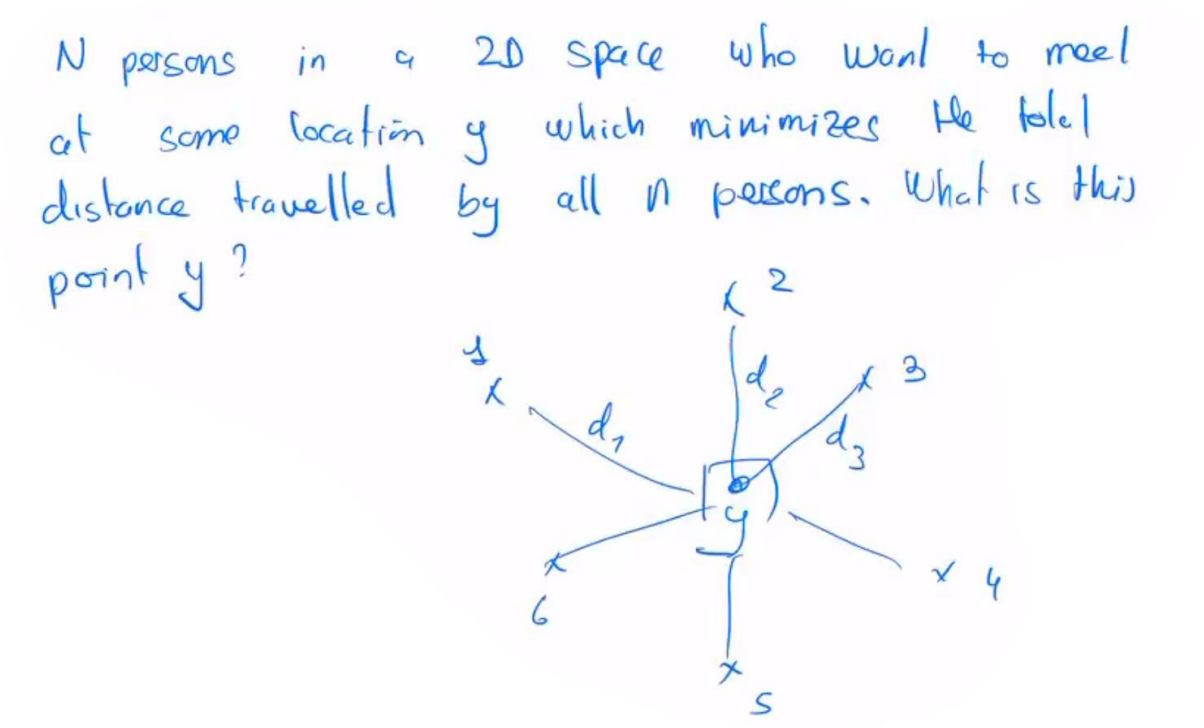
1 - Optimization problems and the search space



- ➔ The search space S, is a set of possible values for y
- ➔ If the problem is in R^2 -> the search Space is R^2, all different possible values for y
- ➔ The search space can be continuous discrete finite or infinite
- ➔ If the search space is too big, we can't do an exhaustive search

- ➔ Lets define a function f: S -> R, this function is called objective function, cost function, energy function, fitness function ..

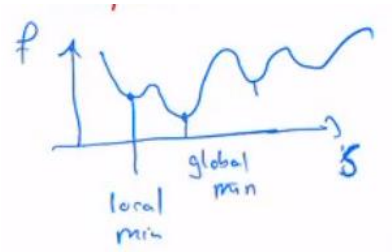
- ➔ Its goal is to quantify a possible value y of the search space S, this way we can compare 2 different possible values for y and choose the best one. The main goal being to find the best value possible for y.

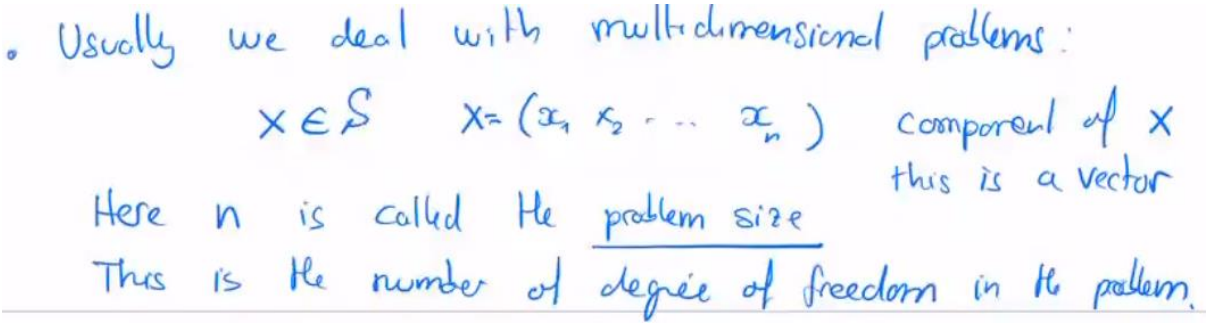
- ➔ This optimal value will either minimize of maximize the function f

$$\begin{cases} x_{opt} = \underset{x \in S}{\arg\min / \arg\max} f(x) \\ f_{opt} = \underset{x \in S}{\min / \max} f(x) = f(x_{opt}) \end{cases}$$

- ➔ This Xopt may not be unique

➔ One difficulty for these problems will be to distinguish global optimums from local optimums, visually it looks like this:





s

metaheuristics is the way to explore S when no polynomial algorithms exist to find the optimal solution

2 - Main principles of metaheuristics, neighbourhood, movements, exploration operator, population metaheuristics

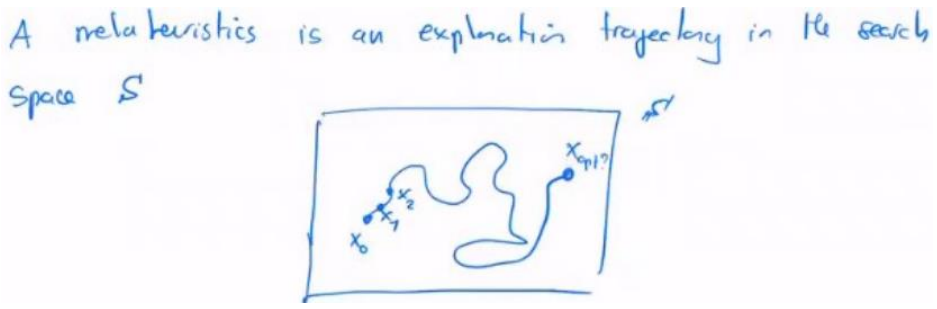
GOAL of metaheuristics: explore a large search space in a clever way. It is needed to solve large problems, for which only exponential algorithms are known

- ➔ It is a compromise between CPU time and quality of solution
- ➔ No guarantee on the quality of the solution
- ➔ As opposed to heuristic where the algorithm is specific to a problem, metaheuristic algorithm can be applied to many different problems

Characterization:

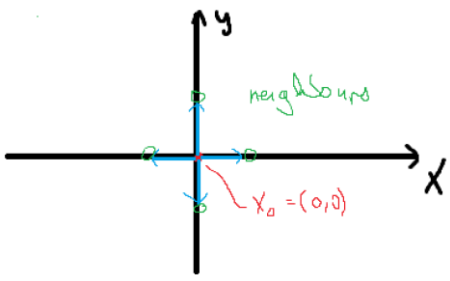
- no hypothesis on the mathematical properties of the fitness function
- Require guiding parameters that defines how the space should be explored. The quality of the solution will depend on how good these parameters are
- need a starting point (usually random)
- need a stopping condition (iteration, no more improvement, time, value …)
- inspired by natural processes (ant system, beehive …)
- can be parallelisable
- most of them are stochastic, use random numbers to guide a search

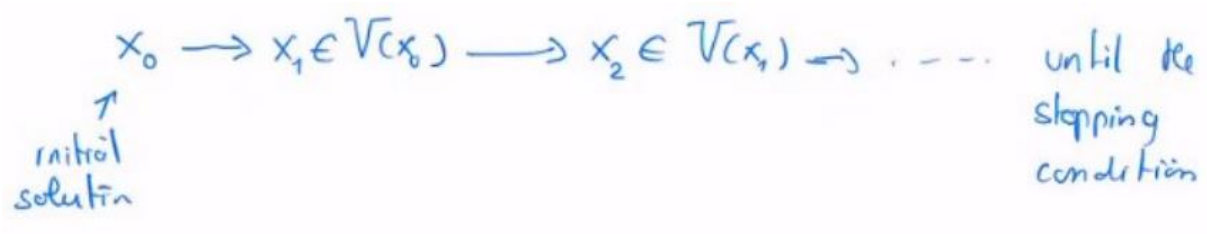
Main principle:



To move across the search space, we use the concept of neighbourhood.

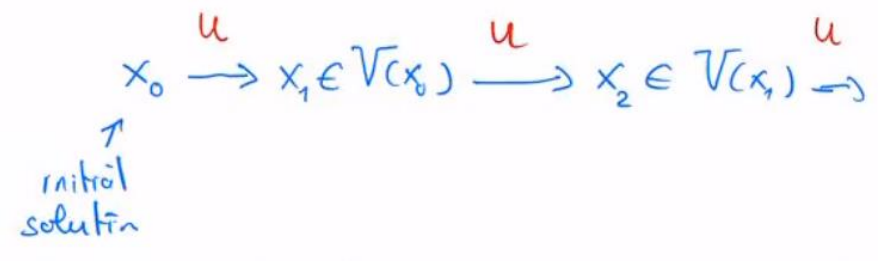
- ➔ If we are currently exploring a point x, the next point we explore is a neighbour of x
- ➔ The neighbourhood is found by doing elementary transformations on the current explored state. neighbourhood of x is called V(x)





Search operator:

- ➔ When we are at a state x, we find its neighbourhood V(x), and now we need to choose which neighbour to explore next.
- ➔ This is done via the search operator -> $u : V(x) \rightarrow y \in S$



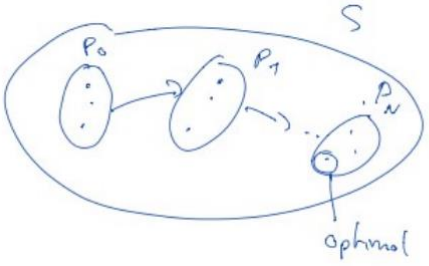
Population metaheuristics

So far we assumed that at each iteration we consider only one possible solution:

$x_0 \rightarrow x_1 \in V(x_0) \rightarrow x_2 \in V(x_1) \dots$

But we could also consider a population of solutions:

$P_0 = \{ x_1^{(0)}, x_2^{(0)}, \dots x_M^{(0)} \}$ M possible solution at iteration 0



3 – The space of permutations

S is the set of permutation of n objects

$n = 3$ $\quad S = \{(1,2,3), (1,3,2), (2,1,3)(2,3,1), (3,1,2)(3,2,1)\}$

$|S| = n! = 6$

What kind of transformation can we define on a permutation

- The simple way is to use transpositio or swap of two entries in the permutation.

$T_i = (i\ j)$ which will swapp entry i with entry j

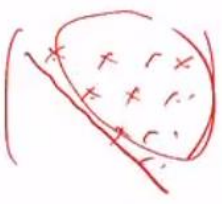
For instance: $T_{(2,3)}((1,2,3)) = (1,3,2)$

How many transpositio do we have ?

$n = 5$

$T_i \in \{(1,2), (1,3), (1,4), (1,5),$
$(2,3)\ (2,4)\ (2,5),$
$(3,4), (3,5), (4,5)\}$

How many are Here ? $O(n^2)$



The neighborhood is of size n^2
whereas the search space is of size $n!$

Can we reach any point in S starting from any initial condition, using a finite number of these transformation ?

