REPUBLIQUE ALGERIENNE DEMOCRATIQUE ET POPULAIRE

Ministère de l'enseignement supérieur et de la recherche scientifique

Université Larbi Ben M'Hidi Oum El Bouaghi

Faculté des sciences exactes et science de la nature et de la vie

Département de mathématique d'informatique et

Cours : Méthodes Algorithmiques

Chapitre Initialisation



Proposé par Dr. Kenza Belhouchette

Edition 1.0

2024/ 2025

Introduction

L'optimisation et la complexité sont deux concepts fondamentaux en informatique, en particulier dans l'analyse des algorithmes. Comprendre ces notions permet d'évaluer l'efficacité d'un algorithme, tant en termes de temps d'exécution qu'en termes d'utilisation des ressources mémoire. Dans ce chapitre, nous nous pencherons sur les bases de l'optimisation et de la complexité.

1. L'optimisation

Un problème d'optimisation consiste à trouver la meilleure solution possible parmi un ensemble de solutions admissibles.

1.1 Définition mathématique d'un problème d'optimisation :

Soit $X \subseteq \mathbb{R}^n$ un sous-ensemble non vide et $f : X \rightarrow \mathbb{R}$ une fonction à valeur réelle.

Le but est de trouver

$$x^* \in X \text{ tel que } f(x^*) \leq f(x), \forall x \in X.$$

Exemple :
Minimiser $f(x) = x^2$.

Ici, la solution optimale est $x^* = 0$ car $f(0) = 0$ est la valeur minimale.

1.2 Terminologie

- **Variables de décision :** Le vecteur $x \in X$ est appelé vecteur de décision. Les éléments formants le vecteur de décision sont appelés variables de décision. Ce sont les variables recherchées dans le problème.
- **Dimension de problème** : La dimension du problème est représentée par le nombre entier $n \in N^*$. Elle représente le nombre de variables de décision. Une solution à un problème de 3 dimensions est un vecteur de 3 éléments (3-uplet).
- **Espace de recherche et Solution admissible** : L'ensemble X de dimension n contient les valeurs que le vecteur x peut prendre. Ces dernières sont appelées solutions admissibles ou faisables (ou encore solutions candidates).

  Une solution admissible est une affectation de toutes les variables de décision , qui répond aux contraintes du problème. L'ensemble X est connu sous différentes appellations : ensemble admissible, espace de solution, espace de recherche, domaine de recherche etc.

- **Contraintes** : L'espace de recherche X est limité par quelques règles. Une contrainte est une condition que doivent respecter les vecteurs de décision du problème.

- **Solution optimale** : C'est la meilleure solution x ∈ X du problème. . Elle est appelée solution globale ou optimale.

- **Fonction objectif** : la fonction f, à valeur scalaire, sert de critère pour déterminer la meilleure solution du problème. Elle est appelée fonction d'objectif (fonction fitness, critère, etc). Elle attribue à chaque solution x ∈ X de l'espace de recherche, un nombre réel indiquant sa valeur (son score).

- **Minimisation** : Une minimisation revient à trouver un élément x de X tel que : ∀x ∈ X :f(x) f(x). On dit qu'on cherche à minimiser la fonction f sur l'ensemble X. Dans ce cas la fonction objectif f est connue comme une fonction de coût . Le vecteur x est une borne inférieure de l'ensemble X.

- **Maximisation** : Une maximisation revient à trouver un élément x de X tel que : ∀x ∈ X :f(x) f(x). On dit qu'on cherche à maximiser la fonction f sur l'ensemble X. Dans ce cas la fonction objectif f est connue comme une fonction d'utilité ou de profit .

  Le vecteur x est une borne supérieure de l'ensemble X. Minimum et maximum stricte :

  On parle d'un minimum stricte (respectivement un maximum stricte) si :

  $$\forall x \in X : f(x) < f(x) \text{ (respectivement } f(x) > f(x)).$$

- **Voisinage** : Une solution voisine d'une solution x est définit comme une légère modification de la solution x . Cette modification est appelée aussi mouvement. Le voisinage de la solution x est l'ensemble des solutions voisines de x.

- **Optimum global et local** : Un optimum global est la valeur de la fonction objectif de la solution globale . On peut avoir plusieurs solutions globales mais un seul optimum globale . Un optimum local reflète la meilleure solution dans un entourage voisin.

