{n+1} = x_n - \frac{f'(x_n)}{f''(x_n)}$$

univariate

$$x_{n+1} = x_n - H_f^{-1}(x_n)\nabla_f(x_n)$$

multivariate

- The second derivative is given by the Hessian matrix (denoted as H) which is a square matrix that contains the second-order partial derivatives
- The first derivative is represented by the gradient (denoted using the upside down triangle)

Gradient Ascent (Descent)

- Sometimes the Hessian is too computationally expensive to compute or it cannot be inverted
- In this case, we can "replace" the second derivative with a step size constant γ

$$x \leftarrow x + \gamma \nabla F(x)$$

- The gradient (the upside down triangle) gives the direction of steepest ascent. The step cost (gamma) determines how far we step in that direction.
- For minimization, we would change the addition to subtraction (i.e. we want to move in the direction opposite to the direction of steepest ascent)
- Some information about the step size gamma:
  - The user can set the step size to any (typically positive) value
  - A step size too small results in slow progress. A step size too large can overshoot the minimum/ maximum.
  - The step size can be determined using a line search (think binary search). However, make sure that the line search itself isn't computationally expensive
  - The step size can change at each iteration. It doesn't have to stay the same
- The stopping criteria: gradient sufficiently close to zero, or difference between new and old points below threshold

Gradient Ascent (Descent)

function GRADIENT-ASCENT(F, γ) returns solution

$\quad \nabla F \leftarrow \text{COMPUTE-GRADIENT}(F)$

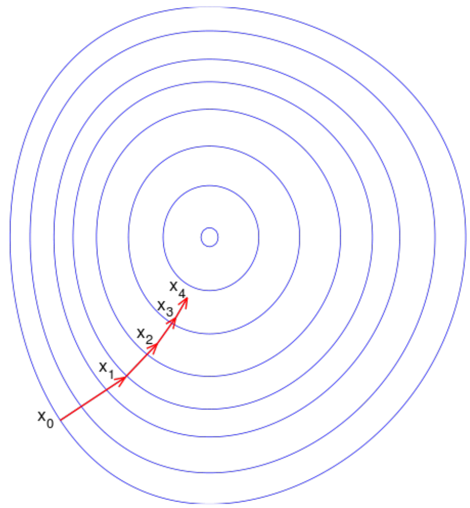
$\quad x \leftarrow \text{a randomly selected value}$

$\quad$ while stopping criteria

$\quad\quad x \leftarrow x + \gamma \nabla F(x)$

$\quad$ return x

Gradient Ascent (Descent)



Local search summary

- Hill-climbing search
  - Stochastic hill-climbing search
  - First-choice
  - Random restart hill-climbing
- Simulated annealing
- Local beam search
  - Stochastic local beam search
- Genetic algorithms
- Gradient-based methods
  - Newton-Rhapson
  - Gradient ascent (descent)