ENCS3340 - Artificial Intelligence

Local Search and Optimization

Introduction

- So far we considered Global Search - methods that systematically explore the search space, possibly using principled pruning (e.g., A*).
- Current best such algorithm can handle search spaces of up to 10^{100} states / around 500 binary variables
- What if we have much larger search spaces?
    - Search spaces for some real-world problems might be much larger - e.g. $10^{30,000}$ states. Also some spaces are continuous!
- A completely different kind of method is called for: Local search methods or Iterative Improvement Methods.

Local Search and Optimization

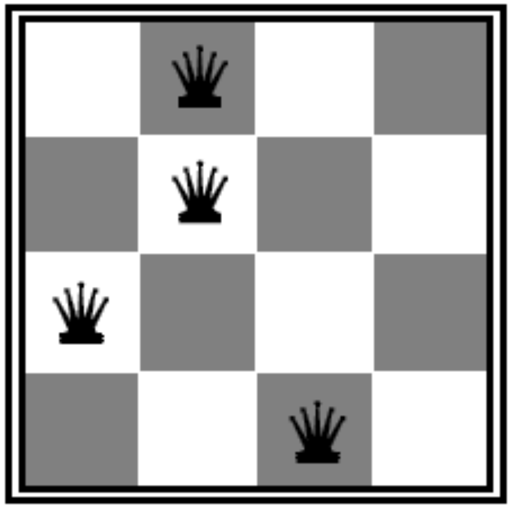
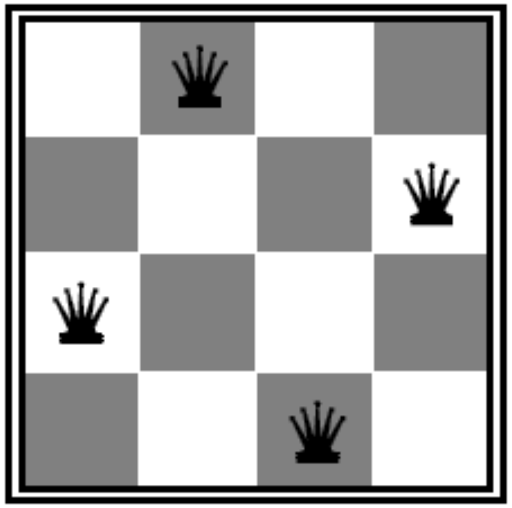
- for some problem classes, it is sufficient to find a solution
  - the actual path to the solution is not relevant
- memory requirements can be dramatically reduced
  - modifying the current state to advance the search
  - only information about the current state is kept
  - all information about previous states is discarded including paths to nodes in the search tree
  - may have serious consequences
    - no path information kept => states may be re-visited (loops)
    - impacts completeness, optimality
- since only information about the current state is kept, such methods are called local

Example: Traveling salesman problem

- Find the shortest tour connecting a given set of cities
- State space: all possible tours
- Objective function: length of tour

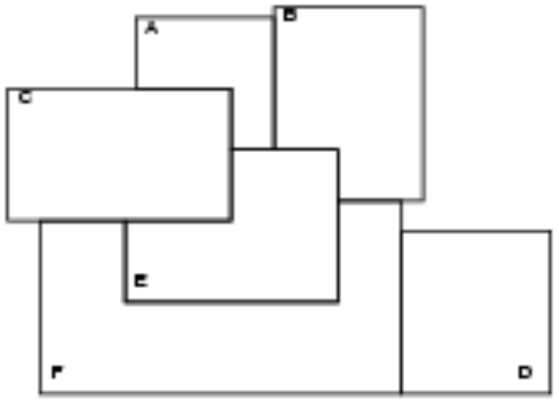
Example: *n*-queens problem

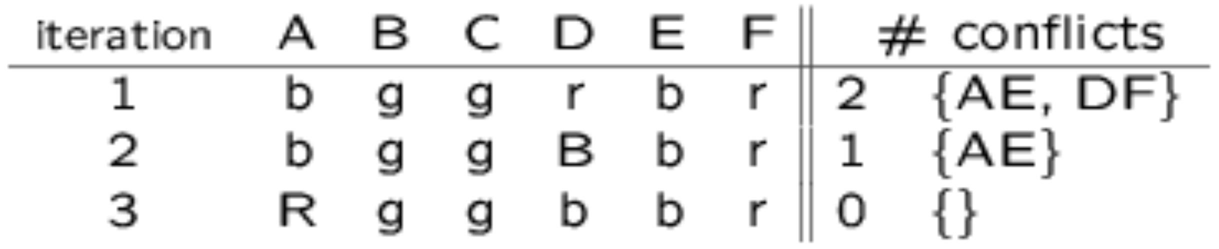
- Put n queens on an n × n board with no two queens on the same row, column, or diagonal
- State space: all possible n-queen configurations
- What's the objective function?
  - Number of pairwise conflicts

Example: Graph Coloring

- Start with random coloring of nodes
- Change color of one node to reduce # of conflicts
- Repeat 2





iteration	A	B	C	D	E	F	# conflicts	
1	b	g	g	r	b	r	2	{AE, DF}
2	b	g	g	B	b	r	1	{AE}
3	R	g	g	b	b	r	0	{}

Local Search

Key idea (surprisingly simple):

- Select (random) initial state (initial guess at solution)
- Make local modification to improve current state
- Repeat Step 2 until goal state found (or out of time) (or close enough to an optimal state)

Requirements:

- generate an initial (often random; probably-not-optimal or even valid) guess
- evaluate quality of guess
- move to other states (well-defined neighborhood function) choosing better states

. . . and do these operations quickly

. . . and **don't save paths followed**

Local Search

Algorithms

- Hill Climbing
- Simulated Annealing
- Local Beam Search
- Genetics algorithms

Hill-Climbing Search

- continually moves uphill
    - increasing value of the evaluation (objective) function
    - gradient descent search is a variation that moves downhill

