Stochastic Optimization

Simulation and Optimization

Edoardo Fadda
Dip. di Scienze Matematiche – Politecnico di Torino
e-mail: edoardo.fadda@polito.it

Version: October 22, 2024

Definitions

System: the facility or process of interest.

To study it, we often have to make a set of assumptions about how it works. These assumptions are leveraged to build a *model*.

If the model is simple enough, it may be possible to use mathematical methods to obtain exact information on questions of interest (referred as *analytic* solution).

However, most real-world systems are too complex to allow realistic models to be evaluated analytically, and these models must be studied using *simulation*.

In a *simulation* we use a computer to evaluate a model *numerically*.

Example: Consider a manufacturing company evaluating to build a large plant but is unsure if the potential productivity gain would justify the construction cost.

State of a system: collection of variables necessary to describe a system at a particular time.

A *discrete system* is a system having state variables that change instantaneously at separated points in time (e.g. queue).

A *continuous system* is a system having state variables that change continuously with respect to time (e.g. airplane).

A system that we will use as a playgroud is a M/M/1 queue, where we have:

- *Arrivals* occur at rate λ according to a Poisson process.
- *Service times* have an exponential distribution with rate parameter μ ($\frac{1}{\mu}$ is the mean service time).
- Arrival times and services times are independent.
- a single server serves customers one at a time from the front of the queue, according to a first-come, first-served discipline.
- When the service is complete the customer leaves the queue and the number of customers in the system reduces by one.
- The buffer is of infinite size, so there is no limit on the number of customers it can contain.

We call waiting times the time that a customer spent in the queue plus the service time.

Let us see `2_MM1_simulation`...

Usually, simulation data are non stationary and correlated therefore the results of the CLT cannot be applied directly.

Let $Y_1, Y_2, \ldots$ be an output stochastic process (e.g., the profit of a system over time).

Let $y_1^1, y_2^1, \ldots, y_m^1$ be a realization resulting from making a simulation run of length m observations using the random numbers $u_1^1, u_2^1, \ldots u_m^1$. If we run the simulation with a different set of random numbers $u_1^2, u_2^2, \ldots, u_m^2$ then we will obtain a different realization $y_1^2, y_2^2, \ldots, y_m^2$.

In general, given n repetitions, we have:

$$\begin{array}{ccccc} y_1^1 & \ldots & y_i^1 & \ldots & y_m^1 \\ y_1^2 & \ldots & y_i^2 & \ldots & y_m^2 \\ \ldots & \ldots & \ldots & \ldots & \ldots \\ y_1^n & \ldots & y_i^n & \ldots & y_m^n \end{array}$$

The observations in each row are not iid, but they are with respect to the columns.

The *independence across runs* is the key for what we are going to discuss. For example:

- An unbiased estimator of $\mathbb{E}[Y_i]$ is

$$\bar{y}_i(n) = \frac{\sum_{j=1}^{n} y_i^j}{n}.$$

- we can build confidence interval of the average of $\mathbb{E}[Y_i]$ using the sample mean $\bar{y}_i(n)$ and standard deviation $\bar{\sigma}_i(n)$ as

$$[\bar{y}_i(n) - z_\alpha \frac{\bar{\sigma}_i(n)}{n}, \bar{y}_i(n) + z_\alpha \frac{\bar{\sigma}_i(n)}{n}]$$

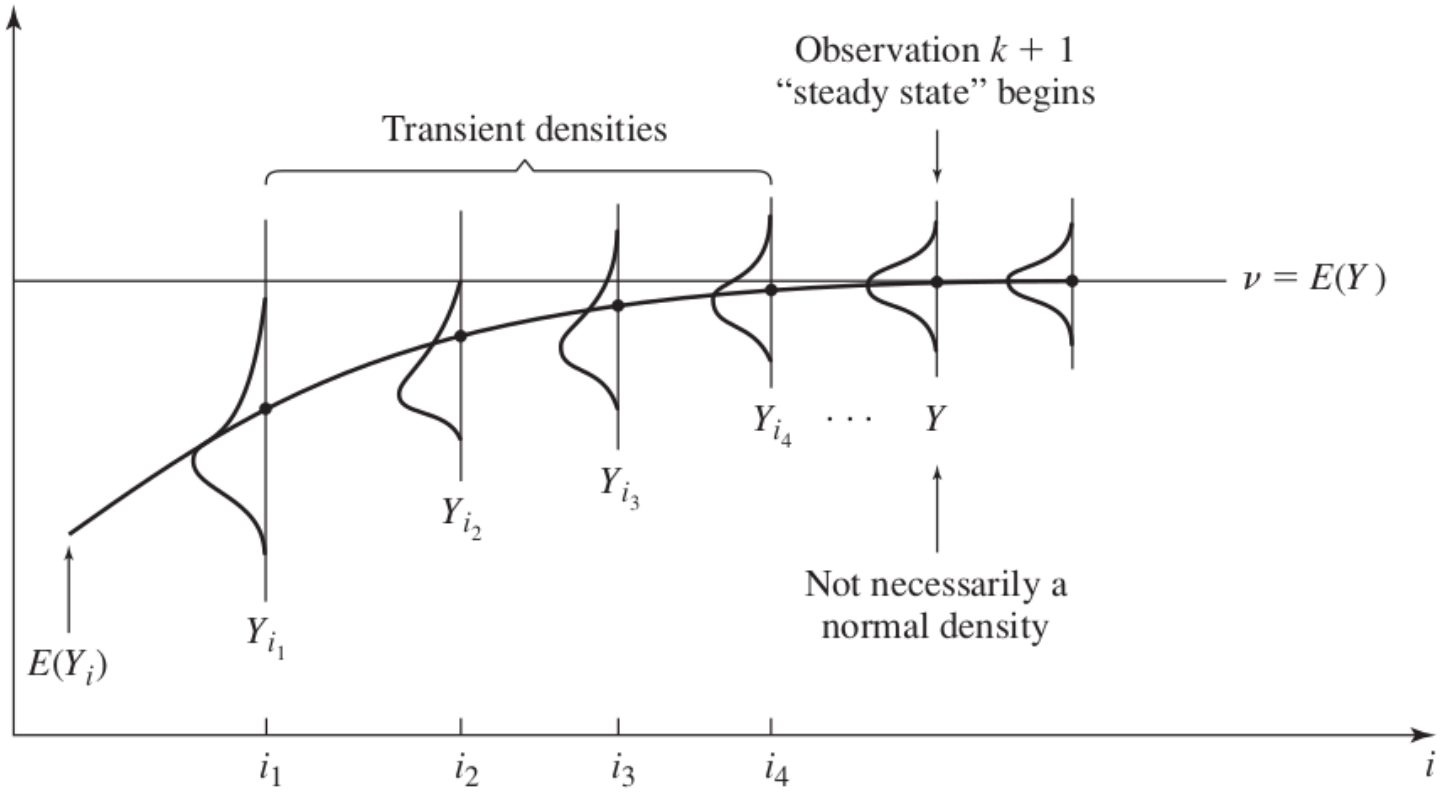
In general we have different types of simulation:

- Terminating simulation;
- Non terminating simulation:
  - Steady state parameters
  - Steady state cycle parameters

Transient and Steady State Behavior

We call $F_i(y|I)$ the *transient distribution* of the output process at time i for initial condition I.

If $F_i(y|I) \to F(y)$ as $i \to +\infty$ for every possible initial condition I, then $F(y)$ is called the *steady-state distribution* of the process $Y_1, Y_2, \ldots$.



In practice, there will be a point in time, say k, such that the distribution from this point on will be approximately the same as the steady state one.

N.B. The steady-state distribution (if it exists) does not depend on the initial condition I, but the rate of convergence do so (see `2_MM1_simulation`).

Statistical Analysis For Terminating simulations

Suppose we make n independent replications of a terminating simulation, where each replication terminates by the event E and is begun with the same initial condition. E.g. we want to simulate the queue of a given working day.

Independence of replications is accomplished by using different random numbers for each replications.

$$\begin{array}{ccc} Y_1^1 & \ldots\ldots & Y_m^1 \\ \ldots & \ldots\ldots & \ldots \\ Y_1^n & \ldots\ldots & Y_m^n \end{array} \tag{1}$$

Each column is a set of i.i.d. random variables, therefore we can apply standard statistics (e.g., on the mean).

Statistical Analysis For steady State parameters

Let $Y_1, Y_2, \ldots$ be an output stochastic process from a single run of a nonterminating simulation and suppose that

$$P(Y_i \leq y) = F_i(y) \to F(y) = P(Y \leq y) \qquad \text{as} \qquad i \to \infty$$

where Y is the steady-state random variable of interest with distribution function F (We have suppressed in our notation the dependence of F_i on the initial conditions I).

We call ϕ the steady-state parameter of interest (e.g., $E(Y)$, $P(Y \leq y)$, etc.).

One difficulty in estimating ϕ is that the distribution function of Y_i is different from F, since it will generally not be possible to choose I to be representative of *steady-state behavior*.

For example, the sample mean $\overline{Y}_m = \sum_{i=1}^{M} y_i$ will be a biased estimator of $\nu = E(Y)$ for all finite values of m. The problem we have just described is called the problem of the *initial transient* or the *startup problem*.

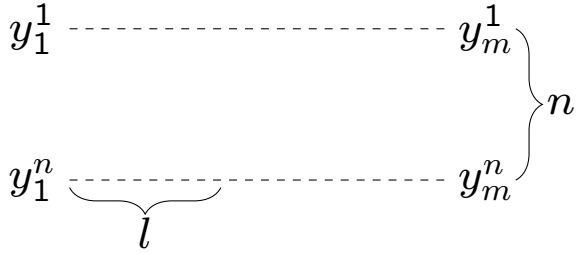
For the sake of simplicity, let us consider the estimation of

$$\mu = \lim_{i \to +\infty} \mathbb{E}[Y_i].$$

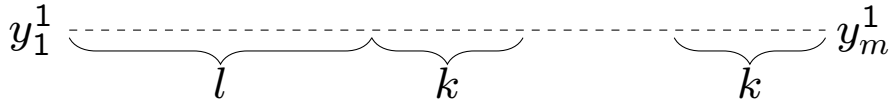
The techniques most often used is called *warming up the model* or *initial-data deletion*. The idea is to delete the initial data and estimate ν using:

$$\bar{Y}(m,l) = \frac{\sum_{i=l+1}^{m} Y_i}{m-l}.$$

To compute confidence interval, we do n repetitions of the process $Y_1, Y_2, \ldots$, delete the first l observation for all the replicas. Let $\bar{Y}_j(m,l)$ be the sample mean of the $m-l$ observation in replication j. Then, using the $\bar{Y}_j(m,l)$ we be view as independent, therefore we can build a confidence interval.



Another possibility can be to create one long trajectory $Y_1, \ldots, Y_m$, we can split it in batches of k:

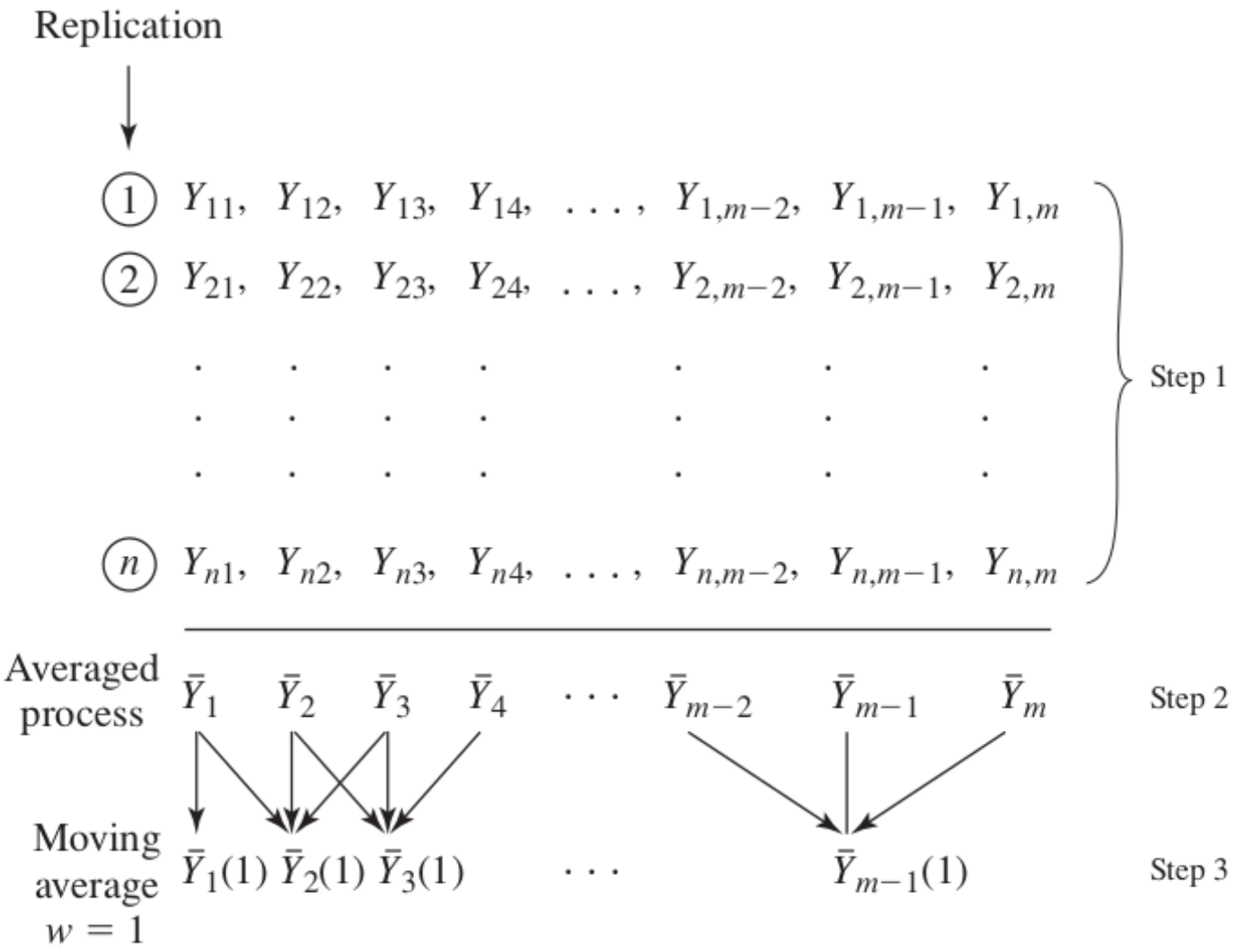


N.B.: In order for this method to work effectively, the batches must be made so that the values of $\bar{Y}_j(k)$ are approximately uncorrelated (see ACF).

How to decide the *warmup period* l? $\iff$ how to decide when $\mathbb{E}[Y_i]$ "flattens out"?

- If l is small, probably $\mathbb{E}[\bar{Y}(m,l)] \neq \nu$.
- If l is big $\bar{Y}(m,l)$ will have a big variance (due to a reduced number of observations).

A simple techniques to define l is the *Welch's Algorithm*.



Plot $\bar{Y}_i(w)$ and choose l to that value of i beyond which $\bar{Y}_i(w), \bar{Y}_{i+1}(w)$ appears to have converged.

Variance Reduction

Sometimes the cost of even a modest statistical analysis of the output can be so high that the precision of the results, perhaps measured by confidence-interval width, will be unacceptably poor.

You should therefore try to use any means possible to increase the simulation's efficiency.

We focus on statistical efficiency, as measured by the variances of the output random variables from a simulation.

Common Random Numbers

Common Random Numbers (CRN) are used when we are comparing two or more alternative configurations (e.g., two different replenishment rule).

The basic idea is that we should compare the alternative configurations "under similar experimental conditions" so that we can be more confident that any observed differences in performance are due to differences in the system configurations rather than luck.

To see the rationale, let us consider two alternative configurations and let $X_j^{(1)}$ and $X_j^{(2)}$ be the observations. We want to estimate

$$\zeta = \mu_1 - \mu_2 = \mathbb{E}[X_j^{(1)}] - \mathbb{E}[X_j^{(2)}].$$

If we make n replications of each system and let $Z_j = X_j^{(1)} - X_j^{(2)}$ for $j = 1, 2, \ldots, n$, then $\mathbb{E}[Z_j] = \zeta$ so

$$\bar{Z}(n) = \frac{1}{n} \sum_{j=1}^{n} Z_j$$

is an unbiased estimator of ζ. Since the Z_j's are IID random variables,

$$\mathrm{Var}[\bar{Z}(n)] = \frac{\mathrm{Var}(Z_j)}{n} = \frac{\mathrm{Var}(X_j^{(1)}) + \mathrm{Var}(X_j^{(2)}) - 2\,\mathrm{Cov}(X_j^{(1)}, X_j^{(2)})}{n}$$

If the simulations of the two different configurations are done independently

$$\text{Cov}(X_j^{(1)}, X_j^{(2)}) = 0.$$

On the other hand, if we could somehow do the simulations of configurations 1 and 2 so that $\text{Cov}(X_{1j}, X_{2j}) > 0$, $\text{Var}[\bar{Z}(n)]$ is reduced.

CRN is a technique where we try to induce this positive correlation by using the same random numbers to simulate both configurations. What makes this possible is the deterministic, reproducible nature of random-number generators.

N.B.:

- Some important characteristic to ensure is synchronization;
- In general experiments are needed since there is no general theory about this method.

Antithetic Variates

Antithetic Variates is a techniques when we simulate a **single** system.

Suppose that we make n pairs of runs of the simulation resulting in observations

$$(X_1^{(1)}, X_1^{(2)}), \ldots, (X_n^{(1)}, X_n^{(2)}),$$

where $X_j^{(1)}$ is from the first run of the jth pair, and $X_j^{(2)}$ is from the antithetic run of the jth pair.

