Genetic Algorithms

Tom Austin
San Jose State University

Introduction

- A Genetic Algorithm (GA) emulates biological evolution to solve a complex problem.
- GAs rely heavily on randomness. Instead of trying to solve the problem directly, they create random solutions and randomly mix them up until a good solution is found.

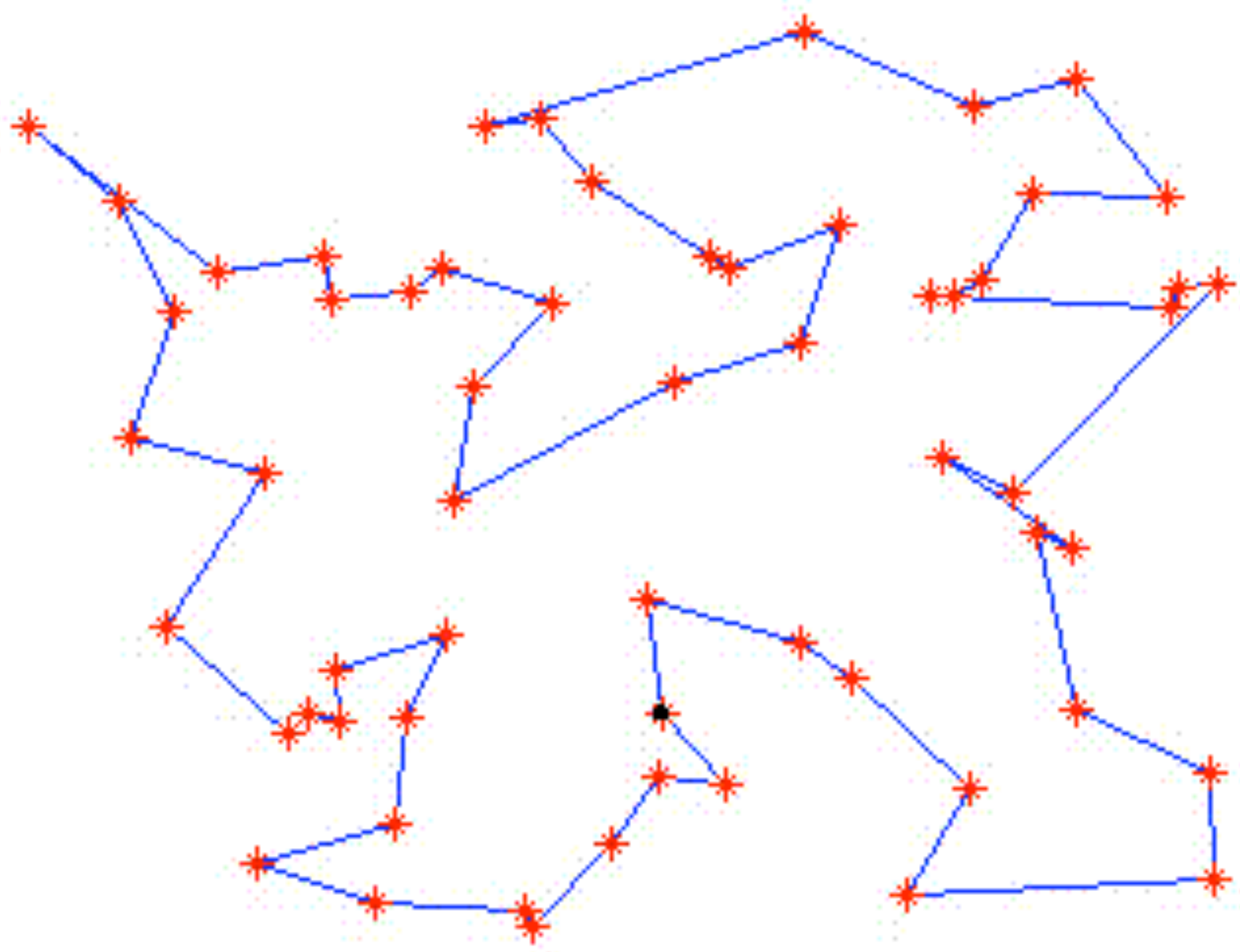
NP-Hard problems

- These are problems where no efficient algorithm is known to exist.
- Computing power is irrelevant – computers will **never** get fast enough to find solutions for more than the most trivial instances of these problems.
- Some famous problems: Traveling Salesman Problem, 0-1 Knapsack Problem.

Example:Traveling Salesman Problem (TSP)

- Given a collection of cities and the cost of travel between each pair of them, what is the cheapest way of visiting all of the cities?
- This problem is NP-hard – we will never be able to find the optimal solution. (Finding the optimal solution would take N!).

