Metaheuristics Sept 26, 2022

- Bastien Chopard
- Assistant: Francesca Zucchelli, Mira Arab.
- Exam: practical: Exercises (TP)
  - ↓ 1 pt based report to write
  - 60% presence mandatory at the exercise session.
  - oral
    - ↙ jan - Feb 2023
    - About theory
    - 20 min preparation
    - 20 min discussion
      - ↳ list of questions

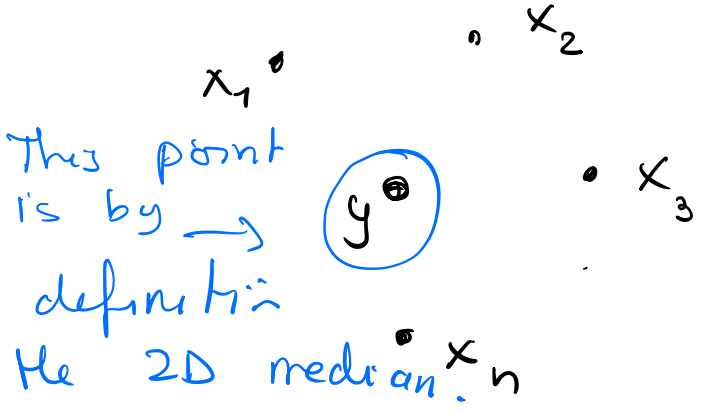
⚠ No document during the oral exam.

Document for the class:
- Notes on the screen

Chapter 1 Introduction to metaheuristics

1.0 Example:

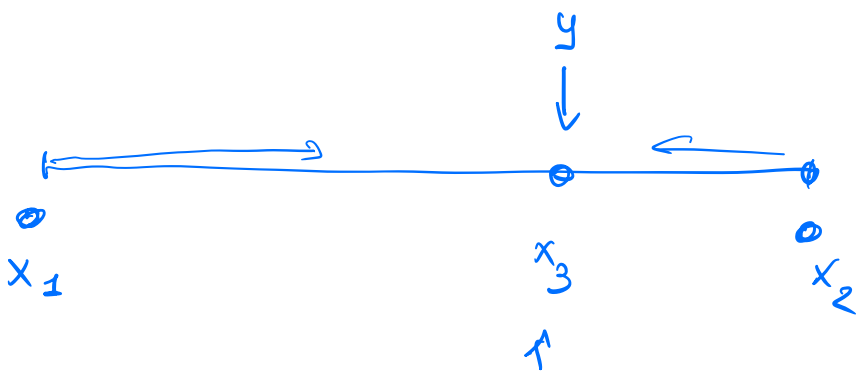
We have n persons in a 2D space



This point is by definition the 2D median.

We want to determin where they should meet to minimize the total travelled distance.

It is not the center of mass.



what is this point: The geometric median!

How to compute y ?

travelled distance

$$d = \sum_{i=1}^{n} \|x_i - y\|$$

~~center of mass~~ 2

Find y to minimize d

$$y^* = \underset{y \in \mathbb{R}^2}{\operatorname{argmin}} \sum_{i=1}^{n} \| x_i - y \|$$

No analytical solution but an iterative solution

$$y_{k+1} = \left(\sum_{i=1}^{n} \frac{x_i}{\| x_i - y_k \|} \right) \Big/ \sum_{i=1}^{n} \frac{1}{\| x_i - y_k \|}$$

A numerical solution is easy to find.

1.1 Search Space

We want to solve an optimization problem. It means to find

$x \in S$ such that a given functio[n] maximum

$f(x) \in \mathbb{R}$ is optimal / → minimum

S is the **search space** that is the set of all possible response (or solution)

The **optimal solution** is by definition:

$$x_{opt} \in S \Rightarrow x_{opt} = \underset{x \in S}{\text{argmin/argmax}}\ f(x)$$

The **optimal value** is by definition

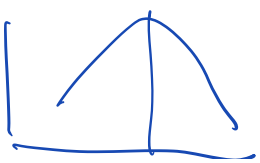
$$f_{opt} = f(x_{opt})$$

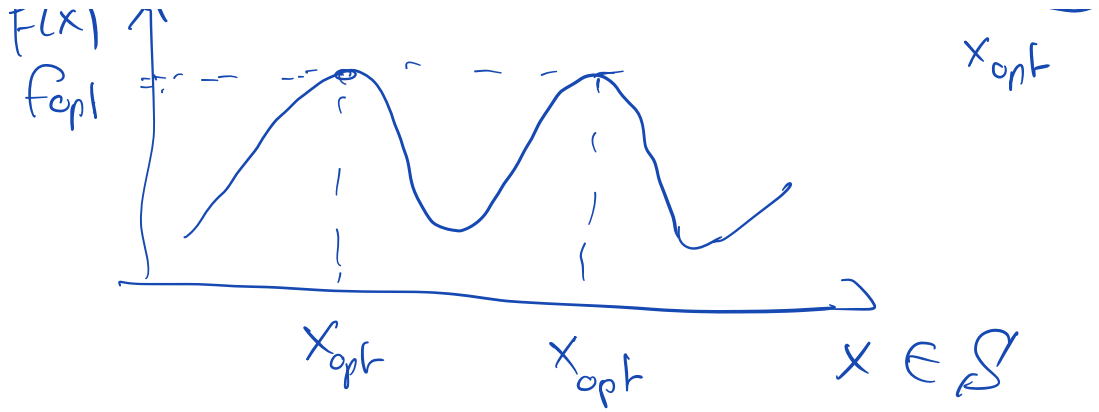
The function f is called:

- Here we mostly use the name **fitness**
- Objective function
- Cost function

} it is how the quality of the solution is quantified

The optimal solution is in general not unique:





But the optimal value is always unique. Otherwise take the best one.

It is common to have constraints on x in an optimization problem.

find y that minimize

$$d(y) = \sum_{i=1}^{n} \|x_i - y\|$$

where y should be only in some specified subspace

In what follows, we assume that these constraints are implemented in S by elimination non-acceptable

Solutions.

Another way to specify an optimization problem is:

x_{opt} is such that

$$f(x_{opt}) \geq f(y) \quad \forall y \in S$$

↑ maximization problem

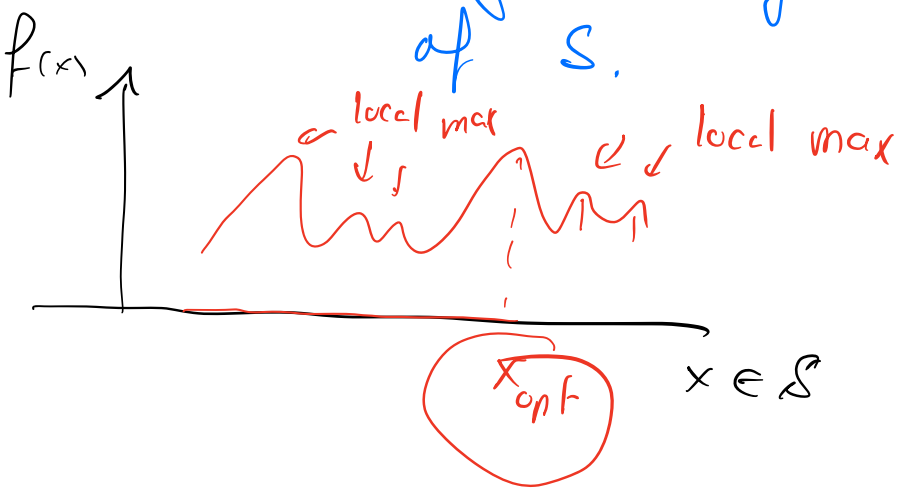
$$f(x_{opt}) \leq f(y) \quad \forall y \in S$$

↑ minimization problem

We usually have

- Global optimum: what we want to find
- local optimum: solutions that are optimal is

a given region of S.



In many numerical schemes these local optimal may attract the solution

Usually we consider multidimensional problems:

$$x \in S \quad x = (x_1\ x_2\ \dots\ x_n)$$

x is a vector with n components.

The value of n is called the **problem size**: the number of degrees of freedom (or variable)

But this is not the size of the search space $|S|$

$x_i \in S_i \qquad S = S_1 \times S_2 \times \cdots \times S_n$

$$|S| = |S_1| \times |S_2| \cdots$$

$$= |S_i|^n$$

Our main problem is that S is way too large (even when discrete) to explore it exhaustively

A <u>metaheuristics</u> is a way to explore S to find x_{opt} when no polynomial algorithm exist to find it.

1.2 Examples

For some of the examples, there is no need for restrictions, because efficient algo exist.

I. $S = \mathbb{R}^n$ $f(x)$: continuous differentiable.

$\forall i \quad \frac{\partial f}{\partial x_i} = 0$

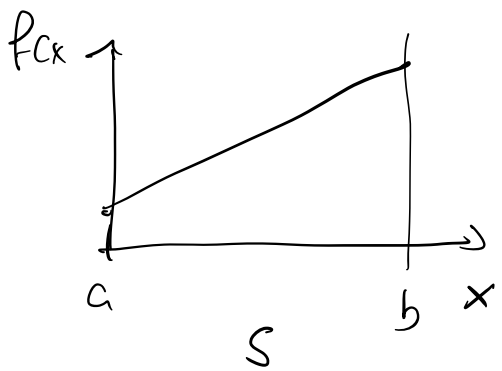
↑ grad $f = 0$

gives the local and global optimum

↑

still have to exchael it from all solutic of $\frac{\partial f}{\partial x_i} = 0$

There is another difficulty:



If the optimal solution is at the boundary of S

Max is $f(b)$

$x_{opt} = b$

$\frac{\partial f}{\partial x} \neq 0$ at $x = b$

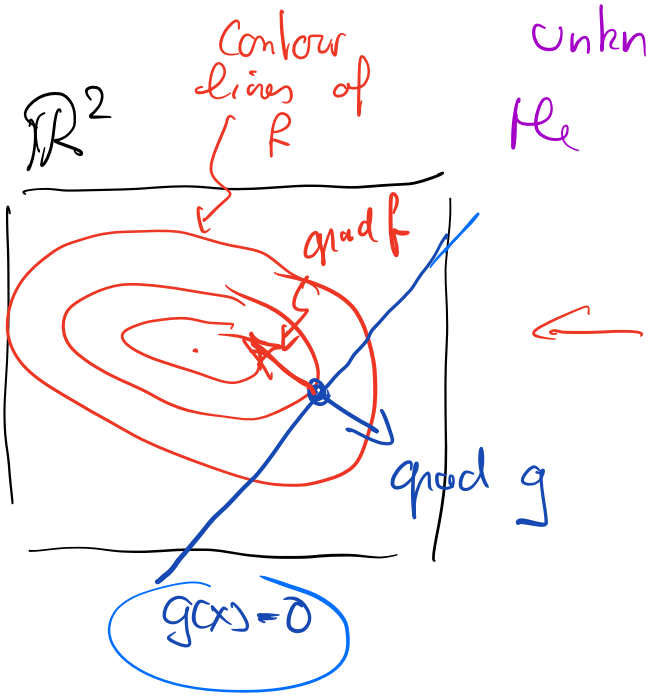
the derivative does not vanish.

what about constraints: find
x that maximize $f(x)$
such that $g(x)=0$

You may use the Lagrange multiplier:

Solve: $\boxed{\text{grad } f = \lambda \text{ grad } g}$ (A)

λ: unknown, called the Lagrange multiplier.



II Linear programming

$$\max \quad z = f(x) = \sum_{i=1}^{n} c_i x_i$$

(x_i: how much you produce of x_i; c_i: profit)

subject to constraints

$$\sum_j a_{ij} x_j \leq b_i \qquad i = 1, \dots m$$

The SIMPLEX algorithm solves the problem.

III Knapsack problem: we have n objects of value p_i and size w_i. You want to take with you objects so as to maximize the value, but your bag has a limited volume.

Maximize $f(x) = \sum_{i=1}^{n} p_i x_i$

$x_i = \begin{cases} 0 & \text{I don't take object } i \\ 1 & \text{I take it} \end{cases}$

with constraint

$$\sum w_i x_i \leq C$$ the knapsack volume.

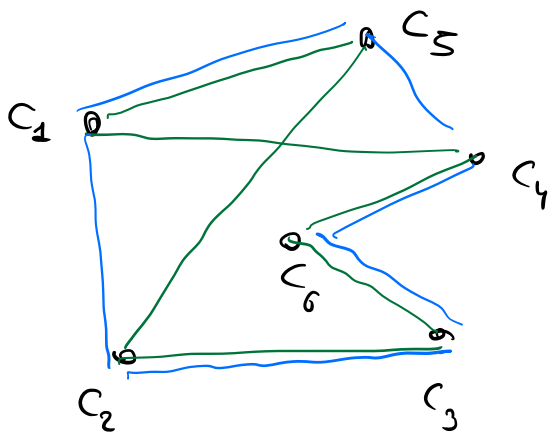
Search space?

$$S = \{0, 1\}^n$$

IV Traveling Salesman problem (TSP)

An example of what is known as QAP problem: Quadratic Assignment problem.

we have n cities that need to be visited once only once by a salesman. The tour should finish at the starting point and be of minimal distance:



- possible tour is it the shortest one?
- Another tour, shorter.

The problem is to find the order in which the cities must be visited to produce the shortest tour.

What is S ? $\quad - c_1 c_4 c_5 c_3 c_2 c_6$

$- c_1 c_2 c_3 c_6 c_4 c_5$

$-$ ⋮

S is the set of permutation of the 6 cities.

$$|S| = n! = n(n-1)(n-2)\cdots 1$$

V : Nk-problems

Those are often artificial (or synthetic) problems, whose difficulty can be adjusted. They provide benchmark for optimization methods.

- First instance: the MaxOne problem.

  Consider n Boolean variables

  $x_1 x_2 \cdots x_n$

  and find their value so as to maximize the number of 1

The solution is obvious for us

$$x = \underbrace{1\,1\,1 \ldots 1}_{n \text{ times}}$$

but for a machine algorithm, this is no so obvious.

$$S = \{0,1\}^n \qquad |S| = 2^n$$

↑ problem size.

General formulation of a NK-problem:

fitness to be maximized is defined as:

$$f(x) = \sum_{i=1}^{N} f_i(x_i, \underbrace{\text{neighbors}}_{k \text{ arguments}})$$

$x = (x_1 x_2 \ldots x_N)$ to be specified

MaxOne: $f(x) = x$ $\quad k = 1$

$$f(x) = \sum_{i=1}^{N} x_i$$

($k=0$ on Internet)

- $k=3$, the neighbors of x_i are x_{i-1} and x_{i+1}

$$x_1 \; x_2 \; \underbrace{x_{i-1} \; x_i \; x_{i+1}} \; \ldots \; x_N$$

$$f_i(x_{i-1}, x_i, x_{i+1})$$

$$f(x) = \sum_{i=1}^{N} f_i(x_{i-1}, x_i, x_{i+1})$$

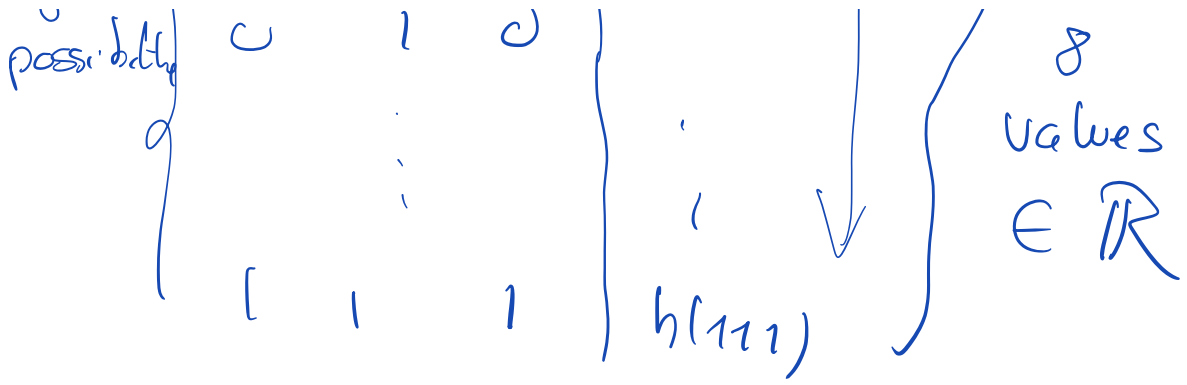
+ periodic boundary condition.

We can further consider the case where all the f_i are the same:

$$f(x) = \sum_{i=1}^{N} h(x_{i-1}, x_i, x_{i+1})$$

Then h can be defined through a table

x_{i-1}	x_i	x_{i+1}	h
0	0	0	$h(000) =$
0	0	1	$h(001) =$



If $h(000)$ is very large

$x = \overbrace{000}\,0\ldots$ 000 is optimal

If $h(101)$ is large but
$h(010)$ is small

$\overbrace{10}\,0\,1\,0\,1\,0\,1$

It is

1.3 Reminder on computational complexity

We need an efficient algo to

solve an optimization problem.

How do we quantify the efficiency?

- Computational complexity either in time, or in space (memory)

- We are interested in the asymptotic complexity

$T(n)$ time complexity for n large

↑ problem size.

$M(n)$: space complexity

Complexity is often expressed as big O notation.

$$T(n) = O(g(n))$$

It means that there exist n_0 and k such that:

if $n \geq n_0$ $T(n) \leq k\,g(n)$

Example:

- $T(n) = 0.5n^2 + 2n = O(n^2)$

- $T(n) = n + \log n = O(n)$, since $\log n < n$

For some problems, there are **polynomial** algorithms (sorting, matrix multiplication, etc)

$$T(n) = O(n^m)$$

where m is usually small

These problems are said to belong to the **P class**

For other problems, we do not know any polynomial algorithms, but only exponential ones:

$$T(n) = O(2^n)$$

These problems are computationally difficult as $T(n)$ grows very fast with n.

Complexity Classes

In complexity theory problems are classified as:

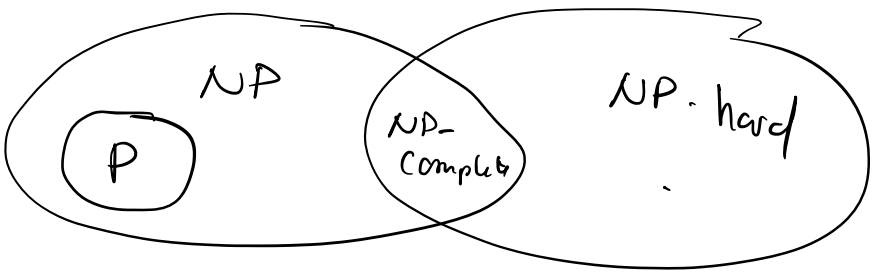
P, NP, NP-complete, NP-hard

- The P-class contains the problems that can be **solved** in polynomial time.
- The NP-class contains the problem whose solution can be **verified** in a polynomial time.
- P problems are obviously in NP

- NP-hard problems are those solutio solves any NP problems, up to a polynomial time.

- NP-complete problems are those that are both in NP and NP-hard

In practice NP-hard and NP-complete problem need an exponential time to be solved.



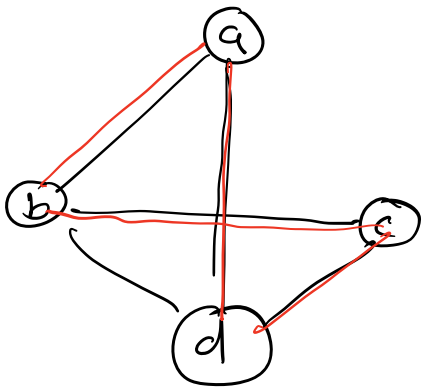
We will show that TSP is NP-hard but not NP-complete

Example Hamiltonian cycle

It is known to be NP-complete.

The problem is to find a close path

in a graph that goes through all nodes once and only once.

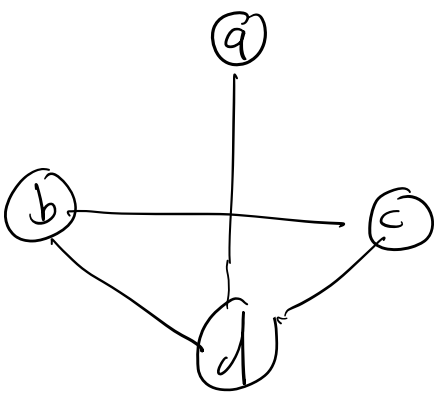


This graph has an hamiltonian cycle

- The length of this path is 4 = number of nodes.

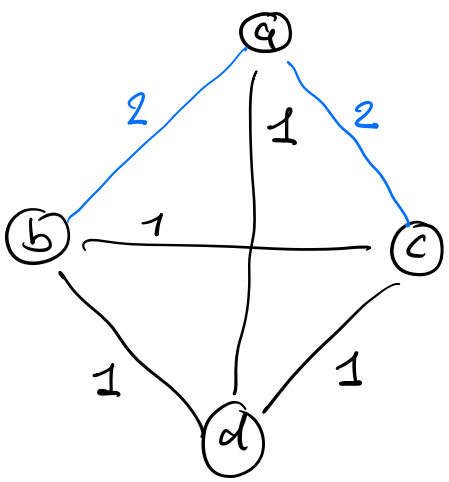
Let us now consider the following graph:

consider a TSP problem



Is there an Hamiltonian cycle ?

NP-complete problem.



We put a length 1 to all existing links.

Then we add missing links so as to

obtain a complete graph. Now we solve the TSP problem on this new graph.

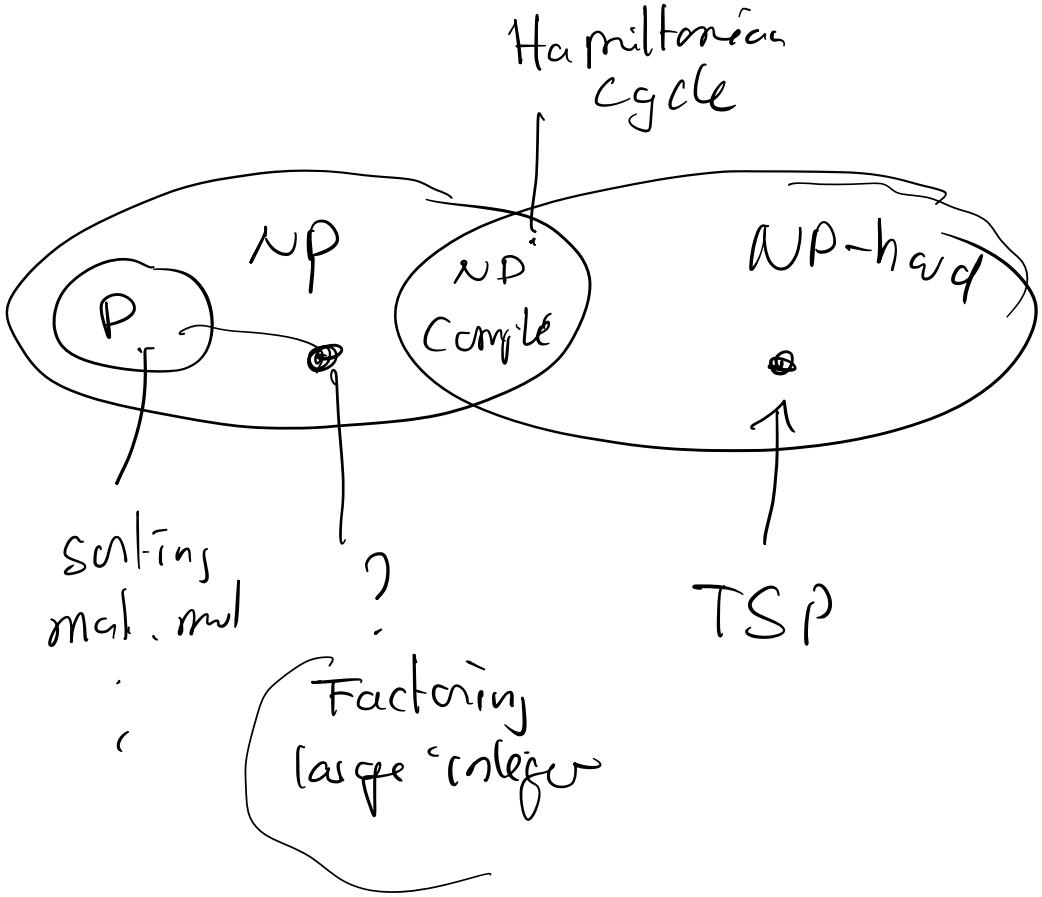
If the best tour is of length 4 ($= n$) it means that only original links were used $\Rightarrow$ there is an hamiltonian cycle.

If it is larger than 4, the new links were used and the original graph has no hamiltonian cycle.

Conclusion : TSP solves the hamiltonian cycle problem, which in turns solves any NP problem.

TSP is not in NP because we cannot verify in polynomial time whether the tour is minimal.

$\Rightarrow$ TSP is NP-hard



- Many optimization problems that need metaheuristics are NP-hard
- A deterministic algo would be exponentially long.

Metaheuristics Oct 3, 2022

1.4 Characterization of Metaheuristics

For many practical optimization problems, there are no polynomial algorithms, and the exponential algorithms are too costly because the search space S grows too fast with the problem size.

- A metaheuristics is a way to explore S in a clever way, in order to find optimal solution to an optimization problem.

- It is usually a compromise between CPU resources and the quality of the result.

- The term metaheuristics was proposed

in 1986 by Glover.

- A heuristics is a strategy to find a solution to a problem ; use some specificity of the problem to speedup its resolution.

- A metaheuristics is a heuristics which is applicable to a wide range of problems
  → but there is no guarantee of quality.

Metaheuristics are very general :

- No specific hypotheses on the fitness function f (like continuity, convexity ...)

We only ask that $f(x)$ can be computed for all $x \in S$

- Metaheuristics use guiding parameters that adjust the way the search is done. The optimal value of these guiding parameters is unknown and depends on the problem.
- Metaheuristics need an initial point (a possible solution) to start the exploration. Most of the time, this initial point is chosen at random in S. But if some properties of the optimal solution is know, we can use them to better choose the starting point
- We need a stopping condition. Since it is not an exact algorithm we never know if we reached the optimal solution. So the stopping criteria are:

- A given amount of CPU time.
- Stagnation of the quality of the solution: no more improvement for several iterations

- Metaheuristics are inspired by natural process (biology, physics, animal behavior, ...).
- They often are stochastic
- They use a lot of CPU, but are easy to implement and often can be parallelized

Metaheuristics are base on two main search principles:

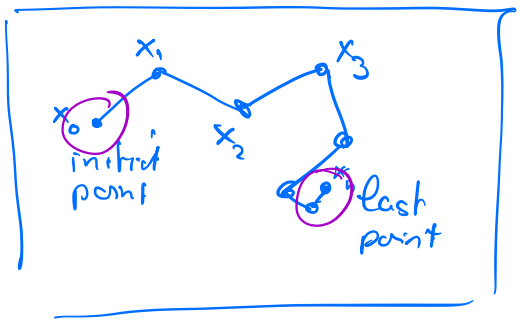
- intensification (= exploitation)
- diversification (= exploration)

It means that we combine an improvement of a promising solution by searching around it. Or we can try to explore new regions of S

1.5 How metaheuristics work

First we need to code the search space and its solution is a way which is recognized by a computer.

A metaheuristics is a trajectory in S

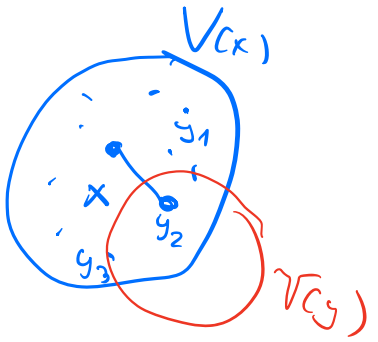


Some metaheuristics give as a final response the last point on the search trajectory, or the best point along the trajectory

How do we choose the next point from the current one ?

- We define a neighborhood $V(x)$ for all $x \in S$ the neighborhood of x contains all the points $y \in S$ that can be reached

from x



$x_0 \xrightarrow{u} x_1 \in V(x_0) \xrightarrow{u} \dots V(x_2) \ni x_3 \xleftarrow{u} x_2 \in V(x_1)$

u is the exploration or search operator that determines the next candidate in the neighborhood.

