Partitionnement de droites

Projet FIG : Partitionnement de droites

- Enoncé
- Résolutions
  - Algorithmes Naïfs
  - Tentatives non naïves
  - Algorithme aléatoire
- Conclusion
- Installer et lancer le projet

Enoncé

Sujet 5 : partitionnement de droites

Étant donnée une liste de droites dans l'espace, trouvez un sous-ensemble de 10 droites qui minimise la distance vers toutes les autres.

Formellement, soit $\mathcal{D} = \{D_1, \ldots, D_n\}$ l'ensemble de droites et $\mathcal{D}' \subset \mathcal{D}, |\mathcal{D}'| = 10$ le sous-ensemble. La distance entre deux droites est $d(D_i, D_j)$. La distance d'une droite D_i vers le sous-ensemble de triangles est $d(D_i, \mathcal{D}') = \min\{d(D_i, D_j) : D_j \in \mathcal{D}', D_j \neq D_i\}$, c'est-à-dire la distance vers la droite la plus proche. Nous cherchons a minimiser $f(\mathcal{D}') = \max\{d(D_i, \mathcal{D}') : D_i \in \mathcal{D}\}$. Il s'agit donc de trouver 10 droites qui soient proches de toutes les autres.

- La distance entre deux droites est zéro seulement si elles s'intersectent.
- Données d'entrée : sujet5.instance.zip. Chaque droite est définie par un point et un vecteur directeur.
- Données de sortie : distance ; liste de droites. Chaque droite est définie par sa position dans le fichier d'entrée. Exemple :

```
{
  "distance": 12.8,
  "droites": [1, 4, 6, 13]
}
```

Figure 1: image

Résolutions

On présente ici les différentes pistes explorées pour la résolution du problème. La solution retenue est l'algorithme aléatoire.

Toutes les solutions proposées sont gérénalisables pour prendre un sous-ensemble de K droites de l'ensemble de N. On aura par défault $K = 10$ et $N = 100000$ (d'après la taille du fichier json).

Algorithmes Naïfs

Algorithme naïf basique L'algorithme va tester toutes les combinaisons possibles de K droites parmi N droites et calculer à la volée toutes les distances des droites à ce sous-ensemble, pour au final afficher quelle combinaison a eu le meilleur score (*ie* la distance la plus faible aux restes des droites).

Résultats: Cette méthode est impraticable pour le sujet initial. La complexité est en $O(n\binom{n}{k})$ et il faudrait des années pour savoir quel est le meilleur ensemble. Juste pour $K = 10$ et $N = 23$, il lui faut environ 30s. Conseil: éviter d'attendre la fin de l'algorithme s'il existe trop de combinaisons.

Algorithme naïf - Amélioré (tableau des distances) L'algorithme va tester toutes les combinaisons possibles de K droites parmi N droites mais au lieu de re-calculer à chaque fois les distances à la volée, va remplir un tableau $N \times N$ contenant les distances d'une droite à l'autre.

Résultats: Cette méthode est bien plus rapide que la précédente mais reste impraticable pour le sujet initial car stocker un tel tableau de `double` demanderait de bien trop grandes ressources. Cet algorithme est donc préférable au précédent pour des N raisonnables.

Algorithme naïf - Optimisation Si on reprend le premier algorithme naïf, on peut constater que calculer le maximum des minima pour ensuite prendre le minimum de ces maxima, peut se révéler être redondant. En effet, une amélioration est de passer à la combinaison suivante dès lors qu'une distance d'une droite au sous-ensemble actuel est supérieure au minimal actuel (*ie* la distance pour la meilleure combinaison actuelle), car le max sera forcément supérieure à cette distance donc la combinaison ne sera pas retenue. Cela permet donc de sauter rapidement des combinaisons et donc de réduire la complexité. Cependant il faut toujours tester toutes les combinaisons possibles…

Résultats: Bien qu'optimisé, il faudrait facilement plusieurs trillions d'années pour parcourir toutes les combinaisons (*ie* $\binom{100000}{10} = 2.10^{43}$ combinaisons).

Tentatives non naïves

Nous avons donc réfléchi à des méthodes mathématiques de résolution afin de ne pas à avoir à parcourir toutes les combinaisons pour avoir la solution optimale.

Un énoncé proche Voici l'idée derrière l'algorithme: Imaginons qu'on ait un tableau des distances (plus simple pour la visualisation). On parcourt chaque colonne (*ie* chaque droite) et on récupère les K plus grandes distances. On compare ensuite la K-ème plus grande valeur de chaque colonne et on prend la colonne ayant la valeur la plus faible. La droite associée à cette colonne est la première dans la combinaison et les $K - 1$ autres sont les droites les plus loins de cette droite (*ie* les $K - 1$ plus grandes valeurs de sa colonne). L'algorithme

est en $O(n^2)$ car il suffit de parcourir le pseudo tableau des distances une seule fois pour avoir les droites qui composent le meilleur sous-ensemble.