From *http://en.wikipedia.org/wiki/Travelling_salesman_problem*



Heuristics to the rescue

- A heuristic algorithm does not try to find the optimal solution. Instead, it attempts to find a good solution in a reasonable running time.
- Example: Greedy algorithm
  - Visit the nearest city each time (TSP problem).
  - Does not give a good solution for some problem instances.
- GAs are also heuristic algorithms. Unlike greedy solutions, they can be tuned to better solve different problem instances.

GA Process

1. Randomly generate initial population
2. Until a solution is found:
    1. Score each individual in the population
    2. Randomly select individuals as survivors
    3. Perform crossover on survivors
    4. Perform mutation on new generation (small chance)
3. Report the best solution

0-1 Knapsack Problem

- This is another famous NP-hard problem.
- In this case, you want to select the items that will fit in your knapsack and which will maximize your total profit. Each item has a weight and a profit.

Sample Knapsack Problem

- Knapsack has a capacity of 80
- Available items:
  - A: profit=55 weight=20
  - B: profit=40 weight=18
  - C: profit=30 weight=16
  - D: profit=27 weight=16
  - E: profit=20 weight=12
  - F: profit=13 weight=8
  - G: profit=9 weight=6
  - H: profit=7 weight=5
  - I: profit=4 weight=3
  - J: profit=1 weight=1

Initial population

- This should be generated randomly.
- Solutions may be invalid, but fitness value should reflect this.
- Solutions are often represented as binary strings, but this is not required.

Sample initial population

Value:155 weight:79 items:[A, C, D, E, F, G, J]

Value:122 weight:55 items:[A, B, E, H]

Value:112 weight:49 items:[A, B, F, I]

Value:80 weight:44 items:[B, E, G, H, I]

Value:71 weight:31 items:[A, G, H]

Value:84 weight:38 items:[A, E, G]

Value:75 weight:38 items:[B, C, I, J]

Value:54 weight:27 items:[B, F, J]

Value:107 weight:64 items:[C, D, E, F, G, H, J]

Value:165 weight:78 items:[A, B, C, D, F]

Fitness function

- A fitness function is needed to score the solutions.
- This can be designed to either maximize profit or minimize cost.
- In Java, we can’t have independent functions, so you probably want to create a Scorer class.
- For the knapsack problem, scoring is easy. A knapsack’s fitness is the total profit of its contents (unless there are overfilled knapsacks.)

Roulette Wheel

- Each individual solution has a random chance of surviving.
- More fit solutions will have more slots on the the wheel.
- java.util.Random is your best friend here.

Wheel Distribution

[A, C, D, E, F, G, J]:****************
[B, E, G, H, I]:********
[A, B, F, I]:***********
[A, E, G]:*********
[B, C, I, J]:********
[A, B, E, H]:************
[B, F, J]:******
[C, D, E, F, G, H, J]:***********
[A, B, C, D, F]:*****************
[A, G, H]:*******

Crossover

Take two of the survivors and create two new solutions with characteristics of the old:

Father: [B, D]

Mother: [A, D, F, H]

Son: [A, D]

Daughter: [B, D, F, H]

Mutation

- Each new child should also have a small chance of a mutation. This is a slight modification to the child solution.
- For the Knapsack problem, this translates to adding or removing an item from the knapsack.

Before: [A, B, G, J]

After: [A, B, G, H, J]

When are we finished?

- This is up to you. Some possibilities:
  – The process has run for X generations.
  – The most fit individual has not changed for X generations.
  – The most fit individual has a fitness greater than F.
- When you have finished, return the most fit individual as your solution.

GA vs. Greedy results

- Best GA Solution obtained

  Value:167 weight:80 items:[A, B, C, E, F, G]
- Greedy Solution

  Value:166 weight:79 items:[A, B, C, D, F, J]

Some notes on GAs

- Allow time for tweaking the parameters at the end. Make it easy to configure:
  - Population size
  - Mutation chance
  - Run time
- Crossover often produces big jumps in fitness.
- Mutations tend to produce less healthy offspring, but paradoxically they help improve the overall health of the population.

Advanced GA techniques

- Elitism – Carry over some portion of the best solutions to the next generation.
- Variable operators – Create multiple types of crossovers and mutations. Track the health of the offspring they produce, and adjust their usage accordingly.
- Tribes – Create separate populations that only occasionally mix. This may help avoid converging on local maxima.

Multiple Sequence Alignment

- One practical application of genetic algorithms is the multiple sequence alignment problem.
- We need to align multiple DNA or protein sequences.
- ClustalW is the standard tool for this, so we have a baseline to compare against.

Sample generated alignment

GA Best Solution:	ClustalW Solution:
`ATTGCC-ATT`	`ATCTTCTT--`
`ATGGCC-ATT`	`ATCCAATTTT`
`ATCCAATTTT`	`ATT-------`
`ATCTTC-TT-`	`--GGCCAT--`
`ATT-------`	`ATGGCCATT-`
`--GGCC-AT-`	`ATTGCCATT-`
`ATTG------`	`------ATTG`
Fitness: -74.0	Fitness: -84.0