# Algorithmes d'optimisation

Version 1.6

Michaël Guedj

Algorithmes d'optimisation de Michaël Guedj est mis à disposition selon les termes de la licence Creative Commons Attribution 4.0 International.

# Table des Matières

# 1 Approche des problèmes d'optimisation

Nous considérons une veille radio que l'on trouve dans une brocante ; elle comporte une molette : $A$ ; et 3 boutons : $B$, $C$ et $D$.

- $A$ permet de capter des fréquences (100 fréquences disponibles) ;
- $B$, $C$ et $D$ sont trois boutons qui peuvent prendre 10 valeurs chacune.

Le but est de trouver une station diffusant une chanson que l'on aime bien (la meilleure possible) , et avec une bonne qualité de diffusion (la meilleure possible).

Une solution est la donnée d'un quadruplet $(a, b, c, d) \in A \times B \times C \times D$ où $a$, $b$, $c$ et $d$ indiquent des positions, de la molette et des différents boutons.

Au total : $100 \times 10 \times 10 \times 10 = 100\ 000$ solutions possibles.

Pour chaque solution, i.e. chaque possibilité de quadruplet, on donne une note.

**Exemple de notation.**

On entend :

- Seulement des bruits de grésillement $\rightarrow 0/20$ ;
- Des voix, avec beaucoup de grésillement $\rightarrow 5/20$ ;
- Une chanson avec grésillement $\rightarrow 12/20$ ;
- Des voix ; avec bonne qualité d'écoute $\rightarrow 14/20$ ;
- Une chanson "sympathique" ; avec bonne qualité d'écoute $\rightarrow 17/20$ ;
- Une chanson que l'on aime davantage ; avec bonne qualité d'écoute $\rightarrow 19/20$.

Nous nous faisons face à un problème d'**optimisation** (maximisation / minimisation) ; il s'agit, en effet, de trouver une solution qui **maximise** la note.

Etant donné le nombre de solutions possibles ; on utilise alors une heuristique de résolution. (Dans notre exemple, tester 100 000 configurations possibles n'est pas acceptable pour un humain).

# 2 Définitions

## 2.1 Problèmes d'optimisation

Soient :

(i) un ensemble d'entrées $E$ contenant les solutions possibles ;

(ii) une fonction objectif $f : E \to \mathbb{R}$ associant une valeur à chaque solution possible.

One note

$$\operatorname{argmax} f := \{s \in E \mid \forall s' \in E, f(s') \leq f(s)\}$$

Un problème d'optimisation consiste à trouver la meilleure solution.

**Définition** (problème d'optimisation)**.**

- Entrée.
  - Un ensemble d'entrées $E$.
  - Une fonction objectif $f : E \to \mathbb{R}$.
- Objectif.

$$s^* = \operatorname{argmax} f$$

## 2.2 Heuristiques

**Définition** (heuristique)**.** Une heuristique (du grec ancien *eurisko* « je trouve ») est une méthode de calcul qui fournit "rapidement" une solution "réalisable" (non nécessairement optimale ou exacte) pour un problème d'optimisation considéré "difficile".

**Note :** Une heuristique s'impose quand les algorithmes de résolution optimale ou exacte sont de complexité exponentielle, et dans beaucoup de problèmes jugés "difficiles". L'usage d'une heuristique est également pertinent pour calculer une solution approchée d'un problème, ou encore pour "accélérer" un processus de résolution exacte. Notamment, une heuristique peut-être conçue pour un problème particulier, en s'appuyant sur sa structure propre, tout en pouvant contenir des principes plus généraux.

## 2.3 Algorithmes gloutons

**Définition** (algorithme glouton)**.** Un algorithme glouton (*greedy algorithm* en anglais) 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 heuristique (la solution optimale n'est pas garantie).

## 3 Retour sur trace (*Backtracking*)

**Définition** (problème des $n$ reines)**.**

- Entrée.
  - Un échiquier $n \times n$.
  - $n$ reines, issues de l'ensemble

$$R := \{R_i \mid i = 1, ..., n\}$$

- Problème.
  - Placer les $n$ reines sur l'échiquier, de sorte à ce qu'aucune ne puisse en "manger" une autre.

**Théorème.** *La résolution du problème des $n$ reines, par l'algorithme de retour sur trace, est en $O(n!)$.*

*Preuve.*

- $n$ majore le nombre de positions de $R_1$ ;
- $R_1$ placée ; $n-1$ majore le nombre de positions de $R_2$ ;
- $R_1$ et $R_2$ placées ; $n-2$ majore le nombre de positions de $R_3$ ;
- ... ;
- $R_1, ..., R_i$ placées ; $n-i$ majore le nombre de positions de $R_{i+1}$ ;
- ... ;
- $R_1, ..., R_{n-2}$ placées ; $n-(n-2) = 2$ majore le nombre de positions de $R_{n-1}$ ;
- $R_1, ..., R_{n-1}$ placées ; $n-(n-1) = 1$ majore le nombre de positions de $R_n$.