- u is specific to each metaheuristics
- The neighborhood is usually choosen by the scientist.
  The size of the neighborhood is a compromize:
  - if too small there is little chance to find a good succesor.
  - if too large, it takes too much time to find the succesor.

Neighborhood:

How can one specify the neighborhood?

- We could consider to list explicitly all the y in V(x): $V(x) = \{y_1, y_2 \dots y_k\}$

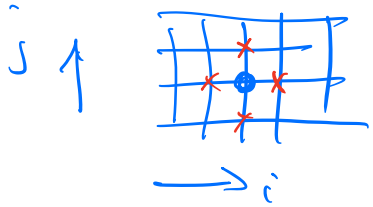
- But usually the neighborhood will be generated through k transformations (= movements)

We define functions $T_i(x) : S \to S$

$x \to T_i(x)$

$$V(x) = \{ y \in S \mid y = T_i(x), \text{ for } i = 1, \dots k \}$$

Example $S = \mathbb{Z}^2$ (= mesh)



$x \in S \Rightarrow x = (i, j)$

A typical neighborhood is North, South, East, West

$T_1(i,j) = N(i,j) = (i, j+1)$

$T_2(i,j) = E(i,j) = (i-1, j)$

$T_3(i,j) = S(i,j) = (i, j-1)$

$T_4(i,j) = W(i,j) = (i+1, j)$

Example movements for permutations

S = set of permutation of n objects

$n = 3$

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

$|S| = n!$ $\quad n=3 \quad n! = 6$

The neighborhood of a permutation will be specified by transformation or movement

- For instance, we can define the movement as the swap of two element.

  A transposition (i, j) consists in exchanging elements at position i j

- For $n = 5$

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

So the permutation $(a_3, a_2, a_4, a_5, a_1)$ has a neighbor $(a_2, a_3, a_4, a_5, a_1)$ with the movement $(1,2)$

How many neighbors do we generate like this ?

$$|V(x)| = 10 \quad \text{for} \quad n = 5$$

$$= O(n^2) \quad = \frac{n(n-1)}{2}$$

Note that the size of the neighborhood $O(n^2)$ is small with respect to the size of S, which is $O(n!)$

$$\frac{n^2}{n!} \to 0 \quad \text{if} \quad n \to \infty$$

Note also that any permutation is S can be reach from any other with a finite number of transposition.
This is important because any point in S can be explored with such a neighborhood.

- Some metaheuristics explore the entire neighborhood to select the next point. Others just generate at random one of the neighbors.

Some simple search methods

- Random search : take at random a point in S and repeat until some ending condition.
  $V(x) = S$ U : random pick.
- Random walk : take the successor at random in a neighborhood which is small : walker in $\mathbb{Z}^2$
  N, S, E, W
- Hill Climbing (gradient descent, steepest descent...)
  Take the best of your neighbors in term of the fitness value.
- Probabilistic hill climbing ; the successor y are selected with a probability p(y) that depends on their fitness

$$p(y) = \frac{f(y)}{\sum_{y' \in V(x)} f(y')} \qquad y \in V(x)$$

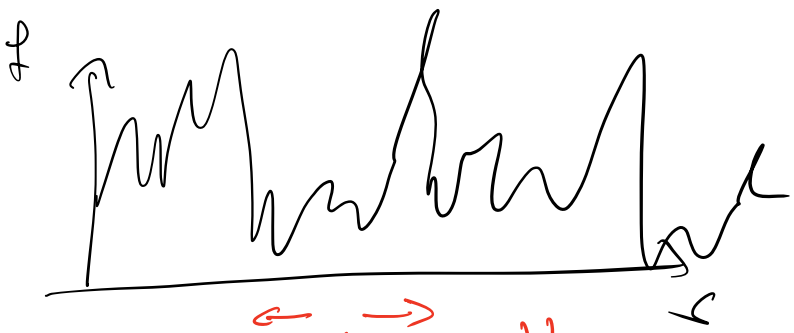
This approach is very useful to avoid to be trapped in local optimum.

Fitness landscape

This is the representation of the fitness

function as a function of x.

Smooth fitness landscape. It is easy to find the optimal solution.



hard fitness landscape.

In general, the fitness landscape is a characterization of the variation of f across a neighborhood.

to represent that in one-dimension, the neighborhood should be small $V(x) = \{x, x-1, x+1\}$

Example:

The role of the coding and the definition of the neighborhood has an impact on the effectiveness of the search.

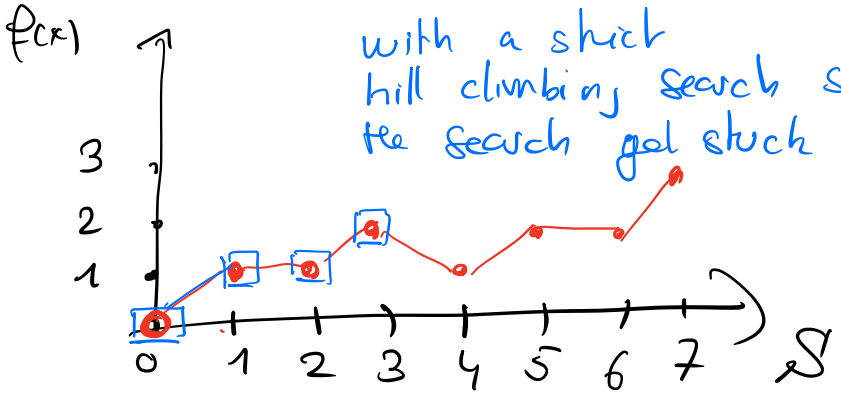
Let us take the **max One** problem, with

two different coding:

$N = 3 \qquad x = (x_1 x_2 x_3) \in \{0,1\}^3$

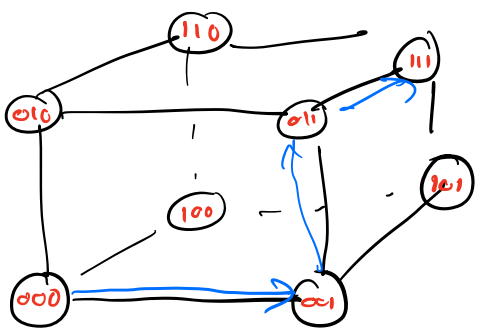
We can interpret x as a binary number between 0 and 7

In this case, we define $V(x) = \{x-1, x, x+1\}$



with a stuck hill climbing search starting from x=0 the search got stuck a x=3

2nd coding, as a hypercube:



Neighbors of x is all the points that differ by 1 bit only.

The same hill climbing search from 000 goes easily to the optimal solution 111

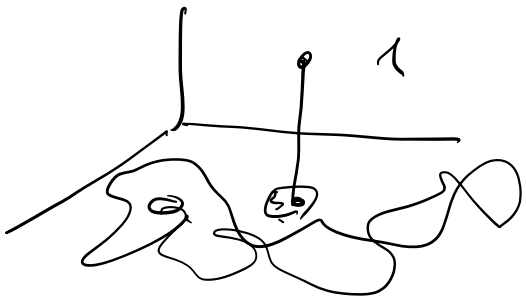
A very difficult problem for a metaheuristics:

$x \in \{0,1\}^N \qquad f(x) = x_1 \cdot x_2 \cdot \ldots \cdot x_n$

to be maximized.

The optimal solution is $x = (1\,1\,\ldots\,1)$

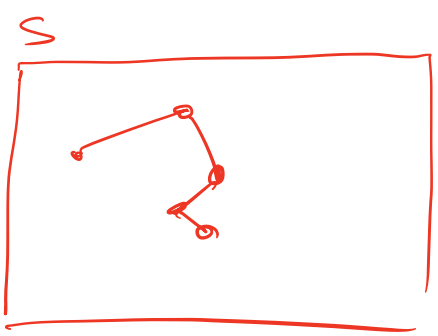
The fitness landscape is

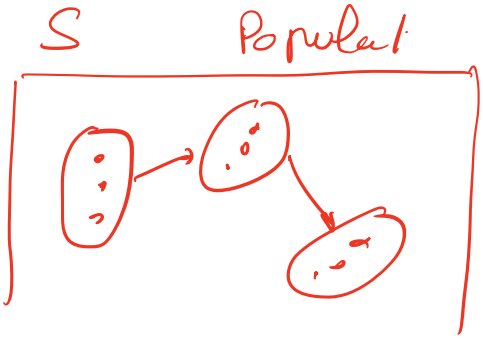


Population metaheuristics

So far we discussed individual metaheuristics, in the sense that only one possible solution is considered at each iteration.

But we can also have a population of solution that we consider at each iteration.





Eagle Genetic algorithms

Co evolving several solution at the same time.

Chapter 2 : the Tabu Search

Proposed by Glover in 1986 to solve Quadratic Assignment problems (QAP)

Search explore S as usually:

$$x_n \xrightarrow{U} x_{n+1} \in V(x_n)$$

The exploration operator U works as follows:

- U chooses the <u>best</u> element in $V(x_n) - \{x_n\}$. Best means the best fitness in $V(x_n) - \{x_n\}$ considering the <u>non-tabu</u> points

It means that some points in S are getting a tabu attribute so that the search operator cannot select them.

- If in the non-tabu neighbors two have the

Same best fitness, one of the two is chosen at random.

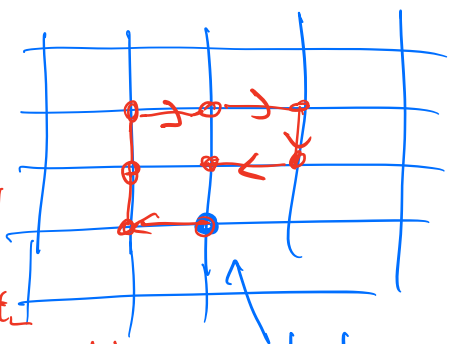
- Since the current point is excluded from the neighborhood, we may move to a point of less quality fitness.

[The goal of the tabu attribute is to prevent the search to go back to a solution that was already visited.]

The question is how a point becomes tabu. The tabu search is then based on a tabu list which contains, at a given stage, the banned solutions. Elements will be added and removed from this tabu list.

Example walker in $\mathbb{Z}^2$

Here, with the N,S,E,W neighborhood we can get stuck after a few iterations as all the neighbors



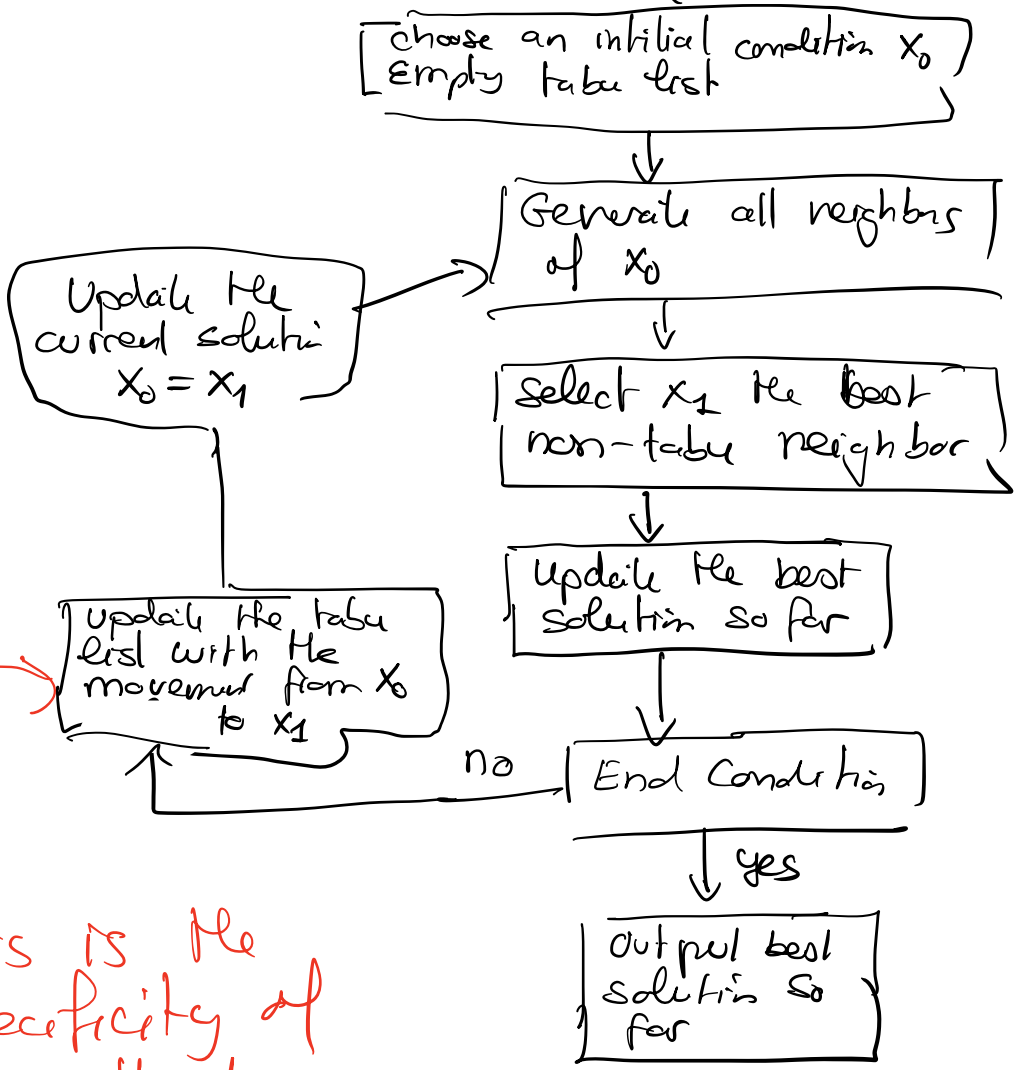
Let us assume that a point becomes tabu as soon as it has been visited

- tabu point.

are in the tabu list.

⇒ the tabu list should be dynamic and be updated regularly.

Flow chart of the method



This is the specificity of the method.

Convergence

This is the property of a metaheuristics to find the global

optimum.

For the tabu search we have the following result:

- If S is finite, and the neighborhood is symmetric

$$s \in V(t) \Rightarrow t \in V(s)$$

and any $s' \in S$ can be reached from any $s \in S$ in a finite number of steps.

Then a tabu seach that memorizes all the visited points as tabu but allows the search to restart from the oldest tabu point will visit the entire space S (and find the global optimum since it explored all the space).

But in practice this is not useful as this complete search can take

a time exponential in $|S|$

In practice we hope that the tabu search will never go to such a situation.

2.2 The tabu list

Goal to avoid visiting points we already know. Avoid cycles.

- What info should we put in the tabu list?
  - visited solution
  - attribute of the visited solutions.
  - But, the most common case is to put movements in the tabu list.

$$V(x) = \{ y \in S \mid y = m_i(x) \quad i = 1 \dots k \}$$

Now some of the m_i will be banned.

Typically we will ban inverse movements of that we just did $\therefore$ it prevents cycle.

Example $\mathbb{Z}^2$: mvt : N, E, W, S

If South is chosen, then North will be forbidden.

But this cannot be enforced for ever. Then there will be a duration of the tabu attribute.

After a few iteration, the tabu movement will be again available.

Chapter 2 Tabu Search

$$x_i \xrightarrow{u} x_{i+1} \in V(x_i) \setminus \{x_i\}$$

u chooses the best non tabu successor as required by the so called tabu list

The tabu list may contain:

- possible solutions
- attribute of possible solutions
- movement (or transformation)

The goal of the tabu list is to avoid to explore part of the search space that has already been explored.

The tabu list cannot be permanent: it has to be updated continously by adding or removing elements.

How to implement the update ?

- A FIFO memory of size M.
  The M most recent points are tabu then they are forgotten.
- banning / tenure time (temps d'interdiction)

A short term memory approach

- A solution or a movement is banned for the next k iterations, where k is a rather small number selected at random for each new entry in the tabu list.

This has to be complemented by a long term memory mechanism, whose goal is to avoid a bias in the exploration of S

If some movement has not been used for a long time, it will be imposed.

Example of a banning tabu list

Walker in $\mathbb{Z}^2$

tabu list

	Short term memory: banning time	long term memory: frequency
North		0.5
East		0.5
South	3	0
West	5	0

In addition we measure the frequency of each mvt. North and East have been used 1 out of two times.

$t = 1$: North is chosen → South will be tabu for k iterations

In the table, we write for South an banning time

$t+k$, for instance $k=2$, $t=1$, and one set the table to $k+1=3$

$\Rightarrow$ as long as the iteration t is smaller or equal to 3, South cannot be chosen.

Supposed that at $t=2$, East is chosen. West is taboo, and with $k=3$, we get $5=2+3$ for West

After a prescribed number of iterations, the long term memory is activated and the banning time can be disregarded.

Another example

The tabu list is based on an attribute $h(x)$ of solution $x \in S$. h is supposed to be an positive integer

The tabu list is an array of size M

Let x be the current solution at iteration t.

Then in $T(h(x) \bmod M)$ one set the banning time to $t+k$

Then, a solution x' at time t' will be tabu if

$t' < T(h(x') \bmod M)$

We will see that for QAP problems there is a specific way to build the tabu list, which combines the above ideas.

To implement a tabu list, we need several parameters:

- the memory size M
- the range of k values
- when the long term memory should be activated, and what is considered as a biased?

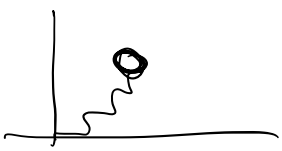
All these parameters are called guiding parameters and their optimal value is unknown. Their optimal value can be problem dependant. We usually rely on empirical knowledge to choose them.

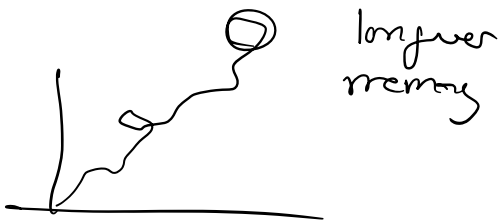
Exploration and exploitation
(diversity) (intensification)

→ if the tabu list is large then the search is driven by exploration (many

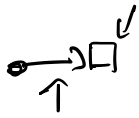
promising points are tabu, let us go away).

→ If the tabu list is short, we will be able to choose the best neighbor. And we can be stuck with a local optimum. (exploitation)





→ Aspiration : this is a mechanism that is added to the tabu search : the goal is to be able to accept a tabu movement if it leads to a better solution than ever seen so far.



tabu movement : the aspiration lets go go there.

10^{20}

2.3 Quadratic Assignment Problems (QAP)

It is an important class of optimization problems.

- We are given n objects and n possible location for these objects
  - We know the distance d_{rs} between locations r and s
  - We know the "flow" f_{ij} between object i and j

Goal: find an optimal placement of the n objects on the n location so as to minimize a cost of fitnen functi

$$\rightarrow \quad f = \sum_{i,j} f_{ij} d_{r_i r_j} = \sum_{r,s} f_{i_r i_s} d_{rs}$$

r_i: location of object i

i_r: object at location r

The minimization is done on the choice of the relation

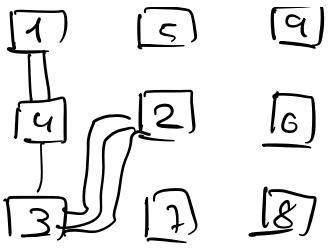
$$i \longrightarrow r_i$$

which is a permutation of n object.

The search space S is the set of permutat. of n objects.

Illustration:

$p = \underset{i \to r_i}{\operatorname{argmin}} f$

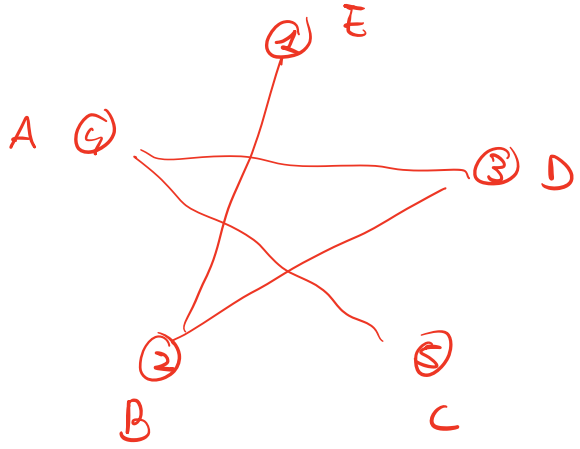


— flow = 1
= flow = 2
≡ flow = 3

1. bank
2: supermarket
3: post office
4: bus

Note that the distance d_{rs} are given by the respective position of the boxes.

TSP is a QAP problem.



objects are the order of the visit

flow:

$f_{i,i+1} = 1$

otherwise $f_{ij} = 0$

④ A B C D E ← location
(4 2 5 3 1) ← object

1 2 3 4 5 ← order (= object)
⑥ (E B D A C) ← location

Solution with a tabu search

We start with a permutation

$$p = (i_1 \; i_2 \; \dots \; i_n) \in \text{object}$$

(1, 2, ..., n ← location)

Initial $x_0 \in S$

We then explore S through moves.

For instance we can define the move $(r\ s)$ as the exchange of object at location r and s

$$(i_1 \; i_2 \dots i_r \dots \; i_s \dots i_n) \xrightarrow{(r,s)} (i_1 \dots i_s \dots i_r \dots i_n)$$

swap

There are $O(n^2)$ such moves.

From these $O(n^2)$ neighbors one chooses the best non-tabu one.

Note that computing the fitness

of the new candidate takes $O(n)$ operation as most of the terms in the double sum $\sum_{rs}$ have not changed.

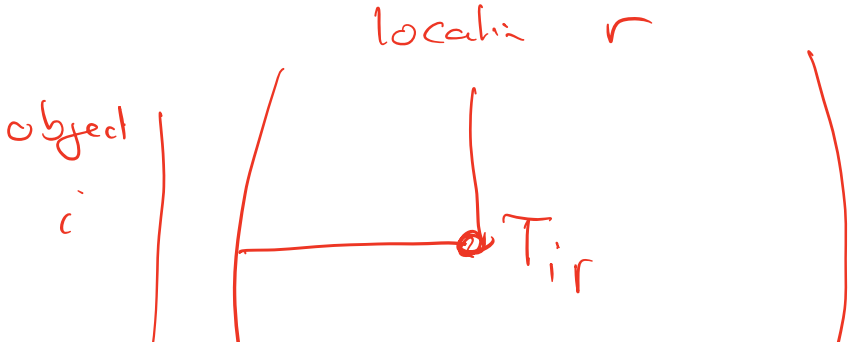
Tabu list

let us consider the movement (r,s)

$$(\dots i_r \dots i_s \dots) \xrightarrow{(r,s)} (\dots i_s \dots i_r \dots)$$

It is forbidden to place again the object i_r at location r and i_s at location s during the next k iteration.

The tabu list is a $n \times n$ matrix



T_{ir} is the iteration t at which i left location r, augmented by the banning time R.

Thus the movement (r, s) will be tabu if T_{is} and T_{ir} both contains a value larger than the current iteration

The banning duration R is chosen at random in the following interval:

$$R \in [0.9 \times n,\ 1.1 \times n + 4]$$

and R is an integer

Chapter 3 Simulated Annealing

(recuit simulé)

3.1 Motivations

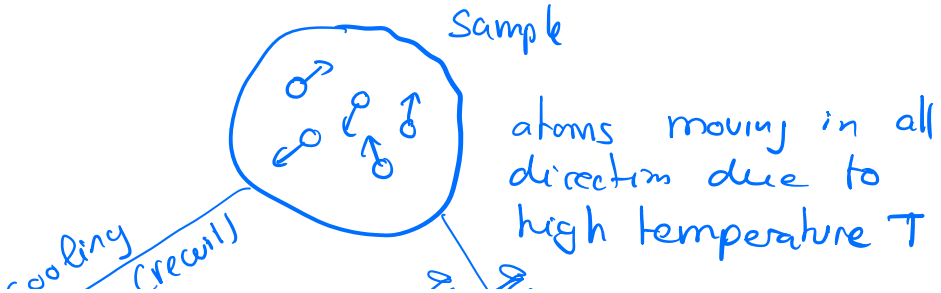
It is a metaheuristics inspired by physics (Energy minimization) and metallurgy:

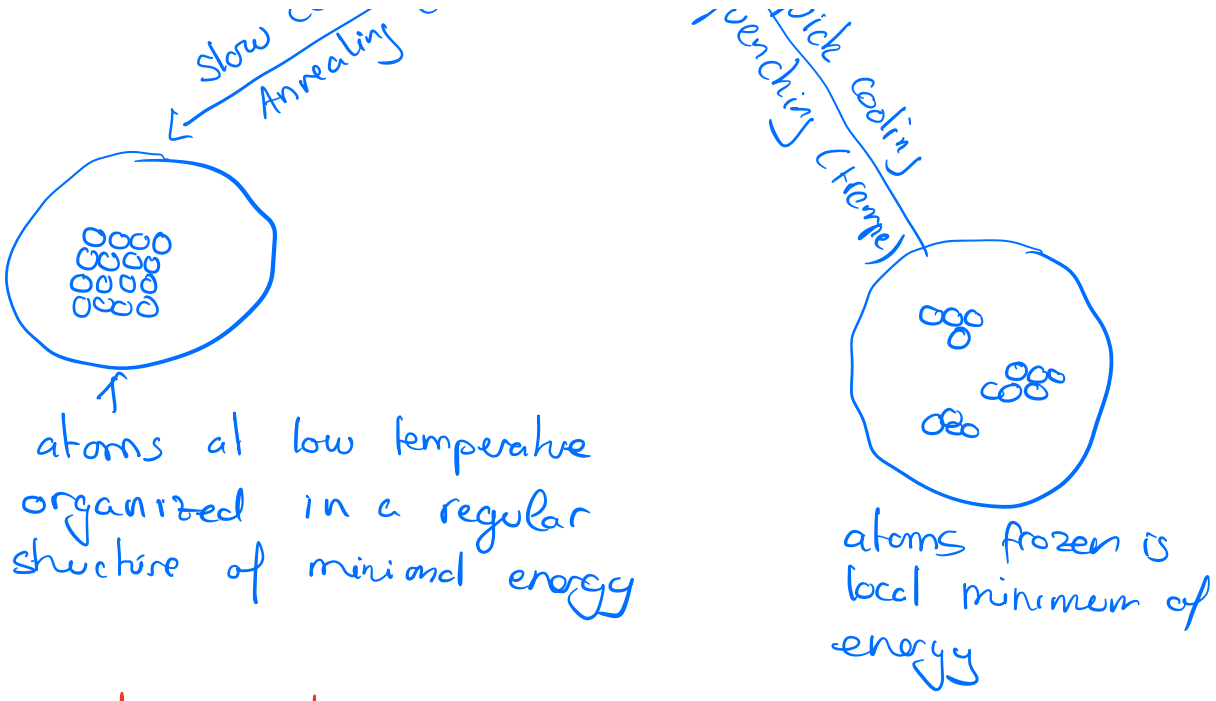
- Annealing (recuit): a process by which a metal is cool down progressively.

  Annealing usually leads to a local minimum of energy.

It contrasts with quenching (trempe) where the cooling is very fast.

Illustration





Interpretation:

Nature tends to minimize the energy of a system.

At high temperature, the system explore many possible states. If a low energy state is found the system will stay longer in this state; if the temperature is reduced, the system will be trapped in such low energy states.

3.2 Principles of the algorithms

We want to find the global minimum of a fitness function E (E for energy)

One specifies the search space according to the problem, one defines the movement (or transformation)

One also chooses an initial point in S, often called "configuration"

At each iteration, a neighbor of the current solution is generated at random.