Résultats: L'algorithme donne une réponse très rapidement (moins d'une minute, voire en quelques secondes avec optimisation). Cependant, cet algorithme ne donne pas tout à fait le résultat attendu. En effet, les droites dans le sous-ensemble ne sont pas forcément proches, ainsi, cet algorithme répond seulement à l'énoncé où on minimise $g(D') = max\{d(D_i, D'), D_i \in D/D'\}$.

Une preuve par l'absurde peut se faire pour démontrer que cette combinaison est la meilleure (laissée au lecteur). Et la distance donnée par cet algorithme est loin d'être la meilleure (environ 6)… En parcourant au hasard quelques combinaisons on trouve très facilement mieux.

Division de l'espace On réalise qu'un sous-ensemble de 10 droites est un nombre arbitraire (surtout au vu du nombre important de droites). On commence donc évaluer les cas $K = 1$, $K = 2$, … pour en déduire une méthode de calcul qui n'est pas combinatoire. Le principe repose ici sur *construire la meilleure combinaison*, en faisant une analogie avec un nuage de points.

Cas $K = 1$ Cela revient à trouver, parmi toutes les droites, la droite dont la distance maximale est la plus petite. Pour simplifier la visualisation, on suppose qu'on dispose d'un tableau stockant toutes les distances calculées. Pour chaque droite (donc chaque ligne du tableau), on ne garde que la valeur max. À chaque droite est associée sa valeur max. Le sous-ensemble correspond à la droite ayant la plus petite valeur max.

Nommons cette valeur d_1. Intuitivement, on voit ici que le sous-ensemble (composée d'une seule droite) est le "centre" de toutes les autres droites: il ne peut pas s'agir d'une droite plus excentrée, sinon sa distance max serait plus grande.

On conjecture que $d1 \geq d2 \geq d3$ …

Cas $K = 2$ On cherche d_2. Pour un sous-ensemble de deux droites donné, la distance d'une droite au sous-ensemble correspond à la plus petite distance entre cette droite et les deux droites du sous-ensemble. Parmi toutes les distances calculées, on prend la valeur max : c'est la distance au sous-ensemble. d_2 correspond à la plus petite de ces distances.

Intuitivement, on voit que le sous-ensemble constituent deux "centres". Notons D_1 et D_2 les deux droites du sous-ensemble final. Une partie des droites de D aura une distance minimale avec D_1 et une autre partie aura une distance minimale avec D_2. L'espace est donc partitionné en deux, ce qui signifie qu'on peut se ramener au cas $K = 1$ suivant cette partition de l'espace. Comment réaliser cette partition ?

On part de la droite trouvée au cas précédent, et on se place sur la droite la plus éloignée de celle-ci. La première partition correspond à toutes les droites

situées à moins de d_1 de cette droite; la deuxième partition regroupe toutes les autres. Chaque nouveau "centre" D_1 et D_2 sera inclut dans chaque partition et correpond au "centre" de chaque partition.

Le sous-ensemble final correspond donc à la réunion des deux droites trouvées dans chaque sous-ensemble. On dispose maintenant de d_2 (qui est en fait la plus petite distance des deux "sous-sous-ensembles" : pas besoin de recalculer).

Cas $K = 3$ On suit la même méthode: on se place sur la droite la plus éloignée d'un des deux centres. La première partition correpond aux droites de distances inférieures à d_2. On fait de même pour l'autre centre: on obtient la deuxième partition. Les autres droites qui ne sont pas dans une des deux partitions constituent la troisième partition.

> *Il faudrait démontrer qu'utiliser d_1 fournit deux partition, utiliser d_2 fournit trois partitions, etc.*

Pour chaque partition, on se ramène au cas $K = 1$.

Le sous-ensemble final est la réunion des trois droites trouvées.

Cas général On discerne déjà l'algorithme itératif utilisée:

- À chaque incrémentation, on dispose d'une distance d_k
- On calcule les $k + 1$ partitions à l'incrémentation d'après grâce à d_k pour se ramener à k problèmes de type $K = 1$. On dispose maintenant de d_{k+1}
- On continue jusqu'à atteindre le nombre d'éléments dans le sous-enseble souhaité

Complexité À chaque itération, on teste tous les distances une fois, pour trouver le centre dans chaque partition, donc complexité en $O(Kn^2)$.

Algorithme récursif On pourrait améliorer la rapidité de l'algorithme récursivement: au lieu d'incrémenter, en divisant toujours la partition en deux:

- Cas $K = 1$: on se retrouve avec deux partitions
- Cas $K = 2$: on trouve le "centre" de chaque partition et on l'utilise pour diviser chaque partition en deux
- etc

Inconvient: on obtient des sous-ensembles avec un nombre d'éléments correpondant à une puissance de 2 uniquement. Comme on cherche un sous-ensemble avec 10 éléments, cette méthode ne convient pas.

