Résolution de problèmes difficiles : algorithmes d'approximation, algorithmes probabilistes, heuristiques et métaheuristiques

Table des matières

Il existe de nombreux problèmes *intéressants* — ayant des applications pratiques importantes, par exemple, production d'itinéraires ou d'horaires de transport, placement de composants électroniques dans un circuit, conception de réseaux, etc. — pour lesquels il n'existe aucun algorithme connu ayant un temps d'exécution *polynomial*. En d'autres mots, pour de nombreux problèmes, les seuls algorithmes connus qui permettent d'obtenir *une* solution *optimale* sont de complexité asymptotique *très* élevée, c'est-à-dire de complexité exponentielle ($\Theta(2^n)$) ou factorielle ($\Theta(n!)$).

En pratique, malheureusement, les instances réelles de plusieurs de ces problèmes difficiles peuvent souvent atteindre plusieurs centaines, sinon des milliers, d'éléments. La recherche d'une solution exacte optimale à l'aide d'algorithmes de complexité exponentielle ou factorielle est donc impensable, à moins d'être très patient et d'avoir beaucoup de ressources de calcul — en 1997, le record pour le problème du voyageur de commerce était pour une instance comptant 7397 villes, résultat qui avait demandé plus de *trois années* de temps CPU sur un réseau de stations Sun, alors que le record actuel (2004) est de 24,978 villes [1].

La théorie des problèmes NP-difficiles,[2] que nous n'aborderons pas dans le cadre de ce cours, porte sur cette classe de problèmes pour lesquels il n'existe aucun algorithme connu qui soit de complexité polynomiale. Tous ces problèmes ont la particularité que si jamais une solution polynomiale efficace était trouvée *pour un de ces problèmes*, alors on obtiendrait aussitôt une solution efficace *pour tous les autres problèmes de cette classe*. Bien que cela n'ait encore jamais été prouvé formellement, la conjecture généralement acceptée veut qu'il sera toujours *impossible* d'obtenir un algorithme efficace pour ces problèmes.

Pour ces problèmes difficiles, puisqu'il est impossible d'obtenir une solution exacte en temps raisonnable, on va donc souvent accepter d'utiliser des algorithmes qui produiront des solutions "*pas trop mauvaises*", donc pas nécessairement optimales, mais qui pourront toutefois le faire *en temps raisonnable* (polynomial). C'est ce genre d'algorithmes, pour lesquels il existe diverses catégories, que nous allons examiner dans les sections qui suivent.

0 Les différents classes d'algorithmes d'approximation et d'heuristiques

Notons tout d'abord que la terminologie exacte employée varie souvent selon les auteurs. En gros, on peut catégoriser comme suit les différents types d'algorithmes permettant d'obtenir des solutions pas trop mauvaises, mais non exactes, à des problèmes d'optimisation :

1. Les algorithmes d'approximation :

   Un *algorithme d'approximation*, plutôt que résoudre de façon exacte un problème et trouver une solution optimale, permet d'obtenir une solution *pas trop éloignée* de la solution optimale, donc une solution qui *approxime* la solution optimale.

   Notons qu'on réserve habituellement le qualificatif *algorithme d'approximation* aux algorithmes pour lesquels on est capable de définir de façon précise la notion de "pas de trop éloignée de la solution optimale" ou "solution pas trop mauvaise". On verra quelques exemples à la Section 1.

2. Les algorithmes heuristiques :

   Un algorithme heuristique, ou plus simplement une *heuristique*, produit aussi une solution qui approxime la solution optimale. Toutefois, contrairement à un algorithme d'approximation, il n'est pas nécessairement possible de borner de façon exacte la qualité de la solution produite relativement à la solution optimale. Ainsi, il peut même

[1] Pour plus d'informations, voir le site *web* `http://www.tsp.gatech.edu`.

[2] On parle de problèmes NP-complets dans le cas de *problèmes de décision*. Un problème d'optimisation qui correspond à un problème de décision NP-complet est dit NP-difficile (*NP-hard*).

exister certains cas où l'heuristique produira une solution très mauvaise, ou encore sera tout à fait incapable de produire une solution.

Selon la ou les techniques utilisées pour obtenir les solutions, les heuristiques peuvent être classées en deux principales sous-catégories [DS04] :

- Les méthodes *constructives* : Ces méthodes génèrent des solutions de façon *incrémentale*, donc à partir d'une solution initialement vide à laquelle, petit à petit, sont ajoutés des éléments jusqu'à ce qu'une solution complète soit obtenue.
- Les méthodes de *fouilles locales* : Ces méthodes partent d'une solution initialement complète (mais possiblement peu intéressante) et, de façon répétitive, tente *d'améliorer* cette solution en explorant son voisinage immédiat.

3. Les algorithmes probabilistes :

   Les algorithmes probabilistes ont la particularité que, lorsqu'ils sont confrontés à un choix, plutôt que "perdre du temps" à essayer de déterminer l'alternative qui semble la meilleure, ils effectuent plutôt un choix *aléatoire*, en espérant que le temps ainsi sauvé permettra, en moyenne, d'obtenir quand même un bon résultat, et ce en temps raisonnable. De tels algorithmes ont alors un comportement *non déterministe*, terme que nous expliquerons plus en détail ultérieurement. De nombreuses heuristiques qui ne sont pas des algorithmes d'approximation sont des algorithmes non déterministes. Toutefois, comme on le verra ultérieurement, un algorithme non déterministe n'est pas nécessairement une heuristique, c'est-à-dire, que certains algorithmes probabilistes peuvent produire, de façon certaine, des solutions exactes.

