Introduction à la métaheuristique pour la combinatoire optimisation

Gilles Trombettoni

Université de Montpellier; http://www.lirmm.fr/~trombetton/cours/local.pdf

Optimisation

Objectif: Trouver une solution **satisfaisant** les contraintes et qui est **optimal** par rapport à un critère donné.

Deux problèmes:

1. Optimisation contrainte (optimisation sous contraintes): voir ci-dessus
2. **Optimisation: il suffit de minimiser un critère**

Deux grandes approches:

1. Algorithmes complets / exacts / garantis
2. **Algorithmes incomplets / inexacts (heuristique ou métaheuristique)**

Objectif Fonction

Dé fi nition: la fonction de coût (critère) à optimiser. La fonction objectif peut être évaluée sur une instanciation complète.

(Une estimation de la fonction objectif peut souvent être donnée sur une instanciation partielle.)

Exemples de fonctions objectives:

- Problèmes de planification: minimisation de la date d'échéance de la dernière tâche du problème.
- Problèmes d'allocation des ressources: minimiser le nombre de ressources
- Problèmes de configuration ou de conception: minimiser le prix de production
- MAX-CSP: minimiser le nombre de contraintes violées (ou une somme pondérée des contraintes violées)

Contenu

Aperçu

Les heuristiques d'optimisation fonctionnent sur un **solution actuelle (con fi guration):** un point de l'espace de recherche (solution réelle ou non).

Quartier: l'ensemble des con fi gurations qui peuvent être obtenues par une transformation locale de la configuration courante. Exemples:

- Graph-Coloring: changer une couleur SAT:
- «flip» d'une variable booléenne
- CSP: modi fi cation d'une valeur de variable ($n\ (d - 1)$ voisins)

Evaluation d'une configuration: coût de la configuration: valeur à minimiser lors de la recherche

Recherche locale: améliorer une configuration courante par des transformations locales itératives

Modèles SAT et CSP

Dé fi nitions

Exemples: Sudoku, coloration de graphes, nombreux problèmes industriels

Résoudre le problème par **minimiser** les conflits dans Max-SAT (minimisant le nombre de clauses violées) et Max-CSP (minimisant le nombre de contraintes violées)

Algorithme de descente (algo de descente)

Aussi appelé *escalade* ou algorithme gourmand

Configuration initiale

- aléatoire: n'importe quel point de l'espace de recherche, ou
- donné par un algorithme déterministe (gourmand): un point initial pas trop mauvais devrait conduire à une bonne configuration

Recherche locale

Tant qu'un critère d'arrêt n'est pas entièrement rempli et qu'une meilleure configuration est trouvée, procédez comme suit:

1. Recherchez une meilleure configuration (ou la meilleure) dans le voisinage de la configuration actuelle.
2. Modifiez la configuration actuelle sur le voisin sélectionné (le cas échéant).

Inconvénient

L'algorithme de descente trouve un **optimum local.**

Inconvénients de la métaheuristique

- **Incomplétude:** la recherche n'est pas systématique (toutes les possibilités ne sont pas essayées) = ⇒
  aucune garantie que la meilleure solution a été trouvée (aucune preuve de la meilleure solution)

- **Optimal local:** une recherche locale peut être bloquée dans un optimum local.
  Il peut être bloqué sur un plateau et visiter parfois plusieurs fois les mêmes configurations.

- **Sensibilité à la con fi guration initiale**

Plusieurs améliorations

La plupart des améliorations suivantes visent à éviter les minima et les plateaux locaux. Plus généralement, ils suivent un **Intensi fi cation / Diversi fi cation** mécanisme.

