République Algérienne Démocratique et Populaire
Ministère de l'Enseignement Supérieur et de la Recherche Scientifique

Université des Sciences et de la Technologie d'Oran – Mohamed Boudiaf
Faculté des Mathématiques et Informatique
Département d'Informatique

Polycopié de cours

Recherche Opérationnelle et Combinatoire

Préparé par : **Mahmoud Zennaki**
Docteur en Informatique
Maître de Conférences
mahmoud.zennaki@univ-usto.dz

Domaine : **Mathématiques et Informatique**

Filière : **Informatique**

Spécialités : **Intelligence Artificielle et ses Applications**
Réseaux et Systèmes Informatiques Distribués
Systèmes d'Information et Données
(Master académique – Semestre 1)

Année : 2020

Table des matières

INTRODUCTION GENERALE

Qu'est-ce que la recherche opérationnelle ?
La recherche opérationnelle (RO) vise à l'amélioration du fonctionnement des entreprises et des organismes publics par l'application de l'approche scientifique. Reposant sur l'utilisation de méthodes scientifiques, de techniques spécialisées et des ordinateurs, la RO permet d'obtenir une évaluation quantitative des politiques, stratégies et actions possibles dans le cours des opérations d'une organisation ou d'un système.

La RO est apparue en Grande-Bretagne durant la seconde guerre mondiale, lorsqu'on décida d'employer des méthodes scientifiques pour étudier divers aspects des opérations militaires. Depuis lors, la RO est devenue un élément important du processus de prise de décision dans de nombreux contextes commerciaux, industriels et gouvernementaux, car elle permet d'appréhender de façon systématique la complexité toujours grandissante des problèmes de gestion auxquels est confronté tant le secteur privé que public.

À la suite des succès obtenus dans le domaine militaire durant la seconde guerre mondiale, la RO a été appliquée durant de nombreuses années à des problèmes de nature opérationnelle dans l'industrie, le transport, etc. Depuis une vingtaine d'années, le champ d'application de la RO s'est élargi à des domaines comme l'économie, les finances, le marketing et la planification d'entreprise. Plus récemment, la RO a été utilisée pour la gestion des systèmes de santé et d'éducation, pour la résolution de problèmes environnementaux et dans d'autres domaines d'intérêt public. Parmi les sujets d'application récents de la RO, on peut mentionner :

- les études logistiques (forces armées),
- la sécurité ferroviaire,
- la conception d'emballages,
- la gestion prévisionnelle du personnel,
- le transport aérien,
- les opérations forestières,
- l'optimisation du carburant nucléaire,
- l'affectation des ressources dans un hôpital,
- l'étude des réseaux de commutation,
- la planification de la production,
- l'apprentissage artificiel.

Une carrière en recherche opérationnelle
Pour réussir, le chercheur opérationnel doit faire preuve de grandes habilités analytiques, d'un esprit ouvert et d'un intérêt marqué pour la résolution de problèmes pratiques.

À l'heure actuelle, on retrouve des personnes possédant une formation en RO à l'intérieur d'équipes spécialisées en RO œuvrant pour certains dans divers secteurs d'organismes privés ou publics. Les méthodes de la RO sont aussi appliquées par des économistes, des ingénieurs, des scientifiques, des administrateurs et des cadres supérieurs dans la résolution de problèmes de gestion et de politique d'entreprise.

Depuis l'apparition de la RO, les décideurs et les gestionnaires y ont recours très fréquemment. Ses applications sont en pleine expansion alors que l'envergure et la complexité des problèmes soumis aux praticiens de la RO n'ont pas cessé de s'accroître. En conséquence, le développement de techniques et méthodes nouvelles pour faire face à ces nouveaux problèmes a provoqué chez les gestionnaires de tous les secteurs des affaires, de l'industrie et du gouvernement une prise de conscience sans cesse grandissante de la nécessité de telles techniques et méthodes ainsi qu'une plus grande confiance en ce que la RO peut faire pour la gestion et la prise de décisions. Il y a dans les secteurs public et privé une demande croissante et un besoin certain des services de la RO.

PARTIE 1 : OPTIMISATION COMBINATOIRE

1. DEFINITION D'UN PROBLEME D'OPTIMISATION COMBINATOIRE (POC)

Découvrir le plus court chemin pour se rendre à un endroit précis, concevoir un circuit électronique, placer des antennes pour diffuser des chaînes TV, ou des relais pour les réseaux GSM, reconstruire le code génétique des êtres humains,… cette liste peut paraître hétérogène et provenant de domaines différents, mais elle contient des problèmes qui ont tous une structure *combinatoire*. En effet, chacun d'entre eux revient à choisir la *meilleure combinaison* parmi un nombre souvent *exponentiel* de possibilités.

Définition : Un POC peut être décrit en général sous la forme suivante : $\boldsymbol{min}\ \boldsymbol{F}(\boldsymbol{x})$ $\boldsymbol{x} \in \boldsymbol{X}$

Où F désigne une fonction à valeurs réelles dite fonction objectif évaluant les solutions désignées par x.

Il s'agit de minimiser la fonction F sur l'ensemble de toutes les solutions réalisables noté X. Les solutions $x \in X$ peuvent être des objets mathématiques de nature diverse ; souvent x représentera un vecteur d'un espace à n dimensions et F désignera une fonction de cet espace dans $\mathbb{R}$. L'ensemble X est généralement déterminé par des contraintes souvent exprimées comme des inégalités. Suivant la nature de l'ensemble X et de la fonction F, le POC peut être facile (résoluble en temps polynomial) ou difficile. Par exemple, si X est un polyèdre convexe $\subseteq \mathbb{R}^n$ et F une fonction linéaire, on retrouve un programme linéaire (PL) résoluble efficacement par le simplexe ou même par d'autres algorithmes polynomiaux. Par contre si $X \subseteq \mathbb{Z}^n$, on retrouve des programmes en nombre entiers qui sont des problèmes difficiles ; aucun algorithme polynomial n'a été trouvé à l'heure actuelle pour les résoudre.

2. NOTION DE THEORIE DE LA COMPLEXITE

Les problèmes d'optimisation combinatoire se répartissent en 2 catégories : ceux résolus "optimalement" par des algorithmes efficaces (de complexité polynomiale), et ceux dont la résolution optimale peut prendre un temps exponentiel sur les grands cas. La théorie de la complexité permet d'évaluer et de classer les divers algorithmes disponibles pour un problème, et permet de mieux comprendre pourquoi certains problèmes ne disposent toujours pas d'algorithmes efficaces.

2.1. Complexité des algorithmes

La complexité d'un algorithme A est une fonction $C_A(N)$ donnant le nombre d'instructions exécutées par A dans le pire des cas pour une donnée de taille N. La complexité est basée généralement sur le pire cas afin de borner le temps d'exécution de l'algorithme. De plus la connaissance du pire cas est critique pour les applications temps réel (contrôle aérien, robots, systèmes d'armes, etc.). Néanmoins, la complexité pire cas peut fausser notre vision sur l'efficacité d'un algorithme. Le simplexe en est l'exemple type. Malgré sa complexité pire cas exponentielle, cet algorithme est efficace pour la plupart des PLs.

On distingue en général les algorithmes polynomiaux (complexité de l'ordre d'un polynôme), et les autres, dits exponentiels. Des exemples de complexités polynomiales sont $\log(n), n^{0.5}, nlog(n), n^2$, … Une complexité exponentielle peut être une vraie exponentielle au sens mathématique (e^n , 2^n) mais aussi des fonctions comme n^{logn} (sous exponentielle), la fonction factorielle $n!$ ou pire encore n^n.

Un bon algorithme est polynomial. Ce critère d'efficacité est confirmé par la pratique :

- Une exponentielle dépasse tout polynôme pour n assez grand. Par exemple, 1.1^n croit d'abord lentement, mais finit par dépasser n^{100}. En plus il est rare de trouver des complexités polynomiales d'ordre supérieur à n^5 en pratique.
- L'ensemble des polynômes a d'intéressantes propriétés de fermeture. L'addition, la multiplication et la composition de polynômes donnent des polynômes. On peut ainsi constituer de grands algorithmes polynomiaux à partir de plus petits.

L'esprit humain a du mal à appréhender ce qu'est une croissance exponentielle. Le tableau suivant illustre ces croissances, par rapport à des croissances polynomiales. Les temps de calculs supposent 0.1 nanoseconde par instruction (équivalent à un processeur cadencé à 10 GHZ). Les cases non remplies ont des durées supérieures à 1000 milliards d'années (supérieur à l'âge estimé de l'univers !).

Taille Complexité	20	50	100	200	500	1000
$10^7\ n\ log_2\ n$	0.09 s	0.3 s	0.7 s	1.5 s	4.5 s	10 s
$10^6\ n^2$	0.04 s	0.25 s	1 s	4 s	25 s	100 s
$10^5\ n^3$	0.08 s	1.25 s	10 s	80 s	21 min	2.7 h
$n^{\log n}$	0.04 ms	0.4 s	32 min	1.2 ans	5. 10^7 ans	--
2^n	100 µs	31 h	--	--	--	--
$n!$	7.7 ans	--	--	--	--	--

Ces calculs montrent qu'il est faux de croire qu'un ordinateur peut résoudre tous les problèmes combinatoires. « L'explosion combinatoire » est telle que l'augmentation de la puissance des ordinateurs ne change pas fondamentalement la situation. En particulier, il faut se méfier des méthodes énumératives, consistant à examiner tous les cas possibles, puisqu'elles conduisent à des algorithmes exponentiels.

2.2. Complexité des POC

Nous allons passer en revue la complexité de certains POC classiques, d'abord ceux résolus par des algorithmes polynomiaux, puis ceux qui n'ont été résolus que par des algorithmes de complexité exponentielle.

2.2.1 Problèmes faciles (polynomiaux)

- *Chemin de coût minimal (min-cost path, shortest path)*

Donnée : $G(X, U, C)$ un graphe orienté valué et deux sommets s et t de X.
But : Trouver un chemin de coût minimal entre s et t

Ce problème apparaît dans les transports, mais également en sous problème dans de nombreux problèmes de graphes. Si G et C sont quelconques, on dispose de l'algorithme de Bellman en $O(NM)$. L'algorithme de Dijkstra, en $O(N^2)$, est applicable quand les coûts de C sont positifs.

- *Flot maximal (max-flow)*

Donnée : Un réseau de transport $G(X, U, C, s, t)$.
But : Maximiser le débit du flot Φ.

Les problèmes de flots servent pour les problèmes de transport en général (de transport des fluides dans des réseaux divers par exemple) et sont aussi utilisés comme outils dans des problèmes d'ordonnancement. L'algorithme de Ford-Fulkerson permet de résoudre ce problème en $O(NM^2)$.

- *Arbre recouvrant de poids minimal (Minimum spanning tree)*

Donnée : $G(X, U, C)$ un graphe non orienté valué.
But : Trouver un arbre recouvrant de poids minimal.

On utilise ce modèle pour construire des réseaux optimaux (lignes à haute tension, oléoducs,…). On dispose de l'algorithme de Prim en $O(N^2)$ et celui de Kruskal, en $O(Mlog_2N)$.

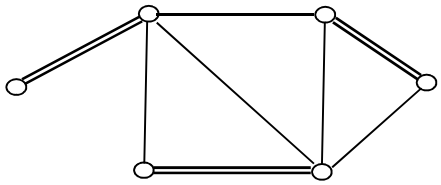
- *Test de planarité (planarity testing)*

Donnée : Un graphe simple $G(X, U)$.
Question : G est il planaire ? (C'est-à-dire qu'on peut dessiner dans le plan sans croisement d'arêtes).

Ce problème est important dans la conception de circuits électroniques, où il faut implanter un circuit donné par un schéma théorique en un nombre réduit de cartes de circuits. Il existe un algorithme compliqué en $O(N)$, dû à Hopcroft et Tarjan, et quelques algorithmes plus simples mais moins efficaces.

- *Problème de couplages (matching)*

Soit $G(X, U)$ un graphe simple. Un couplage est un sous-ensemble d'arêtes tel que deux quelconques d'entre elles n'aient aucun sommet commun. Un couplage est parfait si ses arêtes contiennent tous les sommets. Un couplage parfait a un cardinal maximal. Voici un exemple de couplage parfait (cardinal = 3) :



Donnée : Un graphe non orienté $G(X, U)$.
But : Trouver un couplage de cardinal maximal.

Les couplages modélisent des problèmes d'appariement et d'affectation. Par exemple, des relations de compatibilité entre personnes peuvent être codées par un graphe où une arête joint deux sommets (personnes) si ces personnes peuvent travailler en équipe. La recherche d'un nombre maximal de binômes compatibles est un problème de couplage. Un problème de couplage peut être converti en problème de flots d'où une complexité similaire.

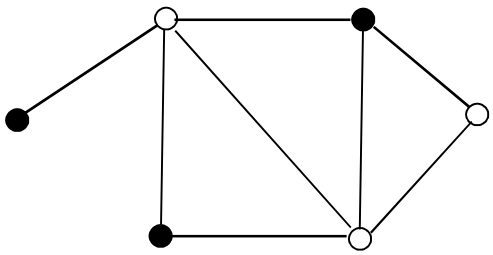
- *Programmation linéaire (linear programming)*

La PL est un outil de modélisation mathématique très puissant, utilisé, par exemple dans les industries de process (aliments de bétails, raffineries de pétrole,…) et en planification (exploitation forestière, gestion de production,…). Ce type de problèmes est résoluble par l'algorithme du simplexe. Cet algorithme consiste à énumérer intelligemment les solutions de base qui correspondent aux sommets du polyèdre défini par les contraintes. L'algorithme du simplexe est excellent en moyenne mais Minty a construit des cas où l'algorithme visite presque toutes les solutions de base (de l'ordre de C_n^m). C'est donc un algorithme exponentiel. Malgré cela, la PL est classée parmi les problèmes faciles, vu l'efficacité du simplexe. En plus, d'autres algorithmes sophistiquées mais de complexité polynomiale ont été trouvés[1].

2.2.2 Problèmes difficiles (sans algorithmes polynomiaux connus)

- *Stable maximal (max independent set)*

Soit un graphe simple $G(X, U)$. Un sous ensemble de sommets S est un stable s'il n'y a pas d'arête entre deux sommets quelconques de S. Le but est de trouver un stable de G de cardinal maximal. Voici un stable de cardinal 3 :

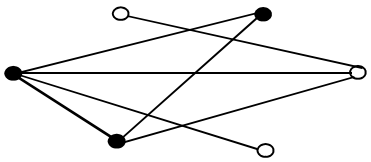


Supposons qu'on veuille organiser un nombre de matchs en parallèle, dans un tournoi sportif. On donne un graphe G dont les sommets sont les matchs. Une arête entre deux matchs signifie qu'ils ne peuvent avoir lieu en même temps. La solution au problème est précisément un stable maximal de G.

[1] En pratique, le simplexe reste plus efficace que ces algorithmes dans bon nombre de cas !

- *Clique maximale (max clique)*

Dans un graphe simple $G(X, U)$, un sous-ensemble de sommets C est une clique si C engendre un sous graphe complet de G. Ce problème consiste donc à trouver une clique de G ayant le maximum de sommets. Remarquons qu'un stable maximal est en fait une clique dans le graphe complémentaire.