4. Les métaheuristiques :

   Une *métaheuristique* est une stratégie *générale* de résolution de problème qui utilise, qui coordonne ou guide d'autres heuristiques pour obtenir une solution à un problème difficile. Alors que les heuristiques sont généralement *spécifiques* à un problème, les métaheuristiques sont conçues pour s'appliquer à divers problèmes, évidemment en spécialisant certaines parties :

   > *A* metaheuristic *is a set of algorithmic concepts that can be used to define heuristic methods applicable to a wide set of different problems. [...] A metaheuristic is therefore a general algorithmic framework which can be applied to different optimization problems with relatively few modifications to make them adapted to a specific problem [DS04, p. 33]*

   Comme on le verra plus loin (Section 4), de nombreuses métaheuristiques reposent sur des analogies avec des processus naturels (physique, biologie, éthologie).

En résumé, un algorithme d'approximation est une forme d'heuristique, mais pour lequel on peut borner la différence par rapport à la solution optimale. Un algorithme heuristique, s'il repose sur l'utilisation de nombres aléatoires, est aussi un algorithme probabiliste (généralement de type Monte Carlo). Finalement, un algorithme probabiliste peut aussi être un algorithme produisant un résultat exact (algorithme aléatoire de type Las Vegas).

Dans ce qui suit, nous examinons plus en détails, bien que de façon non exhaustive, quelques exemples d'algorithmes d'approximation (Section 1), d'heuristiques (Section 2), d'algorithmes probabilistes (Section 3) et de métaheuristiques (Section 4).

1 Les algorithmes d'approximation

Lorsque le problème est trop "difficile", plutôt que tenter de trouver une solution optimale *exacte*, on peut essayer de trouver une solution qui *approxime* la solution optimale. La solution

ainsi obtenue pourra ensuite être utilisée comme solution finale, ou encore être utilisée comme (un bon) point de départ d'un autre algorithme exact (par exemple, une borne plus précise pour un algorithme *branch-and-bound*).

On réserve habituellement le qualificatif *algorithme d'approximation* aux algorithmes pour lesquels on est capable de définir de façon précise la notion de "pas de trop éloignée de la solution optimale" ou "solution pas trop mauvaise", notion qui peut être formalisée comme suit.[3]

Définition 1 (k-approximation) *Soit un problème d'optimisation pour lequel on doit* minimiser *la valeur de la solution. Soit A un algorithme d'approximation pour ce problème. Soit I une instance de ce problème pour lequel on veut calculer une approximation. Notons alors comme suit les valeurs des différentes solutions obtenues pour I :*

- *$Opt(I)$ la valeur de la solution* optimale.
- *$A(I)$ la valeur produite par l'algorithme A.*

Soit k un nombre réel supérieur à 1. On dit que l'algorithme A produit une k-approximation si, pour toute instance I, on a la relation suivante entre la valeur de la solution produite par l'algorithme A et celle produite par la solution optimale :

$$A(I) \leq k \times Opt(I)$$

1.1 Exemple : le problème du voyageur de commerce géométrique (avec inégalité du triangle) et l'arbre de recouvrement

Pour le problème du voyageur de commerce, l'approximation basée sur l'arbre de recouvrement minimal produit une 2-approximation, c'est-à-dire produit un circuit hamiltonien dont la valeur est *au plus deux (2) fois* le poids d'un circuit optimal.

1.2 Exemple : le problème du *bin packing*

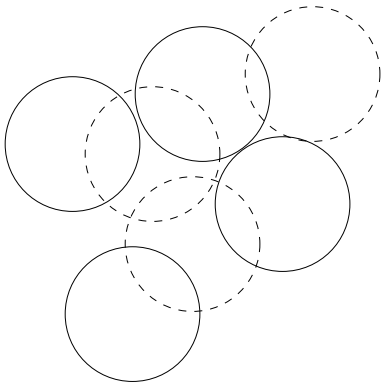
Voir exercices.

1.3 Exemple : identifier le plus grand ensemble indépendant de disques

Remarque : La partie qui suit est adaptée des transparents produits par Cédric Chauve dans le cadre du cours INF7440 de l'automne 2004.

Le problème. On a un ensemble D de n disques de rayon 1 sur le plan et on veut isoler le plus grand ensemble I de disques (donc $I \subseteq D$) *qui ne se chevauchent pas.*

[3] Une définition duale appropriée peut aussi être formulée dans le cas d'un problème de maximisation.

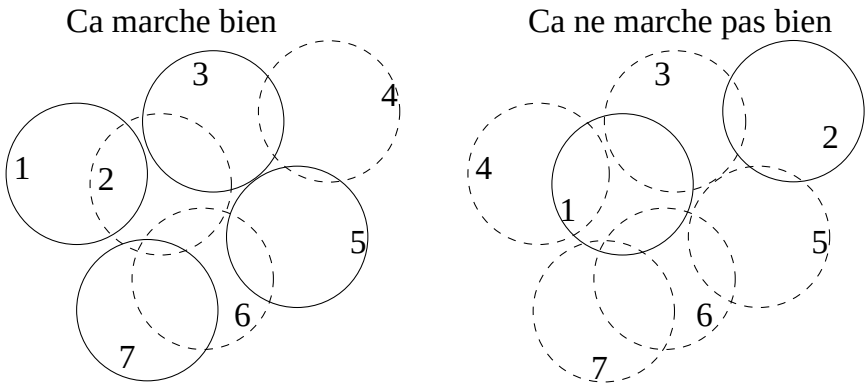


Quatre (4) disques qui ne se chevauchent pas

Il s'agit ici d'un cas particulier du problème d'identifier *le plus grand ensemble indépendant* d'un ensemble de sommets dans un graphe, c'est-à-dire, isoler le plus grand ensemble de sommets (disques) d'un graphe de sorte que pour toute paire de sommets il n'existe pas d'arête les reliant (ne se chevauchent pas), problème qui est NP-difficile.

