Chapitre 6: Meta-heuristiques sur une solution (Algorithmes de recherche locale)

F. Bannay
(d'après les supports de J. Mengin et P. Muller,
et le livre (Talbi 2009))

2019-2020

Les deux méthodes itératives vues dans la partie B

Méthode **itérative** : on part d'une solution complètement décrite et on la fait évoluer vers une meilleure.

- Algo de Ford-Fulkerson :
```
• on part d'un flot compatible
Repeat
    • marquage: chaîne augmentante
    ↪ nouveau flot compatible de valeur supérieure
Until on ne peut plus augmenter le flot
```
- Algo du Simplexe :
```
• on part d'une base réalisable
Repeat
    • une variable sort et une variable rentre
    ↪ nouvelle base réalisable avec meilleur gain
Until on ne peut plus augmenter le gain
```


Les méthodes constructives

- Méthode **constructive** : on part d'une solution vide et on affecte les variables une à une jusqu'à obtenir une solution complètement décrite.
- Algo de Kruskal (arbre couvrant de poids min)
```
• on part d'un ensemble vide d'arêtes
Repeat
    • sélectionner une arête de poids minimum
    • l'ajouter si pas de cycle sinon la jeter
Until graphe connexe
```


Méthodes Exactes vs Incomplètes

- Méthode exacte :
  - si solution optimale existe alors l'algo la trouve.
  - exploration systématique de l'espace de recherche
- Pour de nombreux problèmes : pas de méthode exacte efficace.
- Problème d'explosion combinatoire
  - ex : voyageur de commerce avec n villes

  ↪ $n!$ séquences possibles
- Idée :
  - méthodes incomplètes **suffisamment** efficaces dans certains cas

  ↪ les **meta-heuristiques**.

Méthodes Exactes vs Incomplètes (suite)

Méthodes exactes	Méthodes Incomplètes
■ Programmation dynamique ■ Branch and bound/ cut / price ; A* (heuristique), IDA* ■ Programmation par contraintes ; Programmation Linéaire ; Flots ■ ...	■ Algo d'approximation : garantie écart solution optimale $\leq \varepsilon$ ■ Algo Heuristiques : spécifiques / meta-heuristiques (une solution/ une population) ■ ...

Quand utiliser des meta-heuristiques ?

- Complexité du problème et taille des instances

Taille instance / **Complexité pb**	**Petite**	**Grande**
Faible	exacte	exacte ou incomplète (temps réel)
Forte	exacte (peut suffire) ou incomplète	incomplète

- Structure des instances : des structures spécifiques peuvent être résolues par des méthodes exactes
- Moralité :
  1. étudier complexité
  2. réduire à un problème connu
  3. estimer faisabilité selon taille instance

Meta-heuristiques sur une solution : recherche locale

Algorithme générique de **recherche locale** :

```
• on part d'une solution s
Repeat
   • choisir un voisin s' de s
   • si s' meilleure_que s alors s ← s'
Until s suffisamment_bon ou time-out
```

solution : affectation de toutes les variables

voisin : solution obtenue par "petite" modification ("locale")

meilleure_que/suffisamment_bon : selon critère(s) à optimiser et tolérance (acceptable)

essai : séquence de choix de voisins

time-out : nombre max de voisins choisis atteint ou plus de voisins

Meta-heuristiques sur une solution : recherche locale

Recherche locale
F. Bannay
Introduction
Algo R. Locale
Paramètres
Solutions
Voisinage
Hill-Climbing
Optima locaux
Conclusion
Références

Algorithme générique de **recherche locale** :

```
• on part d'une solution s
• nb_choisis=0
Repeat
   • choisir un voisin s' de s
   • si acceptable_voisin(s',s) alors s ← s'
   • nb_choisis ++
Until acceptable_solution(s) ou nb_choisis=MAX ou vois(s)=∅
```

solution : affectation de toutes les variables

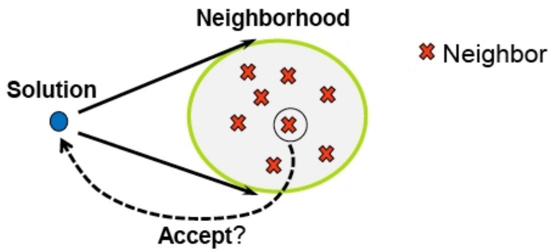
voisin : solution obtenue par "petite" modification ("locale")

meilleure_que/suffisamment_bon : selon critère(s) à optimiser et tolérance (acceptable)

essai : séquence de choix de voisins

time-out : nombre max de voisins choisis atteint ou plus de voisins

Les paramètres de la recherche locale

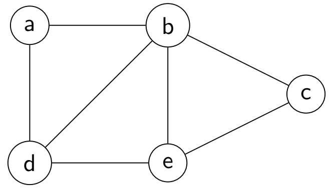


Paramètres à définir :

