Partie 1 — Heuristiques

1 Heuristiques et approximations

1.1 Optimisation combinatoire

Un problème d'optimisation combinatoire à résoudre consiste d'une **fonction objectif** et de la recherche d'une meilleure solution parmi l'ensemble de **solutions réalisables**.

Formellement, un problème d'optimisation combinatoire se définit par :

— ensemble **fini** S de solutions admissibles,

— fonction objectif $f : S \to \mathbb{R}$ qui associe une valeur aux solutions,

On cherche alors une solution $x^* \in S$ minimisant/maximisant la valeur de la fonction f.

$$\min_{x \in S} f(x)$$

1.2 Exemples

1.2.1 Arbre recouvrant de poids minimum

Dans un graphe connexe $G = (V, E)$ aux arêtes pondérées, on cherche un sous-ensemble d'arêtes formant un arbre recouvrant et de poids total minimum.

Pour un graphe complet sur n sommets il existe n^{n-2} arbres recouvrants différents. Le problème est cependant facile et les algorithmes de Kruskal ou de Prim permettent de le résoudre efficacement en temps polynomial.

1.2.2 Problème du voyageur de commerce

Étant donné un graphe complet $G = (V, E)$ sur n sommets et une matrice de distances entre chaque paire de sommets, on cherche une tournée, c.à-d. un cycle hamiltonien passant une fois par chaque sommet, de longueur totale aussi faible que possible.

Ici aussi le nombre de solutions admissibles est gigantesque (égal à $\frac{(n-1)!}{2}$ si les distances sont symétriques). Contrairement aux deux exemples précédents, la recherche d'une tournée optimale est un problème difficile. Aucun algorithme de complexité polynomiale n'est connu pour le résoudre dans le cas général.

1.2.3 Problème du stable de cardinal maximum

Étant donné un graphe $G = (V, E)$, on cherche un stable, c.-à-d. un sous-ensemble de sommets deux à deux non adjacents, aussi grand que possible.

Même si dénombrer tous les stables d'un graphe est très difficile, leur nombre est fini et borné supérieurement par $|\mathscr{P}(V)| = 2^n$ $(n = |V|)$. Ici aussi le problème est difficile et aucun algorithme de complexité polynomiale n'est connu pour le résoudre dans le cas général.

1.2.4 Problème d'optimisation linéaire

La résolution d'un problème d'optimisation linéaire, aussi appelé **programme linéaire**, consiste à déterminer l'optimum (minimum ou maximum) d'une fonction linéaire de n variables soumises à des contraintes (équations ou inéquations) linéaires.

Si les variables ne peuvent prendre que des valeurs entières (typiquement binaires), le problème est difficile dans le cas général.

Si les variables sont réelles, le problème peut être résolu en temps polynomial.

1.3 Algorithmes de résolution

1.3.1 Les méthodes exactes

Un algorithme exact fournit toujours une solution optimale du problème.

Pour les problèmes faciles, un bon algorithme exact a une complexité polynomiale et demande donc un temps de résolution (nombre d'itérations, nombre d'opérations) borné par un polynôme en la taille du problème.

Pour les problèmes difficiles, la méthode peut nécessiter un nombre exponentiel d'itérations.

1.3.2 Les algorithmes d'approximation

Une approximation fournit une solution sous-optimale en général, elle assure une qualité minimale de la solution produite, elle est de complexité raisonnable (polynomiale).

1.3.3 Les heuristiques

Une heuristique fournit une solution sous-optimale en général, elle ne donne aucune garantie sur la qualité de la solution fournie, elle est efficace en pratique, on constate empiriquement qu'elle trouve souvent une bonne solution.

Il existe plusieurs types d'heuristiques :

— **heuristiques constructives** qui construisent itérativement une solution approchée,

— **heuristiques d'amélioration** qui partent d'une solution admissible du problème (obtenue, par exemple, à l'aide d'une heuristique constructive) et qui essaient, en appliquant un ensemble de règles, de modifier cette solution afin d'en obtenir une nouvelle, dite **voisine**, de meilleure qualité. Ces modifications, souvent appelés **échanges**, sont enchaînées tant que des améliorations sont possibles,

— **métaheuristiques** qui travaillent sur un schéma plus global.

1.4 Approximations

Un algorithme d'approximation est une méthode approchée (une heuristique) pour laquelle **on peut garantir une qualité minimale de la solution produite**.

