Chapitre 4

Méthodes de Résolution en Optimisation Combinatoire

4.1 Introduction

Face à un problème d'optimisation combinatoire donné, la question à laquelle on doit répondre est la résolution du problème. Ce dernier possède généralement un nombre énorme de solutions réalisables. La résolution la plus évidente est de lister toutes les combinaisons possibles afin de trouver celles qui sont valides et meilleures. On appelle ce type d'algorithme : **Algorithmes Exhaustifs**.

Ces algorithmes sont très gourmands en termes de complexité. Ils passent d'algorithmes polynomiaux à non-polynomiaux avec l'augmentation de la dimension du problème à traiter.

Par exemple, si on veut résoudre le problème de voyageur de commerce, on se retrouve avec $n!$ combinaisons possibles. Alors chercher un chemin hamiltonien de 4 villes donne $4! = 24$ combinaisons possibles, tandis que la résolution du même problème avec 10 villes nécessite $10! = 3\ 628\ 800$ combinaisons possibles.

4.2 Problème d'optimisation

L'optimisation appartient au domaine de la Recherche Opérationnelle qui représente la science de prise de décision. Un problème d'optimisation en mathématique consiste à trouver la **meilleure solution possible** pour un problème donné.

Définition 4.2.1 (Problème d'optimisation) *Soit :*

- $n \in \mathbb{N}^*$ *un entier strictement positif,*
- $X \subset \mathbb{R}^n$ *un sous-ensemble non vide*
- $f : X \longrightarrow \mathbb{R}$ *une fonction à valeur réelle.*

*Un **problème d'optimisation** consiste à trouver la meilleure solution $\overline{x} \in X$ appelée **solution globale**.*

La meilleure solution du problème varie selon l'objectif. On parle d'une minimisation ou une maximisation de la fonction f sur l'ensemble X. On note :

$$\overline{x} = min_{x \in X} f(x) \text{ ou } \overline{x} = max_{x \in X} f(x).$$

4.3 Terminologie

- **Variables de décision** : Le vecteur $x \in X$ est appelé **vecteur de décision**. Les éléments formants le vecteur de décision sont appelés **variables de décision**. Ce sont les variables recherchées dans le problème.
- **Dimension de problème** : La dimension du problème est représentée par le nombre entier $n \in \mathbb{N}^*$. Elle représente le nombre de variables de décision. Une solution à un problème de 3 dimensions est un vecteur de 3 éléments (3-uplet).
- **Espace de recherche et Solution admissible** : L'ensemble X de dimension n contient les valeurs que le vecteur x peut prendre. Ces dernières sont appelées **solutions admissibles** ou **faisables** (ou encore solutions candidates). Une solution admissible est une affectation de toutes les variables de décision, qui répond aux contraintes du problème.

  L'ensemble X est connu sous différentes appellations : ensemble admissible, espace de solution, espace de recherche, domaine de recherche etc.
- **Contraintes** : L'espace de recherche X est limité par quelques règles. Une contrainte est une condition que doivent respecter les vecteurs de décision du problème.
- **Solution optimale** : C'est la meilleure solution $\overline{x} \in X$ du problème. Elle est appelée **solution globale** ou **optimale**.
- **Fonction objectif** : la fonction f, à valeur scalaire, sert de critère pour déterminer la meilleure solution du problème. Elle est appelée **fonction d'objectif** (fonction fitness, critère, etc). Elle attribue à chaque solution $x \in X$ de l'espace de recherche, un nombre réel indiquant sa valeur (son score).
- **Minimisation** : Une minimisation revient à trouver un élément $\overline{x}$ de X tel que :

$$\forall x \in X : f(\overline{x}) \leqslant f(x)$$

On dit qu'on cherche **à minimiser** la fonction f sur l'ensemble X. Dans ce cas la fonction d'objectif f est connue comme une fonction de **coût** . Le vecteur $\overline{x}$ est une **borne inférieure** de l'ensemble X.

- **Maximisation** : Une maximisation revient à trouver un élément $\overline{x}$ de X tel que :

$$\forall x \in X : f(\overline{x}) \geqslant f(x)$$

On dit qu'on cherche **à maximiser** la fonction f sur l'ensemble X. Dans ce cas la fonction objectif f est connue comme une fonction d'**utilité** ou de **profit**. Le vecteur $\overline{x}$ est une **borne supérieure** de l'ensemble X.

- **Minimum et maximum stricte** : On parle d'un minimum stricte (respectivement un maximum stricte) si :

$$\forall x \in X : f(\overline{x}) < f(x) \text{ (respectivement } f(\overline{x}) > f(x))$$

- **Voisinage** : Une solution voisine d'une solution x est définit comme une légère modification de la solution x . Cette modification est appelée aussi **mouvement**. Le voisinage de la solution x est l'ensemble des solutions voisines de x.
- **Optimum global et local** :
  - — Un optimum global est la valeur de la fonction objectif de la solution globale. On peut avoir plusieurs solutions globales mais un seul optimum globale .
  - — Un optimum local reflète la meilleure solution dans un entourage voisin.

  Dans la figure 4.1, on remarque qu'il existe deux solutions différentes x_1 et x_2 qui reflètent un seul optimum global $f_1 = f_2$. La solution x_3 représente une solution locale reflétant un optimum local.

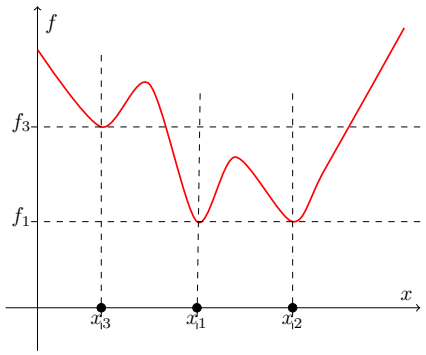


FIGURE 4.1 – Optimum local et global.

4.4 Classification de problème d'optimisation

Il existe plusieurs famille d'optimisation. Il est important de bien identifier à quelle catégorie appartient le problème pour pouvoir choisir la conduite de la résolution. Les problèmes d'optimisation diffèrent selon l'espace de recherche, les contraintes, les objectifs, etc.

4.4.1 Optimisation continue, optimisation discrète et combinatoire

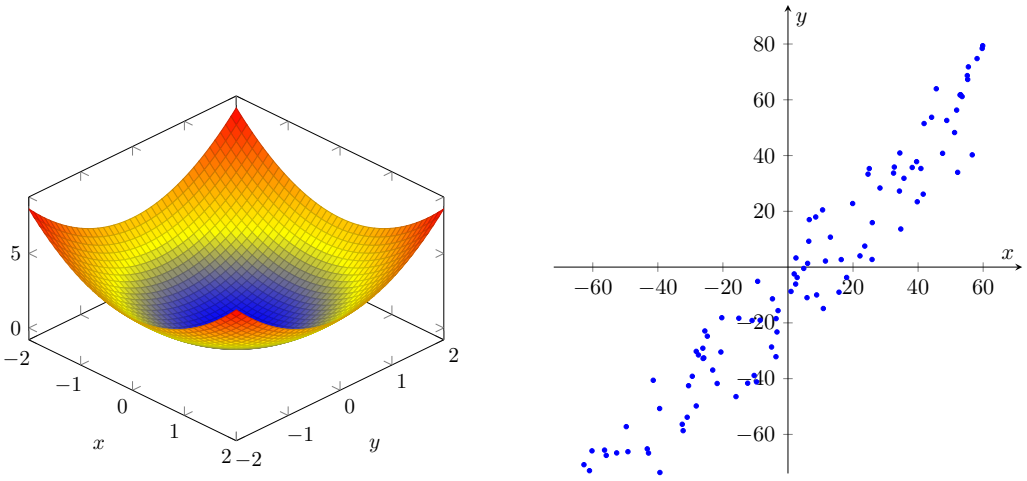
Selon la nature de **l'ensemble de recherche** X on distingue deux classes d'optimisation : l'optimisation continue et l'optimisation discrète.

Optimisation continue : On parle d'**Optimisation continue**, si l'espace de recherche X est continu (par exemple l'ensemble des nombres réels $\mathbb{R}$). La fonction objectif est aussi continue. La fonction sphère (Figure 4.2a) est une fonction continue définie par l'équation 4.1.

$$f(x) = \sum_{i=0}^{n} x_i^2 \tag{4.1}$$

La minimisation de cette fonction est une optimisation continue.

Optimisation discrète : On parle d'**Optimisation discrète**, si l'espace de recherche X est discret et infini. La fonction objectif qui définit un tel problème est représentée par un nuage de points (Figure 4.2b). Les problèmes de cette classe d'optimisation sont plus difficiles que ceux de l'optimisation continue.



(A) Fonction continue (B) Fonction discrète

FIGURE 4.2 – Représentations de fonctions objectif

Optimisation combinatoire : L'**optimisation combinatoire** est une branche de l'optimisation **discrète** dont l'espace de recherche est **fini et dénombrable**.

4.4.2 Optimisation avec et sans contraintes

L'espace de recherche peut être limité par des contraintes. Ces dernières peuvent être des bornes inférieures ou supérieures, ou un ensemble d'équations définissant des situations précises. L'optimisation avec contraintes est plus compliquée à résoudre.

Dans le cas contraire, l'espace de recherche n'est pas limité par des contraintes. Il est alors infini et toutes les solutions incluses sont candidates. L'optimisation sans contraintes est plus facile à résoudre.

4.4.3 Optimisation mono ou multi-objectifs

Un problème d'optimisation peut avoir un ou plusieurs objectifs. Un problème **mono-objectif** est défini par une **unique** fonction objectif. S'il se trouve plusieurs objectifs qui génèrent une contradiction entre eux, on parle alors d'un problème **multi-objectifs**.

Pour des raisons de simplification, un problème multi-objectif peut être reformulé avec une seule fonction objectif ou en transformant des objectifs sous forme de contraintes.

4.5 Domaines d'applications d'optimisation combinatoire

L'optimisation combinatoire intervient sur toute situation nécessitant une planification ou une organisation efficace. Elle est liée à la minimisation de risque, du temps ou du coût ainsi qu'à la maximisation de la qualité, du profit ou de l'efficacité.

L'industrie et l'ingénierie sont les domaines qui profitent le plus des études de l'optimisation combinatoire. On cite quelques exemples :

- La conception de réseaux (électriques, téléphoniques ..)
- La planifications des tâches des employés d'une entreprise.
- Le calcul d'itinéraire pour les applications de géolocalisation.
- La planification du trafic aérien ou routier.
- L'allocation de ressources matérielles et humaines.
- L'ordonnancement et la planification de la production.
- La gestion de portefeuilles.
- Maximisation du profit et minimisation du coût.

4.6 Démarche en optimisation combinatoire

La démarche en optimisation combinatoire comprend plusieurs étapes : l'identification du problème, sa modélisation mathématique, sa résolution par des méthodes exactes ou approchées, et enfin l'implémentation de la solution. L'objectif est de trouver la meilleure solution parmi un ensemble fini de possibilités, même si cet ensemble est très grand et rend une exploration exhaustive impossible.

4.6.1 Exemple

C'est la fin de l'année ! Les étudiants doivent se préparer pour passer leurs examens. Ali passera **3 modules** et ne possède que 100 **heures** pour la révision. Il révise pendant t_i heures le module i ($i = 1, 2, 3$). On suppose que la note n_i du module i dépend uniquement du temps t_i passé à réviser. Supposons que :

$$n_1 = 2 \times t_1, \quad n_2 = 2 \times t_2 - 4, \quad n_3 = 3/2 \times t_3$$

Ali veut optimiser sa tâche et avoir la moyenne maximale en respectant la contrainte de temps. Les variables de décisions x_i pour ce problème représentent les temps de révision (t_i), estimés en minutes, attribués à chaque module .

Le problème revient à trouver $x = (x_1, x_2, x_3)$ qui donne le maximum de moyenne :

$$f(x) = (2 \times x_1 + 2 \times x_2 - 4 + 3/2 \times x_3)/3$$
où
$$x = (x_1, x_2, x_3) \in X = \{x_1, x_2, x_3 \in \mathbb{N}, : x_1 + x_2 + x_3 \leqslant 100 \times 60; \}.$$

Le problème est alors noté comme suit : $max_{x \in X} f(x)$.

