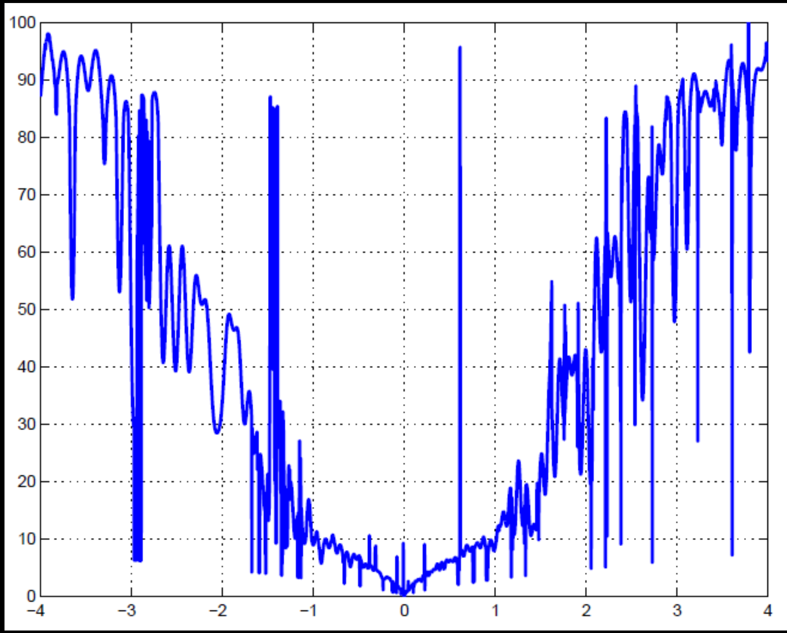
Search for Optimization

Real-World Problems

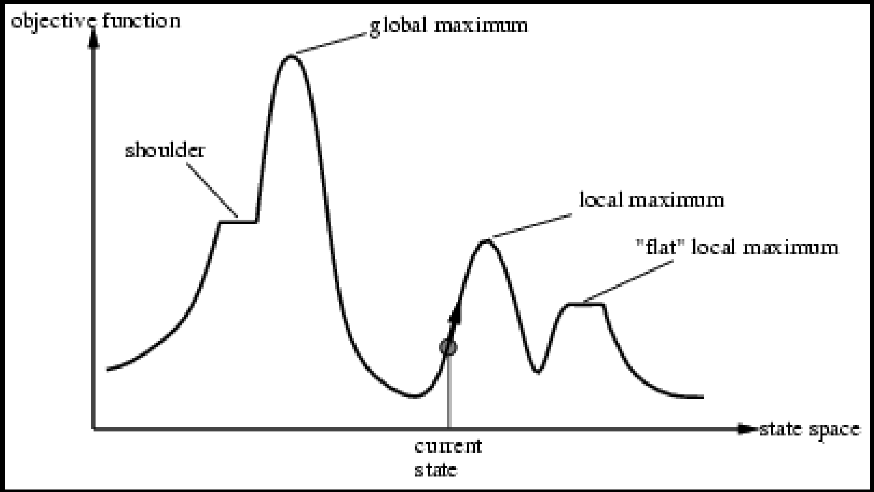


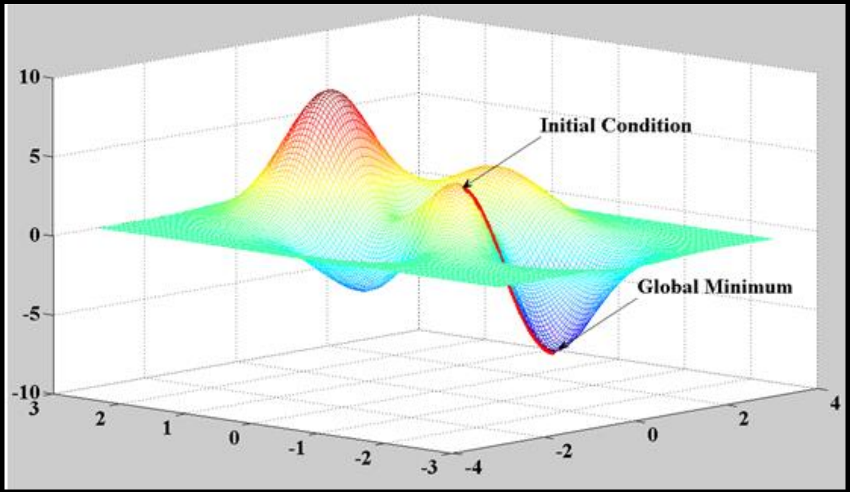
"*many real world problems have a landscape that looks more like a widely scattered family of balding porcupines on a flat floor, with miniature porcupines living on the tip of each porcupine needle, ad infinitum.*"

-Russell and Norvig

A different view of optimization

- In lectures on convex optimization, we assumed the function was convex, now we remove that assumption!





