Implémentation C++ générique de programmation dynamique pour le clustering d'un front de Pareto

Omar Arharbi

Mai 2025

Table des matières

1 Introduction

L'optimisation multiobjectif (OMO) consiste à rechercher des solutions équilibrées entre plusieurs objectifs souvent contradictoires. Par exemple, concevoir un système peu coûteux mais aussi très robuste peut conduire à des compromis difficiles, car améliorer un critère peut en dégrader un autre. Dans ce contexte, le *front de Pareto* joue un rôle essentiel : il regroupe les solutions non dominées, c'est-à-dire celles qu'on ne peut améliorer sur un objectif sans détériorer un autre.

Cependant, le front de Pareto peut contenir un très grand nombre de solutions, ce qui rend leur analyse difficile. Le *clustering* permet alors de résumer ces solutions en un petit nombre de représentants pertinents. Parmi les algorithmes de clustering robustes figurent **k-medoids** et **p-median**, qui consistent à sélectionner k points réels (appelés *médodoïdes* ou *médians*) minimisant la distance aux autres points dans leur groupe.

Ce travail vise à implémenter en C++ un algorithme générique et optimisé pour effectuer du clustering (comme le problème de k-medoids) sur des ensembles de points 1D ou 2D issus d'une optimisation multiobjectif. L'objectif est de concevoir un code modulaire, capable de gérer différentes variantes de clustering, tout en permettant l'intégration de techniques avancées comme la **parallélisation** pour gagner en efficacité.

Ce rapport présente donc les fondements de l'optimisation multiobjectif, les algorithmes de clustering utilisés, ainsi que l'implémentation proposée et les résultats obtenus.

2 Quelques définitions

2.1 Front de Pareto

2.1.1 Principes Fondamentaux

Le front de Pareto désigne l'ensemble des solutions optimales où aucune amélioration d'un critère ne peut se faire sans dégradation d'au moins un autre.

2.1.2 Caractéristiques

- **Optimalité de Pareto :** Une solution est dite optimale si elle n'est dominée par aucune autre.
- **Solutions dominées :** Toute solution située en dehors du front peut être améliorée sur au moins un critère sans perte sur les autres.

2.2 Clustering sur un Front Pareto

Dans un contexte OMO, le regroupement (clustering) permet de résumer un grand ensemble de solutions (de taille N) en un sous-ensemble plus réduit (de taille K) représentatif des compromis disponibles.

2.2.1 k-medoids : une méthode robuste

k-medoids forme k groupes de points, chacun centré sur un *médoid*, un point réel du groupe. Ce médoid est celui qui minimise la somme des distances aux autres points du cluster.

Contrairement à **k-means** (qui utilise une moyenne mathématique, sensible aux valeurs aberrantes), k-medoids choisit un point réel existant, ce qui le rend plus robuste aux données erronées.

Exemple intuitif : Si l'on regroupe des personnes selon leur taille, avec la plupart entre 1,50m et 1,90m, et une erreur indiquant 10m, **k-means** placera son centre vers cette valeur absurde. **k-medoids**, lui, choisira une taille réaliste proche des autres, **ignorant la valeur aberrante**.

Différence clé :

- **k-means :** centre = moyenne, pas un vrai point
- **k-medoids :** centre = point du groupe avec plus petite somme des distances

2.2.2 p-median : une approche logistique optimale

p-median forme p groupes de points, chacun centré sur un *median*, un point réel du groupe choisi pour minimiser la somme des distances euclidiennes (non au carré) vers tous les autres points du cluster. Cette méthode trouve ses origines dans les problèmes logistiques : placer p dépôts pour minimiser la distance totale parcourue par les clients vers leur dépôt le plus proche. Contrairement à **k-medoids** qui utilise les distances au carré (pénalisant fortement les points éloignés), p-median utilise la distance euclidienne simple, offrant une approche plus équilibrée.

Exemple intuitif : Pour implanter 3 centres de distribution dans une région, **p-median** choisira des emplacements réels (villes existantes) qui minimisent la distance totale de livraison. Si un client isolé se trouve très loin, **p-median** ne sur-pénalisera pas cette distance comme le ferait k-medoids, permettant une solution plus équitable.

Différence clé avec k-medoids :

- **k-medoids :** minimise $\sum d^2$ (distance au carré), pénalise fortement les points éloignés
- **p-median :** minimise $\sum d$ (distance simple), traitement plus équitable des distances

2.3 Programmation Dynamique (DP)

La programmation dynamique est une technique algorithmique puissante, principalement utilisée pour optimiser des solutions récursives présentant des

appels répétés sur les mêmes entrées. Elle consiste à mémoriser les résultats intermédiaires des sous-problèmes afin d'éviter des recalculs inutiles lors de l'exécution de l'algorithme. Grâce à cette approche, il est souvent possible de réduire la complexité temporelle d'un problème, passant d'un temps de calcul exponentiel à un temps polynomial.