It become the next solution with probability p

Metropolis rule → $$p = \min\left(1, e^{-\Delta E / T}\right)$$

where $\Delta E = E_{new} - E_{old}$ = variation of fitness

and T stands for temperature (which

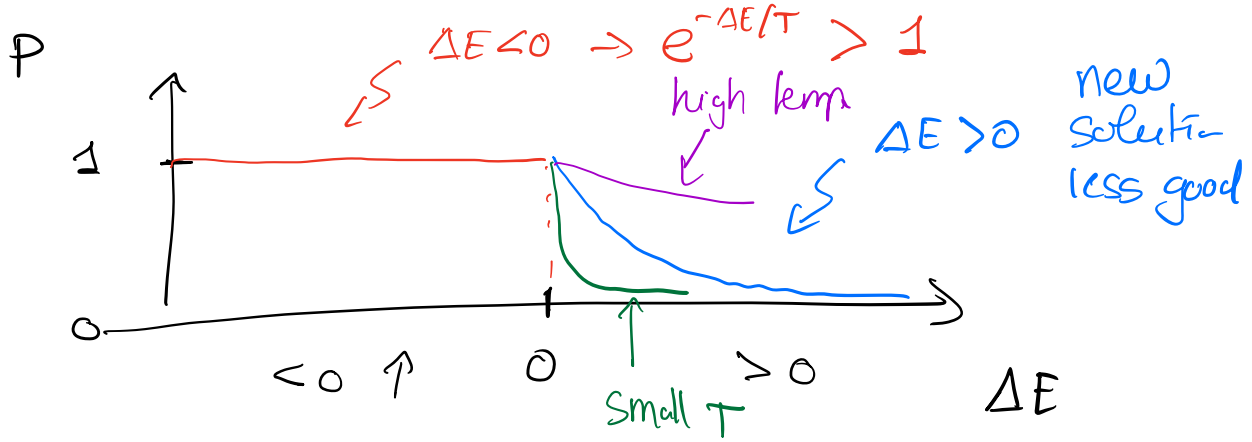
is here just a numerical value.

In practice the new candidate is accepted if:

$$\text{Random}(0,1) < e^{-\Delta E/T}$$

Otherwise, the candidate is rejected and one needs to generate a new neighbor.

Illustration of the Metropolis rule



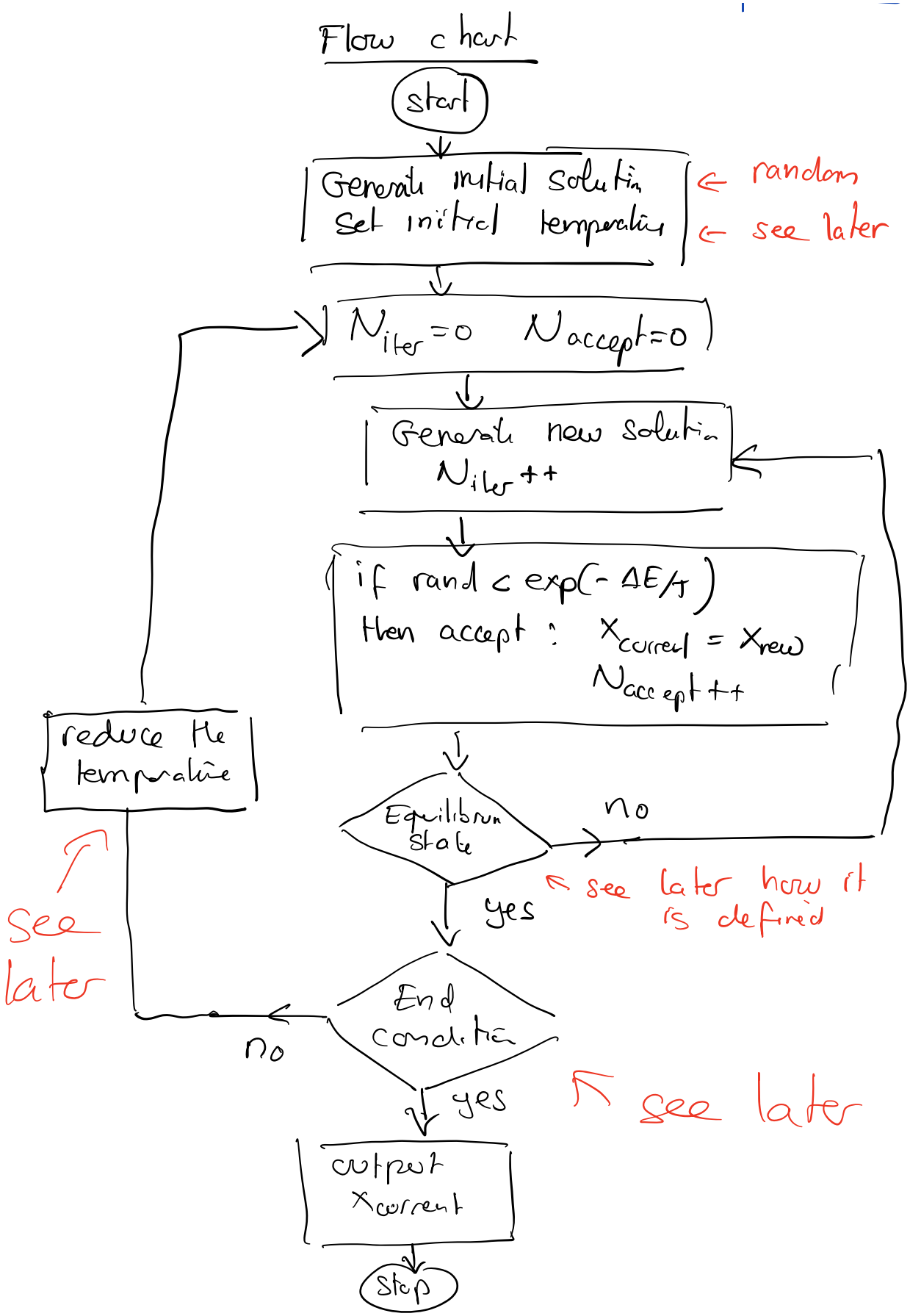
$$p = \min\left(1, e^{-\Delta E/T}\right)$$

$p = 1$ if $\Delta E < 0$ which means that the new candidate has a lower energy

We always accept a new solution which is better than the previous one.
If $\Delta E > 0$, the candidate is of lesser quality than the current solution. Still, we can accept it, but with a probability that decreases as ΔE gets larger, and as T gets smaller.

The temperature T will decrease as the search progresses.

- We start with high T
  → most movements are accepted ⇒ exploration
- Then temperature is decreased according to a temperature program. So bad solutions are less and less likely to be accepted ⇒ exploitation


Flow chart
start
Generate initial solution
Set initial temperature
← random
← see later
$N_{iter}=0$ $N_{accept}=0$
Generate new solution
$N_{iter}++$
if rand < exp(−ΔE/T)
then accept : $X_{current} = X_{new}$
$N_{accept}++$
reduce the temperature
Equilibrium State
no
yes
← see later how it is defined
see later
End condition
no
yes
see later
output $X_{current}$
Stop

Metaheuristics, Oct 17, 2022

Chapter 3: Simulated Annealing (Recuit simulé)

Explore S by generating one neighbor at random and accepting it with the Metropolis rule:

p = prob of acceptance

$= \min(1, e^{-\Delta E/T})$

ΔE is the variation of Energy (= fitness that we minize)

and T is a parameter that evolves during the search: T decreases

3.3 Convergence

Does SA (Simulated Annealing) find the optimal solution? The global one, not a local one!

SA can be analyzed mathematically.

There is a convergence in probability

[With a probability arbitrary close to one, we get a solution arbitrary close to the global optimum.]

But the condition for this theorem to hold are very demanding and impossible to impose in practice.

Conditions for convergence

- The transformation (or movement) must reversible $s \in V(r) \Rightarrow r \in V(s)$
- Any $s \in S'$ can be reached by a finite number of transformation.
- The initial temperature must be high enough
- The temperature should decrease slowly as iteration goes on:

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

(t: iteration)

and not $T_{next} = 0.9\, T_{current}$

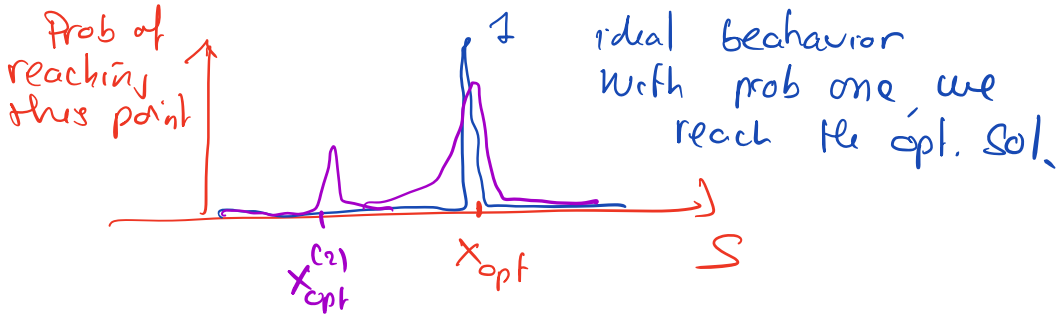
C is an unknown constant, reflecting the possible variation of the fitness.

This temperature program is very slow and of no practical use. Thus, in practice SA is not guaranteed to reach the optimal solution.

Illustration

We want to illustrate on a simple example the constraint of this convergence theorem.

- What does it mean to have a convergence in probability ?

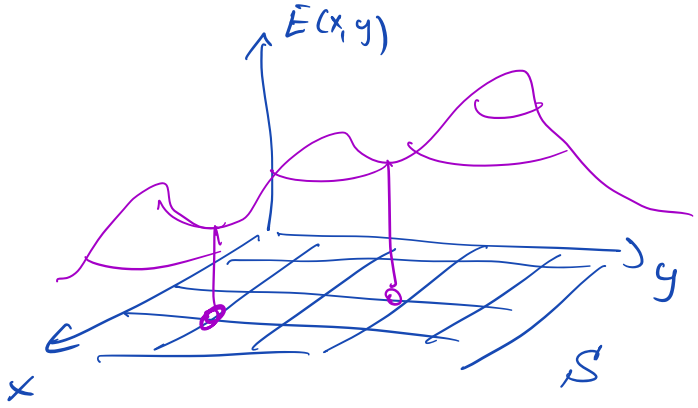


- We want to understand the question of the initial temperature that must be big.

- We want to understand the role of the temperature program : what happens if T

decreases too fast ?

Example



$S = \{1, 2, \ldots 10\} \times \{1, 2, \ldots 10\}$

x y

$V(x, y) = \{(x+1, y), (x-1, y), (x, y+1), (x, y-1)\}$

Global min of E is at $(x, y) = (9, 2)$

Second min of E is at $(x, y) = (6, 7)$

- We want to build a probability map for each $(x, y) \in S$, the prob that SA select this point as a optimal solution.

Let us denote (i) the points in S

$i = n_x(y-1) + x$ visit S in a 1.n

↑ 10

way.

$$x = \mathrm{mod}(i-1, n_x) + 1 \qquad y = \mathrm{int}\left(\frac{i-1}{n_x}\right) + 1$$

Let $P(t, i)$ the probability that at iteration t the SA is at configuration i

What can we say about $P(t+1, j)$

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

↑ prob to choose j as a neighbor of i at iteration t

$W_{ij}(t)$ is a matrix of size $\underset{(n_x \times n_y)^2}{100 \times 100}$

The value of W_{ij} is given by the Metropolis rule.

$$W_{ij}(t) = \begin{cases} 0 & \text{if } j \text{ is not a neighbor of } i \\ \frac{1}{4} P_{\text{metropolis}}(E_i, E_j, T(t)) & \text{if } j \text{ is neighbor of } i \end{cases}$$

↑ because 4 neighbors

We also have to define the prob to stay at configuration i, i.e j is rejected

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

Now we want to compute $P(t, i)$ knowing that at time $t=0$, we where in l, with prob $P(0, l)$, which is uniform if the initial point is chosen at random.

$$P(t,k) = \sum_j P(t-1, j) W_{jk}(t-1)$$

$$= \sum_j \sum_i P(t-2, i) W_{ij}(t-2) W_{jk}(t-1)$$

$$= \sum_i P(t-2, i) \left[W(t-2) W(t-1) \right]_{ik}$$

Following this way, we have

$$P(t,k) = \sum_l P(0, l) \left[\prod_{t'=0}^{t-1} W(t') \right]_{lk}$$

$$= \sum_e P(0,e) \underbrace{[W(0,t-1)]_{ek}}$$

↑ by definition.

the behavior of this matrix depends on T_{init}.

Numerically we observe that if $T(t=0)$ is large enough, $W(0,t-1)$ has all its lines identical.

$$[W(0,t-1)]_{ek} = [W(0,t-1)]_{1k}$$

↑ or any index

Thus:

$$\boxed{\mathcal{P}(t,k) = \sum_e P(0,e)\, W(0,t-1)_{ek}}$$

$$= W(0,t-1)_{1k} \underbrace{\sum_e P(0,e)}_{1}$$

so the search does not care where it started

$$= W(0,t-1)_{1k}$$

Otherwise, $[W(0, t-1)]_{ek}$ depends on e if $T(t=0)$ is low. Then the result depends on the initial configuration e

For $T(t=0)$ large enough, we have

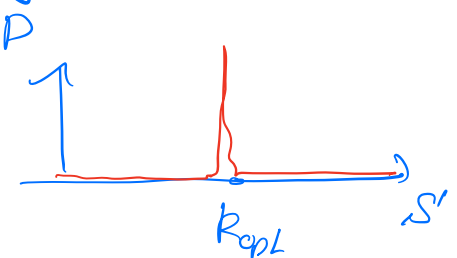
$$P(t, k) = W(0, t-1)_{1k}$$

It depends only in the final state k.

So we can represent $P(t, k)$ as $P(t, x, y)$ on a 2D map.

Then, on this map we can visualize the impact of the temperature program: T should decrease slowly.

$W(0, t-1)_{1k}$ $\xrightarrow{t \text{ large}}$



Let see what we get for our numerical example.

3.4 Pratical guide to SA

- Represent the search space as a data structure (coding)
- Choose the transformation (movements)
- Check that the computation of ΔE is fast.
- Possible constraints to the solution should be expressed by restricting the transformation so as not to generate impossible solutions. Or one can add a extra cost to the fitness.
- Choose an initial condition at random.
- To choose the initial temperature $T(t=0)$ proceed as follows:
  - Generate 100 movements starting from the initial condition, choosing them at random uniformly

- Compute $\langle \Delta E \rangle$ which is the average value of the energy variation on this 100-step trajectory. ⇒ gives a hint of how the fitness landscape is around x_0

- Choose an initial acceptation rate (probability) τ_0 of movements that make the fitness less good.

  Typically one choose

  - $\tau_0 = 0.5$ if the quality of x_0 is supposed to be poor.

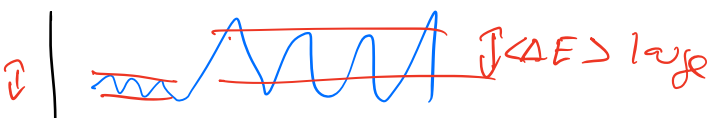
  - $\tau_0 = 0.2$ if x_0 is supposed to be good.

Then $T_0 = T(t=0)$ is computed such as to satisfy:

$$e^{-\langle \Delta E \rangle / T_0} = \tau_0$$

$$\Rightarrow \quad T_0 = \dots$$

Illustration:



<4ES small.

- Temperature program

Temperature is lowered when an equilibrium state is obtained

If not, space is still explored with the same level of temperature

An equilibrium state is reached when

→ $N_{accept} \geq 12\,N$ or if $N_{iter} \geq 100\,N$

where N is the problem size = nb of variable in the solution space.

At an equilibrium state, T is decreased as

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

k is the equilibrium index.

- Stopping condition:

SA stops if during 3 consecutive temperature levels, no solution improvement are observed

- "Validation": Repeat SA with different

initial conditions, different sequence of random number and make sure that the quality of the solutions is comparable.

Metaheuristics, Oct 24. 2022

3.5 Parallel tempering

It is a way to parallelize SA

The origin is from physics simulation of protein folding (Monte. Carlo)

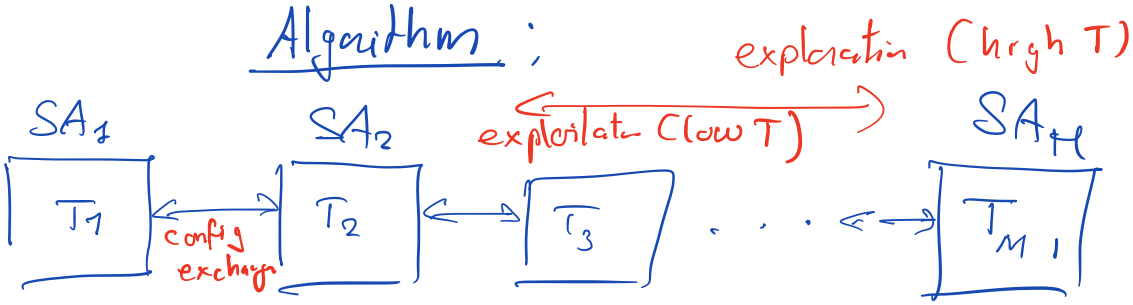
Two simulations at temperatures close to each other can be exchanged, as the probability distribution overlap.

This process prevents to be blocked on local energy minimum.

So parallel tempering is also expected to improve and speedup convergence

Tempering: process in which temperature is increased, then decreased several time.

Algorithm :



We have M replica (copies) of a SA process, each with its own temperature T_i which mostly stay constant during the entire process.

The condition is that

$$T_1 < T_2 < T_3 \ldots < T_M$$

Moreover adjacent SA can exchange their configuration at appropriate time, if a condition discussed below is satisfied.

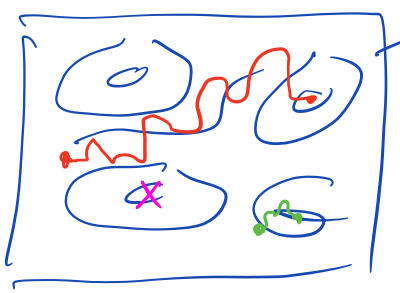
Configuration exchange means that the current solutions of each SA are swapped. Iterations then continues but each with a new temperature

The goal of this is to <u>exploite</u> a configuration which is good at high temperature by continuing the iteration at a lower temperature. On the other hand, a bad solution is sent to a SA of higher temperature for further <u>exploration</u>

So, in Parallel Tempering, temperature is no longer varying is fine, it is varying across the replica.

So, the temperature T_i are fixed during the process, except if there is no more exchanges, or too many of them. Then temperatures are rescaled.

Illustration

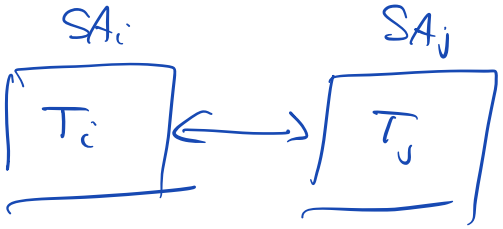


contour line of the Energy function

2D search space

We see from the example that it is indeed good to swap the high and low temperature solution to help convergence.

Condition to exchange configuration



$j = i \pm 1$

Two configurations of energy E_i and E_j

are exchanged with probability p

$$p = \min\left(1, e^{-\Delta_{ij}}\right)$$

with

$$\Delta_{ij} = (E_i - E_j)\left(\frac{1}{T_j} - \frac{1}{T_i}\right)$$

One can verify that if $E_j < E_i$ and $T_i < T_j$ then $p = 1$. Otherwise p is smaller than 1 and the more the degradation the less the probability

Guiding parameters

- How many SA should we consider ?
  It is suggested to have $M \sim \sqrt{N}$ where N is the problem size

- When should an exchange be considered ?
  For instance at the so-called equilibrium state, when no more progress is observed at the given temperature.

- What is the range of temperature ? $T_1 = ?$ low enough to converge
  $T_M = ?$ T_0

  This range should be rescaled is
  there are too many exchanges, or not enough.

  If this exchange rate is below 0.5%, then
  all temperature are decreased by $\Delta T = 0.1 (T_1 - T_M)$

If the exchange rate is to high (>2%), all temperatures are increased by ΔT.

Chapter 4 Ant colony and swarm intelligence

4.1 Motivations

- This metaheuristics is inspired by the field of entomology (science of insects) and ethology (science of animal behavior)
- Ants are able to solve optimization problems, in particular to find the shortest path between their nest and a food source.

Swarm intelligence (intelligence d'essaim)

As a group, individuals can do more than what they could do alone.

The individual power is low, but together they can produce new behaviors, not reachable to one single individual

This is observed for many species of animals:
- bees, wasps, termites have a complex social organization, with high level output.
But also birds, fishes, etc

- There is usually no centralized control
In biology, we call that heterarchy instead of hierarchy
- Local actions, with local interaction can lead to a global and coherent output.
- Emergent behaviors, or collective behaviors are observed in complex systems (a large set of simple entities that interact)
- These collective behavior lead to what is called "auto-organization" (= no central control is needed)

- Such an organization is robust: even if some entities fail, the rest continue to work.
- These organization can also be adaptative as some entities, by making mistakes, can produce new and better situations.
- Such structures are good for parallelization.

Thus, in 1992, three researchers have proposed a new metaheuristic to solve QAP problem using an agent based model.

One of the key ingredient of these model is the communication between agents. Here we will copy what ants do: deposit pheromones on promising regions of the search

... promising region of the search space.

Since then Ant Colony metaheuristics has been proposed, many others have been developed: Bee colony, PSO, firefly, ...

4.2 Pheromone track: a natural optimization process

Here we consider reals ants. What makes them succesful as a group?

It is well know that ants drop a chemical, called pheromone, on their path to attract other ants. So they mark a promising region of space.

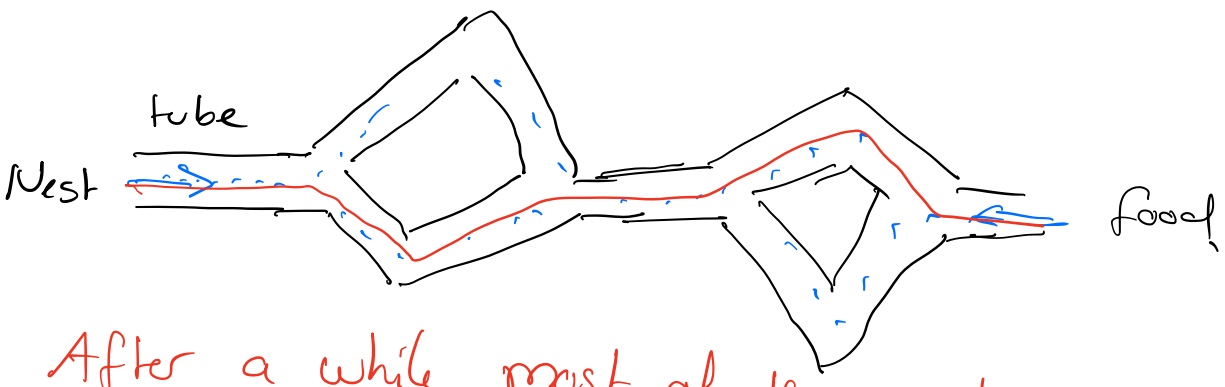
This mode of communication is called Chemotaxy: ants smell the pheromone and move towards the place where there are more

abundant.

The action of pheromone is not permanent. If no more ants keep adding new pheromones, the smell will evaporate and the region is space will be forgotten.
Thus, suboptimal region will no longer be maintained.

An experiment by Gross 1989

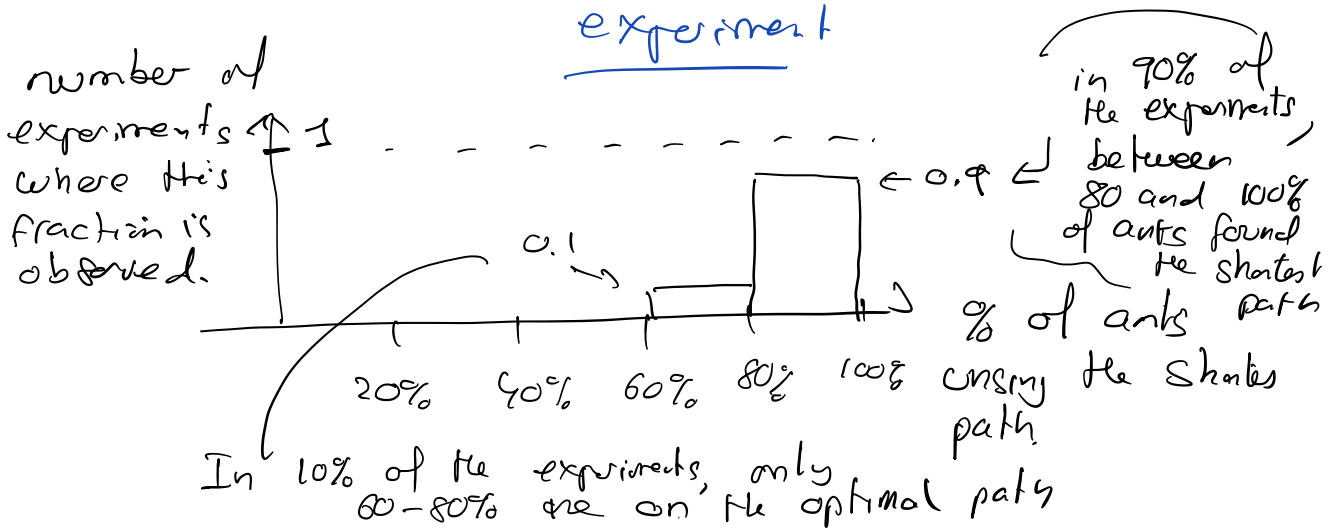
This scientist analyzed the behavior of ant to determine the effect of the pheromone to find the shortest path



After a while most of the ants the follow the red trail which is the shortest. There is an equal

... exploration period, followed by an exploitation phase.

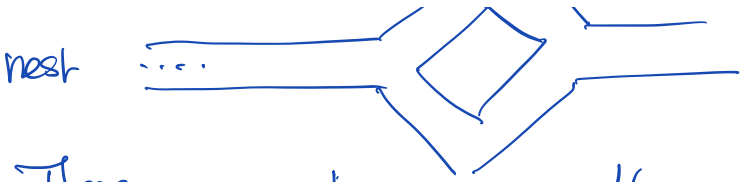
Result of the experiment



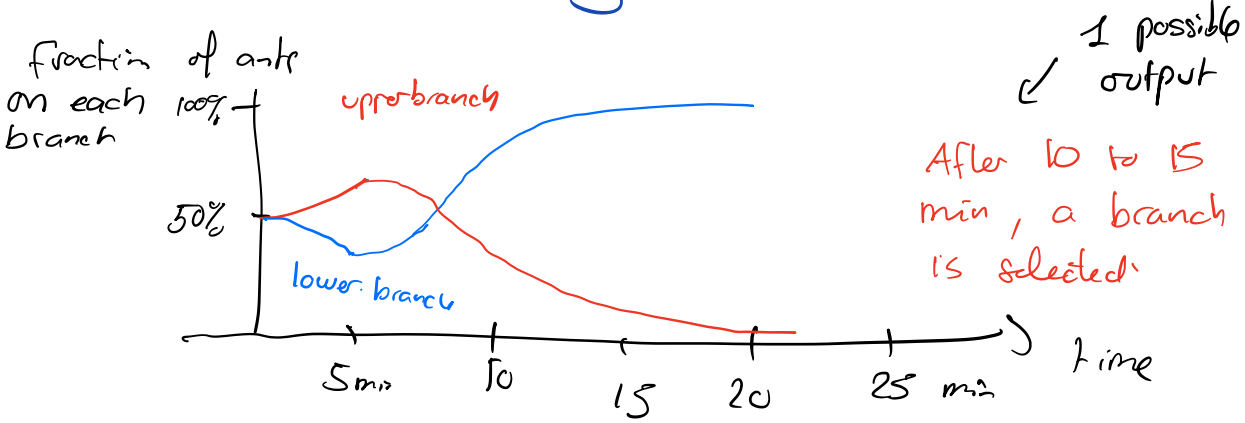
Interpretation: by chance, one ant took the shortest path. It is the first to reach food and it returns by the same way, amplifying the pheromone trail on this path.

The following ants, reaching the food through a longer path will return by the most marked path, that is the shortest.

We can better quantify ants behavior with a second experiment:



There are two possible path of exact same length. In the long run they select one of the two path. But it can be the upper of lower path, both are statistically equiprobable.



Gross and Deneubourg propose a probabilistic mathematical model to explain the observed behavior:

Let m be the number of ants that already went into the system.

Let U_m and L_m the number that took the upper and Lower branches.

$$m = U_m + L_m$$

What will be the behavior of ant $m+1$?