- Removing this allows us to consider
  - non-convex continuous problems
  - problems where variables are discrete (really important!)
- Unfortunately, not much we can prove in non-convex optimization ☹

Outline

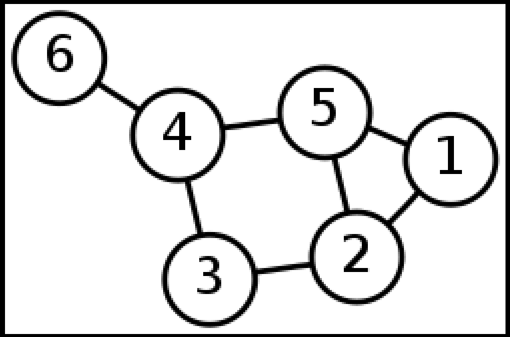
- Graphs
- Local search methods
  - Hill Climbing
- Simulated Annealing
- Genetic Algorithms

Graphs

- A graph is a set of vertices (also called nodes) V and edges E:

$$G = (V, E)$$

- Example:

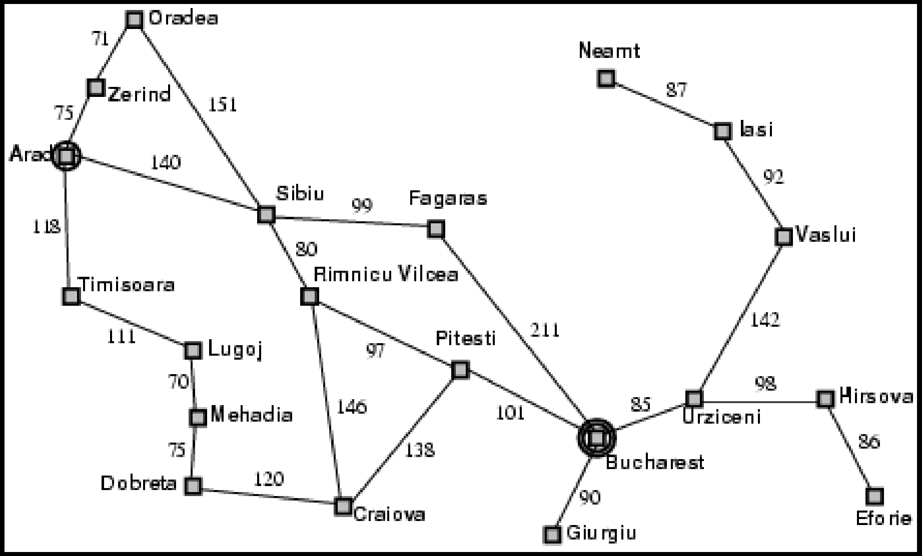


  - V = {1,2,3,4,5,6}
  - E = {(6,4),(4,5),(4,3),(3,2),(5,2),(2,1),(5,1)}

- In general there may be an infinite number of vertices and edges

Graphs: Why this representation?

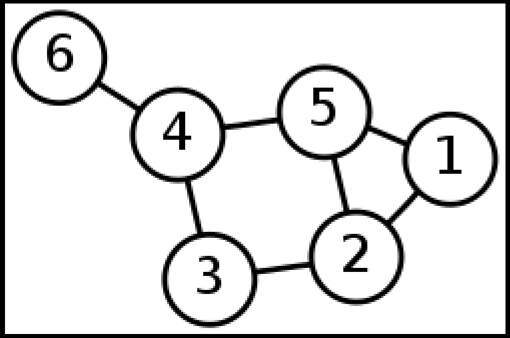
- Graphs capture the idea of adjacency – i.e. what is “next to” what
  - A node *u*’s neighbors are the set of nodes connected to *u* by an edge
- We can use adjacency relationships to search the graph for a certain node or a path between nodes
- Example:



Graphs for optimization

- For optimization, adjacency between nodes can be used to determine what solutions to explore next

- Consider each node in the graph to represent a solution to a problem:

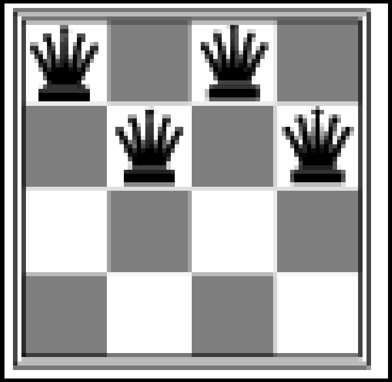


- Here solution 1 is adjacent to solution 2 because they are “close” to each other in solution space
- Adjaceny must be defined by the algorithm designer
  - A good adjacency definition helps you search the space of solutions systematically

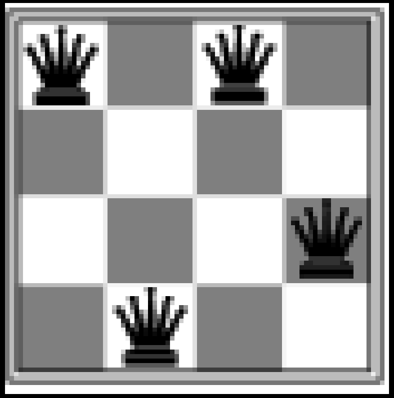
Example: *n*-queens

- The problem: Put *n* queens on an *n* × *n* board with no two queens on the same row, column, or diagonal

