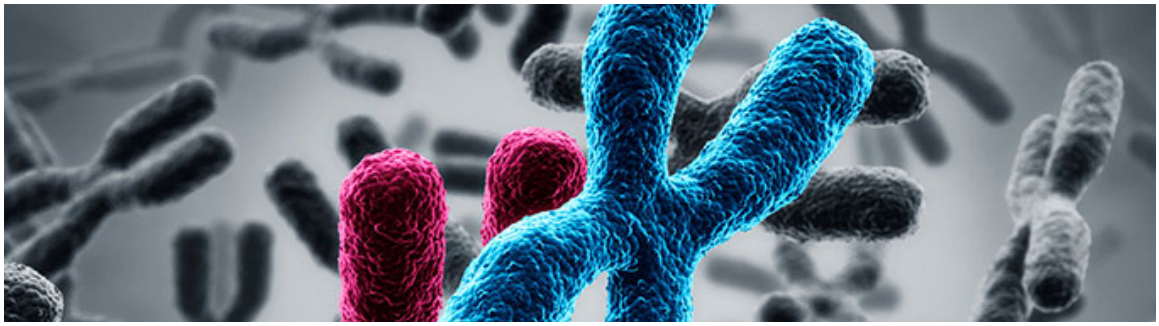
Genetic Algorithms

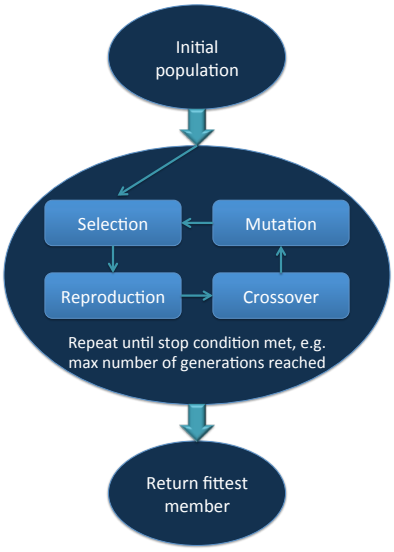
created by Lisa Wang



Chromosomes. Source: http://engineering.stanford.edu/sites/default/files/Haploid_image_680x320.jpg

Genetic algorithms (GA) are inspired by evolutionary biology and mimic the process of natural selection. They are heuristics used in search and optimization problems, e.g. to design aerodynamic race car designs, to generate puns, to enhance encryption or to find solutions to traffic routing problems, e.g. the Traveling Salesman Problem (TSP): Given a list of cities and their pairwise distances, what is the shortest route that visits each city exactly once, ending at the first city?

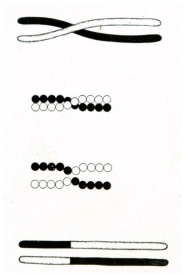
How Genetic Algorithms Work:



1. The solutions to the problem are encoded as idealized chromosomes[1]. An idealized chromosome is a linear structure that consists of smaller blocks. E.g. in TSP, every solution is a route. The route can be encoded as "ACBA" (Start at city A, go to C, then B, then back to A).
2. Generate a set of n random chromosomes which represent possible solutions. The number of chromosomes should be significantly smaller than the number of all possible solutions.
3. For each chromosome, evaluate its fitness by passing it into a fitness function. In the genetic algorithm for TSP, the fitness function calculates the length of the route. The shorter it is, the higher the fitness rating. Notice that the fitness function is different for each problem.
4. To create the next generation of chromosomes, select two parent chromosomes to create two child chromosomes. Based on the principle of natural selection, a chromosome is more likely to be selected to "reproduce", if it has a higher fitness rating. Each selection is independent from previous selections. Hence, a specific

1 In biology, a chromosome is a complex structure of DNA, RNA or protein found in cells. For genetic algorithms, the chromosome has been idealized and simplified to its essential properties. An idealized chromosome is a simple linear structure consisting of building blocks, the so-called genes. We can also perform mutations and crossovers on idealized chromosomes, inspired by their biological counterparts.

parent chromosome can be selected multiple times and another one might never be selected.
5. By crossing over the parent chromosomes, two child chromosomes are born. The crossover point is randomly chosen.
6. After the child chromosomes have been created, apply random mutations to some of their genes. Usually, the mutation rate is very low.
7. Repeat steps 3-7 until one or more of the following conditions are met:
   A. Reached maximum number of generations
   B. Achieved required fitness rating
   C. Passed the runtime limit
   D. or any other reasonable stop condition.
8. Return the solution with the highest fitness rating.