Answer : Yes see math :
It means that the entire space of permutation is accessible

4 - Complexity and the need for metaheuristics, exploration versus Exploitation

Complexity classes:

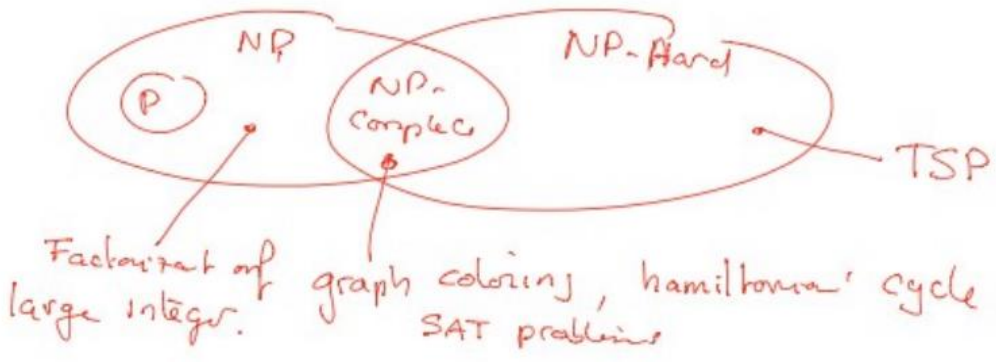
Class P : There is an algorithme that solves the problem in a polynomial time $T(n) = O(n^m)$

Class NP : Problems for which a solution can be checked in a polynomial time.

Thus $P \subseteq NP$

Class NP-hard : Those are problems whose solution can be used to solve any NP problem up to a polynomial additional time.

Class NP-Complete Are problems that are both in NP and in NP-hard.



Typically, NP-hard and NP-complete problems can be solved by exponential algorithms

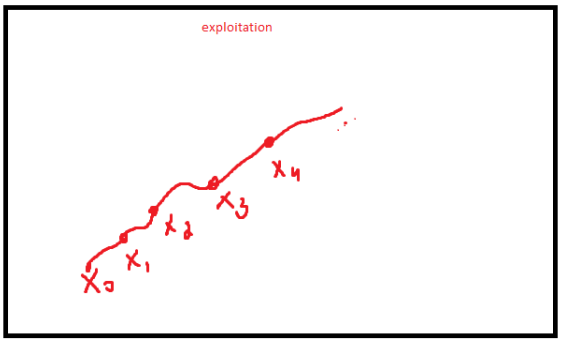
Example:

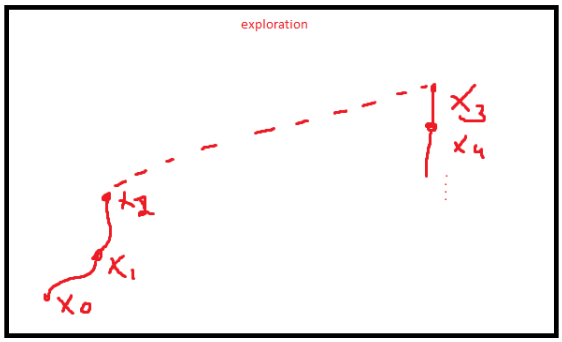
Hamilton cycle: find a graph that goes through all nodes once and only once can use TSP to be solved.

Exploration vs exploitation:

In all cases, a metaheuristic traverses the search space trying to combine two actions: intensification and diversification, also called exploitation and exploration respectively

- ➔ (exploitation) intensification phase the search explores the neighbourhood of an already promising solution in the search space
- ➔ (exploration) diversification a metaheuristic tries to visit regions of the search space not already seen

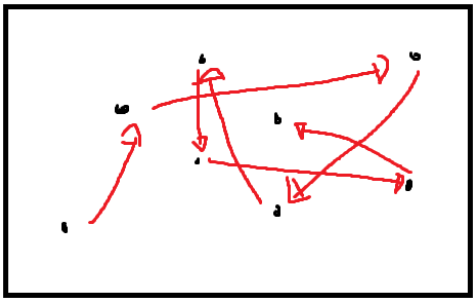




5 - Random search, random walk, hill climbing

Random search:

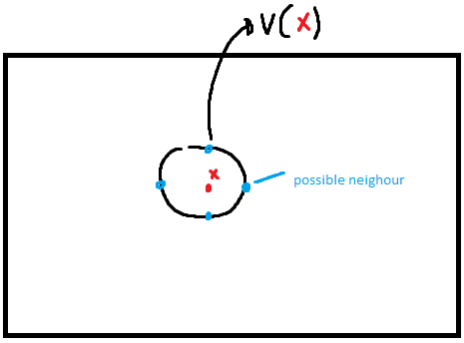
The simplest search method is random search where the next point to test in the search is chosen uniformly at random in the whole search space S. Usually, one keeps the solution having the best fitness after having performed a prescribed number of steps.



Random walk:

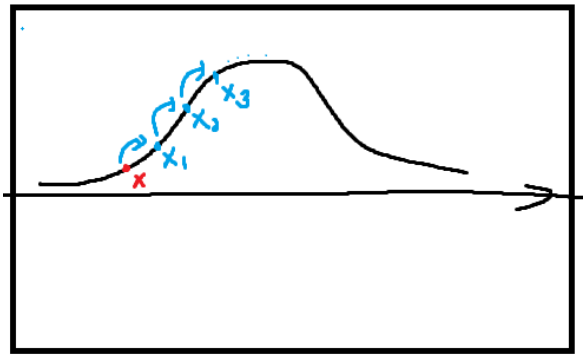
random search to the neighbourhood of the current solution then we have what is called a random walk-in search space.

➔ We don't use the operator function to choose which is the best neighbour to explore



Hill climbing:

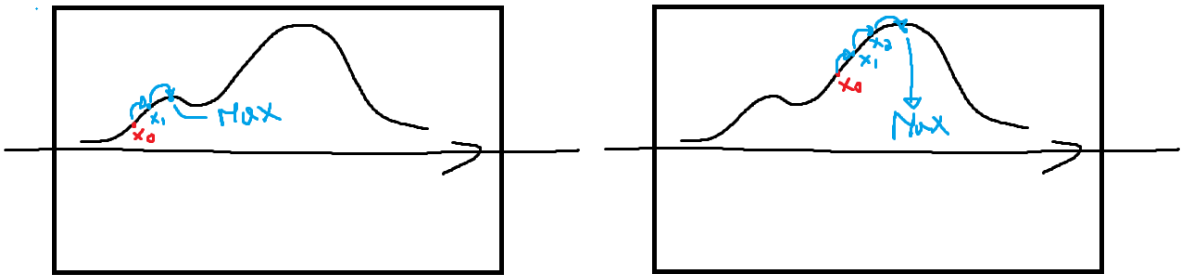
More logical one, we start at a random point x, loot at all our neighbours and choose the one that has the highest (or lowest, depending on the type of problem) fitness to explore next, gives it an effect of climbing (or descending) an hill



We stop when we have climbed the hill -> all neighbours are lower fitness than the current state.

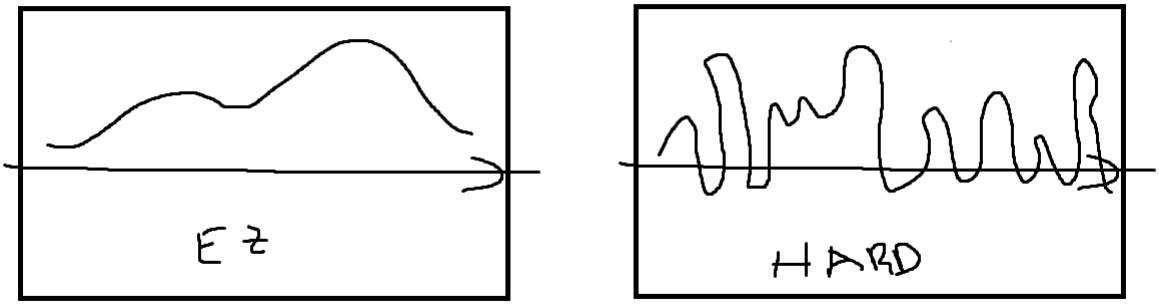
Comment:

Of the three methods the one with the highest chance is the hill climbing. However, it is very easy to get stuck in a local minimum / maximum, it highly depends on the starting point:



as we can see, on the left we got stuck in a local maximum.

Its performance highly depends on the landscape as well



6 - NK-problems: motivation, definition, goal

Inspired by biological problems -> genetic regulatory networks

NK problems belong to np-hard class !

Lets give an example of nk landscape problems:

- ➔ We consider N persons or agents, labelled with an index i, i = 1, . . ., N
- ➔ Each agent i acts according to two possible strategies, denoted xi = 0 or xi = 1
- ➔ The success of an agent depends on the strategy it chooses and the type of relation it has (competition or collaboration) with the other persons it interacts with
- ➔ If we assume that each agent i depends on K other agents, we may define a function fi(x i, ..., x i+k) which gives the profit resulting from the chosen strategy and that of the connected agents

The problem is specified once the f_i are given, as well as K, N

We want to maximize

$$f = \sum_{i=1}^{N} f_i$$ the profit of the population.

Our main interest with NK-problems is to be able to generate synthetic problems of increasing difficulty. (as N and k increase, it become more and more difficult to solve)

Example MaxOne problem.

Find a chain of N bits that maximize the number of ones.

Obvious solution: $X = (x_1 x_2 \dots x_N) = \underbrace{111\dots1}_{N \text{ times}}$

$f_i(x_i, \dots) = x_i \quad K = 1$

$$f = \sum_{i=1}^{n} f_i = \sum_{i=1}^{n} x_i$$

$f = N$

Another example

$x = x_1 x_2 \cdots x_N$ $\quad f_i(x_{i-1}, x_i, x_{i+1})$ $\quad K = 3$

f_i is max for 1 1 1 Then it is a trivial solut

Bl if f_i is large for 101 and small for 010

Then the chain 101010101

7- Tabu Search: main principles and convergence

Used for quadratic assignment problem and TSP

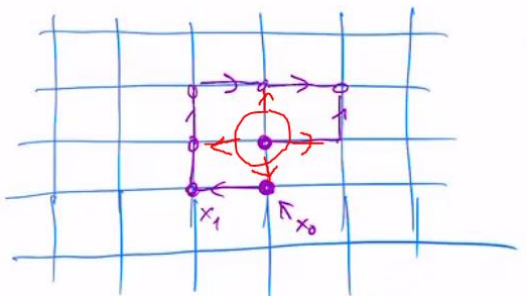
- ➔ The search space is explored by going from neighbour-to-neighbour xn → xn+1 ∈ V(xn)
- ➔ The xn+1 is chosen by the search operator, as the non-tabu xn+1 that locally optimizes fitness -> (independent of the current fitness of xn -> can be worse) -> if many points have the same fitness, it is random choice
- ➔ A tabu list, is a list of already explored states, so that we don't go back
- ➔ However, it is possible for a state to leave the tabu list -> after a certain amount of iterations

Tabu list -> prevents us from going back

The fact that xn+1 can have worse fitness than xn means we won't get stuck in local max

A state can go out of tabu list, so that the code doesn't get stuck

- ➔ This is why we need the list to be updated (if we are in red state, we are stuck):



Convergence:

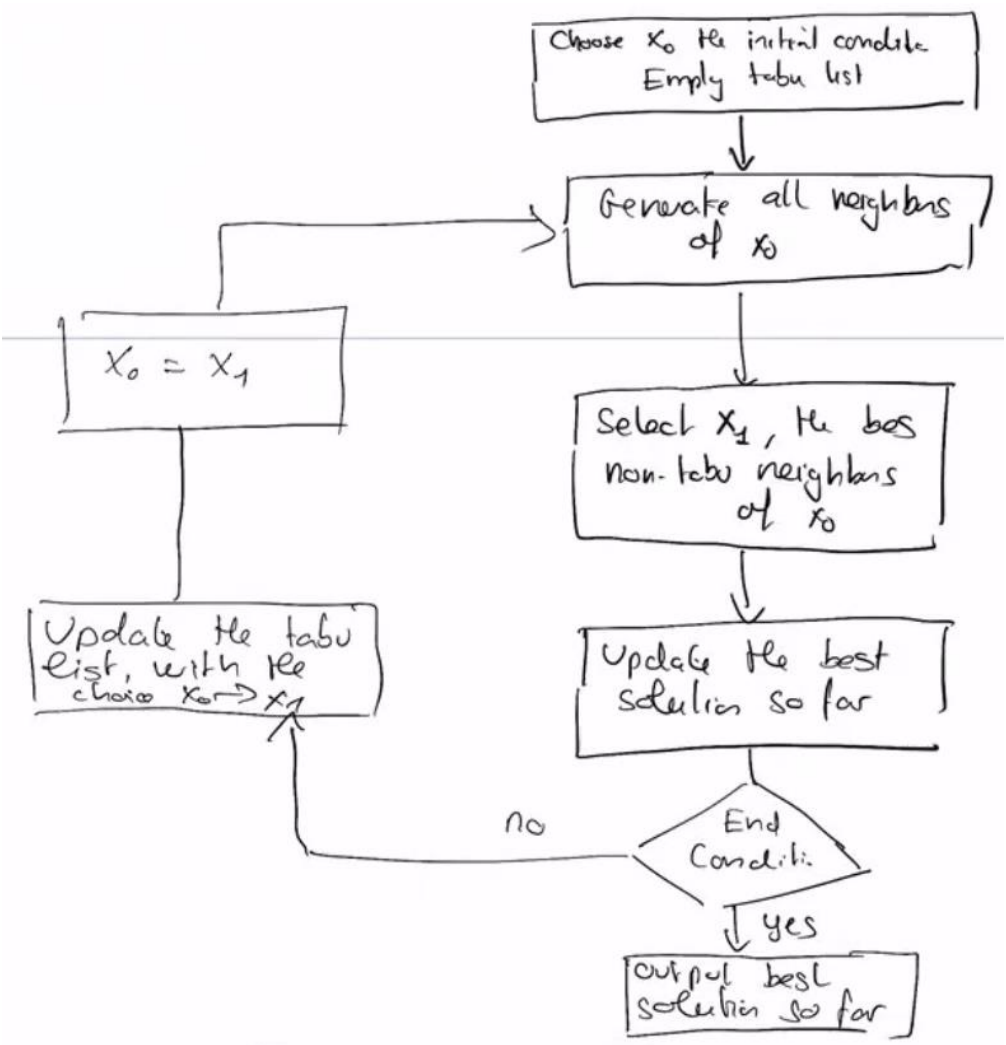
To converge means to find the global optimum!!

-tabu search will converge if:

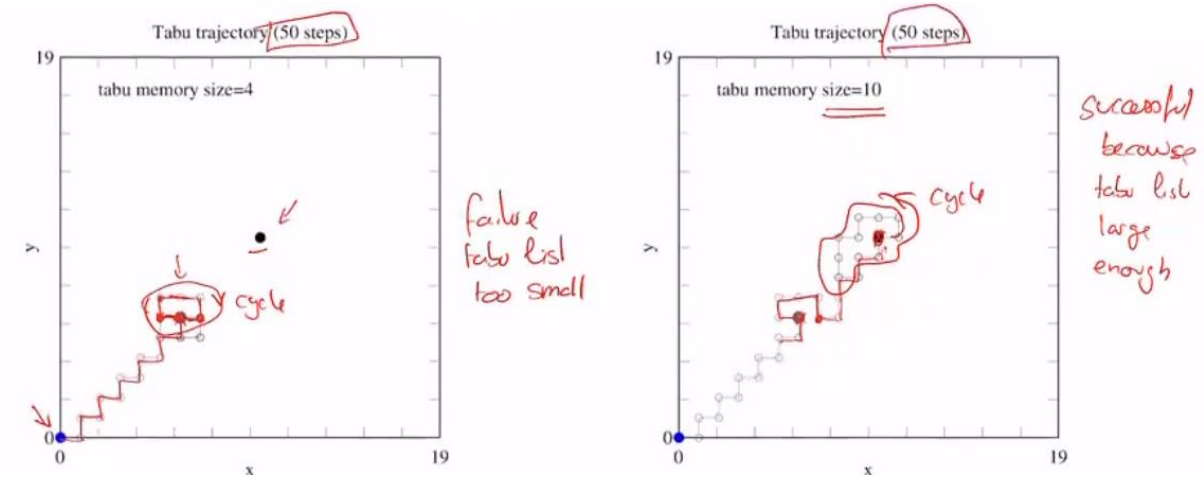
- the search space is finite
- neighbourhood is symmetric: x in v(y) ⇔ y in v(x)
- any state in the search space is reachable by a finite number of steps

- ➔ Memorizes all the visited points, but allows the trajectory to use the oldest tabu point to unlock itself -> will visit all the search space -> and find the optimal solution

Flowchart of tabu list algorithm:



Example of the same search space, with 2 different tabu list memories:



8- The different ways to implement the tabu list

The goal of the tabu list is to prevent search to explore solution that have been seen previously

There are many versions of a tabu list:

- ➔ Visited solutions kept in tabu list for N iterations
- ➔ Keep in tabu list all states that have the same fitness as already visited states (if we have visited a state with fitness n, then all neighbours with fitness n go into the tabu list)
- ➔ Movements (of transformations) that is forbidden, for example, it is impossible to go left for the next 2 moves -> TO avoid going back to x if we just moved from x to x', the inverse movement is in the tabu list

Updating the tabu list -> adding and removing items: short- and long-term memory

- ➔ Short term memory:
  - Use a finite size tabu list, so as we add to the tabu list, the oldest item is deleted
  - Associate a duration to the tabu items (banned time), a movement is banned for N iterations -> N is defined based on knowledge of the problem or by trial and error

- ➔ Long term memory
  - Used to prevent that some movements are never used; the long-term memory can disregard the forbidden movement in the short-term list
  - To implement the long-term memory, we track the movements / states / transitions never done/visited, and at a certain point we force the algorithm to visit them

- ➔ Short list = exploration -> no diversification
- ➔ Long list = exploitation -> with diversification

- We can also put fitness above all, making it ok to revisit a state that has a higher fitness, even if it was in the taboo list

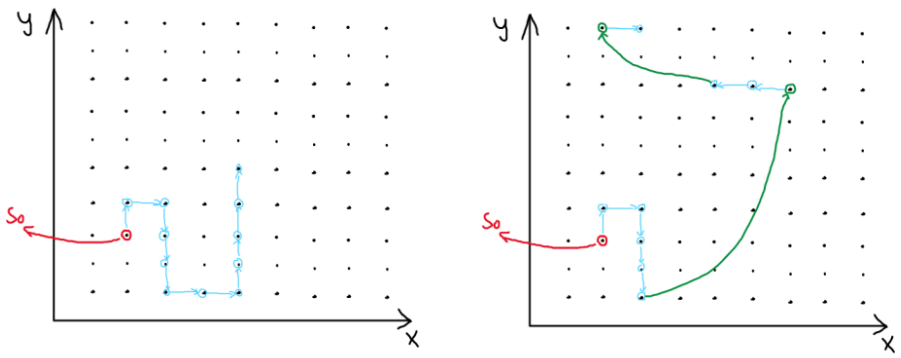


Figure 3: Exploration of Z^2 plane, left without diversification, right with diversification

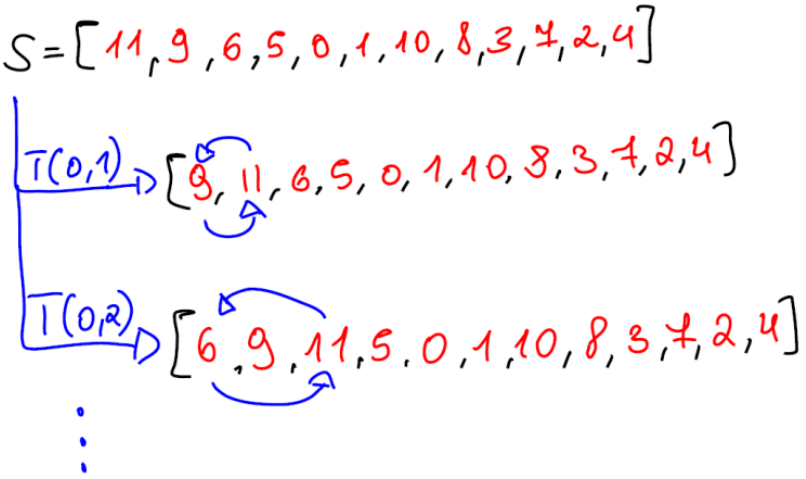
9 - Quadratic Assignment Problems

Combinatorial optimization problem

N objects and n possible locations -> must find the best permutation

- ➔ We have n objects and n locations
- ➔ We know the distance drs between locations r and s
- ➔ We are given flows fij between all pairs ij of objects

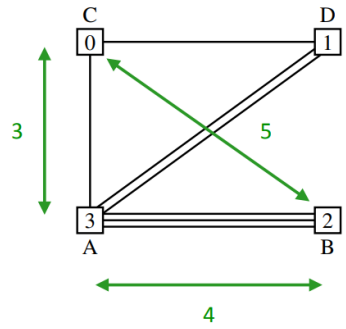
- ➔ GOAL: find the optimal location for each object -> minimize distance flow -> sum of products



$$f = \sum_{ij} f_{ij}\, d_{r_i r_j}$$

- Example: Find the best location (A, B, C, D) for each facility (0, 1, 2, 3) in order to minimize

$$I(\psi) = \sum_{i,j=0}^{n-1} w_{ij} \times d_{\psi_i,\psi_j}$$



distances	flows
$d_{AB} = d_{CD} = 4$	$w_{13} = 2$
$d_{AC} = d_{BD} = 3$	$w_{01} = w_{03} = 1$
$d_{AD} = d_{BC} = 5$	$w_{23} = 3$

Fitness $I(\psi) = w_{01} \times d_{\psi_0\psi_1} + w_{03} \times d_{\psi_0\psi_3} + w_{13} \times d_{\psi_1\psi_3} + w_{23} \times d_{\psi_2\psi_3}$

Here $\psi = (C, D, B, A)$

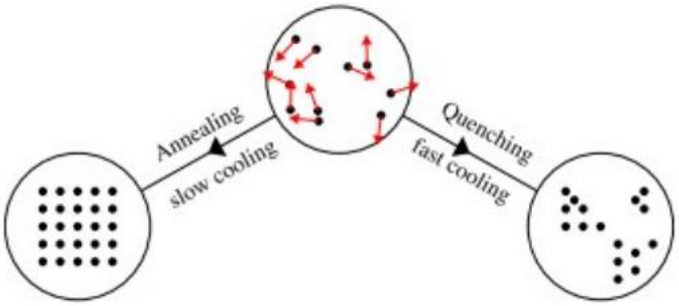
Hence

$$I(\psi) = d_{CD} + d_{AC} + 2d_{AD} + 3d_{AB} = 29$$

10 - Simulated annealing, main principles and the Metropolis rule

- Inspired by nature (by physics and metallorgy more specifically)
- Annealing is a process by which a sample is cooled down slowly -> finds global min
- Quench it is a quick cooling down -> finds local min
- Fitness function is called energy function

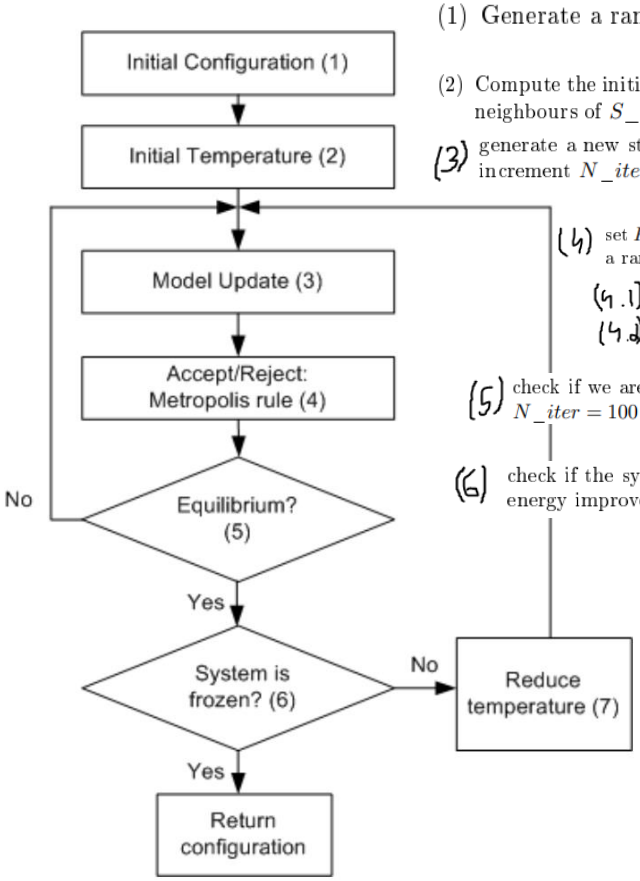
Nature minimizes energy with theses processes, and we want to capture this same property with our algorithm