An example solution (not good!):



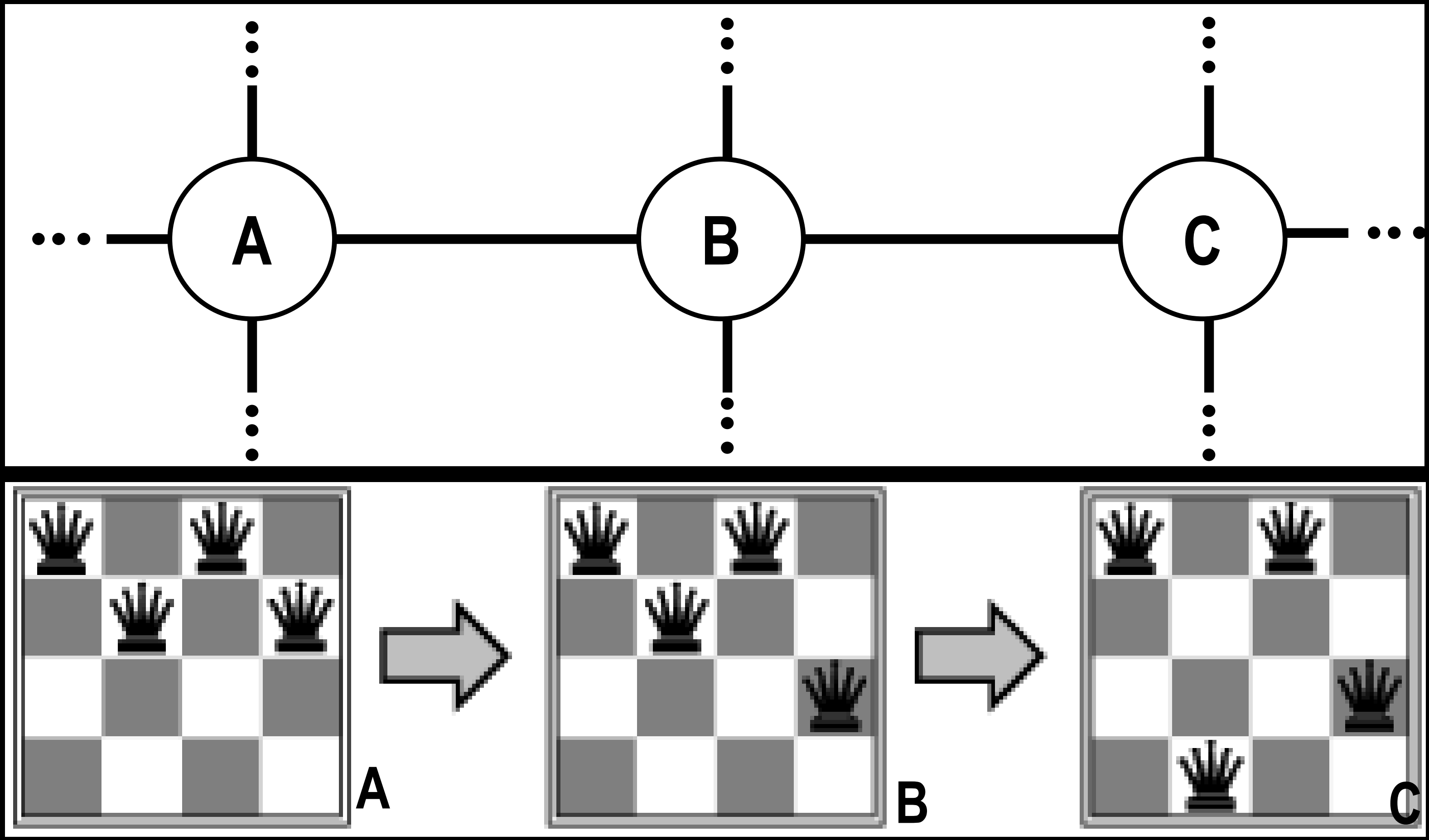
A much better solution:



- Define adjacency: Two solutions *u* and *v* are adjacent iff $n - 1$ queens are at the same position in *u* and *v*
  - You could make up other definitions, too!

Example: *n*-queens

- Starting with a bad solution, can use adjacency to search for a good one:



Local search algorithms

- To try to solve optimization problems using *search* we
  1. Define the set of possible solutions
     - This is then the set of nodes
  2. Define adjacency between solutions
     - This defines the set of edges
  3. Define a cost/value/fitness function for nodes
     - Outputs how good a solution is

- Can use local search algorithms to search the graph for the best solution
  - keep a (sometimes) single "current" state, try to improve it

- Descent methods (from before) are a form of local search algorithms
  - But these only work for convex continuous functions

- Some local search methods can also handle discrete variables

Local Search

- Operates by keeping track of only the current solution and moving only to neighbors of that solution
- Often used for:
  - Optimization problems
  - Scheduling
  - Task assignment
  - …many other problem where the goal is to find the best state according to some **objective function**

