

Théorie de la complexité des algorithmes

Problème des K-centres

FRANÇOIS HERNANDEZ - LÉO PONS
CentraleSupélec
February 6, 2017

Contents

Introduction	3
I Implémentation	4
I.1 Objets utilisés	4
I.2 Génération d'instances	4
I.3 Affichage des données	5
I.4 Algorithmes de résolution	5
I.5 Conversion en fichier texte	6
II Théorie, NP-complétude et approximation	6
II.1 Problème d'ensemble dominant	6
II.2 Réduction	7
III Méthodes de résolution	8
III.1 Exact	8
III.2 DeuxApprox	8
III.3 Dominant	8
III.4 Descente	9
IV Évaluation des performances	10
IV.1 Comparaison d'algorithmes - Instances uniformes	10
IV.2 Comparaison d'algorithmes - Instances en 10 clusters	11
IV.3 Comparaison selon les types d'instances	12
IV.4 Performances de la Descente	13

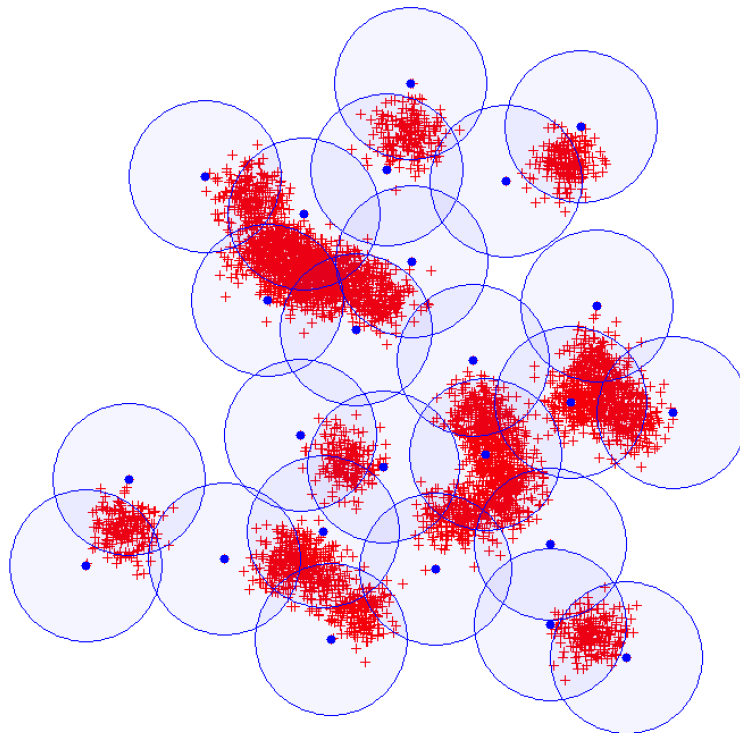
Introduction

Le problème des K-centres est un problème d'optimisation combinatoire. Le problème peut se décrire de façon informelle ainsi (Wikipedia) : étant donné n villes, il faut ouvrir une caserne de pompiers dans k villes, tel que la distance entre chaque ville et la plus proche caserne soit minimisée.

Il existe plusieurs définitions des problèmes K-centres. Ici, nous considérons la formulation suivante. Supposons un ensemble E de n points dans un espace vectoriel ou un graphe complet muni d'une fonction de distance satisfaisant l'inégalité triangulaire. Il s'agit de trouver k "centroids" afin de minimiser la distance maximale entre un point de E et le centre le plus proche.

Ce problème, dans sa forme problème de décision, est NP-complet.

Au cours de ce projet, nous avons mis en place un programme en Java constituant une représentation du problème, et proposant différents algorithmes de résolution sur différents ensembles d'instances.



I Implémentation

Nous avons choisi d'implémenter ce problème dans le langage Java. Nous avons mis en place des fonctions de génération et d'affichage d'instances (ensemble de points), ainsi que différents algorithmes de résolution à tester, et des méthodes de lecture et écriture des données sous format texte.