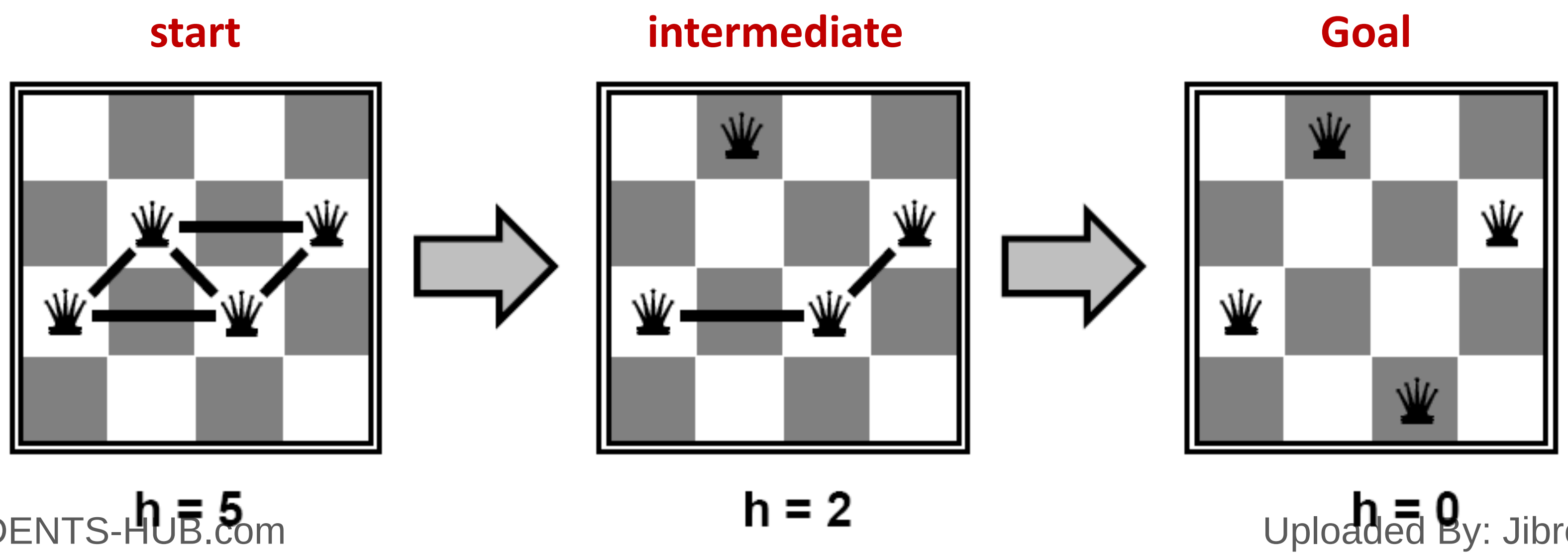
- very simple strategy with low space requirements
    - stores only the state and its evaluation (height), no search tree

Hill-climbing search

- Initialize current to starting state
- Loop:
    - Let next = highest-valued successor of current
    - If value(next) < value(current) return current
    - Else let current = next
- Variants: choose first better successor, randomly choose among better successors

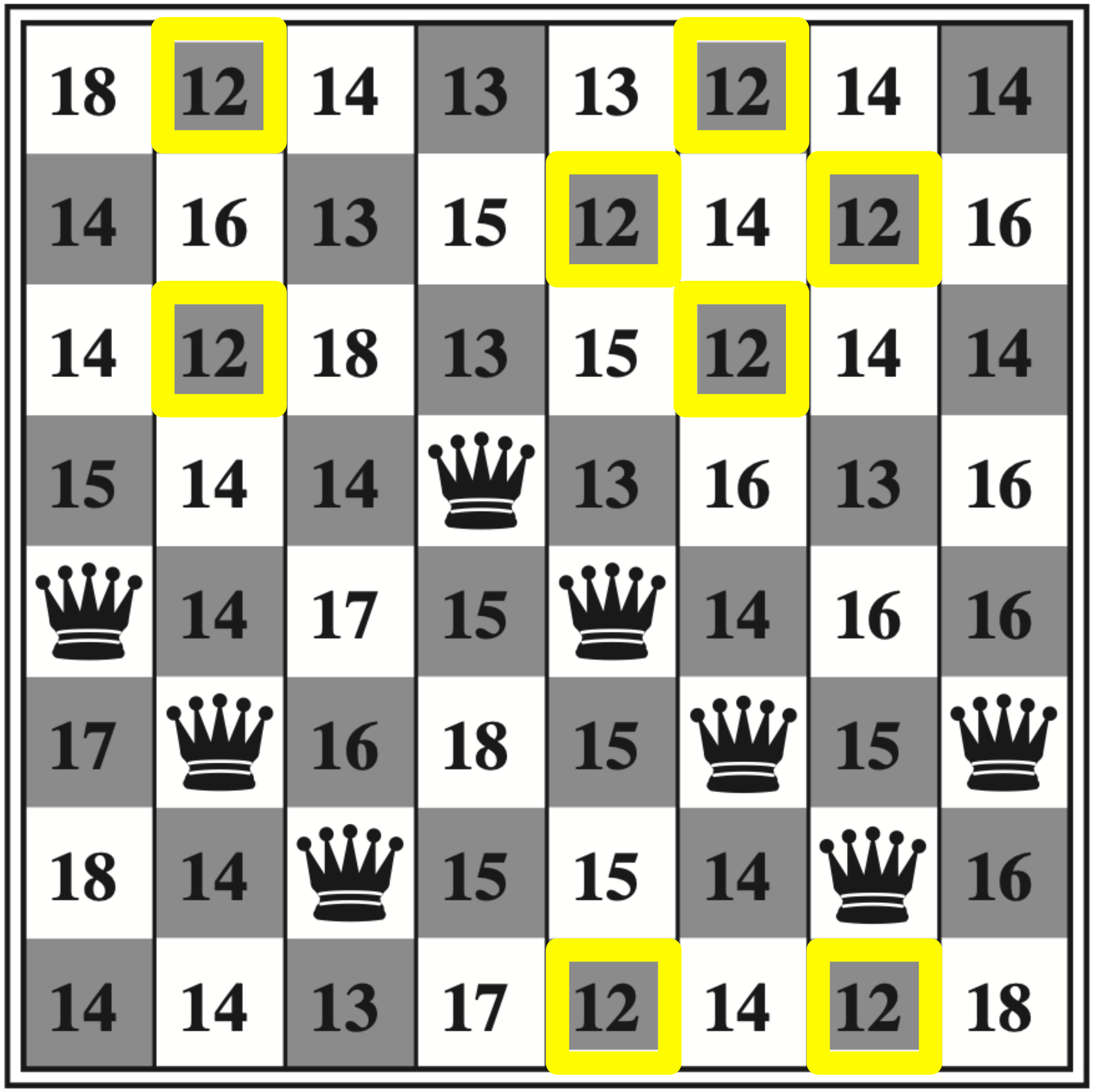
Example: n-queens problem

- Put n queens on an n × n board with no two queens on the same row, column, or diagonal
- State space: all possible n-queen configurations
- Objective function: number of pairwise conflicts
- What's a possible local improvement strategy?
  - Move one queen within its column to reduce conflicts