Local Search: Hill-climbing

Hill-climbing search

- Consider next possible moves (i.e. neighbors)
  - Pick the one that improves cost/value/fitness function the most

- “Like climbing Everest in thick fog with amnesia”

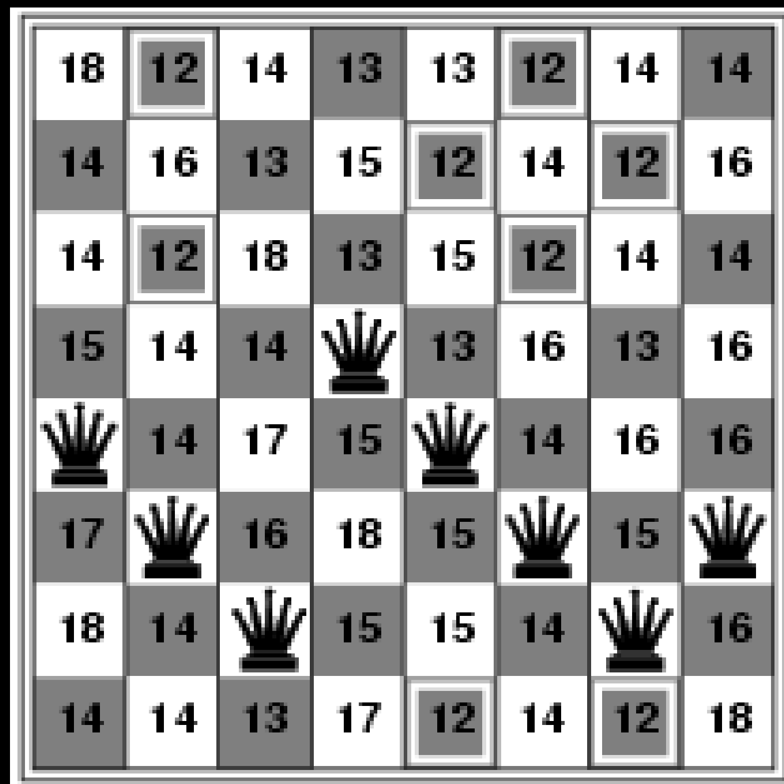
Hill-climbing search

```
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
```

“successor” is
a synonym for
neighbor

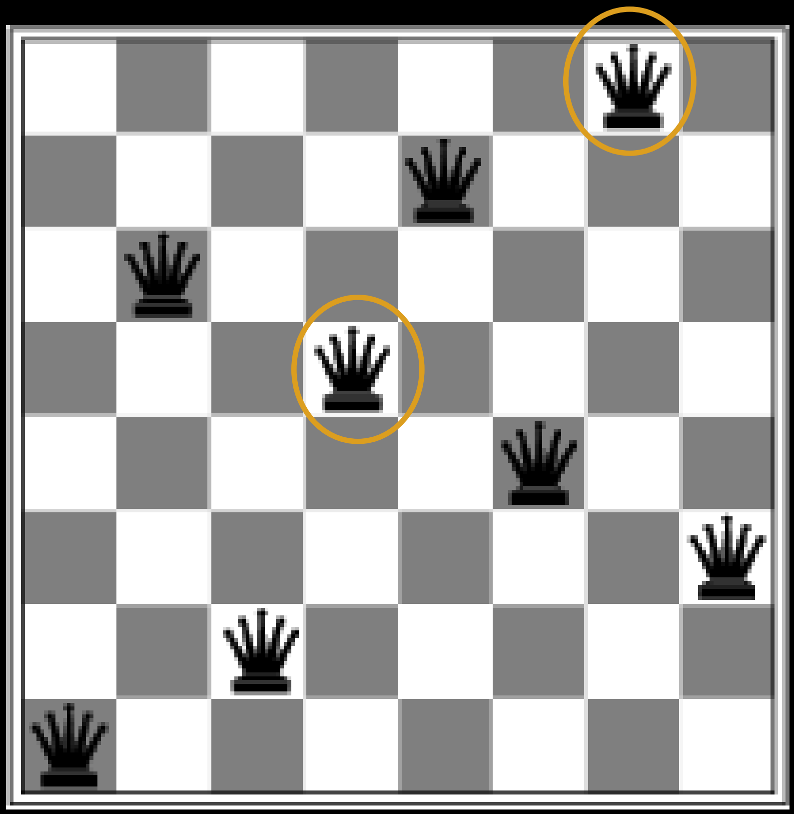
Hill-climbing search: 8-queens problem



- *h* = number of pairs of queens that are attacking each other, either directly or indirectly
- *h* = *17* for the above state