Le nombre de positions effectuées, par l'algorithme de retour sur trace, est alors majoré par $n!$. □

# 4 Problème de la somme du sous-ensemble (*Subset Sum Problem*)

## 4.1 Définition du problème

**Définition** (problème de la somme du sous-ensemble)**.**

- Entrée.
  - Un tableau $t$, de taille $n$, à valeurs entières.
  - Une capacité $c \in \mathbb{N}$.
- Problème.

  Trouver $k$ indices de $t$, distincts, soient $i_1, ..., i_k$, tels que la somme

  $$t[i_1] + ... + t[i_k]$$

  soit la plus grande possible, tout en assurant la contrainte

  $$t[i_1] + ... + t[i_k] \leq c$$

**Théorème.** *Le problème de la somme du sous-ensemble est NP-difficile.*

*Proof.* Admis. □

## 4.2 Heuristique gloutonne

**Algorithm 1** `greedy`$(t, n, c)$

```
1: trier t selon l'ordre décroissant
2: res ← [ ]
3: val ← 0
4: for i ← 0, ..., n − 1 do
5:     if val + t[i] ≤ c then
6:         res ← res + [ i ]
7:         val ← val + t[i]
8:     end if
9: end for
10: return res
```

Complexité : $O(n. \log n)$.

# 5 Problème de la coloration de graphe (*Graph Coloring Problem*)

## 5.1 Définition du problème

**Définition** (problème de la coloration de graphe)**.**

- Entrée.

  Un graphe $G$, non orienté, sans boucle.

- Problème.

  Trouver le plus petit entier $k$, tel que $G$ soit $k$-colorable.

**Théorème.** *Le problème de la coloration de graphe est NP-difficile.*

*Proof.* Admis. □

## 5.2 Coloration gloutonne

**Algorithm 2** `Greedy_Colouring`$(G = (S, A))$ tel que $S = \{s_i : i \in \{0, ..., n-1\}\}$

1: $color \leftarrow$ un dictionnaire vérifiant : $\forall s \in S, color[s] = -1$
2: **for** $i \leftarrow 0, .., n-1$ **do**
3: $\quad C_i \leftarrow \{color[s_j] : (s_i, s_j) \in A\}$ ▷ ensemble des couleurs des sommets adjacents à $s_i$
4: $\quad color[s_i] \leftarrow \min\{c \in \mathbb{N} : c \notin C_i\}$ ▷ le plus petit entier positif n'appartenant pas à $C_i$
5: **end for**
6: **return** *color*

# 6 Problème du voyageur de commerce (*Travelling Salesman Problem*)

## 6.1 Définition du problème

**Définition** (cycle hamiltonien)**.** Un cycle hamiltonien, d'un graphe, est un cycle passant par tous les sommets du graphe, une fois, et une seule.

**Définition** (problème du voyageur de commerce)**.**

- Entrée.

  Un graphe $G = (S, A, cost)$, tel que :
  - $G$ est non orienté ;
  - $G$ est complet ;
  - Les arcs de $G$ sont valués par la fonction $cost : A \to \mathbb{R}$.

- Problème.

  Trouver le cycle hamiltonien ayant le coût le plus faible.

**Théorème.** *Le problème du voyageur de commerce est NP-difficile.*

*Proof.* Admis. □

## 6.2 Heuristique : algorithme du plus proche voisin

**Algorithm 3** `nearest_neighbour`$(G = (S, A, cost))$

1: $todo \leftarrow \{\}$
2: $res \leftarrow$ `[ ]`
3: $s_0 \leftarrow$ choisir un sommet arbitraire de $G$
4: $todo \leftarrow G - \{s_0\}$
5: $res \leftarrow res +$ `[` $s_0$ `]`
6: $s \leftarrow s_0$
7: **while** $todo \neq \{\}$ **do**
8: $\quad s' \leftarrow$ le sommet le plus proche de $s$
9: $\quad todo \leftarrow todo - \{s'\}$
10: $\quad res \leftarrow res +$ `[` $s'$ `]`
11: $\quad s \leftarrow s'$
12: **end while**
13: **return** $res \leftarrow res +$ `[` $s_0$ `]`

# 7 Algorithmes génétiques

On cherche à maximiser la fonction objectif $f : E \to \mathbb{R}$ ; où $E$ est un ensemble d'entrée fini.

Exemples d'ensembles $E$ (où $n$ est un entier) :

- $E = \texttt{float}^n$ ;
- $E \subset \texttt{float}^n$ ;
- $E = \mathbb{B}^n$ ;
- $E = S_n$ (l'ensemble des permutations de $\{1, ..., n\}$).

**Algorithm 4** `algorithme_genetique`( $n$ : taille d'un individu, $p$ : nombre d'individus, ...)

```
1:  P ← Population initiale
2:  res ← None                                  ▷ en considérant f(None) = −∞
3:  while not (Critère Fin) do
4:      Calculer la valeur de fitness de chaque individu   ▷ i.e. f(p), ∀p ∈ P
5:      best ← le meilleur individu de P   ▷ au sens de la maximisation de f
6:      if f(best) > f(res) then
7:          res ← best
8:      end if
9:      P ← selection(P)
10:     P_new ← reproduction(P)
11:     P_new ← mutation(P_new)
12:     P ← P_new
13: end while
14: return res
```

(Les phases de *selection* et de *reproduction* peuvent être regroupées en une routine).