Example: 8-queens

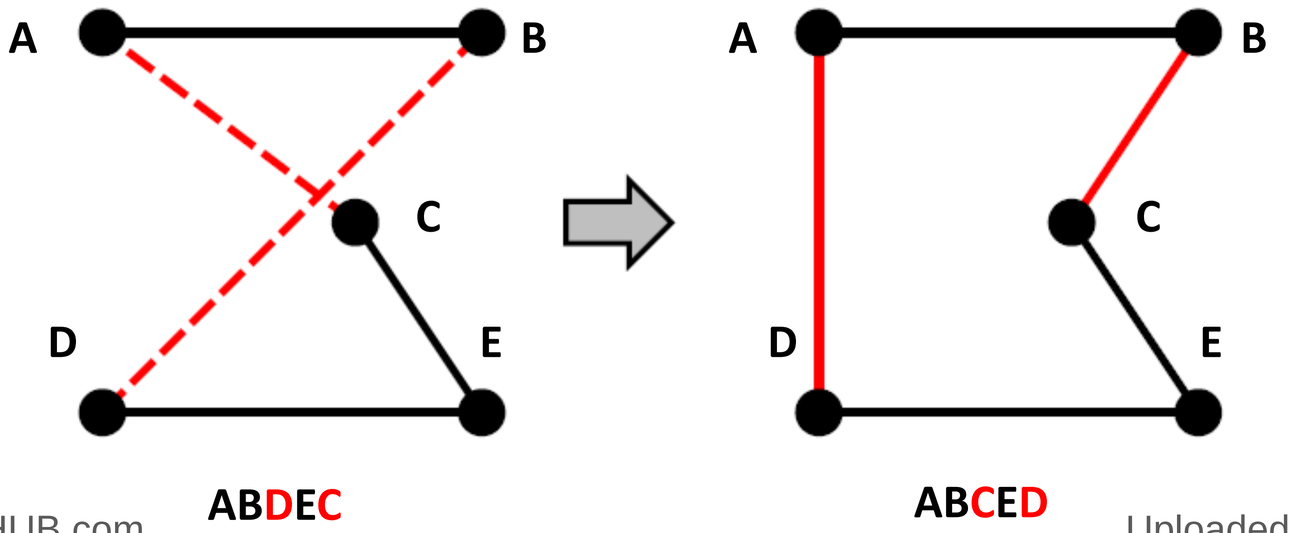
- h = number of pairs of queens that are attacking each other
- h = 17 for the state below
- best moves are marked (h=12)



18	12	14	13	13	12	14	14
14	16	13	15	12	14	12	16
14	12	18	13	15	12	14	14
15	14	14	♛	13	16	13	16
♛	14	17	15	♛	14	16	16
17	♛	16	18	15	♛	15	♛
18	14	♛	15	15	14	♛	16
14	14	13	17	12	14	12	18

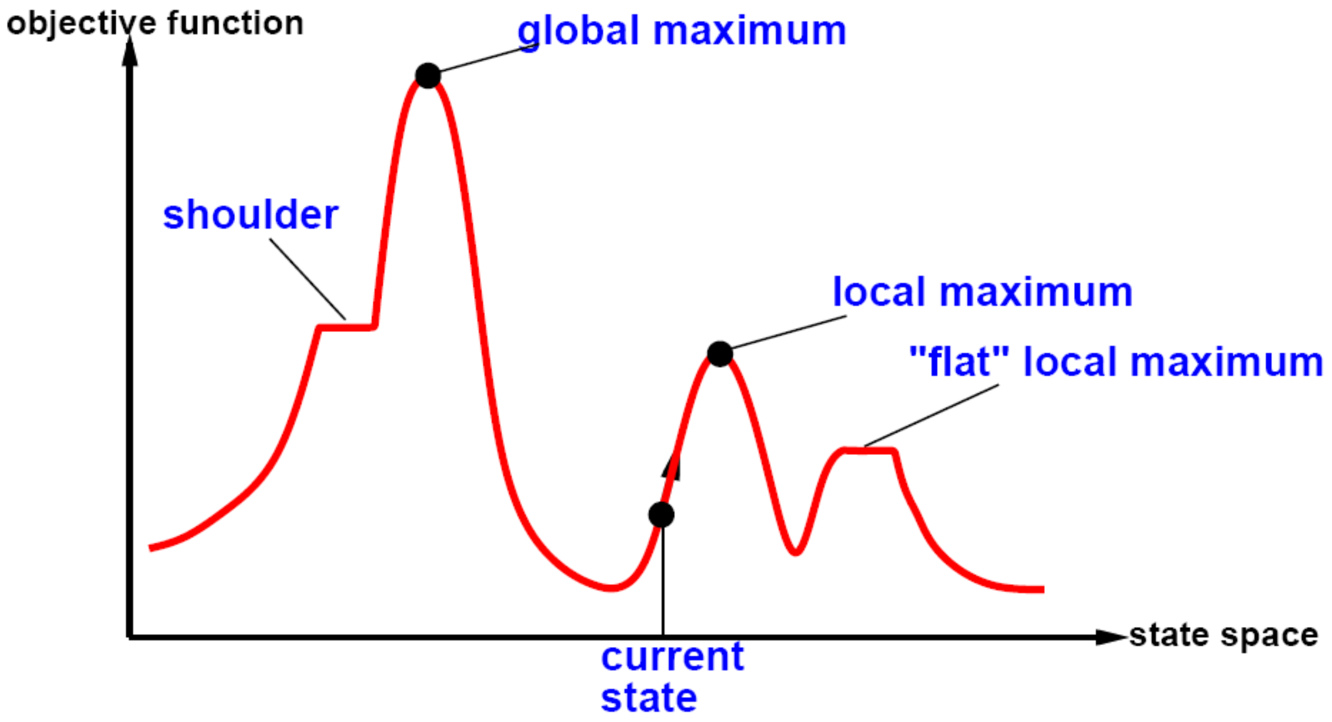
Example: Traveling Salesman Problem

- Find the shortest tour connecting n cities
- State space: all possible tours
- Objective function: length of tour
- What's a possible local improvement strategy?
  - Start with any complete tour, perform pairwise exchanges