Pour un problème de minimisation P, un algorithme H est une ρ-approximation si, quelle que soit l'instance de P (le jeu de données particulier) de valeur optimale $z^* > 0$, l'algorithme H s'exécute en un temps polynomial et retourne une solution admissible du problème de valeur z_H vérifiant :

$$z_H \leq \rho z^* \quad \Leftrightarrow \quad \frac{z_H}{z^*} \leq \rho$$

Ainsi, une 2-approximation fournit une solution dont la valeur (le coût) ne dépasse jamais le double de la valeur optimale (c.-à-d. minimale) du problème.

2 Coloration de graphes

2.1 Coloration des sommets d'un graphe

Soit $G = (V, E)$ un graphe simple, non orienté, une **coloration des sommets de** G consiste à attribuer une couleur à chaque sommet du graphe de sorte que deux sommets adjacents reçoivent des couleurs différentes ou, dit autrement, de sorte que les extrémités de chaque arêtes aient des couleurs différentes.

Une coloration des sommets d'un graphe en k couleurs est une k-coloration. Un graphe G qui admet une coloration de ses sommets en k couleurs est dit k-coloriable ou k-chromatique.

Le plus petit entier k pour lequel G admet une k-coloration (c.-à-d. le nombre minimum de couleurs nécessaires pour colorier les sommets de G) est appelé le nombre chromatique de G et est noté $\chi(G)$.

2.2 Coloration séquentielle

Les méthodes de **coloration séquentielle**, aussi appelées **méthodes gloutonnes**, sont des heuristiques constructives qui considèrent les sommets du graphe les uns après les autres dans un ordre prédéfini ou calculé au fur et à mesure et essaient de colorier chaque sommet avec une des couleurs déjà utilisées jusque là.

Choix classiques pour l'ordre de coloration des sommets :

— **ordre décroissant des degrés**,

— ordre croissant des degrés,

— ordre lexicographique/aléatoire des sommets du graphe.

Règles classiques pour le choix de la couleur à réutiliser :

— **couleur la plus ancienne** (parcours des couleurs déjà utilisées dans l'ordre croissant, règle par défaut),

— couleur la plus récente (parcours des couleurs déjà utilisées dans l'ordre décroissant),

— couleur la plus utilisée, la moins utilisée,

— couleur choisie au hasard.

2.3 LF (Plus grand degré en premier)

La complexité, aussi bien spatiale que temporelle, de l'heuristique est linéaire en la taille du graphe, c.-à-d. en $O(n+m)$ pour un graphe comptant n sommets et m arêtes.

Comme toutes les colorations gloutonnes, cette heuristique utilise au plus $\Delta(G) + 1$ couleurs pour un graphe G dont le degré maximal des sommets est $\Delta(G)$.

2.4 SL (Plus petit degré en dernier)

Le sommet v_1 est le sommet de plus petit degré du graphe $G = (V, E)$.

Pour k de 2 à n $(n = |V|$, le nombre de sommets de $G)$, choisir pour v_k le sommet de plus petit degré dans le sous-graphe induit par $V \setminus \{v_1, \ldots, v_{k-1}\}$.

La coloration séquentielle du graphe s'effectue ensuite dans **l'ordre inverse** : $(v_n, v_{n-1}, \ldots, v_2, v_1)$.

La complexité, aussi bien spatiale que temporelle, de l'heuristique est linéaire en la taille du graphe, c.-à-d. en $O(n+m)$ pour un graphe comptant n sommets et m arêtes.

2.5 DSATUR

Même structure que LF et SL, avec un ordre de coloration construit **dynamiquement**.

Plus précisément, pour une coloration partielle du graphe G, la méthode définit le **degré de saturation** du sommet v comme le nombre de couleurs **différentes** utilisées par les sommets adjacents à **v et déjà coloriés**.

L'algorithme commence par affecter une première couleur au sommet de degré maximal du graphe.

Pour les itérations suivantes, le sommet à être colorié (par la méthode séquentielle) est choisi comme celui ayant un **degré de saturation (actuel)** maximal et, en cas d'égalité, celui de degré maximal dans le sous-graphe induit par les sommets non coloriés (ou dans le graphe initial).

L'heuristique peut être mise en œuvre avec une complexité, aussi bien spatiale que temporelle, en $O(n^2)$ pour un graphe d'ordre n.

2.6 RLF (Recursive largest first)

Coloration d'un graphe $G = (V, E)$ en construisant séquentiellement une **partition des sommets du graphe en sous-ensembles stables** $\{C_1, \ldots, C_k\}$, chaque ensemble stable C_i correspondant aux sommets recevant la couleur i.

La complexité temporelle de l'heuristique est en $O(nm)$ pour un graphe comptant n sommets et m arêtes.