➔ We start with high temperature, in this state the system explores many possible states
➔ When the temperature starts to cool down the system is "trapped" does exploitation process

The hope is to have found the global min with the use of both exploration and exploitation

Algorithm:



(1) Generate a random state S, let's call it $S\_current$

(2) Compute the initial temperature: $e^{\frac{-\langle \triangle E \rangle}{T_0}} = 0.5$, where $\langle \triangle E \rangle$ is the average energy change of 100 (random) neighbours of $S\_current$

(3) generate a new state, let this state be a random neighbour of our current state, let's call it S_new, and increment N_iter++

(4) set $P = min(1, e^{\frac{-\triangle E}{T}})$, where $\triangle E = E(S\_new) - E(S\_current)$, and T the current temperature, generate a random number r between 0 and 1:

(4.1) if $r < P$, we set the $S\_current = S\_new$, and increment $N\_accept++$

(4.2) if $r > P$, we keen our current state

(5) check if we are at an equilibrium state. We are at an equilibrium state if either $N\_accept = 12 * n$ or $N\_iter = 100 * n$

(6) check if the system is frozen. A system is frozen if for the last three temperature changes there were no energy improvements

(7) **Temperature Reduction:** Use the following simple rule:

$$T_{k+1} = 0.9T_k$$

The current state at the last step -> is the one we return

Metropolis rule:

(4) set $P = min(1, e^{\frac{-\triangle E}{T}})$, where $\triangle E = E(S\_new) - E(S\_current)$, and T the current temperature, generate a random number r between 0 and 1:

(4.1) if $r < P$, we set the $S\_current = S\_new$, and increment $N\_accept++$

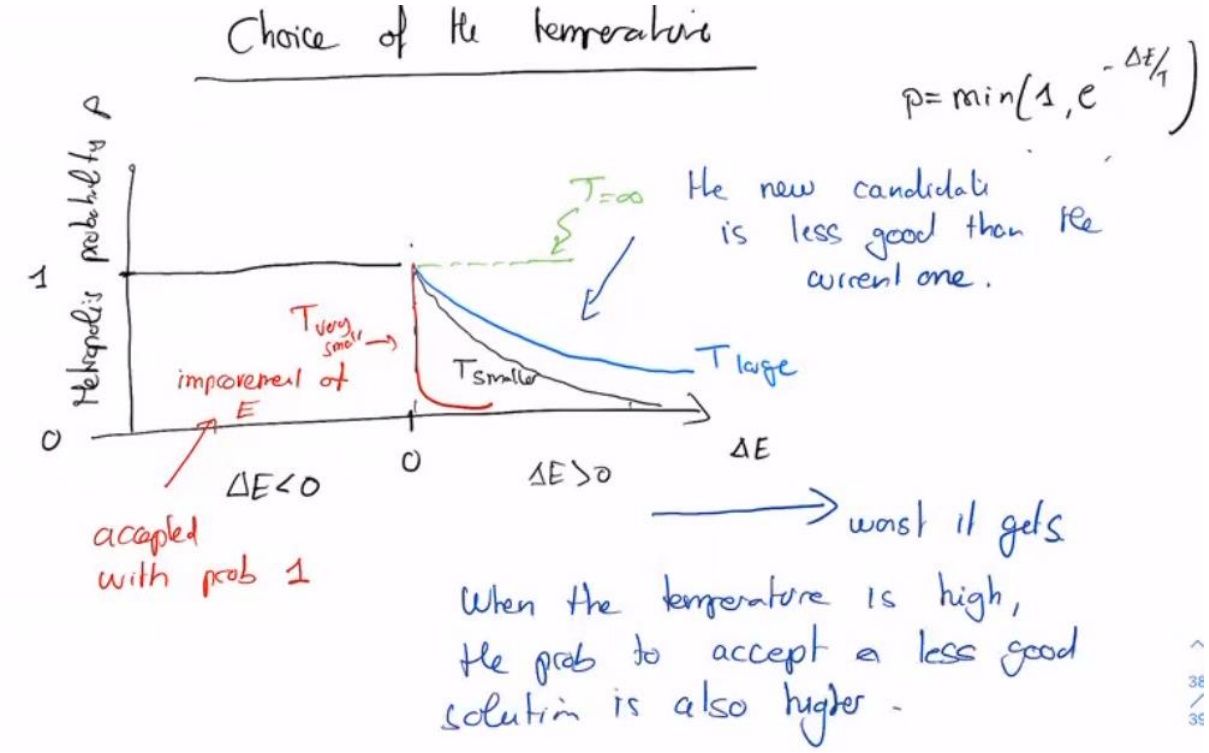
(4.2) if $r > P$, we keen our current state

The metropolis rule makes sure we always accept a lower energy state:

- Because: E(S_new) – E(S_curr) < 0 => E(S_new) < E(S_curr)
- And when E(S_new) – E(S_curr) < 0 => P = 1, we accept the new state with probability P=1

If the state is of worse energy:

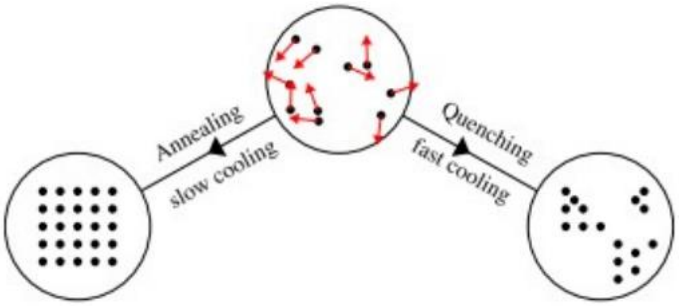
- Because: E(S_new) – E(S_curr) > 0 => E(S_new) > E(S_curr)
- Then there is a chance we accept it still, the higher the temperature, the higher chance to accepting it



11 - Simulated annealing, flow chart and choice of parameters

- Inspired by nature (by physics and metallorgy more specifically)
- Annealing is a process by which a sample is cooled down slowly -> finds global min
- Quench it is a quick cooling down -> finds local min
- Fitness function is called energy function

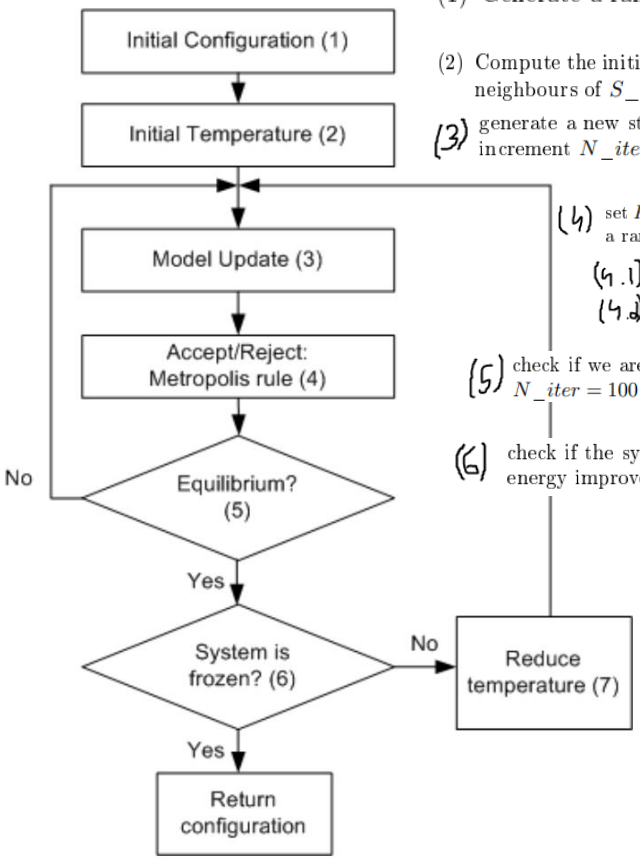
Nature minimizes energy with these processes, and we want to capture this same property with our algorithm



➔ We start with high temperature, in this state the system explores many possible states
➔ When the temperature starts to cool down the system is "trapped" does exploitation process

The hope is to have found the global min with the use of both exploration and exploitation

Algorithm:



(1) Generate a random state S, let's call it $S\_current$

(2) Compute the initial temperature: $e^{\frac{-\langle \triangle E \rangle}{T_0}} = 0.5$, where $\langle \triangle E \rangle$ is the average energy change of 100 (random) neighbours of $S\_current$

(3) generate a new state, let this state be a random neighbour of our current state, let's call it S_new, and increment N_iter++

(4) set $P = min(1, e^{\frac{-\triangle E}{T}})$, where $\triangle E = E(S\_new) - E(S\_current)$, and T the current temperature, generate a random number r between 0 and 1:

(4.1) if $r < P$, we set the $S\_current = S\_new$, and increment $N\_accept++$

(4.2) if $r > P$, we keen our current state

(5) check if we are at an equilibrium state. We are at an equilibrium state if either $N\_accept = 12 * n$ or $N\_iter = 100 * n$

(6) check if the system is frozen. A system is frozen if for the last three temperature changes there were no energy improvements

(7) **Temperature Reduction:** Use the following simple rule:

$$T_{k+1} = 0.9T_k$$

The current state at the last step -> is the one we return

choice of parameters:

equilibrium:

at step (5), to check if we are at an equilibrium, we check that we have tried 100*n states, or accepted 12*n states -> depending on these values it will be faster or slower have less good or better results

system frozen:

a system is frozen – stop condition – if for the last 3 temperatures there was no fitness improvement -> can increase the number or decrease

temperature:

tk+1 = tk *0.9 -> can lower 0.9 to be in quenching (faster cooling) -> usually worse results

initial temperature might be too low if our random initial state has a low energy

12- Convergence of simulated annealing: how to formulate it, how to compute it.

➔ Does simulated annealing find the global optimum and under which conditions?

Simulated annealing can be analysed mathematically, it converges in probability.

It gives a solution arbitrary close to the global optimum with a probability arbitrary close to 1

➔ The conditions are:
- Movements are inversible
- Any point in the search space can be reached from any other point in a finite number of movements
- The initial temperature must be large enough (do exploitation before exploration)
- The temperature should not decrease too fast, at iteration t:

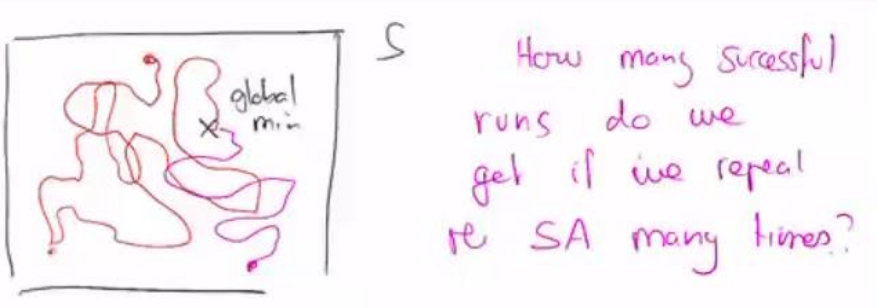
$$T(t) \sim \frac{c}{\log t} \quad \text{for } t \gg 1$$

C is an unknown constant that reflects the variation of E in S.

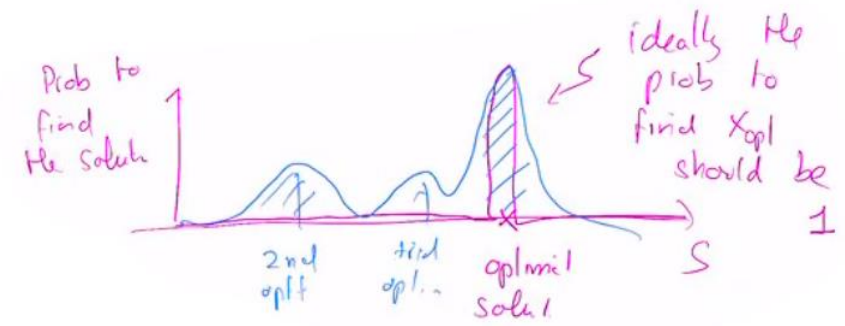
!!! If temperature is decreased too fast, we risk not converging (quenching) !!!

