Rapport du mini-projet d'algorithmique avancée
Ligne brisée

Laszlo ABADIE - Samy CHAABI

22 avril 2025

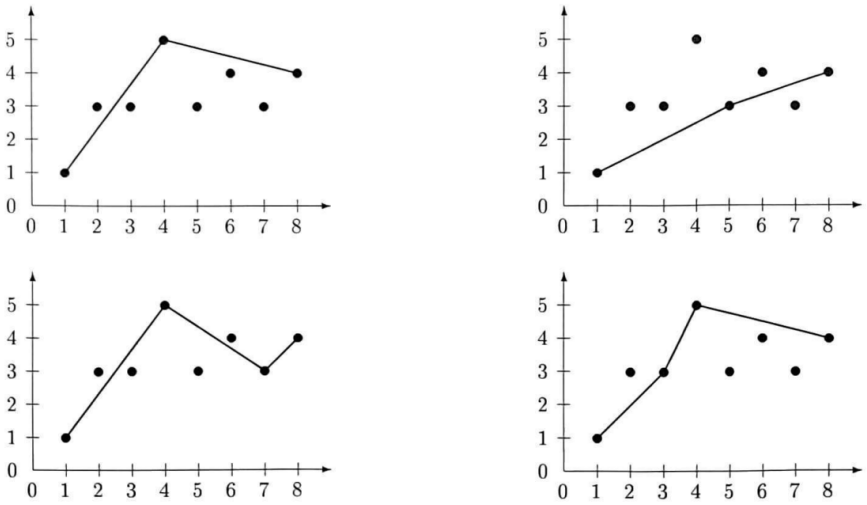

Table des matières

1 Introduction

1.1 Contexte

Soit S un ensemble de n points répartis sur un plan aux abscisses 1 à n et à des ordonnées quelconques. On cherche la meilleure approximation de cet ensemble par une ligne brisée passant par le premier et le dernier point. Pour ce faire, nous introduisons la notion de score qui se calcule de la manière suivante :

$$score = SD + m \times C$$

- **SD :** la somme des distances des points au segment dont les extrémités encadre le point
- **m :** le nombre de segments de la ligne brisée
- **C :** une constante multiplicative fixée arbitrairement

1.2 Question préliminaire

Il y a 2^{n-2} lignes brisées avec un ensemble S de n points, car le nombre d'ensembles d'ensembles possibles avec n points en prenant forcément le premier et le dernier est le nombre d'ensembles d'ensembles de $n-2$ points donc 2^{n-2}. Ainsi, on sait qu'un algorithme qui essaierait toutes les lignes brisées pour trouver la meilleure serait en complexité exponentielle par rapport au nombre de points de S à savoir $\Theta(2^n)$.

1.3 Aspect géométrique

Pour implémenter nos algorithmes, nous avons besoin de fonctions géométriques pour faire les différentes mesures nécessaires au calcul du score.

- **euclidian_distance(a, b) :** Retourne la distance euclidienne entre les points a et b.

```
fonction euclidian_distance(a, b)
    x1 ← a[0]
    y1 ← a[1]
    x2 ← b[0]
    y2 ← b[1]
    retourner racine_carrée((y2 - y1)^2 + (x2 - x1)^2)
fin fonction
```

- **orth_projection(c, a, b) :** Retourne la projection orthogonale du point c sur le segment $[a,b]$.

```
fonction orth_projection(c, a, b)
```

```
    xa ← a[0] ; ya ← a[1]
    xb ← b[0] ; yb ← b[1]
    xc ← c[0] ; yc ← c[1]

    xv ← xb - xa
    yv ← yb - ya

    facteur ← ((xc - xa) * xv + (yc - ya) * yv) / (xv^2 + yv^2)

    xh ← xa + facteur * xv
    yh ← ya + facteur * yv

    retourner [xh, yh]
fin fonction
```

De plus, nous avions le choix de l'implémentation, ainsi pour la fonction *distance*, nous avons choisi de garder la distance au projeté orthogonal même si ce dernier n'est pas sur le segment des points qui l'encadrent pour simplifier le code. Nous avons donc, dans certains cas particuliers des lignes brisées qui ne semblent pas optimales mais qui en réalité le sont en prenant en compte cette façon de calculer la distance. Sachant que la chance de tomber sur une telle configuration est minime, il n'est pas nécessaire de changer cette fonction, il suffit de régénérer un nouvel ensemble de points.

– **distance(i, j) :** Calcule la somme des distances des points de i à j au segment $[i,j]$.

```
fonction distance(i, j)
    somme ← 0
    A ← [i, S[i-1]]
    B ← [j, S[j-1]]

    pour k allant de i à j - 1 faire
        P ← [k, S[k-1]]
        H ← projection_orthogonale(P, A, B)
        somme ← somme + distance_euclidienne(P, H)
    fin pour

    retourner somme
fin fonction
```

2 Réalisation

2.1 Essais successifs

On envisage de calculer la meilleure approximation d'un ensemble S donné par une méthode à essais successifs de type solution à trous.