- *Problème du Voyageur de Commerce (Travelling Salesman Problem)*

Le problème d'existence d'un parcours hamiltonien dans un graphe est déjà difficile. Le fameux Problème du Voyageur de Commerce (PVC ou TSP) consiste à trouver un circuit ou cycle hamiltonien, de coût minimal, dans un graphe valué complet. Il modélise un représentant qui doit partir de chez lui, passer une seule fois par un ensemble de villes, et revenir chez lui en minimisant la distance parcourue.
Le PVC est l'exemple type du problème d'optimisation combinatoire coriace, qui résiste à l'assaut des chercheurs. Il est fréquent dans les transports et a des applications surprenantes en ordonnancement.

- ***Problème de satisfiabilité (en abrégé SAT)***

Donnée : n variables booléennes x_i et un ensemble de m clauses C_j (pex : $x_1 \vee \overline{x_3} \vee x_7$).
Question : Peut-on affecter à chaque variable une valeur de façon à rendre vraies toutes les clauses ?

Ce problème théorique est à la base de la théorie de la complexité et est crucial pour l'informatique théorique et l'intelligence artificielle. Par exemple, dans les systèmes experts, le problème de savoir si un but donné peut être trouvé en partant d'un certain nombre de faits peut se ramener à SAT.

- *Problème du sac à dos (Knapsack problem)*

Le terme sac à dos regroupe plusieurs problèmes difficiles pouvant se modéliser par le programme linéaire en nombre entiers (PLNE) suivant :

$$\begin{cases} \max c.x \\ a.x \leq b \end{cases}$$ (où a, c et x sont des n-vecteurs d'entiers et b un entier)

n objets sont disponibles, l'objet i ayant un poids a_i et une valeur c_i. Chaque objet est disponible en un nombre non limité d'exemplaires. On cherche à déterminer les x_i , nombre d'exemplaires à emporter de chaque objet, pour maximiser la valeur totale sans dépasser un poids maximal b.

- *Problème de placement (Bin packing)*

On donne n objets de « poids » a_i et un nombre non limité de « boîtes » de capacité b. Le but est de répartir les objets en un nombre minimal de boîtes. Par exemple, le problème consistant à sauvegarder des fichiers en utilisant un nombre minimal de disquettes est un problème de bin packing.

2.3 Classes de complexité

La théorie de la complexité (TC) a été développée vers 1970 pour répondre à une question fondamentale : *Existe-t-il réellement une classe de problèmes combinatoires pour lesquels on ne trouvera jamais d'algorithmes polynomiaux, ou est-ce que les problèmes difficiles ont en fait de tels algorithmes, mais non encore découverts ?* L'un des principaux résultats de la TC est que tous les problèmes difficiles sont liés : la découverte d'un algorithme polynomial pour un seul d'entre eux permettrait de déduire des algorithmes polynomiaux pour tous les autres.

Vu que la TC est basée sur des outils de logique mathématique, elle ne traite que des problèmes d'existence (PE) ; ce sont des problèmes dont la réponse est Oui-Non (tels que SAT, test de planarité). Ceci n'est pas trop gênant pour les POC, puisqu'on peut associer à chaque POC un PE de la manière suivante :

Problème d'optimisation combinatoire	⇔	Problème d'existence associé
$\min_{x \in X} F(x)$		Etant donné k, existe-t-il x tel que $F(x) \leq k$?
On cherche une solution de coût minimal		On cherche une solution de coût au plus k

Un POC est donc au moins aussi difficile que son problème d'existence. Par conséquent, tout résultat établissant la difficulté du PE concernera à fortiori le POC correspondant.

- **Classes P et NP**

L'ensemble des problèmes d'existence (PE) qui admettent des algorithmes polynomiaux forme **la classe P**. Cette classe inclut tous les PE associés aux POC faciles vu précédemment.

La **classe NP** (Non-déterministe Polynomial) est celle des PE dont une proposition de solution « Oui » est vérifiable en un temps polynomial. Prenons par exemple un des problèmes difficiles à savoir le problème du sac à dos :

POC	⇔	PE
$\begin{cases} \max c.x \\ a.x \leq b \end{cases}$ et x entier		Etant donné k, existe-t-il x tel que $a.x \leq b$ et $c.x \geq k$?

Considérons un n-vecteur x solution du PE. L'algorithme de vérification consiste à vérifier les 2 inégalités $a.x \leq b$ et $c.x \geq k$. Cette vérification est possible en O(n) donc en un temps polynomial. Le problème est donc dans NP (mais n'est pas dans P)[2].

- **Problèmes NP-complets**

Il s'agit des problèmes les plus difficiles de NP. La notion de problème NP-complet est basée sur celle de la transformation polynomiale d'un problème. Par exemple, nous avons vu qu'un stable d'un graphe G était une clique dans le graphe complémentaire G_C. Supposons qu'on ait un algorithme efficace A pour trouver une clique maximale et qu'on veuille savoir si G contient un stable maximal. Ce problème peut être résolu par transformation polynomiale en un problème de clique maximale dans G_C. La transformation consiste à construire G_C qui est possible en $O(N^2)$.

Les problèmes NP-complets représentent le noyau dur de NP, car si on trouvait un algorithme polynomial pour un seul problème NP-complet X, on pourrait en déduire un autre pour tout autre problème difficile Y de NP. En effet, ce dernier algorithme consisterait à transformer polynomialement les données pour Y en données pour X, puis à exécuter l'algorithme pour X.

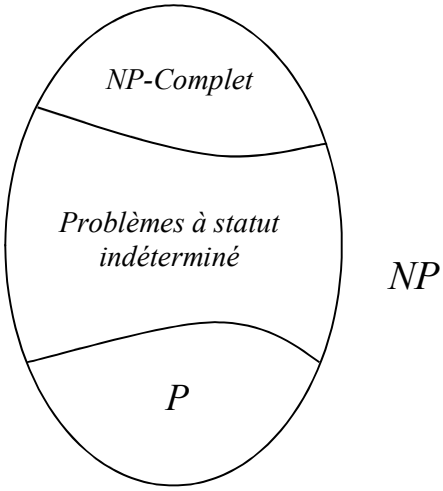
Depuis que le logicien Cook a montré en 1970 que le problème de satisfiabilité (SAT) est NP-complet, des chercheurs ont montré que la plupart des PE associés aux POC difficiles (sans algorithme polynomial) sont en fait NP-complets.

- **Conjecture $P \neq NP$**

La question cruciale de la TC est de savoir si les problèmes NP-complets peuvent être résolus polynomialement, auquel cas P = NP, ou bien si on ne leur trouvera jamais d'algorithmes polynomiaux, auquel cas $P \neq NP$. Cette question n'est pas définitivement tranchée. La découverte d'un algorithme polynomial pour un seul problème NP-complet permettrait de les résoudre facilement tous. Comme des centaines de problèmes NP-complets résistent en bloc à l'assaut des chercheurs, on conjecture aujourd'hui que $P \neq NP$.

[2] Tout PE de la classe P est bien évidemment dans NP, puisque l'algorithme de vérification de la solution possède au plus la même complexité que l'algorithme qui a permis d'obtenir cette solution.

Le schéma suivant résume bien cette conjecture. Les problèmes à statut indéterminé sont des problèmes qui n'ont pas d'algorithme polynomial connu, mais personne n'a réussi à montrer qu'ils sont NP-complets (problème de l'isomorphisme de 2 graphes par exemple).



- **Méthodes exactes & méthodes approchées**

Les méthodes exactes permettent de chercher une solution optimale d'un problème donné dans un intervalle de temps bien déterminé. Le principe essentiel d'une méthode exacte et d'énumérer de manière intelligente l'ensemble des solutions de l'espace de recherche. Bien évidemment, les méthodes exactes actuelles ont toutes une complexité exponentielle sur les problèmes NP-complets. Parmi ces méthodes on retrouve les algorithmes relatifs aux graphes, simplexe, branch & bound, programmation dynamique, relaxation lagrangienne…

Face au caractère intraitable de certains problèmes, et vu la nécessité de fournir des solutions raisonnablement bonnes de problèmes pratiques en des temps raisonnables, se sont développées des méthodes approchées appelées heuristiques[3]. Ces algorithmes sont généralement spécifiques à chaque problème et leur performance est évaluée empiriquement par comparaison avec d'autres approches. D'autres heuristiques sont conçues comme une méthode générale qui peut être adaptée à divers problèmes d'optimisation ; elles sont désignées sous le terme « méta-heuristique ».

- **Méthodes exactes ou heuristiques ?**

Apprendre ou démontrer qu'un problème est NP-complet sonne, pour bon nombre d'informaticiens, le glas des méthodes exactes et oriente immédiatement et exclusivement vers des algorithmes fournissant des solutions proches de l'optimum. Il faut bien constater, cependant que des progrès considérables ont été accomplis en RO dans la résolution exacte de problèmes NP-difficiles particuliers. C'est le cas pour le problème du voyageur de commerce (PVC/TSP), des méthodes sophistiquées permettent de trouver une solution optimale pour des instances qui comportent des centaines et même des milliers de villes. Ce genre de résultat devrait faire comprendre que ces méthodes peuvent avoir de l'intérêt, d'autant plus que la théorie de la complexité est une analyse du pire cas. Dans l'autre direction, et tant que $P \neq NP$, les techniques heuristiques et méta-heuristiques ont acquis le droit d'être citées en RO ; ils font désormais partie des outils standard de résolution des POC les plus difficiles.

[3] Le mot « heuristique » vient du grec heurein (découvrir) et qualifie tout ce qui sert à la découverte, l'invention et à recherche.

3. MODELISATION DES PROBLEMES COMBINATOIRES

3.1 Techniques de modélisation

La résolution (exacte ou approchée) des problèmes d'optimisation combinatoires (POC) nécessite une modélisation efficace et adaptée.

- L'outil le plus performant de formulation d'une multitude de problèmes (Sac à dos, PVC, …) est incontestablement les variables binaires 0-1. Elles permettent de représenter le fait qu'un événement se réalise ou non au sein d'une solution qu'on recherche (x = 1 si l'événement se réalise, 0 dans le cas contraire). Le modèle ainsi obtenu permet de représenter diverses structures dans les graphes telles que des chemins, des cycles, des arbres, etc. La formulation conduit à un Programme Linéaire en Nombre Entiers à variables binaires 0-1 (PLNE0-1 / 0-1ILP) dont voici la structure algébrique :

$$\begin{cases} \text{Min ou Max } Z = \sum_{j=1}^{n} c_j x_j \\ \sum_{j=1}^{n} a_{ij} x_j \quad (\leq, =, \geq) \quad b_j \qquad \forall i \qquad \text{(PLNE 0-1)} \\ x_j \in \{0, 1\} \end{cases}$$

- D'autres problèmes nécessitent pour leur modélisation, l'utilisation de variables entières (problème de découpe, sac à dos entier,…), on obtient un Programme Linéaire en Nombre Entiers (PLNE / ILP) :

$$\begin{cases} \text{Min ou Max } Z = \sum_{j=1}^{n} c_j x_j \\ \sum_{j=1}^{n} a_{ij} x_j \quad (\leq, =, \geq) \quad b_j \qquad \forall i \qquad \text{(PLNE)} \\ l_j \leq x_j \leq u_j \\ x_j \quad \text{entier} \end{cases}$$

Ce type de problèmes (PLNE, PLNE0-1) représente des problèmes NP-difficiles, puisque la propriété de convexité du domaine des solutions n'est plus valable. En plus, les PLNE0-1 présentent une autre difficulté qui réside dans le grand nombre de variables nécessaires pour modéliser des situations réelles.

- Certains problèmes modélisés sous forme de graphes sont résolus par des algorithmes spécifiques (plus court chemin, ARPM, max-flow,…)

- D'autres problèmes sont modélisés par des contraintes non linéaires et/ou une fonction objectif non linéaire (Problème d'affectation quadratique, Support Vector Machine, …). On obtient des Programmes Non Linéaires (PNL). Les formes les plus fréquentes des PNL sont $x_i.x_j$ ou x_i^2.

3.2 POC classiques

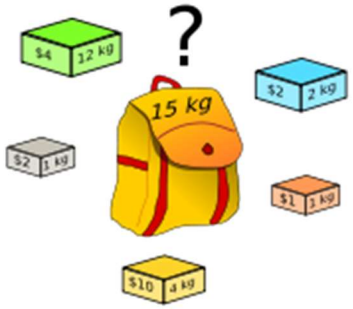
Nous allons passer en revue dans ce qui suit, la modélisation de certains POC connus et qui trouvent leur application dans des domaines très variés.

3.2.1 Problème du sac à dos (Knapsack problem)

a- Description

Le **problème du sac à dos**, aussi noté **KP** (en anglais, Knapsack Problem) est un problème d'optimisation combinatoire. Il modélise une situation analogue au remplissage d'un sac à dos, ne pouvant supporter plus d'un certain poids, avec tout ou partie d'un ensemble d'objets ayant chacun un poids et une valeur. Les objets mis dans le sac à dos doivent maximiser la valeur totale, sans dépasser le poids maximum.

Pour justifier son nom, le problème se pose lorsqu'un randonneur au moment de préparer son périple est confronté au problème de la capacité limitée de son sac à dos. Il lui faut donc trancher pour prendre les choses dont il a le plus besoin (maximiser le profit) sans dépasser la capacité du sac à dos (15 kg).



b- Formulation

Etant donné un ensemble de n objets chacun ayant un certain poids a_j et une certaine valeur c_j, et soit b un réel qui représente la charge (poids, volume, capacité) maximale que l'on peut emporter dans un sac à dos. La formulation du problème conduit à un PLNE0-1 à une seule contrainte :

$$\begin{cases} \text{Max } Z = \sum_{j=1}^{n} c_j x_j \\ \sum_{j=1}^{n} a_j x_j \le b \\ x_j \in \{0,1\} \end{cases}$$

$x_j = 1$ signifiant que l'objet j est choisi.

c- Applications

Ce problème est utilisé pour modéliser diverses situations, quelquefois en tant que sous problème :

- La sélection d'investissement « Capital Budgeting Problem » de manière à maximiser le rendement, sans bien sûr, dépasser la somme disponible.
- Dans le chargement de bateau ou d'avion « Cargo Loading Problem » : tous les bagages à destination doivent être amenés, sans être en surcharge ;
- Dans la découpe de matériaux « Cutting Stock Problem » : pour minimiser les chutes lors de la découpe de sections de longueurs diverses dans des barres en fer.

d- Variantes

Le problème du sac à dos possède une multitude de variantes dont voici quelques-unes :

- ***Knapsack entier***
  S'il existe plusieurs objets de chaque type ; soit N_j le nombre d'objets de type j, on obtient un PLNE avec $x_j \in \{0, 1, \ldots, N_j\}$ (ou x_j entier si $N_j = \infty$). Le problème ainsi obtenu est appelé Knapsack entier.

