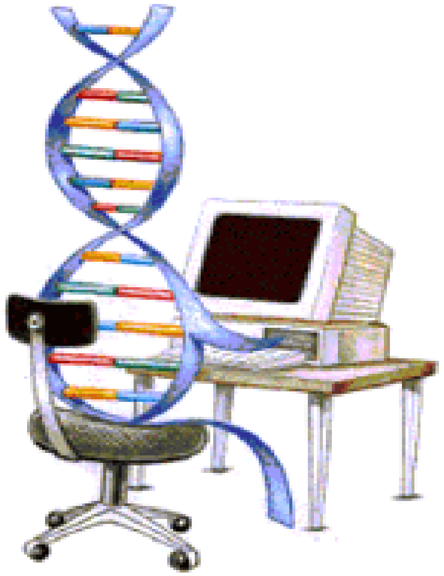
# Hello, world!

## Learning to optimize

*Lecture 01 by Marina Barsky*



# Example: Learning the best schedule

Setup:

- Six family members meet up in New York.
- They arrive on the same day and leave on the same day, and they want to share transportation to and from the airport.
- There are many possible flights per day for each family member: 8 inbound flights and 8 outbound flights
- Flights are arriving-leaving at different times. They also vary in price and in duration.

Task:

- Find optimum flight schedule for this group of people:
  - Minimize total cost
  - Minimize flight duration
  - Minimize waiting time at the airport

# Example: Learning the best schedule

Setup:

- Six family members meet up in New York.
- They arrive on the same day and leave on the same day, and they want to share transportation to and from the airport.
- There are many possible flights per day for each family member: 8 inbound flights and 8 outbound flights
- Flights are arriving-leaving at different times. They also vary in price and in duration.

Task:

- Find optimum flight schedule for this group of people:
  - Minimize total cost
  - Minimize flight duration
  - Minimize waiting time

We have too many possible combinations of flights:
$8^{12} = 68{,}719{,}476{,}736$

Exhaustive search over all these combinations is infeasible

**Can we set up an algorithm so that the machine would learn the best possible schedule on its own?**

## Search for a global minimum (maximum) of a function

- We have a very large domain of possible solutions
- We need to learn the values of many variables which would yield the best possible solution
- When there are too many possible solutions to enumerate them all - we can apply a new type of algorithms: ***stochastic optimizations***

There are many interesting algorithms in this group:
https://en.wikipedia.org/wiki/Global_optimization#Heuristics_and_metaheuristics

Most of them are inspired by nature

## Stochastic optimizations

- I. Baseline: random guess
- II. Hill climbing
- III. Simulated annealing
- IV. Genetic algorithm

## I. Random “optimization”

- Simple idea: generate a huge amount of random solutions and select the one with the minimum score
- No matter how many random guesses you try – the fitness of the resulting solution is very low (the distance from the target remains high)
- This “optimization” does not make use of a better solution that has already been found, it just tries another random guess

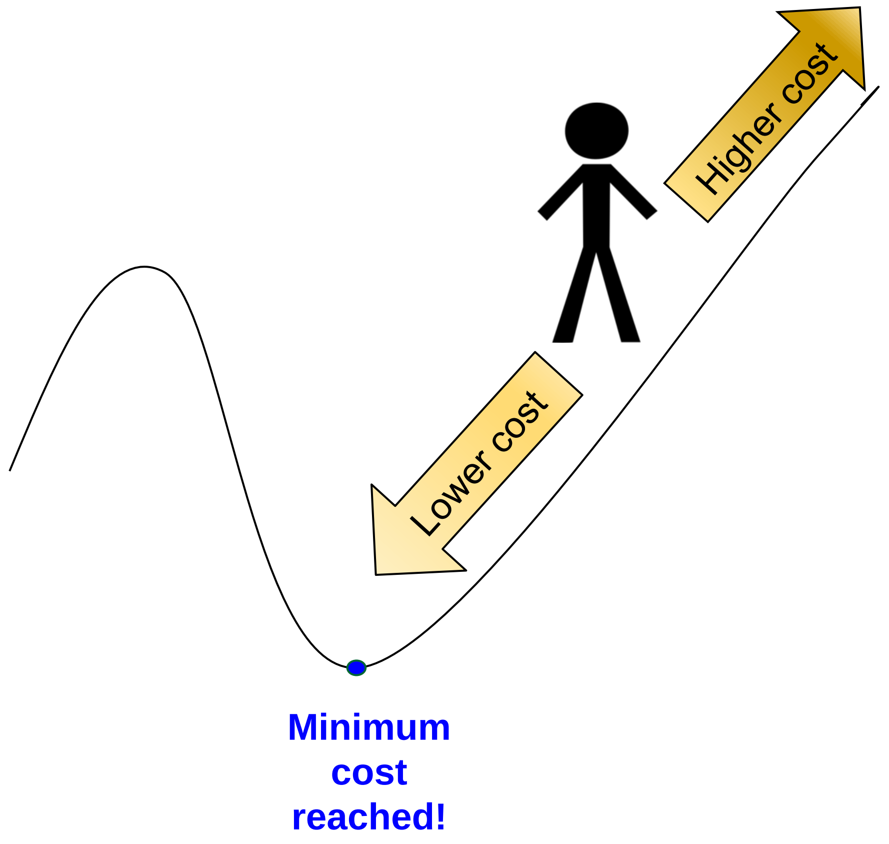
- A better idea: move towards the solution with a better fitness score

## II. Hill climbing

- An alternate method of random searching is called a ***hill climbing***
- Hill climbing starts with a random solution and looks at the set of neighboring solutions
- If one of these neighboring solutions has a better score, the algorithm replaces the original solution with a new one
- You continue moving to a better solution until you reached the minimum