2.1.1 Question 1

Dans une procédure *appligbri* nous avons besoin de plusieurs valeurs et fonctions passées en paramètre :

- **i :** abscisse du point à traiter -1 (car tous les points sont décalés vers la gauche si l'on ne prend pas le premier – le dernier n'est pas pris non plus, mais cela n'a pas d'incidence).
- **X :** Liste de $n-2$ booléens (représentés par 0 ou 1 pour plus de concision)

```
X = [0, 1, 0, \dots, 1]
```

- S_i **:** Valeur binaire dans {0,1} représentant la sélection du point i.
- **satisfaisant**(x_i) **:** Teste si la configuration courante est acceptable.

```
fonction satisfaisant(x_i)
    retourner vrai
fin fonction
```

  (Toutes les configurations sont valides dans notre cas.)
- **enregistrer**(x_i) **:** Enregistre la décision prise pour le point i.

```
fonction enregistrer(x_i)
    X[i] ← x_i
fin fonction
```

- **soltrouvée :** Indique si une solution complète a été construite.

```
si i = n - 2 alors
    soltrouvee ← vrai
fin si
```

- **optimal :** Le coût total $SD + m \times C$ est minimal (par rapport à une solution connue).
- **optEncorePossible :** Vérifie s'il est encore pertinent d'explorer cette branche : on calcule le score qu'on aura si on ajoute le point i ce qui revient à retirer le segment entre $i-1$ et $i+1$ pour rajouter un segment entre $i-1$ et i et en rajouter un autre entre i et $i+1$.

```
currentScore ← SD + d(i-1, i) + d(i, i+1) - d(i-1, i+1) + (m+1) * C
si currentScore  currentOpt alors
    optEncorePossible ← faux
```

```
    fin si
```

– **défaire**(x_i) **:** Annule la décision prise pour x_i.

```
fonction defaire(x_i)
    X[x_i] ← 0
fin fonction
```

2.1.2 Question 2

Nous avons une complexité au pire cas : $O(2^n)$, cependant, avec la fonction de distance, nous parcourons en plus tous les points, donc la complexité finale est de $O(n2^n)$. Nous avons donc bien une complexité exponentielle.

2.1.3 Question 3

Le but de *optEncorePossible* est d'élaguer les possibilités pour limiter les appels récursifs. En effet, si une branche a, à un certain point, un score supérieur (ou égal) à un score optimal trouvé précédemment, il n'est pas dans notre intérêt de continuer à perdre du temps à explorer cette branche, car tous les scores trouvés à la fin seront forcément supérieurs au score optimal. Nous introduisons donc une simple condition d'élagage :

```
fonction optEncorePossible()
    retourner (score < optScore)
fin fonction
```

2.1.4 Question 4

On décide de calculer le coût de l'approximation particulière suivante : on joint tous les points impairs (ainsi que n si ce dernier est pair). Ainsi, en calculant le score de cette configuration particulière, nous pourrons commencer l'exécution de notre procédure d'essais successifs avec un score plus intéressant. En effet, en prenant la moitié des points, on se rapproche plus d'un score "moyen" et nous n'aurons pas besoin de passer par les branches où on ne prend qu'un seul point qui, généralement, a un score trop grand. Ainsi, en commençant par un score optimal initialisé au score d'une configuration où l'on prend la moitié des points, nous aurons, en moyenne, grâce à notre condition d'élagage, moins de branches parcourues.

2.1.5 Stratégie

Pour résoudre notre problème de recherche de la meilleure approximation d'un ensemble de points, nous avons appliqué la méthode à trous par essais successifs (backtracking). Cette stratégie consiste à construire progressivement une solution partielle en laissant volontairement certains emplacements où l'on "ne

fait rien" (appelés "trous") dans la configuration. À chaque étape, nous explorons les différentes possibilités pour combler ces trous, en évaluant systématiquement chaque configuration intermédiaire à l'aide de la fonction de score définie. L'algorithme élimine au fur et à mesure les pistes qui ne permettent pas d'améliorer ce score grâce à notre condition d'élagage, concentrant les recherches sur les solutions les plus prometteuses. Cette approche s'apparente à une recherche par backtracking, guidée par une évaluation heuristique permettant de limiter l'exploration à un sous-ensemble pertinent des possibilités.

2.2 Programmation dynamique

Nous allons ici détailler le fonctionnement de notre recherche de solution suivant les principes de la programmation dynamique.

2.2.1 Question 1

On définit la fonction $approx - opt(i,j)$ fournissant la valeur de l'approximation optimale entre le point d'abscisse i et le point d'abscisse j. On note de plus $SD_{i,j}$ sa somme des distances euclidiennes des points (i + 1) à (j - 1) au segment joignant les points i et j.