Algorithme 1 : RLF

```
k ← 0
tant que existe des sommets non coloriés faire
    k ← k + 1                          // nouvelle couleur nécessaire
    Calculer C_k des sommets pour la couleur k
    V ← V \ C_k                        // V sans les sommets de C_k
    G ← G(V)                           // m.à.j. G
```

Algorithme 2 : Calcul d'une classe C pour RLF

```
Déterminer le sommet v ∈ V de degré maximal
C ← {v}
U ← Adj[v]
W ← V \ (Adj[v] ∪ {v})
tant que W ≠ ∅ faire
    Déterminer le sommet w ∈ W ayant un nombre
      maximal de voisins dans U
    // w est le sommet ayant le plus de voisins
       communs avec les sommets déjà dans C ⇒
       on l'ajoute C
    C ← C ∪ {w}
    U ← U ∪ Adj[w]
    W ← W \ (Adj[w] ∪ {w})
```

2.7 Performances

Les quatre heuristiques de coloration séquentielle présentées utilisent au plus $\Delta(G) + 1$ couleurs pour un graphe G dont le degré maximal des sommets est $\Delta(G)$.

Cette performance n'est pas spécifique aux méthodes présentées mais est vérifiée par tous les algorithmes de coloration gloutonne

Les heuristiques DSATUR et RLF sont optimales pour les graphes bipartis (elles n'utilisent jamais plus de deux couleurs pour colorier de tels graphes).

3 Voyageur de commerce

3.1 Définition

Données : un ensemble $\{1, ..., n\}$ de n villes et pour chaque couple (i, j) de villes distinctes une distance $d(i, j)$.

Objectif : trouver la tournée la plus courte visitant chacune des villes une et une seule fois.

3.2 Variantes

— problèmes **symétriques** : $d(i, j) = d(j, i)$,

— problèmes **asymétriques** : $d(i, j)$ pas forcément égal à $d(j, i)$,

— problèmes vérifiant **l'inégalité triangulaire** : $d(i, j) + d(j, k) \geq d(i, k)$,

— problèmes **euclidiens** et **géométriques** : villes sont des points du plan et les distances qui les séparent sont égales aux distances euclidiennes.

3.3 Heuristique : Plus proche voisin

Principe : Construire la tournée itérativement en se déplaçant à chaque fois vers la plus proche ville non encore visitée

Variante **par les deux bouts** : considérer les insertions depuis le début ou la fin de la tournée.

3.4 Heuristique gloutonne

Modèle : Une tournée correspond à un cycle **hamiltonien** (c.-à-d. passant une et une seule fois par chaque sommet) dans le graphe complet K_n dont les n sommets correspondent aux villes du problème.

Principe : Construire le cycle hamiltonien en considérant les arêtes les plus courtes d'abord.

3.4.1 Algorithme

1. Parcourir les arêtes de la plus courte à la plus longue et sélectionner une arête si et seulement si
   (a) elle ne crée aucun sous-tour avec celles déjà sélectionnées, c.-à-d. aucun cycle de longueur inférieure à n (la longueur correspondant ici au nombre d'arêtes ou de sommets du cycle) ;
   (b) elle ne crée aucun sommet de degré 3.

3.5 Heuristique : Insertion la moins chère

Principe : : Partir d'une tournée partielle sur deux villes seulement et insérer les villes manquantes les unes après les autres en choisissant à chaque itération celle qui provoque la plus petite augmentation de la longueur totale de la tournée (approche gloutonne).

3.5.1 Algorithme

1. Construire une tournée partielle en choisissant une ville et sa plus proche voisine.
2. Déterminer l'arête $\{i, j\}$ de la tournée actuelle et la ville k n'en faisant pas partie avec un coût d'insertion $d(i, k) + d(k, j) - d(i, j)$ minimum. Insérer k dans la tournée entre i et j.
3. Répéter le point précédent tant qu'il reste des villes à insérer.

3.5.2 Cas euclidien

Partir de la tournée partielle définie par l'enveloppe convexe des points.

3.6 Heuristique : Insertion la plus éoignée

Principe : Fixer rapidement la forme générale de la tournée.

Cette méthode donne de meilleurs résultats, en général, que la précédente.

3.6.1 Algorithme

1. Partir de la tournée partielle formée des deux villes les plus éloignées.
2. Déterminer la ville la plus éloignée de la tournée actuelle. Insérer cette ville de manière optimale dans la tournée.
3. Répéter le point précédent tant qu'il reste des villes à insérer.

3.6.2 Cas euclidien

Partir de la tournée partielle définie par l'enveloppe convexe des points.

3.7 Méthode de Christofides

Principe : Transformer un cycle eulérien en un cycle hamiltonien.

Trouver le meilleur parcours du cycle eulérien, apportant les raccourcis les plus intéressants et fournissant la meilleure tournée (le cycle hamiltonien le plus court), est un problème difficile.