4.6.2 Démarches

L'optimisation suit les mêmes démarches de travail de la Recherche Opérationnelle.
La figure 4.3 résume les différentes étapes pour résoudre un problème d'optimisation combinatoire

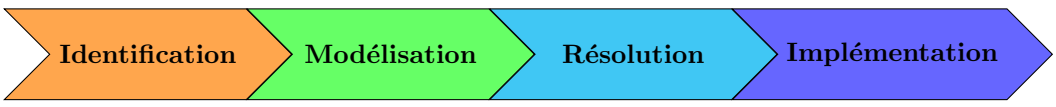


FIGURE 4.3 – Démarche de résolution d'un problème d'optimisation.

4.6.3 Identification

Cette phase est cruciale et très importante au bon fonctionnement des étapes suivantes. A ce stade on tire les informations nécessaires à la modélisation du problème :

1) **Variables de décision** : Identification des variables du problème d'optimisation. Les variables de décision dans l'exemple 4.6.1 sont présents dans le vecteur du temps $x = (x_1, x_2, x_3)$.
2) **Critère** : Définition d'une fonction objectif permettant d'évaluer l'état du système ainsi que le besoin de maximisation ou de minimisation (ex : rendement, performance, coût...). La fonction de la moyenne $f(x) = (2 \times x_1 + 2 \times x_2 - 4 + 3/2 \times x_3)/3$ est la fonction objectif de l'exemple 4.6.1.
3) **Contraintes** : Description des contraintes imposées aux variables de décision. Les contraintes tirées de l'exemple 4.6.1 sont :

$$\begin{cases} x_1, x_2, x_3 \in \mathbb{N} \\ x_1 + x_2 + x_3 \leqslant 100 \times 60 \end{cases}$$

4.6.4 Modélisation

Dans cette étape, on définit les équations nécessaires pour donner une description mathématique au problème. La modélisation peut être mathématique (linéaire et non linéaire), graphique (sous formes de graphes, arbres, graphes bipartis etc.), etc.

On modélise un problème d'optimisation combinatoire en Programme Mathématique qui est définit comme suit :

$$\begin{array}{c} \textit{Maximiser } f(x) \\ \textit{S.C.(sous les contraintes)} \begin{cases} g_i(x) \leqslant 0 \quad i = 1, .., m \\ x \in X \end{cases} \end{array} \tag{4.2}$$

Où

- $X \subset \mathbb{R}^n$ est l'espace de recherche et n est la dimension du problème.
- $x \in X$ est le vecteur de variables de décision.
- $f : X \longrightarrow \mathbb{R}$ est la fonction objectif.
- $g_i : X \longrightarrow \mathbb{R}$, $i = 1, .., m$ sont les m égalités et inégalités qui représentent les contraintes du problème.

Selon la nature de l'ensemble X, on définit les programmes mathématiques **continus** et **discrets**. Dans le cas discret on parle de :

- **Programme mathématique entier**, si l'espace de recherche X ne contient que des nombres entiers ($X \subset \mathbb{Z}^n$).
- **Programme mathématique binaire**, si l'espace de recherche X ne contient que des nombres binaires ($X \subset \{0, 1\}^n$).
- **Programme mathématique mixte**, si l'espace de recherche X est formé de plusieurs sous-ensembles hétérogènes : des sous-ensembles avec des variables qui prennent des valeurs **entières** et d'autres qui prennent de valeurs **continues**.

4.6.5 Résolution

Il existent plusieurs approches de résolution de problèmes d'optimisation selon l'optimalité de la solution globale. Les deux classes essentielles sont les **méthodes exactes** et les **méthodes approchées**.

4.6.6 Implémentation

Une fois la solution établie (via une méthode de résolution) est validée, l'implémentation du système peut être concrétisée. Cela peut impliquer la création de code pour automatiser le processus ou pour intégrer la solution dans un système plus large. Cette étape permet de vérifier que le modèle mathématique et la solution répondent bien aux besoins du problème réel.

4.7 Quelques problèmes classiques de l'optimisation combinatoire

Un problème est dit **combinatoire** lorsqu'il implique l'examen d'un ensemble extrêmement vaste de configurations ou de combinaisons possibles. La recherche d'une solution consiste alors à identifier, parmi ces configurations, celles qui satisfont strictement l'ensemble des contraintes et critères définis par le problème.

4.7.1 Le problème du sac-à-dos (Knapsack)

En fait ce problème se rencontre dans différents domaines de la vie courante au niveau des entreprises et peut être étendu. On peut penser au remplissage d'un camion avec des colis : il faut maximiser l'espace occupé en respectant la contrainte sur le poids total que peut transporter le camion. Vu que la capacité du camion (sac à dos) est **limitée** (un poids ou un volume maximum), on doit faire une décision pour pouvoir le remplir.

Formalisme

On dispose d'un sac-à-dos dont le contenu ne peut pas excéder une capacité c. Considérons un ensemble de n objets, numérotés par l'indice i de 1 à n, possédant chacun un poids w_i (weight en anglais) et une valeur p_i (profit en anglais).

Le problème du sac-à-dos consiste à trouver le sous-ensemble d'objets à charger dans le sac de capacité c afin de **maximiser** la somme des profits.

Il existe plusieurs instances du problème sac à dos : problème de sac à dos à variables binaires, problème de sac à dos à variables entières, problème de sac à dos multidimensionnel, problème de sac à dos multi-objectif, etc.

Instance problème de sac à dos à variables binaires

Cette modélisation se présente comme suit :

- **Variables de décision** : Les variables sont définies sur l'ensemble $\{0, 1\}$: La variable x_i du i-ème élément, $x_i = 1$ si l'élément i est mis dans le sac, $x_i = 0$ s'il est laissé de côté.
- **Contraintes** : Le problème sac-à-dos est limité par deux contraintes. En plus de la contrainte d'intégrité $xi \in 0, 1$, le sac-à-dos ne doit pas dépasser sa capacité c (volume ou poids) :

$$\sum_{i=1}^{n} w_i \, x_i \leqslant c$$

- **Critère** : La fonction objectif du problème sac-à-dos se définit comme la maximisation des profits des objets mis dans le sac :

$$f(x) = \sum_{i=1}^{n} p_i \, x_i \tag{4.3}$$

où p_i est le profit de l'objet i.

Résoudre le problème du sac à dos revient à trouver le vecteur solution $x = (x_1, .., x_n)$ qui **maximise** la fonction objectif définie par l'équation 4.3.

Mathématiquement, un problème de sac à dos est représenté comme suit :

Soient n nombre d'objets, p_i profit de l'objet i, $w_i \in \mathbb{N}^*$ poids de l'objet i et $x = (x_1, .., x_n)$ vecteur de décision, c capacité de sac-à-dos.

$$S.C \begin{cases} Maximiser\ f(x) = \sum_{i=1}^{n} p_i\ x_i \\ x_i \in \{0,1\} \qquad i \in \{1, .., n\} \\ \sum_{i=1}^{n} w_i\ x_i \leqslant c \end{cases} \tag{4.4}$$

Quelle est la complexité du problème ?

Elle est définie par le nombre de parties de k éléments dans un ensemble de n éléments, pour k variant de 1 à n : $\sum_{k=1}^{n} C_k^n = 2^n - 1$

La complexité est donc $O(2^n)$.

4.7.2 Problème de voyageur de commerce (TSP : Travelling Salesman Problem)

Le problème du voyageur de commerce consiste à **trouver le circuit le plus court** qui visite un ensemble de villes exactement **une fois et retourne au point de départ**. Quelques conditions limitent le voyage de l'agent. Chaque ville doit être visitée une est une seule fois. Pour chaque paire de ville, on connaît la distance entre les deux villes.

Formalisme

Considérons n villes représentées par un graphe complet de n sommets, valué par les distances entre ces villes.

On note ce graphe $G = (V, E, c)$ où $V = 1, .., n$ est l'ensemble de sommets (de villes), E est l'ensemble d'arêtes et c est une fonction de coût qui représente la distance entre les 2 villes (c_{ij} est la distance entre les villes i et j).

Le problème de voyageur de commerce consiste à trouver le cycle de longueur minimale passant par chaque ville une et une seule fois. Un tel cycle est dit **hamiltonien** (appelé aussi tour ou tournée).

Le graphe représentant le problème de voyageur de commerce peut être orienté ou non-orienté. Il est orienté si la distance entre les deux villes n'est pas la même pour l'aller et le retour ($c_{ij} \neq c_{ji}$). Dans ce cas, on parle de problème de voyageur de commerce **asymétrique**.

Instance problème de voyage de commerce symétrique

On peut utiliser un programme linéaire binaire ou entier pour modéliser le problème de voyageur de commerce symétrique. La modélisation binaire se présente comme suit :

- **Variables de décision** : Les variables sont définies sur l'ensemble $\{0, 1\}$: La variable x_{ij} correspond à la présence de l'arc $i \longleftrightarrow j$ dans le cycle hamiltonien du voyageur.
  On note $x_{ij} = 1$ si le voyageur prend le chemin existant entre les villes i et j, sinon $x_{\mathbf{ij}} = 0$.
- **Contraintes** : Le problème de voyageur de commerce est limité par plusieurs contraintes :
  1) Contrainte d'intégrité : $x_{ij} \in \{0, 1\}$.
  2) Le voyageur doit entrer une seule fois dans une ville : $\sum_{i=1}^{n} x_{ij} = 1$.
  3) Le voyageur doit sortir une seule fois d'une ville : $\sum_{j=1}^{n} x_{ij} = 1$.

- **Critère** : La fonction objectif du problème de voyageur de commerce se définit comme la **minimisation** de la longueur du chemin :

$$f(x) = \sum_{i=1}^{n} \sum_{j=1}^{n} c_{ij}\ x_{ij} \tag{4.5}$$

où c_{ij} est la distance entre les villes i et j.

Résoudre le problème de voyageur de commerce revient à trouver la matrice $\overline{x} = \begin{bmatrix} \overline{x}_{11} & \cdots & \overline{x}_{1n} \\ \vdots & \ddots & \vdots \\ \overline{x}_{n1} & \cdots & \overline{x}_{nn} \end{bmatrix}$ solution qui minimise la fonction objectif définie par l'équation 4.5.

Mathématiquement, un problème de sac à dos est représenté comme suit :

Soient n nombre de villes, $c_{ij} \in \mathbb{N}^*$ distance entre les villes i et j, et $x = \begin{bmatrix} x_{11} & \cdots & x_{1n} \\ \vdots & \ddots & \vdots \\ x_{n1} & \cdots & x_{nn} \end{bmatrix}$

$$\begin{array}{c} \textit{Minimiser } f(x) = \sum_{i=1}^{n} \sum_{j=1}^{n} c_{ij}\ x_{ij} \\ S.C \begin{cases} x_i \in \{0,1\} & i \in \{1,..,n\} \\ \sum_{i=1}^{n} x_{ij} = 1 & j = 1,..,n \\ \sum_{j=1}^{n} x_{ij} = 1 & i = 1,..,n \end{cases} \end{array} \tag{4.6}$$

Quelle est la complexité du problème ?

Étant donné n sommets d'un graphe, on a :

- n possibilités de choix pour le premier sommet
- puis $n-1$ possibilités de choix pour le second sommet
- ...

La complexité est donc $O(n!)$, sachant que la ville de départ n'a pas d'importance, on peut dire qu'elle est de $(n-1)!$.

4.7.3 Problème d'ordonnancement de tâches

Un problème d'ordonnancement de tâches consiste à organiser l'exécution des tâches d'un projet dans le temps, en respectant des contraintes de ressources (humaines, matérielles) et des contraintes temporelles (délais, dépendances entre tâches), afin d'atteindre un objectif spécifique, tel que minimiser le temps total du projet.

Il s'agit d'un problème d'optimisation qui intervient dans de nombreux domaines comme l'informatique, la logistique, la construction et la gestion de production.

En général, le problème d'ordonnancement est relatif aux points suivants :

- Un ensemble de tâches ou jobs à effectuer.
- Un ensemble de ressources ou machines à utiliser par ces jobs.
- Un programme à identifier, pour allouer les ressources aux tâches.

Les problèmes d'ordonnancement sont des problèmes d'optimisation définis en terme de : **tâches** et **ressources**

Formalisme

Une **tâche** (un **job**) i est une unité élémentaire du travail à effectuer qui commence dans une date t_i (start time) et se termine dans une date c_i (completion time). Elle utilise les ressources pendant un

temps $p_i = c_i - t_i$ (processing time). Un job peut être **interrompu**, c'est-à-dire il est composé de plusieurs opérations. Dans le cas où le job ne peut pas être interrompu, il est vu comme une seule tâche à effectuer.

Une **ressource** est tout ce qu'on utilise pour effectuer un job. Les ressources sont limitées en temps et en quantités. Elles peuvent être des machines, ouvriers, équipements, etc.

