Chapitre 8

Méthodes incomplètes

Toutes les méthodes complètes présentées précédemment ont l'inconvénient majeur d'être extrêmement consommatrices en temps. En effet :

- Les méthodes complètes/exactes explorent de façon systématique l'espace de recherche. Cela ne marche donc pas dans de nombreux problèmes à cause de l'explosion combinatoire du nb d'états possibles.
- Ces méthodes sont généralement à base de recherche arborescente. On est donc contraint par l'ordre des noeuds dans l'arbre. Cela implique une impossibilité de compromis qualité/temps ($\neq$ algo "anytime" [1]).

Or il y a souvent des cas où on peut se contenter d'une solution approchée, pourvu qu'elle soit suffisamment "bonne".
Pour trouver cette solution approchée, on a deux types de stratégie :

- garder une méthode complète et stopper après un temps déterminé (mais suivant les problèmes, cela n'est pas toujours possible)
- plonger dans l'arbre de recherche avec des heuristiques sans jamais revenir en arrière (mais : comment être sûr qu'on va trouver vite une bonne solution ?)

La seconde possibilité correspond à la notion de méthodes "incomplètes", dites aussi "méthodes approchées" (ou méthodes "heuristiques" [2]). Ces méthodes ne garantissent pas d'obtenir la solution optimale mais sont en général suffisamment "efficaces" pour être exploitables : le rapport qualité de la solution trouvée/temps mis pour la trouver est raisonnablement bon.

Il existe beaucoup de familles de méthodes différentes. Nous allons aborder ici uniquement la famille des **méthodes locales** dont le principe est le suivant :

1. Travailler avec un nouvel espace : celui des solutions (il est différent de l'espace des états mais ils peuvent avoir des sommets en commun).
2. Partir d'une solution sinon approchée, du moins potentiellement bonne et essayer de l'améliorer itérativement. Pour améliorer une solution on ne fait que de légers changements (on parle de changement local, ou de solution voisine).
3. Relancer la méthode plusieurs fois en changeant le point de départ pour avoir plus de couverture.
4. Tout problème est considéré comme un problème d'optimisation (même les problèmes de satisfaction : le coût à optimiser est alors le nombre de contraintes insatisfaites).

1. Un algorithme anytime est un algorithme capable de donner une solution valide à un problème même s'il est interrompu avant d'avoir terminé. L'algorithme trouve de meilleures solutions au fur et à mesure de son exécution.

2. Attention à ne pas mélanger avec la notion d'heuristique présentée pour l'algorithme A*. Ici le mot "heuristique" s'entend comme "exploitant une stratégie pour aller plus vite" et pas comme une métrique associée à chaque état et estimant le coût restant avant d'atteindre la solution. Ceci étant, le fait que le même mot soit utilisé montre bien que cette métrique sert aussi de stratégie pour aller plus vite.

5. Ici la notion de complétude est liée à la découverte de la meilleure solution (et plus à la découverte d'une solution ! Cela correspond donc à la notion d'admissibilité des méthodes complètes).

Précisons le vocabulaire utilisé ici :

Définition 14

Solution : *c'est un état-but du problème tel que défini pour les méthodes complètes (dans le sens où il permet d'affecter une valeur à toutes les variables du pb).*

Fonction d'évaluation `eval` **:** *c'est une fonction qui évalue un état solution.*

Solution optimale : *c'est une solution de coût minimal (resp. maximal) selon* `eval`.

Mouvement : *c'est une opération élémentaire permettant de passer d'une solution à une solution voisine.*

Voisinage d'une solution : *c'est l'ensemble des solutions voisines, c'est-à-dire l'ensemble des solutions accessibles par un mouvement (et un seul).*

On a donc l'espace des solutions qui pourrait être représenté sous forme graphique avec les solutions servant de sommets et les mouvements permettant de définir les arcs entre deux solutions voisines.

Un algorithme de recherche locale typique pour le cas d'une minimisation est l'algorithme 3.