3.7.1 Algorithme

1. Déterminer un arbre recouvrant T de poids minimum dans le graphe pondéré complet $G = (V, E)$ représentant le problème.
2. Dans le sous-graphe de G induit par les sommets de degré impair dans T, déterminer un couplage parfait M de poids minimum.
3. Ajouter les arêtes de M à l'arbre optimal T (en dupliquant les arêtes appartenant à M et à T). Le graphe obtenu a tous ses sommets de degré pair et possède donc un cycle eulérien.
4. Construire la tournée en parcourant ce cycle tout en prenant des raccourcis afin d'éviter de visiter une ville plus d'une fois.

3.7.2 Facteur d'approximation

Pour les problèmes symétriques vérifiant l'inégalité triangulaire (en particulier les problèmes euclidiens), l'algorithme de Christofides est une $\frac{3}{2}$-approximation. Soit, pour L_{CH} la longueur de la tournée fournie et L^* la longueur optimale, on aura :

$$L_{CH} \leq \frac{3}{2} L^*$$

3.8 Heuristique : Meilleures fusions

Dans cette méthode on commence par choisir une ville (disons la ville 1) qui joue le rôle de *dépôt central* et on considère la pseudo-tournée consistant à faire l'aller-retour depuis ce dépôt jusqu'à chacune des autres villes. Les sous-tournées sont ensuite fusionnées en remplaçant une arête retour $\{i, 1\}$ et une arête aller $\{1, j\}$ par le trajet direct $\{i, j\}$, les fusions étant considérées dans l'ordre décroissant des gains (les « épargnes » ou *savings* en anglais) qu'elles apportent (approche gloutonne).

4 Heuristiques d'amélioration et d'échanges

4.1 Heuristiques d'amélioration

Les heuristiques d'amélioration sont des exemples particuliers de méthodes de recherche locale.

Ces techniques se basent sur les observations suivantes, valables pour de nombreux problèmes d'optimisation combinatoire :

- Il est souvent facile d'obtenir une solution **admissible** (éventuellement de mauvaise qualité) à un problème d'optimisation combinatoire donné.
- Il est souvent possible de modifier légèrement et localement une solution tout en conservant son admissibilité.

La structure de base de ces techniques est alors la suivante :

1. Générer une solution admissible initiale ;
2. Modifier localement la solution courante afin d'obtenir une nouvelle solution admissible, meilleure que l'actuelle (de valeur inférieure dans un problème de minimisation) ;
3. Répéter l'étape précédente tant qu'on trouve une modification améliorante.

Une modification locale d'une solution est appelée un **mouvement**.

Ces mouvements correspondent le plus souvent à l'application de **règles** de modifications et sont donc définis implicitement

La solution obtenue en appliquant un mouvement m à une solution s est souvent notée $s \oplus m$.

Une telle solution est dite **voisine** de la solution s.

Un choix donné de modifications locales (de mouvements) définit donc une **structure de voisinage** sur l'ensemble S des solutions du problème.

4.2 Algorithme de descente

4.2.1 Données

- Une solution initiale $s_0 \in S$;
- Une fonction objectif f à minimiser ;
- Pour chaque solution $s \in S$, un ensemble $M(s)$ de mouvements.

4.2.2 Résultat

Une solution s correspondant à un minimum local de la fonction f relativement à la structure de voisinage définie par les mouvements M.

4.2.3 Algorithme

Tant que l'on ne trouve pas au moins un mouvement améliorant, on répète :

- Parcourir toutes les solutions voisines de la solution actuelles, autrement dit considérer tous les mouvements de $M(s)$ jusqu'à

  Variante 1 :

  Trouver un mouvement m tel que $f(s \oplus m) < f(s)$

  Remplacer alors s par $s \oplus m$.

  Variante 2 :

  Trouver le mouvement m qui minimise $f(s \oplus m)$

  Remplacer alors s par $s \oplus m$ si $f(s \oplus m) < f(s)$.

4.3 Heuristiques k-opt

Les méthodes k-opt sont des heuristiques d'échanges pour le problème du voyageur de commerce qui partent d'une tournée initiale (obtenue à l'aide d'une heuristique constructive) et qui cherchent à la modifier tant qu'il est possible d'en diminuer la longueur.

Chaque modification consiste à supprimer k arêtes de la tournée et à les remplacer par k autres (on parle alors d'un k-échange) tout en prenant soin de reconnecter les morceaux de manière à conserver une tournée.