Une première solution. Un premier algorithme glouton, A_1, est le suivant :

```
PROCEDURE plusGrandEnsembleDisques( D: set{Disque} ): set{Disque}
DEBUT
  I ← {}
  POUR chaque disque d ∈ D FAIRE
    SI d ne chevauche aucun disque de I ALORS
      I ← I ∪ {d}
    FIN
  FIN
  RETOURNER( I )
FIN
```



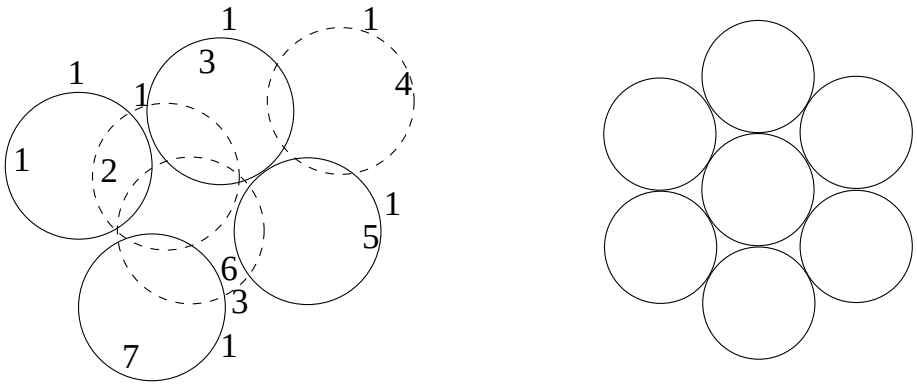
Théorème. L'algorithme A_1 est une 5-approximation.

Preuve. Soit I^* une solution optimale quelconque. Pour chaque disque $d \in I^*$, on définit $ch(d)$ comme suit :

- $ch(d) = 1$ si $d \in I$

- $ch(d) =$ nombre de disques qui étaient dans I et chevauchaient d au moment où il a été laissé de côté par l'algorithme A_1, si $d \notin I$.

On peut montrer (graphiquement) que n'importe quel disque d vérifie toujours $ch(d) \leq 5$:



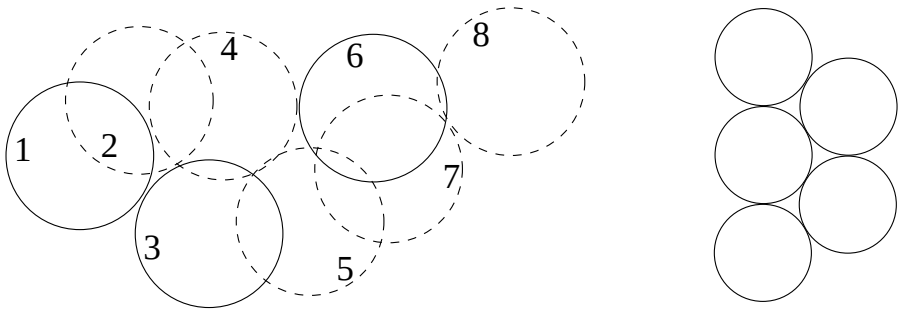
Donc, chaque disque de I^* a une charge $ch(d)$ d'au plus 5, ce qui veut dire qu'il chevauche au plus 5 disques de I, ce qui mène à $|I^*| \leq 5|I|$ — en d'autres mots, si un disque est chevauché par k disques qui sont disjoints deux à deux, alors $k \leq 5$.

Deuxième solution. Pour un disque d, on note x_d l'abscisse (coordonnée x) de son centre. Soit alors A_2 l'algorithme glouton suivant, dérivé de A_1 :

```
PROCEDURE plusGrandEnsembleDisques( D: set{Disque} ): set{Disque}
DEBUT
  I ← {}
  POUR chaque disque d ∈ D pris en ordre croissant de x_d FAIRE
    SI d ne chevauche aucun disque de I ALORS
      I ← I ∪ {d}
    FIN
  FIN
  RETOURNER( I )
FIN
```

Théorème. L'algorithme A_2 est une 3-approximation.

Preuve. La preuve est similaire à celle de l'algorithme A_1, la seule différence résidant dans le fait que le plus grand nombre de disques chevauchant un disque d et dont le centre est à droite de x_d est au plus 4.



2 Les algorithmes heuristiques

Tel qu'indiqué précédemment, un algorithme heuristique, ou plus simplement une *heuristique*, produit aussi une solution qui approxime la solution optimale. Toutefois, contrairement à un algorithme d'approximation, il n'est pas nécessairement possible de borner de façon exacte la qualité de la solution produite relativement à la solution optimale. Ainsi, il peut exister certains cas où l'heuristique produira une solution très mauvaise, ou encore sera tout à fait incapable de produire une solution.

Le terme algorithme "*heuristique*" est plus général que les autres termes introduits. Un algorithme d'approximation est une forme d'heuristique, puisque l'algorithme produit une solution pas trop mauvaise ; par contre, une heuristique ne produit pas nécessairement une approximation, car on ne peut pas toujours borner la qualité de la solution. De même, un algorithme probabiliste de type Monte Carlo qui produit un résultat correct avec une forte probabilité est une forme d'heuristique. Par contre, certaines heuristiques sont déterministes.

2.1 Exemple : deux heuristiques pour le voyageur de commerce

Voir exercices.