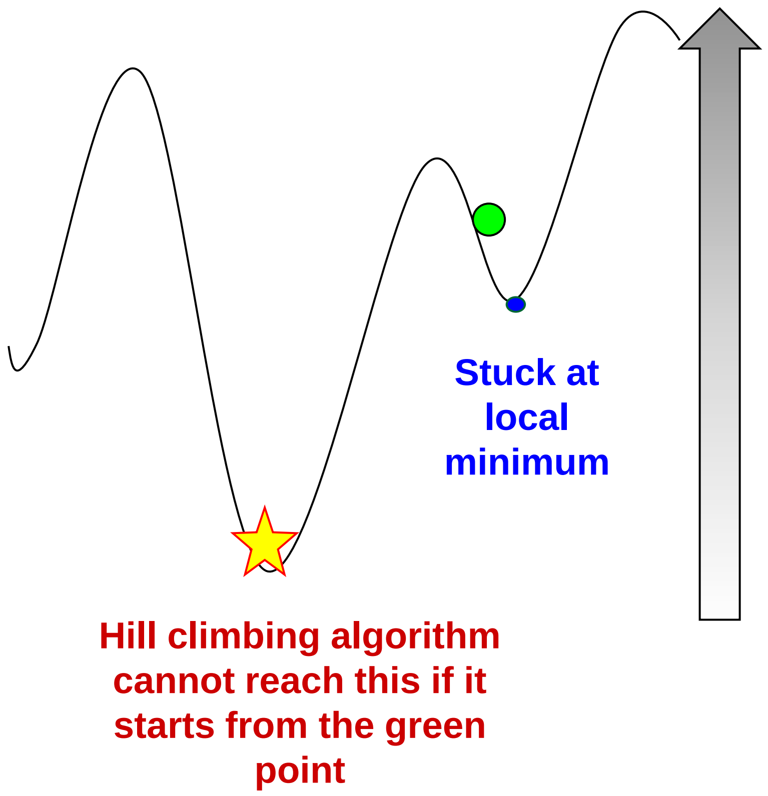
# Finding minimum down the hill

- You have been randomly dropped into this landscape
- You want to reach the lowest point to find water
- You look in each direction and walk in the direction of the downward slope
- You continue to walk in the most steeply sloping direction until you reach a point where the terrain is flat or begins sloping uphill



# Hill climbing: good and bad

- Hill climbing runs quickly
- Usually finds a better solution than random searching
- Drawback: the final solution is a *local minimum*, a solution better than those around it but not the best overall
- The best overall is called the *global minimum*, which is our ultimate goal


Stuck at local minimum
Hill climbing algorithm cannot reach this if it starts from the green point

## III. Simulated annealing

- *Annealing* is the process of heating up an alloy to extremely high temperature and then cooling it down slowly
- The atoms first jump around a lot and then gradually settle into a low-energy state, finding a configuration with the lowest energy overall

- This physical process is simulated by an algorithm to find the global minimum

https://mathworld.wolfram.com/SimulatedAnnealing.html

# Simulated annealing algorithm

- Pick a random solution
- Set up the temperature, which starts very high and gradually gets lower
- In each iteration, one of the numbers in the solution is randomly chosen and changed in a certain direction (as in Hill climbing)

- Main difference:
    - If the new solution cost is *lower*, the new solution becomes the current solution (as before)
    - If the cost is *higher*, the new solution may still become the current solution with a certain probability - this may help to get out of the local minimum

## Probability of moving in the wrong direction

- In some cases, it’s necessary to move to a worse solution before you can get to a better one
- Simulated annealing improves the final result because it will always accept a move for the better, and because it is also willing to accept a worse solution near the beginning of the process

- As the process goes on, the algorithm becomes less and less likely to accept a worse solution, until at the end it will only accept a better solution

# Probability of accepting worst solution decreases as the system "cools down"

- The probability of a higher-cost solution being accepted:

$$p = \frac{1}{e^{\Delta S/T}}$$

T – temperature

ΔS = new_score – old_score (always positive, because we assume that the new score is greater than the old score)

- Now, if the temperature is low (T→1), then the probability of accepting worst solution becomes extremely low
- Since the temperature (the willingness to accept a worse solution) starts very high, the exponent will at first be close to 0, so the probability will almost be 1
- As the temperature decreases, the difference between the high cost and the low cost becomes more important – a bigger difference leads to a lower probability, so the algorithm will favor only slightly worse solutions over much worse ones

## IV. Genetic Algorithms (GA)

- "Evolutionary Computing" was introduced in the 1960s by **I. Rechenberg**.
- **John Holland** wrote the first book on Genetic Algorithms 'Adaptation in Natural and Artificial Systems' in 1975.
- In 1992 **John Koza** used genetic algorithm to **evolve programs** to perform certain tasks. He called his method "**Genetic Programming**".

## What Are Genetic Algorithms (GAs)?

- Genetic Algorithms are *search* and *optimization* techniques based on Darwin's Principle of *Natural Selection*
- Evolution is known to be a successful, robust method to produce adaptations (solutions) to different environments (problems)
- GAs can search a very large space of hypotheses containing complex interacting parts

## Darwin’s Principle Of Natural Selection

IF there are organisms that reproduce, and

IF offspring inherit traits from their progenitors, and

IF there is variability of traits, and

IF the environment cannot support all members of a growing population,

THEN those members of the population with less-adaptive traits (determined by the environment) will die out, and

THEN those members with more-adaptive traits (determined by the environment) will thrive and continue to the next generation

The result is the evolution of species.