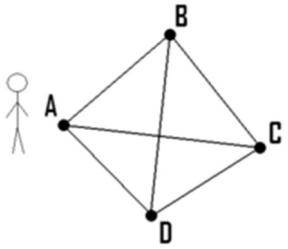
- ***Knapsack multi-dimensionnel***
  On considère ici que le sac à dos a d dimensions, avec $d > 0$ (*d-KP*). Par exemple, on peut imaginer une boîte. Chaque objet a trois dimensions, et il ne faut pas déborder sur aucune des dimensions. La contrainte est alors remplacée par d contraintes :

$$\sum_{j=1}^{n} a_j x_j^k \le b^k \qquad \forall k = \{1,\ldots,d\}.$$

3.2.2 Problème du voyageur de commerce (Travel Salesman Problem)

a- Description

L'énoncé du *problème du voyageur de commerce* PVC (ou TSP en anglais) est le suivant : étant donné *n* points (des villes) et les distances séparant chaque point, trouver un chemin de longueur totale minimale qui passe exactement une fois par chaque point et revienne au point de départ. Voici un exemple simple :



Si un voyageur part du point A et que les distances entre toutes les villes sont connues, quel est le plus court chemin pour visiter tous les points et revenir au point A ?

Ce problème est plus compliqué qu'il n'y paraît ; on ne connaît pas de méthode de résolution permettant d'obtenir des solutions exactes en un temps raisonnable pour de grandes instances (grand nombre de villes) du problème. Pour ces grandes instances, on devra donc souvent se contenter de solutions *approchées*, car on se retrouve face à une explosion combinatoire. Le nombre de chemins hamiltoniens étant égal[4] à (n–1)!/2.

b- Formulation

Soit *n* villes et c_{ij} le coût (ou la distance) correspondant au trajet i – j. Le problème consiste à déterminer un tour ou circuit hamiltonien, c'est-à-dire ne passant qu'une et une seule fois par les *n* villes, et qui soit de coût minimum. Soit la variable x_{ij} qui vaut 1 si le tour contient le trajet i – j, et 0 autrement. Le problème s'écrit :

$$\begin{cases} \text{Min } Z = \sum_{i=1}^{n}\sum_{j=1}^{n} c_{ij}x_{ij} \\ \sum_{j=1}^{n} x_{ij} = 1 \qquad \forall i \qquad (1) \\ \sum_{i=1}^{n} x_{ij} = 1 \qquad \forall j \qquad (2) \\ \sum_{i\in Q}\sum_{j\in \overline{Q}} x_{ij} \geq 1 \qquad \forall Q \qquad (3) \\ x_{ij} \in \{0,1\} \qquad \forall i, \forall j \end{cases} \qquad \text{(PVC/TSP)}$$

Où Q représente un sous-ensemble de {1,…,n} et $\overline{Q}$ son complémentaire. Les contraintes (3) expriment que la permutation des *n* villes doit être un tour, c'est-à-dire qu'il ne peut pas exister de sous-tour.

c- Applications

Le PVC fournit un exemple d'étude d'un problème NP-complet dont les méthodes de résolution peuvent s'appliquer à d'autres problèmes de mathématiques discrète. Néanmoins, il a aussi des applications directes, notamment dans les transports, les réseaux et la logistique. Par exemple, trouver le chemin le plus court pour les bus de ramassage scolaire ou, dans l'industrie, pour trouver la plus courte distance que devra parcourir le bras mécanique d'une machine pour percer les trous d'un circuit imprimé.

3.2.3 Problème d'implantation de dépôts (Facility Location Problem)

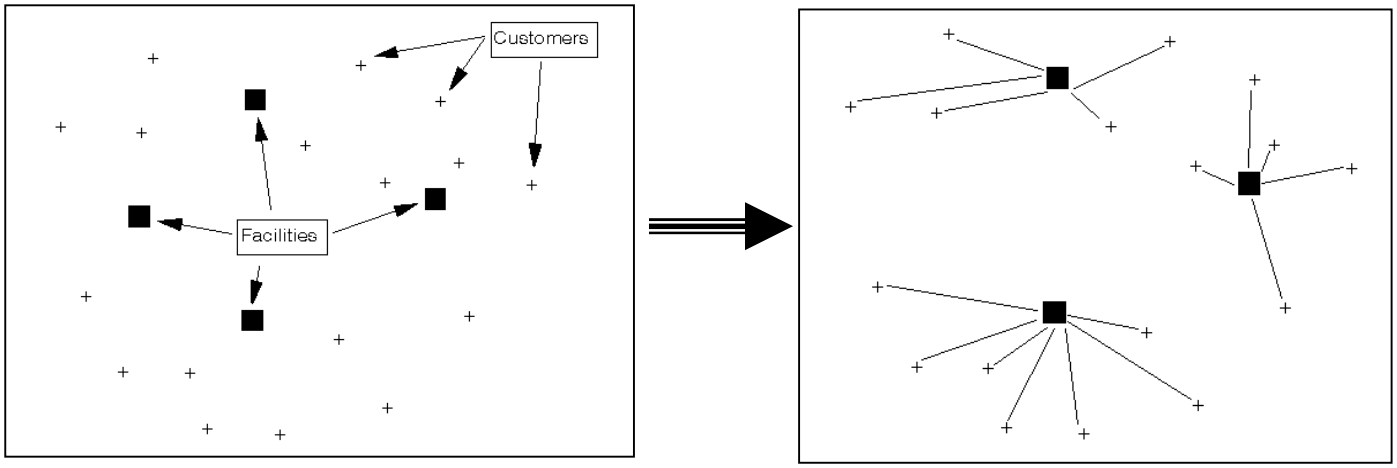
a- Description

Le problème d'implantation de dépôts (ou équipement) se pose comme suit :
Etant donné un ensemble de sites potentiels pour l'implantation de dépôts, et un ensemble de clients devant être servis par les dépôts installés, le problème consiste à déterminer :

- Quels sites potentiels seront utilisés pour l'installation des dépôts (équipement) ?
- Comment affecter les clients aux dépôts installés ?

[4] Cas de TSP symétrique (distance entre deux points A et B est égale à celle entre B et A)

Le schéma suivant montre le problème posé (Facilities = Equipement ou dépôt et Customers = Clients), ainsi qu'une solution possible au problème.



b- Formulation

Le problème d'implantation de dépôts existe en deux variantes suivant que les dépôts ont des capacités limitées ou non. Dans le premier cas, le dépôt ne peut satisfaire qu'un nombre limité de clients.

- Sans contraintes de capacité (Uncapacited FLP)

Soit :

- J un ensemble de sites potentiels ; f_j, $j \in J$, représente le coût fixe d'ouverture d'un dépôt.
- I un ensemble de clients ; c_{ij}, $i \in I$, $j \in J$, représente le coût de service du client i par le dépôt ouvert en j.

Le modèle correspondant utilise deux types de variables de décision :

$x_j = 1$ ou 0 selon qu'un dépôt est installé en j ou non.

$y_{ij} = 1$ ou 0 selon que le client i est approvisionné (servi) par le dépôt installé en j ou non.

Le problème se formule comme suit :

$$\begin{cases} \min Z = \sum_{j\in J} f_j x_j + \sum_{i\in I}\sum_{j\in J} c_{ij} y_{ij} \\ \sum_{j\in J} y_{ij} = 1 \qquad \forall i \in I \qquad (1) \\ y_{ij} \le x_j \qquad \forall i \in I, \forall j \in J \qquad (2) \\ x_j = \{0,1\}\ \forall j \in J \qquad y_{ij} = \{0,1\}\ \forall i \in I, \forall j \in J \end{cases} \qquad \text{(UFLP)}$$

Les contraintes (1) et (2) indiquent respectivement que tout client i doit être rattaché à un et un seul dépôt, et que tout approvisionnement d'un client i par le dépôt j ne peut se faire que si le dépôt est installé en j. ($x_j=1 \Rightarrow y_{ij}=0$ ou $1\ \forall i$; $x_j=0 \Rightarrow y_{ij}=0\ \forall i$)

- Avec contraintes de capacité (Capacited FLP)

Dans ce cas, chaque client i possède une certaine demande d_i à satisfaire, et chaque dépôt (éventuel) j possède une capacité maximale q_j. La formulation se fait avec les mêmes variables x_j que précédemment mais avec des variables $y_{ij} \ge 0$ (réelles) représentant la quantité fournie au client i à partir du dépôt j (c_{ij} est alors un coût unitaire) :

$$\begin{cases} \min Z = \sum_{j\in J} f_j x_j + \sum_{i\in I}\sum_{j\in J} c_{ij} y_{ij} \\ \sum_{j\in J} y_{ij} = d_i \qquad \forall i \in I \qquad (1) \\ \sum_{i\in I} y_{ij} \le q_j x_j \qquad \forall j \in J \qquad (2) \\ x_j = \{0,1\}\ \forall j \in J \qquad y_{ij} \ge 0\ \forall i \in I, \forall j \in J \end{cases} \qquad \text{(CFLP)}$$

c- Applications

Ce problème possède une multitude d'applications dans divers domaines tels que :

- Les Réseaux : Implantation de concentrateurs (Relais),…
- Logistique : Gestion des aéroports (bagages par exemple), Installations de points de vente,…
- Urbanisme : Choix des emplacements d'écoles, de mosquées…

3.2.4 Problèmes de couverture, de partition et d'emballage
(SET COVERING PROBLEM / SET PARTITIONING PROBLEM / SET PACKING PROBLEM)

- Problème de couverture

Soit *m* objets, $i \in I$, *n* sous-ensembles P_j de I. Une couverture de I est une collection de P_j vérifiant la condition :

$$\bigcup P_j = I$$

Soit c_j le coût associé à P_j. Le problème de la recherche d'une couverture de coût minimal consiste à sélectionner des sous-ensembles P_j de façon à ce que chaque objet apparaisse au moins une fois, et cela avec le coût global le plus petit. Si on pose $x_j = 1$ si P_j appartient à la couverture, 0 sinon, On obtient le modèle suivant :

$$\begin{cases} \min Z = \sum_{j=1}^{n} c_j x_j \\ \sum_{j=1}^{n} a_{ij} x_j \geq 1 \qquad \forall i \in I \\ x_j \in \{0,1\} \qquad j = 1,\ldots,n \end{cases} \qquad \text{(Set Covering Problem)}$$

Où $a_{ij} = 1$ si $i \in P_j$, 0 sinon.

- L'exemple d'application classique du problème de couverture est celui de l'ouverture d'un nombre minimum de magasins dans une région pour couvrir l'ensemble de la zone. Par ailleurs, des problèmes de routages, de livraisons, d'ordonnancements et d'implantation d'équipements ou d'infrastructures sont aussi des problèmes qui peuvent se modéliser en problème de couverture.

- Problème de partition

Une partition est une couverture qui vérifie les conditions supplémentaires $P_j \cap P_k = \varnothing \qquad \forall j,k$.

Le problème de la recherche d'une partition de coût minimal se formule comme suit :

$$\begin{cases} \min Z = \sum_{j=1}^{n} c_j x_j \\ \sum_{j=1}^{n} a_{ij} x_j = 1 \qquad \forall i \in I \\ x_j \in \{0,1\} \qquad j = 1,\ldots,n \end{cases} \qquad \text{(Set Partitioning Problem)}$$

- Le problème de partition se pose lorsque chaque client doit être servi par exactement un serveur, chaque citoyen doit être assigné à un seul district (organisation des votes par exemple).

- Problème d'emballage

Dans ce problème, les partitions P_j ont une valeur (ou bénéfice) plutôt qu'un coût ; il consiste à sélectionner des sous-ensembles disjoints de façon à maximiser la valeur totale :

$$\begin{cases} \max Z = \sum_{j=1}^{n} c_j x_j \\ \sum_{j=1}^{n} a_{ij} x_j \le 1 \qquad \forall i \in I \\ x_j \in \{0,1\} \qquad j = 1,\ldots,n \end{cases}$$ (Set Packing Problem)

- Le problème d'emballage se pose généralement lorsqu'on veut mettre en œuvre une production de valeur maximale dans un contexte de ressources limitées. D'autres applications incluant des problèmes d'ordonnancements où on cherche à satisfaire le plus de demandes possibles sans créer de conflits, correspondent aussi au modèle du problème d'emballage.

3.2.5 Problème de découpe (Cutting Stock Problem)

a) Description & Applications

Lorsqu'on se propose, à partir de certaines "entités" disponibles (bandes de papier, plaques d'acier,…) de disposer de "sous-entités" déterminées, en suivant une politique optimale (minimisation de la chute causée par la découpe), on rencontre certains problèmes de la PLNE, connus sous le nom de *problèmes de découpe*.

Les industries sont nombreuses à être confrontées quotidiennement à de tels problèmes (sidérurgiques, automobiles, textiles, du cuir, du papier, du bois, du verre,…), où la priorité est donnée à l'optimisation de la découpe. Ceux-ci sont également couramment rencontrés en VLSI et en système.

- Un cas particulier du problème de découpe consiste à couper une barre en des barres de dimensions inférieures. On parle alors de *problème de découpe à une dimension (CSP1-D)*.

- Plus complexe est la découpe d'un rectangle en sous rectangles tout en minimisant la chute. On parle alors de *problème de découpe à deux dimensions (CSP2-D)*.

b) Formulation

Exemple : Cet exemple est pris de l'industrie du papier. Le papier produit est délivré sous forme d'une bande continue enroulée sous forme de bobines de largeur constante. Pour répondre aux commandes client, l'entreprise doit découper ces bobines en d'autres de largeur plus petite.

Largeur bobines brutes = 3000 mm
Soit le carnet de commande suivant : * Client 1 : 240 bobines de largeur 800 mm
* Client 2 : 160 bobines de largeur 600 mm
* Client 3 : 300 bobines de largeur 850 mm

Une solution serait de découper :
- 80 bobines brutes chacune en 3 bobines de 800 mm de largeur et 1 bobine de 600 mm de largeur.
- 16 bobines brutes chacune en 5 bobines de 600 mm de largeur
- 100 bobines brutes chacune en 3 bobines de 850 mm de largeur

Ainsi, en combinant les commandes d'une certaine manière, on arrive à obtenir la combinaison optimale.