- représentation de la solution,
- choix d'une solution initiale
- choix d'un voisin,
- fonction d'évaluation de la solution (`acceptable_solution`)
- fonction d'évaluation d'un voisin (`acceptable_voisin`)
- voisinage = la génération des voisins

Illustration de “solution” sur 3 problèmes

- $\max \sum_{i=1}^{N} \sum_{j=1}^{N} (-1)^{3i+j}.(2)^{i+7j} T[i].T[j]$ où T chaîne binaire de taille N
- coloration de graphe t.q. somme des couleurs est min



- voyageur de commerce avec N villes et une matrice $N \times N$ des distances

Choix d'un voisin

Les choix de voisins couramment utilisés :

rapide calculer un seul voisin (n'améliore pas forcément)

plus long énumérer tous les voisins, garder le meilleur

intermédiaire générer échantillon de voisins, garder le meilleur

premier qui améliore calculer le premier voisin qui améliore (pire cas : explore tous les voisins)

amélioration aléatoire sélectionner aléatoirement un voisin parmi ceux qui améliorent

Illustration de voisinage (1)

Voisinage de solutions binaires : distance de Hamming

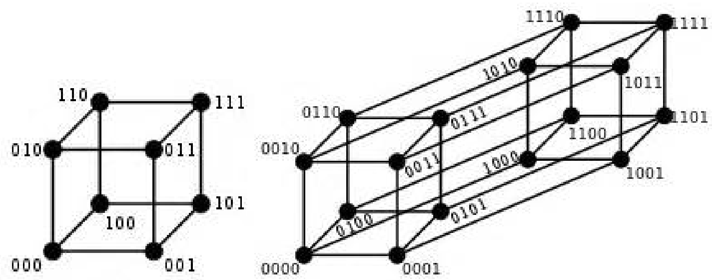


Illustration de voisinage (2)

Voisinage de la solution C dans espace de dimension 2 : disque.
(Dans $\mathbb{R}^n$ c'est une boule de dimension n).

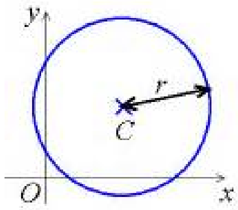
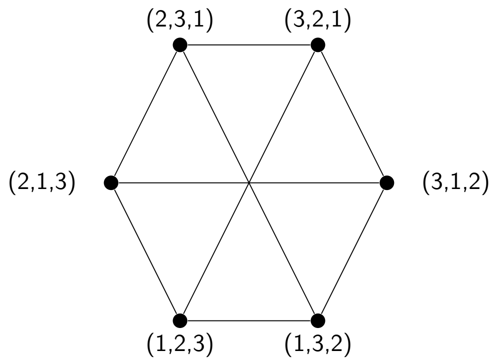


Illustration de voisinage (3)

Voisinage pour un problème de permutation de taille 3 (échange de 2 éléments)



Exemple de recherche locale : le hill-climbing

Hill-climbing (ou *descente en gradient* si min recherché)

- choisir_un_voisin : choix aléatoire voisin
- acceptable_voisin(s',s) : si s' meilleur que s

méthode opportuniste

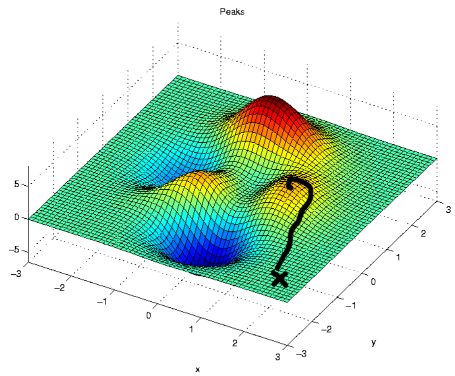
Hill-climbing version glouton

- choisir_un_voisin : choix d'un voisin qui améliore s
- acceptable_voisin(s',s) : toujours

[...]