## Basic Idea of Natural Selection

***"Select The Best, Discard The Rest"***

# Evolution Through Natural Selection

Initial population

Struggle for existence-survival of the fittest

Surviving Individuals Reproduce,
Propagate Favorable Characteristics

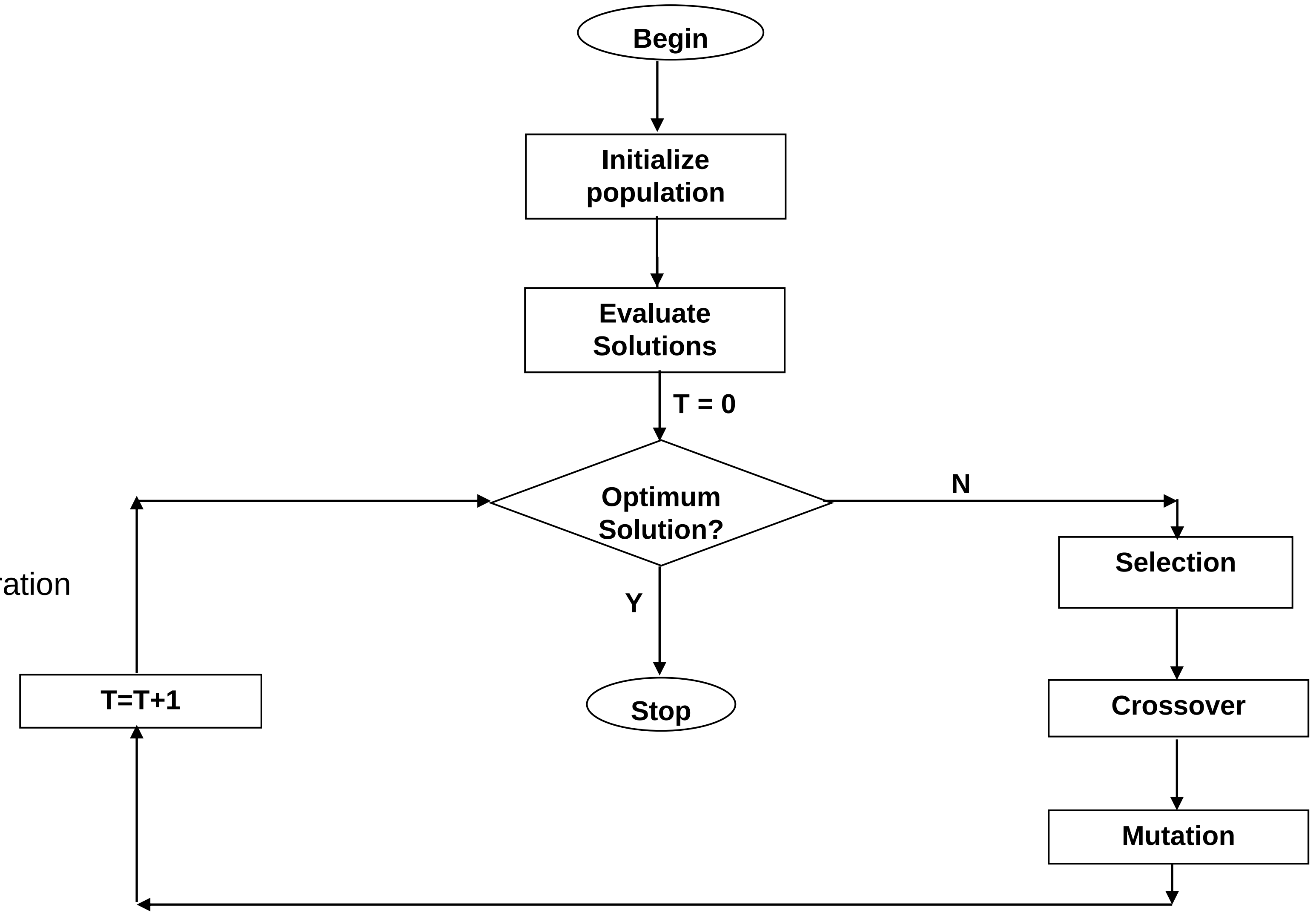
**Loop: Millions Of Years**

Evolved Species

**(Favorable characteristic now a trait of Species)**

Genetic Algorithms implement optimization strategies by simulating evolution of species through natural selection

# Flowchart of GA



T - generation or epoch

## Mapping Nature to an Algorithm

| Nature | Computer |
|---|---|
| Population | Set of initial hypotheses (possible solutions) |
| Individual | Solution to a problem |
| Fitness | Quality of a solution |
| Chromosome | Encoding for a Solution |
| Gene | Part of the encoding of a solution |
| Mating (crossover) | Crossover (exchange parts of solutions) |
| Mutation | Mutation (change part of a solution) |

## GA design

1. *Solution encoding*
2. *Fitness function*
3. *Selection*
4. *Mating (crossover)*
5. *Mutation*

## 1. Solution encoding

*The process of representing the solution in the form of a **string** that conveys the necessary information.*

- Just as in a chromosome, each gene controls a particular characteristic of the individual, similarly, each bit in the string represents a characteristic of the solution

# Solution encoding 1/2

- **Binary Encoding –** Most common method of encoding. Chromosomes are strings of 1s and 0s and each position in the chromosome represents a particular characteristic of the problem.

| | |
|---|---|
| Chromosome A | 10110010110011100101 |
| Chromosome B | 11111110000000011111 |

## Sample Problem

The **Traveling Salesman Problem** is defined as:

*We are given a set of cities and a symmetric distance matrix that indicates the cost of travel from each city to every other city.*

*The goal is to find **the shortest circular tour,** visiting every city exactly once, so as to **minimize the total travel cost**, which includes the cost of traveling from the last city back to the first city.*

## Solution encoding 2/2

- **Permutation Encoding –** Useful in scheduling problems such as the Traveling Salesman Problem (TSP).