Hill-climbing search

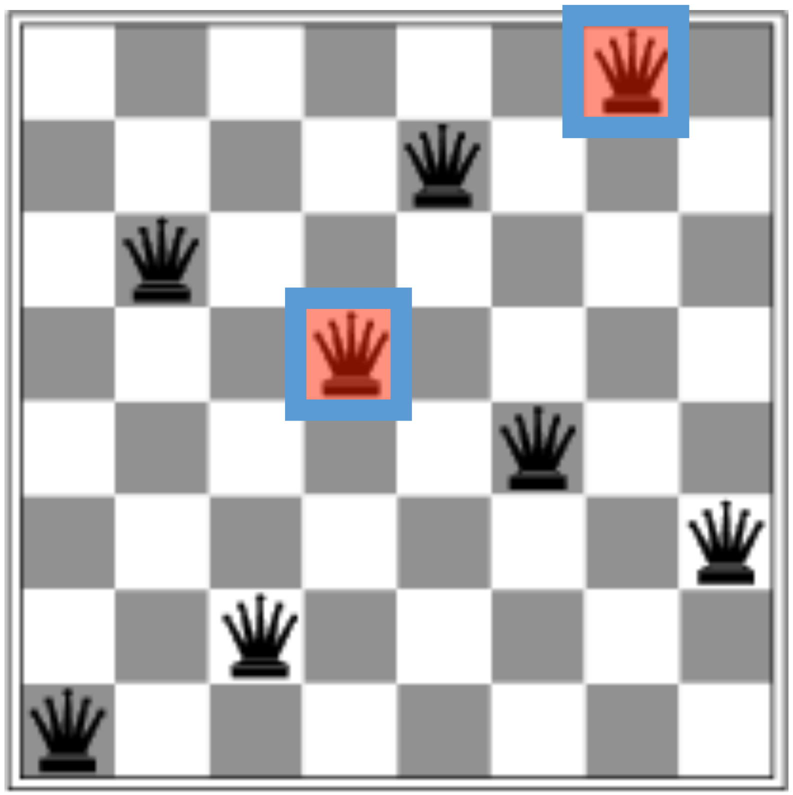
- Problem: depending on initial state, can get stuck in local maxima