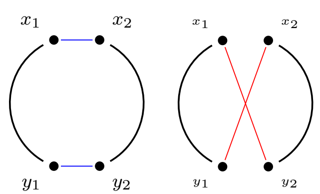
4.3.1 2-opt

Un échange 2-opt consiste à remplacer deux arêtes $\{a, b\}$ et $\{c, d\}$ d'une tournée par les arêtes $\{a, c\}$ et $\{b, d\}$.

Le « coût » d'une modification peut être calculé en temps constant mais la modification de la tournée nécessite l'utilisation d'une « bonne » structure de données (une approche directe peut demander un temps $O(n)$ par modification, car le sens de parcours entre b et c, ou entre a et d, est inversé après la modification).

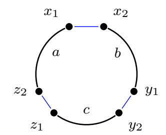
Vérifier qu'une tournée est 2-optimale (qu'il n'existe plus d'échanges 2-opt permettant de diminuer sa longueur) demande un temps en $O(n^2)$.

Une tournée 2-optimale n'est pas forcément de longueur minimale !



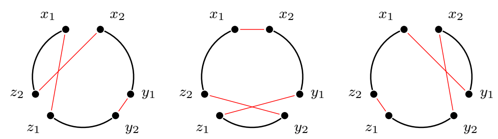
4.3.2 3-opt

Base :



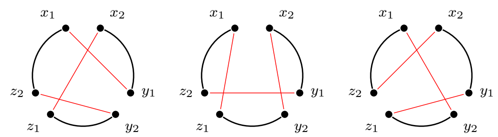
Équivalent à un mouvement 2-opt :

$ac'b' = a'bc$ $\quad$ abc' $\quad$ $ab'c$



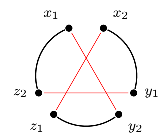
Équivalent à deux mouvements 2-opt :

$ab'c' = a'cb$ $\quad$ $ac'b = a'b'c$ $\quad$ acb'



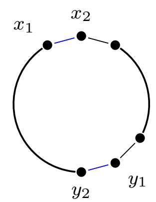
Équivalent à trois mouvements 2-opt :

$acb = a'b'c'$



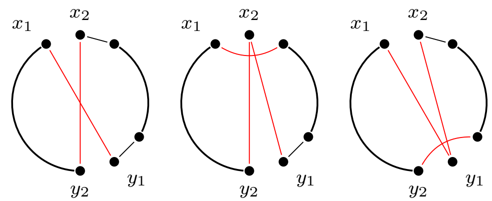
4.3.3 2.5-opt

Base :



Transitions :

Variante A $\quad$ Variante B $\quad$ Variante C



4.3.4 or-opt

Équivalent à 3-opt, sauf que les mouvements sont restreints à l'inversion de seulement un segment, afin de garder la gestion structurelle dans l'ordre linéaire.

Il y a donc seulement les mouvements 3-opt abc', $ac'b$, et acb qui sont acceptés par les contraintes Or-opt.

Partie 2 — Programmation linéaire

1 Formulation d'un programme linéaire

La structure d'un programme linéaire est standard, c'est un problème d'optimisation visant à maximiser (ou minimiser) une fonction objectif linéaire de n variables **réelles** soumises à un ensemble de contraintes linéaires prenant la forme d'équations/inéquations linéaires.
Une équation/inéquation est linéaire si elle est de la forme

$$a_1x_1 + a_2x_2 + \cdots + a_nx_n \left\{ \begin{matrix} \leq \\ = \\ \geq \end{matrix} \right\} b$$

équivalent à

$$\sum_{k=1}^{n} a_kx_k \left\{ \begin{matrix} \leq \\ = \\ \geq \end{matrix} \right\} b$$

1.1 Ordre de définition

1. **Variables de décision** : On note, par exemple, x_{ij} = qté achetée à i au mois j, $i \in \{1,\dots\}, j \in \{A,\dots\}$.
2. **Fonction objectif** : On l'exprime à l'aide des variables de décision, par exemple $z = \sum_{i\in\{1,\dots\}}\sum_{j\in\{A,\dots\}} p_{ij}x_{ij}$ où p_{ij} sont les poids correspondants pour une variable.
3. **Contraintes** : On annonce par exemple $x_{iA}+x_{iC} \leq d_i$, $i \in \{1,\dots\}$ avec d_i un coefficient. On rajoute aussi éventuellement des bornes pour les variables de décision.