Résultats Malheureusement, tout repose sur une analogie avec des points. Cependant la distance manipulée ici (distance entre deux droites), n'est pas une distance…

En effet une distance est une application d de $E \times E$ dans R_+ vérifiant:

- $d(x, y) = d(y, x)$ *(symétrie)*
- $d(x, y) = 0 \iff x = y$
- $d(x, z) \leq d(x, y) + d(y, z)$ *(inégalité triangulaire)*

Hors ici, la "distance" entre deux droites ne vérifient pas les propriétés:

- 2) car deux droites distinctes s'intersectant donnent une distance nulle
- 3) car si on prend les droites x et z qui ne s'intersectent pas ($d(x, z) > 0$), et qu'on prend une droite y qui intersecte à la fois x et z alors on aurait $d(x, y) + d(y, z) = 0$ et donc on a contradiction de l'inégalité triangulaire.

Nous avons donc choisi de ne pas implémenter cette méthode.

Inclusion des solutions Au lieu de diviser l'espace pour construire la meilleure combinaison, nous sommes partis du principe que l'espace des solutions est croissant avec K. Cela signifie que pour trouver le sous-ensemble de taille $K + 1$, il suffit de trouver la bonne droite à ajouter au sous-ensemble optimal de taille K.

On cherche donc la première droite D_1 (cas $K = 1$). Cela se fait en complexité $O(n^2)$ car on a juste à parcourir le tableau des distances. On dispose de d_1.

Concrètement, cette droite est la "meilleure" droite: n'importe quelle autre droite augmente la distance au sous-ensemble. Il faut donc conserver cette droite et ajouter des droites pour améliorer le résultat.

Trouver la meilleure droite à ajouter se fait au pire en $O(n^3)$ si on teste toutes les droites. Et on répète jusqu'à avoir un sous-ensemble de taille K.

Résultats: Encore malheureusement, il n'y a pas inclusion entre les espaces de solution, ce qui rend faux la construction de cette "solution optimale".

> Par ailleurs, l'algorithme peut vite gagner en complexité car en réalité, il peut y avoir plusieurs solutions optimales à chaque K, il faut donc toutes les garder en mémoire pour tester toutes les combinaisons, et donc la complexité peut exploser en 2^K.

Algorithme aléatoire

Au lieu de chercher la solution optimale à ce problème d'optimisation combinoire, en parcourant les combinaisons dans l'ordre, on peut parcourir les combinaisons dans un désordre aléatoire et retenir la meilleure solution au fil du temps.

En effet, le problème de parcourir les combinaisons dans l'ordre est que les mêmes droites sont toujours testés (il y a 10^{39} combinaisons qui commencent avec la droite 0).

Parcourir aléatoirement toutes les combinaisons permet de laisser une chance à n'importe quelle combinaison.

Résultats: Cette méthode permet facilement d'obtenir de meilleurs résultats rapidement que les méthodes précédentes. Cependant, elle reste aussi lente que les algorithmes naïfs. On effleure une minime partie des combinaisons donc tout repose sur l'aléatoire de trouver une des meilleures solutions.

Conclusion

Après avoir pris conscience qu'il serait vain de trouver la solution optimale, nous avons décidé de faire tourner l'algorithme aléatoire (optimisé pour les calculs et déterminer si la combinaison est meilleure) pendant plusieurs jours.

L'algorithme aléatoire a été modifié de tel sorte à s'arrêter au bout d'un certain temps ou d'un certain nombre d'itérations (le temps max et le nombre d'itération peuvent être infinis cependant, si on souhaite que cela ne s'arrête jamais).

L'algorithme commence avec la combinaison trouvé dans `résultat.json` (la meilleure combinaison actuelle). Et si jamais il trouve une meilleure combinaison, il modifie ce fichier pour y stocker la nouvelle meilleure combinaison.

Résultat final:

```
"distance": 1,
"droites": [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Installer et lancer le projet

Cloner le projet puis

```
cmake .
make
```

Vous obtiendrez les *executables* `main` et `verif`.

`.\main` permet de lancer le programme principal, vous pouvez modifier à votre guise la fonction *main* pour lancer les algorithmes souhaités et régler le temps maximal / nombre d'itérations maximal pour la recherche aléatoire.

`.\verif` permet de vérifier que le fichier `resultat.json` est correct (*ie* la distance correspond bien à la distance du sous-ensemble donné dans "droites").

Utiliser le repo À cause des nombreuses formules utilisées, on utilise des formules LaTeX dans le markdown. Pour la compilation du fichier `README.md`, on utilise `readme2tex`. Il faut éditer `README_latex.md` et compiler avec:

```
python -m readme2tex --output README.md INPUT.md
```

On peut éventuellement exporter en pdf avec `pandoc`:

```
pandoc ./README_latex.md --pdf-engine=xelatex -o rendu.pdf
```