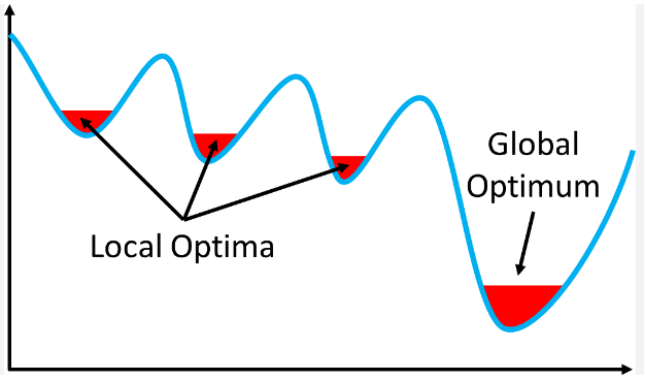


Fig1. **Optimum global et local**

1.3 Classification des problèmes d'optimisation

Optimisation continue

L'optimisation continue traite des problèmes où les variables de décision peuvent prendre une infinité de valeurs dans un domaine continu.

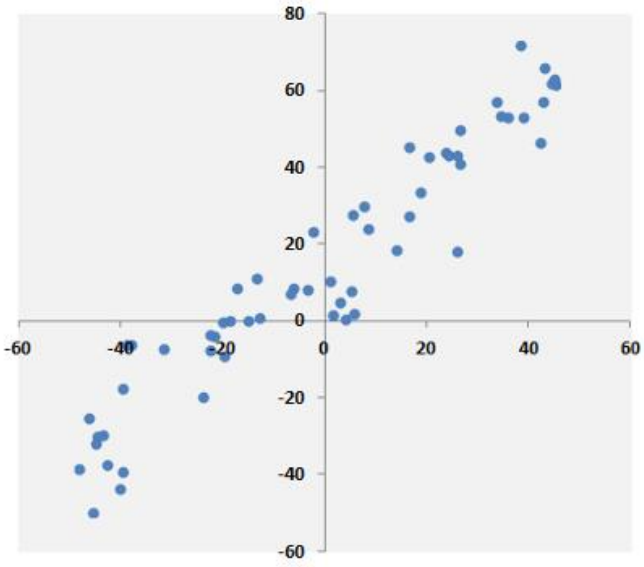


Fig2. **Optimisation continue**

Exemple :
minimiser $f(x) = x_1^2 + x_2^2$ sous la contrainte $x_1 + x_2 \leq 5$.

Optimisation discrète

L'optimisation discrète concerne les problèmes où les variables de décision ne peuvent prendre que des valeurs discrètes (comme des entiers).

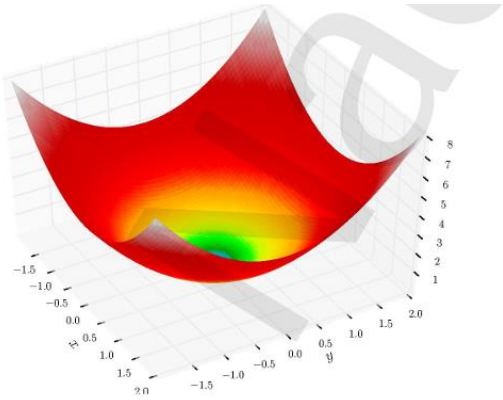


Fig3. **Optimisation discrète**

Exemple :
minimiser $c^T x$ sous contraintes $Ax \leq b$, où les variables x sont des entiers.

Les problèmes de sac à dos sont un exemple typique. (sera detaillé au chapitre suivant)

Optimisation combinatoire

L'**optimisation combinatoire** est un sous-ensemble de l'optimisation discrète qui concerne les problèmes où il faut **sélectionner, arranger, ou combiner** des objets discrets selon certaines contraintes afin d'optimiser une fonction objectif

Un exemple classique est le problème du voyageur de commerce (TSP). Exemple : Minimiser la somme des distances entre plusieurs villes en suivant un chemin optimal.

2. Rappels sur la complexité algorithmique

La complexité algorithmique est une mesure de l'efficacité d'un algorithme, généralement en termes de :

- **temps** (temps d'exécution)
- **mémoire** (espace utilisé)

L'analyse de la complexité permet de comparer différents algorithmes et de déterminer lesquels sont les plus adaptés pour résoudre un problème donné, surtout lorsqu'il s'agit de grandes quantités de données.