Ainsi un programme linéaire à n variables et m contraintes est un problème d'optimisation de la forme

$$\begin{aligned} &\operatorname{Max} z = \sum_{j=1}^{n} c_jx_j \\ &\text{s. c.} \quad \sum_{j=1}^{n} a_jx_j \left\{ \begin{matrix} \leq \\ = \\ \geq \end{matrix} \right\} b_i \qquad i = 1,\dots,m \\ &\qquad\quad l_j \leq x_j \leq u_j \qquad\qquad j = 1,\dots,n \end{aligned}$$

1.2 Terminologie

Représentation dans le plan :
L'ensemble de solutions d'une équation linéaire correspond à une droite dans $\mathbb{R}^2$.
L'ensemble des solutions d'une inéquation linéaire correspond à un demi-plan dans $\mathbb{R}^2$.
L'ensemble des solutions d'un système d'équations et d'inéquations linéaires correspond à l'intersection des demi-plans et des droites associés à chaque élément du système.
Solution d'un problème :
Une **solution** est une affectation de valeurs aux variables du problème.
Elle est **admissible**/réalisable si elle satisfait toutes les contraintes du problème.
La **valeur** d'une solution est la valeur de la fonction objectif.
Le **domaine admissible** D d'un PL est l'ensemble des solutions admissibles.

1.3 Résolution graphique

Les **lignes de niveau** de la fonction objectif sont des droites parallèles dans $\mathbb{R}^2$.
Il existe des solutions admissibles de valeur z si la ligne de niveau associée à cette valeur intersecte le domaine admissible D du problème.
Les **points de contact** ainsi obtenus correspondent aux solutions optimales du PL.

1.3.1 Domaine admissible

Le domaine admissible d'un PL peut être

- **vide**, dans un tel cas le problème est sans solution admissible,
- **borné**, dans un tel cas le problème possède toujours au moins une solution optimale, quelle que soit la fonction objectif,
- **non borné**, dans ce cas selon la fonction objectif choisie, le problème peux posséder plusieurs solutions optimales ou il peut exister des solutions admissibles de valeur arbitrairement grande/petite.

1.4 Forme canonique d'un PL

Un programme linéaire de forme canonique est un problème de **maximisation**, dont les variables sont toutes **non négatives** et dont toutes les autres contraintes sont formulées par une inéquation du **type plus petit ou égal** ($\leq$).

1.4.1 Techniques de transformation

Minimisation $\leftrightarrow$ **Maximisation** : $\min f(x) = -\max(-f(x))$. Pour minimiser $z = \vec{c}^T\vec{x}$, il faut maximiser $w = -\vec{c}^T\vec{x} = (-\vec{c})^T\vec{x}$ et multiplier la valeur optimale de w par -1 pour obtenir celle de z.
Inéquation $\geq \leftrightarrow \leq$:

$$a_1x_1 + \cdots + a_nx_n \geq b \Leftrightarrow -a_1x_1 - \cdots - a_nx_n \leq -b$$

Inéquation $\rightarrow$ inéquation $\leq$:

$$a_1x_1 + \cdots + a_nx_n = b \Leftrightarrow \begin{cases} a_1x_1 + \cdots + a_nx_n \leq b \\ a_1x_1 + \cdots + a_nx_n \geq b \end{cases}$$

Inéquation $\leq \rightarrow$ équation : On ajoute une **variable d'écart**

$$a_1x_1 + \cdots + a_nx_n \leq b \Leftrightarrow \begin{cases} a_1x_1 + \cdots + a_nx_n + s = b \\ s \geq 0 \end{cases}$$

Variable libre $\rightarrow$ non négative :

$$x \in \mathbb{R} \Leftrightarrow \begin{cases} x = x^+ - x^- \\ x^+, x^- \geq 0 \end{cases}$$

Variable bornée inférieurement :

$$x \geq b \Leftrightarrow \begin{cases} x = x' + b \\ x' \geq 0 \end{cases}$$

1.5 Techniques de linéarisation

$$expr_k = \vec{c}_k^T\vec{x} + d_k = \sum_{j=1}^{n} c_{kj}x_j + d_k$$

1.5.1 min / max

Pour $\min(expr_1,\dots,expr_k) \geq expr_0$, on écrira

$$\begin{cases} expr_1 \geq expr_0 \\ \vdots \\ expr_k \geq expr_0 \end{cases}$$

Ainsi pour $\max(expr_1,\dots,expr_k) \leq expr_0$, on écrira

$$\begin{cases} expr_1 \leq expr_0 \\ \vdots \\ expr_k \leq expr_0 \end{cases}$$

Les autres combinaisons ne sont pas linéarisables.