Hill-climbing search: 8-queens problem

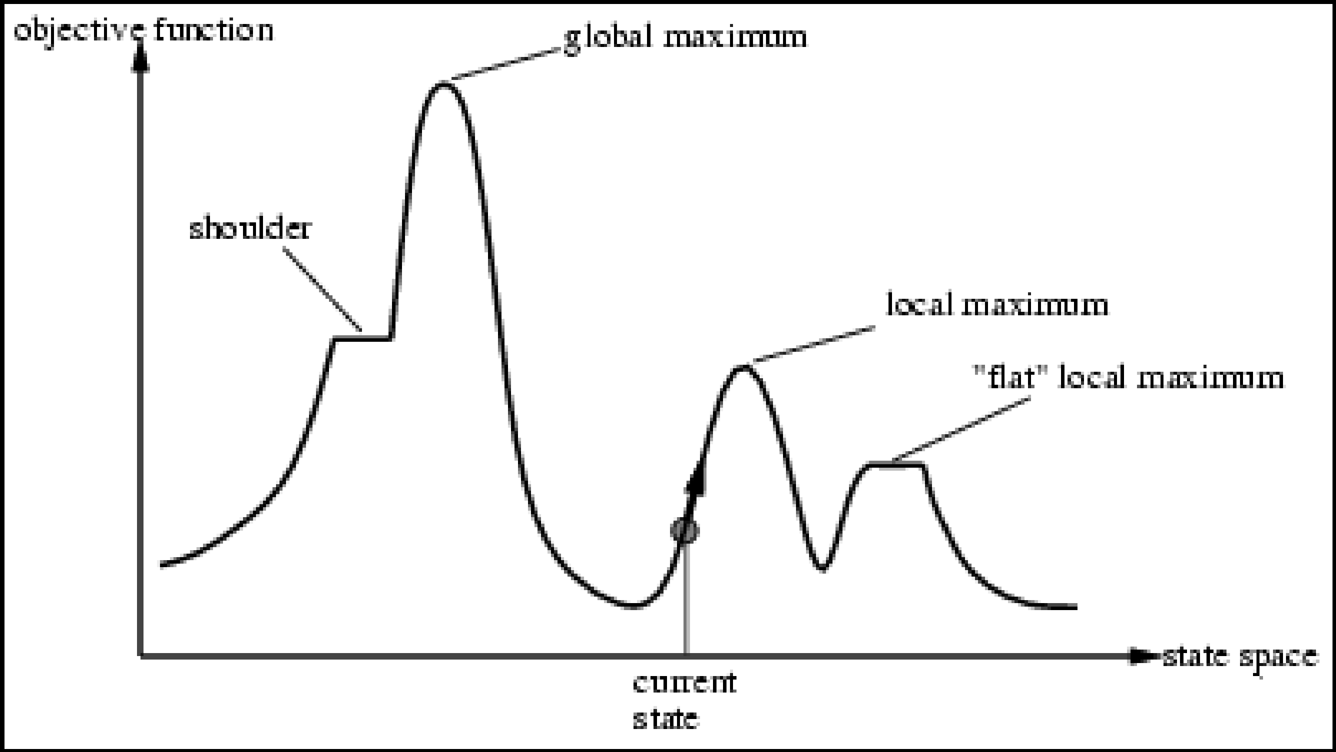
- 5 steps later…



- A **local minimum** with *h = 1* (*a* common problem with hill climbing)

Drawbacks of hill climbing

- Problem: depending on initial state, can get stuck in local maxima
  - I.e. where you end depends on where you start



Hill Climbing with Local Minima: Try, try again

- Run algorithm some number of times and return the best solution
  - Initial start location is usually chosen randomly
- If you run it “enough” times, will get answer (in the limit)

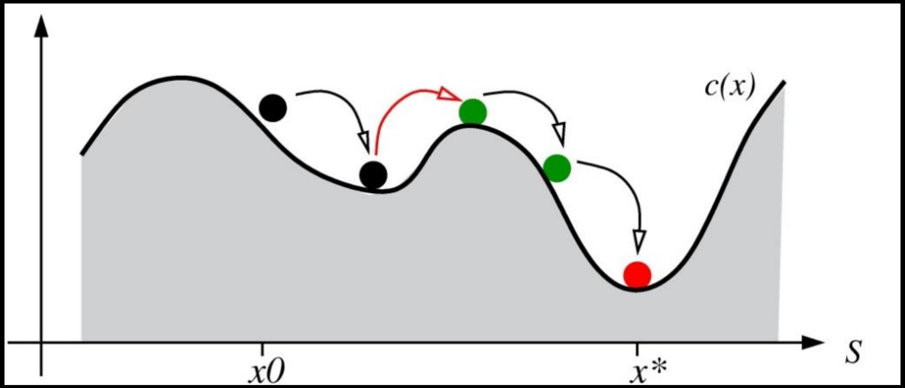
- Drawback: takes lots of time, never sure when to terminate

Simulated Annealing

Simulated annealing

- Hill climbing problems
  - Gets stuck on plateaus
  - Returns sub-optimal solutions (local minima)

- Simulated annealing main idea: explicitly inject variability into the search process



Properties of simulated annealing

- More variability at the beginning of search
  - Since you have little confidence you're in right place
- Variability decreases over time
  - Don't want to move away from a good solution
- Probability of picking a move is related to how good it is
  - No decrease or slight decrease are more likely than major decreases in quality