2.1 Complexité en Temps

La complexité en temps mesure le temps d'exécution d'un algorithme en fonction de la taille de l'entrée n.
Les classes de complexité les plus courantes sont :

O(1) : Temps constant

L'algorithme prend toujours le même temps, quelle que soit la taille de l'entrée.

Exemple : Accès direct à un élément dans un tableau.
O(log n) : Temps logarithmique.

Le temps de traitement croît de façon logarithmique avec la taille de l'entrée.

Exemple : Recherche binaire dans une liste triée.

O(n) :Temps linéaire :

Le temps d'exécution est proportionnel à la taille de l'entrée.

Exemple : Parcourir tous les éléments d'une liste.

O(n log n) : Temps quasi-linéaire.

Croît un peu plus rapidement que linéairement, souvent rencontré dans les algorithmes de tri efficaces.

Exemple : Tri fusion (Merge Sort).

O(n^2) : Temps quadratique.

Le temps d'exécution croît comme le carré de la taille de l'entrée

Exemple : Tri à bulles (Bubble Sort).
O(2^n) : Temps exponentiel.

Le temps d'exécution croît de manière exponentielle avec la taille de l'entrée, rendant l'algorithme impraticable pour de grandes entrées.

Exemple : Résolution du problème de la somme des sous-ensembles par force brute.
O(n!) : Temps factoriel.

Très coûteux, généralement associé à des algorithmes d'exploration exhaustive

Exemple : Problème du voyageur de commerce résolu par énumération des permutations.

2.2 Complexité en Espace

Mesure la quantité de mémoire qu'un algorithme utilise pendant son exécution, en fonction de la taille de l'entrée. Tout comme la complexité en temps, la complexité en espace est souvent exprimée en notation asymptotique O(·), qui décrit comment la mémoire utilisée par un algorithme augmente avec la taille de l'entrée.

Exemple de complexité en espace :

Complexité constante O(1) :

Un algorithme a une complexité en espace de O(1) s'il n'utilise qu'une quantité fixe de mémoire, quelle que soit la taille de l'entrée.

Exemple : Une fonction qui inverse deux variables utilise toujours la même quantité de mémoire, quel que soit le nombre d'éléments dans la liste.

Complexité linéaire O(n):

Un algorithme a une complexité en espace de O(n) s'il utilise une quantité de mémoire proportionnelle à la taille de l'entrée.

Exemple : Si un algorithme doit stocker tous les éléments d'une liste de taille n, la quantité de mémoire qu'il utilise est proportionnelle à n.

Complexité quadratique O(n^2):

Un algorithme a une complexité en espace de O(n^2) s'il utilise une quantité de mémoire proportionnelle au carré de la taille de l'entrée.

Exemple : Les algorithmes qui utilisent des matrices, comme l'algorithme de multiplication de matrices, ont souvent une complexité en espace de O(n^2) car une matrice de taille n×n nécessite n^2unités de mémoire.

2.3 Relation entre la complexité en temps et en espace :

Dans certains algorithmes, il y a un compromis entre la complexité en temps et la complexité en espace. Par exemple, certains algorithmes de tri peuvent utiliser moins de mémoire (comme le tri par insertion), mais au prix d'une augmentation de la complexité en temps.

La complexité en espace est importante lorsqu'on manipule de grandes quantités de données, par exemple dans le traitement de grandes bases de données, la simulation scientifique, ou l'apprentissage automatique, où l'utilisation de la mémoire doit être optimisée pour éviter une surcharge.

Conclusion

Ce chapitre a offert une introduction aux notions fondamentales d'optimisation et de complexité en informatique. Nous avons exploré les définitions clés et les outils nécessaires pour évaluer l'efficacité des algorithmes, tant en termes de temps que de mémoire. La compréhension de ces concepts est cruciale pour concevoir des solutions performantes et adaptées aux contraintes des applications réelles. À travers les exemples et les notions abordées (au cours et au TD), vous disposez désormais d'une base solide pour analyser, comparer et optimiser les algorithmes, préparant ainsi le terrain pour des études plus approfondies en algorithmique dans les chapitres suivants.