- How to escape local maxima?
  - Random restart hill-climbing

Hill-climbing search: 8-queens problem

- A local minimum with h = 1
- one pair of queens attacking each other



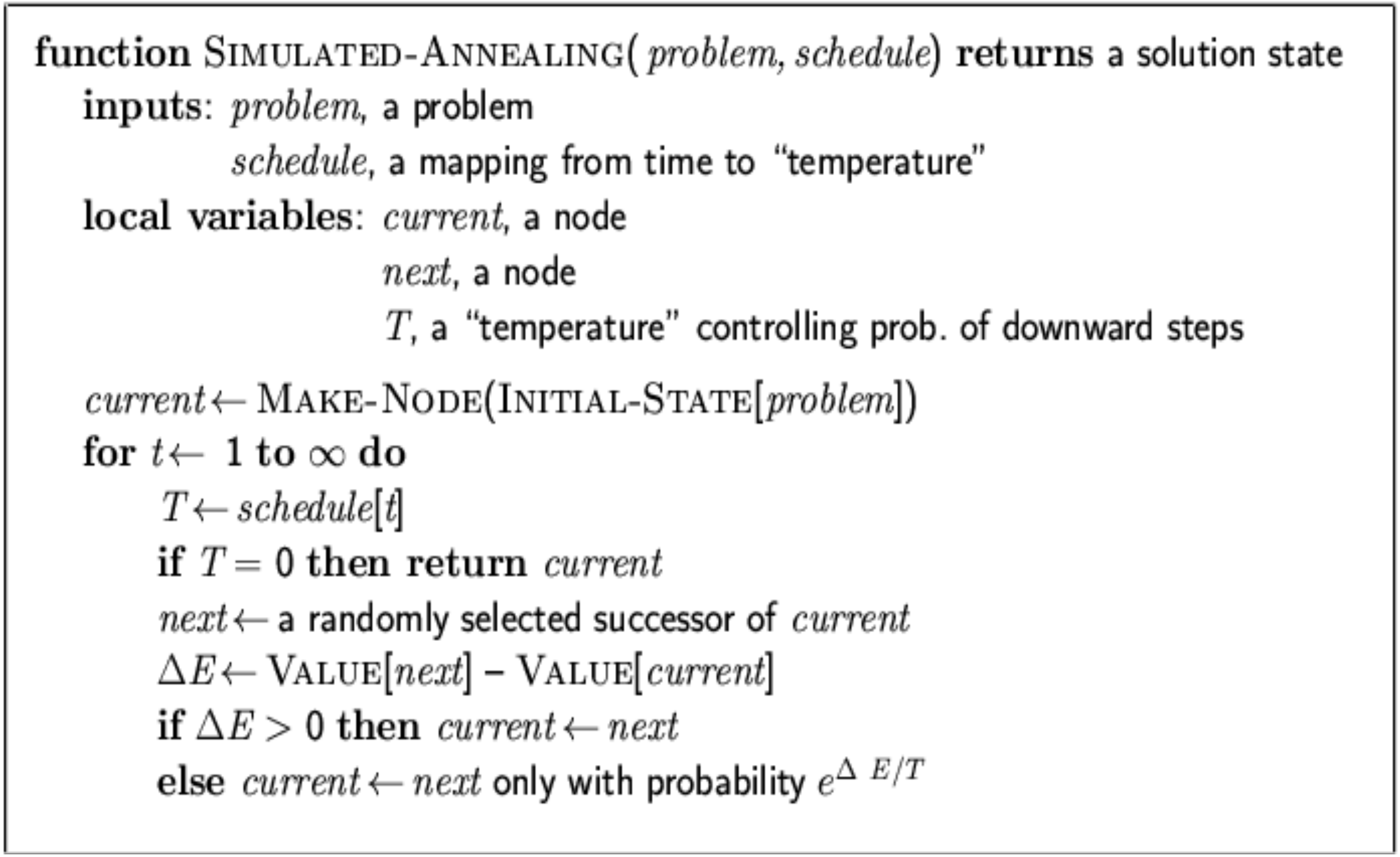
Hill-climbing variations

- Random-walk hill-climbing
  - At each step do one of the two
    - Greedy: With prob p move to the neighbor with largest value
    - Random: With prob 1-p move to a random neighbor
- Random-restart hill-climbing
  - If at first you don’t succeed, try, try again!
  - Different variations
    - For each restart: run until termination vs. run for a fixed time
    - Run a fixed number of restarts or run indefinitely
- Hill-climbing with both
  - At each step do one of the three
    - Greedy: move to the neighbor with largest value
    - Random Walk: move to a random neighbor
    - Random Restart: Resample a new current state

Simulated Annealing

- similar to hill-climbing, but some down-hill movement
  - occasional random move instead of the best move
  - depends on two parameters
    - ΔE: energy difference between moves
    - T: temperature (temperature is slowly lowered, making bad moves less likely)
- analogy to annealing
  - gradual cooling of a liquid until it freezes
  - also used in metal processing to increase strength
- will find the global optimum if the temperature is lowered slowly enough
  - may take an infinite number of annealing steps
- applied to routing and scheduling problems
  - VLSI layout, scheduling

Simulated Annealing



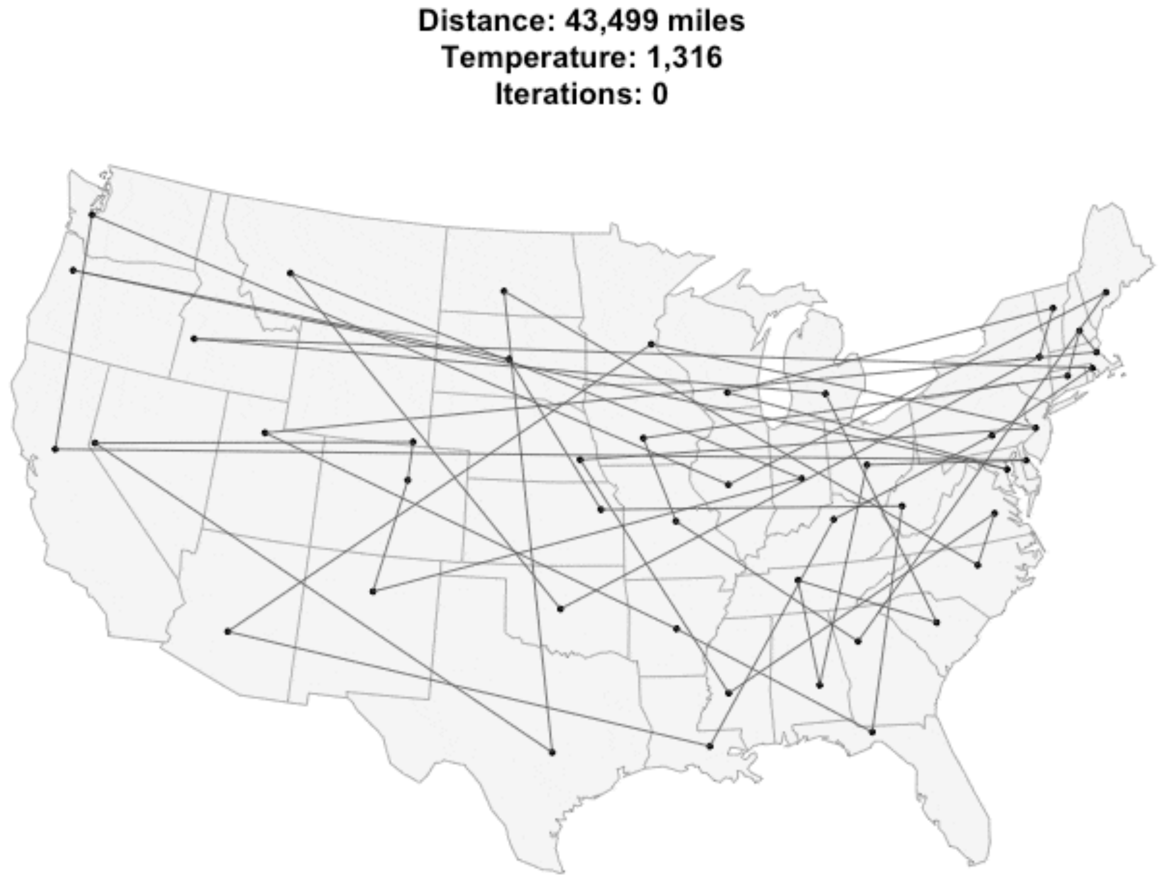
```
function SIMULATED-ANNEALING(problem, schedule) returns a solution state
    inputs: problem, a problem
            schedule, a mapping from time to "temperature"
    local variables: current, a node
                     next, a node
                     T, a "temperature" controlling prob. of downward 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 $e^{\Delta E/T}$
```

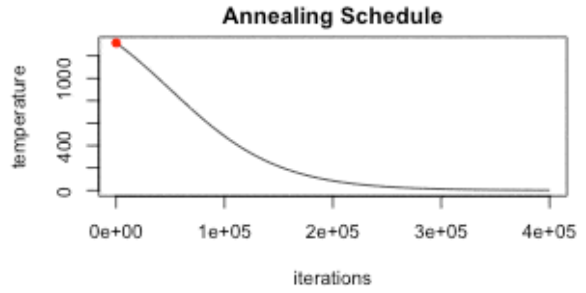
Example of Simulated Annealing

- Traveling Salesman Problem (TSP)
  - Given 6 cities and the traveling cost between any two cities
  - A salesman need to start from city 1 and travel all other cities then back to city 1
  - Minimize the total traveling cost

Example: SA for traveling salesman

- Solution representation
  - An integer list, i.e., (1,4,2,3,6,5)
- Search mechanism
  - Swap any two integers (except for the first one)
  - (1,4,2,3,6,5) -> (1,4,3,2,6,5)
- Cost function
  - Path length