Simulated annealing implementation

- Create a "temperature schedule" for how variability changes as we iterate

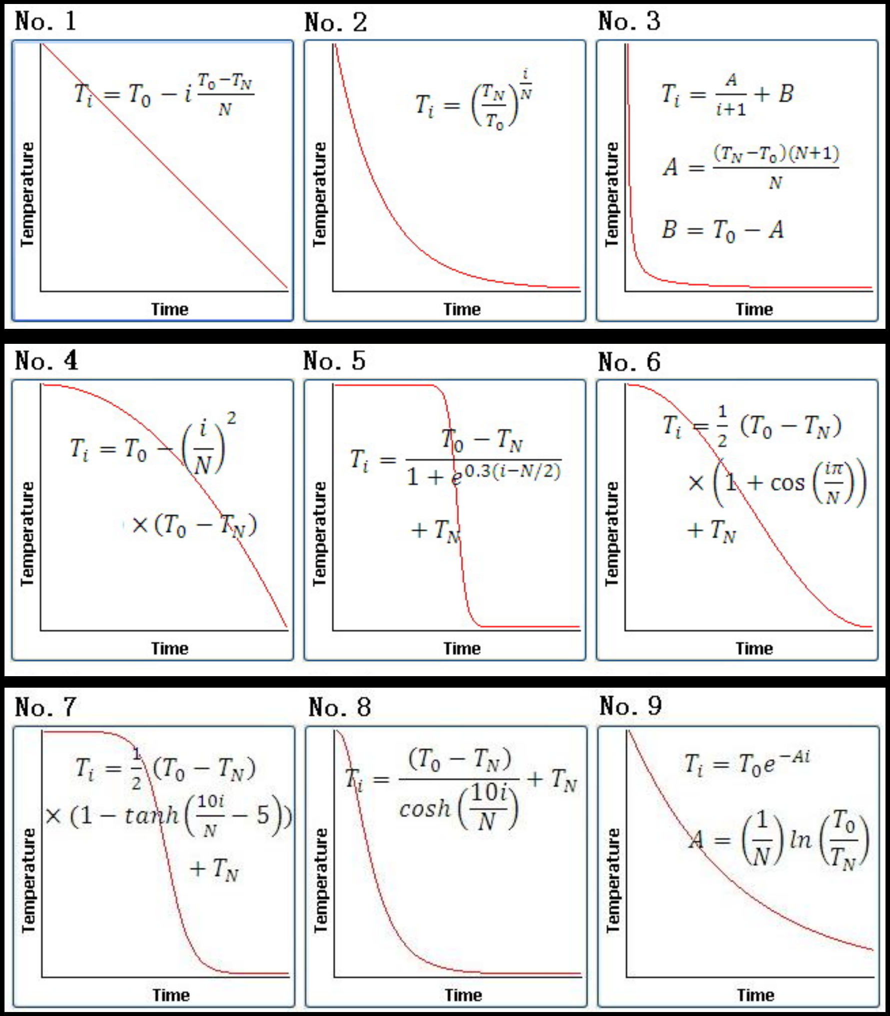
```
function SIMULATED-ANNEALING(problem, schedule) returns a solution state
    inputs: problem, a problem
            schedule, a mapping from time to "temperature"
    local variables: current, a node
                     next, a node
                     T, a "temperature" controlling prob. of downward 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)
```

Much faster than considering all neighbors (in high dimensions)

Temperature schedules

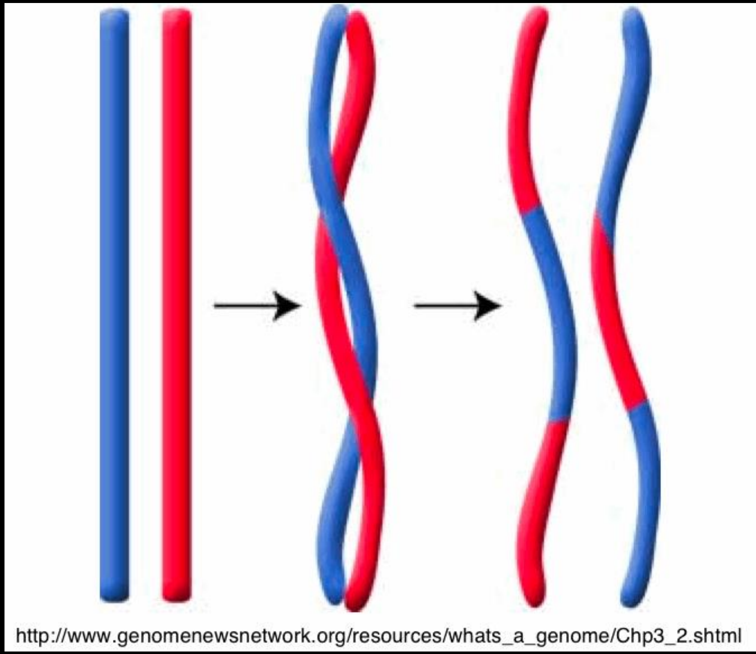
- Which one to use?