Algorithme 3 : Algorithme de recherche locale (pour minimisation) "sans reprise"

Données :
max_mouvements : le nb max de mouvements à faire,
`nouvelle_solution` : la fonction générant une solution (aléatoirement ou non),
`eval` : la fonction d'évaluation,
`choisir_voisin` : la fonction de choix du voisin à exploiter

Variables :
E : solution courante,
E* : la meilleure solution atteinte,
E' : le voisin choisi,
m : le mouvement courant

début
- E ← `nouvelle_solution`()
- E* ← E
- **pour** *m=1 à max_mouvements* **faire**
  - E' ← `choisir_voisin`(E)
  - **si** `eval`*(E')* < `eval`*(E)* **alors**
    - // on améliore E
    - E ← E'
- **si** `eval`*(E)*<`eval`*(E*)* **alors**
  - // on améliore E*
  - E* ← E
- **retourner** E*,`eval`(E*)

On peut ensuite améliorer cet algorithme en le répétant plusieurs fois avec une solution initiale différente dès qu'on ne peut plus améliorer (ce sera la version "avec reprise" illustrée dans l'algorithme 4). Cette nouvelle version est elle-aussi incomplète mais, à défaut, elle permettra peut-être de trouver un meilleur optimum local.

Algorithme 4 : Algorithme recherche locale (pour minimisation) "avec reprise" si pas d'amélioration

Données :
max_reprises : le nb max de reprises à faire,
`nouvelle_solution` : la fonction générant une solution (aléatoirement ou non),
`eval` : la fonction d'évaluation,
`choisir_voisin` : la fonction de choix du voisin à exploiter

Variables :
E : solution courante,
E* : la meilleure solution atteinte,
E' : le voisin choisi,
r : la reprise courante

début
E ← `nouvelle_solution`()
E* ← E
pour *r=1 à max_reprises* **faire**
E' ← `choisir_voisin`(E)
tant que `eval`*(E')* < `eval`*(E)* **faire**
// on améliore E
E ← E'
E' ← `choisir_voisin`(E)
si `eval`*(E)*<`eval`*(E*)* **alors**
// on améliore E*
E* ← E
E ← `nouvelle_solution`()
retourner E*,`eval`(E*)

De manière générale, ce type d'algo se comporte de la manière suivante :

1. une majorité de mouvements améliore la solution courante,
2. puis le nombre d'améliorations devient de plus en plus faible,
3. il n'y a plus d'améliorations : on est dans un optimum, qui peut être local,
4. à chaque reprise, on espère explorer une autre partie de l'espace des solutions mais on n'améliore pas forcément l'optimum local déjà trouvé.

Avec ce type de méthode, les questions à se poser sont les suivantes :

— quand faut-il s'arrêter ? Plus précisément :
  — pour la reprise en cours : nb de mouvements max ? comment détecter un optimum local ?
  — pour la recherche elle-même : le nb max de reprises ? coût de la solution convenable ? limite de temps atteinte ?

— faut-il être opportuniste ou gourmand [3] ? Plus précisément :
  — si aucune recherche d'amélioration : comment trouver les bonnes solutions ?
  — si recherche d'amélioration, comment éviter les optimums locaux ?
— comment ajuster les paramètres nombre de reprises/nombre de mouvements ?
— comment comparer les performances de deux méthodes différentes ? (qualité de la solution vs. temps consommé)

Dans la suite, nous allons successivement illustrer ces idées d'algorithme sur différentes méthodes : le *Hill-climbing* (ou escalade), le *Steepest Hill-climbing* (ou escalade par la plus grande pente), le *Tabou*.

8.1 Les stratégies Hill-Climbing et Steepest Hill-Climbing

Le Hill-Climbing est aussi appelé recherche par gradient [4]. Ici on a :
— `choisir_voisin` : choix aléatoire dans le voisinage courant,
— mise à jour de E : seulement s'il y a une amélioration.
C'est donc une méthode opportuniste (ne cherche pas à trouver le meilleur voisin mais n'utilise un voisin que s'il est meilleur que la solution courante).
Il s'agit donc d'une stratégie profondeur d'abord combinée avec le meilleur des fils selon une fonction heuristique. Elle est principalement utilisée pour la recherche d'une solution optimale.
Cette stratégie ne maintient pas d'arbre de recherche (on ne revient jamais en arrière) et ne regarde pas très loin. Le risque est d'être bloqué sur un optimum local, même en utilisant la version "avec reprise" (algorithme 4 page précédente).