Both $X_j^{(1)}$ and $X_j^{(2)}$ are legitimate observations of the simulation model, so that $\mathbb{E}[X_j^{(1)}] = \mathbb{E}[X_j^{(2)}] = \mu$.

Each pair is independent of every other pair. Let

$$X_j = \frac{X_j^{(1)} + X_j^{(2)}}{2},$$

and let the average of the X_j's, $\bar{X}(n)$, be the (unbiased) point estimator of

$$\mu = E(X_j^{(l)}) = E(X_j) = E[\bar{X}(n)].$$

Then since the X_j's are IID,

$$\mathrm{Var}[\bar{X}(n)] = \frac{\mathrm{Var}(X_j)}{n} = \frac{\mathrm{Var}(X_j^{(1)}) + \mathrm{Var}(X_j^{(2)}) + 2\,\mathrm{Cov}(X_j^{(1)}, X_j^{(2)})}{4n}$$

If the two runs within a pair were made independently, then $\text{Cov}(X_j^{(1)}, X_j^{(2)}) = 0$. On the other hand, if we could indeed induce negative correlation between $X_j^{(1)}$ and $X_j^{(2)}$, then $\text{Cov}(X_j^{(1)}, X_j^{(2)}) < 0$, which reduces $\text{Var}[\bar{X}(n)]$; this is the goal of AV.

N.B.: We cannot be completely sure that it will work.

Simulation Optimization

In simulation optimization our goal is to solve:

$$\min_{\mathbf{x}} F(\mathbf{x})$$

where F can be whatever function. For example, $F(\mathbf{x})$ can be:

- expected steady state values
- the value of a two stage stochastic program, etc.

N.B.: The more the assumption on $F(\cdot)$ you can make, the better method you can apply. Here we consider almost no information of $F(\cdot)$, thus we will see the heuristic algorithms.

Heuristics are algorithms exploiting problem-dependent information to find a good solution to a specific problem.

Metaheuristics are design patterns or general algorithmic ideas that can be applied to a broad range of problems.

The first ingredient of metaheuristics is solution representation.

- binary vector;

- permutations;
- continuous vector;
- matrices, etc.

Notation:

- x a solution to the problem
- $\mathcal{N}(x)$ a neighborhood of the solution x,i.e , set of solutions "in some sense" close to x, for example, because they can **be easily computed** from p or because they **share a significant amount of structure** with x.
- $f(x)$ is the objective function measuring the goodness of x.

Let us consider a general problem:

$$\min\{F(x)|x \in \mathcal{X}, \mathcal{X} \subseteq \mathcal{S}\}$$

where:

- $\mathcal{S}$ is the solution space;
- $\mathcal{X}$ is the feasible space.

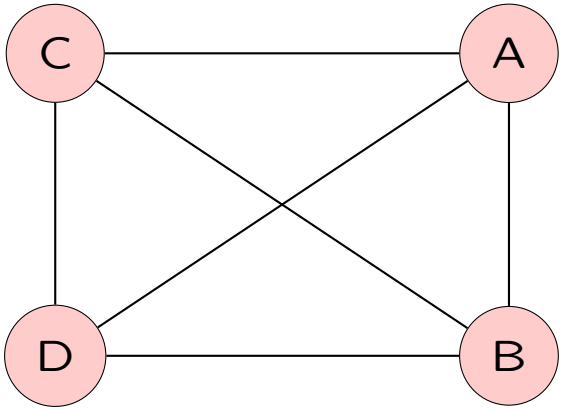
Two main strategies:

- *Exploration*;
- *Exploitation*.

Two general types of metaheuristics:

- *Single Solution-based metaheuristic*: manipulate a single solution during the search
  - " *exploited* oriented: better intensification "
- *Population-based metaheuristic*: a whole population of solutions is evolved
  - *explored* oriented: better diversification
  - usually nature-inspired...

Traveling salesman problem: Let us consider a graph which edges are characterize by a length:



The goal is to visit all the nodes with traveling the smallest possible length.

We can represent the solution as a permutation $[A, B, C, D]$, $[A, B, D, C]$, etc.

Given a solution, $[A, B, C, D]$, a possible neighborhood solution can be:

$$\{[A, B, D, C], [A, C, B, D], [B, A, C, D], [D, B, C, A]\}$$

if we consider every possible swap of two near nodes.

Knapsack problem: We have a set of I items, each characterized by a value v_i and by a weight w_i, we want to maximize the value that we collect taking these items without exceeding a given weight limit W. If we define $x_i \in \{0, 1\}$ is a binary decision variable indicating whether item i is selected, we can write the model as

$$\begin{aligned} \max \quad & \sum_{i=1}^{n} v_i x_i \\ \text{s.t.} \quad & \sum_{i=1}^{n} w_i x_i \leq W \\ & x_i \in \{0, 1\} \forall i = 1, \ldots, I \end{aligned} \tag{2}$$

We can represent the solution as binary vectors, e.g. with 4 items a solution can be $[0, 1, 1, 0]$.

N.B.: with this representation we can have infeasible solutions.

Given $[0, 1, 1, 0]$ a neighborhood can be:

$$\{[1, 1, 1, 0], [0, 0, 1, 0], [0, 1, 0, 0], [0, 1, 1, 1]\}$$

if we consider the swap of one digit.