- Interrompez la recherche en cours et réessayez avec d'autres con fi gurations initiales
  = ⇒ **GSAT (** con fi gurations initiales aléatoires) ou autre **multi-démarrage** stratégies avec des points de relance sélectionnés.
- Gérer plusieurs con fi gurations en parallèle (une «population» de con fi gurations)
  = ⇒ **algorithmes génétiques, GWW (** et métaheuritiques des fourmis et des abeilles).
- Enregistrez les derniers mouvements pour éviter de boucler sur les mêmes con fi gurations
  = ⇒ **recherche tabou (TS)**
- Acceptez parfois une configuration qui donne une configuration pire
  = ⇒ **recuit simulé (SA),** acceptation de seuil (TA)
- Utilisez uniquement la gestion des voisins pour intensifier ou diversifier la recherche: stratégies de liste de candidats (CLS), y compris **IDWalk.**

Une métaheuristique générique

Objectif: concevoir (et expliquer) la plupart des heuristiques d'optimisation existantes

Paramètres:

- Max-Essais: nombre de fois qu'une recherche locale est effectuée (à partir de différentes con fi gurations initiales)
- Max-Moves: nombre maximum de pas dans chaque marche
- Accepté? (X, x ': configurations): booléen
  La fonction vérifiant si le voisin X' de X est une décision acceptable
- Max-voisins: nombre maximum de voisins visités pour chaque déménagement
- Min-voisins: nombre minimum de voisins visités pour tout déménagement
- Non-acceptation: valeur prise en compte lorsqu'aucun voisin n'a été accepté (parmi Max-voisins ceux).
  Peut être égal à:
  - ne bouge pas: aucun nouveau voisin n'est sélectionné (et la marche en cours est arrêtée)
  - un voisin: tout voisin visité est sélectionné (par exemple le dernier)
  - meilleur voisin: un "moins mauvais voisin" est sélectionné

Algorithme GenericMetaheuristic (...) Renvoie: *une con fi guration*

```
meilleur ← ⊥
pour i = 1 à Max-Essais faire
    X ← Configuration initiale (...)
    j ← 0
    meilleure marche ← ⊥
    tandis que j < Max-Moves faire
        X ← Générique-Move ( X)
        meilleure marche ← Minimum (meilleure marche, X)
    fin
    meilleur ← Minimum (meilleure, meilleure marche)
fin
revenir meilleur
```

fin.

Acceptation d'un déménagement

Algorithme Generic-Move (*cur: configuration actuelle)* **Retour:** *voisin con fi guration*

je ← 0

meilleur? ← *(* Min-Voisins> 1) **ou (** No-Acceptation = meilleur voisin) meilleur coût ← *+ ∞;* x-meilleur ← *cur;* X ← *cur;*

accepté? ← **faux tandis que (** *je* < Min-voisins) **ou (** *je* < Max-voisins **et pas(** accepté?))

faire

X ← Générer-voisin (*init)*

si Accepté?(*cur, x)* **puis** accepté? ← **vrai**

si meilleur? **et (** Coût(*x)* < meilleur coût) **puis**

x-meilleur ← *X*

meilleur prix ← Coût(*X)*

fin

je ← *je +* 1

fin

si accepté? **puis**

si meilleur? **puis**

revenir x-meilleur

autre

revenir *X*

fin

fin

si Pas d'acceptation = meilleur voisin **puis retour** x-meilleur

si Pas d'acceptation = un voisin **puis retour** *X*

si Pas d'acceptation = pas de mouvement **puis retour** *cur*

fin.

Méthode de recuit simulé (SA)

En francais: algorithme du *recuit simulé*

L'un des plus anciens algorithmes de recherche locale

Déduit de Générique-Move:

- Générer-voisin (*cur):* n'importe quel voisin (sélectionné au hasard)
- Min-voisins = nombre de voisins
  (une variante: Min-Neighbours = Max-Neighbours)
- Pas d'acceptation = pas de mouvement (+ interruption de la marche)
- Accepté?(*cur, x):* Coût(*X)* ≤ Coût(*cur)* **ou**
  Aléatoire () < exp ($-\Delta$ / *T)*

Méthode de recuit simulé (SA)