2.2 Exemple : une heuristique gloutonne pour le problème du coloriage d'un graphe

À compléter.

3 Les algorithmes probabilistes

Une forme d'heuristique souvent utilisée pour résoudre divers problèmes complexes repose sur l'utilisation d'*algorithmes probabilistes*.

Notons tout d'abord que, de façon générale, les algorithmes (heuristiques ou non) se divisent en deux grandes catégories :

1. Les algorithmes déterministes : Un algorithme est dit *déterministe* lorsque, appliqué sur une instance donnée du problème, il se comporte toujours de la même façon, c'est-à-dire, produit toujours le même résultat avec le même temps d'exécution.
2. Les algorithmes non déterministes : Un algorithme est dit *non déterministe* lorsque, appliqué sur une instance donnée du problème, il produit un résultat qui peut varier d'une exécution à l'autre ou dont le temps d'exécution peut varier. De tels algorithmes reposent évidemment sur l'utilisation de nombres (pseudo-)aléatoires, c'est pourquoi on parle alors d'*algorithmes probabilistes*, d'*algorithmes aléatoires* ou encore d'*algorithmes stochastiques* (*probabilistic algorithms* ou *randomized algorithms*).

La façon habituelle d'introduire du non déterminisme dans un algorithme est évidemment de recourir à l'utilisation de nombres "(*pseudo-*)aléatoires" pour effectuer, à des points clés de l'algorithme, certains choix.[4]

Selon Brassard et Bratley [BB96], on retrouve trois (3) types d'algorithmes probabilistes :

1. Les algorithmes numériques qui produisent comme résultat un "*intervalle de confiance*" du style "avec une probabilité de P %, la réponse est $N \pm e$".

[4] Il est important de rappeler que, dans le cas de nombres pseudo-aléatoires, si au début de l'exécution du programme le *germe* du générateur de nombres aléatoires est toujours initialisé avec la même valeur, alors on obtient un algorithme déterministe. Par contre, une version véritablement non déterministe peut souvent être obtenue en ne spécifiant pas le germe de départ, auquel cas le germe est choisi en utilisant une valeur variant d'une exécution à l'autre, par exemple, le temps de l'horloge.

2. Les algorithmes de *Monte Carlo* qui donnent la réponse exacte avec une grande probabilité, mais qui peuvent parfois donner une réponse incorrecte.

3. Les algorithmes de *Las Vegas* qui, lorsqu'ils donnent une réponse, produisent toujours une réponse exacte, mais qui peuvent aussi, au bout d'un certain temps, indiquer qu'aucune réponse n'a pu être trouvée.

L'exemple suivant (fictif, non algorithmique) illustre les trois types de réponses à la question "Quand Christophe Colomb a-t-il découvert l'Amérique?", pour une série de cinq (5) exécutions d'un algorithme de chacun des types :

1. Algorithme numérique : 1490–1500, 1485–1495, 1491–1500, 1480–1490, 1483–1495.
2. Algorithme Monte Carlo : 1492, 1491, 1492, 1492, 1492.
3. Algorithme Las Vegas : 1492, Pas de solution, 1492, 1492, 1492.

Un algorithme de Monte Carlo va généralement produire rapidement une réponse, mais cette réponse pourrait ne pas être correcte. Inversement, un algorithme de Las Vegas, lorsqu'il produira une réponse, va toujours produire une réponse correcte, mais il pourrait aussi ne pas produire cette réponse de façon rapide ... ou même jamais.

En simplifiant (i.e., en ignorant la possibilité qu'aucune solution ne puisse être trouvée), on peut dire que la différence entre algorithmes de Monte Carlo et algorithmes de Las Vegas est la suivante :

- Les algorithmes de Monte Carlo sont toujours rapides et *probablement* corrects.
- Les algorithmes de Las Vegas sont toujours corrects et *probablement* rapides.

Un exemple d'algorithme de type Las Vegas est présenté plus bas, à savoir, un algorithme de tri rapide où le choix du pivot se fait de façon aléatoire : peu importe le choix du pivot, le résultat final sera toujours correct (suite triée).

Le chapitre sur le voyageur de commerce présente un algorithme (plus précisément, une métaheuristique) de type Monte Carlo, algorithme dont le fonctionnement repose sur la simulation d'un processus physique, appelé *recuit* (*annealing*), utilisé pour produire des métaux. Un autre exemple est présenté plus loin.

3.1 Exemple : le tri rapide avec choix aléatoire du pivot

L'algorithme 1 présente une version du tri rapide (*quicksort*) où le choix de l'élément pivot se fait de façon aléatoire. Le pire cas de cet algorithme correspond à un choix aléatoire de pivot tel que, par exemple, la valeur choisie aléatoirement serait *toujours* le plus petit ou le plus grand élément du tableau, conduisant ainsi à une récursion linéaire, auquel cas l'algorithme serait de complexité $\Theta(n^2)$.[5] Toutefois, cette situation, bien qu'elle ne soit pas impossible, est *hautement improbable*.

Le fait de choisir aléatoirement le pivot, au lieu de simplement choisir le premier élément, fait en sorte que l'élément pivot une chance équivalente d'être n'importe quel élément du tableau. En moyenne, on peut donc s'attendre à ce que le nombre d'éléments à trier dans chacune des sous-séquences sera relativement équilibré, ce qui fait que l'algorithme sera, en moyenne (temps espéré), de complexité $\Theta(n \lg n)$.

3.2 Exemple : un algorithme de type Monte Carlo

À compléter.