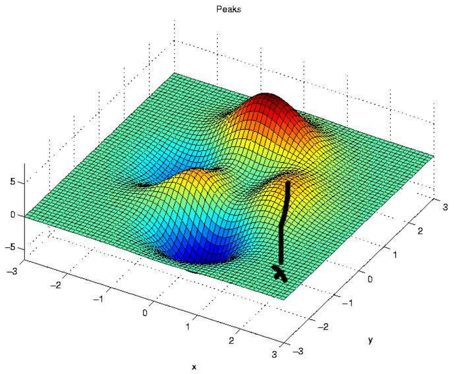
Hill-Climbing Glouton

En montant de façon à améliorer (hill-climbing glouton)



Steepest Hill-Climbing

En montant la plus forte pente (steepest hill-climbing) : meilleur voisin



Optimum local/global

Un **optimum local** est une solution dont tous les voisins sont moins bons.

- Problème important : un **optimum local** n'est pas nécessairement un optimum global
  - un optimum local en Programmation Linéaire ou pour les flots est un optimum global
  - ce n'est pas le cas pour les problèmes d'optimisation combinatoire en général
  - la recherche locale peut se retrouver coincée dans un optimum local (tous ses voisins sont moins bons que lui)
- moins de chance d'en rencontrer si voisinage grand
- mais grand voisinage = modifs complexes et plus de voisins (à l'extrême : voisinage = tout l'espace de recherche...)

Illustration dans le cas de fonctions continues : 1D

Descente en gradient (hill-climbing glouton)

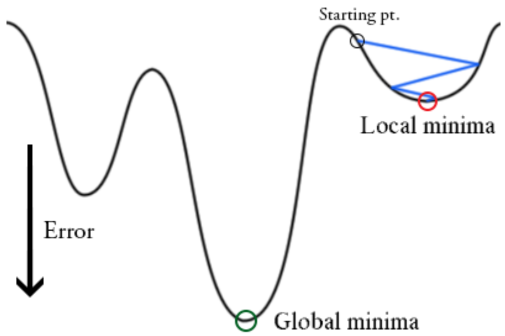
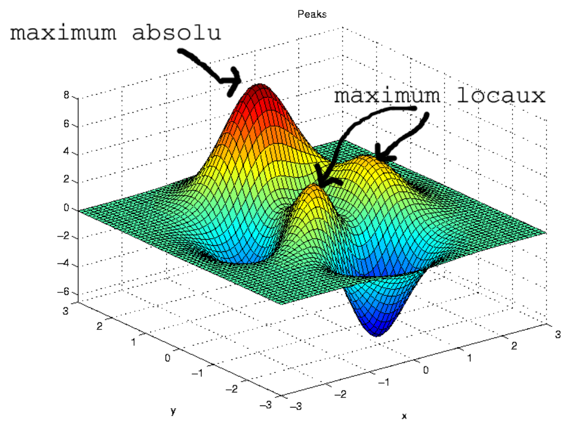


Illustration dans le cas de fonctions continues : 2D



Échapper aux optima locaux

Quatre famille d'approches :

1. Réitérer depuis différentes solutions initiales :
   - recherche locale avec redémarrage
   - recherche locale itérée
   - GRASP
   - ...
2. Accepter des voisins moins bons :
   - recherche tabou
   - recuit simulé
3. Changer le voisinage pendant la recherche
   - recherche à voisinage variable
4. Changer la fonction objectif ou les contraintes :
   - recherche locale guidée
   - stratégie de lissage (smoothing)
   - méthodes bruitées (noisy methods)

Échapper par redémarrage

Recherche locale **avec redémarrages** : succession d'essais

```
• nb_essais=0
Repeat /* faire un essai */
    • choisir nouvelle solution initiale s                         (#1)
    • nb_choisis=0
    Repeat
        • choisir un voisin s' de s
        • si acceptable_voisin(s, s') alors s ← s'
        • nb_choisis ++
    Until acceptable_solution(s) ou nb_choisis=MAX ou vois(s)=∅
    • nb_essais ++
Until acceptable_solution(s) ou nb_essais=MAX2
```

à l'étape (#1) : génération de solutions aléatoires
↪ ne conduira pas forcément au même optimum local.

time-out(recherche locale) : fonction du nombre d'essais.

Recherche locale itérée : GRASP

Le GRASP est basé sur un algorithme constructif puis une recherche locale.

- l'algorithme constructif doit pouvoir être randomisé pour générer différentes solutions initiales
- on lance ensuite une recherche locale depuis une solution initiale
- et on répète
- itérations indépendantes : pas de mémoire de recherche.