Le Steepest Hill-Climbing est une version plus perfectionnée du Hill-Climbing qui consiste à prendre "la plus grande pente". Ici on a :
— `choisir_voisin` : après avoir déterminé l'ensemble des meilleures solutions voisines de la solution courante (exceptée celle-ci), on en choisit une aléatoirement,
— mise à jour de E : si amélioration (donc arrêt sur optimum local).
C'est donc un algorithme glouton (*greedy* : choix de l'optimum local à chaque étape – choix du voisin et mise à jour de E).
Comme pour le Hill-Climbing, cette stratégie ne maintient pas d'arbre de recherche (on ne revient jamais en arrière) et ne regarde pas très loin. Elle regarde juste "un peu mieux autour". Le risque est, là-aussi, d'être bloqué sur un optimum local, même en utilisant la version "avec reprise" (algorithme 4 page précédente).

La différence de comportement entre les deux algos est illustrée sur la figure 8.1 page ci-contre.

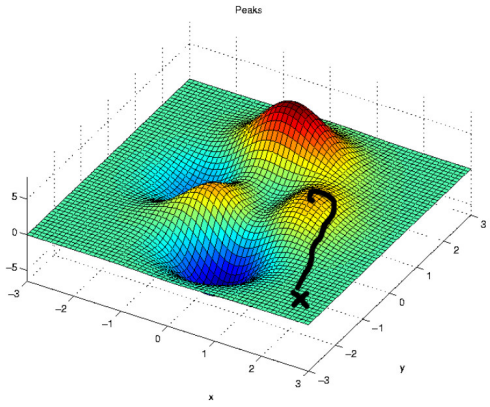
8.2 Stratégie "Tabou"

C'est aussi une méthode locale basée sur une recherche par gradient avec les spécificités suivantes :
— choix dans le voisinage,
— possibilité de détérioration de la solution courante,

3. Un algorithme glouton (*greedy algorithm* en anglais, parfois appelé aussi algorithme gourmand) est un algorithme qui suit le principe de faire, étape par étape, un choix d'optimum local. Dans certains cas, cette approche permet d'arriver à un optimum global, mais dans le cas général c'est une simple heuristique. Cette notion s'oppose à celle d'un algorithme opportuniste qui ne cherche pas à satisfaire un optimum local mais qui n'utilise un voisin que s'il est meilleur que la solution courante.

4. On utilise aussi les termes "descente en gradient" ou "montée en gradient", suivant le sens de l'optimisation (min ou max) recherché.

en montant le plus tôt possible



en suivant la pente la plus forte

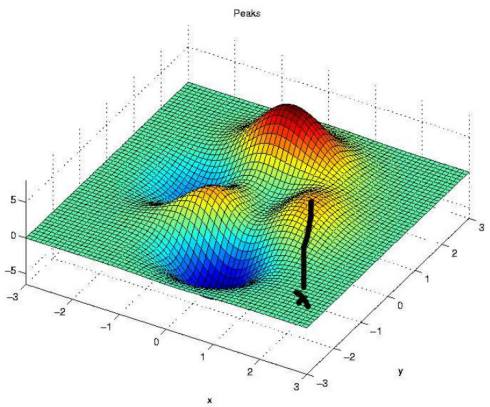


FIGURE 8.1 – Différence entre Hill Climbing et Steepest Hill-Climbing avec l'algorithme "sans reprise"

— pour améliorer la recherche, on mémorise les k dernières solutions courantes et on interdit le retour sur une de ces solutions,
— on autorisera en plus toujours un mouvement conduisant à une meilleure solution que la meilleure obtenue auparavant.

En pratique, au lieu de mémoriser les k dernières solutions courantes (parfois trop coûteux en temps et mémoire), on mémorise les k derniers mouvements (plus restrictif mais peu coûteux en temps).
L'algorithme dans le cas d'une minimisation est l'algorithme 5 page suivante.