La formulation du problème de découpe nécessite la construction de l'ensemble des combinaisons possibles appelées ***plans de coupe*** ou ***activités*** :

N°Combinaison	1	2	3	4	5	6	7	8	9	10
850 mm	3	2	2	1	1	1	0	0	0	0
800 mm	0	1	0	2	1	0	3	2	1	0
600 mm	0	0	2	0	2	3	1	2	3	5
Perte	450	500	100	550	150	350	0	200	400	0

Le problème de découpe revient donc à déterminer le nombre de bobines brutes à découper et la manière dont elles doivent être découpées. Pour cela, on considère les variables x_i (i = 1,..,10) désignant le nombre de bobines brutes à découper suivant la combinaison *i*. Ainsi on déduit le PLNE suivant :

$$\begin{cases} \text{Min } Z = 450x_1 + 500x_2 + 100x_3 + 550x_4 + 150x_5 + 350x_6 + 200x_8 + 400x_9 \\ 3x_1 + 2x_2 + 2x_3 + x_4 + x_5 + x_6 \geq 300 \\ x_2 + 2x_4 + x_5 + 3x_7 + 2x_8 + x_9 \geq 240 \\ 2x_3 + 2x_5 + 3x_6 + x_7 + 2x_8 + 3x_9 + 5x_{10} \geq 160 \\ x_i \geq 0 \quad \text{et entier} \quad i = 1,\ldots,10 \end{cases}$$

Formulation unifiée (1D et 2D) :

Soit M le matériel en stock (brut) avec : M = L ; Longueur du matériel (1D)
M = (L,H) ; Longueur et largeur du matériel (2D)

Et d_i (i = 1,…,m) les *m* formats du carnets de commande à satisfaire en quantité q_i :

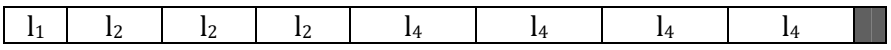
$d_i = l_i$; (1D)
$d_i = (l_i , h_i)$; (2D)

La matrice A (a_{ij}) des contraintes est construite de telle manière que chaque colonne constitue une activité (plan de coupe). Si on désigne par x_j (j = 1,…,n *n* étant le nombre d'activités) le nombre de fois que le j[ème] plan de coupe est utilisé et par c_j le coût de ce plan, on obtient le PLNE suivant :

$$\begin{cases} \text{Min } Z = \sum_{j=1}^{n} c_j x_j \\ \sum_{j=1}^{n} a_{ij} x_j \geq q_i \qquad \forall i = 1,\ldots, m \qquad \text{(CSP)} \\ x_j \geq 0 \text{ et entier} \qquad \forall j = 1,\ldots, n \end{cases}$$

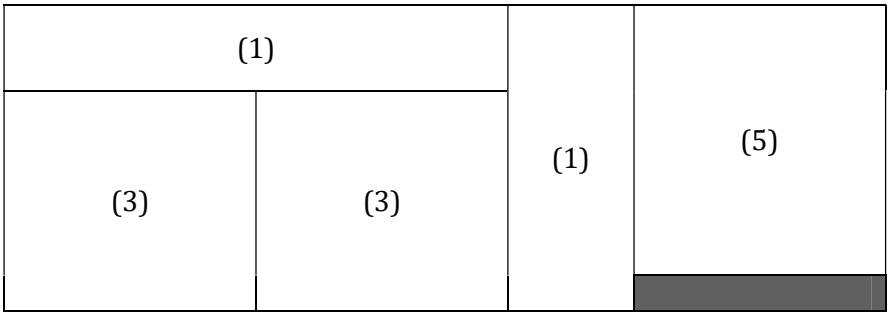
La différence entre le CSP1-D et CSP2-D se trouve au niveau des activités, c'est-à-dire la façon de découper le matériel donné. Voici un exemple :

1-D : L = 28 $l_1 = 2$, $l_2 = 3$, $l_3 = 3.5$, $l_4 = 4$



Cette activité est représentée par le vecteur (1 , 3 , 0 , 4)

2-D : (L,H) = (28,16) (l_1,h_1)=(16,5) (l_2,h_2) = (20,8) (l_3,h_3) = (11,8) (l_4,h_4) = (15,15) (l_5,h_5) = (13,7)



Cette activité est représentée par le vecteur (2 , 0 , 2 , 0 , 1)

c) Difficultés

En plus d'être un PLNE (problème NP-complet) le problème de découpe a d'autres difficultés :

- Le nombre très important d'activités, il a été montré que pour un problème moyen de 30 formats, le nombre d'activités dépasse 200 000. Ainsi, la dimension de la matrice A est de 30 x 200 000.
- Pour le cas à deux dimensions (et 3 dimensions) l'obtention d'activités réalisables représente à lui-même un problème difficile (Trim-Loss Problem). En plus d'autres contraintes dans la découpe peuvent s'ajouter telles que :
  - Découpe guillotine : c'est une découpe partageant un rectangle dans toute sa largeur ou longueur.
  - Découpe quelconque ou découpe non guillotine : en effet certaines industries disposent de moyens de coupes sophistiquées permettant une découpe quelconque.
  - Aussi, dans certains cas, les formats à placer dans l'activité ne sont pas forcément rectangulaires, mais peuvent être circulaires, triangulaires,…

4. METHODES DE RESOLUTION

On distingue deux classes de méthodes de résolution des problèmes d'optimisation combinatoire :

Les méthodes exactes : capables de trouver une solution optimale d'un problème donné dans un intervalle de temps bien déterminé mais exponentiel sur les problèmes NP-complets (notamment les PLNE). Parmi ces méthodes on retrouve la *Méthode de séparation et d'évaluation (Branch & Bound)*, la *Méthode des coupes (Cutting planes)*, la *Programmation dynamique*…

Les méthodes approchées : Ces algorithmes appelés généralement heuristiques consistent à trouver une solution proche de l'optimum en un temps raisonnable. Les heuristiques peuvent aller d'un simple algorithme de recherche locale à une classe générale d'heuristiques appelées méta-heuristiques telles que le *Recuit simulé*, la *Recherche tabou*, les *Algorithmes génétiques*…

4.1 Méthodes exactes

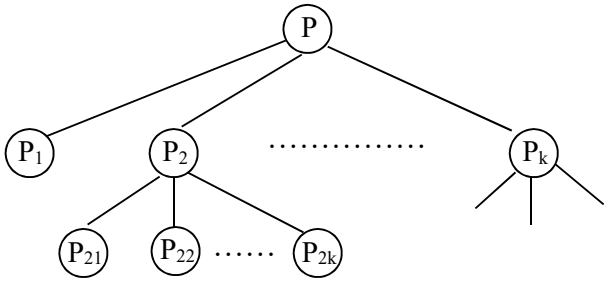
4.1.1. Procédure de séparation et d'évaluation progressive (Branch & Bound)

a) Introduction

Il s'agit essentiellement de diviser (divide and conquer) l'ensemble de toutes les solutions réalisables (problème initial) en sous-ensembles plus petits (sous problèmes) et mutuellement exclusifs. C'est la phase « séparation » (Branch). Puis divers critères sont utilisés pour identifier les sous-ensembles qui peuvent contenir la solution optimale et les sous-ensembles qui ne doivent pas être explorés plus à fond car ils ne peuvent pas contenir la solution optimale. C'est la phase « évaluation » (Bound).

Cette méthode consiste donc à faire une énumération intelligente de l'espace des solutions, puisqu'elle arrive à éliminer des solutions partielles qui ne mènent pas à la solution optimale.

Pour ce faire, cette méthode se dote d'une fonction qui permet de mettre une borne inférieure (en cas de min) sur certaines solutions pour soit les exclure soit les maintenir comme des solutions potentielles. Bien entendu, la performance d'une méthode de branch and bound dépend, entre autres, de la qualité de cette fonction (de sa capacité d'exclure des solutions partielles tôt).



Le processus de séparation d'un sous-ensemble P_i s'arrête dans l'un des cas suivants (cas Min) :

- Lorsque la borne inférieure de P_i est ≥ à la meilleure solution trouvée jusqu'à maintenant pour le problème initial.
- Lorsque P_i n'admet pas de solution réalisable.
- Lorsque P_i admet une solution complète du problème initial.

b) Algorithme général (cas Min)

A chaque instant, on maintient :

- une liste de sous problèmes actifs P_i

- Le coût Z de la meilleure solution obtenue jusqu'à maintenant (Initialisé à $+\infty$ ou à celui d'une solution initiale connue).

A une étape typique :

- Sélectionner un sous problème actif P_i

- Si P_i n'est pas réalisable, le supprimer (stop branch) sinon calculer sa borne inférieure $Z_{inf}(P_i)$

- Si $Z_{inf}(P_i) \geq Z$, supprimer P_i (stop branch)

- Si $Z_{inf}(P_i) < Z$, soit résoudre P_i directement, soit créer de nouveaux sous problèmes et les ajouter à la liste des sous problèmes actifs.

En pratique, il reste quelques petits détails à régler pour pouvoir appliquer cette méthode, en particulier :

- Comment déterminer la borne d'évaluation (borne inférieure en cas de min) ?

  Une possibilité est de calculer la solution du PLC[5] obtenue par relaxation linéaire du PLNE. Cette méthode donnera une bonne évaluation, mais pourra être coûteuse et longue à calculer. Suivant le problème on pourra essayer de trouver une méthode heuristique/astucieuse faisant un compromis entre vitesse d'obtention de la borne et sa fiabilité.

- Quel sommet explore-t-on à chaque étape de la recherche ?

  Là, il n'y a pas de bonne méthode, c'est suivant le problème en question. Les principales stratégies utilisées sont :
  - En profondeur (DFS)
  - En largeur (BFS)
  - Meilleur d'abord. (Best First)

- Suivant quel x_i construit-on l'arbre ?

  Ce choix peut être déterminant pour la rapidité de la solution optimale. Dans le cas du problème du sac à dos par exemple, il est plus efficace de prendre l'ordre des x_i par coût décroissant.

c) Exemple : Problème du sac à dos (Pb. de maximisation) – Utilisation de la stratégie Best First

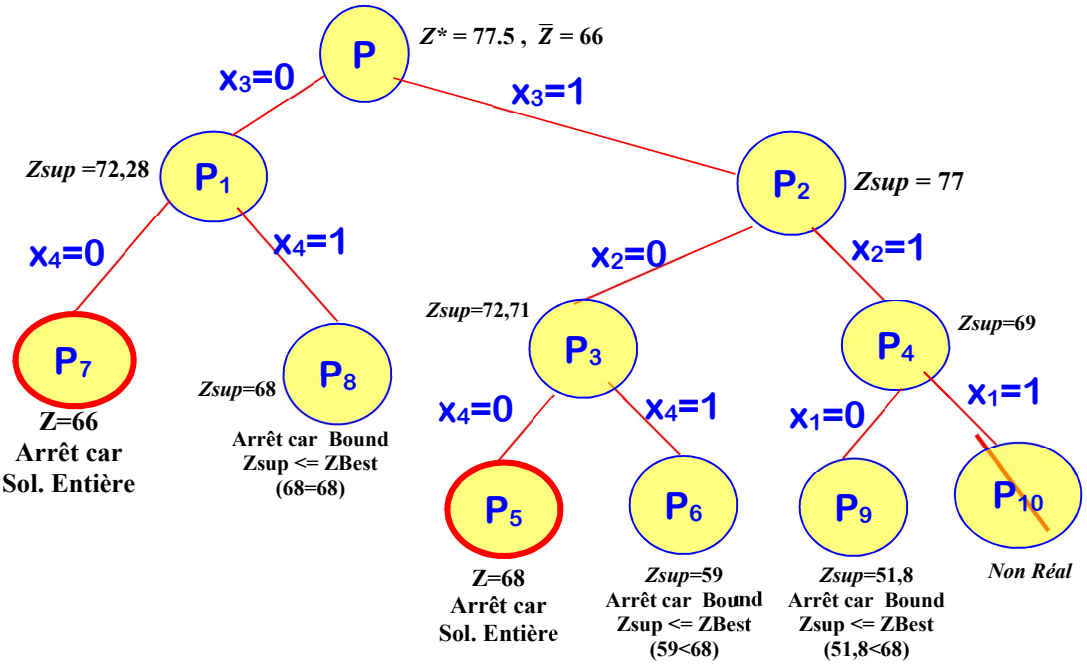
- Une société dispose de 20 MDA à investir.
- Les experts proposent 4 investissements possibles :

	Coût (MDA)	Bénéfice	Rendement
Inv. 1	9	45	5.00
Inv. 2	7	21	3.00
Inv. 3	8	23	2.87
Inv. 4	7	11	1.57

Résolution :

	a_i	c_i	P	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10
1	9	45	1	1	1	1	5/9	1	5/9	1	1	0	1
2	7	21	1	1	3/7	0	1	0	0	1	4/7	1	1
3	8	23	1/2	0	1	1	1	1	1	0	0	1	1
4	7	11	0	4/7	0	3/7	0	0	1	0	1	5/7	?

[5] PLC pour programme linéaire continu, obtenu en considérant des variables réelles au lieu d'entières.



➢ **Relaxation de P**

Solution optimale réelle (PLC) $x^* = (1 , 1 , 1/2 , 0)$ $Z^* = 77.5$

Solution arrondie $\bar{x} = (1 , 1 , 0 , 0)$ $\bar{Z} = 66$

On enregistre cette solution comme meilleure solution rencontrée : $x_{best} = \bar{x}$ et $Z_{best} = 66$

$66 \leq Z_{opt} \leq 77.5$

➢ $Z_{best} = \sout{66}$, 68

➢ La meilleure solution trouvée est donc la solution optimale (Nœud P_5), ainsi

Solution optimale : $x_{opt} = (1 , 0 , 1 , 0)$ et $Z_{opt} = 68$

4.1.2. Programmation dynamique

La Programmation Dynamique est une méthode exacte de résolution de problèmes d'optimisation, due essentiellement à R. Bellman (1957). Bien que très puissante, son cadre d'application est relativement restreint, dans la mesure où les problèmes qu'elle adresse doivent vérifier un principe dit *principe d'optimalité* qui stipule qu'**une solution optimale d'un problème de taille *n* peut s'exprimer en fonction de la solution optimale de problèmes de taille inférieure à *n***. Bien que naturel, ce principe n'est pas toujours applicable ! Prenons l'exemple des chemins dans un graphe : le principe marche si on cherche le plus court chemin entre deux points : si le chemin le plus court entre A et B passe par C, le tronçon de A à C (resp. de C à B) est le chemin le plus court de A à C (resp. de C à B) ; par contre ça ne marche plus si on cherche le plus long chemin sans boucle d'un point à un autre !

L'idée de base est donc de décomposer le problème en une suite de sous problèmes emboîtés plus petits mais généralement de même nature que le problème principal. Résoudre ensuite les sous problèmes séquentiellement en partant du plus petit, les solutions des problèmes d'une étape donnée s'obtenant à partir de celles des problèmes de l'étape précédente (voir des étapes précédentes). Le principe est d'éviter de résoudre plusieurs fois le même sous problème.

On définit une table, à chaque élément correspondra la solution d'un et d'un seul problème intermédiaire. Il faut donc qu'on puisse définir chaque sous problème qui sera traité au cours du calcul... Ensuite il faut remplir cette table. Cet algorithme peut être exprimé de manière itérative ou récursive.

Le plus gros du travail réside dans l'expression d'une solution d'un problème en fonction de celles de problèmes "plus petits" (formule de récurrence). Si on se rend compte qu'on est amené à recalculer plusieurs fois la solution de mêmes problèmes, on est dans le cadre de la programmation dynamique.

Attention, le nombre de sous problèmes peut être grand et l'algorithme obtenu n'est pas forcément polynomial.

- **Application au problème du voyageur de commerce (PVC/TSP)**

Rappelons le problème du TSP : étant donné un graphe valué G(X,U,C), le TSP consiste, en partant d'un sommet donné, de trouver un cycle hamiltonien de poids minimum.

Pour obtenir l'équation de récurrence relative au TSP (expression du TSP initial en fonction de sous problèmes plus petits) procédons comme suit (sommet de départ 1) :

Il est clair que tout cycle est constitué d'un arc $(1, i)$ et d'un chemin simple (partant de i et passant une et une seule fois par tous les sommets de $U - \{1, i\}$. Soit donc $D[i, S]$ la distance d'un plus court chemin partant de *i*, passant par tous les points de *S,* une et seule fois, et se terminant au sommet 1. La relation suivante n'est donc pas difficile à établir :

$$D[i, S] = \min_{j \in S}\{c_{ij} + D[j, S - \{j\}]\}$$

La solution optimale est donc donnée par $D[1, U-\{1\}]$.

$$D[1, U - \{1\}] = \min_{2 \le j \le n}\{c_{ij} + D[j, U - \{1, j\}]\} \quad \text{Avec } D[i,\varnothing] = c_{i1}$$

Il reste à définir la table qui permettra de résoudre les sous problèmes du TSP à partir des plus simples jusqu'au problème initial en faisant varier l'ensemble S de l'ensemble vide jusqu'à $U - \{1\}$.