Soit $J = 1, .., n$ un ensemble de n jobs et $M = 1, .., m$ un ensemble de machines.

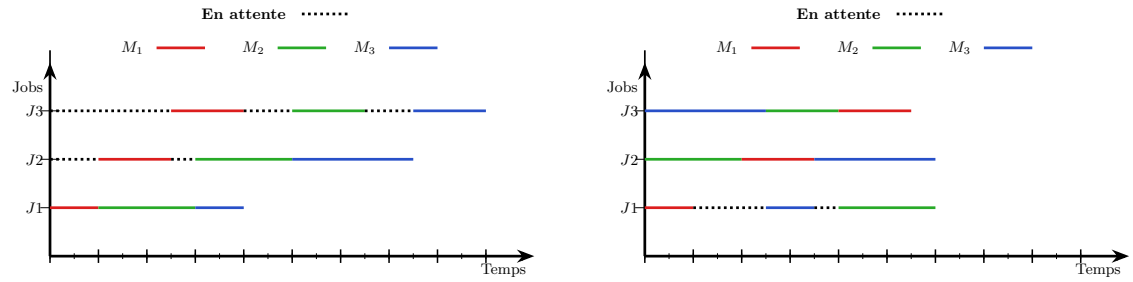
- Chaque job $j \in J$ est composé au plus de m opérations ordonnées $O_{j1}, .., O_{jm}$.
- Chaque opération doit être effectuée sur une machine $k \in M$ spécifique durant un temps spécifique.
- Chaque job doit passer par chaque machine **une et une seule fois**, et chaque machine peut exécuter seulement un job (ou une opération) à la fois sans interruption.

Un **ordre** est une allocation des opérations aux intervalles de temps sur toutes les machines. Un problème d'ordonnancement de tâches consiste à trouver **un ordre réalisable** qui minimise C_{max} le temps nécessaire pour compléter tous les jobs.

Instance problème de d'ordonnancement de tâches

Il existe plusieurs instances pour le problème d'ordonnancement selon le nombres de machines et la nature des jobs. On trouve les ordonnancements sur une machine unique ou sur des machines parallèles. Il existent aussi les ordonnancements linéaire ou multiples.

- **Machine Unique** : L'ensemble des tâches à réaliser est fait par une seule machine. On cherche l'ordre des tâches sur cette machine.
- **Machines parallèles** : L'ensemble des tâches à réaliser est fait par des machines identiques. Les tâches ne sont pas toutes exécutées sur les mêmes machines.
- **Ateliers à cheminement unique (Flow Shop)** : C'est-à-dire, le processus de fabrication est **linéaire** (Figure 4.4a). Les jobs sont découpés en tâches qui doivent s'exécuter sur toutes les machines.
- **Ateliers à cheminements multiples (Job Shop)** : Le processus de fabrication **n'est pas linéaire** (Figure 4.4b). Les jobs sont découpés en tâches qui doivent s'exécuter sur toutes les machines.



(A) Flow Shop (B) Job Shop

FIGURE 4.4 – Représentation de Gantt d'un ordonnancement de 3 tâches sur 3 machines

4.8 Notions de complexité

Avant d'aborder les différentes méthodes de résolution des problèmes d'optimisation combinatoire nous introduisons quelques définitions et notions sur la complexité des problème d'optimisation combinatoire (POC).

Généralement, le temps d'exécution est le facteur majeur qui détermine l'efficacité d'un algorithme, alors la complexité en temps d'un algorithme est le nombre d'instructions nécessaires (affectation,

comparaison, opérations algébriques, lecture et écriture, etc.) que comprend cet algorithme pour une résolution d'un problème quelconque.

Définition 4.8.1 (Complexité) *Une fonction $f(n)$ est $O(g(n))$ (c-à-d $f(n)$ est de complexité $g(n)$), s'il existe un réel $c > 0$ et un entier positif n_0 tel que pour tout $n \geqslant n_0$ on a $|f(n)| \leqslant c\ g(n)$.*

Définition 4.8.2 (Classe P (Polynomial)) *La classe des problèmes **P** est la classe de tous les problèmes qui peuvent être résolu par un **algorithme déterministe** en temps polynomial.*

Définition 4.8.3 (Classe NP (Non-déterministe Polynomial)) *la classe des problèmes **NP** est la classe de tous les problèmes qui peuvent être résolu par un **algorithme non déterministe** dans un temps polynomial.*

Définition 4.8.4 (Algorithme déterministe) *Dans un algorithme déterministe chaque étape de l'algorithme **n'a qu'un seul choix** pour aller à l'étape suivante.*

Définition 4.8.5 (Algorithme non déterministe) *Dans un algorithme non déterministe à chaque étape, l'algorithme a **plusieurs choix** pour aller à l'étape suivante. L'algorithme doit deviner le meilleur choix. Pour cela en peut dire que **la classe des problèmes P est inclus dans la classe des problèmes NP** et en note $P \subset NP$.*

Exemple

- L'algorithme qui recherche d'un élément x dans une liste $A[i],\ i = 1, .., n$.
  Cet algorithme est déterministe de complexité $O(n)$.
- L'algorithme qui consiste à prendre une variable aléatoire $n, i \in [1, n]$ et en test :
  Si $A[i] = x$ alors **succès**, sinon **échec**.
  Cet algorithme est non déterministe de complexité $O(1)$.

Définition 4.8.6 (NP-Complet (NP-Complete)) *Un problème est NP-Complet s'il remplit deux conditions :*

- *Il appartient à NP : On peut vérifier une solution proposée en temps polynomial.*
- *Il est NP-Difficile : Tout problème de NP peut être réduit à ce problème en temps polynomial.*
- *Les problèmes NP-Complets sont les plus difficiles parmi ceux dont on peut vérifier une solution efficacement.*
- *Si on trouve un algorithme polynomial pour un problème NP-Complet $\rightarrow\ P = NP$.*

Définition 4.8.7 (NP-Difficile (NP-Hard)) *Un problème est NP-Difficile si tout problème de NP peut être réduit à lui en temps polynomial, mais :*

- *Il ne doit pas nécessairement appartenir à NP.*
- *On peut ne pas être capable de vérifier sa solution en temps polynomial.*

4.9 Classification des méthodes de résolutions

Les méthodes de l'optimisation combinatoire peuvent être classées en deux grandes familles de classes : les méthodes exactes et les méthodes approchées. La figure 4.6 illustre la taxonomie des méthodes de résolution des problèmes d'optimisation.

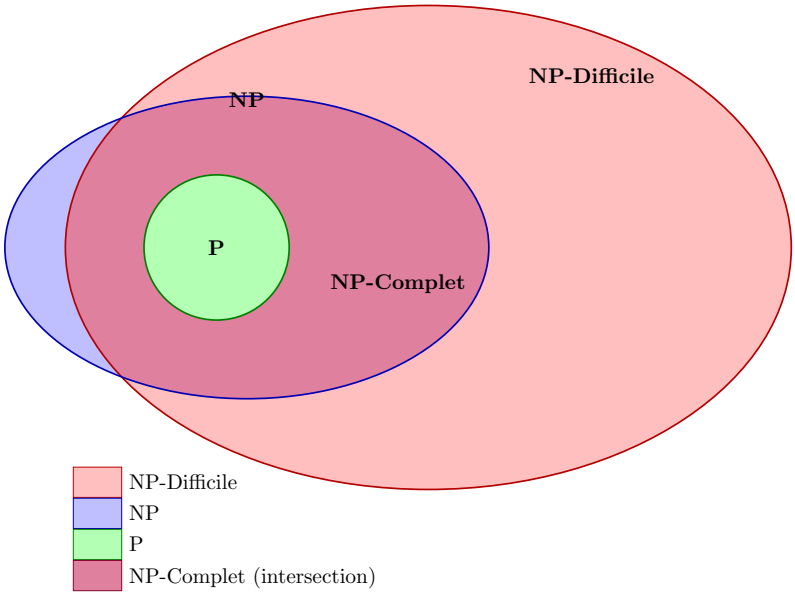


FIGURE 4.5 – Classes des Problèmes.

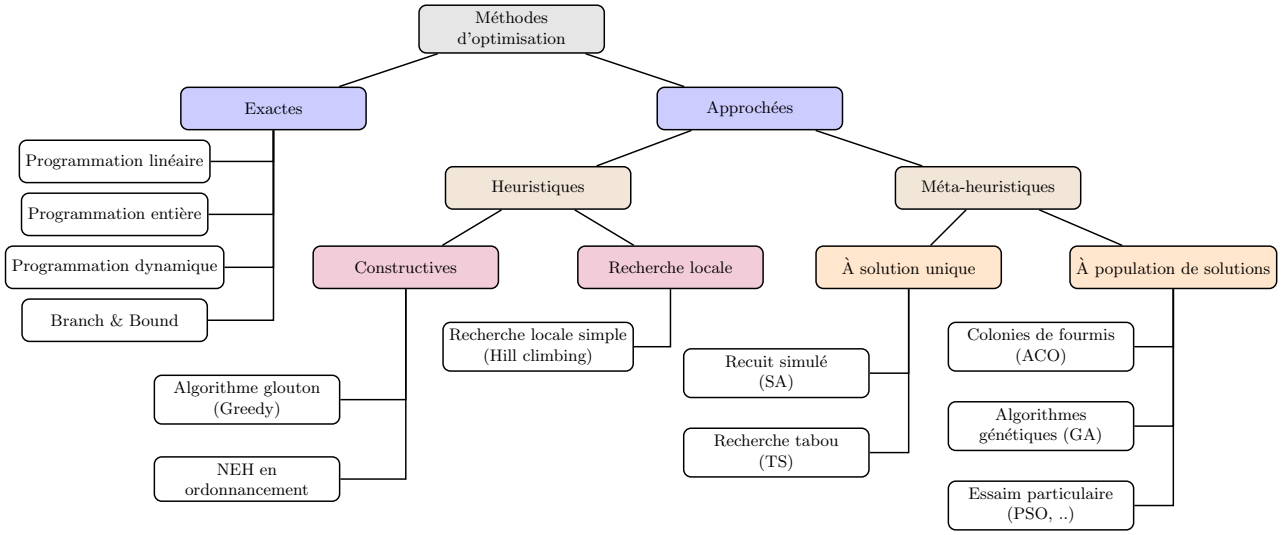


FIGURE 4.6 – Taxonomie des méthodes de résolution de problème d'optimisation.

4.10 Méthodes exactes

Les méthodes exactes (appelées aussi complètes) produisent une **solution optimale** pour une instance de problème d'optimisation donné. Elles se reposent généralement sur la recherche arborescente et sur l'énumération partielle de l'espace de solutions.

Elles sont utilisées pour trouver au moins une solution optimale d'un problème. Les algorithmes exacts les plus connus sont la méthode de séparation et évaluation (Branch & Bound), la programmation dynamique, et la programmation linéaire, etc.

L'inconvénient majeur de ses méthodes est l'**explosion combinatoire** : Le nombre de combinaisons augmente avec l'augmentation de la dimension du problème. L'efficacité de ces algorithmes n'est prometteuse que pour les instances de problèmes de petites tailles.

4.10.1 La méthode séparation et évaluation (B&B : Branch and Bound)

L'algorithme de séparation et évaluation, plus connu sous son appellation anglaise **Branch and Bound (B&B)** (Land et Doig 1960), repose sur une méthode arborescente de recherche d'une solution optimale par séparations et évaluations, en représentant les états solutions par un arbre d'états, avec des nœuds, et des feuilles.

Le Branch-and-Bound est basé sur trois axes principaux : La **séparation**, L'**évaluation**, La **stratégie de parcours**.