It will choose the upper branch with probability

$P_u(m+1)$

and $P_L(m+1)$ for the other branch.

with $1 = P_u(m+1) + P_L(m+1)$

One assumes that

$$P_u(m+1) = \frac{(U_m + k)^h}{(U_m+k)^h + (L_m + k)^h}$$

where k and h are calibrated from the experiment. It was found that

$$k = 20 \quad \text{and} \quad h = 2$$

fit well the data.

- We can then assume that 20 ants are needed to make the trail selective.
- The fact that $h = 2$ means that the attractiveness of pheromones grows quadratically with its amount.

4.3 Algorithm for optimization

in 1992, Dorigo et al proposed a metaheuristic called "Ant System" (AS)
Later on it was improved to give

the ACS (Ant colony system).

We will first define this method for a TSP problem.

There are n towns, with distance d_{ij} between city i and city j
We want to find the shortest tour that goes once and only once through these n towns.

For the AS algorithm one also define:

- The visibility $\eta_{ij} = \frac{1}{d_{ij}}$
- The intensity of the pheromone trail, τ_{ij} between two cities This is the attractivity of this route.

- The number m of ants that are exploring the possible tours
- And some other guiding parameters, typically an evaporation rate for the pheromone.

Principle

Each of the m ants decide on its tour based on τ_{ij} from the previous iteration, and η_{ij}
Once all ants have chosen their tour, the pheromone trail is updated.

Let us consider the case of an ant currently at town i that need to choose the next town j
It will choose town j with prob p_{ij}

$$\begin{cases} 0 & \text{if } j \notin J \end{cases}$$

$$p_{ij} = \begin{cases} \dfrac{[\tau_{ij}(t-1)]^{\alpha}\, \eta_{ij}^{\beta}}{\sum_{c \in J} [\tau_{ic}(t-1)]^{\alpha}\, \eta_{ic}^{\beta}} & \text{if } j \in J \end{cases}$$

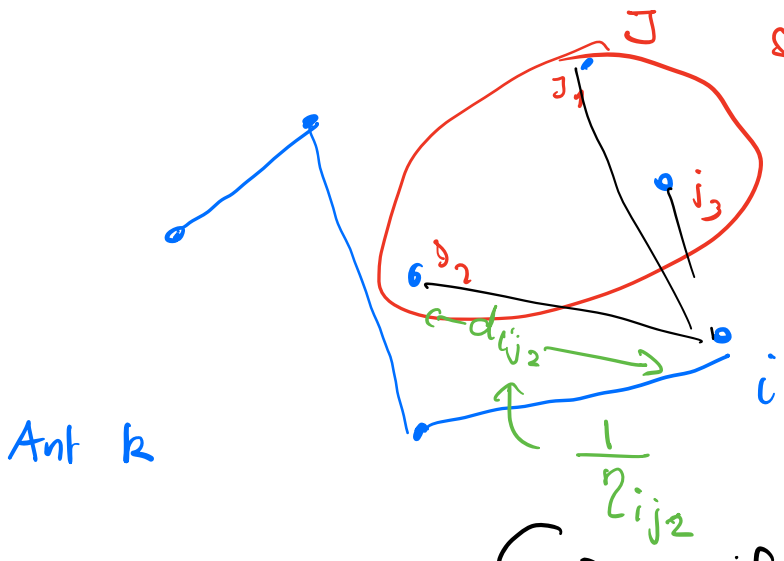
where J is the set of city still to be visited, and α, β are guiding parameters.

Metaheuristics, Oct 31, 2022

Ant systems

Pheromone trail

Algorithm for TSP



Ant k will decide of its next step with a probability

$$p_{ij_2} = \frac{[\tau_{ij_2}(t-1)]^{\alpha}\, \eta_{ij_2}^{\beta}}{\sum_{c \in J} [\tau_{ic}(t-1)]^{\alpha}\, \eta_{ic}^{\beta}}$$

if $j_2 \in J$

$$p_{ij} = \begin{cases} 0 & \text{if } j \notin J \\ \dfrac{[\tau_{ij}(t-1)]^{\alpha}\,[\eta_{ij}]^{\beta}}{\sum_{l \in J} [\tau_{il}(t-1)]^{\alpha}\,\eta_{il}^{\beta}} & \end{cases}$$

- η_{ij} is the visibility or proximity of town j from i
- τ_{ij} is the amount of pheromone on the (i,j) link.
- α and β are guiding parameter.
  If α is large it gives more weight to the pheromone trail $\Rightarrow$ intensification
  If β is large, it favors geographical info, $\Rightarrow$ diversification.
- A tour of the ant k is denoted $T_k(t)$ where t is the iteration
  All the m ants are performing their tour, independently of each

other.

- It is only when the m tours are completed that the pheromone trail is updated

$$\tau_{ij}(t+1) = \underbrace{(1-\rho)\tau_{ij}(t)}_{\text{evaporation}} + \Delta\tau_{ij}(t)$$

- $\rho \in [0, 1]$, a new guiding parameter
- $\Delta\tau_{ij}$ is the amount of new pheromone added to the link $(i\ j)$ due to the m tours of the m ants

$$\Delta\tau_{ij} = \sum_{k=1}^{m} \Delta\tau_{ij}^{(k)}$$

↖ ant index

$$\Delta\tau_{ij}^{(k)} = \begin{cases} 0 & \text{if } (i,j) \notin T_k \\ \dfrac{Q}{L_k} & \text{otherwise.} \end{cases}$$

- Q is a guiding parameter and

- L_k is the length of the tour of and k. So links belonging to a shat tours become more attractive.

Note that a non-zero value for τ_{ij} is needed at the first iteration, otherwise p_{ij} would be undefined.

$\tau_{ij}(t=0)$ is set as a constant value, equal to $1/nL$ where n is the number of towns and L an estimate of the length of a tour.

Empirically, it is found that the following values give good results:

$m = n \quad \alpha = 1 \quad \beta = 5 \quad \rho = 0.5 \quad Q = 1$

Remarks

- It is a population metaheuristics because at each iteration there are m possible solutions
- The "neighbor" of the current solution is produced by the interplay of the pheromone trail and visibility of the city. It can not be explicitly formulated.
- Each iteration can be parallelized as the tour of the m ants are independent.

From AS to ACS

The above algorithm was called "Ant System". An improvement was termed "Ant Colony System".

AS turned out to perform better on some benchmarks than other TSP metaheuristics. But on some other benchmarks, AS was less good.
Thus a variant was proposed : ACS

There is a new parameter, $q_0 \in [0, 1]$ which decide if the ant takes the best next link, or choose all possible ones:

An ant in town i goes to town j according to:

$$j = \begin{cases} \underset{l \in J}{\operatorname{argmax}} \left[\tau_{il}(t-1) \eta_{il}^{\beta} \right] & \text{with prob } q_0 \\ u & \text{with prob } 1 - q_0 \end{cases}$$

where u is determined as before (with $\alpha = 1$)

$$P_{iu} = \frac{\tau_{iu}(t-1) \eta_{iu}^{\beta}}{\sum_{l \in J} \tau_{il}(t-1) \eta_{il}^{\beta}}$$

q_0 close to one is now to favor intensifications.

Update of the pheromone trail

This is also modified compared to AS: It is a two-step update

Each ant adds an amount $\phi\tau_0$ on the link it visited

intermediate update $\tau_{ij} = (1-\phi)\tau_{ij}(t) + \phi\tau_0$ ← what is the value of τ_0? $\simeq 1/L_{av}$

evaporation. addition

ϕ: a new parameter $\phi = 1/2$??

Then an amount $\Delta\tau = 1/L_{min}$ is added to the link of the best tour found (L_{min} is the length of this best tour)

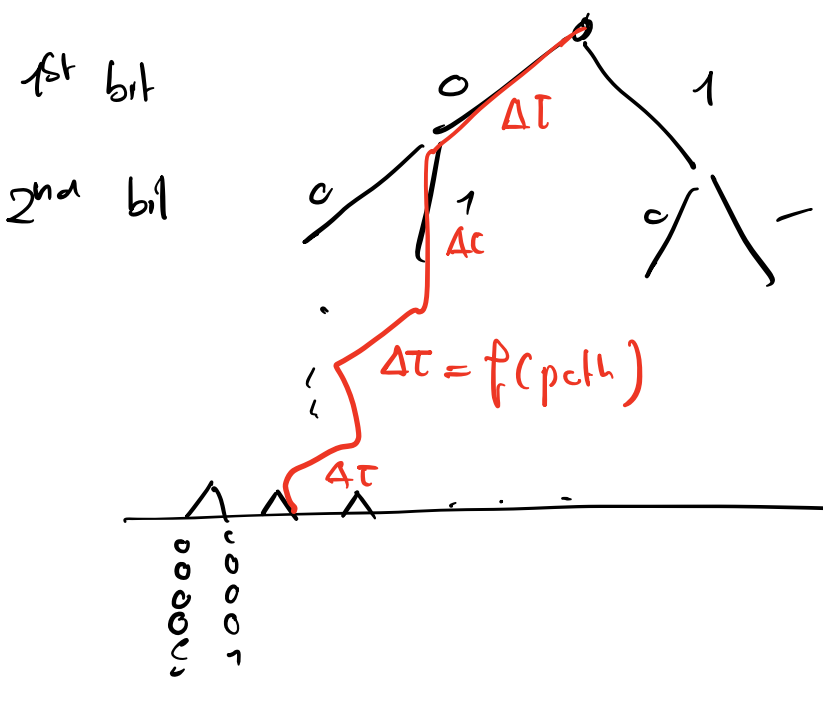
Final update: $\tau_{ij}(t+1) = (1-\rho)\tau_{ij} + \rho\Delta\tau$

4.4 Performance of Ants methods

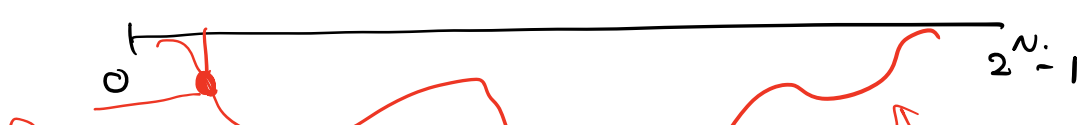
We will consider a simple problem $S = \{0, 1\}^N$, and we will adapt the Ant algorithm to this case

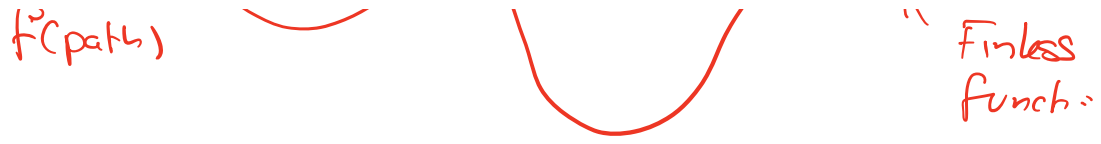
We have to find N bits that will maximize some fitness function.

We will represent the search space as a tree, so that ant can travel along the branches.



Ants will move towards the leaves and reinforce the branches according to the value of the fitness they have reached





How do the ants choose their way in the tree?

At each branch, the ant chooses with prob q_0 the best of the two branches (Best in terms of the amount of pheromone). With prob $1-q_0$ one of the two branches is selected.

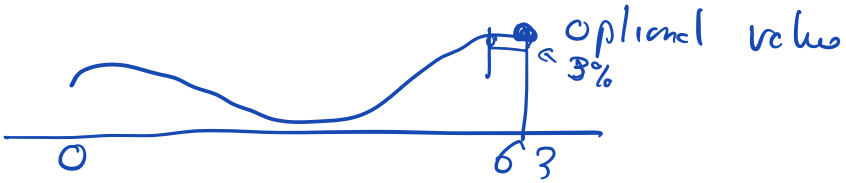
$$P_{left} = \frac{\tau_{left}}{(\tau_{left} + \tau_{right})} \qquad P_{right} = 1 - P_{left}$$

We consider the case of $m = 6$ ants, after 6 iterations

On such a run, there is or not an ant that found the path to the optimal solution.

To estimate the succes rate of this algorithm, we will repeat it 10 times

Computational effort	$6 \times 6 = 36$	= number of solution generated in the search space
success rate	7/10	in 7 out of 10 runs the optimal solution was found.
success rate at 3%	9/10	in 9 out of 10 runs an ant found a solution whose fitness is just 3% lower than the global optimal value.



Is this a good result ?

To give an idea of the quality of the result, let us consider a random search in which we generate 36 solutions in S. What is the probability to get the optimal solution ?

The probability to pick at random

The best solution is

$$p = 1/64$$

Out of 36 random pick, the prob to get at least once the best solution is

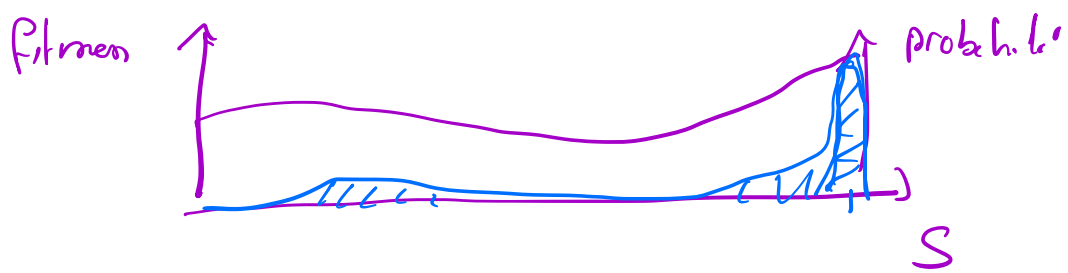
$$1 - (1-p)^{36} = 0.43$$

- $(1-p)$: prob of never getting the best
- $1 - (1-p)^{36}$: prob to get it at least once

to be compared with $7/10 = 70\%$ for ants

Ants algorithms do better than random search.

We can, for this simple example, compute the probability of finding each solution is S



So we observe that many ants find

the best solution, but not all. Several are ending up at other locations of the search space, yet more concentrated where the fitness is locally good.

Chapter 5 Particles Swarm Optimization (PSO)

(Optimisation par essaim de particules)

5.1 Introduction

Proposed in 1995 by Eberhart and Kennedy

Good for continuous optimization problems

$(S \subseteq \mathbb{R}^n)$

as opposed to combinatorial optimizat problems, such as TSP or QAP

- A swarm of particles explore the search space in the hope to find

the global optimum.

Principle

We have a set of individuals travelling in S. On the one hand, individuals try to reach the location in S where the best solution has been found by some individual in the group.

On the other hand, each individual or particle has a location in S which is the best it visited so far.

The movement of the particles in S will be defined by these points in S.

5.2 Algorithm

- Let $X_i(t) \in S$ be the position in S of particle i at iteration t

  X_i is a possible solution, of fitness $f(X_i)$

- The particles move is S according to their velocity $V_i(t)$

- We define as $X_i^{best}(t)$ the point in S that particle i visited up to t, which has the best fitness.

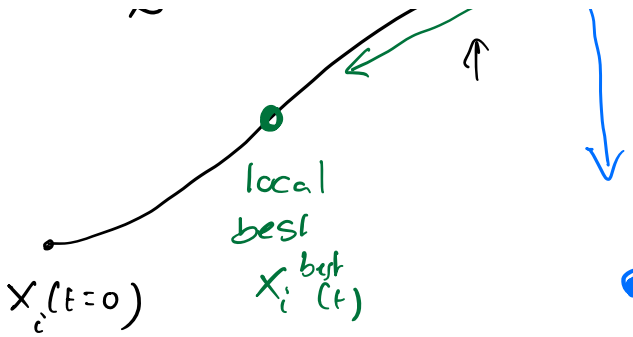
$$\rightarrow \quad X_i^{best}(t) = \begin{cases} X_i(t) & \text{if } f(X_i(t)) \text{ is better than } f(X_i^{best}(t-1)) \\ X_i^{best}(t-1) & \text{otherwise.} \end{cases}$$

- Global best: point in S which is the best found so far by the group:

$$B(t) = \underset{X_i^{best}}{\operatorname{argmax}} f(X_i^{best})$$

Movement

It is affected by the local and global best:



La nouvelle vitesse de la particule sera définie par ces 3 directions.

The new velocity will be determine by these 3 directions : $V_i(t)$, $B(t)$ and $X_i^{best}(t)$

$$\begin{cases} V_i(t+1) = \underbrace{\omega V_i(t)}_{\text{mult}} + c_1 r_1(t+1)\left[X_i^{best}(t) - X_i(t)\right] + c_2 r_2(t+1)\left[B(t) - X_i(t)\right] \\ X_i(t+1) = X_i(t) + V_i(t+1) \end{cases}$$

Note that these equations are vector equations

$V_i \in \mathbb{R}^n$, $X_i \in \mathbb{R}^n$ n = nb of dimensions of S

These equations are defined for S a subset of a vector space so we can add and multiply solutions by a scalar.

For formulation we do not know what

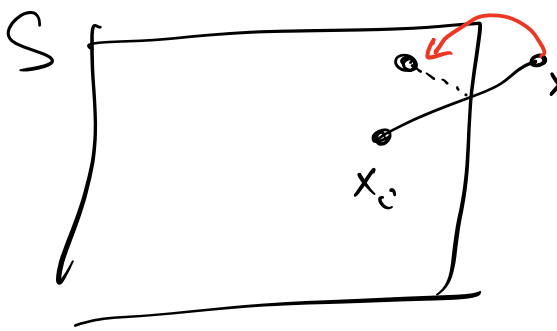
it means.

We still need to specify a few things:

- w, c_1, c_2 are parameters
- r_1 and r_2 are random numbers in a vector space: $r_1 = (r_{11}, r_{12}, r_{13} \dots r_{1n})$
  → n random numbers
  Here, we we write $r_1 \cdot (x_i^{best} - x_i)$ (· = element by element multiplication)
  we mean that each component of $x_i^{best} - x_i$ is multiplied by a random number.
  $r_{ij} \in [0, 1]$ uniform.
- Usually $c_1 = c_2 = 2$
  c_1 is called the cognitive coefficient (depends on the individual only)
  c_2 is called the social coefficient
- $w \leq 1$ is an Inertial coefficient
- The above equations may take the particle arbitrary far in $\mathbb{R}^n$. Usually S is bounded, and we want to

keep the particle in S

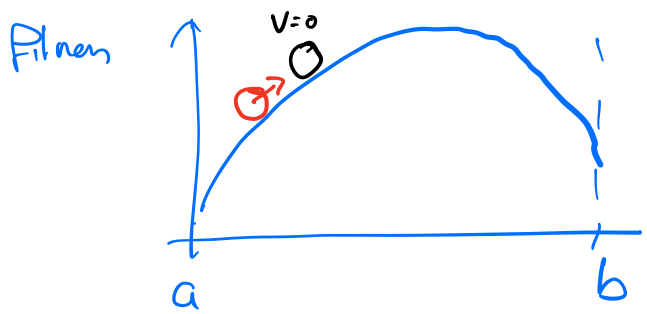
In practice, one enforces a max velocity, and one forces the particle to stay in S



any trick is fine

5.3 Examples

First we consider $S=[a,b]$ an interval in $\mathbb{R}$ with two particles and a simple fitness landscape:



Initially, both particles are at rest.
$V_i(0)=0$

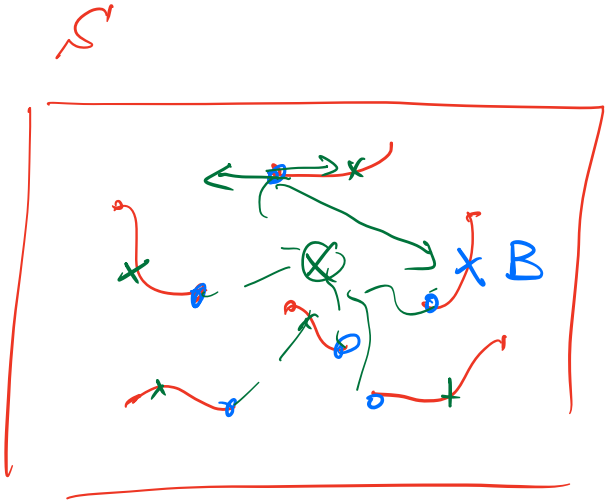
Other examples shows that PSO converge rather well, with not too many iterations and too many particles.

It is considered as a good and flexible

population method to solve continuous optimization problems.

Metaheuristics, Nov 7, 2022

Chapter 5 : PSO Particle Swarm optimization



$S \subseteq \mathbb{R}^d$

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

$$X_i(t+1) = X_i(t) + V_i(t+1)$$

$V_i(t=0) = 0$, $|V_i(t)| \leq V_{max}$

$X_i(t) \in S$ need to deal with boundary.

5.4 Firefly algorithm

Firefly = luciole

The fireflies are insects that emit light to attract mates or preys.

The stronger the light, the stronger the attraction.

- 2010, Xin-She Yang proposed a metaheuristic inspired by PSO and by the behavior of fireflies.
- It is also designed for continuous optimization.

Principle.

We have n fireflies, each of them at a location $x_i(t) \in S$, $i = 1, \dots n$ and t is the iteration. $S \subseteq \mathbb{R}^d$

Each firefly emits some light $I_i(t)$ al

each iteration, as a function of its location in S'.

For a maximization problem, we can assume that the intensity of the light emission is related to the local fitness.

$$I_i(t) = f(x_i(t))$$

where f is the fitness.

At each iteration, all pairs of fireflies are visited:

> if $I_i(t) < I_j(t)$
> then firefly i will move towards firefly j

The amplitude of the movement is related to the attractivity

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

is the fraction of the distance that firefly i will move toward j, where r_{ij} is the distance between them.

If the two fireflies are far away the attractivity is low.

- γ is a parameter. It has two roles:
  1) It fixes the units in which r_{ij} is measured.
  2) It has an impact on exploration versus exploitation: Do you care about a better solution, if it is very far away?

The movement of the fireflies is then made of an attraction part and a random part.

movement of i to j

$$X_i^{new} = X_i + e^{-\left(\frac{r_{ij}}{\delta}\right)^2}(X_j - X_i) + \alpha\left(rand_d() - \frac{1}{2}\right)$$

X_j: firefly j who is better than i

where α is a parameter and $rand_d()$ is a vector of d random numbers, uniformly distributed in $[0, 1[$

pseudo-code

iteration = 0

Initialize the firefly population: $X_i(0) \in S$ $\quad i = 1, \dots n$

while iteration < max-iteration

for $i = 1$ to n

for $j = 1$ to n

```
            if  I_i < I_j  then
                firefly i moves toward j
                update distances r_ij and
                intensity I_i
            endif
        end for
    end for,
    iteration ++
end while
output location of best firefly.
```

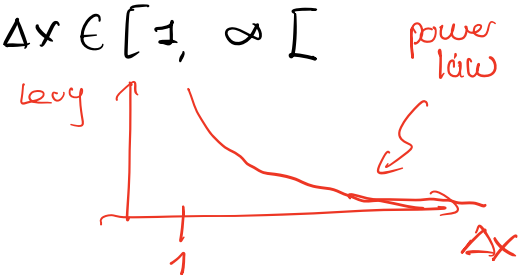
5.5 Random movements

To move randomly in $\mathbb{R}^d$, one has to generate the displacement according a given probability distribution.

For instance, we can generate a ΔX = jump with:

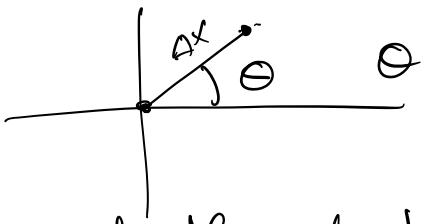
- A Gaussian distribution: $\Delta X \sim \mathcal{N}(0, \sigma)$
- A uniform distribution $\Delta X \sim \text{Unif}(-\sigma, \sigma)$
- Levy flight: $\int \Delta X \sim \text{Levy}(\alpha) = \propto \Delta X^{-1-\alpha}$

$\Delta x \in [1, \infty[$



$$X(t+1) = X(t) + \Delta x(t)$$

In 2D



$\theta \sim Unif(0, 2\pi)$

The choice of the distribution may have an impact on the way the space is explored.

Power law distribution allow large amplitude events to occur. Not always, but frequently enough to have an impact.

Levy flight are the way animal hunt. They stay in a same region for a while, then jump far away.

5.6 Cuckoo Search

Proposed by Yand and Deb in 2010

One more metaheuristics based on a metaphor of animal behavior.

Adapted to continuous optimization

- Cuckoos are birds that lay their egg in the nest of another bird. Some times these eggs are rejected.

For an algorithm, we have the following correspondance:

- Nests: an array containing solution
- Eggs: a possible solution
- Eggs are rejected: solution are discarded and replaced.

- Each solution is considered for an improvement, by a random movement. If it is good, it may replace another solution (egg) ↑ levy flight
- At the end of each iteration, a fraction Pa of the worst solution are discarded and regenerated at random.

Algorithm

```
alpha = 1.5          # a Levy flight exponent
iter = 0
solution = [ random( )  for i in range(nest) ]      # here S = [0, 1]
while iter < maxIter :
    iter += 1
    i = randint(0, nest-1)      # selection of a first nest (or index)
    x = solution[i] + levy(alpha)      # attempt to improve this solution
    x = x - int(x)              # wrap the new solution to [0, 1[
    j = randint(0, nest-1)      # select a second nest in view of a potential replacement
    if fitness(x) > fitness(solution[j]) :
        solution[j] = x         # replacement
    solution.sort(key = fitness)      # sorted by increasing fitness
    solution[0 : nest/4] = [random( ) for i in range(nest/4)]
                                # one replaces one quarter of the worst solutions
best = solution[nest-1]         # because of the sorting, that is the best
```

Chapter 6: Evolutionary Algorithms

6.1 Introduction:

It is a family of metaheuristics base on the idea of Darwinian Evolution

- Basically, the best individuals are selected and able to reproduce.
- Evolution is driven by mutation and crossover (croisement), i.e by combining succesful individuals

- These metaheuristics appeared in the 1970s
  - Genetic Algorithms (GA) proposed by J. Holland
  - Evolution Strategy (Rechenberg)
  - Genetic Programming (Fogel + Koza)

The goal is to implement a simulated evolution process to create good solution to an optimization problem.

6.2 Genetic Algorithms

- We have a population of individuals (= solutions)
- The coding of an individual in S is called its chromosome or its genes.
- The level of adaptation of an individual is determined by its fitness.
- Here we mostly consider problems of maximization.

The population of solutions evolves

at each iteration (= generation), where the individuals have been replaced by their offsprings. Evolution follows these steps:

- The best individuals are selected
- They are then subject to mutations (= random modification) and crossover (= hybridization, recombination) among pairs of individuals.

- The best solution found so far is automatically inserted in the new population. (it typically replaces the worst one)
- The size of the population is constant during all the evolution.

Pseudo-code

```
generation = 0
initialize the population
while (not end-condition)
    generation += 1
    compute the fitness of each individual
```

select individuals
crossover
mutation
insert best
end while
print best.

$P(t) \xrightarrow{\text{select}} P'(t) \xrightarrow{\text{cross over}} P''(t) \xrightarrow{\text{mutation}} P'''(t) = P(t+1)$

↑ population at generation t

Note that P, P', P'' and P''' have all the same size. We have to define select, crossover and mutation to guarantee this property.

Selection (select)

$P(t) \xrightarrow{\text{select}} P'(t)$

One draw at random n times (with replacement) an individual from P, where n is the population size.

Typically the probability to select one individual is proportional to its fitness.

But other selection mechanisms will be discussed.

After the selection phase, the population P' may have several copies of a same individual. High fitness individuals will be over-represented.

Crossover

$$P'(t) \xrightarrow{\text{crossover}} P''(t)$$

The n individuals resulting from the selection are the parents of the next generation:

- New individuals are obtained by hybridization of two selected parents.
- For instance, let (S_i, S_{i+1}) a pair of parent, where $i = 1, \ldots n$ is labeling the individuals.
- With a probability $P_{crossover}$ we

mix the genomes of the two parents. So as to produce to children that will replace the parents.

- with prob $1 - P_{crossover}$, the two parents become the children, without change.

Note: 2 parents $\longrightarrow$ 2 children, always.

We stop crossover when we generated n children.

Mutation

$P''(t) \xrightarrow{\text{mutate}} P'''(t)$

On the n offsprings generated by crossover, we apply random mutation with a probability $p_{mutation}$

This simulates the transcription errors in the evolution process.