1.5.2 Min-max

Pour $\operatorname{Min} z = \max\{expr_1,\dots,expr_k\}$, on introduit une variable auxiliaire t pour obtenir le système

$$\begin{aligned} \operatorname{Min} z = \quad & t \\ \text{s. c.} \quad & t \geq \max\{expr_1,\dots,expr_k\} \end{aligned}$$

qui se transforme

$$\begin{aligned} \operatorname{Min} z = \quad & t \\ \text{s. c.} \quad & t \geq \vec{c}_1^T\vec{x} + d_1 \\ & \vdots \\ & t \geq \vec{c}_k^T\vec{x} + d_k \\ & t \in \mathbb{R} \end{aligned}$$

Pour un objectif « max-min » :

$$\begin{aligned} \operatorname{Max} z = \quad & t \\ \text{s. c.} \quad & t \leq \vec{c}_1^T\vec{x} + d_1 \\ & \vdots \\ & t \leq \vec{c}_k^T\vec{x} + d_k \\ & t \in \mathbb{R} \end{aligned}$$

1.6 Valeur absolue

La contrainte $|expr_1| \leq expr_2$ est équivalente aux deux inéquations linéaires

$$-expr_2 \leq expr_1 \leq expr_2 \Leftrightarrow \begin{cases} expr_1 \leq expr_2 \\ expr_+ \geq -expr_2 \end{cases}$$

L'objectif $\operatorname{Min} z = |expr_1|$ peut se transformer en

$$\begin{aligned} \operatorname{Min} z = \quad & t \\ \text{s. c.} \quad & -t \leq expr_1 \leq t \\ & t \geq 0 \end{aligned}$$

2 Analyse de sensibilité

L'analyse de sensibilité a pour but d'étudier le comportement de la solution optimale d'un programme linéaire et de sa valeur lorsque l'on modifie légèrement les coefficients apparaissant dans le problème.
Analyse de sensibilité de la fonction objectif : Comment se comporte la solution optimale d'un programme linéaire ainsi que sa valeur lorsque l'on modifie un coefficient de la fonction objectif du problème ?
Analyse de sensibilité du second membre : Comment se comporte la solution optimale d'un programme linéaire ainsi que sa valeur lorsque l'on modifie le second membre (le terme constant) d'une des contraintes du problème ?

2.1 Résultats de l'analyse par GLPK

`St.` Statut

- `BS` contrainte inactive
- `NL` " inégalité, borne inférieure active
- `NU` " inégalité, borne supérieure active
- `NS` " égalité, colonne fixée
- `NF` " active et libre, sans bornes

`Activity` valeur primale de la variable auxiliaire
`Slack` valeur primale de la variable d'écart. Une contrainte est active/saturée si la variable d'écart associée est nulle
`Marginal` valeur du coût marginal de la variable auxiliaire
`Bornes d'activité` les bornes de l'intervalle $[a, b]$ dans lequel le second membre (b_i) peut-être modifié tout en conservant la solution optimale actuelle
`Obj. val. brk. point` les valeurs de la fonction objectif lorsque la valeur du second membre atteint une des bornes
Le coût marginal est égal à la variation, en plus ou en moins, de la valeur de la fonction objectif si la valeur du second membre est modifiée d'une unité, en plus ou en moins.
Lors de la modification d'un second membre d'une contrainte non active à l'optimum, l'intérvalle de variation doit se calculer différemment ; on prendra donc valeur actuelle − valeur d'écart et une borne supérieure à $+\infty$.

3 Programmation entière

3.1 Couplage maximum

Pour un graphe $G = (V, E)$ non orienté (simple et connexe) dans lequel on cherche un **couplage maximum**.
Variables de décision

$$x_{ij} = \begin{cases} 1 \text{ si l'arête } \{i,j\} \text{ appartient au couplage} \\ 0 \text{ sinon.} \end{cases}$$

On distinguera que $x_{ij} = x_{ji}$.
Fonction objectif
Maximiser le nombre de sélections : $z = \sum_{\{i,j\}\in E} x_{ij}$
Contraintes
Max 1 incidence

$$\sum_{j\in\operatorname{Adj}[i]} x_{ij} \leq 1 \, \forall i \in V$$

3.2 Brocanteur

Sélection d'un certain nombre de bibelots sous contrainte de poids maximum.

Variables de décision

$$x_i = \begin{cases} 0 \text{ si le bibelot } i \text{ est sélectionné} \\ 0 \text{ sinon.} \end{cases}$$

Fonction objectif
Maximiser le bénéfice

$$z = \sum_{i \in B} b_i x_i$$

Contraintes
Poids maximum

$$\sum_{i \in B} p_i x_i \leq P_{\max}$$

Programme linéaire à résoudre

$$\begin{aligned} \text{Min}\, z = \quad & \sum_{i \in B} b_i x_i \\ \text{s. c.} \quad & \sum_{i \in B} p_i x_i \leq P_{\max} \\ & x_i \in \{0,1\} \quad \forall i \in V \end{aligned}$$