Algorithme SA-Move (*cur: configuration actuelle, T: une température,*
Min-voisins: *nombre de voisins)* **Retour:** *con fi guration du voisin*

```
meilleur prix ← + ∞; x-meilleur ← cur je ← 0; accepté?
← faux
tandis que ( je < Min-voisins) faire
    X ← Générer-voisin ( cur)
    si Coût( X) ≤ Coût( cur) ou Aléatoire () < exp ( - Δ
                                                      T) puis
        accepté? ← vrai
    fin
    si Coût( x) < meilleur prix puis
        x-meilleur ← X
        meilleur prix ← Coût( X)
    fin
    je ← je + 1
fin
si accepté? puis
    revenir x-meilleur
autre
    revenir cur
fin
```

fin.

Méthode de recuit simulé (SA)

Le paramètre important est le **Température** *T,* inspiré du phénomène physique de *recuit:*

- Δ représente le degré de détérioration du critère,

  par exemple, le nombre supplémentaire de contraintes violées dans MAX-CSP.

- Une température élevée *T* permet à l'algorithme d'échapper aux minima locaux.

- Une température basse fait de l'algorithme un algorithme gourmand.
- La température doit diminuer progressivement (processus de recuit):

  Théorique *convergence* avec une diminution infiniment douce.

  Une variante: l'algorithme Metropolis avec une température constante.

La méthode Tabu Search (TS)

Idée: Gestion d'un *liste tabu:* liste de longueur constante L qui enregistre le L derniers coups (FIFO). Pour les CSP, un mouvement $X' - X$ dans la liste tabu se trouve la variable modifiée (la valeur n'est pas stockée). La liste tabu évite de regarder plusieurs fois la même configuration.

Dé fi nition:

- Min-voisins = nombre de voisins
  (une variante: Min-Neighbours = Max-Neighbours)
- Pas d'acceptation = pas de mouvement (+ interruption de la marche)
- Accepté?(*cur, x,* tabu-liste): $X - cur \in$ /tabu-liste
  ou X a le meilleur prix jamais trouvé (*aspiration).*

Un paramètre important est la longueur L de la liste tabou. Question: qu'est-ce qui permet à l'algorithme d'échapper aux minima locaux?

La méthode Tabu Search (TS)

Algorithme TS-Move (*cur: configuration actuelle, entrée-sortie* tabu-liste, *L: longueur max de la liste tabu,* Min-voisin: *nombre de voisins)* **Retour:** *con fi guration du voisin*

meilleur prix ← *+ ∞;* x-meilleur ← *cur je* ← 0; accepté? ←

faux

tandis que (*je* **<** Min-voisins) **faire**

X ← Générer-voisin (*cur)*

si *X - cur* ∈ / tabu-liste *ou* Coût(*x)* **<** meilleur coût (*aspiration)* **puis**

accepté? ← **vrai**

fin

si Coût(*x)* **<** meilleur prix **puis**

x-meilleur ← *X*

meilleur prix ← Coût(*X)*

fin

je ← *je +* 1

fin

si accepté? **puis**

tabu-list.PushEnd (x-meilleur)

si (taille (tabu-list)**>** *L)* **puis** tabu-list.PopFirst ()

revenir x-meilleur

autre

revenir *cur*

fin

fin.

Variantes de la recherche Tabu

Fred Glover a introduit de nombreux mécanismes dans le schéma de recherche tabou pour résoudre divers problèmes de recherche opérationnelle. Deux variantes:

- *Tabu probabiliste:* une probabilité d'acceptation est associée à chaque voisin (la somme est égale à 1). La valeur dépend du moment où un mouvement a été poussé dans la liste, de la qualité du voisin, etc.

- Liste de tabous dynamiques: la longueur L de la liste tabou est modi fi ée au cours du temps:
  - L suit un schéma (par exemple, un sinosoide) modifiant le rapport Intensi fi cation vs Diversi fi cation au cours du temps: *oscillation stratégique.*
  - L est *adaptatif,* c'est-à-dire change en fonction de la dif fi culté d'améliorer la configuration actuelle.