A mutation is a random movement in S.

Metaheuristics Nov 14. 2022

Chapter 6 : Evolutionary Algorithms.

Genetic Algorithm (GA)

A population $P(t)$ of solutions (= individuals) that experience Darwinian evolution

$$P(t) \xrightarrow{\text{select}} P'(t) \xrightarrow{\text{crossover}} P''(t) \xrightarrow{\text{mutation}} P'''(t) = P(t+1)$$

The size n of all these population is constant.

Exploration versus exploitation

Select. favors exploitation (the selection

current good solutions)

Mutation: favors exploration by adding random modification to all individuals

Crossover: It contributes to both depending on the coding of the individuals and how crossover is done.

- Can create completely new individuals
- Or can combine two good individual to create an ever better one.

In GA there are guiding parameters:

- N, the population size
- The crossover and mutation probabilities
- The detailed expression of select, crossover, mutation that may introduce other parameters

6.3 A simple example

We will illustrate the different step of

a GA on the MaxOne problem.

We want to build the bit string that contains the max number of bits equal to one. $X_{opt} = (1\,1\,1 \cdots 1)$

but the GA does not know it. It only know a fitnes value.

$$f(x) = \sum_{i=1}^{\ell} x_i \qquad x_i \in \{0, 1\}$$

ℓ components

For the sake of illustration, we consider a small population $n=6$, and short bit strings, $\ell=10$

Initial population

Let us draw at random ℓ bits for each of the n individuals. Let's assume we get

$S_1 = 111\ 101\ 0101$	$f(S_1) = 7$	$P_1 = 7/34$
$S_2 = 011\ 100\ 0101$	$f(S_2) = 5$	$P_2 = 5/34$
$S_3 = 111\ 011\ 0101$	$f(S_3) = 7$	
$S_4 = 010\ 001\ 0011$	$f(S_4) = 4$	
$S_5 = 111\ 011\ 1101$	$f(S_5) = 8$	
$S_6 = 010\ 011\ 0000$	$f(S_6) = 3$	$P_6 = 6/34$

We will compute the fitness of 34 each individuals in view of selecting the best

Selection

A simple mechanism to select an individual is with a probability proportional to its fitness. One will take S_i with probability p_i

$$p_i = \frac{f(S_i)}{\sum_{k=1}^{n} f(S_k)}$$

34 here

$p_1 = 7/34 = 0.206$

$p_6 = 3/34 = 0.088$

See TP 6 to select individuals with their prob p_i

P'(+)

Suppose we get

$S_1' = 111\ 101\ 0101 \quad (S_4)$

$S_2' = 111\ 011\ 0101 \quad (S_3)$

$S_3' = 111\ 011\ 1101 \quad (S_5)$

$S_4' = 011\ 100\ 0101 \quad (S_2)$

$S_5' = 010\ 001\ 0011 \quad (S_4)$

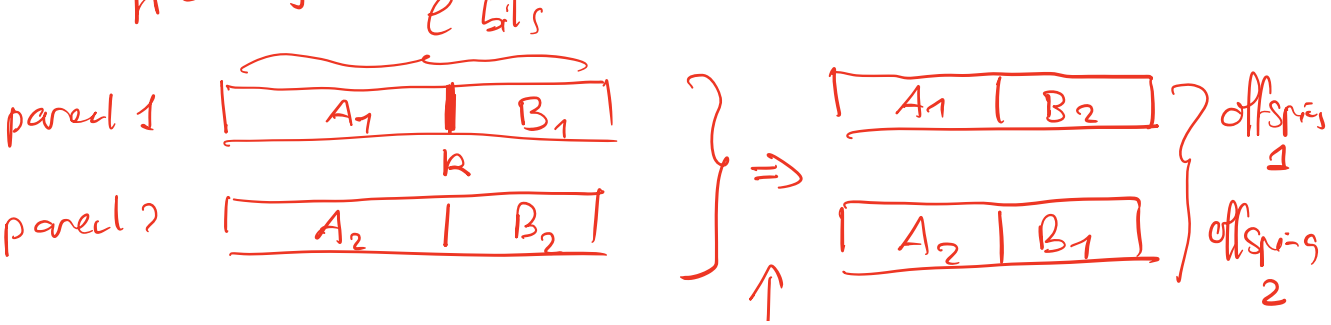
$S_6' = 111\ 011\ 1101 \quad (S_5)$

Note that S_6 was not selected and S_5 was selected twice

Crossover $P''(t)$

The individuals S_i' will recombine to produce a new set of individuals S_i''

We will then take the first k bits from parents one with the last $\ell - k$ bits of parents two to produce an offspring



with prob $p_{crossover} = 0.6$

k is a parameter, call the crossing point. Will be chosen at random for each crossover.

With prob $1 - p_{crossover}$,

$$offspring_1 = parent_1$$
$$offspring_2 = parent_2$$

To decide which are $parent_1$ and $parent_2$ we proceed as follows:

(s_1', s_2'), (s_3', s_4'), (s_5', s_6')

first couple, second couple, third couple

Example:

$k = 5$; case of (s_5', s_6')

k

$s_5' = 010\ 00 \cdot 1\ 0011$

$s_6' = 111\ 01 \cdot 1\ 1101$

After hybridyzation we get:

$$S_5'' = 010\ 00.1\ 1101$$

$$S_6'' = 111\ 01.1\ 0011$$

We see that S_5'' and S_6'' are new individuals. They bring novelty in the population.

Example 2

The pair $(S_1'\ S_2')$, $k = 2$

$$S_1' = 11.1\ 101\ 0101$$

$$S_2' = 11.1\ 011\ 0101$$

After crossover

$$S_1'' = 11.1\ 011\ 0101 \qquad (= S_2')$$

$$S_2'' = 11.1\ 101\ 0101 \qquad (= S_1')$$

Here no novelty because the first k bits of S_1' and S_2' are the same.

Mutations $P'''(t)$

For the mutation, each bit of each individual can be modified (e.g flipped : $S_i'' = 1 - S_i''$) with a probability $p_{mutation}$, usually small. Here we take

$$p_{mut} = 0.1$$

With 6 individuals, each with 10 bits, we have 60 bits that will be flipped with prob 0.1; we expect 6 of them to be affected ($6 = 0.1 \times 60$) on average.

Let's assume that we have the following.

$S_1'' = 111\ 01\bar{1}\ 0101$

$S_2'' = 111\ \bar{1}0\bar{1}\ 010\bar{1}$

$\bar{\ }$ mean that a mutation will flip that bit

Note that as the quality of the individual increases, there will be more 1s and thus more chance that a mutation flips a 1 into a 0 that the opposite. Mutation may degrade the quality after several iterations..

But remember that the best individual of P(t) will replace the worst of $P'''(t)$ so we do not loose good solutions.

Finally let us assume we get:

$S_1(t+1) = S_1''' = 111\ 010\ 0101 \quad f_1 = 6$

$S_2''' = 111\ 111\ 0100 \quad f_2 = 7$

$S_3''' = 111\ 010\ 1111 \quad f_3 = 8$

$S_4''' = 011\ 100\ 0101 \quad f_4 = 5$

$S_5''' = 010\ 001\ 1101 \quad f_5 = 5$

$S_6(t+1) = S_0''' = 111\ 011\ 0001 \quad f_6 = 6$

37 (before it was 34)

There is a 9% improvement (approx)

there is a no improvement compare
to the previous iteration.

The process will continue until an end criteria is reached. (typically stagnation of the fitness, or amount of computational effort)
↑ max number of generations

A more interesting case

We consider a typical value for n, $n = 100$

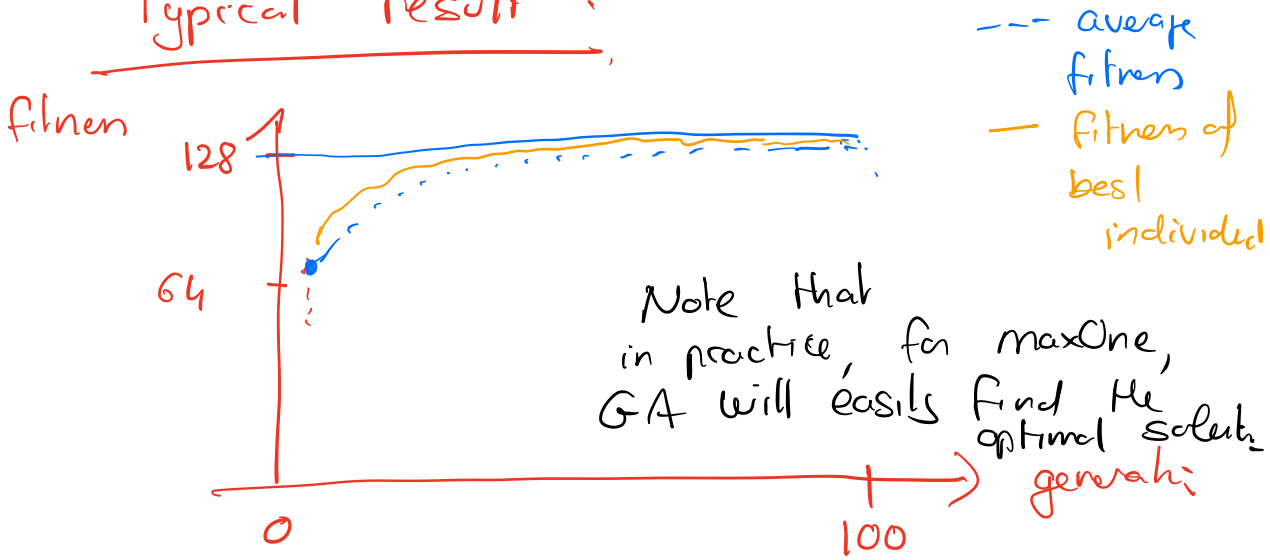
We will make the problem more realistic, with bit strings of length $\ell = 128$

Here we stop after 100 generations or when the optimal string is found (here we know it, since this is a simple benchmark)

The value of $P_{crossover}$ is 0.8

$P_{mutation}$ is 0.05

Typical result:



The initial fitnes is typically $\ell/2$ as each bit is 0 with prob $1/2$ and 1 with prob $1/2$.

It is typical of GA to observe a fast improvement of the fitnes at the early generations, then a saturation.

In general the average fitnes is growing, although, in principle, it could decrease.

Note also that, as generations go on, population diversity goes down. We end up with many copies of the

Same individuals, and crossover does not do any change.

6.4 Cas of a continuous function in $\mathbb{R}$

The search space is $S \subseteq \mathbb{R}$, actually an interval $[a, b]$

We will code an x in S as a set of ℓ bits.

with ℓ bits, we have numbers between 0 and $2^\ell - 1$.

Suppose we want to code for real values between x_{min} and x_{max}

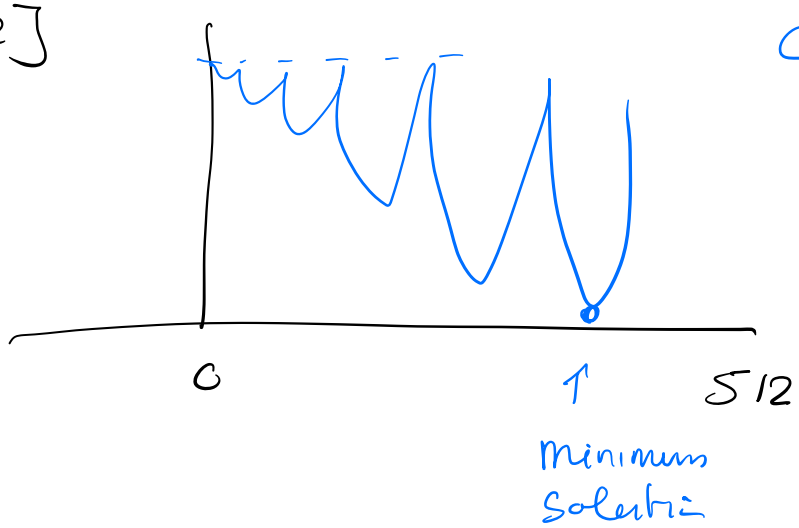
$$X = \frac{(x_{max} - x_{min})}{2^\ell - 1} \cdot S + x_{min} \quad \text{where} \quad S \in \{0, 1 .. 2^{\ell-1}\}$$

$\uparrow$ bit string

Example : 0 . 0 . . 0

$f(x) = -|x \sin(\sqrt{|x|})| + c$

$S = [0, 512]$



$\ell = 10$, $x = \frac{512}{1023} \cdot S$

$\frac{512}{1023} \sim 0.5$ $\Rightarrow$ this implies a limited accuracy on x the optimal solution will be found at this accuracy.

$S = \overbrace{00\ldots0}^{\ell}$

$S = \overbrace{11\ldots1}^{\ell}$

In the example above we consider a fixed point coding of a real number. But we can also use a floating point approach

c $X = \left(b_0 + \sum_{k=1}^{p-1} b_k 2^{-k} \right) \times 2^t$

with $t = \sum_{k=p}^{n-1} b_k 2^{k+2-p}$

This is a n-bit coding of X

with $b_i \in \{0, 1\}$

$X = S \times 2^t$ ← exponent.

↑ mantisse

Here we have p bits for the mantisse and $q = n - p$ for the exponent.

We can apply the GA algorithms such as crossover and mutation separately on the mantisse $(b_0 b_1 \ldots b_p)$ and the exponent $(b_{p+1}, b_{p+2} \ldots b_n)$

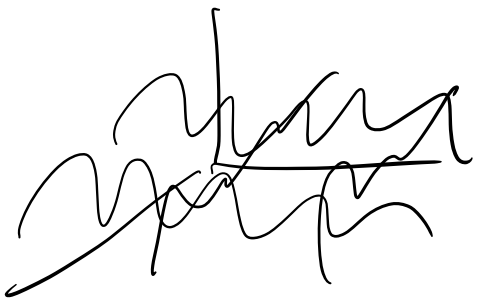
Note: the IEEE 754 standard for n=64 bit is:

1 bit for sign

52 bits for the mantiss (16 decimal digit)

11 bits for the exponent. (of accuracy) (-1024 → 1023)

Such a floating point coding for GA has been used for these benchmarks: Rastrigin and Schwefel function.



These functions are not difficult for GA.

6.5 Other selection operators

The goal is to take n individuals from the current population, in view of the crossover phase.

- A deterministic solution: choose a criteria that decides which individual are selected and which are rejected

Not really used in GA

- Stochastic approach. Try to favor the selection of best individuals

Among those stochastic approaches we have:

- Fitness proportionate, that was used in the example:

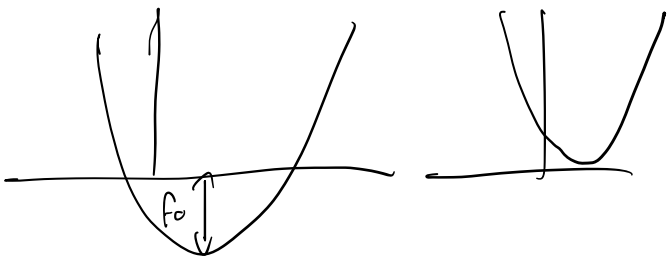
$$p_i = \frac{f_i}{\sum_j f_j}$$

prob to select individual i

Constraint:
- f_i must be positive
- Works for a maximization problem.
- The evolution tends to produce a population of similar fitness

When all the f_i are similar, all the p_i are similar and selection is no longer selective.

Solution: $f_i > 0$: add f_0 to f_i



so that tot $f_i > 0$

Shift of the fitness by f_0 well chosen.

- Minimization: take $-f$ (or $1-p_i$)

- To restore selectivity: scale the fitness of the population between its min and max

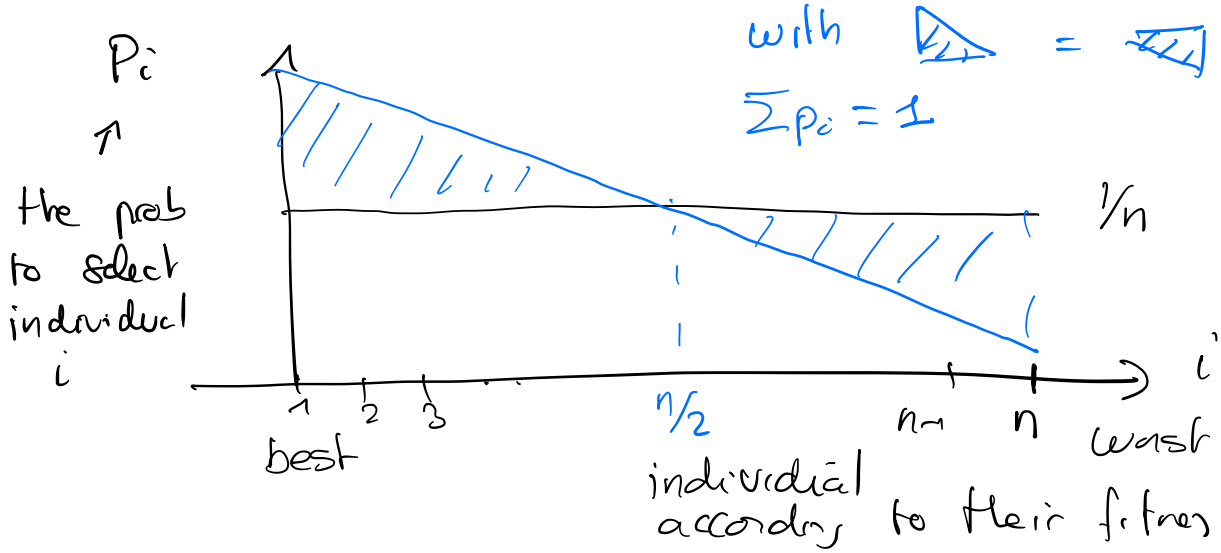
$$\bar{f}_i = f_i - \min_{j \in P} f_j$$

This is a way to amplify the difference between individuals.

Selection by rank

Here we start by sorting the individual by their fitness, the best to the worst.

It works for min or max problems, does not suffer from the sign of f



$\Rightarrow$ This is called a selection by linear rank.

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

$$i = 1, 2, \ldots n$$

where β is a parameter.

We have $p_1 = \frac{\beta}{n} \Rightarrow \beta > 1$ otherwise it is flat

$$p_n = \frac{2-\beta}{n} > 0 \quad \beta \leq 2$$

Metaheuristics, Nov 21, 2022

Genetic Algorithms.

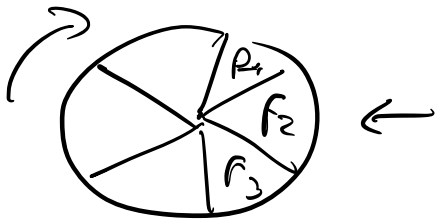
$P(t) \xrightarrow{\text{select.}} P'(t) \xrightarrow{\text{cross over}} P''(t)$

$\xrightarrow{\text{mutate.}} P(t+1)$

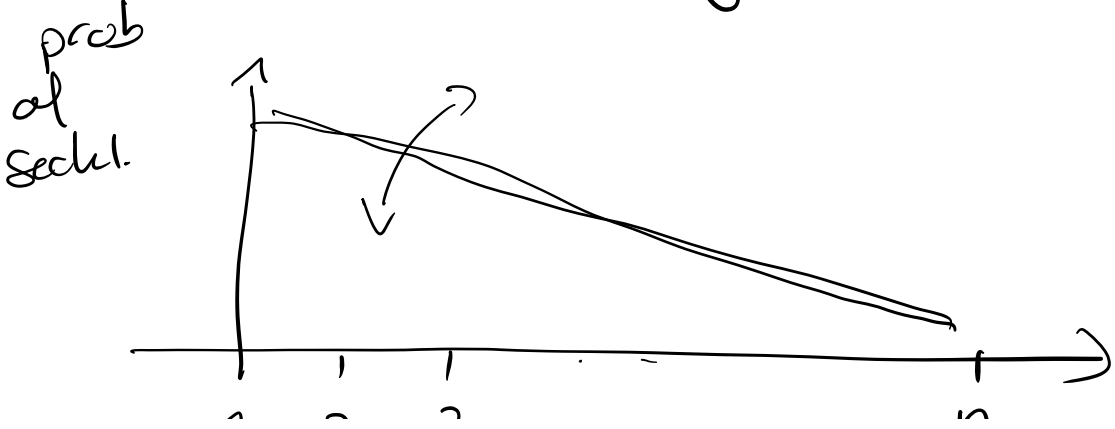
Different selection operator

1) fitness proportionate

(Roulette wheel)



2) Selection by rank



1 2 3 ... n

best fitness individual

worst individual

3) Tournament selection

Select at random, uniformly k individuals from the population that will compete for selection.

The winner is the one with the best fitness and it is selected for P'

This is repeated n times, that is we consider

n k-tournaments → n selected individuals

- The implementation is easy
- The value of k is a guiding parameter.

  $k = 1$: random, uniform selection

  $k = n$: select the best.

  In practice $k \in \{2, 3, .. 10\}$ gives good results.

- Tournament selection is a standard choice as it avoids the problems or complication of the other methods

6.6 Takeover time : intensity of selection

The takeover time is the number of iterations of select (without mutation and crossover) so that the best individual

so that the best individual

invades the entire population

The best individual will spread in the population as select is repeated. The less good individual will be progressively eliminated.

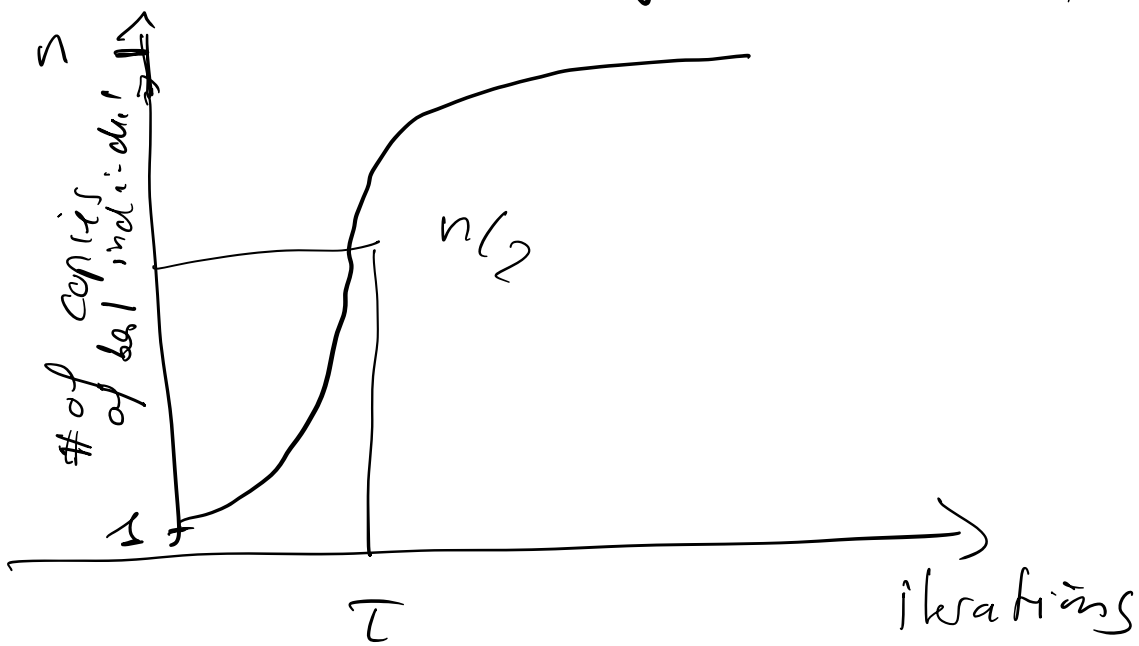
- If τ is small we say that the selection is strong because very quickly, all the population is made of the best individual.
- If τ is large, selection is weak. The best individual is slowly replicated.

Note: for a k-tournament with $k = n$, $\tau = 1$

In practice as select is a

stochastic process, τ is defined as the average time to reach $\frac{n}{2}$ copies of the best individual.

For instance, we typically have the following behavior:



It is expected that the number $m(t)$ of copies of the best individual evolves

as a logistic equation

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

This can be understood as follows:

$$m(t+1) = n \times \text{Prob}(\text{select best})$$

With a probability

$$\frac{m(t)}{n}$$

of selecting the best individual we have:

prob of selecting best with α a coef dep. on fitness

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

fraction of space left for adding more copies of best.

$$\Rightarrow \quad = \alpha m \left(1 - \frac{m}{n}\right)$$

Let us consider the case of a k-tournament to compute $m(t)$ more rigorously.

To select the best individual in a k-tournament, it must be present in the tournament.

When choosing k individual uniformly for the tournament, the prob of never choosing the best one is

k ← k individual are chosen.

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

$\frac{m}{n}$ ~ prob to select

the best: m copies over n choice.

Thus the prob that the best individual is in the k-chosen individual (and then wins) is $\left(1 - \left(1 - \frac{m}{n}\right)^k\right)$

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

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

if $k \geq 2$

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

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

$$\Rightarrow \underbrace{m(t+1) - m(t)}_{\Delta m} = m\left(1 - \frac{m}{n}\right)$$

$\simeq \frac{dm}{dt}$

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

is $$m(t) = \frac{n}{1 + \frac{n - m_0}{m_0} e^{-t}}$$

where $m_0 = m(t=0)$

Solving $m(t) = \frac{n}{2}$ gives.

$$\Rightarrow t = \ln(n-1)$$

$$\Rightarrow \boxed{T = O(\ln(n))}$$

$\Rightarrow$ take over time is short with respect to population size : $T = 10$ with $n = 1024$

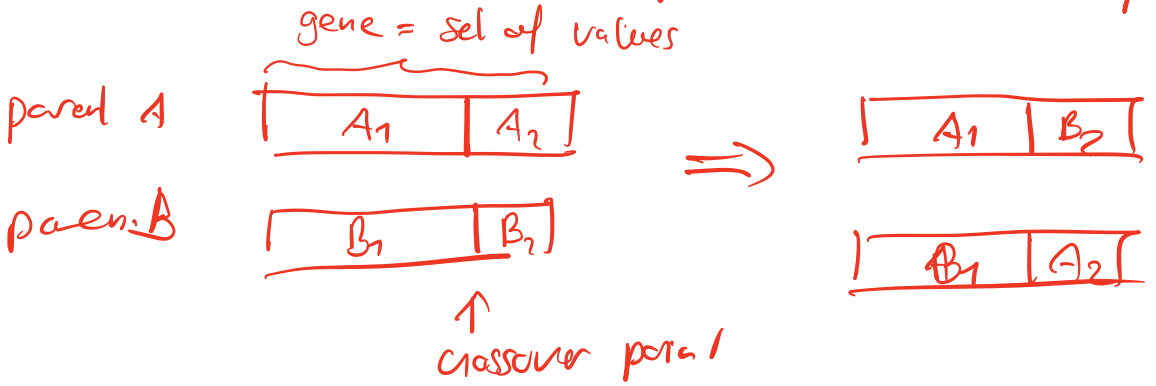
It turns out that fitness

proportionate selection has the same behaviour: $T \sim \log n$

Selection is always strong and mutation and crossover will be needed to create new material and diversity.

6.7 Other crossover and mutation operators

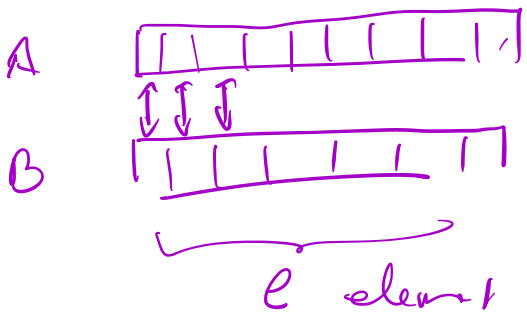
We saw the 1-point crossover operation.



The gene is a set of values that determine the solution: often, 0s or 1s,

but can be real numbers, text, and more.