- Example: every chromosome is a string of numbers, each of which represents a city to be visited in this order.

| | |
|---|---|
| Chromosome A | 1 5 3 2 6 4 7 9 8 |
| Chromosome B | 8 5 6 7 2 3 1 4 9 |

## 2. Fitness Function

*A fitness function quantifies the optimality of a solution (chromosome) so that that particular solution may be ranked against all the other solutions.*

- A fitness value is assigned to each solution depending on how close it is to the perfect solution.

- Ideal fitness function correlates closely to goal + is quickly computable.

- Example. In TSP, *f(x)* is sum of distances between the cities in solution. The lesser the value, the fitter the solution is.

## 3. Selection

*The process that determines which solutions are to be preserved and allowed to reproduce and which ones deserve to die out.*

- The primary objective of the selection operator is to emphasize the good solutions and eliminate the bad solutions in a population, while keeping the population size constant.
- “Selects The Best, Discards The Rest”.

## Elite Selection

- Sort solutions by fitness (descending).
- Make multiple copies of the top solutions (parthenogenesis – cloning).
- Eliminate bad solutions from the population so that multiple copies of good solutions can propagate in the population.

## Preserving elites

*Elitism is a method which copies the best chromosome to the new offspring population before crossover and mutation.*

- When creating a new population by crossover or mutation the best chromosome might be lost.
- Forces GAs to retain some number of the best individuals at each generation.
- Has been found that elitism significantly improves performance.

# Roulette Wheel Selection

- Each current string in the population has a slot assigned to it which is in proportion to it’s fitness.
- We spin the weighted *roulette wheel* thus defined *n* times (where *n* is the total population size).
- Each time Roulette Wheel stops, the string corresponding to that slot is selected for the next generation.

Strings that are fitter are assigned a larger slot and hence have a better chance of appearing in the new population.

https://en.wikipedia.org/wiki/Fitness_proportionate_selection

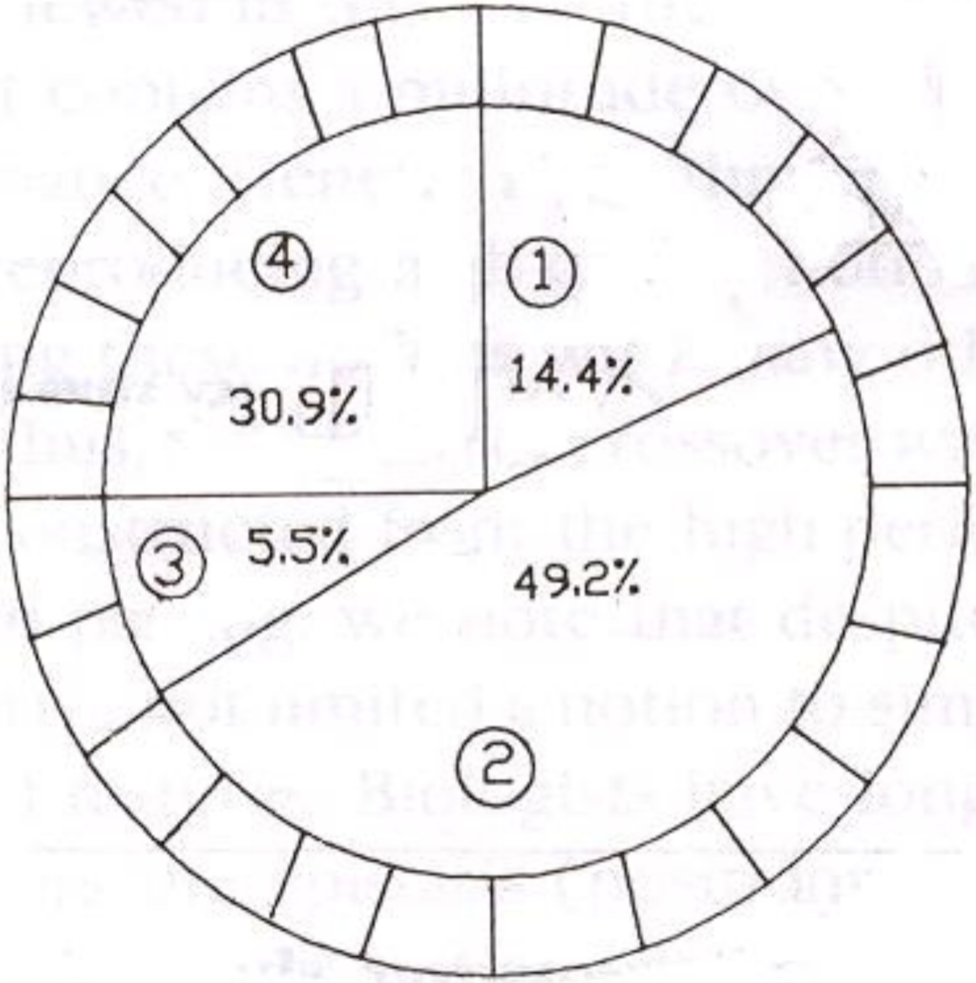
## Example Of Roulette Wheel Selection

Note that in this example the program tries to maximize the fitness score of a solution, unlike in our examples where we try to minimize the cost

| No. | String | Fitness | % Of Total |
|---|---|---|---|
| 1 | 01101 | 169 | 14.4 |
| 2 | 11000 | 576 | 49.2 |
| 3 | 01000 | 64 | 5.5 |
| 4 | 10011 | 361 | 30.9 |
| **Total** | | 1170 | 100.0 |

The solution with a greater fitness has greater proportion on a wheel and a better chance to be selected