Stratégies de liste de candidats et IdWalk

Principe: Soyez très prudent dans l'analyse des candidats (voisins) pour le prochain mouvement: encodez la plupart des mécanismes de recherche locaux (comme Intensi fi cation vs Diversi fi cation) à l'intérieur du
fonction **Generic-move!**
Exemple: le **Marche de diversification d'intensi fi cation (IDW)** algorithme:
Dé fi nition:

- Accepté?(*cur, x): coût (x) ≤ coût (cur) (* composante gourmande: Intensi fi cation)
- Min-voisins = 0
- Pas d'acceptation = un mouvement ou meilleur coup
  (composante aléatoire: Diversi fi cation)
- Paramètre principal à régler: Max-voisins avec 3 rôles:
  1. limite le nombre de voisins explorés,
  2. doit être suffisamment grand pour intensifier la recherche,
  3. doit être suffisamment petit pour diversifier la recherche

  (avec Non-Acceptation).

Algorithme IDW-Move (*cur: configuration courante,* Max-voisin: *nombre de voisins)* **Retour:** *con fi guration du voisin*

```
meilleur prix ← + ∞; x-meilleur ← cur je ← 0; accepté?
← faux
tandis que ( je < Max-voisins et pas( accepté?)) faire
    X ← Générer-voisin ( cur)
    si Coût( X) ≤ Coût( cur) puis fin
        accepté? ← vrai
    si Coût( x) < meilleur prix puis
        x-meilleur ← X
        meilleur prix ← Coût( X)
    fin
    je ← je + 1
fin
si accepté? puis
    revenir X
autre
    revenir x-meilleur / * Une variante: revenir X */
fin
```

fin.

Gestion d'un **population** de con fi gurations, appelées **personnes**

Algorithme *Schéma GA*

tandis que *pas d'individus satisfaisants dans la population* **faire**

Sélectionner des individus de la population pour la reproduction

Appliquer différents opérateurs de reproduction sur des individus sélectionnés:

- **mutation:** génération d'un voisin d'un individu
- **croisement:** mélanger deux con fi gurations (individus) pour générer un nouvel individu

fin

Sélection: conserver un sous-ensemble de la nouvelle population (sélection naturelle)

fin.

Codage: un individu est constitué d'un chromosome: une séquence de bits

Un autre algorithme de population: Go With the Winners

GWW gère plusieurs con fi gurations (appelées *particules)* et un seuil (en français: seuil).

Initialisation: *B* les particules sont distribuées au hasard; un seuil est placé au prix de la pire particule.

Boucle principale: Répète jusqu'à *aucune particule ne reste sous ou au seuil:*

1. **Redistribution: (** mauvais) les particules au-dessus du seuil sont «redistribuées»: une particule redistribuée est remplacée par une copie d'une autre particule (sous le seuil; choisie au hasard).

2. **Randomisation:** Une marche aléatoire de longueur *S* est effectuée: chaque étape de la marche déplace une particule vers un voisin *qui reste égal ou inférieur au seuil.*

3. Réduisez la valeur seuil de 1.

La population est un ensemble de con fi gurations.

Fonction d'évaluation d'un individu: nombre de con fl its, somme pondérée des con fl its ...

Dif fi culté: l'opérateur de croisement n'est pas pertinent
= $\Rightarrow$ difficile de générer un meilleur individu.

- Le point de croisement ne prend pas en compte le nombre de con fl its.
- Un chromosome perd la topologie du système de contraintes (cet inconvénient est également vrai pour la plupart des problèmes combinatoires non structurés.)

Réparation heuristique par Minton (pour les CSP)

Initialisation: instanciation gourmande de variables: choisissez itérativement la variable qui produit le plus petit nombre de conflits avec les variables précédentes.