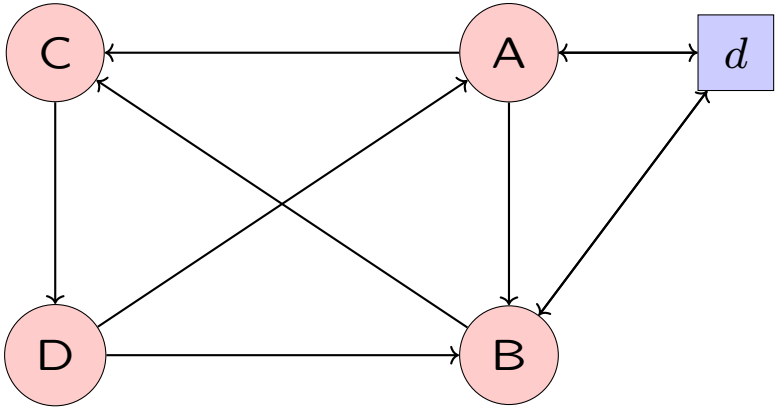
Greedy heuristics

The greedy heuristics are a type of "Single Solution-based metaheuristic" and it is a constructive heuristic:

Algorithm 1 Greedy Heuristic

1: create initial solution $x = \emptyset$
2: **while** x is not complete **do**
3: add to x the best element
4: **end while**
5: return x^*

Example: a directed graph:



- Greedy sol: $d \to A \to B \to C \to D \to B \to d$
- Probably better: $d \to A \to C \to D \to B \to d$

Random search

Random search is a metaheuristic which is based on pure exploration:

Algorithm 2 Random Search

1: create initial solution x^*
2: **while** stopping criteria not met **do**
3: generate x'
4: **if** $f(x') < f(x^*)$ **then**
5: $x^* = x'$
6: **end if**
7: **end while**
8: return x^*

Local search is a metaheuristic which is based on pure exploitation:

Algorithm 3 Local Search

create initial solution x
$x^* \leftarrow x$
while stopping criteria not met **do**
    $x' = \arg\min_{\tilde{x} \in \mathcal{N}(x)} f(\tilde{x})$
    if $f(x') < f(x^*)$ **then**
        $x^* \leftarrow x'$
    end if
    $x \leftarrow x'$
end while
return x^*

N.B.: If we arrive at a local minima, we are stuck there.

GRASP

The *Greedy Randomized Adaptive Search Procedure* (GRASP) can be thought of as a search method that repeatedly applies local search from different starting solutions in $\mathcal{X}$.

- At each step of local search, the neighborhood $\mathcal{N}(x)$ of the current solution x **is searched** for a solution $x' \in \mathcal{N}(x)$ such that $f(x') < f(x)$.
- If such an improving solution is found, it is made the *current solution* or *incumbent solution* and another step of local search is done, i.e., $x \leftarrow x'$ (if $f(x') < f(x^*)$ we also update x^*).
- If no improving solution is found, the procedure picks another point by restarting from scratch or randomly modifying the current solution.

Different searching procedure are possible:

- *First Improvement*: update the solution as soon as a new solution has been found.
- *Best Improvement*: update the solution with the best solution in the neighborhood.

Large Neighborhood Search (LNS) is a solution based meta-heuristics that uses two operators: *destroy* and *repair* methods.

Algorithm 4 Large Neighborhood Search

create initial solution x; $x^* \leftarrow x$
t = 0
while stopping criteria not met **do**
    $x' \leftarrow r(d(x))$
    if accept(x, x') **then**
        $x \leftarrow x'$
        if $f(x') < f(x^*)$ **then**
            $x^* \leftarrow x'$
        end if
    end if
    t += 1
end while
return x_{opt}

$accept(x, x')$:

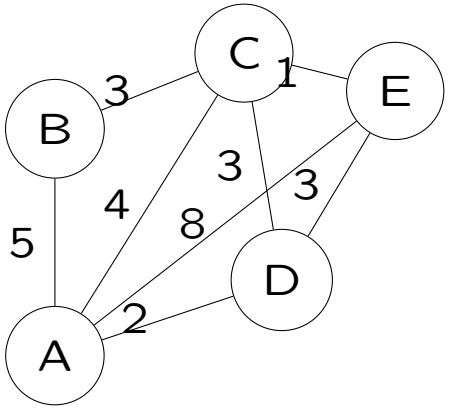
- true if $f(x') < f(x)$
- else true with probability

$$\sim e^{-\gamma t}$$

Example: Consider the TSP:

- destroy part of the solution
- repair using greedy

$$[E, A, B, C, D]\{of : 18\} \xrightarrow{d} [E, \_, \_, \_, D] \xrightarrow{r} [E, C, B, A, D]\{of : 11\}$$
$$\Longrightarrow x_{\text{opt}} = [E, C, B, A, D]$$



Example: Given a knapsack with weight capacity $W = 10$ and the following items:

Item	Value (v_i)	Weight (w_i)	v_i/w_i
1	10	5	2
2	40	4	10
3	30	6	5
4	50	5	10

$$x = [1, 1, 0, 0] \implies \text{Value:} 10 + 40 = 60; \qquad \text{Weight:} 5 + 4 = 9$$

Remove from the solution the item with smallest value / weight (i.e., item 1).

$$d(x) = [0, 1, 0, 0]$$

Sort the items according to value / weight and add all the items that fits (i.e., just item 4).

$$x' = r(d(x)) = [0, 1, 0, 1] \implies \text{Value:} 40 + 50 = 90; \qquad \text{Weight:} 4 + 5 = 9$$

Adaptive Large Neighborhood Search

It is difficult to guess which operator is the most advantageous. Therefore, *Adaptive Large Neighborhood Search* (ALNS) enables the user to consider a set of operators. In the absence of other possibilities, we assume the past success as the best indicator for future success.

Let us consider:

- $D = \{d_i | i = 1, \ldots, k\}$ be the set of k destroy heuristics;
- $R = \{r_i | i = 1, \ldots, l\}$ be the set of l repair heuristics.

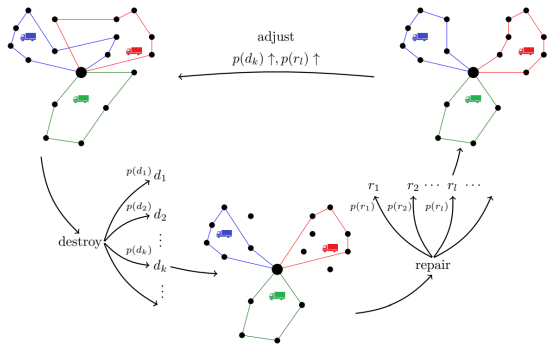
The initially equal weights of the heuristics are denoted by $w(r_i)$ and $w(d_i)$, so that the probabilities to select a heuristic are

$$p(d_i) = \frac{w(d_i)}{\sum_{j=1}^{k} w(d_j)} \qquad p(r_i) = \frac{w(r_i)}{\sum_{j=1}^{l} w(r_j)},$$

respectively.