Genetic Algorithms

Genetic algorithms

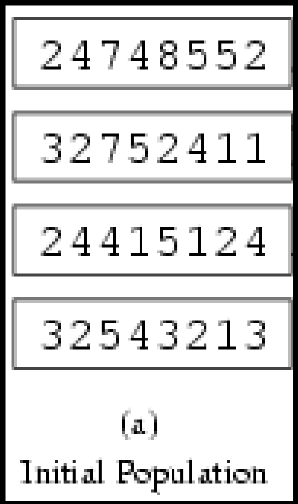
- Inspired (loosely) by the process of evolution in nature
- Operators:
  - Crossover: New states generated from two *parent* states
  - Mutation: Randomly change a component of a state

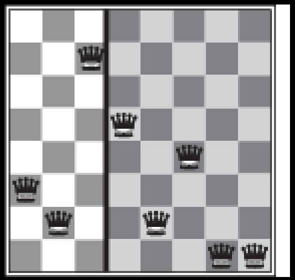


Genetic Algorithms

1. Initialize population (k random states)
2. Select a set of parents from the population for mating (based on fitness)
3. Generate children via crossover of parents
4. Mutation (add randomness to the children's variables)
5. Evaluate fitness of children
6. Replace worst parents with the children
7. Go to step 2

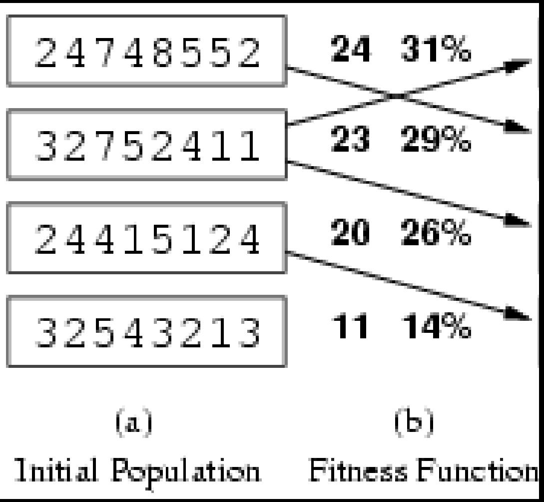
Genetic algorithms





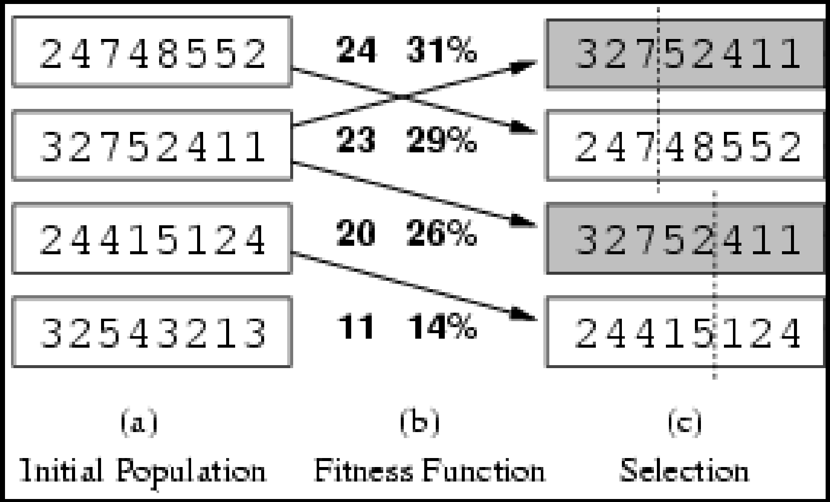
32752411

Genetic algorithms



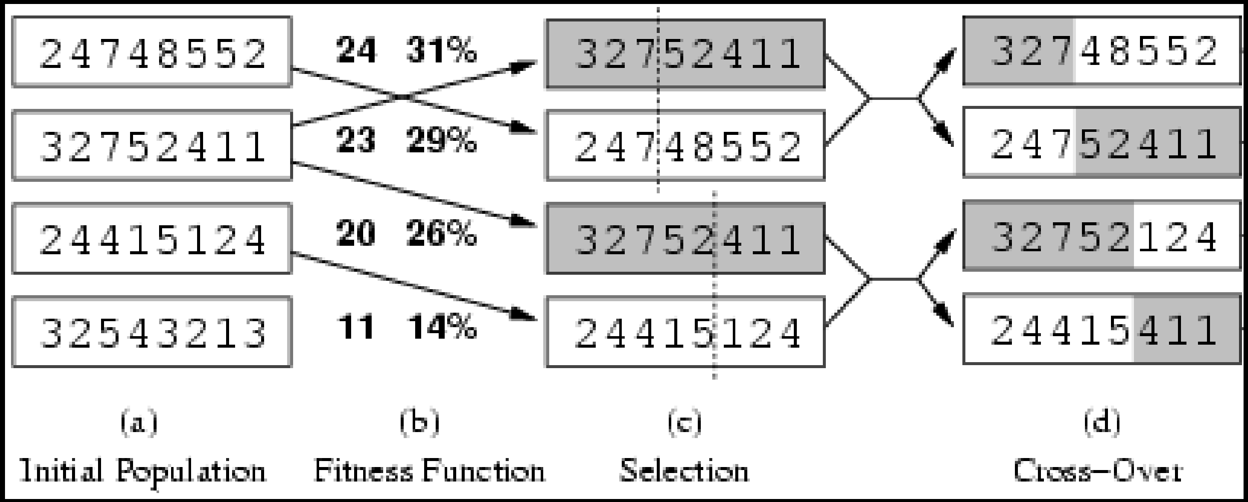
- Fitness function: number of *non-attacking* pairs of queens (min = 0, max = 8 × 7/2 = 28)
- Compute probability of being selected from population:
  - 24/(24+23+20+11) = 31%
  - 23/(24+23+20+11) = 29%
  - … etc.