➔ In practice, log t is too slow. So, temperature will decrease faster

How to compute it:



If we get 100% of success there is convergence, in practice it may end up sub optimal



We want to compute the probability $P(t, i)$ that the SA is at point $i \in S$ at iteration t

$$P(t+1, j) = \sum_i P(t, i) W_{ij}(t)$$

W_{ij} is the transition probability from i to j. Since we have 100 possible values of i an j, W_{ij} is a 100×100 matrix.

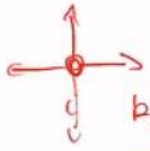
$|S| \quad |S|$

For SA, we have:

$$W_{ij} = \begin{cases} 0 & \text{if } j \text{ is not a neighbor of } i \\ \frac{1}{k_{out}(i)} \times P_{metropolis}(E_i, E_j, T(t)) & \end{cases}$$

$k_{out}(i)$: number of neighbors of i

$P_{metropolis}(E_i, E_j, T(t)) = \min\left(e^{-(E_j - E_i)/T(t)}\right)$



We also add the following

$$W_{ii} = 1 - \sum_{j \text{ neighbor of } i} W_{ij}$$

13- Ant-like algorithms: swarm intelligence, the pheromone trail, and observations about real ants.

These are algorithms that try and mimic effects we see in nature

➔ Ants as a group can solve difficult tasks, like finding the best path between the ant colony and a food source

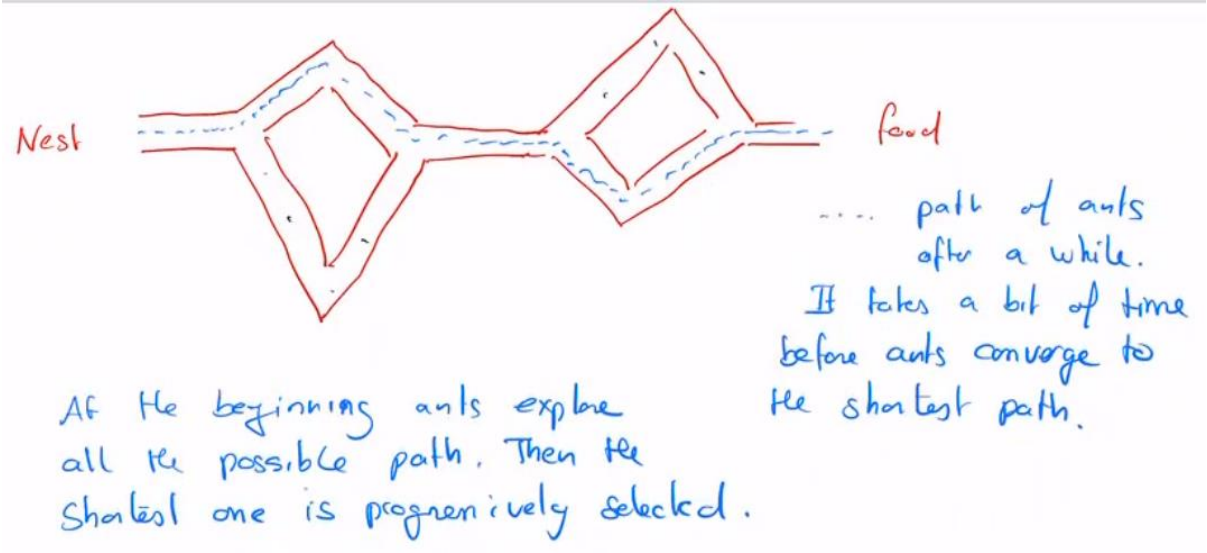
- The important properties of swarm intelligence are:
    - Collaboration is the key to the group's success (the intelligence of the group is more than the sum of the individuals intelligence)
    - This effect of swarm intelligence is also called complex systems, which are a collection of simple entities that interact
    - It results in an emergent behaviour which cannot be understood by any single entity
    - Such behaviours are visible through large scale spatial and temporal organization, sometimes called self-organizations
    - **!!!** there is no central control, every action is spontaneous **!!!**

Advantages:

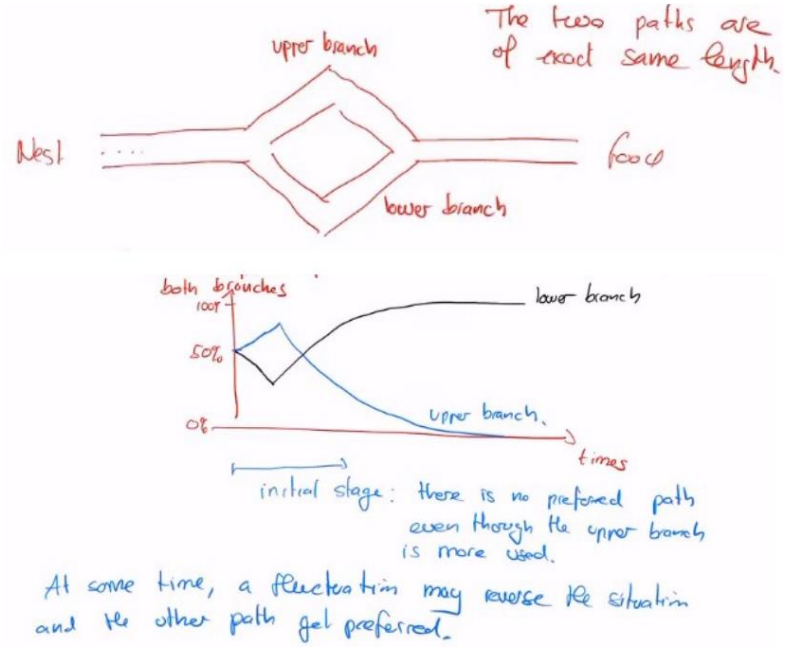
➔ Robustness to fault tolerance: the process can continue even if an entity disappears
➔ Possibility of adaptation to new situation
➔ Natural parallelism in the process, since all entities are 'independent'

Pheromone trail:

➔ Ants use pheromones to find the shortest path, ants can deposit pheromones to attract other ants to a promising location
➔ This process is called chemotaxy
➔ Ants choose the path that smells the strongest
➔ Pheromones will evaporate over time (how ants forget a suboptimal path)



This experiment was run 10 times, and 9 times out of 10, 100% of the ants were on the shortest path. The 10th run, only 80% were on the shortest path.



Explanation:

(1) We start with no pheromones on the path, so every ant chooses a path at random, leaving pheromones in the trail
(2) At the second run, it uses the pheromones to choose a path, this path then receives even more pheromones, which attracts even more ants, and so one
(3) At the end they end up with the shortest path

14- Ant System: description of the algorithm for the TSP problem

Algorithm 1

```
for all t = 1, ..., t_max do
    for all ant k = 1, ..., m do
        choose a city at random
        while there exists a city not visited do
            choose a city j according to (1)
        end while
        mark a path according to (3)
    end for
    update all paths according to (2)
    Keep the best of solutions obtained at last iteration
end for
```

(1)

$$p_{ij}^k(t) = \begin{cases} \frac{(\tau_{ij}(t))^\alpha (\eta_{ij})^\beta}{\sum_{l \in J} (\tau_{il}(t))^\alpha (\eta_{il})^\beta} & \text{if } j \in J \\ 0 & \text{otherwise} \end{cases}$$

First off, J is a set of cities not yet visited in the ant's path, so only cities unvisited can be the next city chosen.

Each unvisited city has a probability of being chosen, this probability is a function of:

(Pheromones in a path to a city * visibility of the city)/

(Sum of all pheromones*visibility for all cities unvisited)

(2)

$$\tau_{ij}(t+1) = (1-\rho)\tau_{ij}(t) + \sum_{k=1}^{m} \Delta\tau_{ij}^k(t)$$

This function allows us to update the pheromones in each path at each iteration, there is an evaporation rate ρ, so each path has some of its pheromones that evaporates + an integer that is a function of how many ants took this path -> this value is given by formula (3)

(3)

$$\Delta\tau_{ij}^k(t) = \begin{cases} \frac{Q}{L^k(t)} & \text{if ant } k \text{ used edge } (i,j) \text{ in its tour} \\ 0 & \text{otherwise} \end{cases}$$

Q is a constant, Lk is the length of the path taken by the ant -> the longer the path -> the smaller is the result -> less pheromones in this path during the next iteration

Choice of Parameters

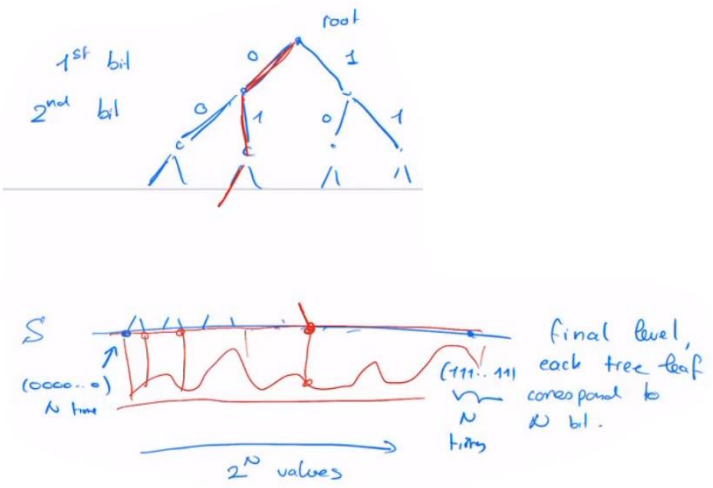
- $\alpha = 1$, $\beta = 5$, $\rho = 0.1$
- $Q = L_{nn}$, $\tau_0 = \frac{1}{L_{nn}}$
- m , t_{max}

15- Ant algorithms: the simple version in {0,1}^N, and discussion of performance

What we are doing here is a benchmark of this algorithm!

- We will consider a simpler version of the ant algorithm
- We want to find a bit string which maximizes a given fitness function: S={0,1}^N
- This makes it so the search space is a tree structure, at each step (node) we have two options for the next bit, 0 or 1. This tree has length of N

Visually it looks like this:



For every value of the search space -> 2^n, there is a fitness function that evaluates the path, then the deposited pheromone on this path is a function of the fitness function result

➔ At each bifurcation an ant will more likely choose the path that has the most pheromone, each path has the following probability of being chosen:

$$P_{left} = \frac{\tau_{left}}{(\tau_{left} + \tau_{right})} \qquad P_{right} = 1 - P_{left}.$$

Statistiques sur 10 répétitions, chacune avec 6 fourmis, après 6 itérations.

effort computationnel	$6 \times 6 = 36$
taux de succès :	7/10
taux de succès à 3% de l'optimum	9/10

we explore 36 configurations
success rate

What about a random search ?
Let us take 36 points of S', at random. What is the probability to find the optimal value within those 36 solutions?
$p = 1/64$ is the prob to find the optimal value at random.
With 36 attempt, the prob to find it is
$0.43 = 1 - (1-p)^{36}$ is the prob to find it at least once.