2.3.1 Application au clustering de fronts de Pareto

Dans le contexte spécifique du clustering sur un front de Pareto bidimensionnel, la programmation dynamique exploite la propriété d'ordonnancement naturel des points non-dominés. Le problème global de partitionnement peut être décomposé en sous-problèmes optimaux portant sur des segments consécutifs du front.

Cette structuration permet l'exploration systématique et efficace de l'espace des solutions. Pour n points à regrouper en k clusters, l'approche par programmation dynamique réduit la complexité algorithmique de $O(n^k)$ (recherche exhaustive) à $O(n^3)$ (pour $k > 3$) [2], indépendamment du nombre de clusters souhaité. Cette amélioration cruciale rend possible la détermination de la partition optimale globale, évitant ainsi les minima locaux caractéristiques des méthodes heuristiques traditionnelles comme l'algorithme PAM (Partitioning Around Medoids) pour k-medoids qui va suivre une implémentation avec un glouton.

3 Implémentations

3.1 Initialisation du nombre de clusters K

Dans notre implémentation, nous avons choisi d'utiliser l'heuristique $k = \sqrt{N}$ pour initialiser le nombre de clusters. Cette règle simple et efficace est largement reconnue dans la communauté de l'analyse de données [1].

Pourquoi cette formule ? Imaginons que nous ayons 100 points à regrouper :

— Si nous créons trop peu de clusters (par exemple $k = 2$), chaque groupe contient des points trop différents, comme mettre des pommes et des voitures dans le même sac.

— Si nous créons trop de clusters (par exemple $k = 50$), nous nous retrouvons avec des groupes presque vides ou très similaires, comme séparer les pommes rouges légèrement différentes dans des sacs distincts.

La règle $k = \sqrt{N}$ offre un bon équilibre. Pour 100 points, elle suggère 10 clusters, ce qui est généralement suffisant pour capturer les structures importantes des données sans créer de groupements artificiels.

Concrètement, si notre jeu de données contient :

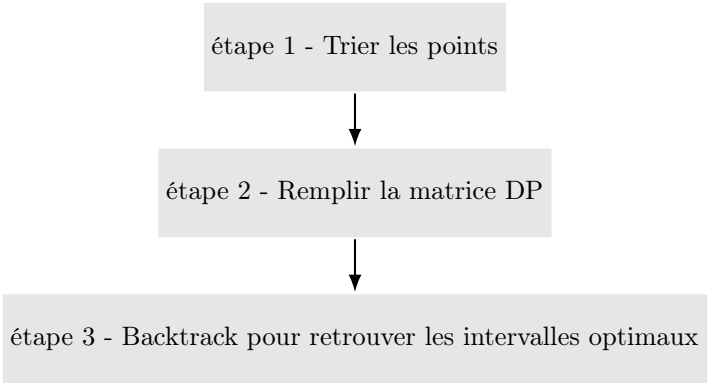
— 25 points $\rightarrow k = 5$ clusters

— 100 points $\rightarrow k = 10$ clusters

— 400 points $\rightarrow k = 20$ clusters

Cette relation entre le nombre de points et le nombre de clusters semble intuitive : lorsque nous avons plus de données, nous pouvons distinguer plus de motifs, mais pas de façon proportionnelle.

3.2 Programmation dynamique et clustering : implémentation générique



étape 1 - Trie :

On commence par trier les points du front de Pareto selon la première dimension (coordonnée x).

Exemple illustratif :

Considérons un front de Pareto 2D composé de 4 points :

— Point A : (10, 40)
— Point B : (25, 15)
— Point C : (5, 45)
— Point D : (20, 20)

Processus de tri :

1. Les points sont initialement dans l'ordre A, B, C, D
2. On trie par ordre décroissant de la coordonnée x
3. Résultat du tri : B, D, A, C

Structure de données après tri :

Le vecteur de points devient [25, 15, 20, 20, 10, 40, 5, 45]

Cette étape est cruciale car elle garantit que les clusters optimaux seront des intervalles contigus [3].

3.2.1 Implémentation de la programmation dynamique

étape 2 - Mémorisation des couts pour différents configurations :

La matrice matrixDP dans cette approche est au cœur de notre implémentation DP. Elle mémorise les coûts optimaux pour différentes combinaisons de points et de clusters, évitant ainsi les recalculs répétitifs, dans notre implémentation :

matrixDP[k][i] représente le coût optimal pour partitionner les points de 0 à i en k+1 clusters.