La lutte contre les optima locaux : mémoire

Mémoire à court terme = **liste tabou**

- Pour sortir d'un optimum local : autoriser le passage par une solution moins optimale.
- le meilleur voisin remplace s même s'il est moins bon
- pb : on risque de retomber immédiatement après dans le même optimum local
- mémoriser à l'aide d'une liste tabou les k solutions récemment explorées, où ça n'est pas la peine de retourner.
- en pratique : mémoriser les k derniers mouvements (moins coûteux mais parfois moins précis, en réalité : stocker l'inverse du mouvement)
- détermination de la taille de la liste par expérimentation

Algorithme d'une recherche tabou


```
• s ← s0                       /* solution initiale*/
• Tabou ← []
Repeat
    • choisir meilleur voisin non Tabou s' de s
    • Tabou ← Tabou + {s}
    • s ← s'
Until condition d'arrêt
```

choisir_voisin(s) : meilleur voisin non Tabou

acceptable_voisin(s', s) : toujours

condition d'arrêt
- soit limite en temps ou en itérations,
- soit condition de non amélioration de la solution.

liste Tabou : implémentée en FIFO de taille k

Améliorations du Tabou

- Critères d'aspiration : conditions permettant de lever le tabou
  - stockage des mouvements dans Tabou peut interdire des solutions intéressantes.
  - ↪ test à ajouter pour autoriser à considérer des solutions dans les voisins non tabous (ex : un mouvement qui génère une solution meilleure que tout ce qu'on a trouvé)
- Intensification : mémoire à moyen-terme qui stocke **l'élite** (meilleures solutions rencontrées durant la recherche)

↪ biaiser la recherche pour favoriser les solutions ayant des attributs commun avec l'élite

- Diversification : mémoire à long-terme qui stocke les solutions visitées

↪ explorer des zones non visitées.

Algorithme d'une recherche tabou amélioré

Ajout des mémoires à moyen et long-terme et des critères d'aspiration.

```
• s ← s0                       /* solution initiale*/
• Tabou ← []
• Moyen-Terme ← []
• Long-Terme ← []
Repeat
    • choisir_voisin_aspi(Tabou) s' de s
    • Tabou ← Tabou + {s}
    • s ← s'
    • mettre à jour Moyen-Terme et Long-Terme
    • If critère_intensification Then intensification
    • If critère_diversification Then diversification
Until condition d'arrêt
```

choisir_voisin_aspi(s) : meilleur voisin non Tabou ou aspirable

Le recuit simulé

- Méthode inspirée de la métallurgie : recuire un métal pour atteindre un état stable.
- Énergie E du système = fonction à minimiser
- Température T = paramètre contrôlant la diversification

↪ T élevée/basse : instabilité/stabilité

↪ on part de T élevée pour que chaque mouvement ait une probabilité d'acceptation de plus de 50% (mouvement très chaotique)

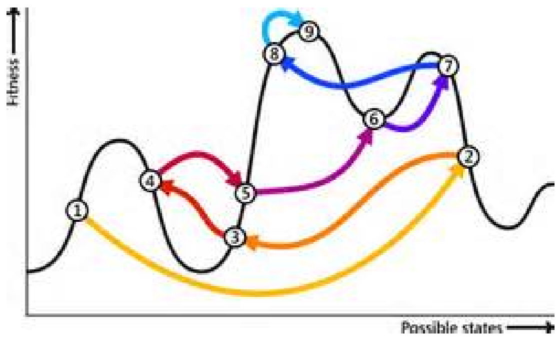
- on refroidit lentement par plateaux (réduction du bruit jusqu'à ne plus accepter de solutions plus mauvaises que s).

Le recuit simulé : Algorithme


```
• s ← s0 solution initiale
• T ← T_MAX température de départ
Repeat
  Repeat /* à une température fixée*/
    • s' ← voisin de s
    • calculer la variation d'énergie Δ_E = f(s') − f(s)
    •  If Δ_E < 0 Then acceptable_voisin (s', s) ← 1
       Else acceptable_voisin(s', s) ← e^(−Δ_E / T)
    • If acceptable_voisin(s', s) Then s ← s'
  Until condition d'équilibre
  • diminuer température: T ← λT avec λ < 1
Untill critère d'arrêt (ex: T < T_min)
```

acceptable_voisin(s',s) $=$ probabilité d'accepter s' depuis s

Illustration du recuit simulé



Réglage des paramètres

- valeur de T_initiale : si la valeur T_init permet d'accepter x% des configurations de départ, alors on la garde sinon on la double et on recommence (x au moins plus de 50, certains auteurs : 80).
- la durée des paliers (longueur du plateau) : le plus simple est de le borner par le nombre de configurations atteignables en un mouvement élémentaire. là encore subtilités possibles
- la décroissance de la température : le plus simple est facteur constant, de l'ordre de 0.9 / 0.95 (décroissance lente). pour raffiner, le réglage se joue entre ce paramètre (+/- rapide) et le précédent (plateau +/- long).
- condition d'arrêt : la température mini ; soit 0 mais pas très efficace, soit quand améliorations très petites (très empirique).