# Roulette Wheel

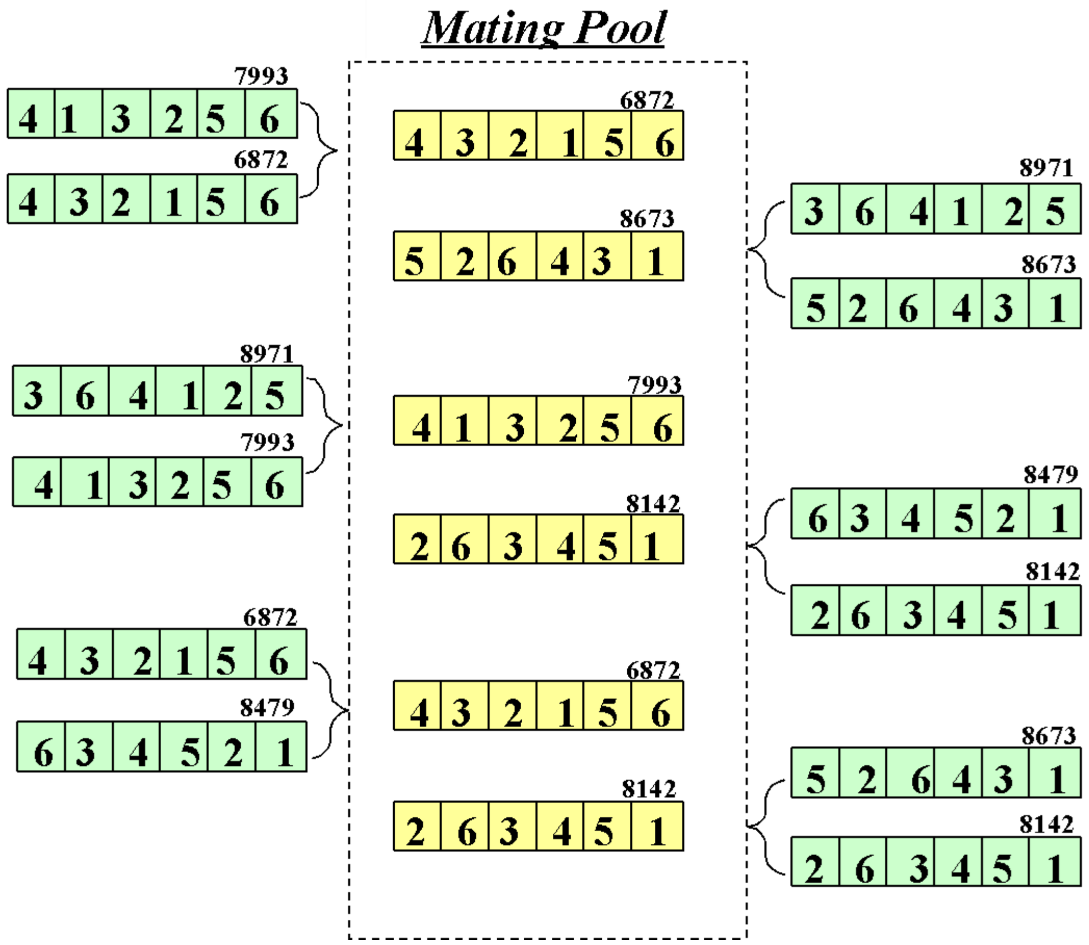


## Tournament selection

- **Tournament Selection**: Two members are chosen at random from a population.
  - With some predefined probability *p* the more fit of these two is then selected, and with probability 1-*p* the less fit solution is selected.

  *Sometimes TS yields a more diverse population that RS.*

# Tournament selection example



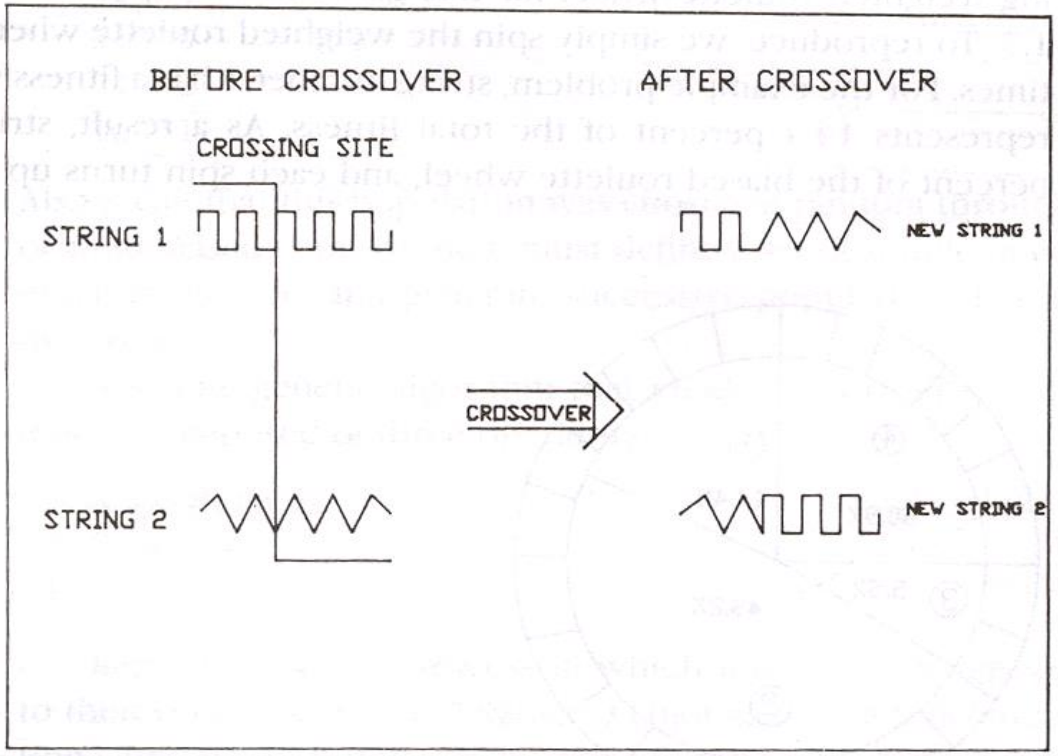
## 4. Mating (Crossover)