Le programme comporte ainsi les packages suivants :

- **Generation** : contient les différentes classes de génération d'instances ainsi que celle associée à la lecture des fichiers .txt, toutes héritant de la classe abstraite *Generateur* ;
- **Graphics** : contient les différentes classes associées à l'affichage des instances ;
- **Resolution** : contient les différents algorithmes de résolution sous forme de classes filles de la classe générique abstraite *Algo* ;
- **Main** : contient le main qui permet de lancer les algorithmes de test, ainsi que la définition des différents objets utilisés ;

I.1 Objets utilisés

Les objets utilisés dans ce programme sont définis de la façon suivante :

- **Point** : ensemble de coordonnées, ici x et y dans l'espace à deux dimensions ;
- **Instance** ensemble de n Points sous forme d'*ArrayList*, et un attribut k définissant le nombre de centres attendus, contient également une méthode permettant d'exporter l'instance au format .txt ;
- **Solution** : ensemble de k Points constituant les centres d'une Instance, ainsi qu'un attribut *rayon* définissant le rayon minimal pour lequel tous les points de l'instance sont couverts.

I.2 Génération d'instances

Nous avons choisi d'implémenter deux classes de génération d'instances :

- **Uniforme** : génère n Points de façon aléatoire dans l'espace défini (en conservant des marges afin de ne pas avoir de points trop proches des bords, pour des raisons d'affichage) ;
- **Cluster** : détermine n_{Cl} centres de clusters de façon aléatoire dans l'espace défini (en conservant une nouvelle fois des marges). On génère ensuite n Points répartis uniformément sur les différents clusters. Les coordonnées de chaque point sont déterminées aléatoirement selon une loi normale, centrée sur le cluster correspondant et d'écart-type σ . La taille des clusters est donc définie par σ et est commune à tous les clusters. Elle est définie à la création de l'objet.

I.3 Affichage des données

Afin d'avoir une représentation graphique claire des différentes instances et des résultats des algorithmes de résolution, nous avons choisi de définir une interface graphique à l'aide des outils Swing. Cela consiste en deux classes, Fenetre et Panel. Fenetre hérite de la classe JFrame et définit les caractéristiques de la fenêtre qui contiendra le Panel. Panel hérite de la classe JPanel et contient les méthodes associées au dessin des représentations. La méthode paintComponent dessine les différents éléments dans un bufferGraphics, attribut d'une image.

I.4 Algorithmes de résolution

Nous avons implémenté 4 algorithmes de résolution différents (*détails en partie III*) :

- Exact : Calcule exactement la solution optimale. L'algorithme teste naïvement toutes les solutions possibles récursivement et est donc très coûteux en temps.
- DeuxApprox (approximation) : On construit la solution centre par centre : on commence par un point quelconque puis on cherche parmi les points non encore choisis celui qui est le plus éloigné de notre ensemble de centres. On itère ainsi jusqu'à avoir choisi k centres.
- Dominant (approximation) : On commence par calculer la liste des distances entre tous les points, qu'on trie par ordre croissant. Ensuite, on prend la première distance et on cherche une solution avec des disques de rayons égaux à cette distance. Si on trouve, on renvoie la solution, sinon, on passe à la distance suivante.
- Descente : Cet algorithme est un peu particulier : il améliore une solution déjà existante (en testant des permutations de centres).

I.5 Conversion en fichier texte

Un standard d'export des instances a été défini afin de pouvoir sauvegarder et échanger différentes instances de test. Celui-ci est le suivant.

```
1 nombre d'instances i
3 n1, k1
  x1, y1, x2, y2, [...], xn1, yn1
5
  n2, k2
7 x1, y1, x2, y2, [...], xn2, yn2
9 [...]
11 ni, ki
   x1, y1, x2, y2, [...], xni, yni
```

La méthode `createFile` de la classe `Instance` permet de créer de tels fichiers à l'aide de la classe `FileWriter` de Java.

La classe `Importer` du package `Generation` permet de lire de tels fichiers à l'aide des classes `FileReader` et `BufferedReader` de Java.

II Théorie, NP-complétude et approximation

Le problème K-centre, dans sa forme de problème de décision, est NP-complet, c'est à dire qu'il suit les conditions suivantes :

- il est possible de vérifier une solution en temps polynomial ;
- il est possible de le réduire en un problème de la classe NP par une réduction polynomiale.