1) **La séparation** consiste à diviser le problème en sous-problèmes. Ainsi, en résolvant tous les sous problèmes et en gardant la meilleure solution trouvée, on est assuré d'avoir résolu le problème initial. Cela revient à construire un arbre permettant d'énumérer toutes les solutions. L'ensemble de nœuds de l'arbre qu'il reste encore à parcourir comme étant susceptibles de contenir une solution optimale, c'est à-dire encore à diviser, est appelé ensemble des nœuds actifs.
2) **L'évaluation** permet de réduire l'espace de recherche en éliminant quelques sous-ensembles qui ne contiennent pas la solution optimale. L'objectif est d'essayer d'évaluer l'intérêt de l'exploration d'un sous-ensemble de l'arborescence. Le Branch-and-Bound utilise une élimination de branches dans l'arborescence de recherche de la manière suivante :
   - La recherche d'une solution de coût minimal, consiste à mémoriser la solution de plus bas coût rencontré pendant l'exploration, et à comparer le coût de chaque nœud parcouru à celui de la meilleure solution.
   - Si le coût du nœud considéré est supérieur au meilleur coût, on arrête l'exploration de la branche et toutes les solutions de cette branche seront nécessairement de coût plus élevé que la meilleure solution déjà trouvée.
3) **La Stratégie de parcours** : Dans la pratique, il existe plusieurs stratégies de lecture :
   - **La largeur d'abord** : Cette stratégie favorise les sommets les plus proches de la racine en faisant moins de séparations du problème initial. Elle est moins efficace que les deux autres stratégies présentées,
   - **La profondeur d'abord** : Cette stratégie avantage les sommets les plus éloignés de la racine (de profondeur la plus élevée) en appliquant plus de séparations au problème initial. Cette voie mène rapidement à une solution optimale en économisant la mémoire,
   - **Le meilleur d'abord** : Cette stratégie consiste à explorer des sous-problèmes possédant la meilleure borne. Elle permet aussi d'éviter l'exploration de tous les sous-problèmes qui possèdent une mauvaise évaluation par rapport à la valeur optimale.

Exemple 1

Le problème de sac à dos consiste à remplir un sac à dos de capacité C avec des produits $x_1, x_2, ..., x_n$, qui ont un poids $p_1, p_2, ..., p_n$ et rapportent un cout $c_1, c_2, ..., c_n$ par unité, de façon à maximiser le profit. On considère l'exemple suivant :

$$S.C \begin{cases} Max\ Z = 4x_1 + 5x_2 + 6x_3 + 2x_4 \\ 33x_1 + 49x_2 + 60x_3 + 32x_4 \leqslant 130 \\ x_i \in \mathbb{N} \end{cases} \tag{4.7}$$

Pour ne pas violer la contrainte du problème chaque variable peut prendre les valeurs suivantes $x_1 \in \{0,1,2,3\}$, $x_2 \in \{0,1,2\}$, $x_3 \in \{0,1,2\}$ et $x_4 \in \{0,1,2,3,4\}$, voir la figure 4.7.

1) Le nœud $x_1 = 3$ avec ($x_2 = 0$, $x_3 = 0$, $x_4 = 0$), dans ce cas, on a mis un seul article dans le sac et prendre comme évaluation du nœud $3 \times 4 = 12$: c'est une évaluation exacte qui correspond à une solution. Cela permet d'initialiser la valeur de la meilleure solution courante à 12.

Algorithm 4.1 Macro-algorithme de B&B

Entrées : On note ici L la structure de données des sous-problèmes à explorer.
L Problème correspondant au programme mathématique discret (PMD)
Tant que L non vide **Faire**
  Extraire un sous-problème P de L.
  Si évaluation de P est $>$ à la borne inf **Alors**
    Créer les problèmes-fils P_i de P
    Pour chaque P_i faisables dont l'évaluation est $>$ à la borne inf **Faire**
      Si on dispose d'une solution associée à P_i **Alors**
        Mise à jour borne inf.
      Sinon
        Ajout de P_i dans L.
    Fin Pour.
  Fin Si.
Fin Tant que.
Fin

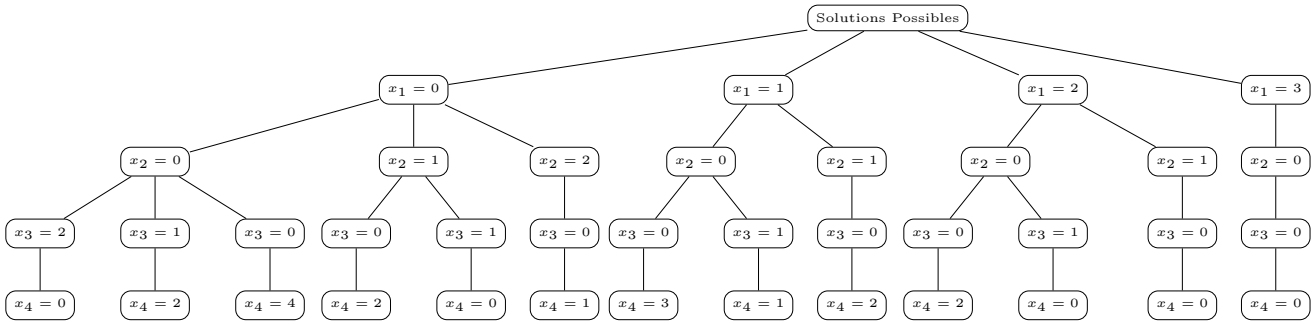


FIGURE 4.7 – Solutions possibles de l'exemple de sac à dos.

2) Le nœud $x_1 = 2$, On fixe l'article qui donne le meilleur rapport **cout/poids**, cela est vérifié pour l'article 2. L'évaluation du nœud est donc calculée par : $8 + (5 \times 64/49) = 14.53$. Puisque $14.53 > 12$, on **divise** ce nœud.
3) Le nœud $x_1 = 2$, $x_2 = 1$, pour le nœud $x_3 = 0$ et $x_4 = 0$, on obtient une évaluation exacte égale à 13. La solution correspondante devient la **nouvelle meilleure solution** courante et la meilleure valeur est égale à 13.
4) Le nœud $x_1 = 2$, $x_2 = 0$ a comme évaluation $8 + (6 \times 64/60) = 14.4$, donc $(x_3 = 64/60, x_4 = 0)$. Puisque $14.4 > 13$, on **divise** ce nœud.
5) Le nœud $x_1 = 2$, $x_2 = 0$, $x_3 = 1$, et $x_4 = 0$ a comme évaluation est exacte égale à 14. La solution correspondante devient la nouvelle meilleure solution courante et la meilleure valeur est égale à 14.
6) Le nœud $x_1 = 1$, a comme évaluation égale à $4 + (5 \times 97/49) = 13.89$, on s'**arrête** à ce nœud (pas d'amélioration) et on **revient en arrière (backtrack)**.
7) On passe au dernier nœud $x_1 = 0$, a comme évaluation égale à $5 \times 130/49 = 13.26$ donc $(x_2 = 130/49)$, on s'**arrête** à ce nœud (pas d'amélioration) et on **revient en arrière (backtrack)**.
8) La valeur de la **solution optimale** égale à 14 et elle consiste à prendre 2 unités du produit 1 et **une** du produit 3 ($x_1 = 2$, $x_2 = 0$, $x_3 = 1$, $x_4 = 0$), voir la figure 4.8.

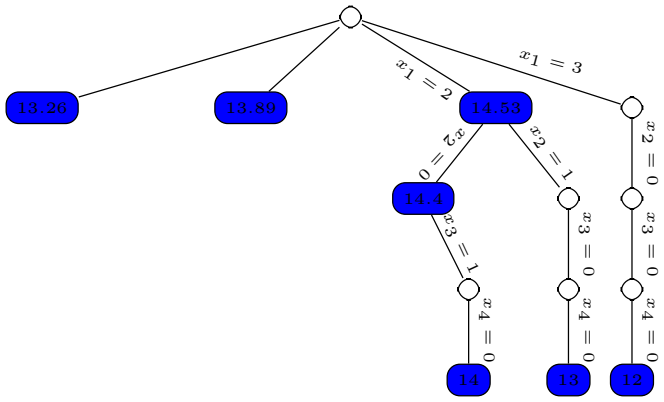


FIGURE 4.8 – Résolution de l'exemple de sac à dos par Branch & Bound.

Exemple 2

Soit le Programme Linéaire suivant :

$$Max\ Z = 8x_1 + 5x_2$$
$$S.C \begin{cases} x_1 + x_2 \leqslant 6 \\ 9x_1 + 5x_2 \leqslant 45 \\ x_1, x_2 \in \mathbb{N} \end{cases} \tag{4.8}$$

Dans un premier temps, nous résoudrons le problème par une méthode de résolution par exemple, la **méthode du simplexe**. Si les solutions des variables sont entières. On s'arrête, on a trouvé l'optimum. Dans l'exemple précédant les résultats obtenus sont donnés comme suit :

$$\begin{cases} x_1 = 15/4 = 3.75 \\ x_2 = 9/4 = 2.25 \\ z = 165/4 = 41.25 \end{cases}$$

La démarche de résolution consiste à concevoir un **arbre** (une **arborescence**) qui, a comme racine, la solution optimale du problème relaxé. Puis, on construit au fur et à mesure, toutes les branches à exploiter en passant d'un sommet a un autre.

La figure 4.9 illustre les phases de séparation et d'évaluation appliquées sur l'arbre de recherche.

La **solution optimale en nombre entier** $x_1 = 5, x_2 = 0$, et $Z = 40$ du problème par l'Éq. 4.8 : est réalisable pour 5 produits de x_1, 0 produit de x_2 et rapporte un bénéfice de 40 unités monétaires.

4.10.2 Programmation dynamique

La programmation dynamique est une méthode algorithmique introduite par Richard Bellman dans les années 1950 pour résoudre **certains** problèmes d'optimisation sous contraintes. Elle repose sur le principe d'optimalité : la solution optimale d'un problème découle des solutions optimales de ses sous-problèmes.

Contrairement à la méthode "**diviser pour régner**", où les sous-problèmes sont indépendants, la programmation dynamique traite des sous-problèmes qui peuvent se **recouper et réutilise leurs solutions** pour éviter les répétitions. Cette technique s'applique essentiellement aux **problèmes séquentiels satisfaisant le principe d'optimalité**, garantissant ainsi que la solution finale obtenue est optimale.

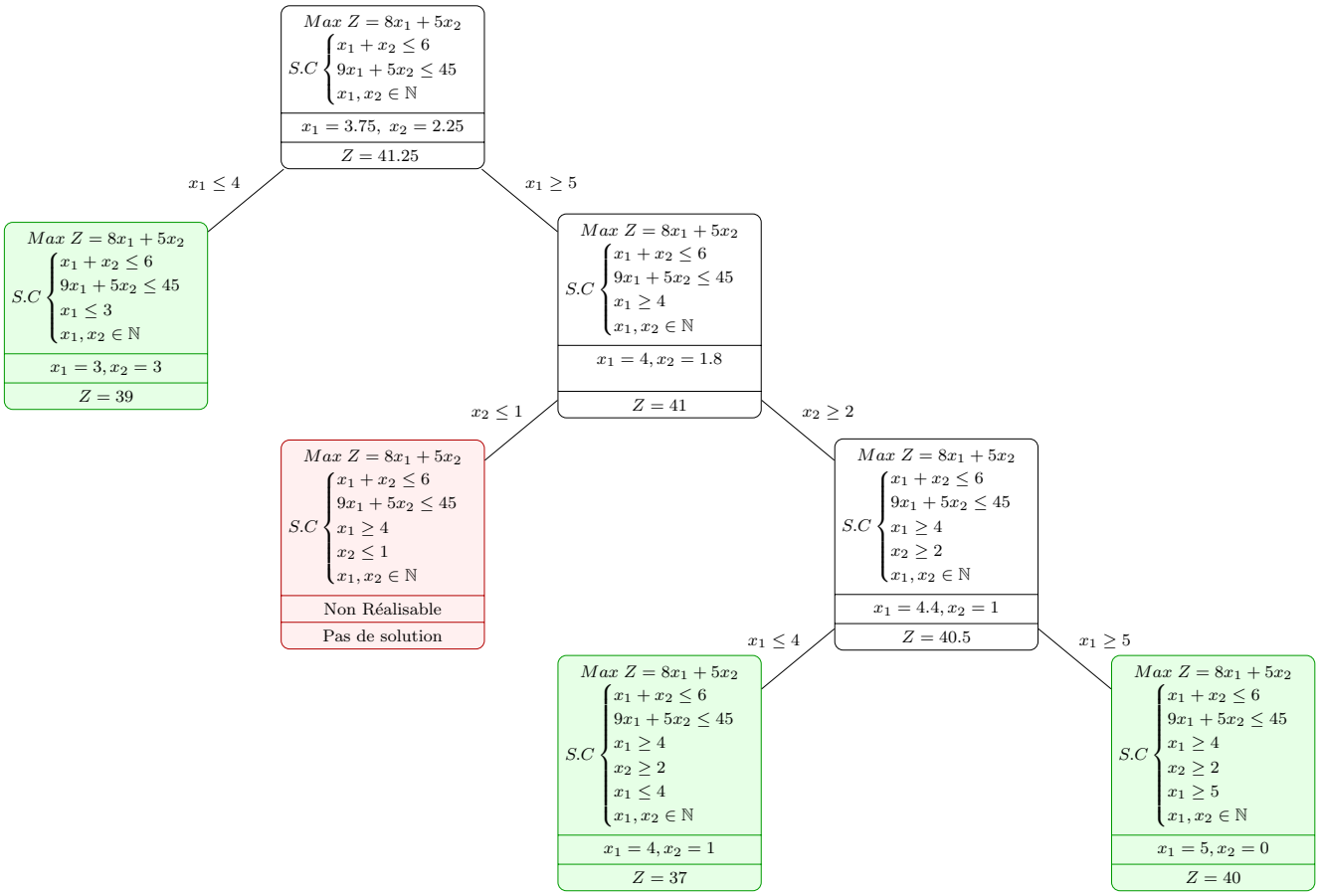


FIGURE 4.9 – Arborescence globale du problème par l'Éq. 4.8.