Algorithm 5 ALNS

create initial solution x.
$x^* \leftarrow x$
while stopping criteria not met **do**
  for $i = 1$ to I **do**
    select $r \in R$, $d \in D$ according to probabilities p
    $x' \leftarrow r(d(x))$
    if accept(x,x') **then**
      $x \leftarrow x'$
      if $f(x) < f(x^*)$ **then**
        $x^* \leftarrow x$
      end if
    end if
  end for
  adjust the weights w and probabilities p
end while



$accept(x, x')$:

- true if $f(x') < f(x)$
- true with probability $\sim e^{-\gamma t}$

Let us call:

- $u(h)$ the number of times the operator h has been used during the I iterations.
- The success $s(h)$ of h is initialized with zero at the beginning of the period of I iterations.

After using h, $s(h)$ is increased by:

- δ_1 if the new solution is the best one found so far.
- δ_2 if the new solution improves the current solution.
- δ_3 if the new solution does not improve the current solution, but is accepted.
- δ_4 if the new solution does not improve the current solution, but it is rejected.

with $\delta_1 > \delta_2 > \delta_3 > \delta_4$.

Finally, the reaction factor $0 \leq \rho \leq 1$ controls the influence of the recent success of a heuristic on its weight. Therefore, we calculate the weights for the following iterations by

$$w(h) = \begin{cases} (1-\rho)w(h) + \rho\frac{s(h)}{u(h)}, & \text{if } u(h) > 0 \\ (1-\rho)w(h), & \text{if } u(h) = 0 \end{cases}$$

As a result, the parameter ρ is decisive for the weight adjustment. With $\rho = 0$, the weights remain constant on their initial level. Hence, for rarely used and unsuccessful heuristics:

$$w(h) = w_{init}(1 - \rho)^{k/I}$$

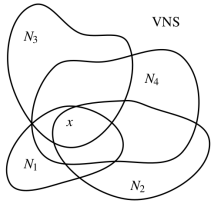
Improvement can be:

- Penalties for time-intensive operators
- Noise and Randomization in operators
- Combine just subsets

Variable Neighborhood Search

Variable Neighborhood Search is a solution based meta-heuristics that leverages the following observations:

- A local minimum w.r.t. one neighborhood structure is not necessarily so for another;
- A global minimum is a local minimum w.r.t. all possible neighborhood structures;
- Local minima w.r.t. one or several $\mathcal{N}_k$ are relatively close to each other, for many problems.



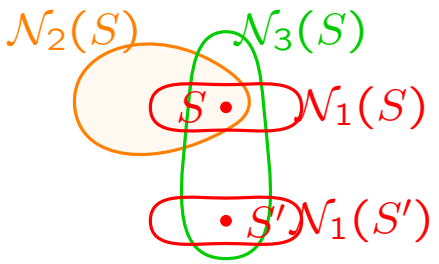
The *basic VNS* is

Algorithm 6 VNS

1: $k = 1$
2: generate x
3: **while** $k = k_{\max}$ **do**
4: $\quad x' \leftarrow \arg\operatorname{opt}_{y \in \mathcal{N}_k(x)} f(x)$
5: $\quad$ Neighborhood change(x, x', k)
6: **end while**

Algorithm 7 Neighborhood change

1: **if** $f(x')$ better than $f(x)$ **then**
2: $\quad x \leftarrow x',\ k = 1$ // make a move
3: **else**
4: $\quad k = k + 1$ // next neighborhood
5: **end if**



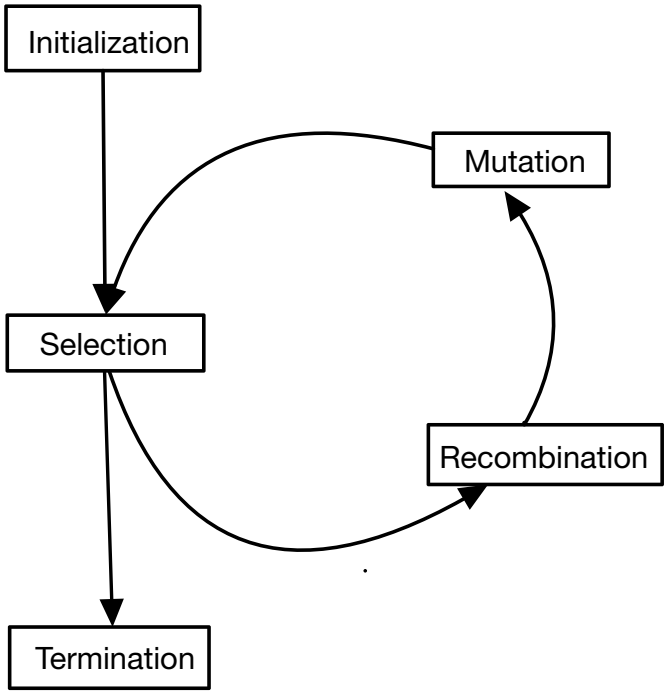
When $\mathcal{N}(x)$ is large, it is convenient not to explore it all. This lead to the Reduced VNS

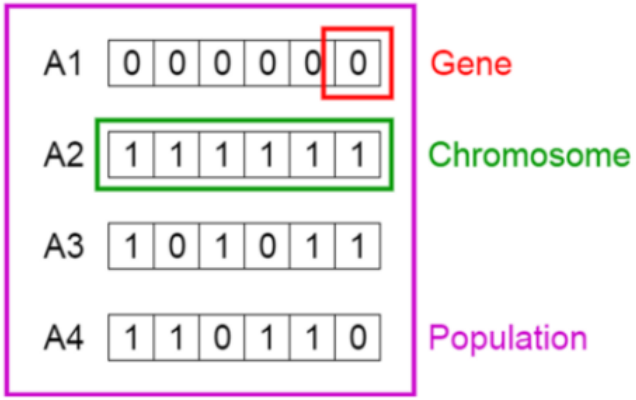
Algorithm 8 Reduced VNS

while $t < t_{\max}$ **do**
  randomly generate x
  $k = 1$
  while $k = k_{\max}$ **do**
    generate x' from $\mathcal{N}_k(x)$ (Possibly destroy and rebuild)
    Neighborhood change
  end while
end while