II.1 Problème d'ensemble dominant

Soit un graphe $G = (S, A)$. Un ensemble dominant pour G est un sous-ensemble D de l'ensemble des sommets S de G tel que tout sommet qui n'appartient pas au dominant soit adjacent à un de ses sommets.

Le problème d'ensemble dominant consiste à déterminer, selon G et un entier naturel k , si G possède un ensemble dominant d'au plus k sommets.

$dom(G)$ est la taille de l'ensemble dominant le plus petit possible pour le graphe G . Trouver un ensemble dominant de taille minimale est un problème NP-complet. Il est possible de réduire le problème K-centre en un problème d'ensemble dominant afin de démontrer sa NP-complétude.

II.2 Réduction

Soit un graphe non orienté $G = (S, A)$ pour le problème de l'ensemble dominant. Considérons le graphe $G' = (S, S \times S)$ muni de la fonction de poids $d : S \times S \rightarrow \mathbb{R}$ pour ses arrêtes.

$$d(u, v) = \begin{cases} 1 & \text{si } (u, v) \in A \\ 2\alpha & \text{sinon} \end{cases} \quad (1)$$

Supposons que G ait un ensemble dominant de taille inférieure ou égale à k .

$$dom(G) \leq k$$

Dans ce cas, il existe une solution de poids 1 au problème k-centre pour G' . Un algorithme d' α -approximation fournit une solution de poids inférieur ou égal à α , avec $\alpha \geq 1$ le facteur d'approximation.

S'il n'existe pas un tel ensemble dominant dans G , alors toutes les instances de k-centre auront un poids supérieur ou égal à 2α , donc supérieur à α .

Supposons qu'il existe un algorithme d' α -approximation pour le problème k-centre. Appliquons cet algorithme de décision sur G' . La solution a un poids inférieur ou égal à α , donc il existe un ensemble dominant de taille inférieure ou égale à k . Autrement, il n'existe pas de tel ensemble dominant. Il y a donc contradiction. Le problème k-centre peut se réduire au problème de l'ensemble dominant, donc il est NP-complet.

III Méthodes de résolution

III.1 Exact

Un premier algorithme est développé pour trouver la solution optimale d'une instance donnée, c'est à dire le choix de k centres qui minimise la distance maximale d'un point aux centres.

On procède par récursivité : On construit une fonction qui renvoie une solution optimale (minimisant le rayon) pour une instance donnée sachant qu'une liste de centres est déjà fixée et qu'il reste un certain nombre de centres à choisir. Cette fonction fonctionne en testant tour à tour le choix des points restants et en s'appelant récursivement avec un centre de moins à choisir. On peut donc sélectionner le meilleur point et utiliser le résultat de l'appel récursif pour les points suivant.

Notre solution principale est ainsi obtenue en appelant la fonction récursive avec le nombre de points à choisir original et avec une liste de points fixés vide. Cet algorithme est optimal mais très coûteux en terme de temps de calcul, ce qu'il le rend vite inutilisable.

III.2 DeuxApprox

Ce deuxième algorithme est une approximation qui permet de diminuer la complexité d'exécution de l'algorithme exact. On construit la solution centre par centre : on commence par un point quelconque puis on cherche parmi les points non encore choisis celui qui est le plus éloigné de notre ensemble de centres. On itère ainsi jusqu'à avoir choisi tous les centres.

Cet algorithme ne donne qu'un résultat approché (non optimal) mais est nettement moins gourmand en calcul, ce qui le rend largement plus utilisable que l'algorithme exact.

III.3 Dominant

Ce deuxième algorithme est une approximation, un peu plus exigeante que la précédente en termes de calcul mais donnant de meilleurs résultats. Il est basé sur la réduction au dominant qui a été présentée en partie II.

On commence par calculer la liste des distances entre tous les points, qu'on trie par ordre croissant. Ensuite, on prend la première distance et on cherche une solution : on construit notre liste centre par centre, en choisissant à chaque fois le point qui possède le plus de voisins non couverts (par des disques de cette distance). Après avoir choisi k centres, on regarde si tous les points sont couverts. Si oui, on renvoie la solution, sinon, on passe à la distance suivante.