16 - Particle Swarm Optimization: algorithm and example

- Explores collectively the search space in order to find the optimal solution
- An individual will, at every iteration, choose his path based on his own best path, his current path, and the groups best path
- The hope is that the group of particles will find the best solution

ALGORITHM

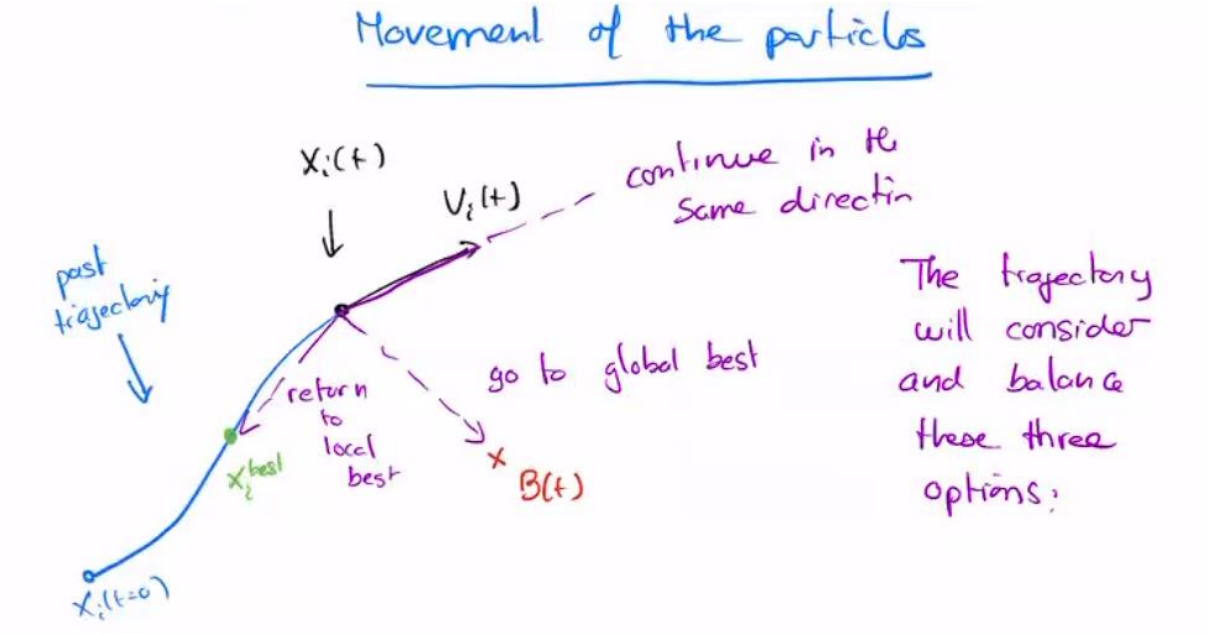
- ➔ It is a population metaheuristic (at each iteration the number of individuals stays constant)
- ➔ Xi(t) is the position of particle I at iteration t
- ➔ Each xi is a possible solution and it has a corresponding fitness
- ➔ Particles explore the search space because they have a velocity, which makes them move from iteration t to iteration t+1

At each iteration each particle will update its local best (his own best solution)

$$X_i^{best}(t) = \begin{cases} X_i(t) & \text{if } f(X_i(t)) \text{ is better than } f(X_i^{best}) \\ X_i^{best}(t-1) & \end{cases}$$

At each iteration the global best is also updated -> B(t)

$$B(t) = \underset{X_i^{best}}{\operatorname{argmax}}\ f(X_i^{best}) \quad \text{(for a maximization problem)}$$



current local best

$$\begin{cases} V_i(t+1) = \omega V_i(t) + c_1 r_1(t+1)\left[x_i^{best}(t) - x_i(t)\right] \\ \qquad\qquad + c_2 r_2(t+1)\left[B(t) - x_i(t)\right] \\ x_i(t+1) = x_i(t) + V_i(t+1) \end{cases}$$

Global

- ω is a parameter or $\omega < 1$ is an inertia parameter. How much of the previous velocity we will keep
- c_1 is called the cognitif coefficient as it takes into account the knowledge of the individual.
- c_2 is called the social coefficient as it considers the knowledge of the group. $c_1 = c_2 \cong 2$
- r_1 and r_2 are random numbers uniformly distributed in $[0, 1[$

- ➔ Everything is a vector!!
- ➔ Particles are placed randomly in S with v = 0
- ➔ We need to have search boundaries, many options are available, like a particle bouncing etc...
- ➔ We maximize V with Vmax

This algorithm must be applied to a problem that defines velocities addition multiplication etc etc

We can use PSO for computing weights for neural networks

initialize all positions randomly-> comptue the current fitness for every particle-> check if the current fitness is better than the local best for each partic

17 - The Firefly algorithm

- Inspired by PSO
- Fireflies attract a mate or prey by emitting light, the higher intensity the higher the attraction is
- For continuous optimization, but exists for discrete

ALGORITHM

Each firefly i, at iteration t, is at location $x_i(t)$ in the search space: $x_i(t) \in S \subseteq \mathbb{R}^d$

Each firefly emits a light with an intensity $I_i(t)$ which depends on the fitness of solution $x_i(t)$

For a maximization problem, we can simply say $I_i = f(x_i(t))$ where f is the fitness.

At each iteration, the fireflies move according to:

- One considers all pairs (i, j) of fireflies with $1 \leq i \leq n$, $1 \leq j \leq n$, where n is the number of fireflies
- If $I_i < I_j$ then firefly i moves towards firefly j, according to the observed attractivity (↑ to be defined)

Attractivity:

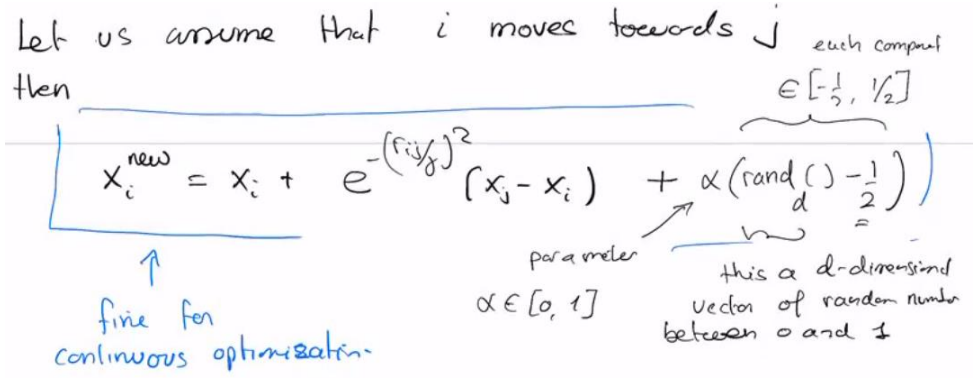
$$\exp\left(-\left(\frac{r_{ij}}{\gamma}\right)^2\right) \in [0, 1]$$

where r_{ij} is the distance separating fireflies i and j and γ is a parameter weighing this distance.

This quantity will be the fraction of the distance r_{ij} that the less intense firefly will move towards the more intense one.

Depending on how we define γ this attractivity will affect far away fireflies, depending on how it is tuned it will be exploration vs exploitation

Movement:



Code:

- ➔ Initialize the fireflies position randomly in the search space
- ➔ for every couple (i,j) of fireflies
- ➔ if intensity of i < intensity of j, we update position and intensity of firefly i

- ➔ we don’t change firefly j!!!

18- Genetic algorithms: inspiration and algorithm

- ➔ Inspired by Darwin evolution
- ➔ On représente les individus dans S par des chromosomes ou son patrimoine génétique
- ➔ Le degré d'adaptation est donné par la fitness
- ➔ The population evolves at each generation
- ➔ We choose the best individuals over the less fit ones
- ➔ The population size stays constant over generations

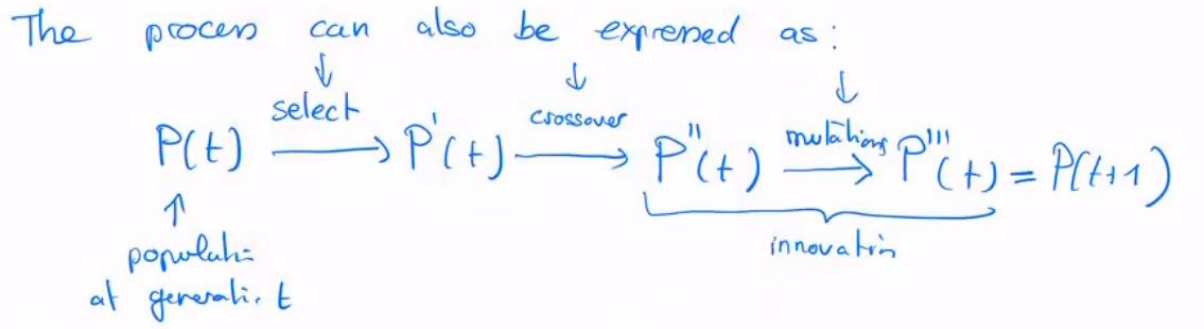
- ➔ We will mostly consider maximization problems

ALGORITHM:

- ➔ We start with a randomly generated generation
- ➔ The next generation is derived from the previous one by following the steps bellow

  (1) Select the best individuals in the population (many strategies can be applied here)
  (2) Apply recombination/ crossover among the individuals as well as mutation

- ➔ If the genetic process loses the best individual, it is readded to the population by replacing the current worse! Which means we track the global best



The selection process focusses on exploitation, and mutation focus on exploration

Crossover does a bit of both -> exploration a bit more

19- The different selection operators

There are several ways to implement the selection step

(1) Fitness proportional

Draws with replacement, we choose randomly n times an individual from the current population. The induvial is chosen proportional to its fitness

More chance to choose a high fitness individual

There can obviously be copies of individuals

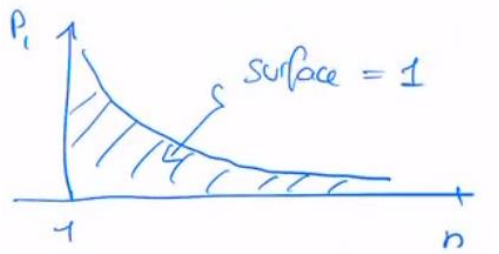
Il faut faire attention à s'il y a des individus avec une fitness négative

$$p_i = \frac{f(s_i)}{\sum_{k=1}^{n} f(s_k)}$$

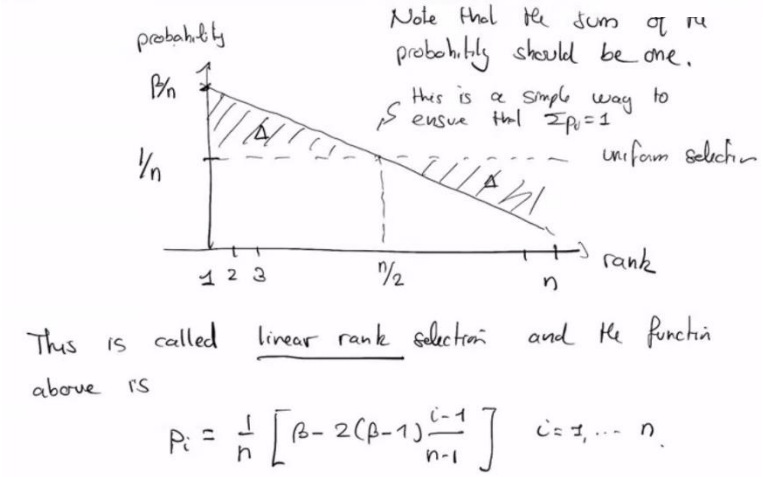
(2) Selection by rank

We set the individuals in an order depending on their fitness value (by rank), this works with negative fitness results, as well as for minimization problems

For each rank we assign a probability of the individual being chosen, by having higher probability for higher ranked individuals



(3) Selection by linear rank



This is called linear rank selection and the function above is

$$p_i = \frac{1}{n}\left[\beta - 2(\beta-1)\frac{i-1}{n-1}\right] \quad i = 1, \dots n$$

$\beta > 1 \rightsquigarrow p_1 > p_n$

$\beta < 2 \rightsquigarrow p_n > 0$

(4) Selection by tournament

k-tournament selection -> draw k random elements from the population, each individual having 1/n probability of being chosen

from the k chosen individuals, select the one with the highest/lowest fitness depending on the problem

do this process n times

➔ One doesn't need to define a fitness function, just needs to be able to compare individuals

20- The takeover time in genetic algorithms: goal, definition, value for tournament selection.