Genetic algorithms

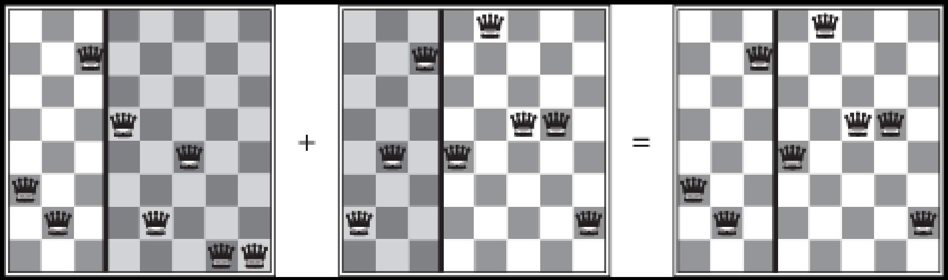


Select individuals based on probabilities

Genetic algorithms

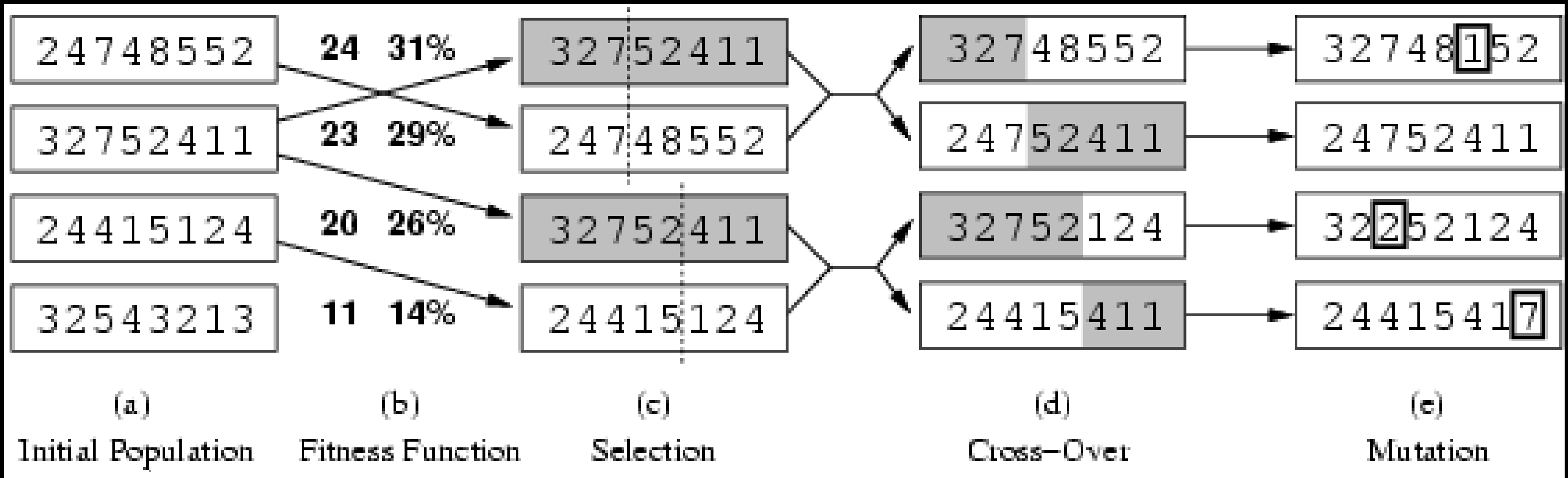


Select individuals based on probabilities



Crossover from the top two parents.

Genetic algorithms



Genetic Algorithms

1. Initialize population (k random states)
2. Calculate fitness function
3. Select pairs for crossover
4. Apply mutation
5. Evaluate fitness of children
6. From the resulting population of 2*k individuals, probabilistically pick k of the best.
7. Repeat.

The Power of Evolutionary Algorithms

- Human-competitive genetic programming:
  - http://www.genetic-programming.com/humancompetitive.html

- Evolving robot morphology and walking:
  - http://www.uvm.edu/~uvmpr/?Page=news&storyID=11482&category=ucommfeatureb

- Evolving soft robots:
  - https://www.youtube.com/watch?v=z9ptOeByLA4

Summary

- Can represent optimization problems as graphs where solutions are nodes and adjacency of solutions determines edges
- Local search (like hill-climbing) often finds local minima
- Can inject randomness to avoid getting stuck in local minima
  - Simulated Annealing
  - Genetic Algorithms

Homework

- ***