3.3 Noyau d'un graphe

Variables de décision

$$x_v = \begin{cases} 1 \text{ si le sommet } v \in V \text{ est sélectionné} \\ 0 \text{ sinon.} \end{cases}$$

Fonction objectif
Minimiser le cardinal

$$z = \sum_{v \in V} x_v$$

Contraintes

— Ensemble stable, donc les sommets sont disjoints deux à deux, en prenant toutes les arêtes du graphe on a : $x_i + x_j \leq 1 \, \forall \{i,j\} \in E$
— Ensemble dominant, un sommet ou un voisin doit être dans le noyau : $x_i + \sum_{j \in \text{Adj}[i]} x_j \geq 1 \, \forall i \in V$

Programme linéaire à résoudre

$$\begin{aligned} \text{Min}\, z = \quad & \sum_{v \in V} x_v \\ \text{s. c.} \quad & x_i + x_j \leq 1 && \forall \{i,j\} \in E \\ & x_i + \sum_{j \in \text{Adj}[i]} x_j \geq 1 && \forall i \in V \\ & x_i \in \{0,1\} && \forall i \in V \end{aligned}$$

3.4 Contraintes additionnelles

3.4.1 Au plus 2 produits

Variables auxiliaires

$$y_i = \begin{cases} 1 \text{ si le produit } i \text{ est choisi} \\ 0 \text{ sinon.} \end{cases}$$

Contraintes additionnelles

— Vérifier max 2 produits $\sum_i y_i \leq 2$
— Assigner y_i pour x_i : $x_i \leq M y_i, \, \forall i$

3.4.2 Ne pas investir dans a et b en même temps

Variables auxiliaires

$$y_i = \begin{cases} 1 \text{ si le produit } i \text{ est choisi} \\ 0 \text{ sinon.} \end{cases}$$

Contraintes additionnelles

— Vérifier 1 des 2 produits $y_a + y_b \leq 1$
— Assigner y_i pour x_i : $x_i \leq M y_i, \, i \in \{a, b\}$

3.4.3 Si a, alors b

Variables auxiliaires

$$y_i = \begin{cases} 1 \text{ si le produit } i \text{ est choisi} \\ 0 \text{ sinon.} \end{cases}$$

Contraintes additionnelles

— Vérifier $y_a \leq y_b$ (si a, alors b)
— Assigner y_i pour x_i : $x_i \leq M y_i, \, i \in \{a, b\}$

4 Modélisation entière

4.1 Variables discrètes

Une variable x **bivalente** définit $x \in \{a, b\}$. On la modélise à l'aide d'une variable $y \in \{0,1\}$.

$$x \in \{a,b\} \Leftrightarrow \begin{cases} x = a + (b-a)y \\ y \in \{0,1\} \end{cases}$$

Dans le cas général, une variable $x \in \{a_1, \ldots, a_q\}$ s'écrit :

$$x \in \{a_1, \ldots, a_q\} \Leftrightarrow \begin{cases} x = \sum_{k=1}^q a_k y_k \\ \text{où } \sum_{k=1}^q y_k = 1 \\ \text{et } y_k \in \{0,1\}, \; k = 1, \ldots, q \end{cases}$$

4.2 Fonction objectif à coûts fixes

Si le coût associé à l'achat de x unités est composé d'un **coût fixe** $K > 0$ à payer dès que $x > 0$ et d'un coût variable $cx \propto \#x$.
On modélise avec une variable binaire $y = 1$ si $x > 0$ sinon 0 et M une borne supérieure sur la valeur de x.

$$\begin{aligned} \text{Min}\, z = \quad & Ky + cx \\ \text{s. c.} \quad & \ldots \\ & x \leq My \\ & 0 \leq x \leq M \\ & y \in \{0,1\} \end{aligned}$$

$x \leq My$ permet d'assigner la valeur 1 à y si x a une valeur positive. Si $y = 1$ alors le coût fixe est forcé.

4.3 Variables semi-continues

Une variable est **semi-continues** si sa valeur est soit nulle soit varie dans l'intervalle $[a, b]$.
On ajoute une variable binaire $y = 1$ si $x \in [a, b]$ sinon 0 et une variable continue $0 \leq t \leq 1$. Ainsi :

$$x \in \{0\} \cup [a,b] \Leftrightarrow \begin{cases} x = ay + (b-a)t \\ t \leq y \\ 0 \leq t \leq 1 \text{ et } y \in \{0,1\} \end{cases}$$

Une généralisation est possible en ajoutant les variables y_i et t_i pour chaque intervalle et une contrainte $\sum_i y_i \leq 1$.

4.4 Contraintes disjonctives

En considérant deux contraintes linéaires dont **une des deux** au moins doit être satisfaite, on donne la représentation suivante :