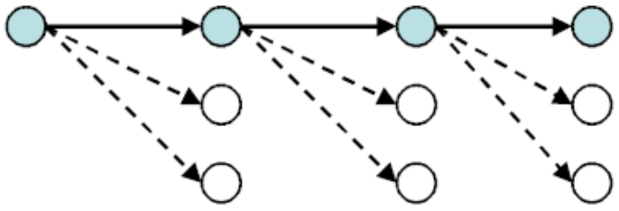


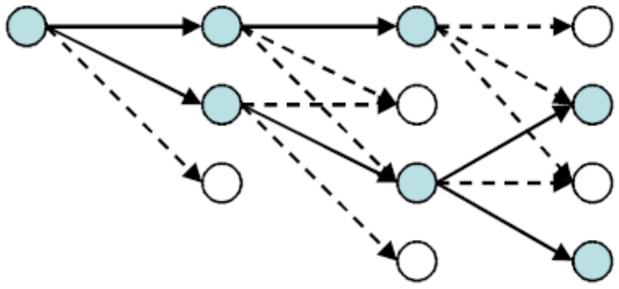


http://toddwschneider.com/posts/traveling-salesman-with-simulated-annealing-r-and-shiny/

Local Beam Search

- Start with k randomly generated states
- At each iteration, all the successors of all k states are generated
- If any one is a goal state, stop; else select the k best successors from the complete list and repeat


Greedy search


Beam search

Genetic Algorithms (GAs)

- variation of stochastic beam search
- A genetic algorithm maintains a population of candidate solutions for the problem at hand, and makes it evolve by iteratively applying a set of stochastic operators
- successor states are generated as variations of two parent states, not only one
- Genetic Algorithms follow the idea of SURVIVAL OF THE FITTEST - Better and better solutions evolve from previous generations until a near optimal solution is obtained.
- mutation provides an additional random element

High-level Algorithm

- Randomly generate an initial population[chromosomes]
- Evaluate the fitness of population [evaluation Function]
- Select parents and “reproduce” the next generation [crossover]
- Replace the old generation with the new generation: may keep some of the old (say by keeping based on Evaluation Function]
- Repeat step 2 though 4 till iteration N

GA Terminology

- Population
  - set of k randomly generated states
- Generation
  - population at a point in time
- Individual [chromosome]
  - one element from the population
  - described as a string over a finite alphabet [string of genes]
    - binary, letters, digits
    - consistent for the whole population
- Fitness function
  - evaluation function in search terminology
  - higher values lead to better chances for reproduction

Stochastic Operators

Cross-over

- decomposes two distinct solutions and then randomly mixes their parts to form novel solutions

Mutation

- randomly perturbs [changes] a candidate solution
  - Flip a 1 to 0 or exchange cities in TSP

Genetic Algorithms

Before we can apply Genetic Algorithm to a problem, we need to answer:

- How is an individual represented?
- What is the fitness function?
- How are individuals selected?
- How do individuals reproduce?

Representing an Individual

- An individual is data structure representing the “genetic structure” of a possible solution.
- Genetic structure consists of an alphabet (usually 0,1)
- Binary Encoding
  - Most Common – string of bits, 0 or 1.

    Chrom: A = 1 0 1 1 0 0 1 0 1 1

    Chrom: B = 1 1 1 1 1 1 0 0 0 0
  - Gives you many possibilities
  - Example Problem: Knapsack problem
  - The problem: there are things with given value and size. The knapsack has given capacity. Select things to maximize the values.
  - Encoding: Each bit says, if the corresponding thing is in the knapsack

Representing an Individual

- Permutation Encoding
  - Used in “ordering problems”
  - Every chromosome is a string of numbers, which represents number is a sequence.

    Chrom A: 1 5 3 2 6 4 7 9 8

    Chrom B: 8 5 7 7 2 3 1 4 9
  - Example: Travelling salesman problem
  - The problem: cities that must be visited.
  - Encoding says order of cities in which salesman will visit.

Generating New Generations

- Reproduction- Through reproduction, genetic algorithms produce new generations of improved solutions by selecting parents with higher fitness ratings or by giving such parents a greater probability of being contributors and by using random selection

- Crossover- Many genetic algorithms use strings of binary symbols for chromosomes, as in our Knapsack example, to represent solutions. Crossover means choosing a random position in the string (say, after 2 digits) and exchanging the segments either to the right or to the left of this point with another string partitioned similarly to produce two new off spring.

- Mutation- Mutation is an arbitrary change in a situation. Sometimes it is used to prevent the algorithm from getting stuck. The procedure changes a 1 to a 0 to a 1 instead of duplicating them. This change occurs with a very low probability (say 1 in 1000)

Crossover Example 1

- Parent A 011011
- Parent B 101100

- “Mate the parents by splitting each number as shown between the second and third digits (position is randomly selected)