Exemple :

Soit le graphe suivant :

0	2	5	8
2	0	3	4
5	3	0	1
8	4	1	0

Dans ce cas, la table est :

	{}	{2}	{3}	{4}	{2,3}	{2,4}	{3,4}
2	2	-	8	12	-	-	10
3	5	5	-	9	-	7	-
4	8	6	6	-	6	-	-

$D[2, \varnothing]=2$, $D[3, \varnothing]=5$, $D[4, \varnothing]=8$

$D[2,\{3\}] = c_{23} + D[3, \varnothing] = 3+5 = 8$, $D[2,\{4\}] = c_{24} + D[4, \varnothing] = 4+8 = 12$

$D[3,\{2\}] = c_{32} + D[2, \varnothing] = 3+2 = 5$, $D[3,\{4\}] = c_{34} + D[4, \varnothing] = 1+8 = 9$

$D[4,\{2\}] = c_{42} + D[2, \varnothing] = 4+2 = 6$, $D[4,\{3\}] = c_{43} + D[3, \varnothing] = 1+5 = 6$

$D[2,\{3,4\}]=\min\{c_{23} + D[3,\{4\}] , c_{24} + D[4,\{3\}]\} = \min\{3+9 , 4+6\} = 10$

$D[3,\{2,4\}]=\min\{c_{32} + D[2,\{4\}] , c_{34} + D[4,\{2\}]\} = \min\{3+12 , 1+6\} = 7$

$D[4,\{2,3\}]=\min\{c_{42} + D[2,\{3\}] , c_{43} + D[3,\{2\}]\} = \min\{4+8 , 1+5\} = 6$

Et $D[1, U - \{1\}] = \min\{(2 + 10), (5 + 7), (8 + 6)\} = 12$

Note : Le détail du cycle hamiltonien optimal peut être retrouvé en retenant l'indice j minimisant l'équation de récurrence détaillé précédemment.

4.2 Méthodes approchées

Les méthodes approchées sont généralement des algorithmes polynomiaux qui donnent une solution quelquefois optimale, souvent bonne. Une méthode approchée peut être déterministe - pour une entrée donnée, elle donnera toujours la même solution - (heuristiques gloutonnes, optimum local) ou non déterministe (recuit simulé, algorithme génétique,...).
Elle peut être basée sur un critère propre au problème (heuristique gloutonne) ou sur une métaheuristique (avec une notion de voisinage adapté au problème).

4.2.1 Heuristiques classiques

Ce sont des heuristiques conçues pour un problème particulier en s'appuyant sur sa structure propre. Cependant, on retrouve en général des principes de base utilisés dans ces heuristiques tels que :

- **Principe glouton** : ce principe repose sur le choix définitif des valeurs de la solution, ce qui interdit toute modification ultérieure.
- **Principe de construction progressive** : c'est une extension du principe glouton dans la mesure où l'on s'autorise, cette fois-ci, à modifier des valeurs déjà assignées. On retrouve par exemple l'algorithme de Ford pour la recherche des chemins optimaux.
- **Principe de partitionnement** : ce principe repose sur le fait que résoudre le problème global se révèle souvent plus complexe que résoudre la somme des sous problèmes qui le composent. Toute la difficulté réside alors dans le fusionnement des solutions de chaque sous problème.

Dans ce qui suit, nous détaillerons le principe glouton utilisé dans beaucoup d'algorithmes exacts ou approchés.

- **Algorithmes Gloutons (Greedy algorithms)**

C'est une méthode (heuristique ou exacte) simple et d'usage général (appelée aussi algorithme du plus proche voisin). Son principe est qu'à chaque étape durant le processus de recherche, l'option localement optimale est choisie jusqu'à trouver une solution (exacte ou non). Les choix fait durant le processus de recherche ne sont jamais remis en cause (pas de retour arrière). Dans les cas où l'algorithme ne fournit pas systématiquement la solution optimale, on parle d'heuristique gloutonne.

Plusieurs algorithmes exacts importants sont issus de cette technique : Algorithme de Dijkstra pour les plus courts chemins et l'algorithme de Kruskal pour les arbres recouvrants minimaux.

Les algorithmes gloutons ont en général une complexité polynomiale, mais encore faut-il que l'algorithme donne bien une solution optimale et qu'on sache le prouver. Là réside souvent la difficulté dans ce type d'algorithmes.

En tant qu'heuristiques, les méthodes gloutonnes sont souvent utilisées pour trouver une solution initiale à d'autres méthodes plus élaborées.

Exemple :
On veut totaliser une somme d'argent S en utilisant un nombre minimal de pièces appartenant à un ensemble donné. A chaque étape on choisit la plus grande pièce <= S tout en mettant à jour la nouvelle somme (S – la pièce choisie):
Pour S = 257 DA et pièces = {100DA, 50DA, 20DA, 10DA, 5DA, 2DA, 1DA}
à l'étape 1 : S1 = 257 , on choisit 1 pièce de 100DA
à l'étape 2 : S2 = 157 , on choisit 1 pièce de 100DA
à l'étape 3 : S3 = 57 , on choisit 1 pièce de 50DA
à l'étape 4 : S4 = 7 , on choisit 1 pièce de 5DA
à l'étape 5 : S5 = 2 , on choisit 1 pièce de 2DA

La solution trouvée est donc 2x100 DA , 1x50 DA , 1x5 DA et 1x2 DA

On peut montrer que dans ce cas, l'algorithme glouton donne toujours une solution optimale. Mais dans le système de pièces (1, 3, 4), l'algorithme glouton n'est pas optimal, comme le montre l'exemple simple suivant. Il donne pour 6 : 4+1+1, alors que 3+3 est optimal.

4.2.2. Heuristiques développées et méta-heuristiques

a) Algorithme de Descente

Une des idées les plus utilisées dans les méthodes d'optimisation est de procéder itérativement en examinant le voisinage de la solution courante, dans l'espoir d'y détecter des directions de recherche prometteuses. Pour cela, on doit disposer :

- d'une notion de voisinage qui structure l'espace X des solutions ;
- d'un moyen efficace pour trouver la meilleure (ou une bonne) solution dans le voisinage d'une solution quelconque.

Partant d'une solution initiale quelconque, on progresse alors de proche en proche, tant que l'on trouve une solution meilleure que la solution courante dans le voisinage de celle-ci. L'algorithme s'arrête lorsque la solution courante est meilleure que toutes les solutions appartenant à son voisinage – ce qui est bien la définition d'un optimum local.

- **Schéma général**

On suppose que pour toute solution $x \in X$, il est défini un ensemble $V(x) \subseteq X$ de solutions voisines de x.

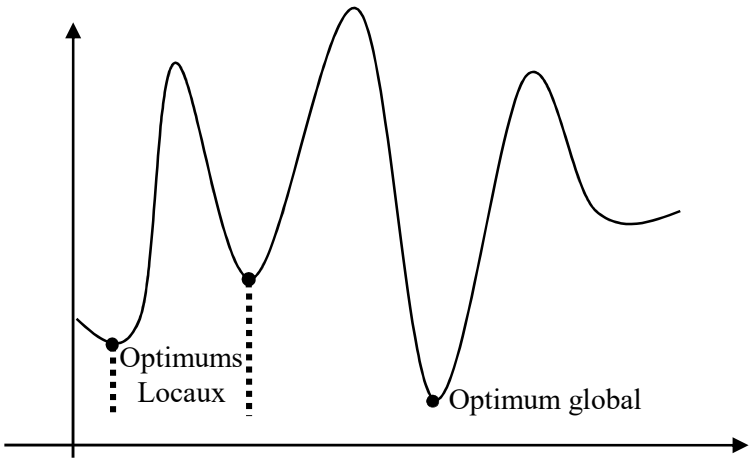
Initialisation : *Soit x_0, une solution initiale*

Etape n : *Soit $x_n \in X$, la solution courante*
Sélectionner la meilleure (ou une bonne) solution $x^* \in V(x_n)$
Si $Z(x^*) \preceq Z(x_n)$ alors $x_{n+1} = x^*$ et passer à l'étape suivante
Sinon x_n est la meilleure solution trouvée ; Arrêt.

Il est évident que le choix du voisinage peut exercer une influence déterminante sur la qualité de solutions obtenues. Illustrons la diversité des voisinages sur le problème du voyageur de commerce. Si x_i représente le nom d'une ville, une solution est décrite par un vecteur $x=(x_1, x_2, \ldots, x_N)$, N étant le nombre de villes, voici quelques définitions usuelles de voisinages de x :

- Le voisinage « 2-échange » : tous les tours qui peuvent être obtenus à partir du tour *x* en permutant deux villes quelconques dans la liste ordonnée (taille du voisinage : $|V(x)| = N(N-1)/2$).
- Le voisinage « k-opt » : tous les tours qui peuvent s'obtenir en rejetant *k* arêtes du tour courant et en reconnectant de la meilleure manière possible les tronçons ainsi formés dans le tour. Les cas particuliers les plus usuels sont 2-opt et 3-opt. Avec 2-opt la taille du voisinage est $O(N^2)$.

L'inconvénient principal des algorithmes de descente est le fait que l'algorithme s'arrête, le plus souvent, dans un optimum local et non dans un optimum global. C'est précisément, pour faire face à ce problème qu'ont été proposées des méthodes qui autorisent une détérioration temporaire de l'objectif permettant ainsi de quitter des minimums locaux tout en maintenant en général une « pression » favorisant les solutions qui améliorent l'objectif. Ces méthodes sont souvent appelées méta-heuristiques ; contrairement aux heuristiques classiques conçus pour un problème particulier, les méta-heuristiques sont des méthodes approximatives générales, pouvant s'appliquer à différents problèmes. On peut citer par exemple : le recuit simulé, les algorithmes génétiques, la recherche tabou, …



b) Recuit Simulé (Simulated Annealing)

- **Principe**

L'idée de base de cette heuristique [Kirkpatrick 1983] provient de l'opération de recuit (annealing), courante en sidérurgie et dans l'industrie du verre : après avoir fait subir des déformations au métal (par exemple après avoir mis en bobines des tôles d'acier laminées), on réchauffe celui-ci à une certaine température, de manière à faire disparaître les tensions internes causées par les déformations, puis on laisse refroidir lentement. L'énergie fournie par le réchauffement permet aux atomes de se déplacer légèrement et le refroidissement lent fige peu à peu le système dans une structure d'énergie minimale.

Cette idée se transpose assez naturellement en optimisation pour modifier l'heuristique de descente. Au lieu de ne permettre que des mouvements (des changements de solution courante) qui diminuent l'énergie (la fonction objectif), on autorise des augmentations, même importantes, de l'énergie au début de l'exécution de l'algorithme, puis, à mesure que le temps passe, on autorise ces augmentations de plus en plus rarement (la température baisse). Finalement, le système « gèle » dans un état d'énergie minimale.

Plus précisément, partant d'une solution x_n, on tire au sort une solution voisine $x^* \in V(x_n)$. Si $Z(x^*) \leq Z(x_n)$, alors x^* devient la nouvelle solution courante x_{n+1} ; si $Z(x^*)>Z(x_n)$, on calcule une probabilité qui va décider si x^* devient la nouvelle solution courante ou si x_n reste la solution courante. Dans ce dernier cas, on tirera au sort un autre voisin de x_n. La probabilité *p* d'accepter x^* décroît avec le temps de l'algorithme et est généralement une fonction $p(T,\Delta Z)$ dépendant d'un paramètre T qui est appelé « température », et de la dégradation de l'objectif $\Delta Z = Z(x^*) - Z(x_n)$.

- **Schéma général**

Initialisation : *Soit $x_0 \in X$, une solution initiale ;* $\hat{Z} = Z(x_0)$

Etape n : *Soit $x_n \in X$, la solution courante ;*
Tirer au sort une solution $x^* \in V(x_n)$;
Si $Z(x^*) < Z(x_n)$, alors $x_{n+1} = x^*$;
Si $Z(x^*) < \hat{Z}$, alors $\hat{Z} = Z(x^*)$ // Enregistrer la meilleure solution trouvée
Sinon, tirer un nombre r au hasard entre 0 et 1 ;
Si $r \leq p$, alors $x_{n+1} = x^*$ Sinon $x_{n+1} = x_n$

Deux éléments restent à préciser : la manière dont *p* dépend de T et de ΔZ et le critère d'arrêt. De nouveau, c'est l'analogie avec les systèmes physiques qui a guidé le choix d'une fonction $p(T, \Delta Z)$ et l'expérimentation dans le champ de l'optimisation combinatoire a souvent confirmé la pertinence de ce choix. C'est la distribution de Boltzmann (appelé critère de Metropolis) décrite comme suit :

$$p(T,\Delta Z) = e^{-\frac{\Delta Z}{T}}$$

Cette quantité est comprise entre 0 et 1 comme une probabilité. Le paramètre température T varie lui-même au cours de l'exécution de l'algorithme. Le profil le plus souvent adopté est une décroissance géométrique par paliers : Une température initiale T_0 est maintenue constante pendant *L* itérations, après quoi on passe, pendant *L* itérations, à une température $T_1 = \alpha\ T_0$ ($0 < \alpha < 1$), que l'on réduit à $T_2 = (\alpha)^2\ T_0$ pour *L* autres itérations, etc.

Les paramètres T_0, *L* et α doivent être déterminés expérimentalement en tenant compte de la taille du problème. Typiquement, T_0 est choisi de sorte qu'au début de l'algorithme, des solutions moins bonnes que la solution courante soient aisément acceptées. La longueur *L* du palier de température doit être déterminée en tenant compte de la taille des voisinages ; elle devrait augmenter avec la taille du problème, puisqu'il faut laisser plus de temps pour explorer l'espace des solutions quand celui-ci est vaste. Enfin, le paramètre de décroissance géométrique de la température α est fixé le plus souvent entre 0,75 à 0,95. C'est lui qui gère la lenteur du refroidissement. Un refroidissement rapide conduit le plus souvent à des optimums locaux de mauvaise qualité (équivalent de la trempe dans la sidérurgie)

Le critère d'arrêt peut tout simplement être l'allocation d'un nombre maximal d'itérations. Le plus souvent, on lui préfère un critère dynamique déterminant le moment où le système est « gelé ». Par exemple, on dira que le système est gelé si la fonction Z a diminué seulement de 0.1% pendant 100 paliers.

c) Recherche Tabou (Tabu Search)

Cette approche emprunte certains de ses concepts de l'intelligence artificielle [Glover 1986], en particulier l'utilisation de mémoire. Comme le recuit simulé, il s'agit, au moins dans sa version de base, d'une variante de la recherche locale. Partant de la solution courante x_n, on calcule la meilleure solution x* dans un sous-voisinage V* de V(x_n), sous-voisinage déterminé en fonction de l'histoire de recherche aux étapes précédentes. Cette solution devient la nouvelle solution courante x_{n+1}, qu'elle soit meilleure ou moins bonne que la précédente. Ceci est indispensable si l'on veut éviter de rester coincé dans des minimums locaux. Il faut cependant mettre en place un mécanisme qui évite le cyclage. Ceci est obtenu grâce à une liste appelée **liste tabou** (ou **restrictions tabou**), qui garde en mémoire les dernières solutions rencontrées. Ainsi, on exclut de V* les solutions récemment rencontrées.