Algorithm 4.2 Algorithme général de la programmation dynamique

1: **Etape 1 : Formulation de la relation de récurrence** :
Déterminer l'équation récursive qui exprime la solution du problème en fonction des solutions des sous-problèmes.

2: **Étape 2 : Initialisation** :
Définir les valeurs de base selon les conditions initiales de la récurrence et remplir les premières cases de la table.

3: **Étape 3 : Calcul progressif des solutions** :
Remplir la table en résolvant progressivement les sous-problèmes de la plus petite taille vers la plus grande, en appliquant la relation de récurrence.

4: **Étape 4 : Extraction de la solution optimale** :
Lire la valeur optimale dans la table puis reconstruire la solution (si nécessaire) en retraçant le chemin inverse des décisions prises lors du remplissage.

5: **Fin**

Exemple (Le problème de la suite de Fibonacci)

On souhaite calculer les termes de la suite de Fibonacci $F(n) etn \in \mathbb{N}$ en utilisant la définition suivante (Éq. 4.9) :

$$F(n) = \begin{cases} 1 & Si \;\; n = 0 \; ou \; n = 1 \\ F(n-2) + F(n-1) & Sinon \end{cases} \tag{4.9}$$

Algorithm 4.3 Fibonacci récursive

```
1: Fonction Fib(n) :
2:     Si (n ⩽ 1) Alors
3:         Return 1
4:     Sinon
5:         Return Fib(n − 1) + Fib(n − 2)
6:     Fin Si :
7: Fin
```

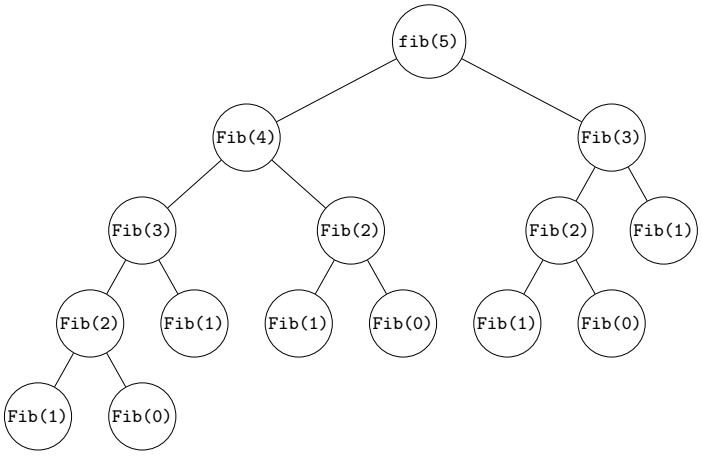


FIGURE 4.10 – Arbre d'appels naïf pour $Fib(5)$.

Le problème de l'algorithme précédent est le nombre d'appels récursifs effectués.
Explication :

- La figure 4.10 montre l'arbre d'appels récursif `Fib(5)` dans l'implémentation naïve : on appelle `Fib(n-1)` et `Fib(n-2)` à chaque nœud. On constate des **redondances** : par exemple `Fib(2)` et `Fib(1)` sont recalculés plusieurs fois.
- Le nombre d'appels croît exponentiellement (en $\Theta(\varphi^n)$ où φ est le nombre d'or), d'où la mauvaise complexité temporelle de l'algorithme naïf.
- La figure 4.11 illustre le principe de la **programmation dynamique** (ou mémoïsation) : on transforme l'arbre en un graphe acyclique orienté (**DAG**) où chaque **sous-problème** `Fib(k)` est calculé une seule fois et son résultat réutilisé. Cela réduit la complexité à $O(n)$ pour la version qui utiliser la programmation dynamique.

Utilisation de la programmation dynamique

Pour améliorer l'efficacité, il faut éviter de recalculer plusieurs fois les mêmes sous-problèmes. En mémorisant chaque résultat dès qu'il est obtenu, par exemple dans un tableau, puis en le réutilisant lorsqu'il est nécessaire, on réduit considérablement le temps d'exécution. Ce principe conduit naturellement à l'approche suivante, basée sur la programmation dynamique.

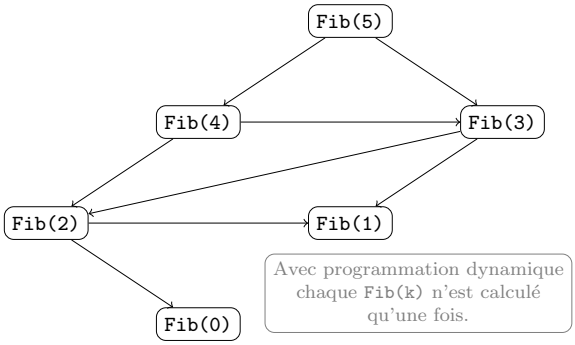


FIGURE 4.11 – Sous-problème `Fib(5)`.

Algorithm 4.4 Fibonacci selon la programmation dynamique

1: **Variable Globale** : Tableau $Tab\_PD[n]$ initialiser à -1
2: **Fonction** $FibPD(n)$:
3: **Si** $(n \leqslant 1)$ **Alors** :
4: $Tab\_PD[n] \leftarrow n$
5: **Return** 1
6: **Fin Si**
7: **Si** $(Tab\_PD[n] \neq -1\,)$ **Alors**
8: **Return** $Tab\_PD[n]$
9: **Fin Si**
10: $Tab\_PD[n] \leftarrow Fib(n-1) + Fib(n-2)$
11: **Return** $Tab\_PD[n]$
12: **Fin**

4.11 Méthodes approchées

Contrairement aux méthodes exactes, les méthodes approchées **ne fournissent pas forcément une solution optimale**, mais seulement une bonne solution (de qualité raisonnable) dans un temps raisonnable.

4.11.1 Heuristiques

Le mot heuristique, dérivé de la langue grec, vient du verbe heuriskein qui signifie **trouver**. Une heuristique est un algorithme qui permet de trouver dans un temps polynomial une solution réalisable, tenant en compte d'une fonction objectif, pas nécessairement optimale (approchée) ou exacte pour un problème d'optimisation difficile.

Ce type de méthodes traduit une stratégie (une manière de penser) en s'appuyant sur la connaissance du problème. Une heuristique est **spécifique** au problème et ne peut pas être généralisée.

Les heuristiques peuvent être classées en deux catégories :

- **Méthodes constructives** qui génèrent des solutions à partir d'une solution initiale en essayant d'en ajouter petit à petit des éléments jusqu'à ce qu'une solution complète soit obtenue,
- **Méthodes de recherche locale** qui démarrent avec une solution initialement complète (probablement moins intéressante), et de manière répétitive essaie d'améliorer cette solution en explorant son voisinage.

  Parmi ces dernières, certaines approches avancées, telles que la recherche Taboue ou le recuit simulé, sont qualifiées de méta-heuristiques.

4.11.2 Méta-heuristiques

Définition 4.11.1 (Méta-heuristiques) *Les méta-heuristiques sont des stratégies **générales d'optimisation** conçues pour orienter la recherche de solutions dans des espaces de grande taille ou complexes. Leur objectif est d'explorer efficacement l'espace de recherche afin d'identifier des solutions de **bonne qualité**, souvent proches de l'optimal.*

Elles englobent des méthodes allant de simples techniques de recherche locale à des mécanismes d'exploration évolués inspirés de processus naturels ou d'apprentissage.

*Les méta-heuristiques sont généralement **non déterministes** et **ne garantissent pas l'optimalité de la solution obtenue**, tout en offrant un compromis efficace entre qualité de solution et temps de calcul.*

Les méta-heuristiques peuvent être classées selon plusieurs critères. Cependant, la distinction la plus couramment adoptée repose sur le nombre de solutions manipulées durant le processus de recherche. Ainsi, on distingue

- d'une part les méta-heuristiques **à solution unique**, qui améliorent progressivement une solution initiale,
- et d'autre part les méta-heuristiques **à population de solutions**, qui explorent simultanément plusieurs solutions afin de favoriser la diversité et d'éviter le piégeage dans des optima locaux.

Il convient toutefois de noter que d'autres classifications existent et peuvent parfois se chevaucher selon la nature et le fonctionnement de la méthode étudiée.

Intensification et Diversification

Les méta-heuristiques se reposent sur la notion d'intensification et de diversification dans le but de tomber sur des nouvelles solutions. Le principe d'intensification et de diversification est un point critique pour les méthodes approchées afin d'éviter les optima locaux. Il consiste à trouver un compromis entre les deux définitions :

- **Intensification** : L'intensification (exploitation) se concentre sur des régions trouvées à l'avance et considérées comme prometteuses en commençant la recherche dans les voisinages de leurs meilleures solutions. Une mémoire doit être utilisée pour implémenter cette stratégie.
- **Diversification** : La diversification (exploration) guide la recherche vers de nouvelles régions dans l'espace de recherche, dans le but de détecter de nouvelles meilleures solutions. Elle peut utiliser l'historique de recherche ou simplement une recherche aléatoire.

Méta-heuristiques à solution unique

Les méta-heuristiques à solution unique (ou **S-méta-heuristiques**) constituent une famille d'algorithmes d'optimisation qui opèrent en améliorant progressivement **une seule solution** au cours du processus de recherche. Elles peuvent être interprétées comme des **trajectoires de recherche** évoluant dans l'espace des solutions. À chaque étape, la solution en cours est transformée en une nouvelle solution, généralement voisine, dans le but d'obtenir une meilleure qualité de solution. Grâce à leur simplicité et leur flexibilité, les S-méta-heuristiques sont largement utilisées et ont démontré leur efficacité dans de nombreux problèmes d'optimisation combinatoire.

Principe : Le fonctionnement des S-méta-heuristiques repose sur un processus itératif alternant deux phases principales :

- **Génération** : À partir de la solution actuelle, un ensemble de solutions candidates $C(s)$ est généré, généralement via des transformations locales (**voisinage**).
- **Sélection** : Une solution $s' \in C(s)$ est choisie pour remplacer la solution courante, selon un **critère de décision** (par exemple, meilleure amélioration ou acceptation conditionnelle).

Ce processus est répété jusqu'à l'atteinte d'un **critère d'arrêt** (nombre d'itérations, temps maximal, ou absence d'amélioration).

La génération et la sélection peuvent être :

- **sans mémoire**, lorsque seules les informations de la solution courante sont utilisées,
- **avec mémoire**, lorsque des informations issues des solutions précédentes sont exploitées afin d'orienter plus efficacement la recherche (ex. : principe utilisé dans la recherche tabou).

La méthode de descente

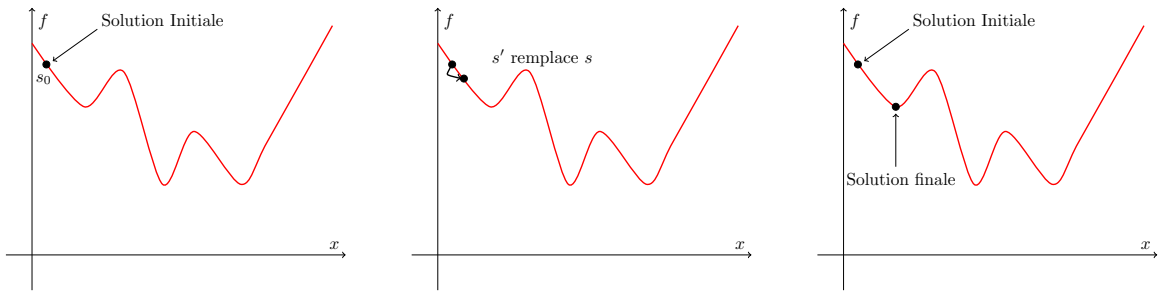
La méthode descente ou recherche locale est probablement la méthode méta-heuristique la plus ancienne et la plus simple. On démarre avec une solution initiale donnée. A chaque itération, l'heuristique remplace la solution actuelle par un voisin qui améliore la fonction objective. La recherche s'arrête lorsque tous les voisins candidats sont pires que la solution actuelle, ce qui signifie qu' un optimum local est atteint.