But we can think of different ways to combine two solutions A and B

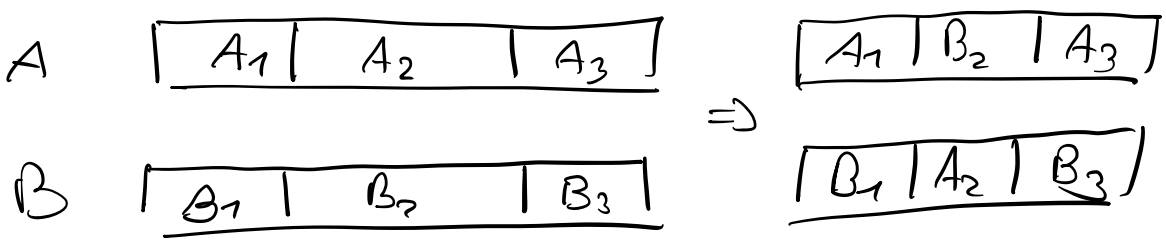


we take each element of the gene, either from A or from B with prob $1/2$

$\Rightarrow$ this is called the

uniform crossover

There is also the two-point crossover:



For some search space, the above crossover do not make sense.

The offspring should still be in the search space ▽

Let us consider the case of the permutation space.

A: 1 - 2 - 4 - | 3 - 8 - 5 - 6 7

B = 1 - 3 4 - | 8 - 2 - 5 - 7 - 6
1 2 x 3 4 5

after such a crossover we get:

A' = 1 - 2 - 4 - 8 - 2 - 5 - 7 - 6
B' = 1 - 3 - 4 - 3 - 8 - 5 - 6 - 7

} no longer permutation of 8 objects.

Work around:

Take the first part of A and complement with B, in the order they appear.

1 - 2 - 4 | 3 - 8 - 5 - 7 - 6

This is a valid combination of A and B

The same with B can be done:

1 - 3 - 4 | 2 ....

But more possibility to define crossover exist

Mutations:

There are also many ways to define mutations: they are random transformation or movement in the search space.

For instance, for the permutation space, we can swap two objects, or insert one at a random place.

6.8 Usefulness of the genetic operators

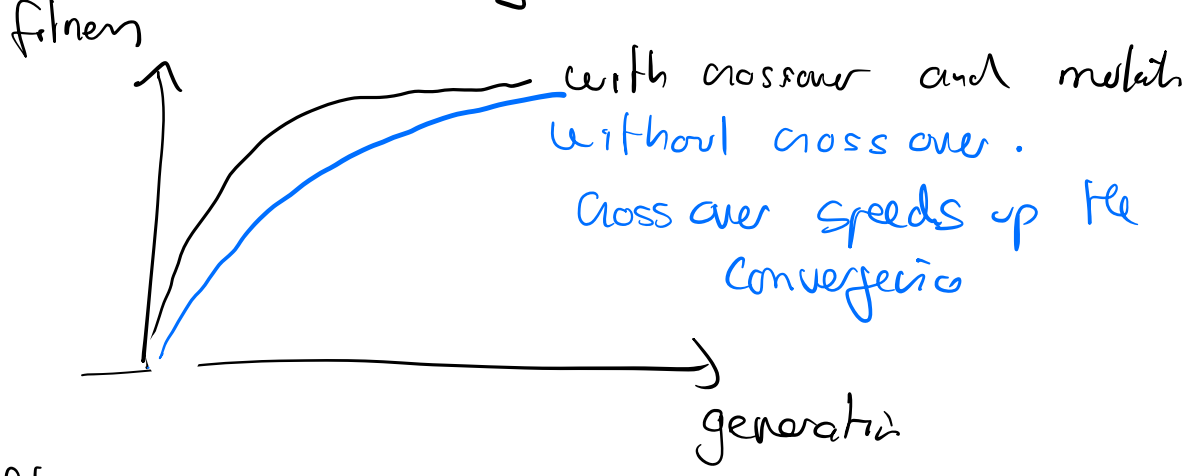
We want to explore numerically the effect of the operators such as selection, crossover and mutation.

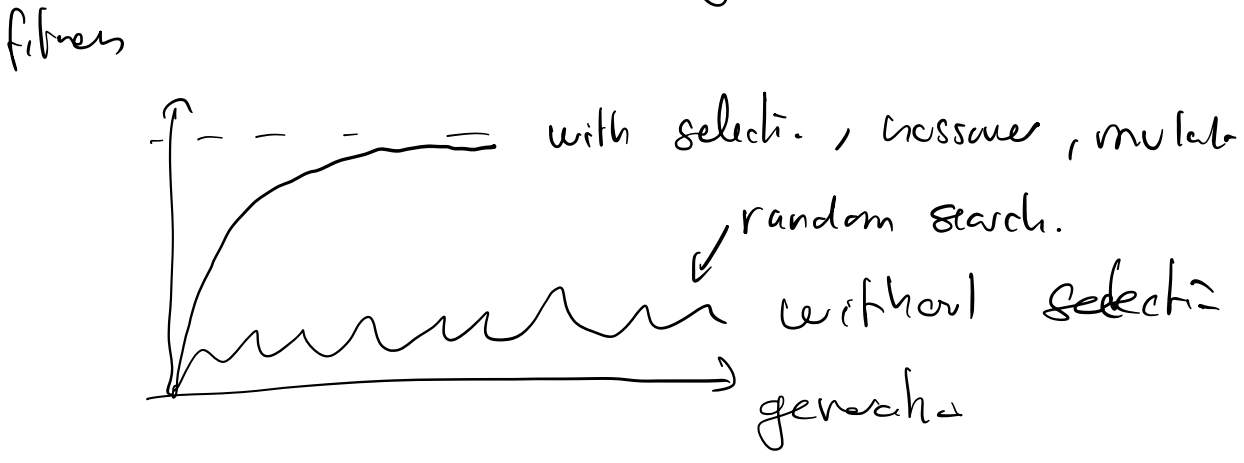
Is crossover really useful? or mutation is enough to bring novelty?

We will run a GA and turn on and

off selection or crossover. Mutation will always be present.

The typical behavior for a NK problem is the following.





6.8 Schema Theorem

The goal is to propose an explanation about the convergence of a GA

The main finding is that patterns in the chromosomes that have a better fitness are amplified by evolution

The final solution is obtained by a combination of these high fitness patterns

Schema (or pattern)

A schema is a regular expression which specifies the value of some bits

For instance

$$S = (1 * * 1 * 0)$$

An individual matches the schema if it has the same value on the imposed position.

(100110) matches (1 * * 1 * 0)

Order and length of a schema

The order $o(S)$ of a schema S is the number of imposed position.

For our example $o(S) = 3$

The length $\delta(S)$ of a schema S is the distance between the last and first imposed position

$$\delta(\underset{1}{1}\ *\ *\ \underset{4}{1}\ *\ \underset{6}{0}) = 6 - 1 = 5$$

$$\delta(*\ *\ *\ \underset{4}{1}\ *\ *) = 4 - 4 = 0$$

$$\delta(*\ \underset{2}{0}\ *\ \underset{4}{1}\ *\ *) = 4 - 2 = 2$$

Fitness of a schema

This is the average fitness of all individuals in the population that match the schema

$$f_P(S) = \frac{1}{M(S)} \sum_{i=1}^{M(S)} f(v_i)$$

- $f_P(S)$: P = population, S = schema
- $M(S)$: number of individual having S
- v_i is an individual with schema S

$v_1, v_2 \ldots v_{n(S)}$ are the individuals in P that satisfy S.

The total fitness is defined as

$$F(t) = \sum_{v \in P(t)} f(v)$$

(t: generation)

Evolution of M(S) during the evolution

$$M(S, t+1) = M'(S, t) \times [\text{prob of survival after crossover}] \times [\text{prob of survival of S after mutation}]$$

where $M'(S, t)$ is the number of individual obeying S after selection.

If $M(S, t+1) > M(S, t)$, then the schema S is amplified: more individuals have it.

otherwise it tends to disappear.

$M'(S, t)$ can be computed in the same way as in the takeover time section

$$M'(S,t) = n \underbrace{\frac{M(S,t) f(S,t)}{F(t)}}_{\text{prob to select an individual obeying } S}$$

$$= \frac{M(S,t)\, f(S,t)}{\bar{F}(t)} \qquad \text{with } \bar{F} = \frac{F}{n}$$

is the average population fitness

Is $M' > M$?

Yes if $f(S,t)/\bar{F}(t) > 1$

Then, it means that a schema S whose fitness is larger than the average fitness $\bar{F}$ will be amplified by selection.

Now we need to compute the effect of crossover and mutation to preserve S.

A schema S is destroyed by a crossover if the crossover point falls within S.

The crossover point is chosen at random, and with let say m genes (or m bits) the prob to break S is less than:

$$\frac{\delta(S)}{\quad} \leftarrow \text{lenght of } S$$

$m-1$ ← number of crossover point with $m-1$ possible positions

If is less than that if the second parent has the same schema.

Crossover is performed with prob p_c.

Thus schema S survives the crossover with a prob [larger] than

$$1 - p_c \frac{\delta(s)}{m-1}$$

Thus, with respect to crossover, we have:

$$M(s, t+1) \geq \frac{M(s,t)\, f(s,t)}{\bar{F}(t)} \left[1 - p_c \frac{\delta(s)}{m-1} \right]$$

We now need to add the mutation part.

Mutations

$O(s)$ is the number of imposed values in S

These values will survived if not muted.

A mutation of a gene (or a bit) is done with prob p_m

Then the schema S will survive mutation with prob

$$(1-p_m)^{O(s)} \approx 1 - O(s)\, p_m \quad \text{if } p_m \ll 1$$

$\underbrace{\quad}$ no mut

to change the bit

Also, when $p_m \ll 1$

$$\left(1 - p_c \frac{\delta(s)}{m-1}\right)\left(1 - o(s) p_m\right) \approx \left(1 - p_c \frac{\delta(s)}{m-1} - o(s) p_m\right)$$

+ correction

And then we get:

$$M(s, t+1) \geq M(s, t) \frac{f(s,t)}{\bar{F}(t)} \left[1 - p_c \frac{\delta(s)}{m-1} - o(s) \cdot p_m\right]$$

We see that for $M(s, t+1) > M(s, t)$, $\delta(s)$ and $o(s)$ should not be too large.

Small schema of short order of fitness better than the average are amplified. But this will occur at the beginning, because after a while, these good schema will increase $\bar{F}$ and the amplification will stagnate.

Overall conclusion: the optimal solution found by a GA is the juxtaposition of short succesful schema.

But in practice, it is not always the case and convergence to suboptimal solutions happens

Metaheuristics, 28 Nov, 2022

6.9 Structured Populations

In standard GA, there is one population and individuals can hybridize with any other.
It is a big mix of the population (a panmictic population)

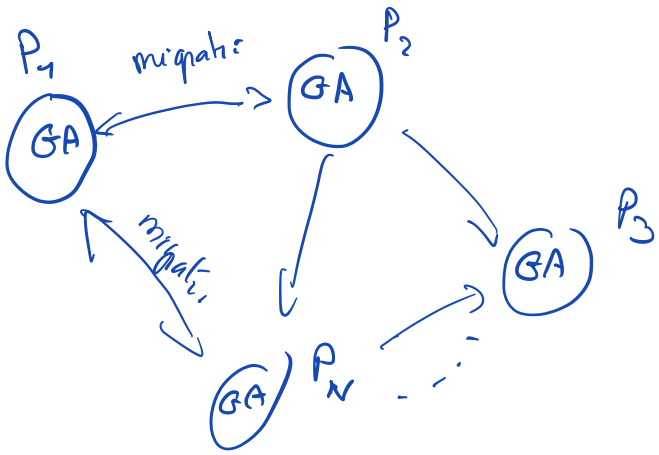
The problem of one unique population is that evolution tends to decrease diversity. ⇒ convergence to a maybe sub optimal solution.

Nature suggests that having isolated sub population may generate news, unique solutions

In GA, we can borrow this idea and have sub populations, each evolving on

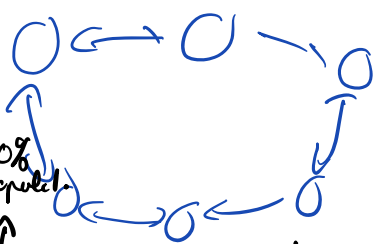
their own way. We may also add migration, the exchange of a few individuals from one population to the other.

P_c : sub population



Usually, the interconnexion topology does not have a strong effect: so a ring topology is good enough;

- how many subpopulations? a few tens
- size of migration? 5-10% of population
- when to migrate. every 10 generations, copy of best individual, replacing the worst



- size of subpopulation? about 50 individuals

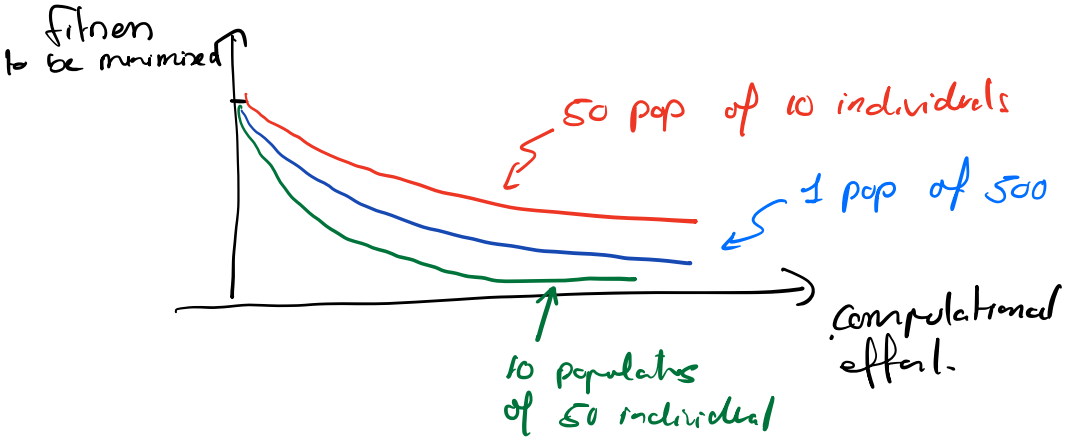
Such a scheme is easily parallelized: one sub-population per processor.

If migration is done in a asynchronous way, no population has to wait for immigrand, and there are no load balancing problems.

This type of GA is called multi-population GA or island GA. Indeed it is better than

a single GA with the sum of all individuals.

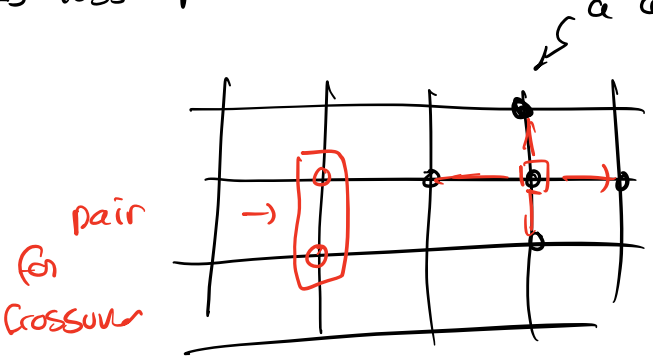
- It goes faster in terms of convergence: less evolution steps. It also goes faster if parallelization is used.



There is a optimal choice of the number of populations and their size.

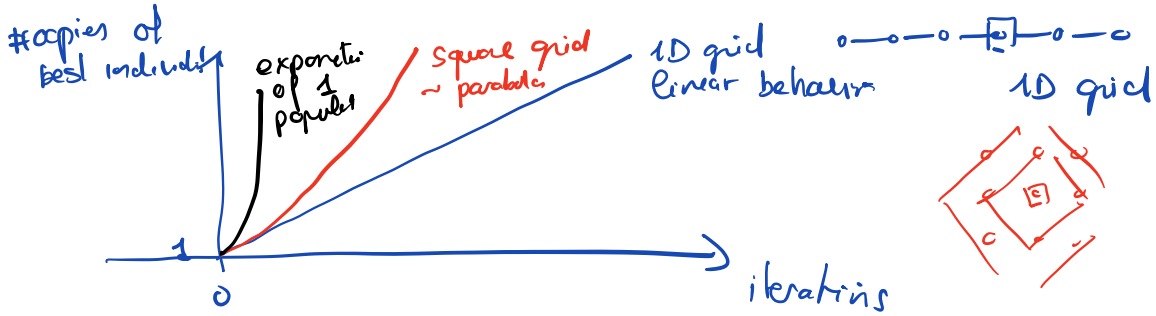
"Cellular" population

This is another way to structure a population and avoid that anyone recombine with anyone (⇒ loss of diversity).



- Selection: tournament between an individual and its neighbor.
- Crossover: recombine with a neighbor, chosen at random

The goal is to increase takeover time and diversity:



Chapter 7 : Genetic programming GP

7.1 Introduction : mostly developed by J. Koza in the 1990s

The search space is a set of computer programms and not a set of numerical values.

Goal : evolve a computer programm so that it can solve a given problem.

$$prog\ P \xrightarrow{\text{mutation}} P'$$

$$P_1\ P_2 \longrightarrow \begin{cases} P_1' \\ P_2' \end{cases} \quad \text{crossover}$$

What we expect of such a program :

- Reproduce a given output from a given input.
- Should be able of generalization : capture the relation between the input and output.

- ⇒ algebraic curve filling, symbolic regression, classification task. Exple: trading model in finance
- Find boundary conditions for a simulation
- Find optimal behavior of robots in a given environment.

⇒ This is a machine learning task! but also an optimization task: how well the "best program" does the job.

10^{20}

7.2 Coding of a program

How can we represent a program so that the genetic operations will not destroy them. We need to produce individuals that are executable.

We will need to simplify the instruction set and the structure of the programms. We will present two approaches:

- Tree based programms
- Stack based linear programs (sequence of instruction)

Fitness calculation.

Let p be a program:

$$y = p(x)$$

y is the output, x the input, where x and y can be complicated data structures.

We consider a training set (learning set) $A = \{\langle x_1, y_1\rangle, \langle x_2, y_2\rangle \dots \langle x_k, y_k\rangle\}$

a collection of examples, with <input, output> pairs.

The fitness $f(p)$ of program p is defined as

$$f(p) = \sum_{i=1}^{k} d(y_i - p(x_i)) = \sum_{i=1}^{k} |y_i - p(x_i)|$$

↑ some distance function

↑ for instance, $d = |.|$

The goal is to minimize f (also called the loss function in ML)

One can further evaluate p with a test set B, with additional examples not used for the training. The fitness of p over this test set should also be good: generalization is achieved and the relation between x and y is discovered.

Otherwise, we say that we have overfitting, meaning that p is only good on the training set but not for new inputs

To define a program, we restrict ourself to the case of finding an expression (algebraic) relating x and y:

For example, we may have observations (x_i, y_i) and we want the relation

$$y = g(x)$$

where g is for instance $g(x) = 2x^2 - 3x + 1$

GP will have to find such an expression.
It is obviously less than asking for more complicated task, such as sorting.

Coding: S-expression (or tree-based expression)

Koza proposed to use a functional language to code P, so that genetic operator can be applied

A typical S-expression is:

$(+\ x\ 2)$ which means $x+2$

$(+\ (*\ (*\ x\ x)\ 2)\ (+\ (*\ 3\ x)\ 1))$

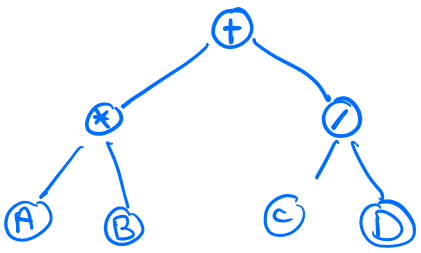
$(*\ x\ x) = x^2$, $(*\ (*\ x\ x)\ 2) = 2x^2$, $(*\ 3\ x) = 3x$, $(+\ (*\ 3\ x)\ 1) = 3x+1$

is $2x^2 + 3x + 1$

One can replace an operand in an S-expression by another S-expression and get something which is valid.

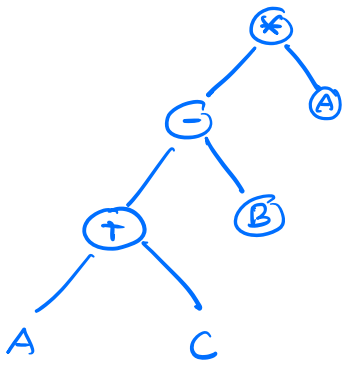
This representation as S-expression can be shown graphically as a tree

(+ (* A B) (/ C D))



A * B + C / D

(* (- (+ A C) B) A)



A * (A + C - B)

Function and terminal sets

A genetic program is built out of function and terminals.

- F is the set of function (node) used in the tree
- T is the set of variables (leaves) used to specify the input.

A simple example is:

$$F = \{+, -, *, /\}$$

$$T = \{A, B, C, 1, 0\}$$

we can specify constant that may be useful.

where A, B, C are names of variables whose values are specified in the training set:

$$A = \{ \langle A_1, B_1, C_1, y_1 \rangle, \langle A_2, B_2, C_2, y_2 \rangle \dots$$

(A_1, B_1, C_1: input; y_1: output)

Another example: Boolean regression

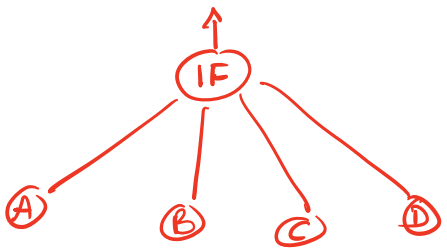
$$F = \{ \text{AND}, \text{NOT}, \text{OR} \}$$

$$T = \{ b_1, b_2 \dots b_k, \text{TRUE}, \text{FALSE} \}$$

($b_1 \dots b_k$: problem variables)

IF operator

On can also define more operators, such as branches.

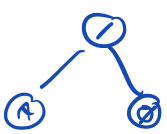


if A ⩾ B then C
else D

But different forms of IFs are also possible.

Properties of the functions in the function set

- Be sure to have the right number of arguments
- Closure property: each function should be able to accept any value returned be the other function



a division by zero should be possible. Division should be extended to

accept any value
A/0 -> A ??

A genetic program is then
any tree with leaves in T and nodes in F

Initial Population

How to create a program out of T and F

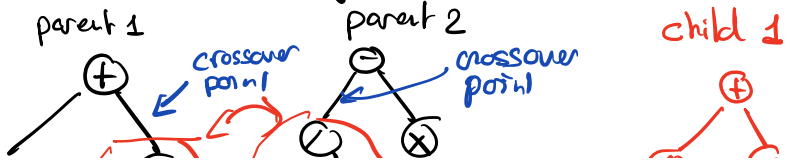
- Each node should be taken from F or T, at random.
- If we take a function as a node, we have to fill the value of the argument of this function
- If we choose a terminal as a node, it stops the tree for this branch.

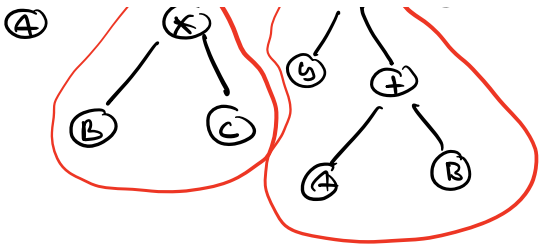
A program can be created at random by choosing a symbol either in F or in T, as long as there are no more open branch

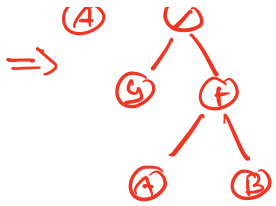
- To control the depth of the tree, one can reduce the probability of choosing from F and increase the prob to choose from T.

Crossover

We can exchange subtrees between two individuals

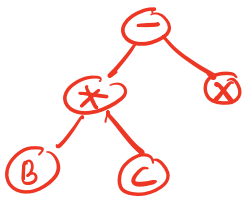






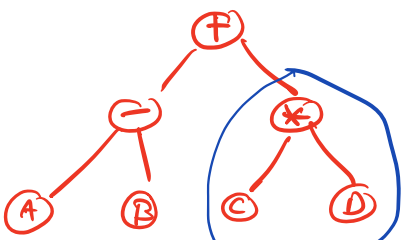
where do we choose the crossover points ?
It is chosen at random, but usually with a non uniform probability to favor a cut near the root rather than near the leaves.

Child 2 :

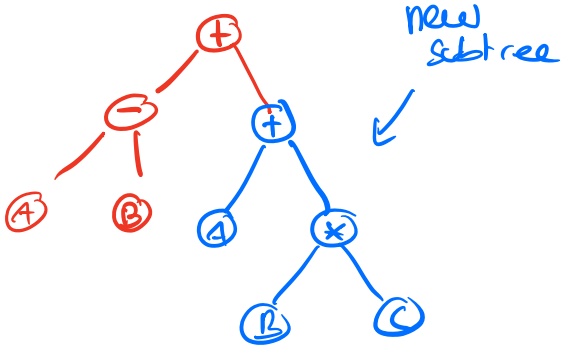


Mutation

parent

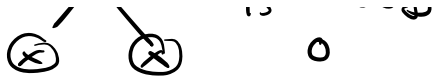


mutation by regrowth: this subtree



- It is observed that such crossover and mutation tend to increase the depth of the trees. This is called "bloat" and this is not good in term of efficiency : long to evaluate, more memory; Usually one impose a maximum depth !

- Also, simplification can be useful. Some subtree do nothing: $x - x$ is always



One can use some techniques to replace a subtree by a simpler one doing the same result.

7.3 Example : create a trading model

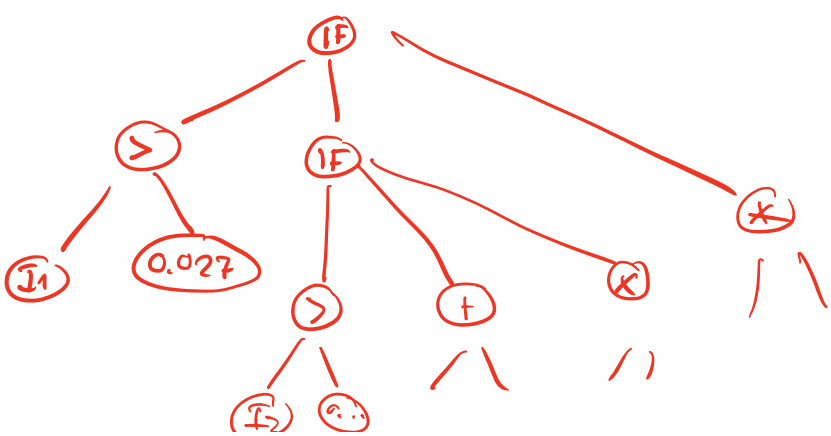
Give recommandation about when to buy or sell a currency in the foreign exchange market.

Such a model was trained using both GA and GP

- GA : to create indicators of the market : $I_1(t)$, I_2.. $I_k(t)$

  The parameters that defines these indicators where obtained by GA

- GP : the trading model was build with I_1, .. I_k as input. The output was the profit if one follows the model recommendation. Another criteria to evaluate the model is the risk : fluctuation of profit.



7.4 GP based on a sequential set of instruction (stack based)

It is a way to give a representation closer to the standard programs: a sequence of instruction executed one after the other.

Still, there is a challenge to produce a system robust to Genetic operators such as crossover and mutation.

Stack based language

Data are placed on a stack and operators take data from the stack, remove them and add the result of the operation on the stack.

- One can have instruction such as A where A is a variable: it puts the value of A on the stack.
- A program like:

  A B ADD C MUL 2 SQRT DIV

  does $\frac{(A+B)\times C}{\sqrt{2}}$ ← output

A program is a sequence of element from T

of F and the output is the values on the stack at the end of the execution (or simply the upper value of the stack)

Crossover

P1 : A B ADD | C MUL 2 SQRT DIV

P2: 1 B SUB | A A MUL ADD 3

crossover point

P1' : A B ADD A A MUL ADD 3

P2' : 1 B SUB C MUL 2 SQRT DIV

The output of P1' is $A+B+A^3$, 3 (2 values on the stack)

P2' is $(1-B)*C/\sqrt{2}$

Constraint : all inst. from F should be adq to deal with (or arguments on the stack than expected.

3 ADD → 3

Mutation replace each instruction (whether T or F) by another one.

Metaheuristics, Dec 5, 2022

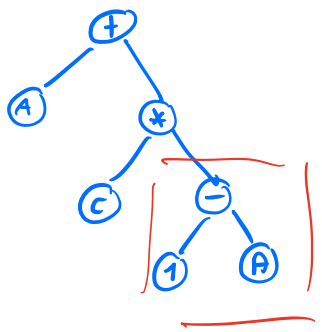
Genetic Programming GP

Goal: to evolve a population of computer programs to solve a given problem.

input $\xrightarrow{\text{prog}}$ output

The fitness is the capability of the programs to produce known output from known input (training set)

To have programs robust to crossover and mutation we can use a tree-base representation (à la Koza)



crossover: subtrees of two parents are exchanged

mutation: a subtree is modified or recreated.