Exemple : On considère le voisinage « 2-échange » pour le TSP composé de 5 villes A, B, C, D, et E. Supposons que la solution courante x_n soit le tour (A,B,C,D,E) et que x_{n+1} soit le tour (A,D,C,B,E), obtenu en permutant les positions des 2ème et 4ème villes du tour. Au lieu d'enregistrer, dans la liste tabou, la description complète des tours rencontrés durant les dernières étapes, seule la transformation appliquée (appelée souvent **mouvement**) est généralement enregistrée. Dans notre cas, le mouvement peut être représenté par la paire (2,4) et pendant un nombre défini d'étapes, on interdira l'exécution de l'inverse de ce mouvement (pour éviter de retourner à la même solution). Cependant la liste tabou peut écarter des solutions non rencontrées, on peut envisager de négliger le statut tabou de certaines solutions si un avantage suffisant en résulte. Ceci est implémenté à l'aide du « **critère d'aspiration** ». Par exemple, on acceptera pour x_{n+1} une solution x* tabou si celle-ci donne à la fonction objectif Z une valeur meilleure que toutes celles obtenues jusqu'à présent.

- **Schéma général**

Initialisation : *Soit $x_0 \in X$, une solution initiale, $\hat{Z} = Z(x_0)$*
k = longueur de la liste tabou ou degré de récence (prohibition period)

Etape n : *Soit $x_n \in X$, la solution courante ;*
Chercher dans un sous-voisinage V de $V(x_n)$ (appelé aussi liste des mouvements candidats) la meilleure solution x* qui soit non tabou ou bien tabou, mais satisfait le critère d'aspiration*
Faire $x_{n+1} = x^*$, Si $Z(x^*) < \hat{Z}$ alors $\hat{Z} = Z(x^*)$
Mettre à jour la liste tabou

- **Intensification et Diversification**

De manière générale, il est indiqué d'implémenter une recherche tabou comme une succession de phases d'intensification et de phases de diversification de la recherche.

- Au cours des phases d'intensification, on explore plus en profondeur le voisinage de la solution courante, par exemple en augmentant la taille de V*.
- Lors des phases de diversification, au contraire, on s'attache à constituer V* de solutions très différentes les unes des autres et très différentes de x_n ; ceci peut être obtenu, par exemple, via des listes tabou qui excluent les solutions ressemblant à celles rencontrées récemment.

d) Algorithmes Génétiques (Genetic Algorithms)

Les algorithmes génétiques [Holland 1975] ont été conçus comme un modèle de système adaptatif complexe capable de simuler l'évolution des espèces. Du point de vue algorithmique, les algorithmes génétiques se distinguent aussi bien du recuit simulé que de la recherche tabou par le fait qu'ils traitent et font évoluer une population de solutions, et non une seule solution. De plus, au cours d'une itération, les solutions de la population courante n'évoluent pas indépendamment les unes des autres, mais interagissent pour fournir la génération suivante.

- **Description d'un algorithme génétique de base**

Les solutions sont codées de manière appropriée. Un codage élémentaire pour un PLNE0/1 est un vecteur *x* de 0 et de 1. Pour le TSP, un codage approprié sera une liste ordonnée des noms des N villes. Un vecteur codant une solution est souvent appelé « **chromosome** » et ses coordonnées (noms des villes) sont appelées « **gènes** ». Le choix d'un codage approprié est très important pour l'efficacité des opérateurs appliqués pour faire évoluer les solutions.

Une population initiale de solutions $X^{(0)}$ est constituée. Une fonction d'évaluation des solutions est également choisie. Traditionnellement, cette fonction notée F est croissante avec la qualité de la solution (on parle de fitness function, mesurant la « santé » de l'individu solution), ce peut être l'opposé de la fonction objectif (ou la fonction objectif en cas de max), mais pour des raisons d'efficacité, on est amené souvent à choisir une fonction fitness plus sophistiquée que la fonction objectif.

Sur la base de la fonction fitness, une génération suivante est créée en appliquant des opérateurs de « sélection », de « croisement » et de « mutation ».

- **Schéma de base**

Initialisation : Soit $X^{(0)} \subseteq X$, une population initiale

Etape n : Soit $X^{(n)} \subseteq X$, la population courante
- sélectionner dans $X^{(n)}$ un ensemble de paires de solutions de haute qualité

- appliquer à chacune des paires de solutions sélectionnées un opérateur de « croisement » qui produit une ou plusieurs solutions « enfants ».

- remplacer une partie de $X^{(n)}$ formée de solutions de basse qualité par des solutions « enfants » de haute qualité.

- appliquer un opérateur de mutation aux solutions ainsi obtenues ; les solutions éventuellement mutées constituent la population $X^{(n+1)}$.

- **Opérateurs utilisés**

a. Sélection

La sélection – aussi bien celle des individus de haute qualité que celle des individus de basse qualité – comporte généralement un aspect aléatoire. Chaque individu x_i de la population parmi laquelle se fait la sélection se voit attribuer une probabilité p_i d'être choisi d'autant plus grande que son évaluation est hausse (basse dans le cas d'une sélection de mauvais individus). On tire au sort un nombre *r* au hasard entre 0 et 1. L'individu *k* est choisi tel que :

$$\sum_{i=1}^{k-1} p_i < r \leq \sum_{i=1}^{k} p_i$$

La probabilité que x_k soit choisi est ainsi bien égale à p_k.

b. Croisement

Soit deux solutions *x* et *y* sélectionnées parmi les solutions de haute qualité. Un opérateur de croisement (crossover) fabrique une ou deux nouvelles solutions *x'*, *y'* en combinant *x* et *y*. Par exemple, si *x* et *y* sont deux vecteurs de 0 et 1, un opérateur de croisement classique consiste à sélectionner aléatoirement deux positions dans les vecteurs et à permuter les séquences de 0 et de 1 figurant entre ces deux positions dans les deux vecteurs. A titre d'exemple, pour des vecteurs à huit composantes :

x = 0 1 1 0 1 1 0 0
y = 1 1 0 0 1 0 1 0

si les positions « après 2 » et « après 5 » sont choisies, on obtient, après croisement :

x' = 0 1 0 0 1 1 0 0
y' = 1 1 1 0 1 0 1 0

L'opérateur de croisement est utilisé pour favoriser la transmission des « bonnes sous-structures » des solutions parents aux enfants.

c. Mutation

Une mutation est une perturbation introduite pour modifier une solution individuelle, par exemple la transformation d'un 0 en un 1 ou inversement dans un vecteur binaire. Dans le TSP, une mutation peut être une permutation arbitraire de deux villes. En général, cet opérateur est appliqué avec une probabilité assez faible. Un but possible de la mutation est d'introduire un élément de diversification, d'innovation comme dans la théorie de l'évolution des espèces.

PARTIE 2 : PROGRAMMATION QUADRATIQUE

1. INTRODUCTION

1.1. Enoncé

Considérons le programme non linéaire (PNL) suivant :

$$\begin{cases} \boldsymbol{min}\, \boldsymbol{f(x)} \\ \boldsymbol{g_i(x) \leq 0} \quad \boldsymbol{i = 1, \dots, m} \\ \boldsymbol{h_j(x) = 0} \quad \boldsymbol{j = 1, \dots, \ell} \\ \boldsymbol{x \in X} \end{cases}$$

où $f, g_1, \dots, g_m$ et $h_1, \dots, h_\ell$ sont des fonctions du paramètre $x \in \mathbb{R}^n$ qui est un vecteur à n composantes $x_1, \dots, x_n$. L'ensemble X est un sous ensemble de $\mathbb{R}^n$ et peut typiquement inclure des bornes inférieures et/ou supérieures sur les variables, qui peuvent jouer un rôle utile dans certains algorithmes de résolution. Cet ensemble pourrait aussi représenter certaines régions particulières ou d'autres contraintes complexes qui doivent être traitées séparément par l'intermédiaire d'un mécanisme spécial.

Pour illustrer un PNL, considérons le problème suivant :

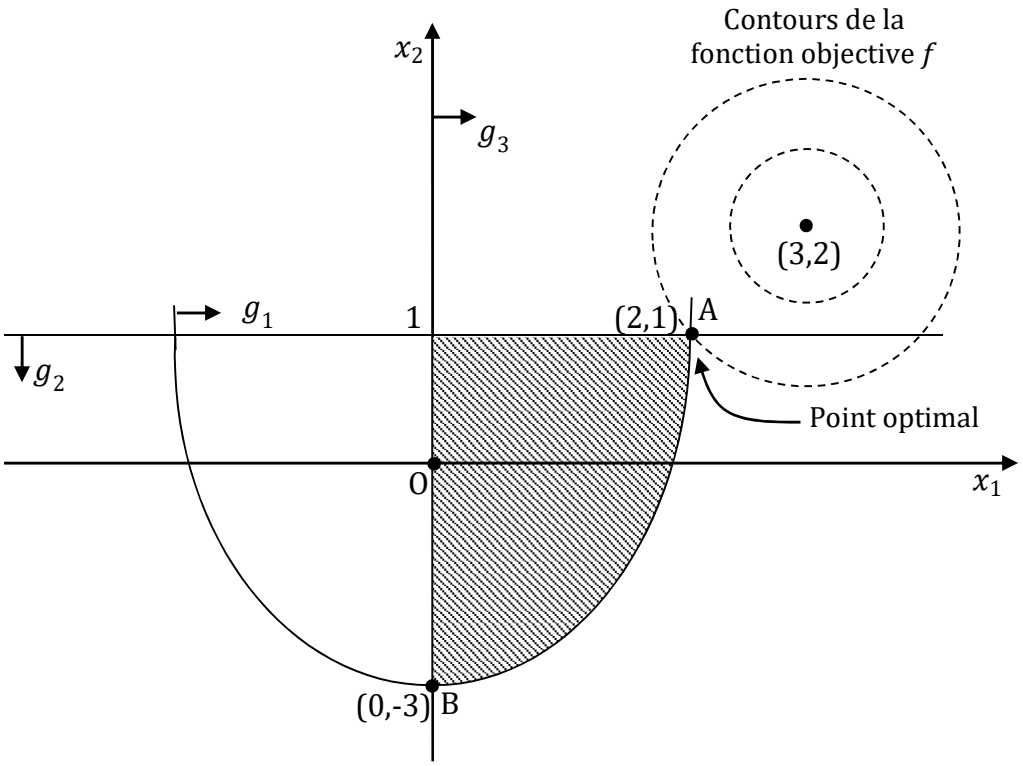
$$\begin{cases} \boldsymbol{min}\ \boldsymbol{(x_1 - 3)^2 + (x_2 - 2)^2} \\ \boldsymbol{x_1^2 - x_2 \leq 3} \\ \boldsymbol{x_2 \leq 1} \\ \boldsymbol{x_1 \geq 0} \end{cases}$$

La fonction objective et les trois contraintes (à inégalités) sont représentées par :

$$\boldsymbol{f(x_1, x_2) = (x_1 - 3)^2 + (x_2 - 2)^2}$$
$$\boldsymbol{g_1(x_1, x_2) = x_1^2 - x_2 - 3}$$
$$\boldsymbol{g_2(x_1, x_2) = x_2 - 1}$$
$$\boldsymbol{g_3(x_1, x_2) = -x_1}$$

La figure suivante illustre la région des solutions réalisables. Le problème est alors de trouver un point dans cette région ayant la plus petite valeur possible de $f(x_1, x_2) = (x_1 - 3)^2 + (x_2 - 2)^2$. A noter que les points (x_1, x_2) vérifiant $(x_1 - 3)^2 + (x_2 - 2)^2 = c$ forment un cercle de rayon $\sqrt{c}$ et dont le centre est positionné à (3,2). Ce cercle représente le *contour* de la fonction objective ayant la valeur c. Puisque nous voulons minimiser la fonction objective, nous devons trouver le cercle ayant le plus petit rayon qui coupe la région réalisable. Comme le montre bien la figure ci-après, la valeur optimale de la fonction objective est $c = 2$ et le point optimal correspond au point A de coordonnées $(x_1, x_2) = (2,1)$.

Remarquons que si on changeait l'objectif de min à max, la solution optimale correspondrait au point B de coordonnées $(x_1, x_2) = (0, -3)$. Evidemment, la résolution graphique n'est possible que pour les petits problèmes (pas plus de deux variables et ayant des contraintes simples !)



Un cas particulier et très fréquent de la programmation non linéaire est la programmation quadratique (QP) où toutes les contraintes sont linéaires et la fonction objective quadratique ; autrement dit seule les formes x_i^2 ou $x_i x_j$ sont autorisées.

Exemple d'un programme quadratique

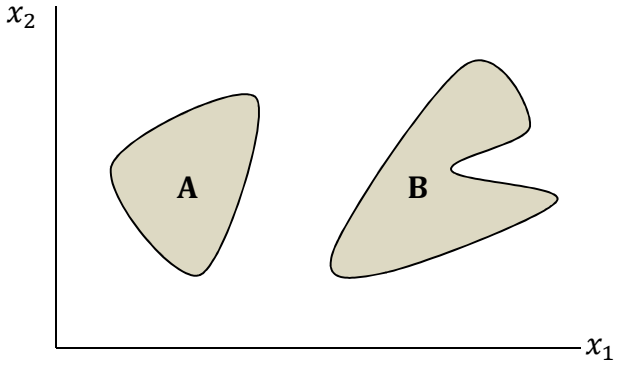
$$\begin{cases} Min\ Z = 3x_1 + 7x_2 + 5x_1^2 + 4x_2^2 + 6x_1x_2 \\ Avec \quad 3x_1 + 4x_2 \geq 3 \\ \qquad\quad 9x_1 + 2x_2 \geq 5 \\ \qquad\quad x_1, x_2 \geq 0 \end{cases}$$

1.2. Analyse convexe

Le concept de convexité occupe une grande importance dans l'étude des problèmes d'optimisation. Ensembles convexes, polytopes, fonctions convexes, gradients, fonctions différentiables, … sont des termes souvent utilisés dans l'analyse et la résolution de divers programmes mathématiques.

Ensembles convexes

Un ensemble $S \subseteq \mathbb{R}^n$ est convexe s'il contient toutes les combinaisons convexes $\lambda x + (1-\lambda)y$, $\lambda \in \mathbb{R}^1$ et $0 \leq \lambda \leq 1$ de paires de points $x, y \in$ S comme le cas de l'ensemble A dans la figure ci-dessous.