Pour un problème de minimisation d'une fonction f , la méthode descente peut être décrite dans l'algorithme 4.5.

Algorithm 4.5 Algorithme de la méthode de descente

1: Solution initiale s ;
2: **Répéter**
3: Choisir s' dans un voisinage $N(s)$ de s ;
4: **Si** $f(s') < f(s)$ **Alors**
5: $s \leftarrow s'$;
6: **Fin Si**
7: **Jusqu'à** ce que $f(s') \geq f(s), \forall s' \in N(s)$
8: **Fin**

La figure 4.12 présente l'évolution d'une solution dans la méthode de descente.



(A) (B) (C)

FIGURE 4.12 – Évolution d'une solution dans la méthode de descente

Discussion :

- L'inconvénient majeur de la méthode de descente est son **arrêt au premier minimum local** rencontré. Pour améliorer les résultats, on peut lancer plusieurs fois l'algorithme en partant d'un jeu de solutions initiales différentes, mais la performance de cette technique décroît rapidement.
- Pour éviter d'être bloqué au premier minimum local rencontré, on peut décider d'accepter, sous certaines conditions, de se déplacer d'une solution s vers une solution $s' \in N(s)$ telle que $f(s') > f(s)$.

- On ne sait pas si on obtient une solution optimale et on ne sait pas quand l'analyse en temps est délicate. Par ailleurs, si n (nombre de solutions voisins) est grand, l'examen systématique des n possibilités peut trop ralentir l'algorithme.
- Ce dernier inconvénient peut être pallié en effectuant des tirages aléatoires de modifications et en ne retenant une modification que si elle est meilleure. Il faut alors prévoir un **critère d'arrêt** (nombre d'essais sans amélioration, par exemple).

Recuit Simulé (Simulated Annealing)

Le recuit simulé (Simulated Annealing) est une méta-heuristique inspirée du procédé physique de recuit des métaux, où un matériau est chauffé puis refroidi lentement pour atteindre une structure stable. De manière analogue, l'algorithme explore l'espace des solutions en acceptant parfois des solutions moins bonnes selon une probabilité basée sur le **critère de Metropolis**, ce qui permet d'éviter les minima locaux.

Il démarre avec une solution initiale et une température élevée, puis génère progressivement de nouvelles solutions. Si une solution améliore la fonction objectif, elle est acceptée ; sinon, elle peut encore être acceptée avec une probabilité décroissante selon la distribution de Boltzmann suivante (Éq. 4.10) :

$$P(E,T) = exp^{-\frac{\triangle E}{T}} \tag{4.10}$$

En fonction du critère de Metropolis, un nombre $\varepsilon \in [0,1]$ est comparé à la probabilité de l'Éq. 4.10. Si $P \leq \varepsilon$ la nouvelle solution est **acceptée**.

Le fonctionnement du critère de Metropolis est interprété par

- Si $\triangle E = f(s') - f(s) < 0$ alors $P > 1$, donc ε est toujours inférieur à cette valeur, et on accepte la solution s'.
- Si $\triangle E > 0$
  - et T est très grande, alors $P \cong 1$, tout voisin est systématiquement accepté.
  - et T est très petite, alors $P \cong 0$, une dégradation a peu de chances d'être acceptée.

La température diminue graduellement au cours du temps, rendant l'algorithme d'abord exploratoire puis progressivement plus sélectif, jusqu'à se comporter comme une descente locale classique lorsque $T \to 0$. Ce mécanisme permet d'obtenir des solutions de bonne qualité pour des problèmes complexes sans garantie d'optimalité absolue.

L'algorithme 4.6 résume la méthode de Recuit Simulé.

Le choix de la température est primordial pour garantir l'**équilibre entre l'intensification et la diversification** des solutions dans l'espace de recherche. En particulier, la température initiale doit être choisie en fonction de la qualité de la solution de départ : lorsqu'une solution initiale est générée de manière aléatoire et peut être de faible qualité, il est recommandé d'utiliser une température initiale relativement élevée afin de favoriser l'exploration de l'espace de recherche.

Avantages :

- La méthode du recuit simulé a l'avantage d'être souple vis-à-vis des évolutions du problème et facile à implémenter,
- Contrairement aux méthodes de descente, recuit simulé évite le piège des optima locaux,
- Excellents résultats pour un nombre de problèmes complexes.

Inconvénients :

- Nombreux tests nécessaires pour trouver les bons paramètres,
- Définir les voisinages permettant un calcul efficace de $\triangle E$.

Algorithm 4.6 Recuit Simulé

Entrée : s_0, s et s' : Solution initiale, optimale et voisine ;
$V(s)$: Liste de solutions voisin de s ;
f : fonction de fitness.
T_0, T, fit : réels ;
Sortie : s : Solution ;
Initialiser s_0 et T_0 ;
$s \leftarrow s_0$;
$T \leftarrow T_0$;
$fit \leftarrow f(s)$;
Répéter
  Choisir aléatoirement $s' \in V(s)$;
  $\triangle = f(s') - fit$;
  Si ($\triangle \leq 0$) **Alors**
    $s \leftarrow s'$;
    $fit \leftarrow f(s)$;
  Sinon
    Prendre ε nombre aléatoire de $[0, 1]$;
    Si ($\varepsilon < exp^{\frac{-\triangle}{T}}$) **Alors**
      $s \leftarrow s'$;
      $fit \leftarrow f(s)$;
    Fin Si
  Fin Si
  Mise à jour de la température T ;
Jusqu'à critère d'arrêt ;
Fin

La Recherche Taboue (Tabu Search)

La recherche taboue a été introduite par Glover (1989). L'exploitation du voisinage permet de se déplacer de la solution courante vers son **meilleur** voisin. Ce dernier n'est pas forcément meilleur que la solution courante. Cette méthode permet un déplacement d'une solution s vers la meilleure solution s' appartenant à son voisinage $V(s)$.

L'algorithme accepte parfois des solutions qui n'améliorent pas toujours la solution courante. Nous mettons en œuvre une **liste taboue (tabu list)** T de longueur k contenant les k dernières solutions visitées, ce qui ne donne pas la possibilité a une solution déjà trouvée d'être acceptée et stockée dans la liste taboue. Ce mécanisme constitue le principe du mouvement « tabou », à l'origine de l'appellation de la méthode.

Alors le choix de la prochaine solution est effectué sur un ensemble des solutions voisines en dehors des éléments de cette liste taboue. Une fois la liste taboue est remplie, la solution la plus ancienne est retirée (selon le principe **FIFO**, c'est-à-dire le premier entré est le premier sorti).

L'algorithme 4.7 résume la procédure de la Recherche Taboue.

Discussion :

- La recherche taboue est une méthode de recherche locale dont la structure algorithmique est similaire à celle du recuit simulé.
- Comme critère d'arrêt, il est possible de définir un **nombre maximal d'itérations** sans amélioration de la meilleure solution s^*, ou bien d'imposer **une durée limite** au-delà de laquelle l'algorithme interrompt la recherche.

Algorithm 4.7 Recherche Taboue

Entrée : s_0, s et s' : Solution initiale, optimale et voisine ;
$V(s)$, *ListeTaboue* : Liste de solutions voisin de s et liste Taboue ;
$Enfiler$, $Défiler$: Les fonctions Enfiler et Défiler dans la *ListeTaboue*.
Sortie : s : Solution ;
Initialiser s_0 et $V(s)$ et *ListeTaboue* ;
$s \leftarrow s_0$;
ListeTaboue $\leftarrow \varnothing$;
Répéter
  Choisir la meilleure $s' \in V(s)$ tel que $s' \notin$ *ListeTaboue* ;
  Si (s' *si elle est meilleure que* s) **Alors**
    $s \leftarrow s'$;
  Fin Si
  $Enfiler(ListeTaboue, s')$;
  Si (*ListeTaboue est pleine*) **Alors**
    $Défiler(ListeTaboue)$;
  Fin Si
Jusqu'à critère d'arrêt ;
Fin

- Elle présente l'avantage d'un paramétrage relativement simple : il s'agit notamment de définir une durée t d'interdiction des mouvements (taille de la liste taboue) et de choisir une **stratégie de mémorisation** adaptée.
- Cependant, cette méthode nécessite une gestion de mémoire plus importante, car elle repose sur des mécanismes de mémorisation élaborés pour éviter le retour vers des solutions récemment explorées.
- La taille de la liste taboue est un **paramètre crucial** qui affecte la résolution du problème. Elle peut être statique ou dynamique.
- Dans la méthode de Recherche Taboue, les mécanismes d'intensification et de diversification sont contrôlés par la taille de la liste taboue : On fait de l'intensification, si on diminue la taille et on favorise la diversification dans le cas contraire.
- Grâce à son efficacité, la recherche taboue est largement utilisée pour résoudre des problèmes d'optimisation combinatoire complexes, tels que le problème du voyageur de commerce, l'ordonnancement, ou encore les tournées de véhicules.

Méta-heuristiques à population de solutions

Contrairement aux algorithmes partant d'une solution singulière (S-méta-heuristiques), les méta-heuristiques à population de solutions (**P-méta-heuristiques**) améliorent, au fur et à mesure des itérations, une **population de solutions**. On distingue dans cette catégorie, les algorithmes évolutionnaires, qui sont une famille d'algorithmes issus de la théorie de l'évolution par la sélection naturelle, énoncée par Charles Darwin [Darwin, 1859] et les algorithmes d'intelligence en essaim qui, de la même manière que les algorithmes évolutionnaires, proviennent d'analogies avec des phénomènes biologiques naturels.

Principe :

- Utiliser un **ensemble de solution** au lieu d'une seule
- Algorithme itérative qui **change** la population **à chaque itération**.
- Utilisation des procédures de sélection, génération et évaluation des individus.

- La plupart des P-méta-heuristiques sont des algorithmes **inspirés de la nature**.

Algorithm 4.8 Méta-heuristiques à Population de solutions

```
1: P ← P_0 ;                                   /*Génération de la population initial*/
2: t ← 0 ;
3: Tant que Condition d'arrêt n'est pas satisfait Faire
4:    Generate(P'_t)                           /* Génération d'une nouvelle population */
5:    P_{t+1} = Select_Population(P_t ∪ P'_t)
6:    t ← t + 1
7: Fin Tant que
8: Return Meilleure solution trouve durant toutes les itérations ;
9: Fin
```

Colonies de fourmis (ACO : Ant Colony Optimization)

Les algorithmes de colonies de fourmis, introduits par Colorni, Dorigo et Maniezzo en 1992, ont été initialement appliqués au problème du voyageur de commerce. Il s'agit de méta-heuristiques à population inspirées du comportement collectif des fourmis.

Le principe repose sur l'observation selon laquelle les fourmis, lors de la recherche de nourriture, explorent leur environnement puis renforcent le chemin menant à la source alimentaire au **moyen de phéromones**. Les trajets fréquemment empruntés voient leur trace de phéromones s'intensifier, tandis que les autres s'estompent grâce à l'évaporation. Ce mécanisme naturel conduit l'ensemble de la colonie à converger vers les chemins les plus courts.

Dans la figure 4.13, les fourmis sont en train de suivre une piste de phéromones, comme présenté à la figure (a). Lorsque les fourmis rencontrent un obstacle, elles se répartissent aléatoirement à gauche et à droite, car aucune phéromone n'indique un chemin privilégié. Cependant, le chemin le plus court est retrouvé plus rapidement, ce qui entraîne un dépôt de phéromones plus fréquent sur celui-ci. La concentration croissante de phéromones attire alors davantage de fourmis, renforçant encore cette piste, tandis que celle plus longue s'affaiblit en raison de l'évaporation. Ainsi, la colonie converge progressivement vers le chemin optimal.

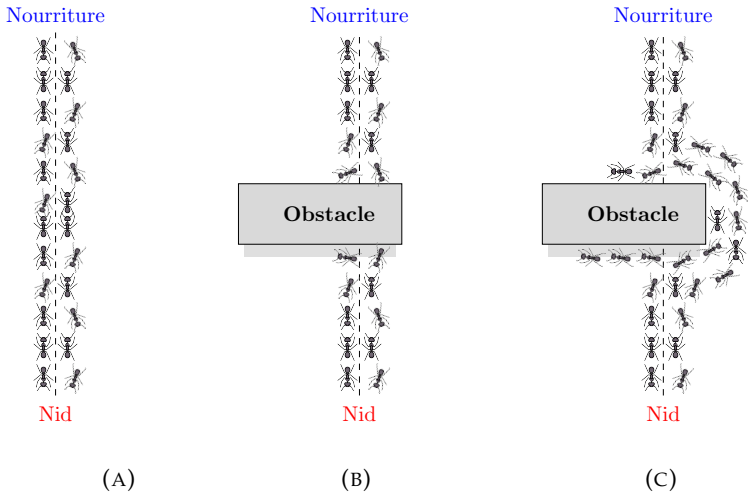


FIGURE 4.13 – Effet de la coupure d'une piste de phéromone.