We define the Function Set $F = \{+, -, *, \ldots\}$

Terminal Set $T = \{A, B, C, 1, 0 \ldots\}$

that is variables that are defined by the input values

Stack-based program representation

We have variable and operators that modifies a data stack:

"A" add its value on the stack.

"ADD" adds the top two element on the stack and replace them by their sum.

- A program is just a list of instruction and variable. They typically have a given length which can be maintained by evolution.

P1: A B | ADD C SUB

P2: 1 SQRT | B ADD SQRT

Example of the stack-based GP

Discover the relation $y = F(x_0, x_1) = (x_0 + x_1)^2$

Training set: $A = [(x_0, x_1, F(x_0, x_1)) \ldots]$ that are known.

Function set: F = ["ADD", "MUL", "SUB", "DUP"]

↑ copies the top of the stack.

Terminal set: ["X_0", "X_1"]

We consider a population of 15 programs of length 5, evolved for 100 generations.

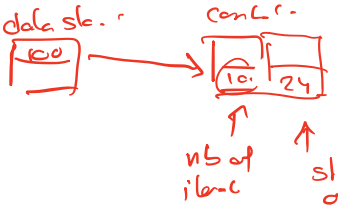
$F(x_0, x_1)$ can be represented as

P: X_0 X_1 ADD DUP MUL

LOOP 23 ADD 24 MUL 25 X1 . — ENDLOOP

Data stack

control stack : takes the data stack top value before the loop execute



ENDLOOP : decrement 100 → 99 and jump to inst 24 until number of inst = 0

8.1 Introduction

Chapter 8 Evolution Strategies

This is our last chapter of evolutionary algorithms.

Evolution strategy (ES) proposed in the 1960s by Rechenberg, Schwefel)

Initially ES was restricted to only one individual $x \in \mathbb{R}^d$ (continuous optimization)

Evolution was only through mutation

Later on, a population was introduced and crossover was added.

8.2 The (1+1)-ES algorithm

1+1 means a child which replaces the parent.

The current solution is $X(t)$ at iteration t

A child is created by mutation.

$$X(t) \in \mathbb{R}^d$$

$$x' = X(t) + \mathcal{N}(0, \sigma)$$

where $\mathcal{N}(0, \sigma)$ is a d-dimensional Gaussian distribution of variance $\underline{\sigma} = (\sigma_1 \; \sigma_2 \; \dots \; \sigma_d)$

The child replaces the parent if it is better:

$$X(t+1) = \begin{cases} X(t) & \text{if } f(x') \text{ is less good than } f(X(t)) \\ x' & \text{otherwise} \end{cases}$$

This is a random walk hill-climbing algorithm.

But the specificity of the (1+1)-ES is that σ can change over time

- If the mutation is accepted too often the size of σ is increased: the search is too local.
- If mutation is rejected too frequently σ is decreased.

Rechenberg suggest that σ should be

such that mutations are accepted with a frequency $1/5$ (computed on a window of k consecutive steps.)

8.3 Evolution strategy: a population model

The number of individual in the population is noted μ.

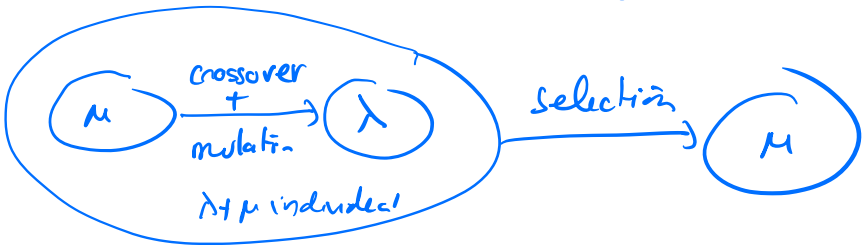
Each individual is a couple: (x^i, σ^i)

where x^i and $\sigma^i \in \mathbb{R}^d$ $\quad x_3^i$: is the third component of individual i

There are two variant of the ES with population:

$(\mu+\lambda)$-ES and (μ, λ)-ES

- $(\mu+\lambda)$-ES : in this case λ children are generated and the $\mu+\lambda$ individual are reduced to μ by selection.



- (μ, λ)-ES : in this case $\lambda \geq \mu$ are generated and μ of them are selected

In both case, selection is deterministic : take the best μ individuals out of $\mu+\lambda$ or $\lambda>\mu$.

How to generate a child :

- Choose two parents at random, uniformly and create a child by crossover (see below)
- Apply a mutation to the child as defined in (1+1)-ES

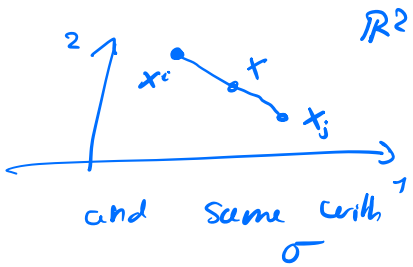
This procedure is repeated λ times to get λ children.

Crossover :

- Arithmetic crossover :

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

↑ child ↑ parent 1 ↑ parent 2



- uniform crossover :
Choose each component of x or σ from one or the other parent, at random

$$(x_e^{child}, \sigma_e^{child}) = (x_e^{(parent 1 \text{ or } parent 2)}, \sigma_e^{(parent 1 \text{ or } parent 2)})$$

Mutation :

The child obtained from crossover is subject to mutation as follows:

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

random modification

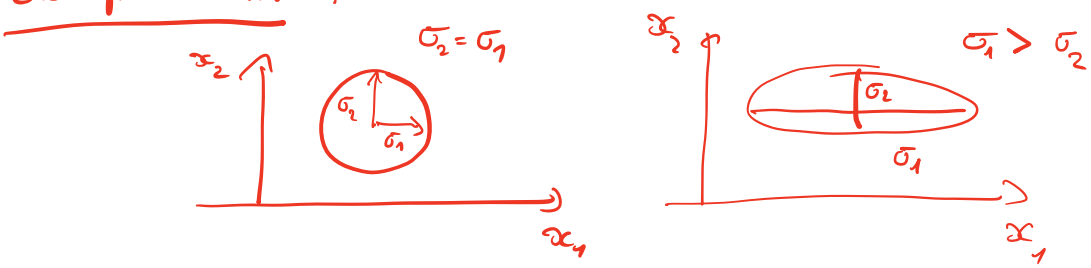
σ is also muted by a random factor.

where $\Delta\sigma$ is a parameter, typically chosen as $1/\sqrt{d}$, where d is the space dimension ($x \in \mathbb{R}^d$) plus a constant value, possibly different for each dimension.

8.4 General version

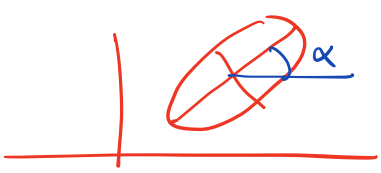
In a more general version, we want to correlate the mutation in the d-dimensions:

Example in $\mathbb{R}^2$:



General, correlated case

Mathematically, we need to have covariance matrices



Mutation in x_1 are correlated with mutation in x_2

$$\begin{cases} \sigma' = \sigma \cdot e^{N(0, \Delta\sigma)} \\ \alpha_j' = \alpha_j + (\beta \times N(0,1)) \\ x' = x + N(0, C') \end{cases}$$

where C' is a covariance matrix with

direction α_j

Chapter 9 Performance of metaheuristics

9.1 Introduction

We want to find ways to quantify the quality of a metaheuristics

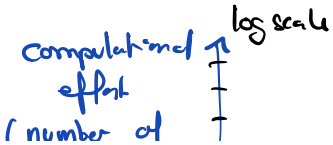
We cannot use the traditional algorithmic complexity because a metaheuristics is not guaranteed to give the right results, its execution time varies and repeating the same execution leads to a different behavior

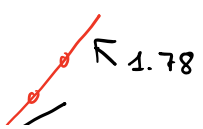
→ We need an empirical evaluation of the performance because the convergence theorems are of little use for practical problems (too much constraints)

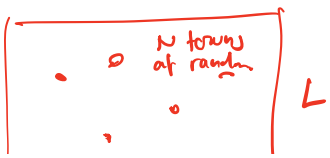
→ We need a statistical approach to face the non-deterministic nature of metaheuristics

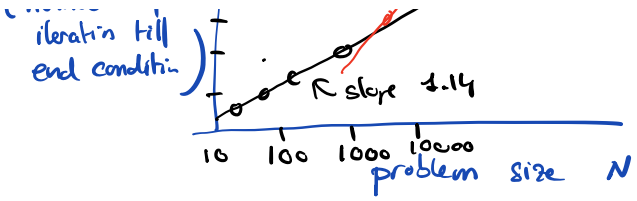
9.2 Illustration of the performance of SA for TSP

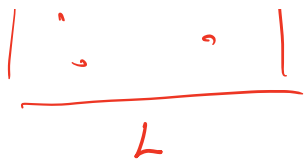
Empirically we can measure the average time to solve the TSP problem over N towns with Simulated annealing.











For $N \in \{20 \to 2000\}$ the time is 'almost' linear with N

$$T(N) = 7100 \times N^{1.14}$$

CPU time varies between 0.03 to 4.3 second on a laptop.

But no evaluation of the quality of the solution

For N in $\{5000 - 50\,000\}$

$$T(N) = 35.5 \times N^{1.78} < O(N^2)$$

CPU time between 11 and 1000 sec.

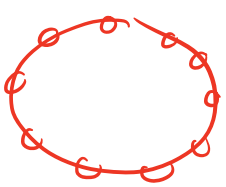
In both cases it is way less than $O(N!)$ of a naive deterministic algorithm.

We see a transition in $T(N)$ as N increases: from almost linear to almost quadratic.

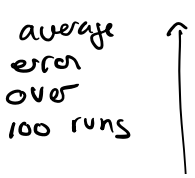
Quality of the solution

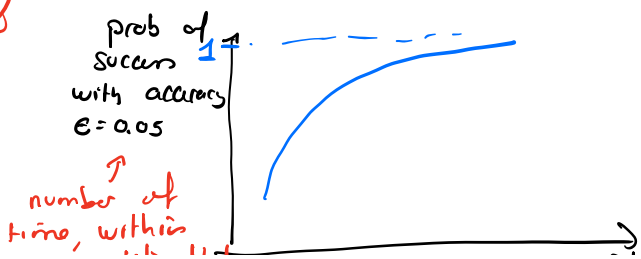
We can consider a benchmark whose optimal solution is known, and we can determine how close the SA is.

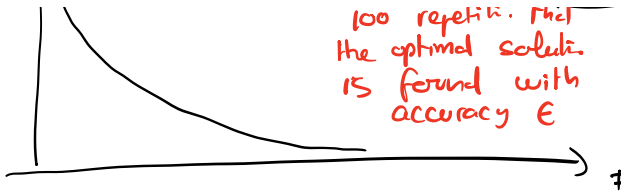
Benchmark:



$N = 50$ towns on a circle.





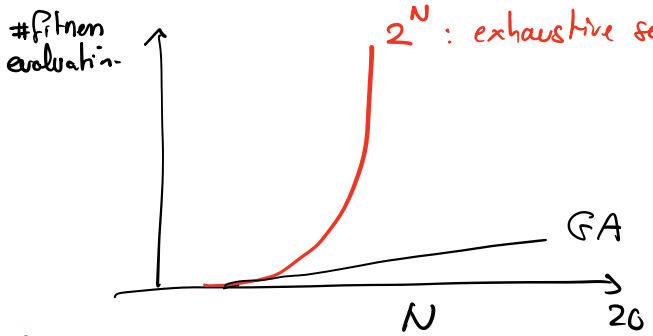


9.3 NK problems

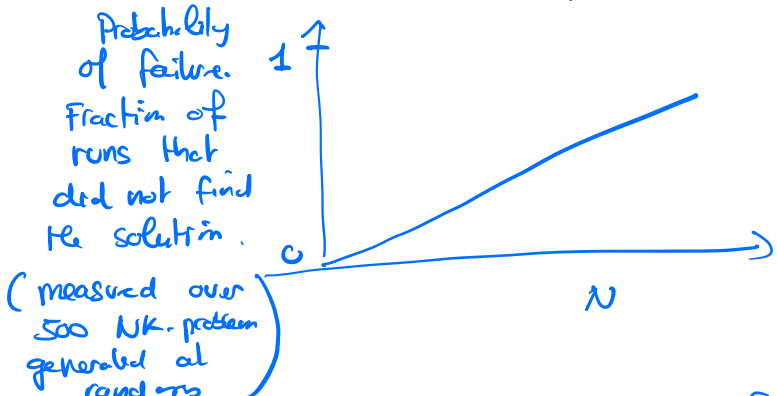
We want to discuss the speed and quality of a solution of a NK problem, solved by GA. (100 individ)

Note: here we know the solution, found by exhaustive search.

We will consider $N \in \{8, 9 \dots 20\}$ and $k = 5$



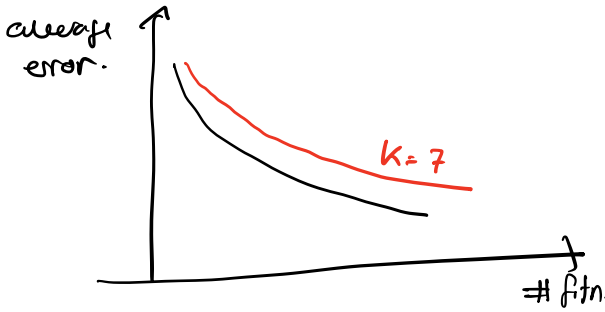
Again we have here a close to linear time to solve the problem.

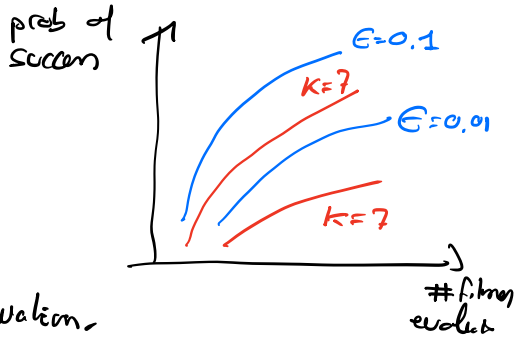


The number of problems that GA is unable to solve (even if the number of generation is increased) increases.

Some NK-problems resist to a GA solution. and this increases with N

Quality of the solutions as a function of the computational effort





We can change K: K=5 → K=7

(Stop evolution even no progress over m generations. Change m to increase the effort.

More difficult problems are more difficult to solve.

9.4 Principles of performance evaluation

- Metaheuristics are tested on benchmark problems based on real or synthetic problems. (NK problems whose difficulty can be tuned with N and K)
- These benchmark problems allow us to compare metaheuristics.

Performance metrics

- Computational effort (number of fitness evaluations for instance, independent of the hardware.)
- Quality of the solutions:
  - average error
  - probability of success: number of runs with the solution at a given accuracy divided by the total

number of runs.

- Increase of computational effort with problem size: should be less than exponential

These quantities, statistics are needed, and repetition on $\sqrt{}$ = 100 – 1000 runs are recommended

To compare two metaheuristics, one needs statistical tests (ex Kolmogorov-Smirnov) to decided if the quality of one is relevant with respect to the other.

Robustness is a quality of a metaheuristics: it works well on many problems, rather than giving a perfect response to one problem only.

9.5 "No Free lunch" theorem (NFL)

If a Metaheuristics A performs better than B on a given class of problem, can we conclude that A is always better?

The NFL theorem says no in an objective way. NFL was proposed in 1997 by Wolpert and Mc Ready

No metaheuristics can be better than any other over all possible problems

If metaheuristics A is better than B for some problems, there are other problems for which B will be better than A.

Hypotheses of NFL

We consider a finite search space S of size $|S|$

- We consider fitness function $f: S \to Y \subseteq \mathbb{R}$ with Y a finite set of size $|Y|$.
- All possible problems are specified by a fitness function. There are $|Y|^{|S|}$ different fitness with S and Y finite.
- We will consider a given computational effort m, that translates to a trajectory in S, $d_m = \{(x_1, f(x_1)), (x_2, f(x_2)), \ldots, (x_m, f(x_m))\}$

NFL Theorem:

Let $P(d_m | f, m, A)$ the probability that the sequence d_m contains the optimal value of f, using metaheuristic A to create the sequence d_m

Then:

$$\sum_f P(d_m | f, m, A_1) = \sum_f P(d_m | f, m, A_2)$$

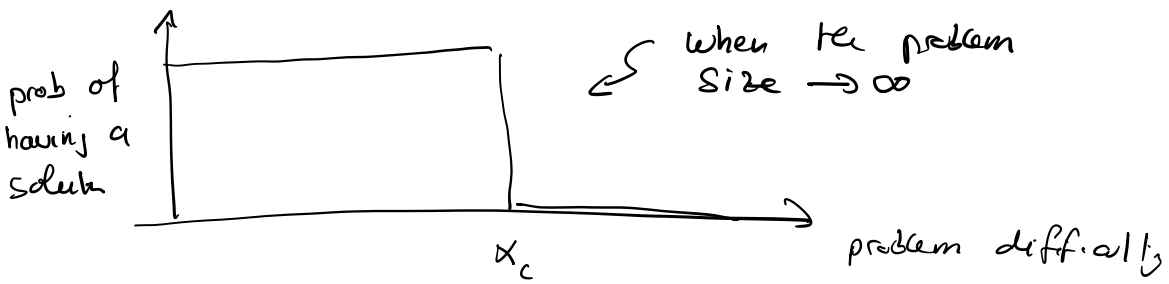
When considering all possible problems f, A_1 has no advantage over A_2

Chapter 10 : Phase transition in optimization problems

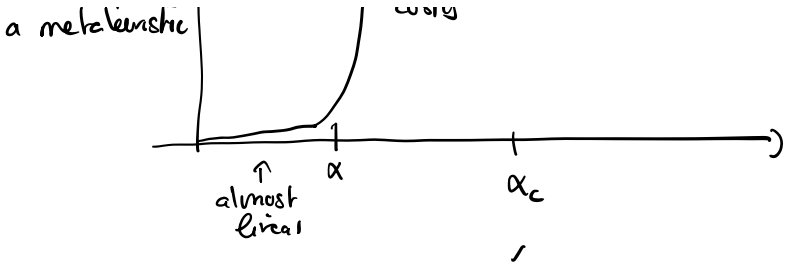
10.1 Introduction :

- We will study the behavior of a metaheuristic as a function of the problem difficulty.
- The type of problems are known as satisfaction problems.
- In some case, we can analyze theoretically these problems and compute the probability that they have a solution

we will see that, as the difficulty increases, the probability to have a solution goes abruptly from 1 to 0. (Phase transition) a brutal change of property)



resolution time of ↑ | exponentially



10.2 The SAT problem

A SAT problem is a set of M (Boolean) equations for N Boolean variables.

- The goal is to find the value of these N variables so that the M equations are satisfied

- Alternatively, we want to find an assignment of the N variables to <u>minimize</u> the number of UNSAT equations.

- We define Energy E, the number of UNSAT equations. A problem is said to be SAT if $E = 0$ is possible

XORSAT Problems

we consider only Boolean equations with XOR operation. XOR = + mod 2

Example:

$$\begin{cases} x_1 + x_2 + x_3 = 1 \\ x_2 + x_4 = 0 \\ x_1 + x_4 = 1 \end{cases} \quad \Big| \quad \begin{matrix} N = 4 \\ M = 3 \end{matrix}$$

The solutions are:

$$(x_1\ x_2\ x_3\ x_4) = \begin{cases} (1\ 0\ 0\ 0) \\ (0\ 1\ 0\ 1) \end{cases} \Rightarrow \text{this is SAT}$$

Example of an UNSAT problem:

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

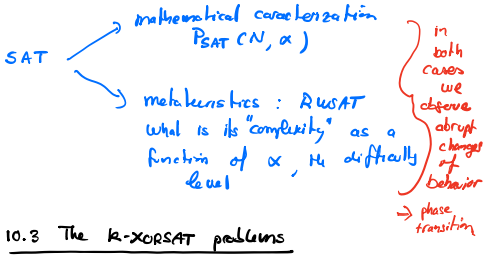
Here the min Energy is $E = 1$

How to solve a XOR SAT problem ?

Since we only use XOR, we can use Gaussian elimination in arithmetic mod 2. It takes $O(N^3)$ time for $N = M$.

Chapter 10 Phase transition in optimization problems.

Goal : to study the behavior of a problem as a function of its difficulty



10.3 The k-XORSAT problems

We have M Boolean equations, N variables, each equ contains exactly k different variables.

$$x_1 + x_{17} + x_{29} = 0$$
$$x_1 + x_2 + x_{15} = 1$$

$k = 3$, $N = 29$, $M = 2$

We want to study the satisfyability of such systems in a statistical way :

We generate at random a system with N, M and k given and see whether it has a solution.

This gives us a way to compute

$$P_{sat}(N, M, k)$$

as the fraction of $\mathcal{N} = 10'000$ systems that are satisfiable. Using Gaussian elimination, this

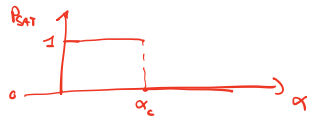
can be done numerically even for N and M large.

A measure of difficulty will be

$$\alpha = \frac{M}{N}$$

Many constraints (M large) makes the problem difficult, few degree of freedom (N small), too.

What is observed is that, if $N \to \infty$, $k \geq 2$ $P_{SAT}(N, \alpha)$ has a phase transition for $\alpha = \alpha_c$



Statistical methods to generate instances of k-XOR-SAT problems

N variables, M equations $\alpha = \frac{N}{N}$, k given

We need to pick k different indices out of N to get a first equation:

$$x_{i_1} + x_{i_2} + \dots x_{i_k} = \nu$$

and $\nu \in \{0, 1\}$ is also selected randomly.

This is repeated M times.

Possibly, the same equation can appear more than once.

Another method is to generate all possible set of k indices out of N. There are

$$\binom{N}{k}$$

For each of these possibility there are 2 equations with $\nu = 0$ or $\nu = 1$

We go through all these possibilities and select each of them with a probability so that in the end we have M equations (actually on average)

$$P = M/H \qquad H = 2\binom{N}{k}$$

10.4 The space of solution

For $k = 1, 2$ and 3, some analytical results can be obtained for $P_{SAT}(N, \alpha, k)$

They may be difficult to obtain an we will just consider the case $k = 1$

A 1-XORSAT problem is

$$\begin{cases} x_{i_1} = \nu_1 \\ x_{i_2} = \nu_2 \\ \vdots \\ x_{i_M} = \nu_M \end{cases}$$

Such a system is satisfiable if not the same variable appear more than once, or if it does, ν_i should be the same.

$P_{SAT} > P$ where P is the prob to never choose twice the same variable.

$$P = 1 \times \left(1 - \frac{1}{N}\right) \times \left(1 - \frac{2}{N}\right) \times \dots \times \left(1 - \frac{M-1}{N}\right) = \prod_{i=0}^{M-1} \left(1 - \frac{i}{N}\right)$$

(one variable is now excluded)

↑ no restriction to choose the first variable

↑ 2 variables are excluded

To estimate P, let us take the log:

$$\log P = \sum_{i=0}^{M-1} \log\left(1 - \frac{i}{N}\right)$$

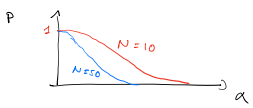
with $\log(1-x) = -x + \frac{x^2}{2} - \frac{x^3}{3} + \dots$

we have $\log P = -\sum_{i=0}^{N-1} \frac{i}{N} + \sum_{i=0}^{N-1} \frac{1}{2}\left(\frac{i}{N}\right)^2 + \dots$

for M/N small enough, we can neglect the higher orders

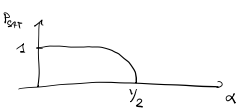
$$\log P = -\underbrace{\frac{M(M-1)}{2N}}_{\sum \frac{i}{N}} + O\left(\frac{M^3}{N^2}\right)$$

$$P_{SAT} > P = \exp\left(-\frac{M(M-1)}{2N}\right) = \exp\left(-\frac{\alpha^2 N}{2}\right) \qquad \alpha = \frac{M}{N}$$



So here, as $N \to \infty$, P goes to zero for all α
Not a very interesting case.

Case of 2-XORSAT

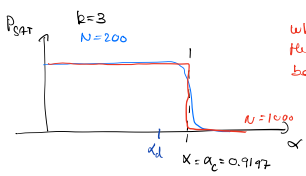


$k=2$: not really an abrupt change. even though for $\alpha > 1/2$, no more solution exist for a 2-XORSAT problem. (if $N \to \infty$)

Case of the 3-XORSAT

Again we will skip the detailed mathematical derivation.

We obtain in this case a true phase transition:



When $N \to \infty$ the jump at $\alpha = \alpha_c$ becomes more and more sharp

There is actually something more about 3-XORSAT problems. It is the distribution of solution in the search space.

What has been found is:

- for $\alpha < \alpha_d = 0.8184$, solutions are distributed uniformly.
- for $\alpha_d < \alpha < \alpha_c$, there are solutions but the appear as separated clusters in the space.
- for $\alpha \geq \alpha_c$: no more solution !

So there is also a phase transition about the way solutions populate the space.

10.5 Behavior of a metaheuristics for the SAT problem.

We will actually consider only the 3-XORSAT problem in what follows.

Our goal is to see how a metaheuristics behaves as a function of the problem difficulty, α.

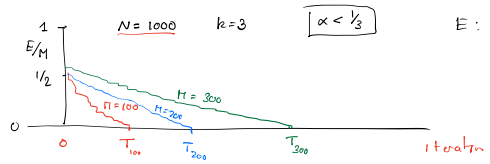
The RWSAT algorithm

RWSAT: Random Walk SAT

1. We initialize the N variables at random
2. We compute the Energy the number of equations not satisfied
3. If E==0, end
3b If number of iteration ≥ max stop and declare UNSAT
4. Else : pick at random one of the UNSAT equations and flip the value of one of its k variable. That makes this equation satisfied, because it is a XORSAT problem. But this may make other equations that were satisfied before, unsatisfied.
5 Goto 2

Behavior α small

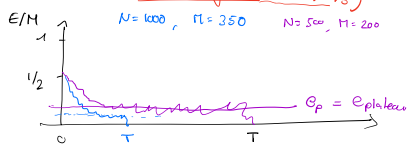
M : number of equations
E : number of unsatisfied equations



For $\alpha < 1/3$, RWSAT finds a solution in a time T that is linear with N

$T \propto N \times t_{res}(\alpha)$ where $t_{res}(\alpha)$ is a coefficient which increases with α

α large ($\alpha \geq 1/3$)



For $\alpha > 1/3$, after an initial drop, E/M oscillates around a plateau value. Only random fluctuations can bring E to 0. The fluctuations are smaller as N increases.

So there is a value $\alpha_E = 1/3$

If $\alpha \geq 1/3$ then

$$T = \exp(N \cdot \tau_{res}(\alpha))$$

T : average time to solution
$\tau_{res}(\alpha)$: coeff that grow with α

So, we see a phase transition at $\alpha = \alpha_E = 1/3$ for RWSAT, from linear complexity to exponential complexity.

Case of backtracking

See slides.