01*1011 10*1100

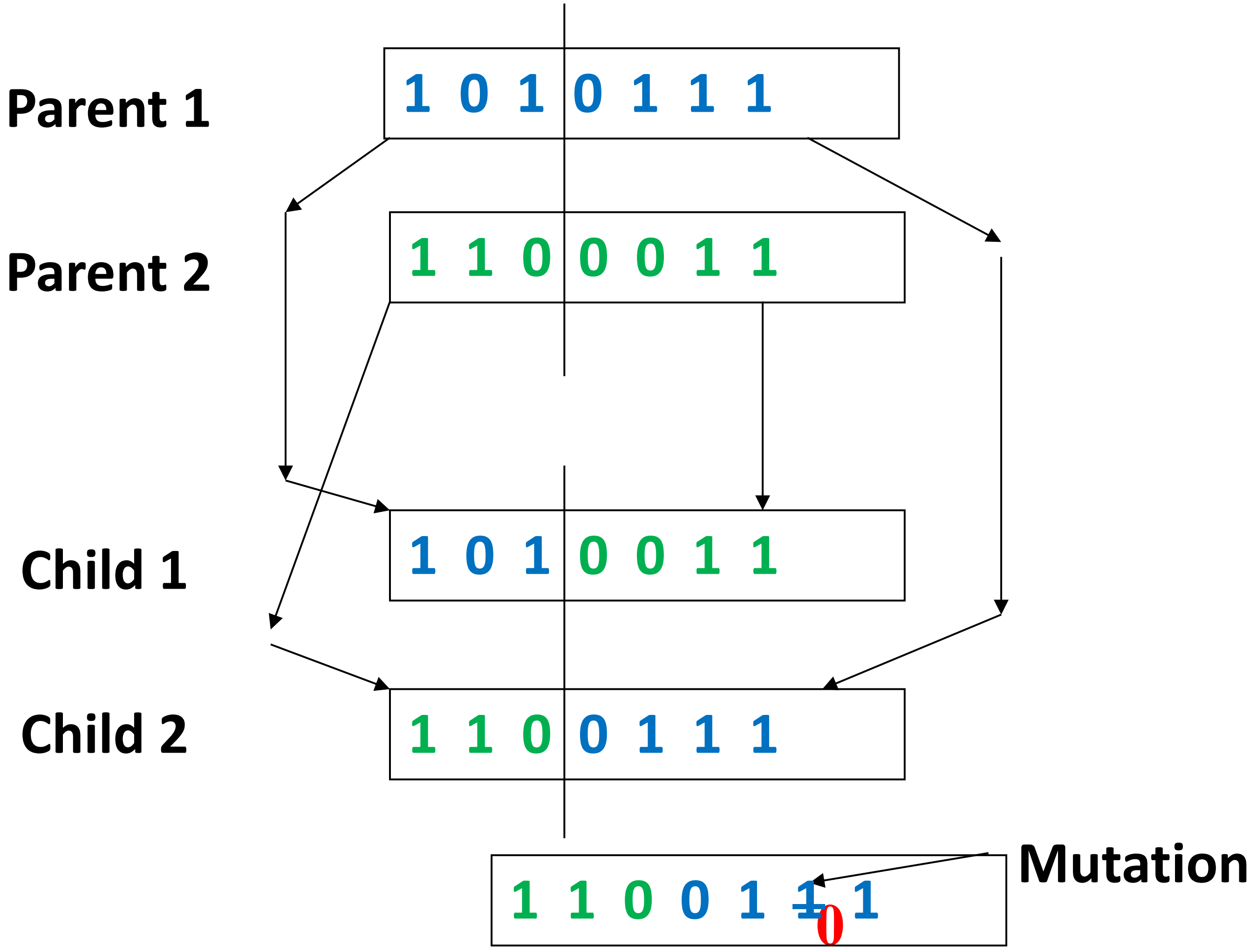
- Now combine the first digits of A with the last digits of B, and the first digits of B with the last digits of A

- This gives you two new offspring

011100 101011

- If these new solutions, or offspring, are better solutions than the parent solutions, the system will keep these as more optimal solutions and they will become parents. This is repeated until some condition (for example number of populations or improvement of the best solution) is satisfied.

Genetic Algorithm Operators: Mutation and Crossover



Selection Criteria

- Fitness proportionate selection, rank selection methods.

- Fitness proportionate – each individual, I, has the probability

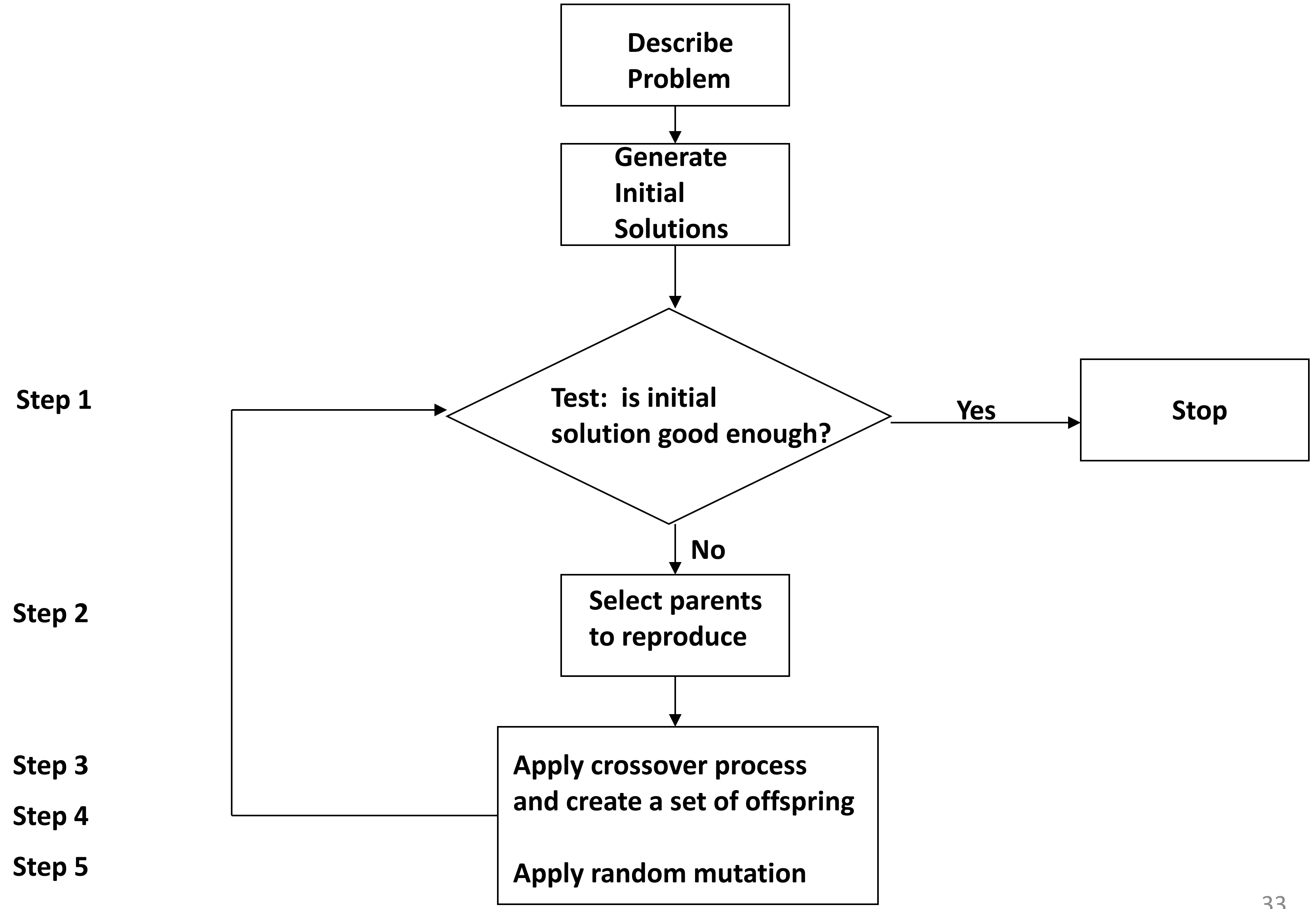
$$\text{Fitness(I)} / \textstyle\sum_j \text{Fitness(j)}$$

  where Fitness(I) is the fitness function value for individual I.