Réparation: Tandis que *il y a un conflit* faire:

- choisissez une variable X dont la valeur donne au moins un conflit
- changer la valeur de X à une nouvelle valeur qui minimise le nombre de con fl its

Min-voisins? Non-acceptation
? meilleur voisin? Accepté? ? ...?

GSAT

Initialement développé pour SAT (satis fi abilité d'une formule booléenne)

Dé fi nition: plusieurs essais de l'algorithme de descente

- Quartier: le n les con fi gurations où une seule variable est «retournée».

- Max-Essais >> 1
- Max-Moves est limité
- Max-Voisins = nombre de voisins
- Min-Neighbours = Max-Neighbours
- Pas d'acceptation = meilleur coup

  (ou pas de mouvement + interruption de la marche)
- Accepté?(x, x'): Coût(X') ≤ Coût(X)

Max-Essais permet à GSAT de trouver plusieurs minima locaux, l'un étant peut-être un minimum global

WalkSAT (pour SAT)

Choisissez un **insatisfait** clause *C*

(Retourner n'importe quelle variable dans *C* va au moins fi xer *C)*

Calculer un «score de rupture»: pour chaque var *v* dans *C,* retournement *v*

briserait combien d'autres clauses?

1. Si *C* a toutes les variables avec un score de rupture de 0:

   Choisissez au hasard parmi les vars avec un score de rupture de 0

2. sinon, avec probabilité *p:*

   - Choisissez au hasard parmi les variables avec un score de rupture minimum
   - else: choisir au hasard parmi toutes les variables de *C (* grande diversi fi cation)

Exemple d'expériences

le15c, le25c et plat28 sont des instances de coloration de graphes (codées comme des instances MAX-CSP); celar6, celar7, celar8 sont des instances d'assignation de fréquence de liaison radio.

Une entrée contient le coût moyen (sur 10 ou 20 essais). Le meilleur coût sur les 10 ou 20 essais apparaît entre parenthèses.

La dé fi nition du quartier est cruciale! (non détaillé ici)

	le15c	**le25c**	**plat28**	**celar6**	**celar7**	**celar8**
# couleurs	15	25	31			
Contre la montre	2 min	14 min	9 min	14 min	6 min	50 min
Metrop.	5,9 (2)	**3,1 (** 2)	0,9 (**0)** 5048 (3906)		6 10 6 (2,9 10 6)	410 (300)
SA	9,6 (**0)**	5,8 (4)	1,8 (**0)** 4167 (3539)		1,2 10 6 (456893)	281 (264)
Tabu	1,5 (**0)**	3,7 (3)	2,5 (1) 4183 (3935)		1,2 10 6 (620159)	373 (315)
IDWalk	0,5 (**0)**	**3,1 (1)**	**0,8 (0)** 3447 (**3389)**		373334 (343998)	291 (273)
GWW	536 (410) 17,1	(14) 6,6 (6) 3648 (3427) 583278 (456968) 276 (265)				
GWW-idw	**0 (0)**	4 (3)	1,3 (**0) 3405 (3389)**		**368452 (343600)**	**267 (262)**

Synthèse

Les heuristiques d'optimisation sont incomplètes, mais sont moins sensibles aux mauvais choix que les méthodes exactes.

Ce qui est important si l'on veut trouver, avec la recherche locale, une bonne solution à un problème d'optimisation donné:

1. Essayez plusieurs manières d'encoder le problème (dé fi nition du voisinage).
2. Comprendre les intuitions (c'est-à-dire les mécanismes utiles) derrière les principales heuristiques d'optimisation.
3. Soyez pragmatique, ne soyez pas convaincu: essayez différentes heuristiques, au lieu d'en régler une seule: utilisez une bibliothèque!
4. Privilégiez les idées les plus simples car elles sont plus rapides à comprendre et à expérimenter.
5. Un réglage fin des paramètres ne paie généralement pas. Privilégiez les
6. paramètres adaptatifs.