8.3 Conclusion sur les méthodes locales

Les méthodes incomplètes correspondent à un ensemble de méthodes empiriques variées, parfois trop, car l'absence de modélisation les rend difficiles à adapter d'un problème à un autre (cela fait un peu bricolage). Par contre, elles sont à peu près indifférentes à la taille du problème, et donnent de bons résultats (approchés) à condition de bien les paramétrer. Elles peuvent donc fournir un bon complément aux méthodes complètes.
Quelques pistes d'amélioration : Pour éviter d'être coincé dans un optimum local, on peut ajouter du bruit : une part de mouvements aléatoires pendant une recherche classique en suivant le principe :

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

On peut jouer aussi au niveau de l'acceptabilité d'un voisin pour la mise à jour de la solution courante : s'il y a une dégradation, l'accepter avec une probabilité p.
Reste le problème : comment régler p ? On a toujours des compromis à faire entre exploration globale/locale.

8.4 Exercices

Énoncé 27 *Définir les fonctions de création d'une solution, d'évaluation et de voisinage pour le problème du sac à dos.*

Énoncé 28 *Soit l'espace des solutions donné sur la figure 8.2.*

Algorithme 5 : Algorithme Tabou (pour minimisation)

Données :

condition_de_fin : cela peut être une limite en temps, en itération, ou une non amélioration de la solution,
`nouvelle_solution` : la fonction générant une solution (aléatoirement ou non),
`eval` : la fonction d'évaluation,
`voisinage` : l'ensemble des voisins d'une solution

Variables :

E : solution courante,
E* : la meilleure solution atteinte,
V : l'ensemble des voisins de la solution courante qui ne sont pas tabous,
E' : le voisin choisi,
T : la liste des solutions tabous

début

```
E ← nouvelle_solution()
E* ← E
T ← ∅
tant que non(condition_de_fin) faire
    V ← voisinage(E) − T
    E' ← min pour eval sur V
    si eval(E') ≥ eval(E) alors
        // on ne peut plus améliorer
        T ← T ∪ {E}
    si eval(E*) > eval(E') alors
        // on améliore E*
        E* ← E'
    // dans tous les cas, on repart du voisin
    E ← E'
retourner E*, eval(E*)
```

1. *Décrivez le fonctionnement du Steepest Hill-Climbing sans reprise sur cet espace avec les hypothèses suivantes :*
   — *on cherche la solution avec la valeur maximum,*
   — *la fonction* `nouvelle_solution` *renvoie* s_0.
2. *Décrivez le fonctionnement du Tabou sans reprise sur cet espace avec les hypothèses suivantes :*
   — *on cherche la solution avec la valeur maximum,*
   — *la fonction* `nouvelle_solution` *renvoie* s_0,
   — *la liste Tabou est illimitée,*
   — *la condition de fin correspond au remplissage de la liste Tabou avec au moins 50% des solutions de l'espace.*

Énoncé 29 *Appliquer le steepest hill climbing au problème du sac à dos avec :*
— *poids :* $\{4, 3, 8, 4, 4, 9\}$
— *valeurs :* $\{7, 2, 32, 9, 2, 8\}$
— *poids max :* 23

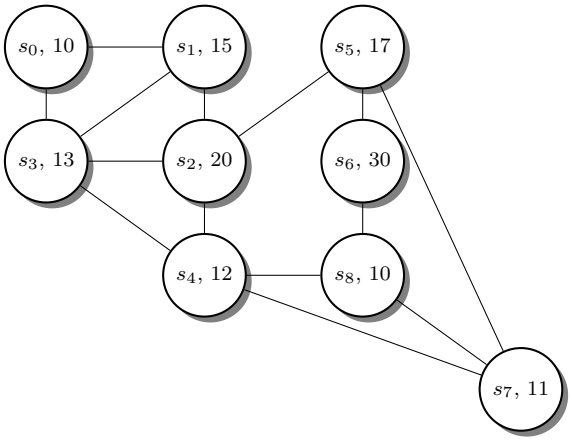


FIGURE 8.2 – Espace des solutions pour l'énoncé 28

Bilan :
• Méthode incomplète : pas de parcours exhaustif de l'espace des états, ni de celui des solutions.
• Pas de garantie d'obtenir la meilleure solution mais plus rapide qu'une méthode complète.
• Plusieurs possibilités : ici uniquement par recherche locale et donc raisonnement sur l'espace des solutions et pas l'espace des états.
• Le principe : on part d'une solution et on cherche à l'améliorer en étudiant ses voisines.
• Donne des solutions approchées (parfois des optimums locaux).
• Deux exemples : les Hill-Climbing et le Tabou.