[5] Soulignons que le pire cas ne correspond pas à une entrée particulière, comme c'est habituellement le cas pour les algorithmes que nous avons vus jusqu'à présent. Le pire cas correspond plutôt à un comportement particulier du générateur de nombres aléatoires.

```
procedure partitionner( ref Item A[*], int inf, int sup, res int posPivot )
{
  posPivot = int(random(inf, sup+1)); # Choix aleatoire de la position du pivot.
  Item pivot = A[posPivot];
  A[inf] :=: A[posPivot];
  posPivot = inf;
  for [i = inf+1 to sup] {
    if (A[i] <= pivot) {
      posPivot += 1;
      A[i] :=: A[posPivot];
    }
  }
  A[posPivot] :=: A[inf];
}

procedure trierRec( ref Item A[*], int inf, int sup )
{
  if (inf >= sup) {
  } else {
    int posPivot;
    partitionner( A, inf, sup, posPivot );
    trierRec( A, inf, posPivot-1 );
    trierRec( A, posPivot+1, sup );
  }
}

procedure trier( ref Item A[*], int n )
{ trierRec( A, 1, n ); }
```

Algorithme 1: Tri *quicksort* avec choix aléatoire du pivot

3.3 Analyse d'un algorithme probabiliste par opposition à analyse probabiliste d'un algorithme déterministe

Un point important à souligner est qu'un algorithme probabiliste et une analyse probabiliste d'un algorithme déterministe sont deux choses tout à fait différentes. Dans le cas d'un algorithme probabiliste, le comportement de l'algorithme, et donc son analyse, dépend *des choix aléatoires effectués par l'algorithme*, et les résultats analytiques ainsi obtenus sont valables pour n'importe quelles entrées fournies à l'algorithme. L'analyse probabiliste d'un algorithme déterministe (par ex., pour déterminer le temps *moyen* d'exécution), quant à elle, dépend des *diverses hypothèses faites sur la distribution des entrées*, et n'est valable que pour l'ensemble des données possibles, *et non pour une entrée spécifique*. Une analyse probabiliste s'effectue donc sur un algorithme déterministe, en supposant une certaine distribution des entrées. Par contre, un algorithme probabiliste n'est pas déterministe, et son analyse ne repose pas sur une hypothèse de distribution pour les entrées mais uniquement sur des hypothèses sur les choix aléatoires effectués par l'algorithme.

4 Les métaheuristiques

Certains auteurs définissent une heuristique comme une technique qui repose sur l'expérimentation plutôt que sur une analyse théorique. Pour d'autres auteurs, on utilise une heuristique simplement lorsqu'on accepte de ne pas forcément trouver la solution optimale à un problème, et ce pour conserver un temps de calcul *raisonnable*.

Diverses heuristiques ont été développées pour des problèmes spécifiques. Par contre, certaines heuristiques ont été développées de façon à pouvoir s'appliquer de façon plus générale, donc *de façon indépendante du problème* — évidemment en spécialisant certaines parties clés en fonction du problème à résoudre. Dans ce cas, on parle alors plutôt de *métaheuristiques*.

Une caractéristique commune à de nombreuses métaheuristiques est qu'elles ont souvent été développées en s'inspirant de processus qu'on retrouve dans la nature, par exemple, en physique ou en biologie. Parmi les plus connues, on retrouve les métaheuristiques présentées dans les paragraphes qui suivent.

4.0 L'exploration locale du voisinage (*local search*)

La stratégie heuristique de base consiste à explorer le voisinage d'une solution dans le but de *l'améliorer*, d'où les termes utilisés par certains auteurs d'algorithmes *d'amélioration itérative* ou encore de *méthode de descente*.[6]

L'idée de base de cette heuristique consiste à identifier une configuration initiale acceptable, possiblement choisie au hasard, puis à appliquer diverses *modifications élémentaires* à cette configuration pour tenter d'arriver à une nouvelle configuration qui améliore (diminue dans le cas d'un problème de minimisation) la valeur de la fonction à optimiser. Si une telle configuration peut être trouvée, alors on répète le processus d'amélioration itérative sur cette nouvelle configuration. Dans le cas contraire, on termine l'algorithme, puisqu'on a trouvé un *minimum local*.

Dénotons par $\mathcal{T}$ l'ensemble des transformations (les modifications élémentaires) qui peuvent être appliquées à une configuration `s` pour obtenir une autre configuration acceptable. Soit alors `T` une de ces transformations : notons par `T(s)` la configuration résultant de l'application de la transformation `T` à `s`. Les grandes lignes d'un algorithme générique pour la méthode heuristique de recherche locale sont présentées à l'algorithme 2.

[6] Notons qu'en anglais, dans le contexte particulier consistant à maximiser une fonction, on utiliserait plutôt le terme *hill-climbing algorithm*, alors que pour un problème de minimisation, on utiliserait le terme *gradient-descent*.

```
DEBUT
  s ← élément arbitraire (aléatoire) de l'espace des solutions
  REPETER
    modifié ← FAUX
    POUR chaque transformation T ∈ 𝒯 FAIRE
      s' ← T(s)
      SI valeur(s') < valeur(s) ALORS
        s ← s'
        modifié ← VRAI
      FIN
    FIN
  JUSQUE NON modifié
  RETOURNER s
FIN
```