Exemple simple :

Prenons 5 points unidimensionnels : [2, 5, 9, 15, 22], Notre objectif est de trouver les meilleures façons de regrouper ces points en différents nombres de clusters (1 à 5).

- La première ligne de la matrice est remplie avec les coûts pour un seul cluster :

matrixDP[0][i] = coût($\{x_0, ..., x_i\}$)

- Pour les lignes suivantes (k > 0), pour chaque ligne k, on cherche la meilleure façon de diviser les points de 0 à i en k+1 clusters en testant toutes les positions possibles j pour la dernière séparation :

matrixDP[k][i] = $\min_{j \in \{k-1,k,...,i-1\}}$\{matrixDP[$k-1$][$j$]+coût($\{x_{j+1}, ..., x_i\}$)\}

On introduit un vecteur v pour stocker les coûts optimaux (avec médoïdes optimaux) pour des clusters de différentes longueurs dans un intervalle donné de points. Ce vecteur sert d'information intermédiaire pour la construction progressive de la solution dans la matrice de programmation dynamique (DP).

Pour k = 2 et un front de Pareto avec les points A, B, C et i=C qui signifie les coûts pour tous les clusters se terminant au point C :

matrixDP[A][2] = coût du cluster [A,B,C] avec A comme médoïde
matrixDP[B][2] = coût du cluster [A,B,C] avec B comme médoïde
matrixDP[C][2] = coût du cluster [A,B,C] avec C comme médoïde
v[2] = min(d[A][2], d[B][2], d[C][2]) = coût minimal pour le cluster [A,B,C]

Ce qu'il faut retenir est que l'algorithme trouve pour chaque niveau de clustering le point de séparation optimal qui minimise le coût total en combinant des solutions déjà calculées avec le coût d'un nouveau cluster

Algorithm 1 Algo générique

```
1: for i = 0 to N − 1 do
2:     v ← clusterCostsFromBeginning()        ▷ Stockera les coûts des clusters
3:     matrixDP[0][i] ← v[i]
4: end for
5: for k = 1 to K − 1 do
6:     for n = k to N − 1 do
7:         v ← clusterCostsBefore(n)          ▷ Stockera les coûts des clusters
8:         minCost ← ∞
9:         bestSplit ← 0
10:        for split = k − 1 to n − 1 do
11:            cost ← matrixDP[k − 1][split] + v[n − split − 1]
12:            if cost < minCost then
13:                minCost ← cost
14:                bestSplit ← split
15:            end if
16:        end for
17:        matrixDP[k][n] ← minCost
18:    end for
19: end for
```

Les méthodes `clusterCostsBefore` et `clusterCostsFromBeginning` vont représenter notre implémentation de l'**algorithme de clustering**. Il s'agit des seules méthodes spécifiques à l'algorithme choisi, Leur rôle principal est de remplir le vecteur v avec les coûts optimaux associés aux différents clusters.

3.2.2 k-medoids et p-median : Implémentation

Algorithm 2 clusterCostsBefore

```
1: function clusterCostsBefore(i, v)
2:     d ← matrice N × N                ▷ Stocke les coûts par médoïde et longueur
3:     for len = 0 to v.size() − 1 do
4:         start ← i − len
5:         end ← i
6:         if len = 0 then
7:             v[len] ← 0
8:             continue
9:         end if
10:        if len = 1 then
11:            computeInitialCosts(start, end, d)
12:        else
13:            updateCosts(start, end, i, len, d)
14:        end if
15:        v[len] ← findMinimumCost(start, end, len, d)
16:    end for
17: end function
```

La matrice d constitue la structure de mémorisation centrale de notre approche en programmation dynamique. Chaque élément $d[c][\text{len}]$ stocke le coût d'un cluster lorsque le point c est choisi comme médoïde pour un intervalle de longueur len. Par exemple, $d[3][1]$ représente le coût du cluster $[3, 4]$ avec le point 3 comme médoïde, où 1 indique que l'intervalle contient deux points (longueur 1).

Le remplissage de cette matrice repose sur deux méthodes principales : `computeInitialCosts` et `updateCosts`. La première initialise les coûts pour les intervalles de base (premiere ligne de la matrice), tandis que la seconde met à jour ces coûts pour des intervalles plus longs. Ces deux méthodes s'appuient sur les composantes mathématiques présentées précédemment afin de calculer les coûts optimaux pour tous les clusters possibles se terminant à un index donné i, ce qui permet en particulier de remplir le vecteur v.