Crossover step

Today's Challenge:
We will implement the crossover and mutation methods.
For this exercise, we will model the chromosomes as linked lists. Each node of the linked list represents a gene, and the list represents the entire chromosome.
For simplicity, each gene is a character. However, depending on the purpose of the genetic algorithm, a gene could also be a string or any other data type.

```
/* struct Gene
 * -----------
 * Stores information about a gene which
 * can be seen as a node in the chromosome
 * linked list.
 */
struct Gene {
    char letter;
    Gene * next;
};
```

```
/* struct Chromosome
 * -----------------
 * A wrapper struct for the linked list
 * that stores a pointer to the head of
 * the list as well as the size.
 */
struct Chromosome {
    int size;
    Gene * head;
};
```

a) Crossover

```
/* Function: crossover
 * Usage: int result = crossover(parentA, parentB, childA, childB);
 * ----------------------------------------------------------------
 * Given two parent chromosomes of the same size, this function creates two new child
 * chromosomes by crossing over the parents and deletes the parent wrapper structs.
 * A crossoverPoint of value x specifies that the first x genes from parentA should go
 * to childA and the rest should go to childB.
 * E.g. if parentA = ABC, parentB = DEF and crossoverPoint = 2, then childA = ABF and
 * childB = DEC.This also implies that if crossoverPoint is 0, the children are swapped
 * copies of the parents and if crossoverPoint is the size of the chromosome, the children
 * are exact copies of their parents.
 * Returns -1 if the parent chromosomes do not have the same size and the crossoverPoint
 * when the crossover succeeded.
 * Note that we have to pass in the child chromosome pointers by reference, since this
 * function modifies them.
 */
int crossover(Chromosome * parentA, Chromosome * parentB, Chromosome * & childA,
                            Chromosome * & childB)
```

b) Mutation

```
/* Function: mutate
 * Usage: mutate(child, geneAlphabet);
 * ----------------------------------
 * Mutates the chromosome by deleting a gene randomly and inserting a gene randomly. The
 * deletion and the insertion do not have to be at the same position in the chromosome.
 * For the insertion, choose a random letter from the geneAlphabet.
 */
void mutateChromosome(Chromosome * chrom, Vector<char> & geneAlphabet)
```

c) Arrays vs. Linked Lists

Instead of using linked lists, we could have also used arrays to implement the chromosomes. However, using arrays instead of linked lists comes with important tradeoffs. Discuss with your partner on a high-level how you would implement the two methods with arrays and what the tradeoffs would be.
You can use the table below for reference:

	Array	Linked List
Size Flexibility	No. Once an array is created, its size cannot be changed.	Yes. You can shrink the list by removing nodes or grow the list by adding nodes.
Random Access (e.g. to retrieve or update one element)	Yes. This is the biggest advantage of arrays. Any element in the array can be accessed in constant time. O(1)	No. Given a pointer to the start of the linked list, you have to traverse the list until you have found the desired node. Hence, it generally takes linear time to access a linked list element. O(n)
Insertion or Deletion at the Front	Requires all elements to be shifted to the back for insertion, given that the array is not full yet, or shifted to the front for deletion. O(n)	Create new node and link of the front of the list for insertion, remove first node and update head pointer for deletion. O(1)
Generalized Insertion or Deletion	Both insertion and deletion require shifting, unless it is done at the end of the array. More difficult to implement. O(n)	Need to traverse the list until the position where the insertion or deletion should happen, which takes O(n). The insertion / deletion itself only requires a few pointer updates, takes constant time and is easy to implement.

Craving for more?

- Take CS274 if you want to learn more about biocomputation!
- Take CS103 if you want to understand how hard NP-hard problems actually are!
- brainz.org article: "15 Real-World Uses of Genetic Algorithms" http://brainz.org/15-real-world-applications-genetic-algorithms/
- Science Daily article: "Genetic approach helps design broadband metamaterial" http://www.sciencedaily.com/releases/2014/05/140505112538.htm