Relaxation du problème : le lissage

- transformer le problème en cachant des optima locaux
- trouver l'optimum s dans le paysage des solutions le plus lisse
- faire une recherche locale à partir de s dans le problème initial
- ou faire une recherche locale dans des variantes de moins en moins lissées du pb initial

Illustration du smoothing

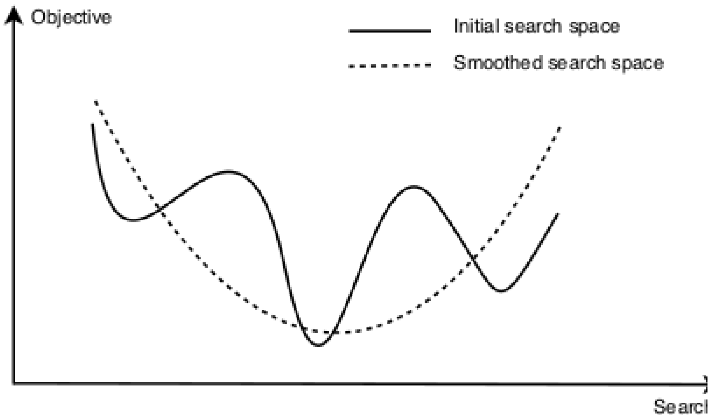
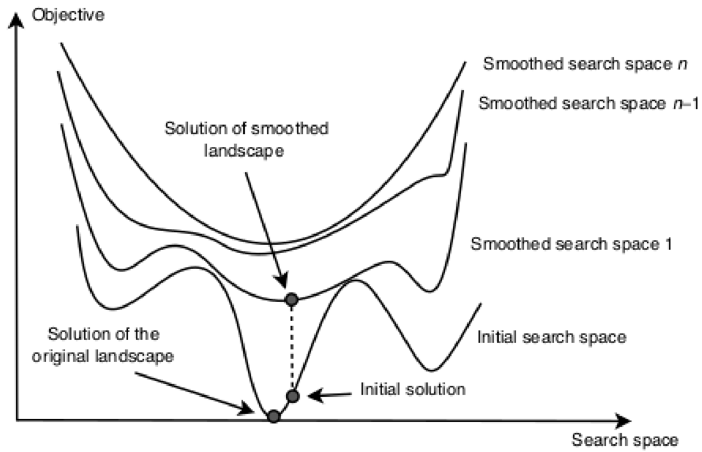


Illustration du smoothing (2)



Méthodes bruitées

Pour éviter d'être coincé dans un optimum local, on peut ajouter du **bruit** à la fonction objectif ou au choix du mouvement.

- le bruit est choisi aléatoirement dans l'intervalle $[-r, +r]$
- à chaque itération le bruit est réduit (l'intervalle est réduit)

Principe du bruit sur voisinage :

- avec une proba p, faire un mouvement aléatoire.
- avec une proba $1 - p$, suivre la méthode originale.

Reste le problème : comment régler p ?

Conclusion Chapitre 6 : Les méthodes locales en question(s)

Comportement général :

1. une majorité de mouvements améliorent la solution courante.
2. le nombre d'améliorations diminue.
3. améliorations terminées : optimum, peut-être local.