Algorithme 2: Algorithme générique d'exploration locale

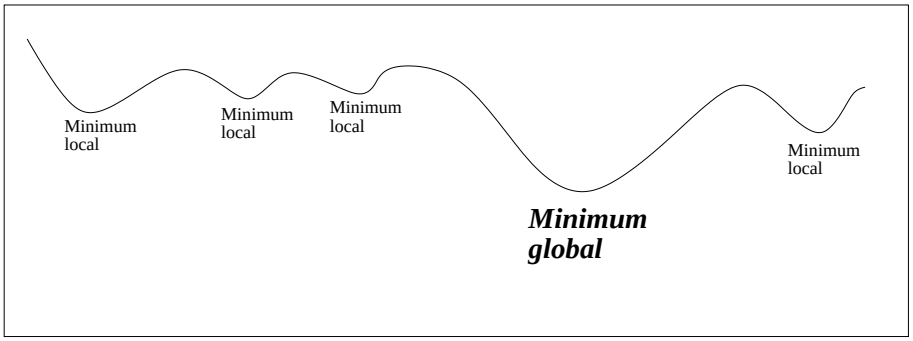


Figure 1: Minimums locaux vs. minimum global

Le principal problème d'une heuristique de recherche locale est qu'il est possible qu'une fonction possède plusieurs minimaux locaux, distincts du minimum global, tel qu'illustré à la figure 1. Dans ce cas, il est donc possible que l'algorithme d'exploration locale *reste pris dans un tel minimum local*.

Une des façons d'éviter ce problème peut alors être de répéter le processus de recherche locale, et ce avec des valeurs de départ différentes, en ne conservant que la meilleure solution trouvée, tel qu'illustré à l'algorithme 3.

Cette dernière heuristique illustre les deux mécanismes présents dans la plupart des heuristiques et métaheuristiques :

- Un mécanisme visant à l'*intensification* des caractéristiques des bonnes solutions — on explore le voisinage local pour tenter de trouver une configuration qui améliore la fonction à optimiser.
- Un mécanisme visant à la *diversification* des configurations explorées — pour éviter de rester pris dans un minimum local, on tente donc de visiter différentes parties de l'espace des solutions.

Le chapitre sur le voyageur de commerce présente une instance (une spécialisation) de la métaheuristique de recherche locale.

```
DEBUT
  REPETER
    s ← élément arbitraire (aléatoire) de l'espace des solutions
    modifié ← FAUX
    REPETER
      POUR chaque transformation T ∈ 𝒯 FAIRE
        s' ← T(s)
        SI valeur(s') < valeur(s) ALORS
          s ← s'
          modifié ← VRAI
        FIN
      FIN
    JUSQUE NON modifié
  JUSQUE valeur(s) est acceptable
  RETOURNER s
FIN
```

Algorithme 3: Algorithme générique d'exploration locale

4.1 La méthode de recherche avec tabous (*Tabu Search*)

Cette méthode est décrite comme suit par Dréo et al. [DPST03, p. 9]:

> La méthode de recherche avec tabous [est basée sur des] mécanismes inspirés de la *mémoire* humaine. [...] [Son] principe de base est simple : comme le recuit simulé, la méthode tabou fonctionne avec une seule "configuration courante" à la fois (au départ, une solution quelconque), qui est actualisée au cours des "itérations" successives. À chaque itération, le mécanisme de passage d'une configuration, soit s, à la suivante, soit t, comporte deux étapes :
>
> - on construit l'ensemble des *voisins* de s, c'est-à-dire l'ensemble des configurations accessibles en un seul *mouvement* élémentaire à partir de s [...] ; soit $V(s)$ l'ensemble [...] de ces voisins ;
> - on évalue la fonction objectif f du problème en chacune des configurations appartenant à $V(s)$. La configuration t, qui succède à s dans la suite des solutions construite par la méthode tabou, est la configuration de $V(s)$ en laquelle f prend la valeur minimale. Notons que cette configuration t est adoptée même si elle est moins bonne que s, i.e., si $f(t) > f(s)$: c'est grâce à cette particularité que la méthode tabou permet d'éviter le piégeage dans les minimums locaux de f.
>
> [...] Pour éviter [de retourner à une configuration déjà retenue lors d'une itération précédente], on tient à jour et on exploite, à chaque itération, une liste de mouvements interdits, la "liste des tabous" ou "liste tabou".

Le chapitre sur le voyageur de commerce présente une instance (une spécialisation) de la métaheuristique de recherche avec tabous.

4.2 Le recuit simulé

Le recuit simulé est décrit comme suit par Dréo et al. [DPST03, pp. 5-6]:

> [La technique du *recuit* en métallurgie] consiste à porter le matériau à haute température, puis à abaisser lentement celle-ci, [ce qui permet d']obtenir un état

> solide "bien ordonné", d'énergie minimale (en évitant les structures "métastables", caractéristiques des minimums locaux d'énergie).
>
> [...]
>
> La méthode du *recuit simulé* transpose le procédé du *recuit* à la résolution d'un problème d'optimisation : la fonction objectif du problème, analogue à l'énergie d'un matériau, est alors minimisée, moyennant l'introduction d'une *température* fictive, qui est, dans ce cas, un simple paramètre de contrôle de l'algorithme.

Cette méthode est examinée plus en détails dans le chapitre sur le problème du voyageur de commerce.