On propose la formule de récurrence suivante permettant de calculer $approx-opt(1,n)$:

$$\begin{cases} \forall i \in \{1,...,n\}, \text{ approx-opt}(i,i) = 0 \\ \text{approx-opt}(1,n) = \min\limits_{j=i+1}^{n} (SD_{i,j} + C + \text{approx-opt}(j,n)) \end{cases}$$

2.2.2 Question 2

Structure tabulaire

Dans notre programme, on utilisera une liste `approx_opt` à 1 dimension telle que `approx_opt[i]` prenne la valeur de approx-opt(i,n).
On utilise également une liste `next` telle que, pour un point d'abscisse i, `next[i]` contient l'abscisse du point auquel il faudra relier i pour poursuivre la ligne brisée de manière optimale [1].

Stratégie de remplissage

Pour remplir nos tableaux, nous utilisons la logique suivante :

- On remplit notre tableau approx-opt entièrement de $+\infty$.
- On utilise notre cas de base pour écrire `approx_opt[n] = 0`.
- On utilise un indice i parcourant la liste à l'envers à partir de l'avant dernière valeur jusqu'à la première, et un indice j qui parcourt à chaque boucle de i parcourt la liste en partant de i+1 jusqu'à la fin de la liste. Pour chaque itération de j, on calcule :

$$\text{cost} = SD_{i,j} + C + \text{approx-opt}(j,n)$$

- On compare ensuite notre cost avec `approx_opt[i]`. S'il s'avère que l'on trouve un coût inférieur à la valeur présente dans notre tableau, alors on remplace `approx_opt[i]` par cost, et on inscrit la valeur de j dans `next[i]`. De ce fait, on sait que si notre ligne brisée passe par le point i, alors le point j et le point suivant i qui permettra de minimiser le plus possible le coût de la solution.

1. Pour des raisons de simplicité d'implémentation, la liste next associe un "point suivant" à tous les points de notre problème, y compris ceux qui ne font pas partie de notre ligne brisée.

2.2.3 Question 3

Complexité temporelle
Pendant l'exécution du programme, on parcourt notre liste `approx_opt` en sens inverse avec l'indice i, et à chaque itération sur i, on le parcourt $n - i$ fois avec l'indice j. On a donc une complexité temporelle de la forme :

$$\begin{aligned}
&\sum_{i=1}^{n}(n-i) \\
&= \sum_{k=0}^{n-1} k \\
&= \frac{(n-1)n}{2} \\
&= \Theta(n^2)
\end{aligned}$$

Complexité spatiale Ajouter un point à notre problème ajoute un point à nos listes `approx_opt` et `next`. La complexité spatiale de notre solution est en $\Theta(n)$.

2.3 Jeux de test

Nombre de points	backtracking	programmation dynamique	s
10	$<$ 1 ms	$<$ 1 ms	
20	0,2 s	$<$ 1 ms	
25	30 s	$<$ 5 ms	
30	$>$ 2 min	$<$ 5 ms	
50	-	20 ms	
100	-	20 ms	
200	-	1,7 s	
500	-	27,4 s	
1000	-	3,7 min	

2.4 Questions complémentaires

Concernant la conception de ces deux algorithmes, la compréhension et la réalisation des essais successifs sont plus simples et intuitives, mais comme le montrent les tests réalisés, la programmation dynamique est beaucoup plus efficace en raison de sa complexité quadratique.

Avec la méthode des essais successifs, on peut aller jusqu'à environ 30 points avant de dépasser les 2 minutes alors qu'on peut arriver à 820 points en programmation dynamique.

Il est tout à fait possible de résoudre le problème à l'aide d'une méthode de diviser pour régner mais cela n'aurait pas grand intérêt étant donné que la

complexité reste la même que celle des essais successifs. En effet, au lieu de parcourir tous nos points de manière linéaire, on commence par celui du milieu, puis on crée deux nouvelles instances du problème avec chacun la moitié des points. On parcourt donc toujours tous les points juste dans un ordre différent. La complexité reste donc exponentielle.

3 Conclusion

Au cours de ce projet, nous avons pu expérimenter la recherche d'une solution optimale à un problème en utilisant les méthodes des essais successifs et de la programmation dynamique. L'objectif était de comparer l'efficacité de ces méthode sur un même problème.

Lors des tests temporels, nous avons observé de bien meilleures performances en moyennes pour la programmation dynamique. Cela c'est logique, lorsqu'on regarde les complexités temporelles estimées théoriquement.

On peut donc en déduire que la recherche d'une solution par programmation dynamique reste à privilégier, bien qu'elle requiert plus d'effort de conception que des essais successifs. Ces derniers sont à utiliser en dernier recours, si aucune autre méthode ne fonctionne.