*It is the process in which two chromosomes (encodings) combine their genetic material (bits) to produce a new offspring which possesses a mix of their characteristics.*

- *Two strings are picked from the mating pool (or from elite) at random to cross over.*
- *The method chosen depends on the Encoding.*

# Crossover methods: 1/3

- **Single-point Crossover-** A random point is chosen on the individual chromosomes (strings) and the genetic material is exchanged at this point.



## Single-point crossover example

| | |
|---|---|
| Parent Chromosome 1 | 11011 \| 00100110110 |
| Parent Chromosome 2 | 11011 \| 11000011110 |
| Offspring 1 | 11011 \| 11000011110 |
| Offspring 2 | 11011 \| 00100110110 |

## Crossover methods: 2/3

- **Two-point Crossover-** Two random points are chosen on the individual chromosomes (strings) and the genetic material is exchanged at these points.

| | |
|---|---|
| **Parent Chromosome 1** | **11011 \| 00100 \| 110110** |
| **Parent Chromosome 2** | **10101 \| 11000 \| 011110** |
| **Offspring 1** | **10101 \| 00100 \| 011110** |
| **Offspring 2** | **11011 \| 11000 \| 110110** |

**NOTE: These chromosomes are different from the last example.**

## Crossover methods: 3/3

- **Uniform Crossover-** Each gene (bit) is selected randomly from one of the corresponding genes of the parent chromosomes.

| | |
|---|---|
| **Parent Chromosome 1** | **11011 \| 00100 \| 110110** |
| **Parent Chromosome 2** | **10101 \| 11000 \| 011110** |
| **Offspring** | **10111 \| 00000 \| 110110** |

**NOTE: Uniform Crossover yields ONLY 1 offspring.**

## Crossover summary

- Crossover between 2 good solutions **MAY NOT ALWAYS** yield a better or as good a solution.
- Since parents are good, probability of the child being good is high.
- If offspring is not good (poor solution), it will be removed in the next generation during “Selection”.

## 5. Mutation

*It is the process by which a string is deliberately changed at random to maintain diversity in the population.*

We saw in the giraffes' example, that mutations could be beneficial.

***Mutation Probability-*** determines how often the parts of a chromosome will be mutated.

# Example of mutation

- For chromosomes using Binary Encoding, randomly selected bits are inverted.

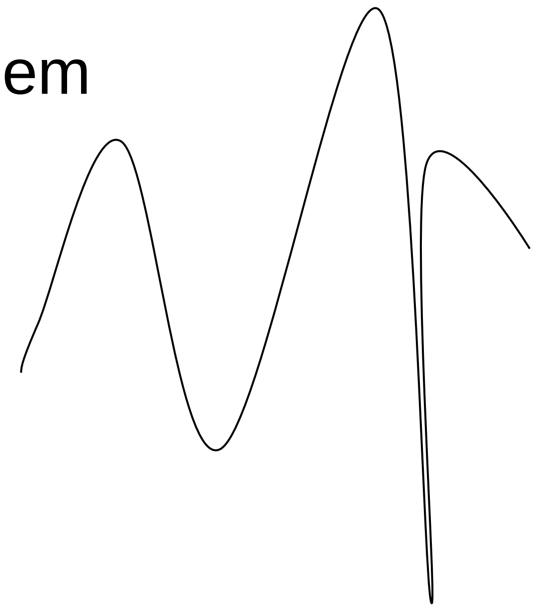
| | |
|---|---|
| **Offspring** | **11011 00100 110110** |
| **Mutated Offspring** | **11010 00100 100110** |

**NOTE: The number of bits to be inverted depends on the Mutation Probability.**

## GA Summary

# When to use Genetic Algorithms

- If you are trying to minimize (maximize) some function *f(x)* over all values of variables *x* in **X**
- When examining every possible combination of *x* in **X** is infeasible
- When you are concerned with the problem of local minimum (maximum) – random mutations can get you out of the trap



## Advantages Of GAs over other optimization methods

- Search for the function optimum starts from a *population of points* in the function domain, not a single point.
- This makes GAs global search methods: they can climb many peaks simultaneously, reducing the probability of being trapped in local minima, which is one of the drawbacks of traditional optimization methods.
- GAs can be easily used on *parallel machines:* since most computational time is spent in evaluating a solution, with multiple processors all solutions in a population can be evaluated in a distributed manner.

# GA applications

GAs are highly effective in searching a large, poorly defined search space even in the presence of high-dimensionality, multi-modality, discontinuity and noise.

