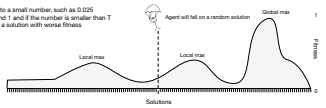
Simulated Annealing

1. Agent evaluates fitness of landing solution, and of those solutions that are around
2. If a neighbouring solution has better fitness, agent moves to that solution
3. Agent keeps doing this until moving onto a high fitness solution, aiming for Global max

Agent can implement a mechanism to get out of local maxima usually when stuck in solutions with Fitness smaller than 0.5.

1. set a Temperature parameter T to a small number, such as 0.025
2. Get a random real between 0 and 1 and if the number is smaller than T
   2.1 Allow the agent to move to a solution with worse fitness
   2.2 Let it run for some time



Simple Genetic Algorithms (Elite based)

1. Generate a seed population **S** containing **P** random solutions
2. Evaluate the fitness of the solutions in **S**, and sort the list from best to worst
3. Copy the top **E** solutions to a set called **NextGeneration**
4. Create **P-E** offspring (see crossover box below) from the **Elite** set
5. Apply **mutation** operator to the offspring (see mutation box below)
6. Add OffSpring to **NextGeneration** and make **S = NextGeneration**
7. Ready to repeat 1-6 for new generation.

GA usually **terminates** either after a number of generations, or when finding solutions with a desired fitness.

PARAMETERS

P: Population size
S: Seed population
E: Elite size
m: mutation rate

Crossover

1. Select two random solutions from the **ELITE** set (in **NextGeneration**)
2. Select a random point **CROSSOVER Point** in the solution string

3. Put the children in a set called **OffSpring** and repeat until you have **P-E** *different* offspring solutions

Mutation

0. Set mutation rate **m** to a very small number such as 0.025
1. For each solution **Si** in the set **OffSpring**
2. And for each part of the solution **Sij**
   2.1 Get a random real between 0 and 1, and if this number is smaller than **m**, then make **Sij** change to a random but valid value.

Cellular Automata

1. Please refer to the PDF chapter on the theory of cellular automata (CA), or otherwise ensure you understand the basics of CAs
2. Remember that we always specify ratio *r*, which corresponds to the number of neighbours every cell considers when updating its state
3. Remember that neighbourhood size **n** is **n = 2r + 1** and that we only consider cells with **k=2** possible states (zero or one)
4. The rule size for rules with **k=2** states is 2 to the nth power. For example for **n=7** our rule size is 2 to the 7th power which is 128
5. We always pass to our code the ratio *r* and the rule **phi** where the rule is expressed as a decimal number. To use the rule you must:
   5.1 with *r* we first compute **n** and the rule size **rsize** = 2 to the nth power
   5.2 convert the rule in decimal to binary, and make each digit be an element of a list.
   5.3 add zeroes to the left of this list until the list contains **rsize** elements, e.g. 128 for **n=7**
   5.4 reverse the list and the rule will be ready to be used.

Now you will typically initialise a lattice **L**, which is a list of a given length initialised with random zeroes and ones. Remember that first and last element are connected. Your initial list will correspond to the network as it is in time **t** = 0. We will use the rule to compute the next state of the lattice, at **t** = 1. To do this:

1. For each element of **L**, take its neighbourhood, *r* elements to the left, and *r* elements to the right keeping the element to be updated in the middle. You will have **n** elements as they are at the current time step (time t = 0 at the start). Initialise a list called **NextL**
2. Convert the binary number of **n** digits to decimal and store it in a variable **i**.
3. Extract the rule **phi** element in position **i**.
4. Append the extracted element to **NextL** in the position **i**.
5. Repeat until the last element of **L**, the resulting **NextL** corresponds to the updated lattice. This can be repeated, and every new **NextL** appended to a spacetime matrix **M**.

NOTE that we are only working with ONE-DIMENSIONAL cellular automata. Check the PDF Chapter if you do not know what that means.

Density Classification Task

We assume you can now compute space time dynamics for cellular automata with r=1,2 and 3. We will focus on r=3 from now on.

The main idea goes like this:

1. We start with a lattice the size of which is odd, for example 151, or 301. We then assign to each lattice element (cell) a random value 0 or 1. Because the lattice size is odd it means that no matter how cell values are assigned there will always be more cells in one state than the other. For example if our lattice is L = 000101010110111 its size is 15, and it has 7 zeroes and 8 ones. The majority in this example is 1.

2. We know that the CA rules work by computing the next state of each cell using neighbour (local) information. This means that no cell in the lattice can know what state is in the majority for the entire lattice. No cell has a global view.

3. We want to find CA rules that when used on such lattices make them, over time, have a state where all cells are in the same state, and we want that state to be the one that was in the majority when it started.

4. We know such rules exist in r=3 (n=7) but we don't know how to find them and this is why we use stochastic search.

Rule Score

1. Create a list with L elements where each is random 0 or 1. Make the list size a parameter. This is your lattice at t = 0.
2. Count the number of 1s and the number of 0s in your lattice. If there are more 1s, make a majority variable have the value 1, else make it 0.
3. Run the automaton for 2L time steps, this should give you a matrix M with dimensions L x 2L.
4. Retrieve the last lattice in M, and compute score = (Count of elements = majority) / L (which must be a number between zero and 1

Rule Fitness

To compute the fitness of a single rule you must compute its score for a large number of different examples, and then compute the average of them all. We recommend using at least 10^4 or 10^5 different scores from different examples.

Searching for rules (The work you will submit)

1. Write function that **initialises a population** of P = 300 unique CA rules in r=7. These rules will appear as 300 random decimal numbers between 0 and 2 to the 128th. Here I provide a rule that has good fitness that you can use just for testing:

339841014795010104073313096675879420072

You can also use the rule 18 in r=1 to test. When you run it over at least 100 time steps on a random lattice with at least 80 cells you will see a pattern of downward pointing triangles (you can google elementary CA rule 18 and see the background PDF chapter).

2. Write another function to **evaluate the fitness of the current generation** using 10000 examples and returns a sorted population from best to worst fitness.

3. Write a third function that **prepares the next generation** by doing the following
   3.1 copy the top 50 CA rules with best fitness into the NextGeneration (Elite)
   3.2 using this elite set, produce the 250 new rules using crossover and mutation

4. Write a last, main function that orchestrates the work, by initialising the population and running the genetic algorithm for a number of generations, try to run at least 500 generations. Remember that step 1 above is done only once to create the first population. The next generations are descendants of this seed original population.

What do you need to submit?

1. your Python code, well documented.
2. A short report in which you show a line plot with the **average fitness of your top ten rules in every generation**. The minimum number of generations you must run your code for is 50. But the more, the better.
3. Include in your report the raw numbers used in your plot.
4. For (2) and (3) Use the EXCEL template provided for this.

Submission notes:

Your two files must be added to a single zip file.
Python code that is not documented properly will be heavily penalised by losing marks.