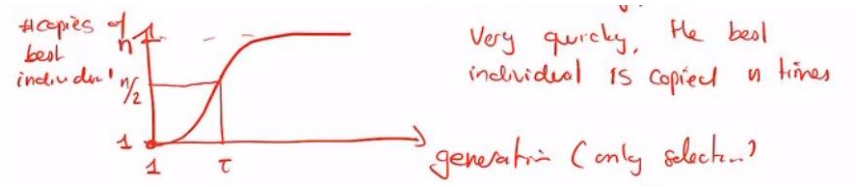
Quantify how fast a selection mechanism eliminates all but the best individuals

Let's define τ the takeover time as the number of selection rounds (no crossover and mutation), to have only the best individuals

It measures the intensity of selection:

- τ Is small -> strong selection
- τ Is large -> weak selection

"large" and "small" is proportional to size of population n -> we will show that $\tau = O(\log n)$



➔ Selection is often exponential

Theory:

➔ The number o copies m(t) of the best individual at generation t goes as:

$$\frac{dm}{dt} = \alpha m \left(1 - \frac{m}{n}\right)$$

prop. to the current number

limitation due to a finite population of size n

This solution for tournament selection:

At each selection we have a probability of choosing the best individual:

$\left(1 - \frac{m}{n}\right)^k$ ← repeated k times

↑ prob to take the best

prob not to take the best

As well as the probability of choosing the individual at least once:

$$\left[1 - \left(1 - \frac{m}{n}\right)^k\right]$$

$$m(t+1) = n\left(1 - \left(1 - \frac{m}{n}\right)^k\right)$$

$$\geq n\left(1 - \left(1 - \frac{m}{n}\right)^2\right)$$

$$= n\left(1 - \left(1 - \frac{2m}{n} + \frac{m^2}{n^2}\right)\right)$$

$$= n\left(\frac{2m}{n} - \frac{m^2}{n^2}\right)$$

$$= m\left(2 - \frac{m}{n}\right)$$

with $t = \ln(n-1) \Rightarrow m(t) = \frac{n}{2}$

To have half of the population filled with the best individual, one has to perform $O(\log n)$ selections.

21- Genetic Programming: goal, fitness evaluation, function and terminal Sets

The goal is to apply evolutionary ideas to computer programs

- ➔ Make the code evolve to solve the problem
- ➔ We will consider a population of "computer programs", that will be selected, recombined and subject to mutation
- ➔ Until we reach the desired solution
- What can be expected of such programs? -> they should discover a function that produces the right output for a given input

FITNESS:

Let us consider a program p such that: y = P(x) -> x and y can be complex data structures

Let A be a training set: A = {<x1,y1>,<x2,y2>, … , <xk,yk>}

For input xi we know the output yi

Which means the fitness will be computed by: $f(P) = \sum_{i=1}^{k} |y_i - P(x_i)|$

FUNCTION SET AND TERMINAL SET:

The space of programs that we can build is specified by 2 sets:

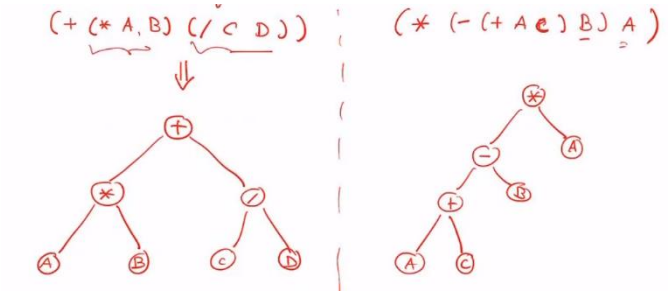
- F: set of nodes functions {+,*,-,/}, {AND, OR, NOT} -> these are the operators that modify the operands
- T: set of terminal nodes {A,B,C,D} -> these set contains the variables and constants involved in the problem

22- Genetic Programming: tree representation, initialization, crossover, mutation, and bloat.

Tree representation: -> it's an S-expression in a visual form, easier to understand the order of operations

operator
operand

(S-expression: (+ x 2) -> x+2)



INITIALIZATION:

- As always, the population is generated randomly with values from the termina as well as function set. With this tree representation, the terminal sets are the leaves, and the function set is the nodes
- We can set a probability p, to draw from F set, and 1-p to draw from T set
- Probability p will influence the depth of the tree
- It is wise to have a dynamic probability as we go deeper in the tree, at the beginning favouring a draw from F set, and later favouring a draw from T set

CROSSOVER:

- We exchange at random the sub-trees of the parents at random

MUTATION:

- We chose a random node and rebuild randomly the sub-tree from this node
- We can change the mutation probability depending how deep we are in the tree

BLOAT:

By doing crossover and mutation our functions may become more and more complex

In practice we should prevent this effect by having max length parameters in place

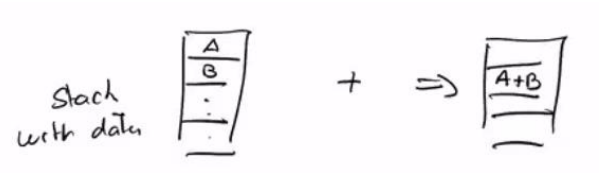
Sometimes it is possible that a subtree can be simplified such as (- x x)

23- Genetic Programming: the stack-based, instruction-driven representation.

The idea is to get closer to a procedural programming language -> set of instructions executed sequentially

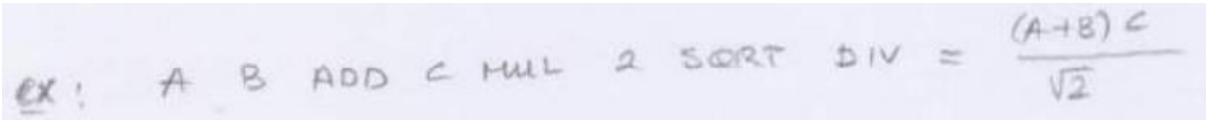
To make sure that a program stays valid after the genetic transformation we use a **stack based** approach

The + operator consumes both top values A and B, performs the operation, and adds the result to the top of the stack



If we code as such, we will be able to have individuals that all have the same length

This will be called **instruction driven representation**

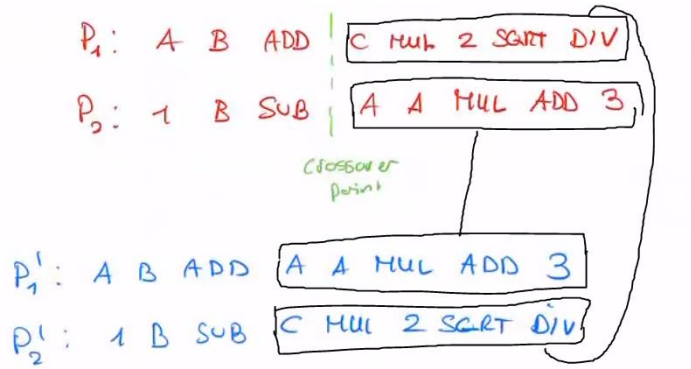


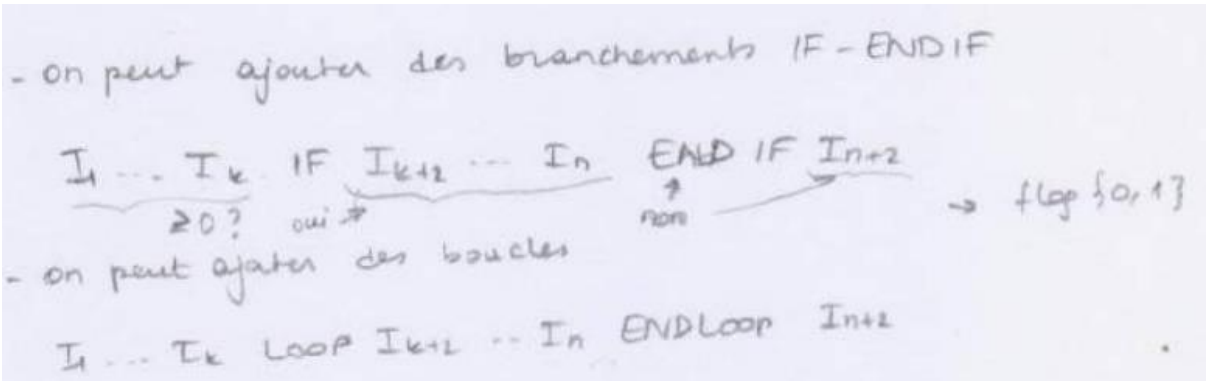
However, it is possible that at some point there aren't enough values in the stack to do an operation (1), or that we end with more than 1 value on the stack (2), if this happens, we:

(1) Skip this operation and leave the stack as is, do the next instruction
(2) Return the value on top of the stack

MUTATION AND CROSSOVER:

We cross the strings of instructions with each other for crossover, and mutation is done with a mutation probability





24- Evolution Strategy: individual and population versions.

Single individual metaheuristic $x \in \mathbb{R}^d$ evolving by mutation only

Population version also including crossover

Metaheuristic for continuous optimization $S \subseteq \mathbb{R}^d$

(1+1) ALGORITHM -> 1 parent and 1 child

$$x' = x(t) + N(0, \sigma)$$

$$x(t+1) = \begin{cases} x' \to f(x') > f(x) \\ x \to f(x) < f(x) \end{cases}$$

- The child is produced by a gaussian mutation of the parent:
- X' is the child, x(t) the parent, and N(0, σ) is the mutation
- The mutation follows a normal distribution
- The child is only accepted if its fitness is better than the parents

➔ The $x \in \mathbb{R}^n$ which means σ is also a vector

➔ $\sigma = (\sigma_1, \sigma_2, \dots, \sigma_d)$

➔ Each dimension can have a different σ

➔ (1+1)-ES is much like a random walk search with hill climbing strategy

➔ If we are accepting too many children, we increase σ, to make larger jumps

➔ If we accept too few children, we decrease σ

➔ We should want to be around 1/5 acceptance rate

POPULATION ALGORITHM

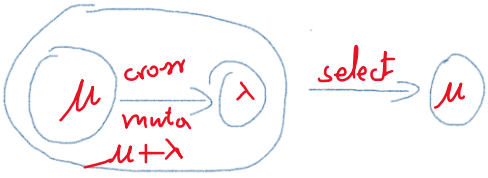
The size of the population (nb of solutions) is denoted μ

Each individual is represented as (x^i, σ^i) $i = 1, \dots, \mu$

$x^i \in S$, σ^i Is the associated mutation parameter

There are 2 variants of the population approach:

(1) $(\mu + \lambda)$-ES: λ children generated from μ parents



(2) (μ, λ)-ES: in this case $\lambda > \mu$ children are generated from the μ parents, μ are selected

This last selection is deterministic, we choose the μ best individuals

CHILDREN GENERATION

MUTATION

➔ Choose 2 parents among the μ possible parents

➔ Apply mutation to that child

➔ Repeat λ times to generate λ children

$$\begin{cases} x' = x + N(0, \sigma) \\ \sigma' = \sigma + e^{N(0, \Delta\sigma)} \end{cases}$$

$\Delta\sigma = 1/\sqrt{d}$

Crossover

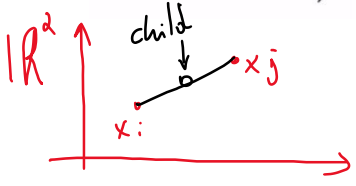
The crossover is performed both on X and σ

. The uniform crossover is the following

$$(x_e^{child}, \sigma_e^{child}) = (x_e^{\begin{cases} parent_1 & \text{prob } 1/2 \\ parent_2 & \text{prob } 1/2 \end{cases}}, \sigma_e^{\begin{cases} parent_1 & \text{prob } 1/2 \\ parent_2 & \text{prob } 1/2 \end{cases}})$$

or one can also use the arithmetic crossover. Let us consider two parents, i and j

$$(x^{child}, \sigma^{child}) = \frac{1}{2}(x^i, \sigma^i) + \frac{1}{2}(x^j, \sigma^j)$$

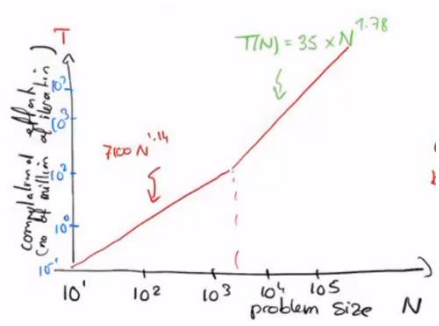


25- Performance of metaheuristics: examples, specificities of the performance evaluation, metrics, approach, "No Free Lunch" theorem