Les questions à se poser :

- quand faut-il s'arrêter : faut-il être opportuniste ou gourmand ?
- comment ajuster les paramètres nombre d'essais/nombre de mouvements ?
- comment comparer les performances de deux méthodes différentes ? (qualité de la solution vs. temps consommé)

Avantages/Inconvénients des recherches locales

Recherche locale

F. Bannay

Introduction
Algo R. Locale
Optima locaux
Conclusion
Références

Avantages :

- Méthodes rapides (et temps paramétrable)
- Faciles à implémenter
- Donnent souvent de bonnes solutions
- Fonctionnement intuitif

Inconvénients :

- Manque de modélisation mathématiques (chaque cas est différent : il faut adapter la recherche à chaque problème particulier)
- Difficiles à paramétrer (= bricolage)
- Aucune évaluation de la distance à l'optimum (pas une approximation, trouve au pire un optimum local qui n'a rien à voir)
- **video** `https://www.youtube.com/watch?v=SC5CX8drAtU`

Conclusion Générale

Recherche locale

F. Bannay

Introduction
Algo R. Locale
Optima locaux
Conclusion
Références

- plein d'autres Structures de données intéressantes (kd-trees, dynamic trees...)
- plein d'autres méthodes incomplètes (A^* incomplet, algorithmes génétiques, colonies de fourmi, essaim de particules, méthodes hybrides globales/locales...)
- les algorithmes de flot et de programmation linéaire sont toujours l'objet d'optimisations algorithmiques
- des outils :
  - Structure de données : `https://people.ksp.sk/~kuko/gnarley-trees`
  - Flots (MatLab, Pseudoflow solver (Chandran and Hochbaum 2007 `http://riot.ieor.berkeley.edu/Applications/Pseudoflow/maxflow.html`)...
  - Programmation Linéaire : `http://www.zweigmedia.com/RealWorld/simplex.html`
  - Meta-heuristiques : outils non commerciaux (ParadisEO, ECj, oMetah) et commerciaux (CPLEX, Mathematica)

Recherche locale

F. Bannay

Introduction
Algo R. Locale
Optima locaux
Conclusion
Références

Ahuja, R., Magnanti, T., Orlin, J., and Reddy, M. (1995).
Applications of Network Optimization, volume 7, chapter 1.
Elsevier Science B.V.

Alj, A. and Faure, R. (1990).
Guide de la recherche opérationnelle : Les applications.
Masson.

Busacker, R. and Gowen, P. (1961).
A procedure for determining minimal-cost network flow patterns.
Technical Report ORO-15, Operational Research Office, John Hopkins University.

Cormen, T., Leiserson, C., Rivest, R., and Stein, C. (2009).
Introduction to Algorithms.

Recherche locale

F. Bannay

Introduction
Algo R. Locale
Optima locaux
Conclusion
Références

MIT Press.

Dinic, E. A. (1970).
Algorithm for solution of a problem of maximum flow in a network with power estimation.
Soviet Math. Doklady, 11 :1277–1280.

Edmonds, J. and Karp, R. M. (1972).
Theoretical improvements in algorithmic efficiency for network flow problems.
Journal of the ACM, 19(2) :248–264.
doi :10.1145/321694.321699.

Faure, R., Lemaire, B., and Picouleau, C. (2000).
Précis de Recherche Opérationnelle.
Dunod, Paris.

Ford, L. and Fulkerson, D. (1955).
A simplex algorithm finding maximal networks flows and an application to the hitchock problem.

Rand Report Rand Corporation.

Fournier, J. (2011).
Théorie des graphes et applications : Avec exercices et problèmes.
Collection Informatique. Hermes Science Publications.

Gondran, M. and Minoux, M. (2009).
Graphes et Algorithmes (4e ed.).
Lavoisier.

Hochbaum, D. S. (2004).
Selection, provisioning, shared fixed costs, maximum closure, and implications on algorithmic methods today.
Management Science, 50(6) :709–723.

Klein, M. (1967).
A primal method for minimal cost flows with applications to the assignment and transportation problems.
Management Science, 14 :205–220.

Kleinberg, J. and Tardos, E. (2005).
Algorithm Design.
Addison Wesley.

Maurin, H. (1967).
Programmation linéaire appliquée.
Éditions Technip.
374 pages.

Orlin, J. B. (2013).
Max flows in o(nm) time, or better.
In *Proceedings of the Forty-fifth Annual ACM Symposium on Theory of Computing*, STOC '13, pages 765–774, New York, NY, USA. ACM.

Papadimitriou, C. and Steiglitz, K. (2013).
Combinatorial Optimization.
Dover Publications Inc.

Roy, B. (1969).

Algèbre moderne et théorie des graphes.
Dunod, Paris.

Talbi, E.-G. (2009).
Metaheuristics : from design to implementation.
Wiley.

Wikipédia (2007).
Recherche opérationnelle.
Wikipédia, l'encyclopédie libre, page
`http://fr.wikipedia.org/w/index.php?title=Recherche_op%C3%A9rationnelle&oldid=17837831`.
[En ligne ; Page disponible le 14-août-2007].