De la même manière, l'algorithme exploite une coopération entre agents artificiels partageant une mémoire indirecte (phéromones virtuelles) afin de guider la recherche vers des solutions de meilleure qualité pour des problèmes d'optimisation complexes.

Principe :

L'*Ant System* (AS), proposé par **Colorni, Dorigo et Maniezzo (1992)**, constitue une implémentation originale de l'algorithme de colonie de fourmis appliqué au *problème du voyageur de commerce* (TSP).

Ce problème consiste à déterminer le plus court cycle hamiltonien reliant toutes les villes d'un graphe pondéré, où chaque arête (i, j) est associée à une distance d_{ij}.

L'algorithme débute par l'initialisation de la quantité de phéromone sur chaque arête à une valeur $\tau_{\text{init}} > 0$. Chaque fourmi construit ensuite un trajet complet en choisissant son prochain sommet de manière probabiliste, selon la formule :

$$p_{ij}^k(t) = \frac{[\tau_{ij}(t)]^\alpha [\eta_{ij}]^\beta}{\sum_{l \in S_i^k} [\tau_{il}(t)]^\alpha [\eta_{il}]^\beta} \tag{4.11}$$

où :

- $\eta_{ij} = 1/d_{ij}$ représente la **visibilité** (préférence pour les villes proches),
- α et β contrôlent respectivement l'influence de la phéromone et de la visibilité.

Un compromis adéquat entre ces deux paramètres permet d'éviter une convergence prématurée vers un chemin sous-optimal.

Mise à jour des phéromones : Une fois que toutes les fourmis ont terminé leur tournée, elles déposent une quantité de phéromone proportionnelle à la qualité de leur solution :

$$\Delta\tau_{ij}^k = \frac{Q}{L^k(t)} \tag{4.12}$$

où $L^k(t)$ désigne la longueur du trajet parcouru par la fourmi k et Q une constante. Ainsi, plus le chemin est court, plus le dépôt de phéromone est important.

Pour éviter la stagnation et encourager l'exploration, un mécanisme d'évaporation est introduit :

$$\tau_{ij}(t+1) = (1-\rho)\tau_{ij}(t) + \Delta\tau_{ij}(t) \tag{4.13}$$

avec ρ le taux d'évaporation ($0 < \rho < 1$), et :

$$\Delta\tau_{ij}(t) = \sum_{k=1}^{m} \Delta\tau_{ij}^k(t) \tag{4.14}$$

où m est le nombre total de fourmis.

Ce mécanisme équilibre l'intensification (exploitation des bons chemins) et la diversification (exploration de nouvelles solutions).

Les algorithmes génétiques

Les algorithmes génétiques (AG) sont des algorithmes d'optimisation stochastique fondés sur les mécanismes de la sélection naturelle et de la génétique. Ils ont été adaptés à l'optimisation par John Holland (Holland 1975), également les travaux de David Goldberg ont largement contribué à les enrichir.

Le vocabulaire utilisé est le même que celui de la théorie de l'évolution et de la génétique, on emploie le terme individu (solution potentielle), population (ensemble de solutions), génotype (une

Algorithm 4.9 Algorithme de la Colonie de Fourmis (Ant System - AS)

Entrées : Graphe $G = (V, E)$, distances d_{ij}, paramètres α, β, ρ, Q, nombre de fourmis m, nombre d'itérations T_{max}
Sortie : La meilleure solution s^* trouvée
Initialiser les traces de phéromones : $\tau_{ij} \leftarrow \tau_{init}, \forall (i,j) \in E$
Initialiser s^* avec une solution générée aléatoirement
Pour $t = 1$ à T_{max} **Faire**
  Pour chaque fourmi $k = 1$ à m **Faire**
    Placer la fourmi k sur une ville de départ aléatoire
    Initialiser la liste des villes non visitées S_i^k
    Tant que toutes les villes ne sont pas visitées **Faire**
      Calculer la probabilité de déplacement par l'Éq.4.11.
      Choisir la prochaine ville j selon $p_{ij}^k(t)$
      Mettre à jour $S_i^k \leftarrow S_i^k \setminus \{j\}$
    Fin Tant que
  Calculer la longueur du trajet $L^k(t)$
  Fin Pour
  Mettre à jour les phéromones par l'Éqs.4.12 & 4.13
  Mettre à jour la meilleure solution s^* si une meilleure solution est trouvée
Fin Pour
Return s^*
Fin

représentation de la solution), gène (une partie du génotype), parent, enfant, reproduction, croisement, mutation, génération, etc.

Leur fonctionnement est extrêmement simple, on part d'une population de solutions potentielles (**chromosomes**) initiales, arbitrairement choisies. On évalue leur performance (**Fitness**) relative. Sur la base de ces performances on crée une nouvelle population de solutions potentielles en utilisant des opérateurs évolutionnaires simples : la **sélection**, le **croisement** et la **mutation**. Quelques individus se reproduisent, d'autres disparaissent et seuls les individus les mieux adaptés sont supposés survivre. On recommence ce cycle jusqu'à ce qu'on trouve une solution satisfaisante. En effet, l'**héritage** génétique à travers les générations permet à la population d'être adaptée et donc répondre au critère d'optimisation, la figure 4.14 illustre les principales étapes d'un **algorithme génétique**.

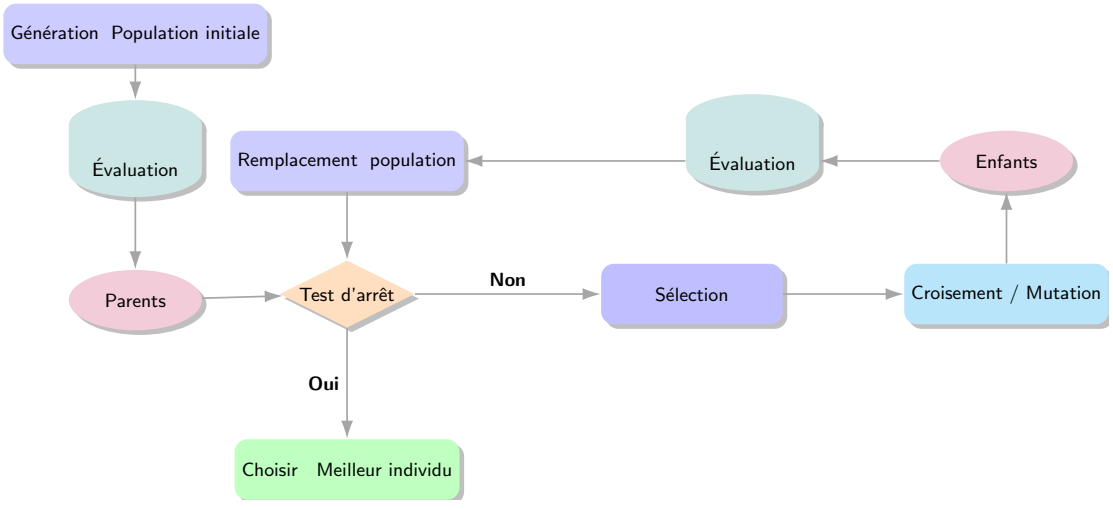


FIGURE 4.14 – Principe de l'algorithme génétique.

Algorithm 4.10 Métaheuristique Génétique (Algorithme Génétique)

1: $P \leftarrow P_0$; /*Génération de la population initial*/
2: $t \leftarrow 0$;
3: **Tant que** Condition d'arrêt n'est pas satisfait **Faire**
4: $P_t^1 \leftarrow Select\_Croisement(P_t)$ /*Sélection de la population pour le croisement */
5: $P_t^c \leftarrow Generete\_Croisement(P_t^1)$ /*Génération des enfants par croisement*/
6: $P_t^2 \leftarrow Select\_Mutation(P_t)$ /*Sélection de la population pour la mutation */
7: $P_t^m \leftarrow Generete\_Mutation(P_t^2)$ /*Génération des enfants par mutation*/
8: $P_{t+1}) \leftarrow Select\_Population(P_t \cup P_t^c \cup P_t^m)$
9: $t \leftarrow t + 1$
10: **Fin Tant que**
11: **Return** Meilleure solution trouve durant toutes les itérations;
12: **Fin**

Les opérateurs des algorithmes génétiques

Le processus de recherche de l'algorithme génétique est fondé sur les opérateurs suivants :

Génération de la population initiale :

L'algorithme génétique (AG) démarre avec une population composée de (N) individus selon le codage retenu. La génération de la population initiale, c'est-à-dire le choix des dispositifs de départ que nous allons faire évoluer, conditionne fortement la rapidité et la qualité de convergence de l'algorithme.

Lorsque la position de l'optimum dans l'espace de recherche est totalement inconnue, il est souhaitable que la population soit répartie de manière homogène sur l'ensemble de l'espace de recherche. Une initialisation aléatoire est souvent utilisée car elle est simple à mettre en œuvre : les valeurs des gènes sont tirées au hasard selon une distribution uniforme.

Cependant, il peut être bénéfique de guider la génération initiale vers des sous-domaines prometteurs de l'espace de recherche. Par exemple, lors d'une recherche d'optima dans un problème d'optimisation sous contraintes, il est préférable de produire **des individus respectant ces contraintes** dès l'initialisation.

Ainsi, la population initiale doit être :

- **suffisamment diversifiée**, pour explorer efficacement l'espace de recherche;
- **de taille adéquate**, afin d'assurer une couverture représentative tout en maintenant un temps de calcul raisonnable.

Une bonne initialisation constitue donc une étape cruciale, car elle influence directement la performance globale et la vitesse de convergence de l'algorithme génétique.

Taille de la population :

La question la plus évidente concernant la mise en œuvre d'un AG est : **quel est la taille de la population à utiliser?**

Des populations de petites tailles entrainent le risque du la non couverture de l'espace des solutions en entier et, ceux de grandes tailles ralentissent, souvent, l'algorithme. Donc, la sélection de la taille de la population est importante, car elle influe sur la vitesse de convergence de l'algorithme. La diversité de la population est aussi importante. La taille de la population est en générale choisie comme constante durant tout le processus. Néanmoins on peut envisager de choisir une taille variable.

Codage des éléments d'une population :

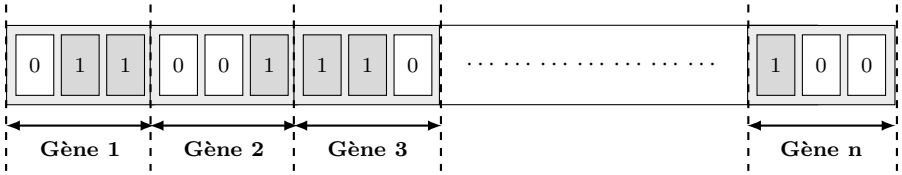
La première étape est de définir et coder convenablement le problème. Cette étape associe à chaque point de l'espace de recherche une structure de données spécifique, appelée **génotype** ou **ensemble de chromosomes**, qui caractérisera chaque individu de la population. Le codage de chaque individu en séquence est essentiel dans l'élaboration d'un algorithme génétique dont dépend notamment l'implémentation des opérateurs de transformations. Ainsi, cette phase détermine la structure de données qui sera utilisée pour coder le génotype des individus de la population. Le codage doit donc être adapté au problème traité.

Un chromosome (une solution particulière) a différentes manières d'être codé selon l'alphabet. Nous distinguons trois types des codages :

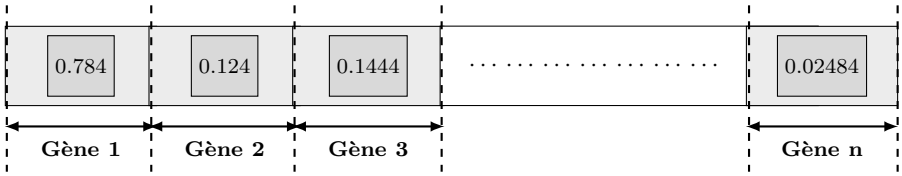
- **Le codage Numérique** : si "l'alphabet" est constitué des chiffres.
- **Le codage Symbolique** : si "l'alphabet" est un ensemble de lettres alphabétiques ou des symboliques.
- **Le codage Alpha-numérique** : si nous utilisons un alphabet combinant les lettres et les chiffres.