Success stories:

- ❑ Finding which concert hall shape gives the best acoustics
- ❑ Designing an optimal wing for a supersonic aircraft
- ❑ Suggesting the best library of chemicals to research as potential drugs
- ❑ Automatically designing a chip for voice recognition

Automatic antenna design with evolutionary algorithms

# Evolving intelligence: genetic programming

## Genetic programming

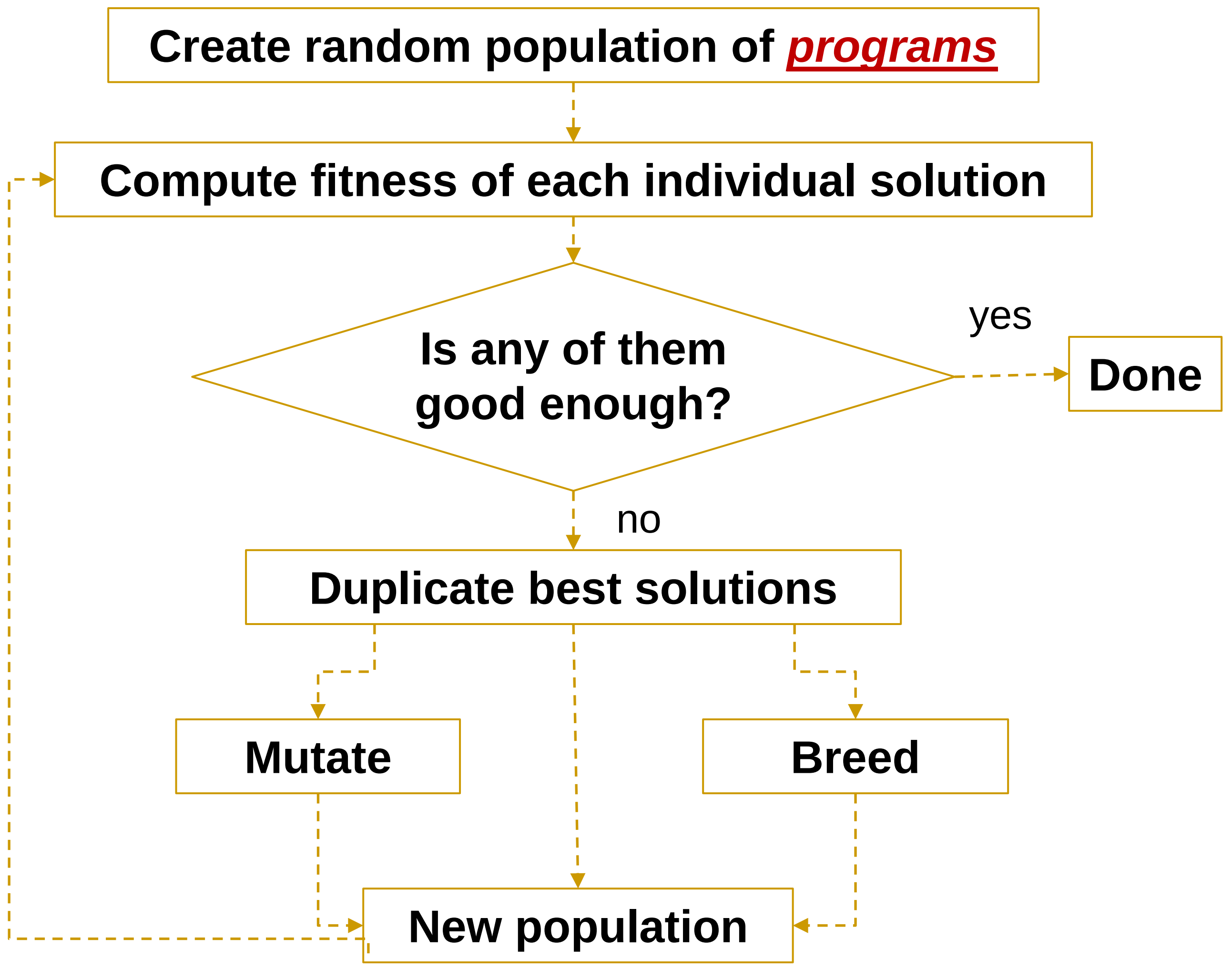
- Application of Genetic Algorithm to the case where solution space consists of computer programs
- The goal is to find a program which performs well at a predefined task
  - Instead of choosing an algorithm that is the best for a predefined task, we make a program that will create such algorithm: we design an algorithm which creates algorithms
- In some cases the algorithm finds programs that are human-competitive

"Human-competitive results produced by genetic programming"

## How does it work

- We start with a large set of programs (*population*), which are either [randomly generated] or [hand-designed to be somewhat good solutions]
- The programs then compete in performing some user-defined task:
  - A game in which the programs compete against each other and the performance is measured by the number of wins
  - A known set of inputs and outputs and the best program (function) that perfectly maps inputs to outputs

# Genetic programming flowchart


Create random population of *programs*
Compute fitness of each individual solution
Is any of them good enough?
yes
Done
no
Duplicate best solutions
Mutate
Breed
New population

# Same steps as in GA

- After evaluating each program using the *fitness test*, we produce *a ranked list of programs*
- The best programs are **replicated** and **modified** in two different ways
  - *Mutation*: certain parts of a program are altered slightly in a random manner in hope that this will make a good solution even better
  - *Crossover* (*breeding*): exchange the portions of best programs
- This replication and modification procedure creates many new programs which are evaluated until the best solution is found

## Programs get better with each new generation

- Since the size of the population is kept constant, many of the *worst* programs are eliminated from the population to make room for new programs
- Because the best programs are being kept and only slightly modified, it is expected that with each generation they will get better and better

## Genetic Programming (GP) vs. Genetic Algorithm (GA)

- GA is an optimization technique
  - As with any optimization, you have already selected an algorithm or metric and you're trying to find the best parameters for it
- In GP the solutions are not just a best set of parameters applied to a given algorithm:
  - The algorithm itself and all its parameters are designed automatically by means of evolutionary pressure

# GP: sample applications

- Developing programs for playing games, such as chess and backgammon
- Used in photonic crystals, optics, quantum computing systems, and other scientific inventions
- In 1998 a robot team that was programmed entirely using genetic programming which placed well in the Robo-Cup soccer contest

  https://cs.gmu.edu/~sean/papers/robocupgp98.pdf

Sample application 1

## GROUP TRAVEL OPTIMIZATION

## Use case: travel optimization

Setup:

- Six family members meet up in New York.
- They arrive on the same day and leave on the same day, and they want to share transportation to and from the airport.
- There are many possible flights per day for each family member: 8 inbound flights and 8 outbound flights
- Flights are arriving-leaving at different times. They also vary in price and in duration.

```
people = [('John','BOS'),
          ('Mary','DAL'),
          ('Laura','CAK'),
          ('Abe','MIA'),
          ('Greg','ORD'),
          ('Lee','OMA')]

# LaGuardia airport in New York
destination ='LGA'
```

```
LGA,MIA,20:27,23:42,169
MIA,LGA,19:53,22:21,173
LGA,BOS,6:39,8:09,86
BOS,LGA,6:17,8:26,89
LGA,BOS,8:23,10:28,149
…
```

Flights in file schedule.txt

## Solution encoding

- We represent each possible solution as a list of numbers.
- Each number represents the position of a flight in a list of flights sorted by time.
- Since each person needs an outbound flight and a return flight, the length of this list is twice the number of people.

# Sample solution

$$[\underbrace{1,4}_{\text{John}},\underbrace{3,2}_{\text{Mary}},7,3,6,3,2,4,5,3]$$

- John takes the second flight of the day from Boston to New York, and the fifth flight back to Boston on the day he returns.
- Mary takes the fourth flight from Dallas to New York, and the third flight back.

```
people = [('John','BOS'),
          ('Mary','DAL'),
          ('Laura','CAK'),
          ('Abe','MIA'),
          ('Greg','ORD'),
          ('Lee','OMA')]
```

## Fitness function design: most challenging and non-trivial part

- The goal is to find a set of flights that minimizes the cost function.
- The cost function has to return a value that represents how bad a solution is.
- There is no particular scale for badness: the only requirement is that the function returns larger values for worse solutions.

## What to include into the fitness function

- Price
  - The total price of all the plane tickets [can be a weighted average that takes financial situations into account].
- Travel time
  - The total time that everyone must spend on a plane.
- Waiting time
  - Time spent at the airport waiting for the other members of the party to arrive.
- Departure time
  - Flights that leave too early in the morning may impose an additional cost by requiring travelers to miss out on sleep.
- Car rental period
  - If the party rents a car, they must return it earlier in the day, or be forced to pay for a whole extra day.

## Run schedule optimization

- Execute cells in *Parts 1 and 2* in https://github.com/mgbarsky/labs_ml_optimizations
- Don’t forget to FORK the repository before cloning it: it contains your lab assignment

How much better is the GA solution comparing to the random pick?

Sample application 2

## BEST STUDENT TO DORM ASSIGNMENT

## Student → Dorm optimization

- The goal is to assign students to dorms depending on their first and second choices.
- Although this is a very specific example, it’s easy to generalize it to other problems—the exact same code can be used to:
  - assign tables to players in an online card game
  - assign bugs to developers in a large coding project
  - even to assign housework to household members
- In all these problems the purpose is to take preferences from individuals and produce the overall optimal result

## Representing solutions

```
# The dorms, each of which has two available spaces
dorms=['Williams','Sage','Lehman','Armstrong','Mills']
```

- In theory we could create a list of numbers, one for each student, where each number represents the dorm in which you put the student.
- The problem is that this representation doesn’t constrain the solution to only two students in each dorm.
- A list of all zeros would indicate that everyone had been placed in Williams, which isn’t a real solution at all.
- We could potentially ….

## Considering only valid solutions

- In general, it’s better not to waste time  searching among invalid solutions
- We need to find a way to represent solutions so that every solution is valid
- A valid solution is not necessarily a good solution; it just means that there are exactly two students assigned to each dorm

Idea (you do not have to use it - come up with your own)

- We think of every dorm as having two slots, so that there are 10 available slots in total
- Each student, in order, is assigned to one of the open slots—the first person can be placed in any one of the ten, the second person can be placed in any of the nine remaining slots, and so on.

## Solution is represented as a list of assignments `[0,0,0,0,0,0,0,0,0,0]`

- The *print_solution* in *Part 3 of your lab* illustrates how this solution representation works
- This function first creates a list of slots, two for each dorm. It then loops over every number in the solution and finds the dorm number at that location in the slots list, which is the dorm that a student is assigned to
- It prints the student and the dorm, and then it ***removes*** that slot from the list so no other student will be given that slot
- After the final iteration, the slots list is empty and every student and dorm assignment has been printed

## Task 1. Each slot has its own domain

```
[0,0,0,0,0,0,0,0,0,0]
```

- If you change the numbers to view different solutions, remember that each number must stay in the appropriate range.
- Your first task is to figure out what is this range (domain) for each student in the solution list

## Task 2. Fitness function

- The *cost* function works similarly to the *print* function.
- A list of all slots is constructed and slots are removed as they are used up
- The cost is calculated by comparing a student’s current dorm assignment to his top two choices.
  - Add 0 if the student is currently assigned to the room of their top choice
  - Add 1 if the student is assigned to their second choice
  - Add 3 if the student is not assigned to either of their choices

## Dorm optimization

- After you defined a valid domain for each position in the solution and created a fitness function, you can reuse the optimization code in Part 2 of the lab:  the algorithms stay exactly the same
- You may need to adjust some parameters for better results

- There is also a ‘challenging’ bonus coding

# Designing optimization algorithms: Summary

- In order for this to work:
  - The problem needs to have a defined cost function
  - Similar solutions should yield similar cost

- To restate any problem as an optimization problem:
  - Represent each solution as a sequence (string!)
  - Define a way to evaluate the quality of each solution: the cost function (solution fitness)

- We start with a random solution (stochastic!) and try to improve its fitness with slight changes