III.4 Descente

Cet algorithme est un peu particulier : il améliore une solution déjà existante. Il faut donc commencer par choisir un algorithme qui déterminera la solution à manipuler, on pourra choisir notamment un des deux algorithmes approximatifs précédemment décrits.

Pour chaque centre dans la solution obtenue par le premier algorithme, on va essayer de le permuter tour à tour avec les autres points de l'instance. Si la permutation améliore la solution, on la retient et on recommence. On s'arrête quand plus aucun échange n'améliore la solution.

Cet algorithme est assez rapide et efficace, et il surtout toujours bon à appliquer à une solution obtenue par une autre heuristique.

IV Évaluation des performances

Dans la mesure ou la génération d'instance repose sur des processus aléatoires, les performances peuvent varier d'une instance à l'autre. En conséquence, toutes les mesures renseignées dans cette partie sont des moyennes de 10 relevés de résultats sur des instances différentes.

Afin de faciliter l'exploitation de ces mesures, nous avons mis en place un petit programme permettant d'exporter les tests effectués sur un certain nombre de valeurs du nombre de centres k et du nombre de points n dans des fichiers .csv. Ce code est présent dans le package Main dans la classe Export.java.

IV.1 Comparaison d'algorithmes - Instances uniformes

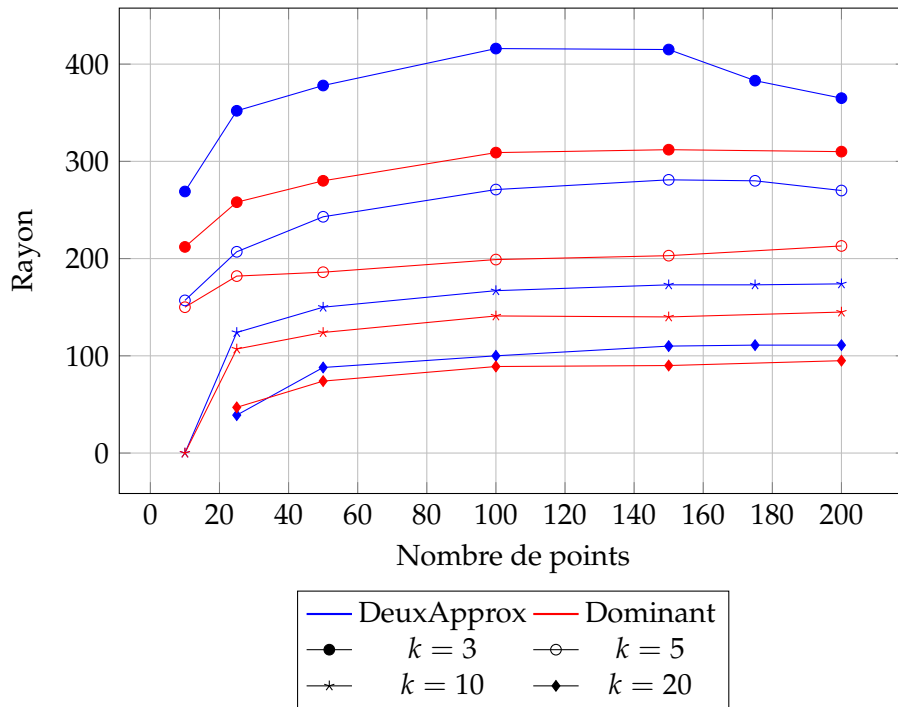


Figure 1: Rayon en fonction de n et k (Instances uniformes)

On observe que pour des instances uniformes, quelque soit le nombre de centres, le rayon augmente avec le nombre de points. Cela peut s'expliquer par le fait que pour des instances uniformes, plus le nombre de points est élevé, moins il y a de "vide" entre les points, et la taille des disques devra donc être plus importante afin de tout couvrir.

Par ailleurs, on remarque que l'algorithme Dominant permet d'avoir de bien meilleurs résultats que l'algorithme DeuxApprox, au détriment de la vitesse de calcul.