Nous présentons ci-dessous les pseudo-algorithmes détaillés de ces deux méthodes, afin d'expliciter leur fonctionnement.

Algorithm 3 computeInitialCosts pour k-medoids

```
for medoid ← start to end do
    for other ← start to end do
        if other ≠ medoid then
            d[medoid][1] ← squaredDistance(other, medoid)
```

Algorithm 4 updateCosts pour k-medoids

```
1: function UPDATECOSTSITERATIVELY(start, end, lastPoint, len, d)
2:     for medoid = start to end do
3:         d[medoid][len] = d[medoid][len-1] +
   squaredDistance(lastPoint, medoid)
4:     end for
5: end function=0
```

L'algorithme `computeInitialCosts` initialise la matrice de coûts en calculant pour chaque médoïde potentiel la somme des distances au carré vers tous les autres points de l'intervalle de base. L'algorithme `updateCosts` exploite la propriété incrémentale des clusters consécutifs : lorsqu'un nouveau point est ajouté à un cluster existant, le coût peut être mis à jour efficacement en ajoutant simplement sa distance au carré vers chaque médoïde potentiel, évitant ainsi un recalcul complet.

L'implémentation du p-median suit la même structure algorithmique que k-medoids, avec une modification cruciale dans le calcul des coûts. La différence principale réside dans la fonction de distance utilisée :

— **k-medoids** : utilise la distance euclidienne au carré ($\alpha = 2$)

— **p-median** : utilise la distance euclidienne simple ($\alpha = 1$)

Cette distinction se traduit par une modification dans les méthodes `computeInitialCosts` et `updateCosts` :

Algorithm 5 computeInitialCosts - Version p-median

```
1: for medoid ← start to end do
2:     for other ← start to end do
3:         if other ≠ medoid then
4:             d[medoid][1] ← √squaredDistance(other, medoid)      ▷ Distance
   euclidienne
5:         end if
6:     end for
7: end for
```

Algorithm 6 updateCosts - Version p-median

```
1: function UPDATECOSTSITERATIVELY(start, end, lastPoint, len, d)
2:     for medoid = start to end do
3:         d[medoid][len] = d[medoid][len-1] +
   √squaredDistance(lastPoint, medoid)
4:     end for
5: end function
```

étape 3 - Backtrack :

Une fois la matrice complétée, nous connaissons les coûts optimaux pour chaque combinaison de nombre de clusters et de points, mais pas encore la structure exacte de ces clusters. La matrice nous indique, par exemple, que le coût optimal pour regrouper nos points en 3 clusters est de 4.5, mais ne précise pas où placer les séparations entre ces clusters.

C'est ici qu'intervient le backtracking (retour sur trace). Cette technique permet de reconstruire la solution optimale en partant du résultat final (coin inférieur droit de la matrice) et en remontant le chemin des décisions optimales. Pour chaque nombre de clusters k, nous déterminons le meilleur point de séparation qui, combiné avec la solution optimale pour $k-1$ clusters, produit le coût minimal.

Cette approche nous permet de construire la solution complète en déterminant les *solutionInterval* les intervalles de points appartenant à chaque cluster qui sont ensuite utilisés pour attribuer les clusters finaux à chaque point.

Algorithm 7 Backtracking pour retrouver les clusters

$remainingPoints \leftarrow N - 1$
for $k = K - 1$ **downto** 1 **do**
    $minCost \leftarrow \infty$
    $bestSplit \leftarrow 0$
    for $split = k - 1$ to $remainingPoints - 1$ **do**
        $cost \leftarrow matrixDP[k-1][split] + \text{coût}(split + 1, remainingPoints)$
        if $cost < minCost$ **then**
            $minCost \leftarrow cost$
            $bestSplit \leftarrow split$
        end if
    end for
    Ajouter l'intervalle $[bestSplit + 1, remainingPoints]$ à la solution
    $remainingPoints \leftarrow bestSplit$
end for
Ajouter l'intervalle $[0, remainingPoints]$ à la solution
Trier les intervalles dans l'ordre croissant

Dans notre implémentation des ces différents algos, On a essayer de suivre au mieux ce qui a été présenter dans [3].

3.2.3 Ce qu'il faut retenir

Ce qu'il faut retenir c'est qu'on utilise deux matrices, 'matrixDP' pour calculer efficacement les coûts des clusters individuels et 'd' pour trouver la partition optimale en k clusters. avec un vecteur 'v' qui stocke temporairement les coûts optimaux (avec leurs médoïdes optimaux) pour des clusters de différentes longueurs se terminant à un index donné, servant d'information intermédiaire cruciale pour construire progressivement la solution dans la matrice 'd'.