4.3 Les algorithmes évolutionnaires

Les algorithmes évolutionnaires sont décrits comme suit par Dréo et al. [DPST03, p. 11]:

> Les algorithmes évolutionnaires (AEs) sont des techniques de recherche inspirées par l'évolution biologique des espèces [...] Le principe d'un algorithme évolutionnaire se décrit simplement. Un ensemble de N points dans un espace de recherche, choisis a priori au hasard, constitue la *population* initiale ; chaque individu x de la population possède une certaine performance, qui mesure son degré *d'adaptation* à l'objectif visé : dans le cas de la minimisation d'une fonction objectif f, x est d'autant plus performant que $f(x)$ est plus petit. Un AE consiste à faire évoluer progressivement, par *générations* successives, la composition de la population, en maintenant sa taille constante. Au cours des générations, l'objectif est d'améliorer globalement la performance des individus [à l'aide des] deux principaux mécanismes qui régissent l'évolution des êtres vivants, selon la théorie de C. Darwin :
>
> - la *sélection*, qui favorise la reproduction et la survie des individus les plus performants,
> - et la *reproduction*, qui permet le brassage, la recombinaison et les variations des caractères héréditaires des parents, pour former des descendants aux potentialités nouvelles.

```
Initialiser la population (avec n individus)
REPETER
  Évaluer les n individus
  Sélectionner le k individus qui vont se reproduire
  Effectuer le croisement entre les k individus sélectionnés
  Effectuer des mutations sur les k enfants résultants
  Évaluer les k enfants (possiblement mutants)
  Sélectionner les parents qui seront remplacés par leurs enfants
  Effectuer les remplacements (pour obtenir une population de n individus)
JUSQUE le critère de terminaison est atteint
RETOURNER l'individu le plus performant
```

Algorithme 4: Algorithme évolutionnaire générique

L'algorithme 4 présente la forme générale d'un algorithme évolutionnaire générique. De nombreuses variantes d'algorithmes évolutionnaires peuvent être obtenues selon la façon dont chacune des sous-tâches est définie :

- Le mécanisme utilisé pour sélectionner les individus qui vont se reproduire, par exemple, probabilité proportionnelle à la "*force*" de l'individu.

- La façon dont le croisement entre parents s'effectue, par exemple, par coupure et échange de composantes.
- Les opérateurs de mutation des nouveaux individus, par exemple, modifications aléatoires de bits, flips de bits, etc.
- Le mécanisme utilisé pour sélectionner les individus qui seront remplacés par leurs enfants, par exemple, les parents sont toujours remplacés, le m meilleurs enfants remplacent leurs parents, etc.

Soulignons que, parmi la classe générale des algorithmes évolutionnaires, on retrouve diverses sous-catégories :

- Les *algorithmes génétiques* : ces algorithmes reposent sur un encodage binaire des individus, c'est-à-dire, une simple chaîne de bits sert à représenter chacun des individus. Ces bits représentent le *génotype* de l'individu. À partir du génotype d'un individu, on peut alors calculer son *phénotype*, spécifique au problème traité (par exemple, la valeur qui lui est associée pour le problème considéré). La sélection des individus se fait traditionnellement de façon proportionnelle à leur force et tous les parents sont généralement remplacés par leur enfants. Les opérateurs de croisement et de mutation sont alors relativement simples — coupure et échange de segments de bits, mutation de bits, etc. — et s'expriment strictement sur le génotype (sur les chaînes de bits), donc indépendamment du phénotype (valeur de la fonction à optimiser).
- La *programmation génétique* : dans cette approche, on ne cherche pas directement à trouver une solution au problème de départ. On cherche plutôt à générer, par évolution, un programme qui permettra de solutionner le problème. Un individu de la population est donc un programme qui, lorsqu'exécuté, visera à produire la solution. Les croisements et mutations se font donc en termes des programmes eux-mêmes (sur les arbres syntaxiques), et l'objectif est d'arriver à générer un programme dont le résultat sera le meilleur possible pour le problème considéré.

4.4 Les algorithmes de colonie de fourmis

Les algorithmes de colonies de fourmis sont décrits comme suit par Dréo et al. [DPST03, p. 12]:

> Cette approche [...] s'efforce de simuler la capacité collective de résolution de certains problèmes, observée chez une colonie de fourmis, dont les membres sont pourtant individuellement dotés de facultés très limitées. [...] En particulier, les entomologistes ont analysé la collaboration qui s'établit entre les fourmis pour aller chercher de la nourriture à l'extérieur de la fourmilière, il est remarquable que les fourmis suivent toujours le même chemin, et que ce chemin soit le plus court possible. Cette conduite est le résultat d'un mode de communication indirecte, via l'environnement : la "stigmergie".[7] Chaque fourmi dépose, le long de son chemin, une substance chimique, dénommée "phéromone" ; tous les membres de la colonie perçoivent cette substance et orientent préférentiellement leur marche vers les régions plus "odorantes".

L'algorithme 5 présente les grandes lignes d'un algorithme de colonies de fourmis pour le problème du voyageur de commerce. On examinera cet algorithme plus en détails en classe (si le temps le permet?).

[7] Le terme "*stigmergy* aurait été introduit initialement par un entomologue pour décrire "*a form of indirect communication mediated by modifications of the environment that [was] observed in the workers caste of two species of termites*" [DS04, p. 21].

```
POUR t = 1, ..., t_max FAIRE
  POUR chaque fourmi f_k (k = 1, ..., m) FAIRE
    v_1 ← ville de départ  (choisie au hasard)
    T^k(t) ← [v_1]
    POUR chaque ville non visitée par la fourmi f_k FAIRE
      Choisir une ville v dans la liste J^k des villes non visitées
          (ce choix s'effectue en fonction de l'intensité de la piste
          et de la visibilité de la ville)
      Ajouter la ville sélectionnée au trajet T^k(t)
    FIN
    Déposer une piste de phéromones sur le trajet T^k(t)
       (son intensité dépend de la qualité de la solution trouvée)
  FIN
  Faire évaporer les pistes de phéromones
     (pour oublier, petit à petit, les mauvaises solutions)
FIN
```

Algorithme 5: Les grandes lignes d'un algorithme de colonie de fourmis pour résoudre le problème du voyageur de commerce (adaptée de [DPST03, p. 123])

Intensification et diversification Deux notions importantes pour caractériser les métaheuristiques sont celles d'*intensification* et de *diversification* :

- Intensification = exploitation de l'information rassemblée par le système à un instant donné.
- Diversification = exploration de régions de l'espace de recherche imparfaitement prises en compte.