Enfin, on peut aussi noter que, de façon évidente, pour un nombre de points égal au nombre de centres, le rayon approximé est bien nul.

IV.2 Comparaison d'algorithmes - Instances en 10 clusters

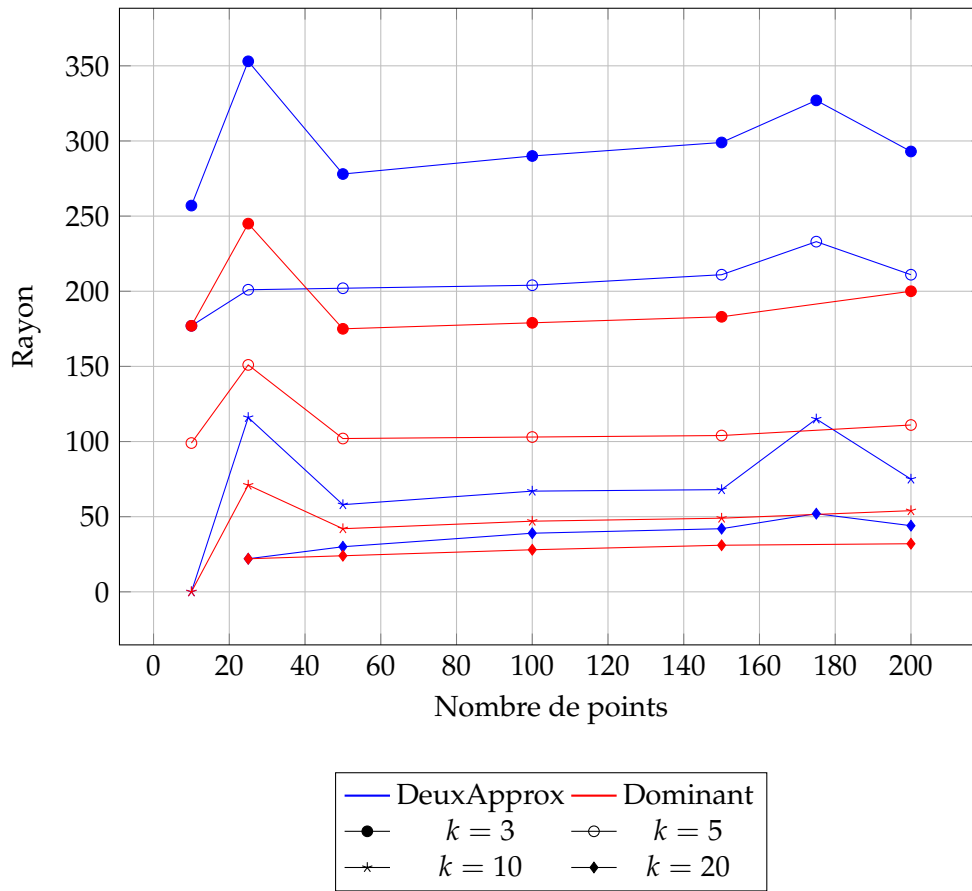


Figure 2: Rayon en fonction de n et du k (10 clusters)

Pour des instances contenant 10 clusters, on observe également une augmentation du rayon en fonction du nombre de points, mais plus légère que pour les instances uniformes. Cela est dû au fait que les points, même plus nombreux, sont concentrés autour de clusters d'une taille fixe.

De façon assez logique, plus le nombre de centre est grand, plus le rayon est faible. Effectivement, un nombre de centre plus grand permet de couvrir autant de surface pour un rayon plus faible, et donc de contenir le même nombre d'instances pour un rayon plus faible.

Par ailleurs, comme pour les instances uniformes, on remarque que l'algorithme Dominant permet d'avoir de bien meilleurs résultats que l'algorithme DeuxApprox, au détriment de la vitesse de calcul.