L'algorithme RWSAT

Pour exprimer cet algorithme en pseudo-code on définit une grandeur E qui est le nombre d'équation non-satisfaites. Le but est alors de minimiser la valeur de E.

```
initialize all variables at random
compute E
t=0
while(E>0 and t<max)
    choose at random a non-satisfied equation
    chose at random one of its three variables
    change the value of that variable to its complement
    compute E
    t=t+1
end while
print E, t
```

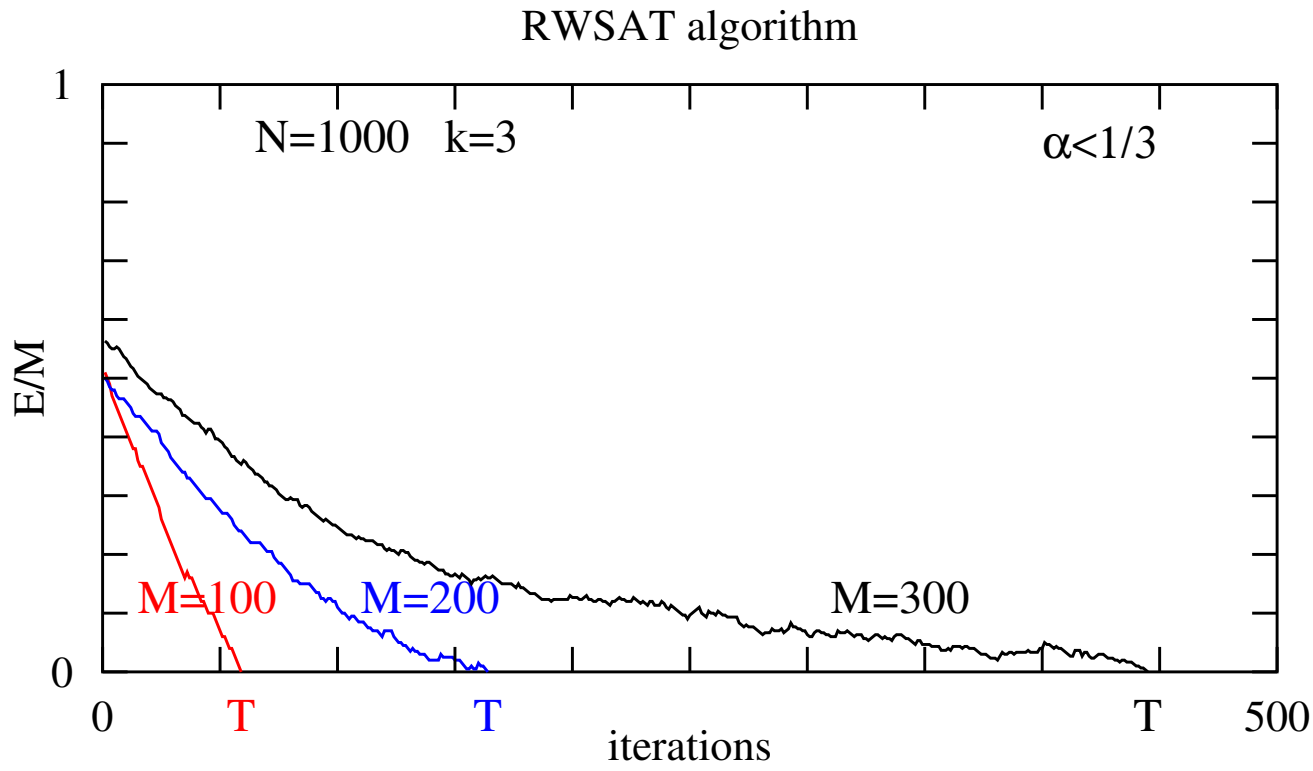
Probabilité que l'algorithme se trompe

- combien de temps faut-il laisser tourner la boucle pour conclure que le problème est UNSAT (non-satisfiable) ?
- Rien ne garantit a priori que, si la solution existe, cet algorithme va la trouver en un temps fini

On peut démontrer que si on fait T répétitions de RWSAT avec chaque fois $\texttt{max}= 3N$, la probabilité que le problème soit SAT et qu'aucune solution n'ai été trouvée durant les T répétitions est

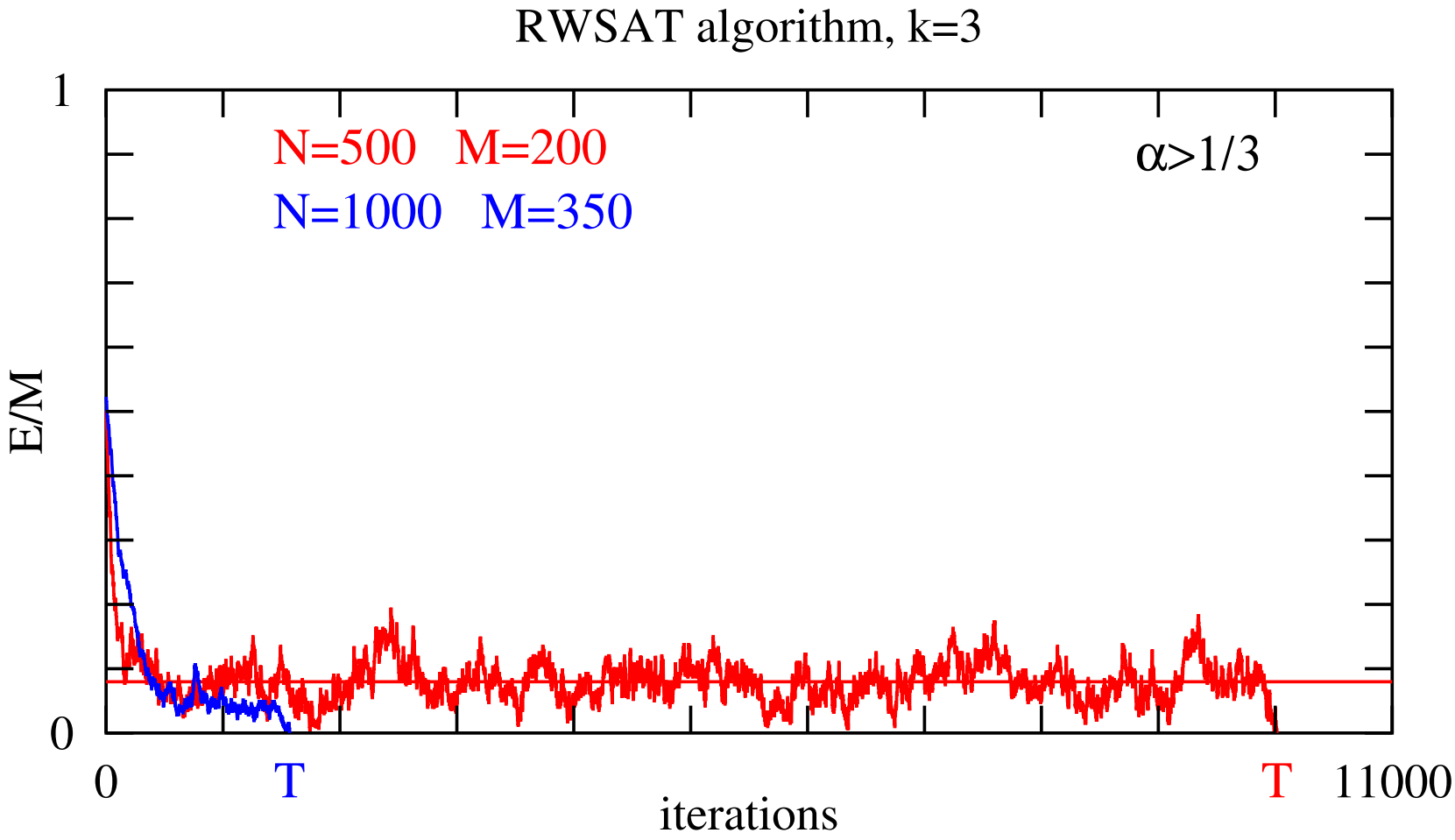
$$P_{SAT} \leq e^{-T\left(\frac{3}{4}\right)^N} \tag{80}$$

Comportement : α petit



E décroit progressivement et atteint la valeur 0. L'algorithme RWSAT donne une solution.

Comportement : α grand



E décroit mais $e = E/M$ oscille autour d'une valeur plateau $e_{plateau} > 0$. L'amplitude des fluctuations est d'autant plus petite que N grandit. C'est grâce à ces fluctuations qu'on peut espérer avoir $E = 0$.

Résumé du Comportement

- Si $\alpha < \alpha_E = 0.33$, alors $E_p = 0$ et on observe un temps de résolution qui croît linéairement avec la taille N du problème :

$$< T_{res} >= N t_{res}$$

où t_{res} est un facteur qui augmente avec α.

- Si $\alpha_E \leq \alpha < \alpha_c$, alors $E_p > 0$ et le temps de résolution devient exponentiel avec N :

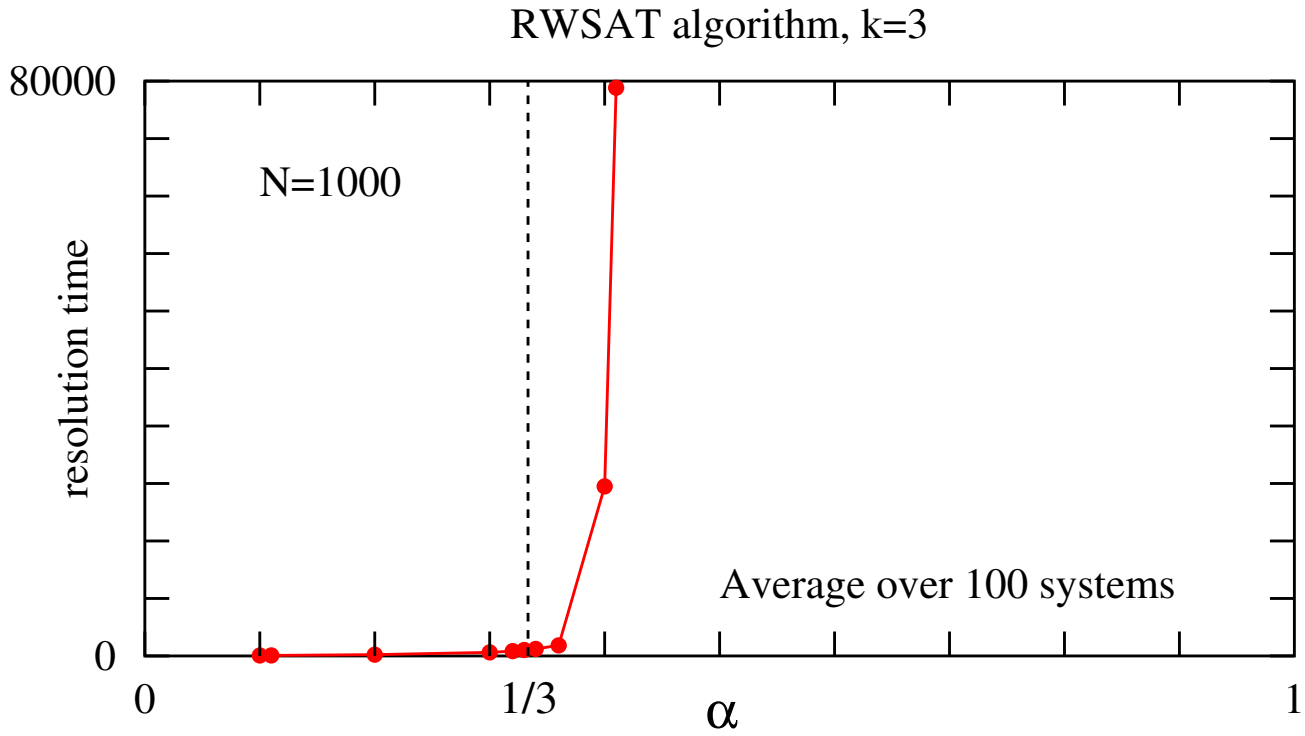
$$< T_{res} >= \exp(N \tau_{res})$$

où τ_{res} est un facteur qui croît avec α.

Il y a donc une transition de phase dans la complexité algorithmique

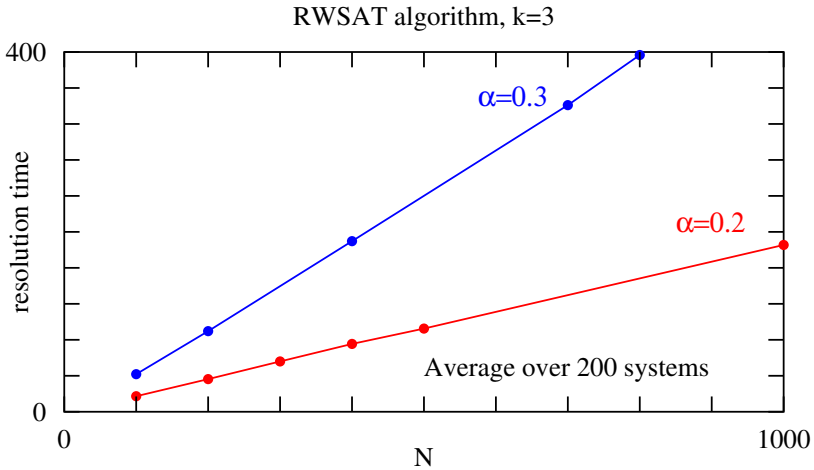
Résumé du Comportement

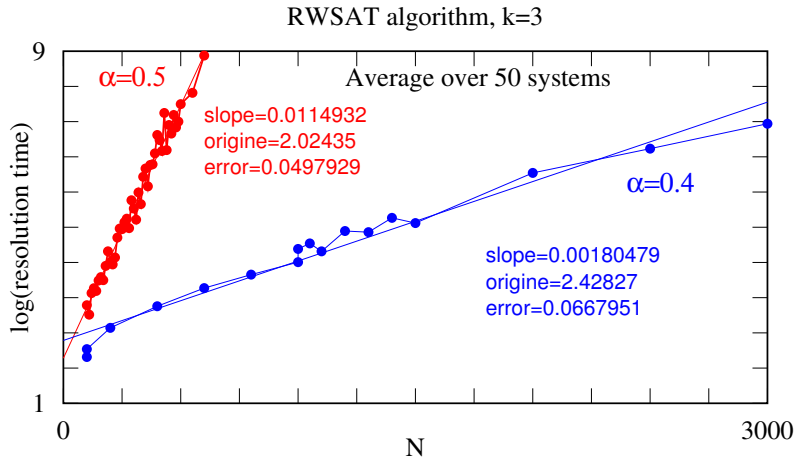
Transition de phase de la complexité algorithmique



Comportement du RWSAT pour un problème 3-XORSAT, en fonction de α, pour $N = 1000$ fixé. On constate que le temps moyen de résolution, moyenné ici sur 100 problèmes aléatoires, change brusquement quand $\alpha > 1/3$.

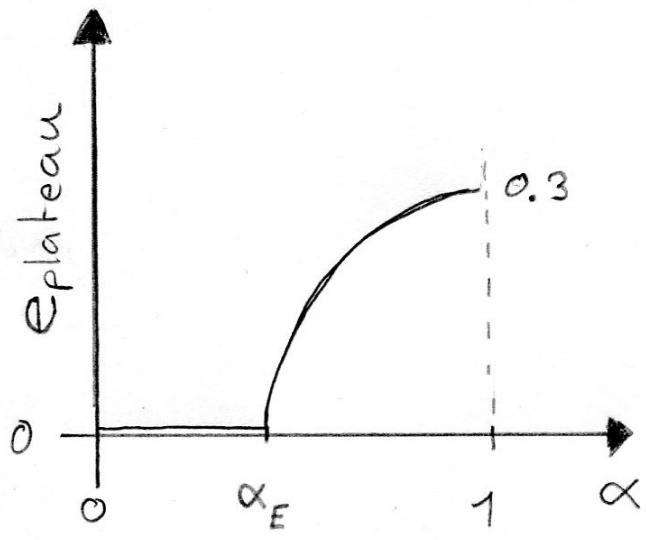
Complexité linéaire et exponentielle

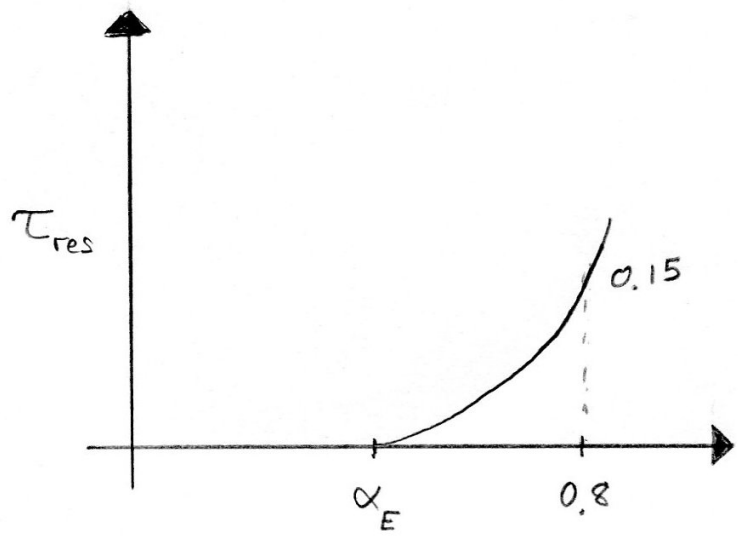




Temps moyen de résolution du RWSAT. Pour faire ressortir le comportement exponentiel dans l'image de droite, on a ici représenté le $\log_{10}\langle T_{res}\rangle$, en fonction de N, ce qui donne une droite dans cette échelle semi-logarithmique.

Dépendance de $e_{plateau}$ et τ_{res} versus α





algorithme du Backtracking

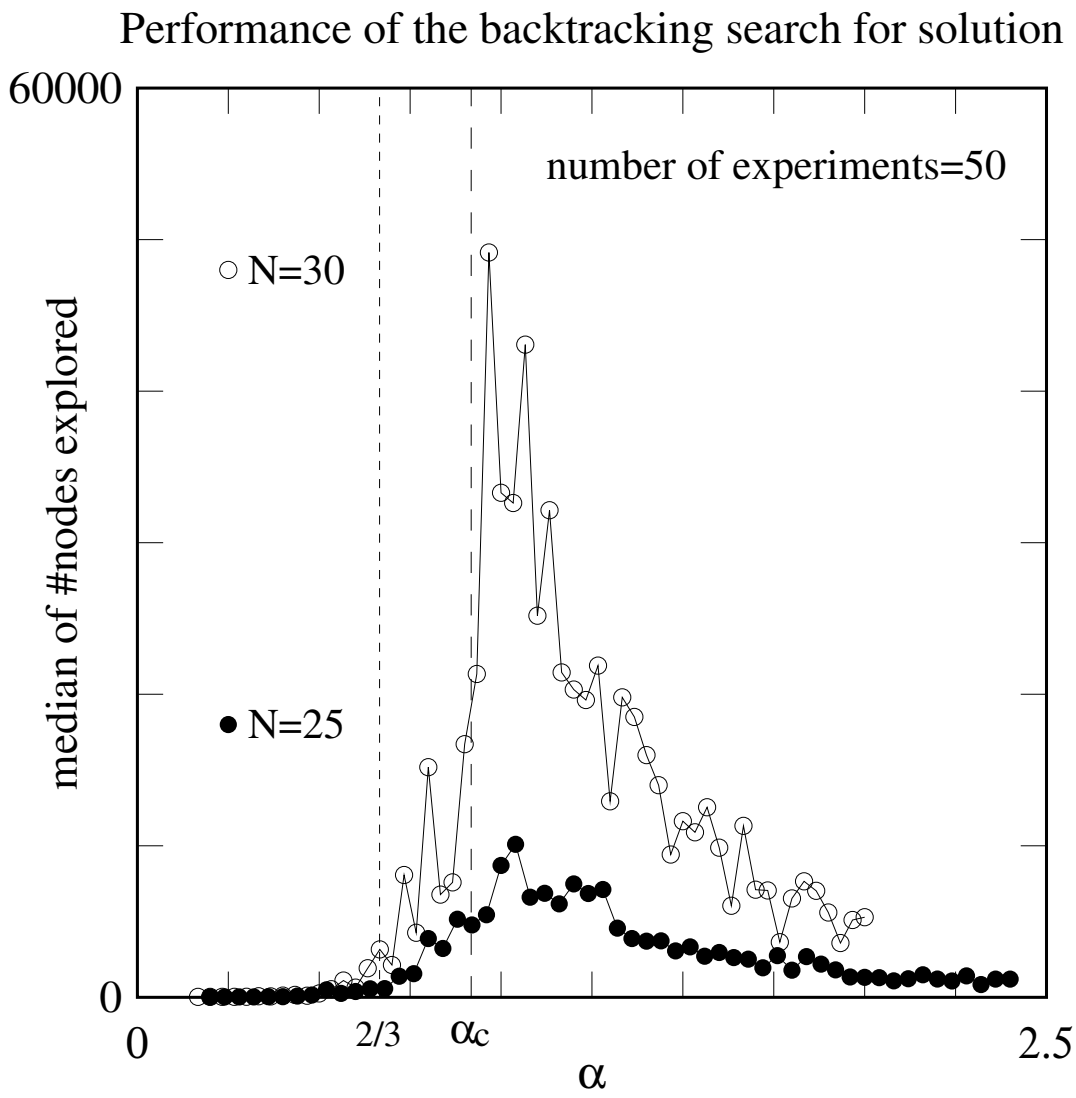
- La difficulté de l'algo RWSAT est la raréfaction des solutions dans l'espace de recherche.
- On peut s'en rendre compte en regardant le comportement d'un algo de backtracking :
- Les variables x_i sont assignées progressivement, selon un arbre binaire que l'on parcourt en profondeur.
- à chaque nouvelle assignation, on vérifie qu'il n'y a pas de contradiction avec les équations dont les variables sont déjà connues.
- S'il y a contradiction, on change la dernière valeur assignée (backtracking)

Performance du Backtracking

- le temps de calcul est le nombre de noeuds de l'arbre de recherche que le backtracking doit générer avant de trouver soit une solution complète (assignation de tous les x_i)
- On observe à nouveau un comportement à seuil (transition de phase) :
- Au delà d'une valeur critique α_E (dont la valeur n'est pas la même que celle du RWSAT), le backtracking met un temps exponentiel à résoudre le problème, alors que si $\alpha < \alpha_E$, ce temps est linéaire

$$< T_{res} >= \begin{cases} Nt & \text{si } \alpha < \alpha_E \\ \exp N\tau & \text{si } \alpha > \alpha_E \end{cases} \tag{81}$$

Graphe des Performances



Si $\alpha > \alpha_c$, le temps du backtracking diminue car, dans cette région, plus α est grand, plus l'algorithme se rend vite compte qu'il n'y a pas de solutions

Valeurs critiques

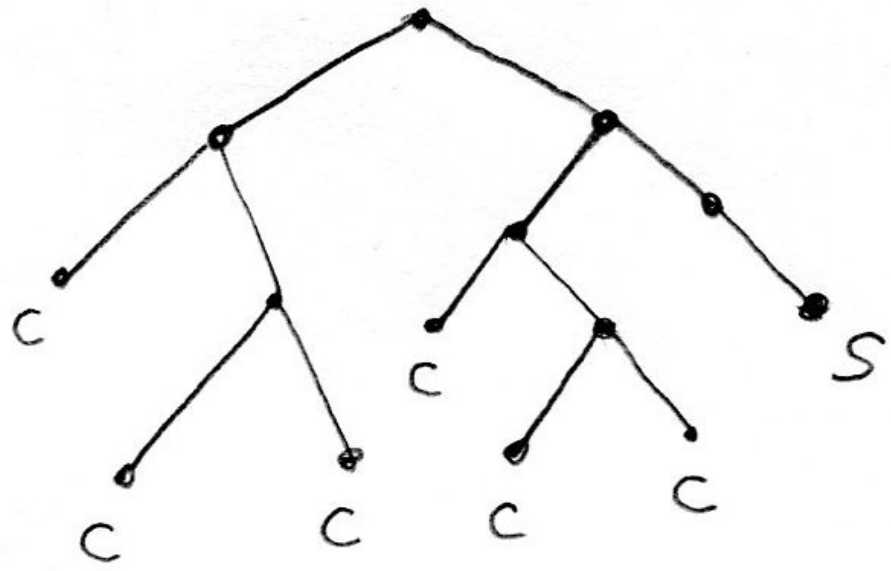
- La valeur de α_E dépend de la façon dont on assigne, à chaque niveau de l'arbre, les variables x_i dans l'algorithme du backtracking
- Si cette assignation se fait au hasard (on parcourt les indices i dans un ordre aléatoire), on trouve que

$$\alpha_E = 2/3$$

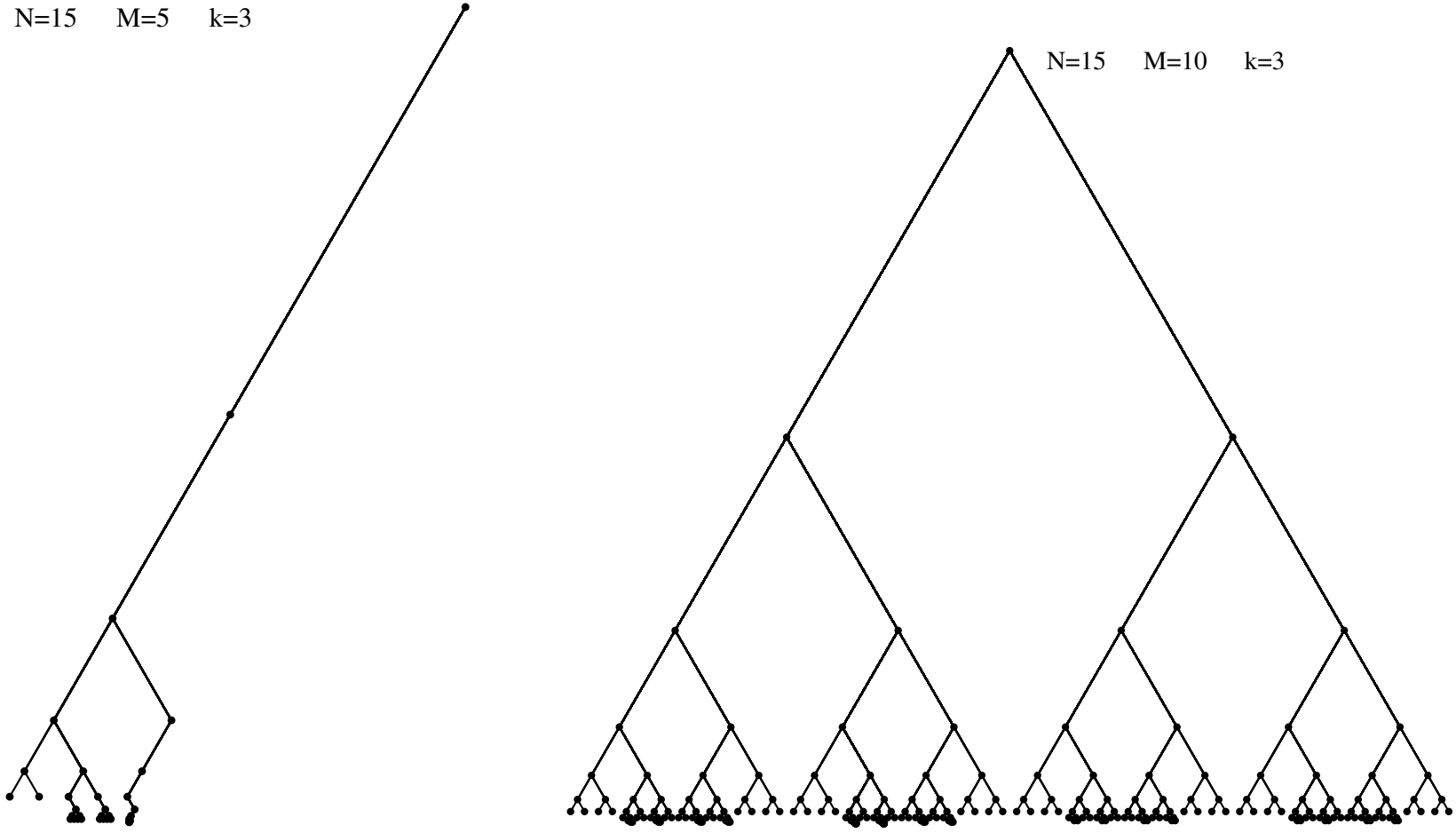
- si on assigne d'abord les variables qui apparaissent dans les équations qui contiennent encore deux variables libres, alors on trouve

$$\alpha_E = 0.7507$$

Parcours de l'espace de recherche



Parcours de l'espace de recherche



Parcours typique du backtracking dans l'arbre de recherche en fonction de la valeur de α. A gauche, petite valeur $\alpha = 1/3$ et à droite grande valeur, $\alpha = 2/3$. Il y a une transition de phase entre un régime où la solution est trouvée avec une exploration succincte et un régime où il faut explorer une partie très importante de l'arbre.