Dans le type numérique il y a 2 méthodes : Le codage **binaire** (représentation sous forme de chaîne binaire) et le codage **réel** (représentation directe des valeurs réelles de la variable). Nous pouvons facilement passer d'un codage à l'autre.

- **Codage binaire** : est le premier à être utilisé dans les algorithmes génétiques, car c'est le plus simple à mettre en œuvre avec un alphabet réduit à deux éléments $\{0, 1\}$, rajouter à ça l'existence de fondements théoriques "**théorie des schémas**" et la facilité de mise en œuvre des opérateurs génétiques, tels que le croisement et la mutation.



- **Codage réel** : Contrairement au codage binaire, un gène est représenté par une suite de nombres entiers ou de nombres réels. Ce type de codage peut être utile notamment dans le cas où l'on recherche le maximum d'une fonction réelle.



Fonction d'évaluation / d'adaptation (Fitness Function) :

L'évaluation de **Fitness** est généralement l'étape dans laquelle on mesure la performance de chaque individu. Pour pouvoir juger **la qualité d'un individu** et ainsi le comparer aux autres, il faut établir une mesure commune d'évaluation. Aucune règle n'existe pour définir cette fonction, son calcul peut ainsi être quelconque, que ce soit une simple équation ou une fonction affine. La manière la plus simple est de poser la fonction d'adaptation comme la formalisation du critère d'optimisation.

Sélection :

La sélection permet d'identifier statistiquement les **meilleurs** individus d'une population et **d'éliminer les mauvais**, pendant le passage d'une génération à une autre, ce processus est basé sur la performance de l'individu. L'opérateur de sélection doit être conçu pour donner également une chance aux mauvais éléments, car ces éléments peuvent, par croisement ou mutation, engendrer une descendance pertinente par rapport au critère d'optimisation. Il existe différentes techniques de sélection, on propose quelques-unes :

- **Sélection uniforme** : On ne s'intéresse pas à la valeur d'adaptation fitness et la sélection s'effectue d'une manière **aléatoire** et uniforme telle que chaque individu i a la même probabilité $Prob(i) = 1/T_pop$ comme tous les autres individus (T_pop : est la taille de la population).
- **Sélection par tournoi** : Deux individus sont choisis au hasard, on compare leurs fonctions d'adaptation et le **mieux adapté** est sélectionné.

- **Sélection par roulette** : La sélection des individus par la méthode de la roulette s'inspire de la roue de loterie sur laquelle chaque individu est représenté par un secteur proportionnel à son fitness. On fait tourner la roue et on sélectionne un individu. Les individus les mieux évalués ont statistiquement plus de chance d'être sélectionnés, mais donne aussi une possibilité aux individus mal adaptés d'être choisis. À chaque individu i une probabilité est associée, d'être choisi proportionnelle à son adaptation f_i :

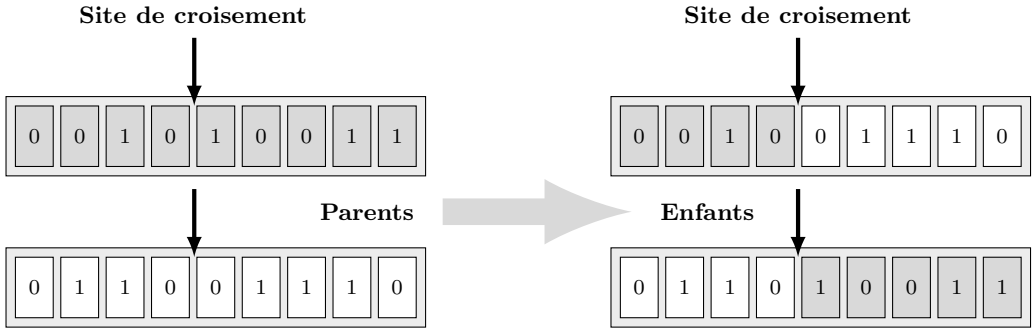
$$Prob(i) = f_i/(\sum f_i) \tag{4.15}$$

  où $\sum f_i$ désigne la somme des adaptations de la population.
- **Élitisme** : Cette méthode de sélection permet de favoriser les meilleurs individus de la population. Ce sont donc les individus les plus prometteurs qui vont participer à l'amélioration de notre population. On peut constater que cette méthode induisait une convergence prématurée de l'algorithme.
- S**élection par rang** : est la même que par roulette, mais les proportions sont en relation avec le rang plutôt qu'avec la valeur de l'évaluation. La sélection par rang trie d'abord la population par fitness. Ensuite, chaque chromosome se voit associé un rang en fonction de sa position. Ainsi le plus mauvais chromosome aura le rang 1, le suivant 2, et ainsi de suite jusqu'au meilleur chromosome qui aura le rang N (pour une population de N chromosomes).

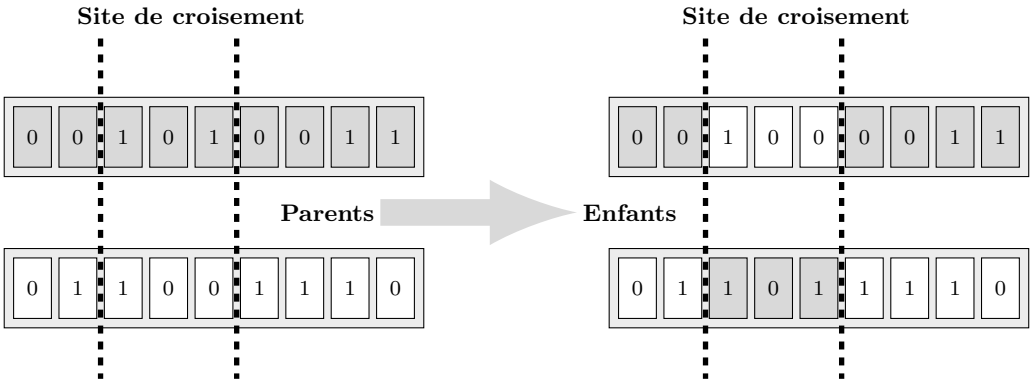
Croisement :

L'opérateur de croisement favorise l'exploration de l'espace de recherche et enrichit la diversité de la population en manipulant la structure des chromosomes, le croisement fait avec deux parents et génère deux enfants, en espérant qu'un des deux enfants au moins héritera de bons gènes des deux parents et sera mieux adapté qu'eux. Le taux maximal de croisement est contrôlé par la probabilité de croisement P_C, qui est fixée par l'utilisateur selon le problème à optimiser. Il existe plusieurs méthodes de croisement, dont nous présenterons deux types :

- **Croisement en un point** : Pour chaque couple, on choisit au hasard un point de croisement. Le croisement s'effectue directement au niveau binaire, et non au niveau des gènes. Un croisement peut être coupé au milieu d'un gène.



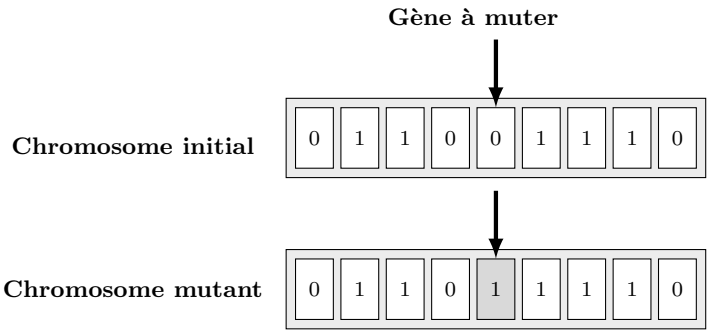
- **Croisement en deux points** : On choisit au hasard deux points de croisements successifs. L'enfant 1 sera une copie du parent 1 en remplaçant la partie entre les deux positions par celle du parent 2. La même opération sera effectuée pour déterminer l'enfant 2, en inversant les rôles du parent 1 et du parent 2. Cet opérateur est généralement considéré comme plus efficace que le précédent.



- **Croisement uniforme** : Il constitue la généralisation du principe d'échange introduit par le croisement en un point. Il procède en effet à l'échange de chaque élément selon une probabilité fixée. Le jeu de paramètres se réduit donc à la donnée de cette probabilité.

Mutation :

L'opérateur de mutation est un processus où un changement aléatoire mineur du code génétique appliqué à un individu pour introduire de la diversité et ainsi d'éviter de tomber dans des optimums locaux. Cet opérateur est appliqué avec une probabilité P_M (fixée par l'utilisateur) généralement inférieure à celle du croisement P_C. La mutation classique consiste à transformer dans un chromosome binaire 1 en 0 ou le contraire.



Critère d'arrêt :

Le critère d'arrêt joue un rôle primordial dans le jugement de la qualité des individus. Son but est de garantir la solution optimale obtenue par l'algorithme génétique. Les critères d'arrêts sont de deux natures :

- Arrêter après un **nombre fixé** de générations. C'est la solution adoptée lorsqu'une durée maximale de calcul est imposée.
- Arrêter lorsque la population cesse d'évoluer ou n'évolue plus suffisamment. On se retrouve alors avec une population homogène que l'on peut supposer proche de l'optimum. Ce critère d'arrêt reste le plus objectif et le plus utilisé.

Fonctionnement d'un Algorithme Génétique

Un algorithme génétique (AG) est un **processus de recherche exploratoire itératif** dont les étapes sont les suivantes :

1) Initialisation de la population initiale P_0 (population de génération 0).
2) Évaluer les chromosomes de la population P_i
3) Sélectionner les meilleurs chromosomes de P_i qui constitueront les « **procréateurs** » de la génération $P_{(i+1)}$.

4) Croiser les chromosomes sélectionnés deux à deux pour générer de nouveaux individus, les « **descendants** ».
5) Effectuer aléatoirement une mutation sur un chromosome également choisi au hasard pour apporter de la diversité dans le patrimoine génétique de la population.
6) Remplacer et générer une nouvelle population $P_{(i+1)}$.
7) Répéter le processus à partir de l'**Étape 2** tant que le critère d'arrêt n'est pas satisfait (c'est-à-dire que la génération N terminale n'a pas été atteinte, la solution optimale n'a pas été trouvée, etc.)

Exemple de la mise en œuvre des algorithmes génétiques

Cet exemple est dû à Goldberg (1989). Il consiste à trouver le maximum de la fonction $f(x) = x$ sur l'intervalle $[0, 31]$, où x est un entier naturel. On a 32 valeurs possibles pour x, on choisit donc un codage discret sur 5 *bits* : on obtient par exemple la séquence 01101 pour 13, la séquence 11001 pour 25, etc.

Initialisation de la population se fait d'une manière aléatoire et en fixant sa taille à 4 **individus**. On définit simplement le **fitness** comme étant la valeur de x, vu qu'on en cherche la valeur maximum sur l'intervalle $[0, 31]$ plus la valeur de x sera élevée plus on se rapprochera du maximum de la fonction identité et donc plus le fitness sera grand. Soit la population initiale suivante :

Individu	Séquence	Adaptation	% du total
1	01011	11	18.6
2	10011	19	32.2
3	00101	5	8.5
4	11000	24	40.7
Total		59	100

Nous utilisons après une sélection par roulette : nous faisons tourner la roue 4 fois de suite, nous obtenons une nouvelle population :

Individu	Séquence
1	11000
2	00101
3	11000
4	01011

Nous appliquons l'opérateur de croisement en un seul point. Les point de croisement (crossover) sont choisis aléatoirement :

- **Couple 1** : l'individu 2 avec l'individu 3.
- **Couple 2** : l'individu 1 avec l'individu 4.

Nous obtenons alors les enfants suivants :

Parents	Enfants
001 || 01 110 || 00	00100 11001
110 || 00 010 || 11	11011 01000

Nous appliquons ensuite l'opérateur de mutation qui choisit de manière aléatoire l'individu et le gène où on doit faire une mutation :

Parents	Enfants
00100 11**0**01	00100 11 **1** 01
01 **0** 00 11011	01 **1** 00 11011

Nous remplaçons entièrement l'ancienne population P_0 par une nouvelle $P^{'}$ de même taille :

$P^{'}$	Séquence	Adaptation
1	00100	4
2	11101	29
3	01100	12
4	11011	27
Total		72

En une seule génération le maximum est passé de 24 à 29, et le fitness global de la population a relativement augmentée pour passer de 59 à 72. Nous continuons à engendrer des générations successives jusqu'à obtenir le maximum global qui vaut 31.