IV.3 Comparaison selon les types d'instances

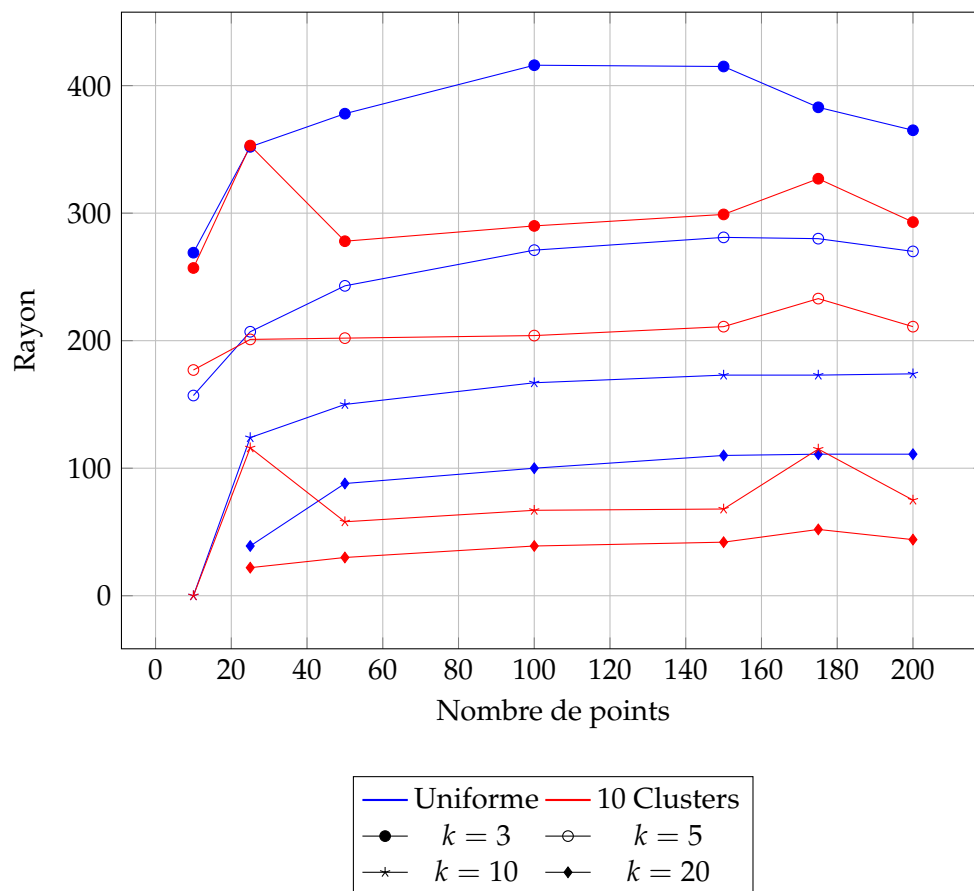


Figure 3: Rayon en fonction de n et k - Uniforme vs. Clusters

On observe que pour les instances uniformes le rayon est supérieur à celui pour les instances en clusters. Cela s'explique par le fait que pour ces dernières, les points sont regroupés en clusters, et donc par définition répartis de façon moins éparse dans l'espace.

IV.4 Performances de la Descente

L'algorithme Descente doit être évalué différemment. En effet, il est utilisé pour améliorer des résultats obtenus avec d'autres algorithmes. On peut donc essayer d'observer son pouvoir d'amélioration. Pour fonctionner, il faut donner à l'algorithme un minimum d'opportunités de permutation. On choisira donc des instances à $k=10$ centres. En figure 4 nous pouvons observer ses effets sur l'algorithme DeuxApprox : c'est assez efficace ! On diminue le rayon de près de 10, et de manière assez notablement constante selon le nombre de noeuds. Cependant sur Dominant, en figure 5, c'est moins efficace. L'algorithme est déjà très performant, c'est donc difficile d'améliorer encore les résultats (c'est même presque toujours impossible pour des nombres faibles de points).