Dans le cas des algorithmes de colonies de fourmis, ces deux aspects peuvent être modulés relativement facilement en jouant avec les paramètres α et β mentionnés précédemment :

- Plus la valeur α est élevée, plus l'intensification sera importante, car plus les pistes auront une influence.

  À l'inverse, plus α est faible, plus la diversification sera importante, car les fourmis éviteront certaines pistes.
- Idem pour β.

Parallélisme et optimisation continue ou dynamique Une caractéristique particulièrement intéressante des algorithmes de colonies de fourmis est leur *parallélisme intrinsèque* : "les solutions de bonne qualité émergent de résultat des *interactions* indirectes ayant cours dans le système, pas d'un codage explicite d'échanges [...] [puisque] chaque fourmi ne prend en compte que des informations locales sur son environnement" [DPST03].

Une autre caractéristique intéressante est que ces algorithmes: peuvent s'appliquer non seulement à des problèmes d'optimisation combinatoire, mais aussi à des problèmes *continus* et/ou *dynamiques* :

- Optimisation continue : lorsque la fonction à optimiser est *continue* (plutôt que discrète).
- Problème dynamique : lorsque le problème varie dans le temps, donc lorsque la solution optimale varie en fonction du temps.

4.5 Quelques exemples d'applications des métaheuristiques

Depuis leur création, les métaheuristiques ont été utilisées pour des applications très diverses. Quelques-unes des applications mentionnées dans [DPST03] sont les suivantes :

- Recuit simulé : organisation du réseau informatique du Loto en France (relier une dizaine de milliers de machines de jeu à des ordinateurs centraux) ;
- Recuit simulé : optimisation de la collecte des ordures ménagères à Grenoble ;
- Recuit simulé : déterminer l'implantation optimale des jours de repos dans un planning hospitalier ;
- Recuit simulé : optimisation en architecture où il fallait, dans un immeuble de 17 étages, répartir les activités entre les différentes pièces de près de 2000 employés ;
- Algorithmes génétiques : optimisation de réseaux mobiles chez France Telecom ;
- Algorithmes génétiques : gestion du trafic aérien ;
- Colonies de fourmis : optimisation de tournées de véhicules.

4.6 Caractéristiques communes aux diverses métaheuristiques

En résumé, les méthodes métaheuristiques ont en commun les caractéristiques suivantes (selon [DPST03, p. 2]) :

- elles sont, au moins pour partie, *stochastiques* : cette approche permet de faire face à l'*explosion combinatoire* des possibilités ;

  [...]

- elles sont inspirées par des *analogies* : avec la physique (recuit simulé, diffusion simulée...), avec la biologie (algorithmes évolutionnaires, recherche avec tabous...) ou avec l'éthologie (colonies de fourmis, essaims particulaires...) :
- elles partagent aussi les mêmes inconvénients : les difficultés de *réglage* des paramètres de la méthode et le *temps de calcul* élevé.

C'est la première caractéristique, l'aspect stochastique (aléatoire) de ces algorithmes, qui permet généralement aux métaheuristiques de s'extraire des minimums locaux, c'est-à-dire, de ne pas rester pris dans un minimum local et d'aller explorer diverses régions de l'espace des solutions.

Quant à la dernière caractéristique, elle est particulièrement importante et fait que ces méthodes portent bien leur nom d'*heuristiques* : il est généralement très difficile d'effectuer une analyse théorique de leurs performances, tant quantitative que qualitative (quelle sera la qualité de la solution trouvée pour un certain temps d'exécution?). Comme l'indiquent Dréo et al. [DPST03, p. 317] :

> [Le] réglage "optimal" des divers paramètres d'une métaheuristique, qui peut être préconisé par la théorie, est souvent inapplicable en pratique, car il induit un coût de calcul prohibitif. En conséquence, le choix d'une "bonne" méthode, et le réglage des paramètres de celles-ci, font généralement appel au savoir-faire et à l'"expérience" de l'utilisateur, plutôt qu'à l'application fidèle de règles bien établies.

On verra que c'est bien le cas dans l'application de la méthode du recuit simulé au problème du voyageur de commerce.

Références

[BB96] G. Brassard and P. Bratley. *Fundamentals of Algorithmics.* Prentice-Hall, 1996. [QA76.6B73].

[DPST03] J. Dréo, A. Pétrowski, P. Siarry, and É. Taillard. *Métaheuristiques pour l'optimisation difficile.* Eyrolles, 2003. [T57.84M48].

[DS04] M. Dorigo and T. Stutzle. *Ant Colony Optimization.* The MIT Press, 2004.

[JS04] R. Johnsonbaugh and M. Schaefer. *Algorithms.* Pearson Education, 2004.

Cette partie des notes de cours est basée principalement sur les références suivantes :

- [BB96, Chapitres 10 et 13]
- [JS04, Chapitre 11]
- [DPST03, Avant-propos et chapitres 3 et 4]
- [DS04, Chapitres 1 et 3]
- Transparents produits par Cédric Chauve, à l'automne 2004, dans le cadre du cours INF7440 :

  `http://www.lacim.uqam.ca/~chauve/Enseignement/INF7440/planning.html`

- *Free On-line Dictionary of Computing* :

  `http://wombat.doc.ic.ac.uk/foldoc`

- *Dictionary of Algorithms and Data Structures* du NIST (*National Institute of Standards and Technology*) :

  `http://www.nist.gov/dads`