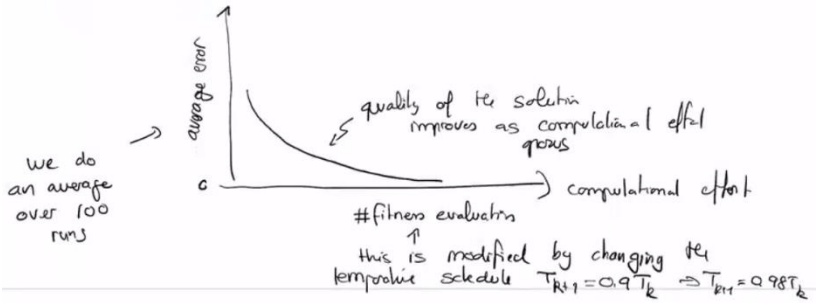
- ➔ No guarantee of quality of solution
- ➔ They are stochastic, so the behaviour is always different
- ➔ To determine the success of an algorithm in a problem, we must use statistics

EXAMPLE:

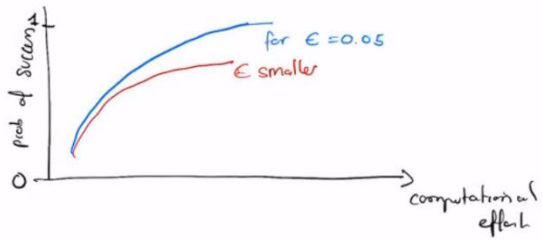
- ➔ Let's consider SA for TSP, we have n towns randomly placed in a 2x2 spatial region
- ➔ We will generate many such problems, varying N
- ➔ Our question is: how long does it take does it take for SA to give an answer? (This does not mean that the answer is the global optimum, but when SA stops)
- ➔ Performance speed:
  - With N $\in$ [20,2000], the difficulty is linear $O(N^{1.14})$
  - With N $\in$ [5000,50000], difficulty is $O(N^{1.78})$ which is smaller than quadratic $< O(N^2)$
  - It is also much better than exhaustive search $O(N!)$



- ➔ Performance quality (1):
  - To compute the quality of a solution one must know the best solution for a given problem
  - We will place 50 cities in a circle and see how SA finds an accurate solution



- ➔ Performance quality (2):
  - We can also compute the probability of success, out of 100 runs, how many gave us the global optimum? (accuracy of ϵ)



The metrics used to evaluate a metaheuristic are:

- The complexity in time to get a solution
- Average error as a function of computational effort
- Probability of success
- Statistics are needed, between 100 and 1000 runs

Question: if metaheuristics A is better than B on a given problem, can we conclude that A is always better?

Let us consider a finite search space S of size $|S|$

We consider fitness function

$f: S \to Y$ where Y is a finite subset of $\mathbb{R}$

All possible problems are then specified by a given f sampled from a number $|Y|^{|S|}$ of possibility.

One considers a computational effort m, meaning that we consider a trajectory of exploration of m points

$(x_i, f(x_i))$ $i = 1, \dots m$

] - m iterations

Formulation of NFL:

- A metaheuristics cannot be better than another one on all possible problems.
- For any performance metric, no algorithm will be better if all discrete fitness function are considered.
- If A is better than B on a given class of problems, there is another class where B will be better.

NFL Theorem

Let $P(d_m | f, m, A)$ the probability that trajectory $d_m = \{(x_1, f(x_1)), \dots (x_m, f(x_m))\}$ generated by metaheuristics A contains the optimal value of f.

$$\sum_f P(d_m | f, m, A) = \sum_f P(d_m | f, m, B)$$

Thus, on average all metaheuristics behaves the same when compared on all possible problems.

But in practice, not all possible problems have the same probability and some are highly pathological and not realistics.

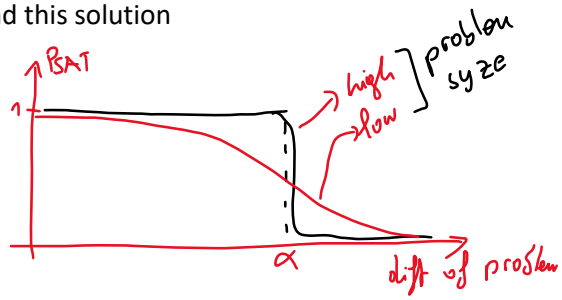
26- Phase transition in optimization problems: problem description and properties.

We will analyse statistically a metaheuristic for solving problems of increasing difficulty

We will consider a sub-class of satisfaction problems

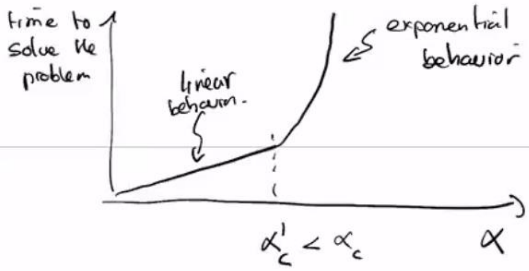
We will analyse this problem analytically, we will define the probability that a random instance of the problem has a solution and whether the metaheuristic will find this solution

The specificity of the problems we will consider is that:
- if the difficulty parameter is low, the probability that a random instance has a solution is one.
- If this parameter increases up to the critical value the probability drops abruptly to zero.



This abrupt behaviour is called phase transition

We will also see that the metaheuristics used to find solution has another phase transition in the CPU time needed to solve the problem:



SAT Problem has N Boolean variables and M Boolean equations

We want to find an assignment of these N variables that satisfy all M equations

The goal is to find an assignment of these N variables that satisfy all M equations, if it exists, we say the problem is satisfiable

These problems can be turned into an optimization problem, the goal is to minimize the energy E, which is the number of unsatisfied equations

➔ Example with only XOR operation

$$\begin{cases} x_1 + x_2 + x_3 = 1 \\ x_2 + x_4 = 0 \\ x_1 + x_4 = 1 \end{cases}$$

N = 4 (4 variables x_1 x_2 x_3 and x_4)

M = 3 (3 equation)

$(x_1\ x_2\ x_3\ x_4) = \{(1\ 0\ 0\ 0), (0\ 1\ 1\ 0)\}$

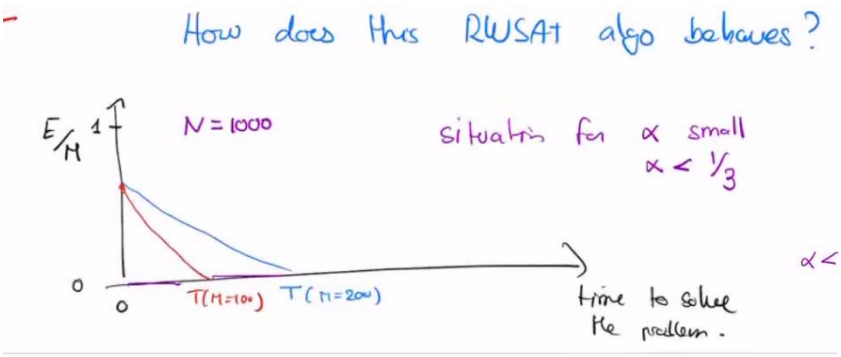
➔ This problem has a solution, so the minimal energy is 0, if a problem can't be satisfied, then its energy optimal is 1

27- Phase transition in optimization problems: the RWSAT algorithm and its behaviour.

RWSAT (random walk SAT) can be used to find solution to any SAT problem, in particular XORSAT problems.

Pseudocode:

- ➔ N variables, M equations, k variables per equation
- ➔ Initialize all N variables at random
- ➔ Compute E # of UNSAT equations
- ➔ t=0 # number of iterations
- ➔ while E>0 and t< tmax:
  - choose at random one of the non-sat equations
  - choose at random one of its k variables
  - set this variable to its complement # 1 -> 0, 0 -> 1
  - compute E #might have worse energy than before the change
  - t = t + 1
- ➔ print E, t # t is time to solution



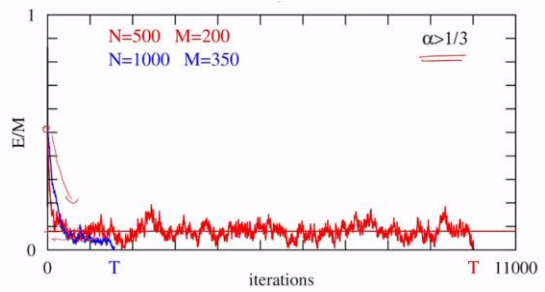
$\alpha < 1/3$ { $\langle T_{res} \rangle = N t_{res}(\alpha)$ (average time to solve the problem)

Resolution time is linear with N, with a coefficient that increases with M

We see that a solution is found (E=0) in a time which increases with M (as $\alpha = \frac{M}{N}$)

HOWEVER, when α = M/N increases, the average execution time becomes exponential

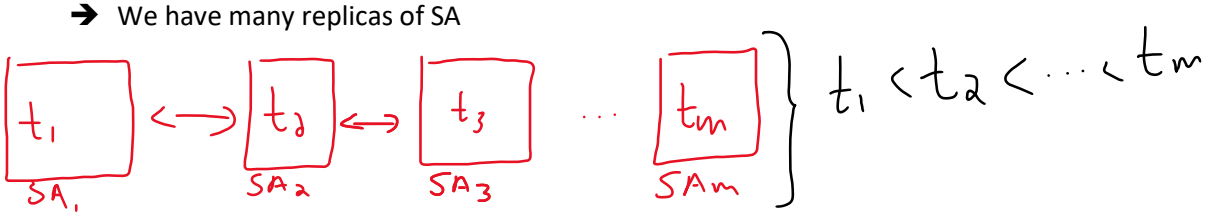
$\alpha > 1/3$ and of course $\alpha < \alpha_c = 0.4179$ { $\langle T_{res} \rangle = \exp(N \tau_{res})$; $\tau_{res} = \tau_{res}(\alpha lpha)$



as we get to 0, we start oscillating

28- The parallel tempering.

- Parallel version of SA
- We have many replicas of SA



- As opposed to standard SA, here the temperature of each Sai is constant (might change just a bit), all these Sai run in parallel. They will interact by exchanging configurations (current solution)
- Neighbouring systems can exchange their current solution according to a probability law
- Let's consider 2 systems, i and j=i+1 (neighbours) with temperatures Ti and Tj, and current solution Ci and Cj, with energy Ei and Ej

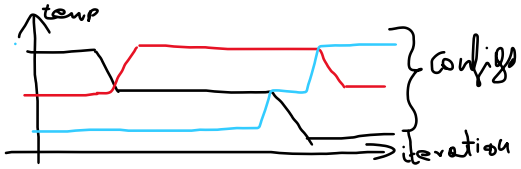
$$p_{ij} = \min(1, e^{-\Delta_{ij}}) \quad \Delta_{ij} = (E_i - E_j)\cdot\left(\frac{1}{T_j} - \frac{1}{T_i}\right)$$

$$p_{ij} = p_{ji}$$

- If Ti < Tj and Ei > Ej -> Δ_{ij} is negative, and pij = 1
- What this means, is that if we have a good solution at high temperature, it will always exchange with a less good solution at lower temp
- The opposite is also possible, a good solution in a high temp can also go to a higher temp, with p < 1

! in parallel tempering the temperature schedule is distributed across the replicas of SA, and not over iterations !

- At low temperature we exploit the solution
- At high temp we explore other solutions and other regions of S
- Exchanging configuration is a way to combine diversification and intensification



GUIDING PARAMETERS

It obviously depends on how we define our parameters

- How many SA? M is often: M=sqrt(N) -> N is the problem size
- At what frequency should we consider exchange of configuration? When both systems reach an equilibrium state, according to the SA definition
- What is the range of temperature between T1 and TM? T1 should be small enough to allow the convergence, TM should be large enough to allow exploration (TM = T0 of standard exploration)
- We can fine tune the temperature of each system over time, if the exchange rate is too large > 2% we increase the temperature difference between all systems
- If the exchange rate is too low < 0.5% all temperature difference between the systems is decreased