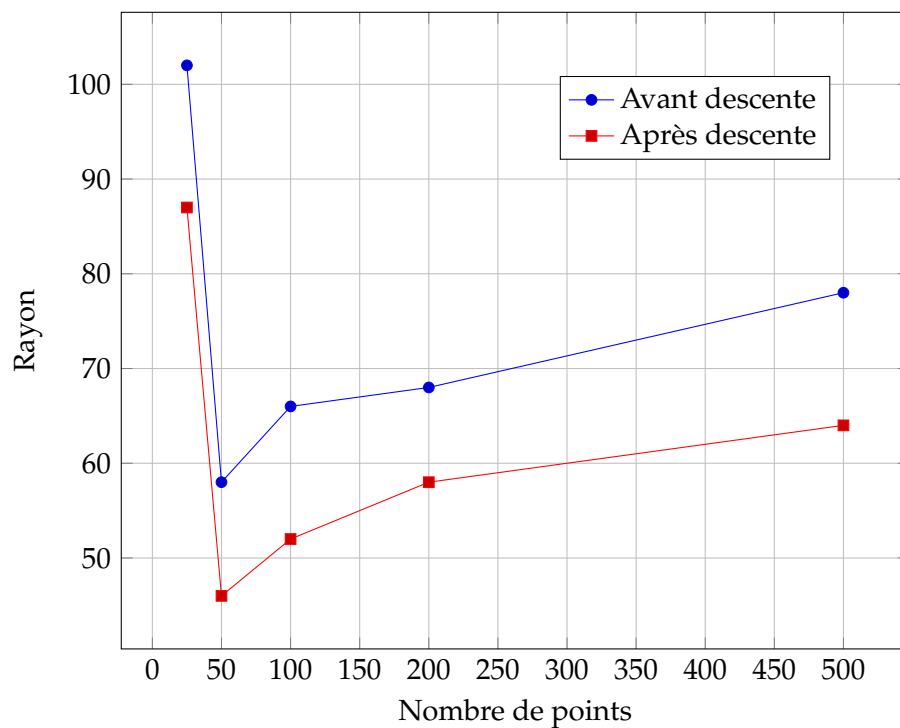


Figure 4: Performances de DeuxApprox avant et après utilisation de Descente.

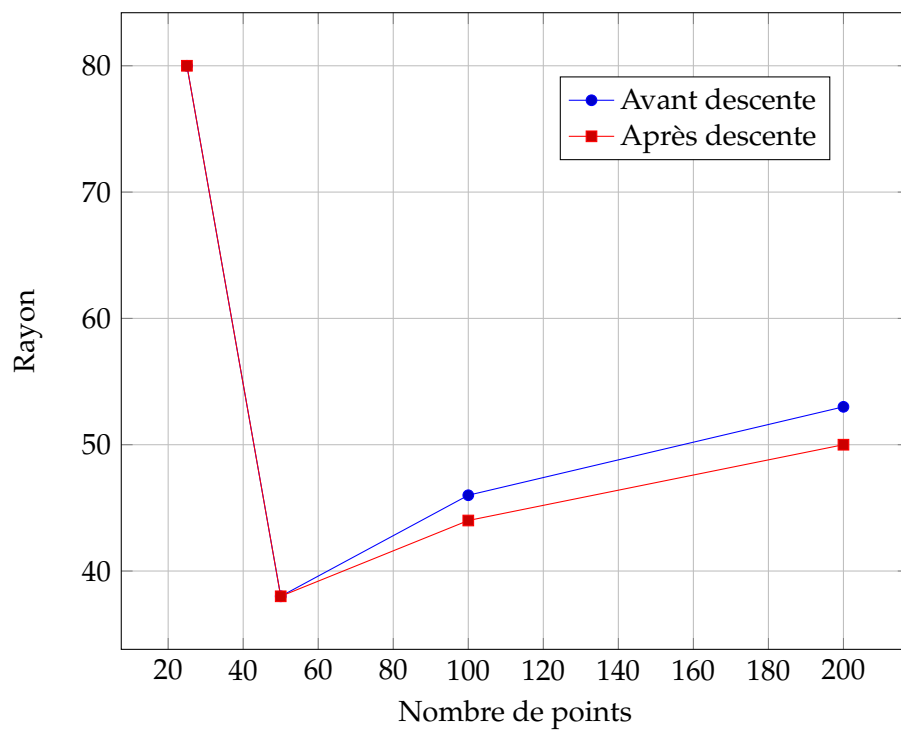


Figure 5: Performances de Dominant avant et après utilisation de Descente.