Fonctions convexes

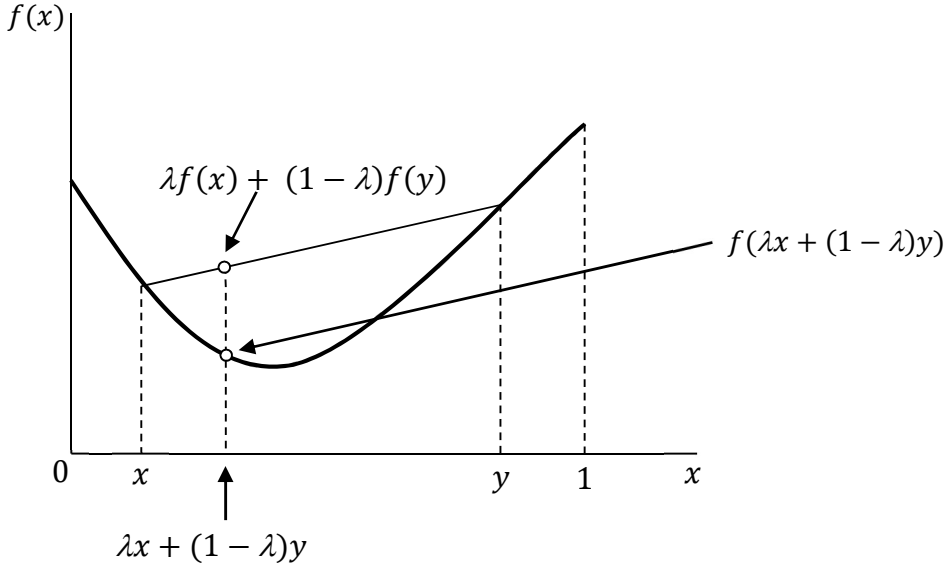
Soit $S \subseteq \mathbb{R}^n$ un ensemble convexe, la fonction

$$f : \mathrm{S} \longrightarrow \mathbb{R}^1$$

est convexe dans S si pour chaque paire de points $x, y \in \mathrm{S}$

$$f(\lambda x + (1-\lambda)y) \leq \lambda f(x) + (1-\lambda) f(y) \qquad \lambda \in \mathbb{R}^1 \quad \text{et} \quad 0 \leq \lambda \leq 1$$

Intuitivement une fonction convexe ressemble schématiquement à une parabole. La condition de convexité implique que les cordes sont toujours sur la courbe de la fonction.



Il est utile de mentionner que les fonctions qui se comportent de manière opposée aux fonctions convexes sont appelées *fonctions concaves* et vérifient donc :

$$f(\lambda x + (1-\lambda)y) \geq \lambda f(x) + (1-\lambda) f(y) \qquad \lambda \in \mathbb{R}^1 \quad \text{et} \quad 0 \leq \lambda \leq 1$$

Si l'inégalité est stricte, on parle de fonction strictement convexe ou strictement concave. Les fonctions suivantes sont convexes, leur négative est évidemment concave :

1. $f(x) = 3x + 4$
2. $f(x) = |x|$
3. $f(x) = x^2 - 2x$
4. $f(x) = -x^{1/2} \quad si\ x \geq 0$
5. $f(x_1, x_2) = 2x_1^2 + x_2^2 - 2x_1 x_2$
6. $f(x_1, x_2, x_3) = x_1^4 + 2x_2^2 + 3x_3^2 - 4x_1 - 4x_2 x_3$

Une importante propriété des fonctions convexes et concaves est qu'elles sont **continues** à l'intérieur de leur domaine.

Fonctions différentiables

Une fonction est différentiable en un point $x = (x_1, \ldots, x_n)$, s'il existe un seul vecteur gradient $\nabla f(x)$ donné par

$$\nabla f(x) = \left(\frac{\partial f(x)}{\partial x_1}, \ldots, \frac{\partial f(x)}{\partial x_n} \right)$$

Où $\partial f(x)/\partial x_i$ est la dérivée partielle de f par rapport à x_i au point x.

Fonctions deux fois différentiables

La matrice H dont l'élément à la ligne i et la colonne j est la dérivée partielle seconde

$$\frac{\partial^2 f(x)}{\partial x_i \partial x_j}$$

est appelée matrice hessienne. Cette matrice joue un rôle essentiel dans la convexité des fonctions deux fois différentiables. Nous avons pour cela :

Une fonction f deux fois différentiable définie sur un ensemble non vide S est :

- Convexe, si la matrice hessienne H est semi-définie positive ($x^t Hx \geq 0\ \forall x \in S$ ou $\lambda_j \geq 0$)
- Concave, si la matrice hessienne H est semi-définie négative ($x^t Hx \leq 0\ \forall x \in S$ ou $\lambda_j \leq 0$).
- Strictement convexe, si H est définie positive ($x^t Hx > 0\ \forall x \in S$ ou $\lambda_j > 0$).
- Strictement concave, si H est définie négative ($x^t Hx < 0\ \forall x \in S$ ou $\lambda_j < 0$).
- Ni convexe ni concave si la matrice hessienne H est indéfinie.

Les λ_j étant les valeurs propres de H déterminées en résolvant le système $\det(H - \lambda I) = 0$.

A titre d'exemple, considérons la fonction $f(x_1, x_2) = 2x_1 + 6x_2 - 2x_1^2 - 3x_2^2 + 4x_1x_2$. Nous voulons savoir si f est convexe ou concave ou ni l'un ni l'autre. Nous pouvons commencer d'abord par écrire la fonction f sous une forme plus pratique comme suit :

$$f(x_1, x_2) = (2 \quad 6)\begin{pmatrix} x_1 \\ x_2 \end{pmatrix} + \frac{1}{2}(x_1 \quad x_2)\begin{bmatrix} -4 & 4 \\ 4 & -6 \end{bmatrix}\begin{pmatrix} x_1 \\ x_2 \end{pmatrix}$$

Pour vérifier si la matrice hessienne H est semi-définie positive ou semi-définie négative ou indéfinie, nous procédons au calcul des valeurs propres de H en résolvant le système

$$0 = \det(H - \lambda I) = \det\left(\begin{bmatrix} -4 & 4 \\ 4 & -6 \end{bmatrix} - \lambda\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}\right) = \det\begin{bmatrix} -4-\lambda & 4 \\ 4 & -6-\lambda \end{bmatrix} = \lambda^2 + 10\lambda + 8$$

Les solutions de cette équation sont $\lambda_1 = -5 + \sqrt{17}$ et $\lambda_2 = -5 - \sqrt{17}$. Puisque les deux valeurs sont strictement négatives, la matrice hessienne H est définie négative et donc f est strictement concave.

Minimum et Maximum

Nous considérons le cas de la minimisation d'une fonction convexe définie sur un ensemble convexe. Le cas de maximisation d'une fonction concave est similaire.

Soit $f: \mathbb{R}^n \to \mathbb{R}$, et considérons le problème de minimisation de $f(x), x \in S$. Un point $x \in S$ est appelée **solution réalisable** du problème. Si $x^* \in S$ et $f(x) \geq f(x^*)$ pour tout $x \in S$, alors x^* est appelée **solution optimale**, ou **solution optimale globale**, ou tout simplement une solution du problème. Si $x^* \in S$ et s'il existe un voisinage $N_\varepsilon(x^*)$ autour de x^* tel que $f(x) \geq f(x^*)$ pour tout $\in S \cap N_\varepsilon(x^*)$, alors x^* est appelée **solution optimale locale**.

La convexité d'une fonction garantit que chaque solution optimale locale est aussi globale. Ceci est très utile dans les algorithmes d'optimisation puisqu'il permet leur arrêt dès que la recherche dans le voisinage d'une solution ne conduit pas à de meilleures solutions.

2. CONDITIONS DE KUHN-TUCKER

Les conditions d'optimalité de Kuhn-Tucker sont à la base de la résolution des programmes non linéaires. Nous allons détailler dans ce chapitre ces conditions en analysant différents types de programmes :

- Programmes sans contraintes,
- Programmes avec contraintes d'égalités,
- Programmes avec contraintes d'inégalités.

Certaines de ces conditions sont dans certains cas, nécessaires et suffisantes pour affirmer que la solution trouvée est optimale, alors que dans d'autres cas, elles sont nécessaires et pas suffisantes. Autrement dit,

une solution peut vérifier ces conditions mais n'est pas une solution optimale. Dans ce cas, il faut considérer d'autres conditions dites conditions suffisantes pour affirmer que l'optimum est atteint. On distingue donc dans la programmation non linéaire deux types de conditions :

- Conditions nécessaires (necessary conditions).
- Conditions suffisantes (sufficient conditions).

Aussi, l'optimalité locale reste un problème majeur dans la résolution des PNL. Si la fonction n'est pas convexe, il n'y a aucune garantie qu'une solution vérifiant les conditions « nécessaires et suffisantes » soit un optimum global, et ne peuvent garantir ainsi que l'optimalité locale.

Le cas très simple de l'optimisation de fonctions à une variable permet d'illustrer ces conditions ainsi que le problème de l'optimalité locale. Prenons l'exemple de l'optimisation (min ou max) de la fonction

$$f(x) = 3x^2 - 5x + 2$$

Toute solution doit nécessairement (conditions nécessaires) vérifier :

$$f' = \frac{df}{dx} = 0$$

Soit :

$$6x - 5 = 0 \longrightarrow \bar{x} = \frac{5}{6}$$

Mais pour s'assurer que ce point est un optimum local (minimum ou maximum) il faut passer aux conditions suffisantes :

$$f''(\bar{x}) = \frac{d^2f}{dx^2} > 0 \longrightarrow \bar{x}\ est\ un\ minimum\ local$$

$$f''(\bar{x}) = \frac{d^2f}{dx^2} < 0 \longrightarrow \bar{x}\ est\ un\ maximum\ local$$

Le cas $f''(\bar{x}) = 0$ (ni positif ni négatif) correspond au cas redouté du point d'inflexion puisqu'il ne s'agit même pas d'un optimum local. Dans cet exemple il s'agit d'un minimum local étant donné que $f'' = f''(\bar{x}) = 6$. Il reste à savoir si cette solution est aussi globale. Ceci n'est garantie que si la fonction f est convexe (ou concave) sur tout l'espace et non pas seulement en un point ; dans le cas de fonctions à une variable, il faut que $f''(x) \geq 0\ \forall x \in \mathbb{R}$ (ou $f''(x) \leq 0\ \forall x \in \mathbb{R}$), ce qui est le cas de cet exemple, f'' est strictement positif $\forall x \in \mathbb{R}$, donc la fonction f est strictement convexe, la solution $\bar{x}$ est un minimum global et de surplus unique.

Ces conditions sont similaires dans le cas général de PNL. Nous commençons par aborder dans la section suivante les conditions d'optimalité dans le cas simple des PNL sans contraintes et les PNL avec contraintes d'égalités, puis nous abordons le cas plus général des PNL avec contraintes d'inégalités pour mettre en évidence les conditions KKT au nom des chercheurs Karush-Kuhn-Tucker qui les ont développées.

2.1. Problèmes sans contraintes

Un problème sans contraintes est un problème de la forme $\min f(x)$ sans aucune contrainte sur le vecteur x. Ce type de problème apparait rarement dans les applications pratiques. Cependant, nous considérons les PNL sans contraintes dans cette section, car les conditions d'optimalité pour les PNL avec contraintes sont une extension logique de celles des PNL sans contraintes. En plus une des stratégies de résolution des problèmes avec contraintes est la résolution successive de plusieurs PNL sans contraintes.

Conditions d'optimalité nécessaires

- Etant donné une fonction $f : \mathbb{R}^n \to \mathbb{R}$ différentiable au point x^*. Si x^* est un minimum local alors :

$$\boldsymbol{\nabla f(x^*) = 0}$$

∇f est appelé vecteur gradient, et cette condition est dite condition nécessaire du premier ordre.

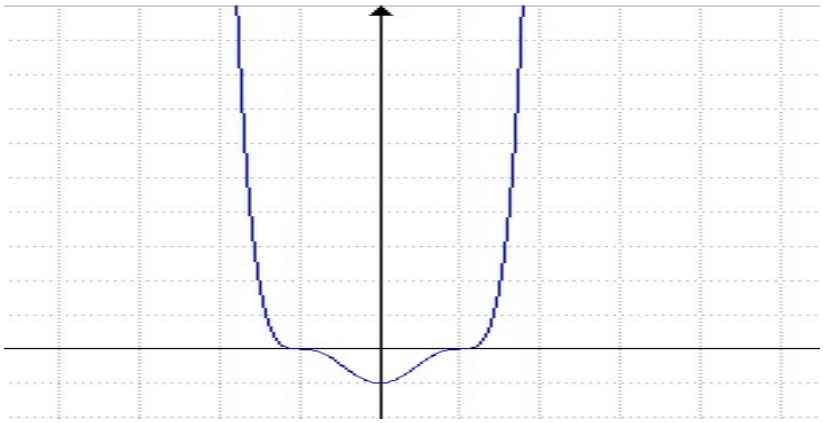
Conditions d'optimalité suffisantes

- Etant donnée une fonction $f: \mathbb{R}^n \to \mathbb{R}$ différentiable au point x^*. Si $\boldsymbol{\nabla f(x^*) = 0}$ et $\boldsymbol{H(x^*)}$ **est définie positive**, alors $\boldsymbol{x^*}$ est un minimum local.

Exemple 1

Considérons le problème de minimisation de la fonction $\boldsymbol{f(x) = (x^2 - 1)^3}$

Nous commençons d'abord par déterminer les points candidats à l'optimalité satisfaisant les conditions nécessaires de premier ordre ; $\nabla f(x) = 0$. Notons que $\nabla f(x) = 6x(x^2 - 1)^2$, et $\nabla f(-1) = \nabla f(0) = \nabla f(1) = 0$. Mais ceci n'implique pas que chacun de ces trois points est un minimum local. En effet, la figure suivante représentant le graphe de la fonction $f(x) = (x^2 - 1)^3$ montre bien que seul le point $x = 0$ est le minimum local. Remarquons en effet que les points -1 et 1 ne vérifient pas les conditions suffisantes qui stipulent que $H(x) = 24x^2(x^2 - 1) + 6(x^2 - 1)^2$ soit définie positive (strictement positive dans le cas de fonctions à une variable), puisque $H(-1) = H(1) = 0$ alors que $H(0) = 6$.



Exemple 2

Considérons le problème de minimisation d'une fonction à deux variables donnée comme suit : $\boldsymbol{f(x_1, x_2) = (x_1 - 8)^2 + (x_2 - 6)^2}$.

Les conditions nécessaires du premier ordre $\nabla f(x) = 0$ conduisent au système suivant :

$$\frac{\partial f}{\partial x_1} = 2(x_1 - 8) = 0 \Longrightarrow x_1 = 8$$

$$\frac{\partial f}{\partial x_2} = 2(x_2 - 6) = 0 \Longrightarrow x_2 = 6$$

Et on obtient une seule solution $x^* = \begin{pmatrix} 8 \\ 6 \end{pmatrix}$

Pour les conditions suffisantes, on détermine la matrice hessienne H :

$$H = \begin{pmatrix} \dfrac{\partial^2 f}{\partial x_1^2} & \dfrac{\partial^2 f}{\partial x_1 x_2} \\ \dfrac{\partial^2 f}{\partial x_2 x_1} & \dfrac{\partial^2 f}{\partial x_2^2} \end{pmatrix}$$

Et on obtient : $H = \begin{pmatrix} 2 & 0 \\ 0 & 2 \end{pmatrix}$ qui est indépendante de x.

Pour connaître le « signe » de la matrice hessienne, on calcule les valeurs propres en résolvant le système $|\lambda \mathrm{I} - H| = 0$

$$\begin{vmatrix} \lambda - 2 & 0 \\ 0 & \lambda - 2 \end{vmatrix} = (\lambda - 2)^2 \Longrightarrow \lambda = 2$$

La valeur propre étant strictement positive, la matrice hessienne est définie positive $\forall x$. Ce qui est une condition suffisante pour que la solution x^* soit un minimum local. Aussi, la convexité de la fonction f est confirmée par le signe de la matrice hessienne : définie positive $\forall x \Leftrightarrow$ convexité stricte. La solution x^* est donc un minimum global.