Genetic Algorithm

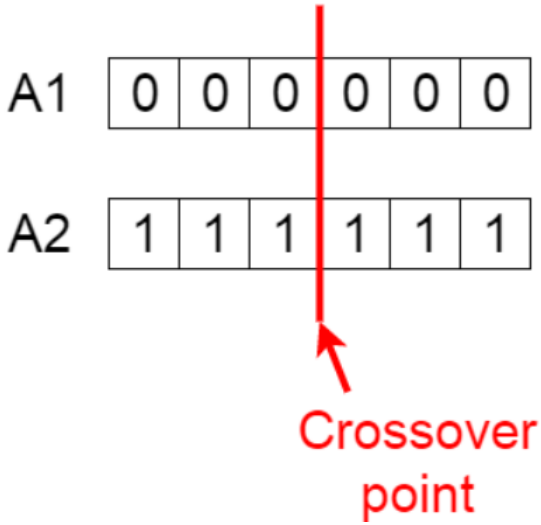
Genetic Algorithm (GA) are Population-based meta- heuristic that Mimics the fundamental processes of genetic evolution: variation and natural selection.

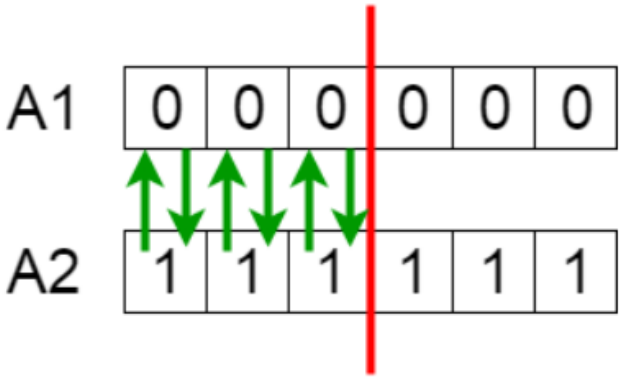


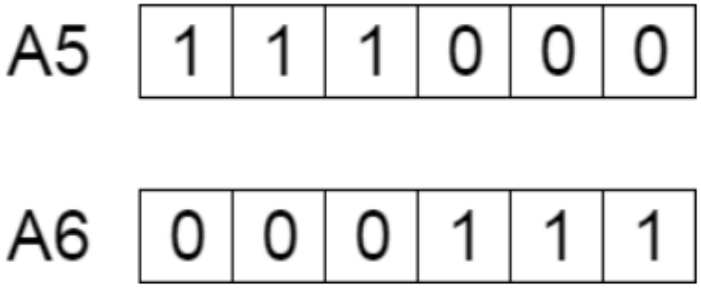


Chromosome: an element of the population, i.e., a solution (may be infeasible) to the model. Also called *individual*.
Fitness function: determines how fit a *chromosome* is.

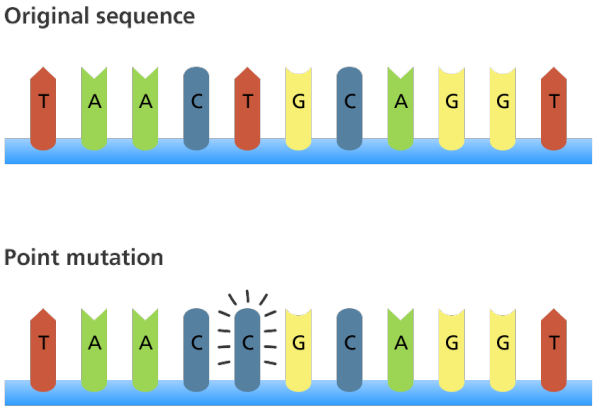
Recobination


A1
0 0 0 0 0 0
A2
1 1 1 1 1 1
Crossover point


A1
0 0 0 0 0 0
A2
1 1 1 1 1 1


A5
1 1 1 0 0 0
A6
0 0 0 1 1 1

Mutation



Before Mutation

A5	1	1	1	0	0	0

After Mutation

A5	1	1	0	1	1	0

- The algorithm may be stopped after a fixed number of iterations
- The algorithm may be stopped when all off-springs are sufficiently close to each other

When you stop the optimal parameters are chosen as those of the "best" individual according to the cost function.

1. Fix an initial population of m solutions
2. Generate a new set of k potential offsprings by
   (a) Applying recombination (also called crossover)
   (b) Applying mutation
3. Apply natural selection to find the m most fit individuals to be used as a new generation

4. If the stopping criterion is met select the best individual in the current generation as the optimal solution, otherwise go back to Step 2

Simulated Annealing

Simulated Annealing is a probabilistic optimization technique used to approximate the global optimum of a given function.
It is inspired by the annealing process in metallurgy.

Key Concepts:

- Works by iteratively making small random changes to the solution.
- A temperature parameter controls the probability of accepting worse solutions.
- Over time, the temperature is lowered, reducing the chance of accepting worse solutions (cooling schedule).

Algorithm:

1. Initialize with a random solution x and an initial temperature T.
2. Repeat until stopping condition:
   - Generate a neighbor solution x' from x.
   - Compute the change in objective function: $\Delta f = f(x') - f(x)$.
   - If $\Delta f < 0$, accept x' (i.e., $x = x'$).

- If $\Delta f \geq 0$, accept x' with probability $P = \exp(-\Delta f / T)$.
- Gradually decrease T according to the cooling schedule.

3. Return the best solution found.

Cooling Schedule:

$$T = \frac{T_0}{1 + \alpha k}$$

where T_0 is the initial temperature, α is the cooling rate, and k is the iteration count.