3.2.4 Plus de vision sur l'implémentation générique

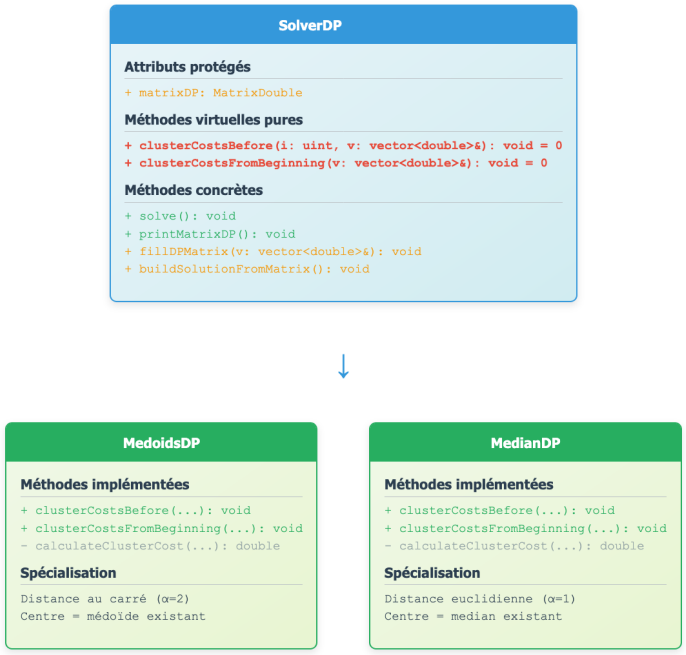


FIGURE 1 – UML

Cette UML illustre parfaitement comment l'**extensibilité** de cette architecture est remarquable : pour implémenter un nouvel algorithme de clustering, il suffit d'hériter de `SolverDP` et de spécialiser uniquement ces deux méthodes selon la métrique de distance souhaitée. Les implémentations concrètes `MedoidsDP` et `MedianDP` démontrent cette simplicité d'extension, différant uniquement par leur fonction de coût (distance au carré versus distance euclidienne) tout en réutilisant l'intégralité de l'infrastructure DP.

Cette conception respecte le **principe ouvert/fermé** de l'ingénierie logicielle : l'architecture est fermée à la modification (le code DP de base reste inchangé) mais ouverte à l'extension (nouveaux algorithmes facilement ajoutables), permettant une maintenance aisée et une évolutivité optimale du système. L'encapsulation des stratégies de calcul de coûts dans les classes dérivées garantit également une séparation claire des responsabilités, chaque algorithme gérant sa propre logique métier sans impacter les autres composants.

Cette approche modulaire facilite considérablement les tests unitaires, le débogage et l'ajout de nouvelles variantes d'algorithmes de clustering, constituant ainsi une base solide pour l'évolution future du système.

3.2.5 Parallélisation

Face à la complexité temporelle en $O(N^3)$ de notre algorithme de programmation dynamique, la parallélisation devient essentielle pour traiter efficacement les grandes instances de clustering. Notre architecture en C++ offre plusieurs opportunités de parallélisation que nous exploitons à différents niveaux de granularité.

Une implémentation avec OpenMP : Nous avons choisi OpenMP comme framework principal de parallélisation pour sa simplicité d'intégration et son efficacité sur architectures multi-cœurs. L'implémentation se fait par l'ajout de directives `#pragma omp` aux endroits stratégiques.

4 Conclusion

Ce projet a permis de développer avec succès une architecture générique et extensible pour la résolution optimale de problèmes de clustering par programmation dynamique, en s'appuyant sur les fondements théoriques établis par les références citées précédemment. L'objectif principal était de transposer fidèlement les algorithmes décrits dans la littérature scientifique en une implémentation logicielle correcte et performante, en suivant une démarche méthodologique rigoureuse : compréhension approfondie des différents algorithmes de clustering présentés dans les papiers, développement d'une abstraction générique, validation sur des petites instances, puis montée en charge vers des instances de grande taille.

Références

[1] Jain, A.K., Murty, M.N., & Flynn, P.J. (1999). Data clustering : a review. *ACM Computing Surveys*, 31(3), 264-323.

[2] Nicolas Dupin · Frank Nielsen · El-Ghazali Talbi (2020). k-medoids and p-median clustering are solvable in polynomial time for a 2d Pareto front.

[3] Nicolas Dupin, Frank Nielsen , and El-Ghazali Talbi. k-medoids clustering is solvable in polynomial time for a 2d Pareto front.