2.2. Problèmes avec contraintes d'égalités

Un PNL avec contraintes d'égalités se présente sous la forme générale suivante :

$$\begin{cases} \boldsymbol{min\ f(x)} \\ \boldsymbol{h_j(x) = 0 \quad j = 1, \dots, \ell} \end{cases}$$

Un tel PNL peut se transformer en un programme sans contraintes en introduisant les multiplicateurs de Lagrange comme suit :

$$\boldsymbol{L(x, \lambda) = f(x) + \lambda h(x)}$$

Ainsi les mêmes conditions d'optimalité des problèmes sans contraintes s'appliquent à ce type de PNL.

Exemple

Considérons le programme à deux contraintes d'égalités suivant :

$$\begin{cases} \boldsymbol{min\ x_1^2 + 2x_2^2 + 2x_3^2} \\ \boldsymbol{x_1 + x_2 + x_3 = 5} \\ \boldsymbol{x_1 + 3x_2 + 2x_3 = 9} \end{cases}$$

Nous avons dans ce PNL :

$$f(x) = x_1^2 + 2x_2^2 + 2x_3^2$$
$$h_1(x) = 5 - x_1 - x_2 - x_3$$
$$h_2(x) = 9 - x_1 - 3x_2 - 2x_3$$

Le lagrangien correspondant s'écrit donc :

$$L(x, \lambda) = x_1^2 + 2x_2^2 + 2x_3^2 + \lambda_1(5 - x_1 - x_2 - x_3) + \lambda_2(9 - x_1 - 3x_2 - 2x_3)$$

Les conditions nécessaires du premier ordre $\nabla L(x, \lambda) = 0$ conduisent au système suivant :

$$\frac{\partial f}{\partial x_1} = 2x_1 - \lambda_1 - \lambda_2 = 0$$

$$\frac{\partial f}{\partial x_2} = 4x_2 - \lambda_1 - 3\lambda_2 = 0$$

$$\frac{\partial f}{\partial x_3} = 4x_3 - \lambda_1 - 2\lambda_2 = 0$$

$$\frac{\partial f}{\partial \lambda_1} = 5 - x_1 - x_2 - x_3 = 0$$

$$\frac{\partial f}{\partial \lambda_2} = 9 - x_1 - 3x_2 - 2x_3 = 0$$

La résolution de ce système conduit à la solution :

$$x_1 = \frac{26}{11}, x_2 = \frac{15}{11}, x_3 = \frac{14}{11}, \lambda_1 = \frac{48}{11}, \lambda_2 = \frac{4}{11}$$

Nous passons aux conditions suffisantes en déterminant la matrice hessienne. Etant donné que les contraintes sont linéaires, on se contente de la matrice hessienne de la fonction f et on obtient :

$$H = \begin{pmatrix} 2 & 0 & 0 \\ 0 & 4 & 0 \\ 0 & 0 & 4 \end{pmatrix}$$

Pour connaître le « signe » de la matrice hessienne, on calcule les valeurs propres en résolvant le système $|\lambda \mathrm{I} - H| = 0$.

$$\begin{vmatrix} \lambda - 2 & 0 & 0 \\ 0 & \lambda - 4 & 0 \\ 0 & 0 & \lambda - 4 \end{vmatrix} = (\lambda - 2)(\lambda - 4)^2 = 0 \Rightarrow \lambda = \begin{cases} 2 \\ 4 \end{cases}$$

Les valeurs propres étant strictement positives, la matrice hessienne est définie positive $\forall x$. Ce qui est une condition suffisante pour que la solution x^* soit un minimum local. Aussi, la convexité de la fonction f est confirmée par le signe de la matrice hessienne : définie positive ⟺ convexité stricte. En plus les contraintes sont linéaires donc forment un ensemble convexe. Le PNL est par conséquent convexe ; La solution x^* est donc un minimum global.

$$x^* = \begin{pmatrix} 26/11 \\ 15/11 \\ 14/11 \end{pmatrix} \text{ de valeur } f(x^*) = 138/11$$

2.3. Problèmes avec contraintes d'inégalités (LES CONDITIONS KKT)

Conditions nécessaires (Conditions de KKT)

Etant donné un PNL sous la forme :

$$\begin{cases} min\, f(x) \\ g_i(x) \leq 0 \end{cases}$$

Nous commençons par ajouter les variables d'écart aux contraintes et on obtient :

$$\begin{cases} min\, f(x) \\ g_i(x) + t^2 = 0 \end{cases}$$

Puis nous « dualisons » les contraintes en ajoutant les multiplicateurs de Lagrange. Nous obtenons le Lagrangien suivant :

$$L(x, \lambda) = f(x) + \lambda(g(x) + t^2)$$
$$\lambda \geq 0$$

Remarquons que si le PNL de départ contient n variables de décision x_i et m contraintes, le Lagrangien sera composé de $n + 2m$ variables comme suit :

$x = (x_1, \dots, x_n)$: variables de décision principales (Decision variables)

$t = (t_1, \dots, t_m)$: variables d'écart (Slack variables)

$\lambda = (\lambda_1, \dots, \lambda_m)$: multiplicateurs de Lagrange (Lagrange multipliers)

La recherche des points stationnaires (conditions nécessaires de premier ordre) se fait en posant $\boldsymbol{\nabla L = 0}$. On obtient :

$$\frac{\partial L}{\partial x} = \nabla f(x) + \lambda \nabla g(x) = 0$$

$$\frac{\partial L}{\partial t} = 2\lambda_i t_i = 0$$

$$\frac{\partial L}{\partial \lambda} = g(x) + t^2 = 0$$

Avec $\lambda \geq 0$

La deuxième équation est transformée comme suit :

$$2\lambda_i t_i = 0 \Leftrightarrow \lambda_i g_i(x) = 0$$

étant donné que $t_i = 0$ est équivalent à $g_i(x) = 0$ puisque nous avons $g(x) + t^2 = 0$

La troisième contrainte est réécrite dans sa forme initiale comme suit :

$$g(x) + t^2 = 0 \Leftrightarrow g_i(x) \leq 0$$

Et on obtient les conditions nécessaires d'optimalité connues sous le nom de **conditions de Karush-Kuhn-Tucker (KKT)** :

$$\boldsymbol{\nabla f(x) + \lambda \nabla g(x) = 0}$$
$$\boldsymbol{\lambda_i g_i(x) = 0}$$
$$\boldsymbol{g_i(x) \leq 0}$$
$$\boldsymbol{\lambda \geq 0}$$

La résolution du système résultant des conditions KKT peut s'avérer très compliquée. En effet, la non linéarité de la fonction objective et aussi des contraintes conduit à un système non linéaire souvent difficile à résoudre. Dans le cas de la programmation quadratique, les conditions KKT conduisent à un système majoritairement linéaire où l'utilisation des techniques de programmation linéaire est possible.

Exemple

Rappelons le PNL vu dans la section 1.1 :

$$\begin{cases} \boldsymbol{min}\ (\boldsymbol{x_1 - 3})^2 + (\boldsymbol{x_2 - 2})^2 \\ \quad \boldsymbol{x_1^2 - x_2 - 3 \leq 0} \\ \quad \boldsymbol{x_2 - 1 \leq 0} \\ \quad \boldsymbol{-x_1 \leq 0} \end{cases}$$

La résolution graphique de ce PNL a montré que la solution optimale est :

$$\begin{cases} x_1 = 2 \\ x_2 = 1 \end{cases}$$

Posons à présent les conditions KKT pour ce PNL. Affectons d'abord pour chaque contrainte, un multiplicateur de Lagrange :

$$\boldsymbol{f(x_1, x_2) = (x_1 - 3)^2 + (x_2 - 2)^2}$$
$$\lambda_1: \quad \boldsymbol{g_1(x_1, x_2) = x_1^2 - x_2 - 3}$$
$$\lambda_2: \quad \boldsymbol{g_2(x_1, x_2) = x_2 - 1}$$
$$\lambda_3: \quad \boldsymbol{g_3(x_1, x_2) = -x_1}$$

$\boldsymbol{\nabla f(x) + \lambda \nabla g(x) = 0}$

$$/x_1 : \ 2x_1 - 6 + 2x_1\lambda_1 - \lambda_3 = 0$$
$$/x_2 : \ 2x_2 - 4 - \lambda_1 + \lambda_2 = 0$$

$\boldsymbol{\lambda_i g_i(x) = 0}$

$$\lambda_1(x_1^2 - x_2 - 3) = 0$$
$$\lambda_2(x_2 - 1) = 0$$
$$\lambda_3(-x_1) = 0$$

$\boldsymbol{g_i(x) \leq 0}$

$$x_1^2 - x_2 - 3 \leq 0$$
$$x_2 - 1 \leq 0$$
$$-x_1 \leq 0$$

$\boldsymbol{\lambda \geq 0}$

$$\lambda_1, \lambda_2, \lambda_3 \geq 0$$

Il est clair que la résolution d'un tel système s'avère compliquée. Nous allons nous contenter de vérifier que la solution trouvée graphiquement à savoir $(x_1, x_2) = (2,1)$ vérifie bien les conditions KKT. Après substitutions, nous obtenons :

$$\boldsymbol{x_1 = 2, \ x_2 = 1, \lambda_1 = 1/2, \ \lambda_2 = 5/2, \lambda_3 = 0}$$

Les multiplicateurs de Lagrange sont bien non négatifs. Le point $(x_1, x_2) = (2,1)$ vérifie bien les conditions KKT puisqu'il est en fait un optimum local et même global.

Remarque

Les solutions obtenues dans les exemples précédents vérifient les conditions KKT qui rappelons le, sont seulement des conditions nécessaires. Pour affirmer que la solution trouvée est un minimum local voir même global, il faut passer aux conditions suffisantes.

Conditions suffisantes

Soit x^* une solution trouvée par le système formé par les conditions de KKT, La convexité de f et des g_i au point x^* suffit à confirmer que x^* est un minimum local.

Finalement, la convexité du PNL ; autrement dit f et les g_i convexes sur tout l'espace de définition du PNL garantit que le minimum local trouvé est aussi global.

3. PROGRAMMATION QUADRATIQUE ET SEMI DEFINIE

3.1. Définition

Un problème de programmation quadratique est un problème qui peut se mettre sous la forme suivante :

$$Min\ Z = \sum_{i=1}^{n} c_i x_i + \sum_{i=1}^{n}\sum_{j=1}^{n} h_{ij} x_i x_j$$

$$Avec \sum_{i=1}^{n} a_{ij} x_i \leq b_j \quad \forall j \in 1,..,m$$

Ou encore avec la notation matricielle :

$$Min\ Z = CX + \frac{1}{2} X^T H X$$
$$Avec\ \ AX \ \leq \ b$$

Exemple :

Considérons le programme quadratique suivant :

$$\begin{cases} MinZ = 3x_1 + 7x_2 + 5x_1^2 + 4x_2^2 + 6x_1x_2 \\ \quad Avec \quad 3x_1 + 4x_2 \leq 3 \\ \quad\quad\quad 9x_1 + 2x_2 \leq 7 \end{cases}$$

En notation matricielle, nous avons :

$$X = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}, \qquad C = [3 \quad 4], \qquad H = \begin{bmatrix} 10 & 6 \\ 6 & 8 \end{bmatrix}, \qquad A = \begin{bmatrix} 3 & 4 \\ 9 & 3 \end{bmatrix}, \qquad b = \begin{bmatrix} 3 \\ 7 \end{bmatrix}$$

3.2. Programmation semi-définie

Un programme semi définie est un cas particulier de la programmation quadratique, pour lequel la matrice hessienne H est semi définie positive, c'est à dire que :

$$\forall X \in \mathbb{R} \ : \ X^T H X \geq 0$$

Cette condition garantie la convexité de la fonction quadratique correspondante (fonction objectif et/ou contraintes) : condition importante pour la méthode de résolution détaillée ci-après.

Exemple : $f(x) = 2x_1^2 + x_2^2 - 4x_1x_2 + 2x_1$

4. RELAXATION SEMI DEFINIE

La résolution des programmes quadratiques et non linéaires en général reste un domaine de recherche très actif vu la difficulté du problème. Parmi les méthodes de résolution existantes, on a la relaxation semi définie consistant à transformer un problème avec contrainte en un problème sans contraintes.

Exemple :

Soit à optimiser le rendement (Z) d'une terre en fonction de la quantité de fertilisant (x_1) et d'insecticide (x_2). Nous obtenons le programme quadratique suivant :

$$\begin{cases} OptZ = x_1 + 3x_1x_2 \\ \\ Avecx_1 + 2x_2 = 8 \end{cases}$$

La relaxation semi définie conduit au Lagrangien suivant : $L(x_1, x_2, \lambda) = x_1 + 3x_1x_2 + \lambda(8 - x_1 - 2x_2)$

L'introduction du multiplicateur de Lagrange λ permet de transformer un problème d'optimisation **avec contrainte** à 2 variables en un problème d'optimisation **sans contrainte** à 3 variables (x_1, x_2, λ).
La recherche du point stationnaire se fait en posant : $\nabla_x L(x, \lambda) = 0$ (Conditions du premier ordre)

$$\frac{\partial L}{\partial x_1} = 3x_2 + 1 - \lambda = 0\,, \quad \frac{\partial L}{\partial x_2} = 3x_1 - 2\lambda = 0\,, \quad \frac{\partial L}{\partial \lambda} = 8 - x_1 - 2x_2 = 0$$

On solutionne ce système d'équations et on trouve $(x_1^*, x_2^*, \lambda^*)$: $x_1^* = 26/6, \quad x_2^* = 11/6, \quad \lambda^* = 39/6.$

En principe, à ce stade-ci, on doit calculer les conditions de second ordre et vérifier si le point stationnaire $(x_1^*, x_2^*, \lambda^*)$ représente un minimum ou un maximum.

Remarque :
La convexité (caractéristique de la programmation semi-définie) garantie l'existence d'un seul point stationnaire (optimum global).

Références bibliographiques

1. Billonnet, A. « Optimisation discrete » Dunod, 2007.
2. Garey, M.S. et Johnson, D.S. « Computers and Intractability: A Guide to the Theory of NP-Completeness » W.H. Freeman, New York, 1979.
3. Glover, F. et Laguna, M. « Tabu search » Kluwer Academic Publishers, Norwell, MA, 2nd Ed., 1997.
4. Mahjoub, A.R. « Approches polyédrales en optimisation combinatoire » Hermes, Lavoisier, 2005.
5. Sakarovitch, M. « Optimisation Combinatoire, Graphes et Programmation Linéaire ». Hermann, Enseignement des sciences, Paris, 1984.
6. Sakarovitch, M. « Optimisation Combinatoire, Programmation Discrète ». Hermann, Enseignement des sciences, Paris, 1984.
7. Teghem, J. et Pirlot, M. « Optimisation approchée en recherche opérationnelle » Lavoisier, Paris, 2002.
8. Zennaki, M. « Intégration des techniques d'apprentissage artificiel aux méta-heuristiques pour la résolution des problèmes d'optimisation combinatoires difficiles » Thèse de Doctorat. Université des Sciences et de la Technologie d'Oran Mohamed Boudiaf, 2017.