Item	Weight	Value
1	2	3
2	3	4
3	4	5
4	5	6

Knapsack Capacity: $W = 7$

Tabu Search

Tabu Search is a metaheuristic algorithm used to solve combinatorial optimization problems.

- **Local Search**: Starts from an initial solution and iteratively moves to neighboring solutions.
- **Tabu List**: Keeps track of recently visited solutions to prevent cycling and to explore diverse regions of the search space.
- **Aspiration Criterion**: Allows solutions in the tabu list if they are better than the current best solution.

The main components of Tabu Search are:

- **Neighborhood exploration**
- **Tabu list management**
- **Aspiration criterion**

Tabu Search Pseudocode:

1. Initialize with an initial solution s and set $s_{best} = s$.
2. Set Tabu list $T = \emptyset$.
3. **While** stopping criterion not met:
   - Explore the neighborhood $N(s)$ to find the best neighbor s_{next}.
   - If s_{next} is better than s_{best}, update $s_{best} = s_{next}$.
   - Add the move to the Tabu list T.
   - If s_{next} satisfies the aspiration criterion, accept the move even if tabu.
   - Remove the oldest move from T if the list exceeds the length limit.
   - Update current solution $s = s_{next}$.
4. Return the best solution found, s_{best}.

Example for the Knapsack Problem:

- Knapsack capacity: $W = 10$
- Items:

Item	Value (v_i)	Weight (w_i)
1	10	5
2	40	4
3	30	6
4	50	3

Goal: Maximize the total value while keeping the total weight ≤ 10.

Initial Solution:

$$s = (0, 1, 0, 1) \quad \text{(Items 2 and 4 selected)}$$

$$\text{Value: } 40 + 50 = 90 \quad \text{Weight: } 4 + 3 = 7$$

Step 1: Neighborhood Exploration:

- Explore the neighbors by flipping the inclusion/exclusion of one item.
- Example neighbor: $(1, 1, 0, 1)$ (Add Item 1).
- New solution: $Value = 10 + 40 + 50 = 100, \quad Weight = 5 + 4 + 3 = 12$ (Infeasible).

Example Track moves to prevent reversing to previous solutions. Tabu List Management

Tabu List:

- Keeps track of recently modified solutions or moves to avoid cycling.

- Example: Adding or removing an item from the knapsack.

Aspiration Criterion:

- Even if a move is in the Tabu list, it can be accepted if it leads to a solution that is better than the best solution found so far.

Final Solution After Tabu Search:

$$s = (1, 1, 0, 1) \quad \text{(Items 1, 2, and 4 selected)}$$

$$\text{Value: } 10 + 40 + 50 = 100 \quad \text{Weight: } 5 + 4 + 3 = 10$$

The optimal solution is found by exploring the neighborhood and using Tabu list to avoid cycles and focus the search.

- Tabu Search is effective for solving combinatorial optimization problems like the Knapsack Problem.

- The Tabu list helps in exploring new regions of the search space while avoiding previously visited solutions.

- Aspiration criteria allow for the acceptance of tabu moves that lead to improvements.

Particle Swarm Optimization

Particle Swarm Optimization (PSO) is a population-based optimization technique inspired by the social behavior of birds and fish. It is used to solve optimization problems by moving a group of candidate solutions (particles) toward the optimal solution.

- Particles represent candidate solutions.
- Each particle has a position and velocity in the search space.
- Particles update their positions based on their own experience and the experience of neighboring particles (social interaction).

Algorithm:

1. Initialize a swarm of particles with random positions and velocities.
2. Evaluate the fitness of each particle.
3. Update each particle's velocity and position:

$$v_i(t+1) = \omega v_i(t) + c_1 r_1 [p_i - x_i(t)] + c_2 r_2 [g - x_i(t)]$$

$$x_i(t+1) = x_i(t) + v_i(t+1)$$

where:

- $v_i(t)$ is the velocity of particle i at time t.
- $x_i(t)$ is the position of particle i at time t.
- p_i is the personal best position of particle i.
- g is the global best position.
- ω is the inertia weight.
- c_1 and c_2 are acceleration coefficients.
- r_1 and r_2 are random numbers between 0 and 1.

4. Repeat until a stopping criterion is met (e.g., a maximum number of iterations or convergence).

Velocity Update:

$$v_i(t+1) = \omega v_i(t) + c_1 r_1 (p_i - x_i(t)) + c_2 r_2 (g - x_i(t))$$

- **Inertia term** ($\omega v_i(t)$): Maintains the particle's previous velocity.
- **Cognitive component** ($c_1 r_1 (p_i - x_i(t))$): Pulls the particle toward its personal best position.

- **Social component** ($c_2 r_2(g - x_i(t))$): Pulls the particle toward the global best position.

Position Update:

$$x_i(t+1) = x_i(t) + v_i(t+1)$$

The new position is determined by adding the updated velocity to the current position.

Important Parameters in PSO:

- **Population size:** Number of particles in the swarm.
- **Inertia weight (ω):** Controls the particle's momentum. A larger inertia weight encourages exploration, while a smaller inertia weight encourages exploitation.
- **Cognitive coefficient (c_1):** Determines how much particles are attracted to their personal best positions.
- **Social coefficient (c_2):** Determines how much particles are attracted to the global best position.
- **Random numbers (r_1, r_2):** Introduce randomness to the movement.

Surface Response method

Sometimes running a simulation can be time expensive and directly solving

$$\min\{F(x)|x \in \mathcal{X}, \mathcal{X} \subseteq \mathcal{S}\}$$

can be too expensive.

The main idea is thus to collect a set of points $x_i, F(x_i)$ and build a metamodel $\widehat{F}(x)$ (easy to compute) that approximates $F(x)$, then we can address

$$\min\{\widehat{F}(x)|x \in \mathcal{X}, \mathcal{X} \subseteq \mathcal{S}\}.$$

The metamodel $\widehat{F}(x)$ can be whatever model you like (Bayesian Networks, Regression, Neural Networks, etc.).