  - Represents a rank of the “representation”
  - It is usually a real number.
  - E.g. the length of the route in the traveling salesperson problem is a good measure, because the shorter the route, the better the solution

- Rank selection – sorts individual by fitness and the probability that an individual will be selected is proportional to its rank in this sorted list.

Flow Diagram of the Genetic Algorithm Process


Describe Problem
Generate Initial Solutions
Test: is initial solution good enough?
Yes
Stop
No
Select parents to reproduce
Apply crossover process and create a set of offspring
Apply random mutation
Step 1
Step 2
Step 3
Step 4
Step 5

Example: The Knapsack Problem

- You are going on an overnight hike and have a number of items that you could take along.
- Each item has a weight (in pounds) and a benefit or value to you on the hike(for measurements sake let's say, in US dollars), and you can take one of each item at most.
- There is a capacity limit on the weight you can carry (constraint).
- This problem only illustrates one constraint, but in reality there could be many constraints including volume, time, etc.

GA Example: The Knapsack Problem

Item:	1	2	3	4	5	6	7
Benefit:	5	8	3	2	7	9	4
Weight:	7	8	4	10	4	6	4

- Knapsack holds a maximum of 22 pounds
- Fill it to get the maximum benefit
- Solutions take the form of a string of 1's and 0's. Also known as strings of genes called Chromosomes
    - 0101010
    - 1101100
    - 0100111

Example: The Knapsack Problem

- We represent a solution as a string of seven 1s and 0s and the fitness function as the total benefit, which is the sum of the gene values in a string solution times their representative benefit coefficient.

- The method generates a set of random solutions (initial parents), uses total benefit as the fitness function and selects the parents randomly to create generations of offspring by crossover and mutation.

- Possible solutions generated by the system using Reproduction, Crossover, or Mutations
  - 0101010
  - 1101100
  - 0100110

Knapsack Example: Solution 1

Item	1	2	3	4	5	6	7
Solution	**0**	**1**	**0**	**1**	**0**	**1**	**0**
Benefit	5	8	3	2	7	9	4
Weight	7	8	4	10	4	6	4

- Benefit 8 + 2 + 9 = 19
- Weight 8 + 10 + 6 = 24

Knapsack Example: Solution 2

Item	1	2	3	4	5	6	7
Solution	**1**	**1**	**0**	**0**	**1**	**0**	**0**
Benefit	5	8	3	2	7	9	4
Weight	7	8	4	10	4	6	4

- Benefit 5 + 8 + 7 = 20
- Weight 7 + 8 + 4 = 19

Knapsack Example: Solution 3

Item	1	2	3	4	5	6	7
Solution	**0**	**1**	**0**	**0**	**1**	**1**	**1**
Benefit	5	8	3	2	7	9	4
Weight	7	8	4	10	4	6	4

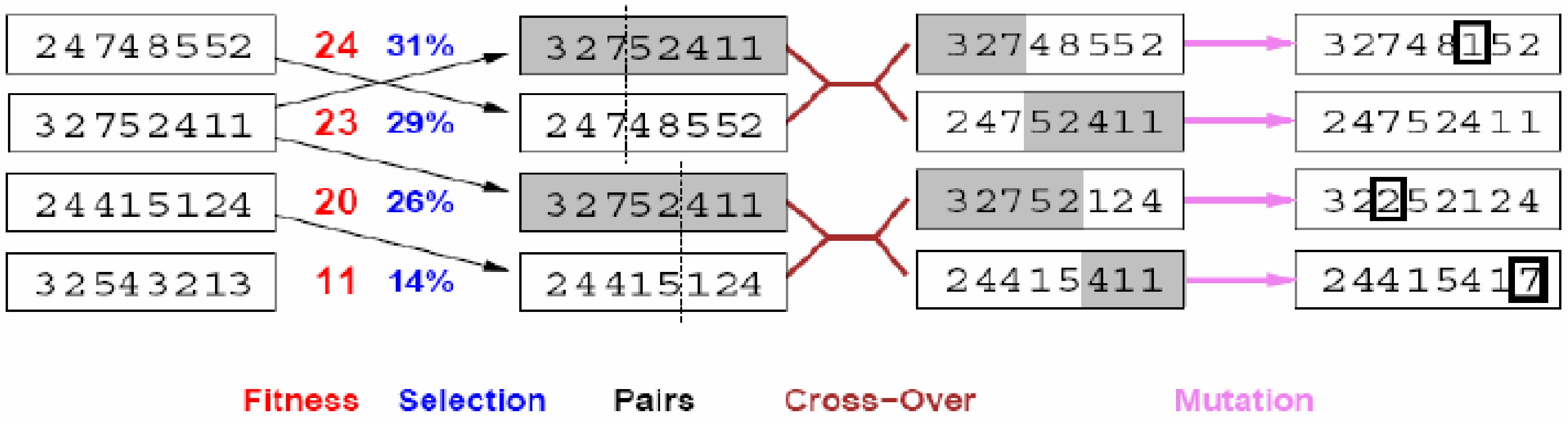
- Benefit 8 + 7 + 9 + 4 = 28
- Weight 8 + 4 + 6 + 4 = 22

Knapsack Example

- Solution 3 is clearly the best solution and has met our conditions, therefore, item number 2, 5, 6, and 7 will be taken on the hiking trip. We will be able to get the most benefit out of these items while still having weight equal to 22 pounds.

- This is a simple example illustrating a genetic algorithm approach.

8 Queen Example



- Fitness function: number of non-attacking pairs of queens (min = 0, max = 8 × 7/2 = 28)
- 24/(24+23+20+11) = 31%
- 23/(24+23+20+11) = 29% etc