Rapport de Projet

Algorithmique et structures de données - 3MMALGO

DIAB Dana, HAYDAR Anass

6 Mai 2023

1 - Introduction

Le but de ce projet est d'élaborer un algorithme qui, à partir d'une distance d et un ensemble de point en 2 dimensions, renvoie les composantes connexes de ce graphe tel que chaque deux points qui sont à une distance inférieure ou égale à d sont reliés. Pour aboutir à notre objectif, nous avons mis en place plusieurs algorithmes qui diffèrent dans les structures de données utilisées et les coûts temporelset spatiaux .

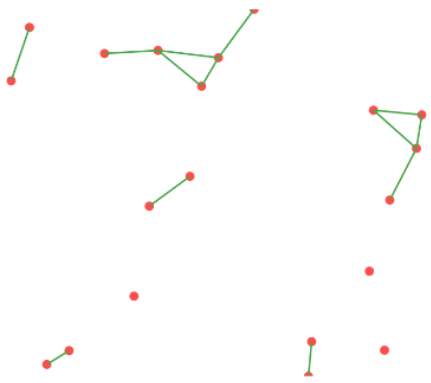


Figure 1: Composantes connexe d'un graphe

2 Présentation des méthodes utilisées :

Pour trouver les composantes connexes d'un graphes, nous avons essayé deux méthodes:

2.1 1ère Approche : Méthode Naïve

Pour commencer, nous avions voulu avoir un premier programme fonctionnel, retournant les résultats attendus, sans prenant en compte des contraintes temporelles et spatiales. Ensuite, après avoir testé notre programme en locale et avec les tests mises en place par le professeur, nous avons essayé d'optimiser la fonction, en supprimant les variables inutiles, en choisissant les structures de données convenables, etc.

Cette méthode consiste à parcourir l'ensemble des points et pour chaque point *point* non visité, on l'ajoute à la list des points à visiter *to_visit*. Tant que *to_visit* n'est pas vide, on extrait le dernier point *curr* de cette list , i.e pop(), s'il n'est pas visité, on l'ajoute au set *visited* et à la list *composante* qui va contenir les points qui appartiennet à la même composante connexe. On va ensuite itérer une deuxième fois sur l'ensemble des points, en ajoutant à *to_visit*, ceux qui sont à une distance maximale d de *curr* et qui n'ont pas été visités. Une fois *to_visit* est vide, on ajoute *composante* à *composantes*, qui est une list contenant l'ensemble des composantes connexes. Enfin, on itère sur *composantes* afin de calculer les tailles des composantes connexes, en les triant avec la méthode sort().

Soit n le nombre de points.

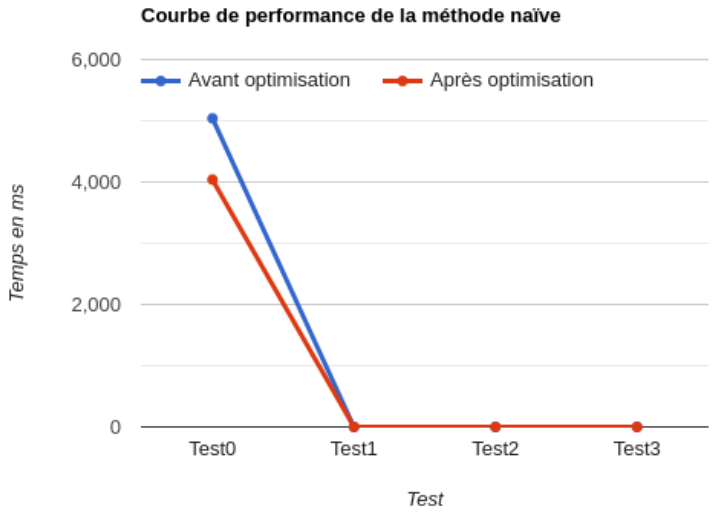
Les structures de données utilisées pour l'implémentation de cet algorithme sont donc,

- Visited : Un set dont le coût spatial est de O(n). On a choisi un Set puisqu'on fait plusieurs tests d'appartenance, et un test est en O(1) en moyenne et de O(n) au pire des cas. Alors que pour une list, ce test est de O(n) en moyenne. De plus, dans un set, il n'y a pas de doublons ce qui évite de faire des tests d'appartenance dans des ensembles plus grand. Ainsi, avec un set au lieu d'une list, on gagne du temps et de l'espace.
- Composantes : List de list, dont le coût spatial est de O(n). La seule action qui est faite sur cette variable est l'append() qui est en O(1) (coût amorti), et, on est sûr que composantes ne va pas contenir de doublons. Donc, l'utilsation d'une list ou d'un set ne va rien changer dans ce cas.

- To_visit : List, dont le coût spatial dans le pire des cas est de O(n), mais sera généralement plus petit. Dans le cas de cette variable, on est pas sûr de l'inexistence de doublons. L'utilisation d'un set nous aurait permis de faire la deuxième itération sur moins de points. Mais, comme on utilise la fonction pop(), qui est de O(1) pour une list et de O(n) au pire des cas pour un set, nous avons préféré l'utilisation d'une list.

Ainsi, le coût spatial de cette méthode naïve est de O(n).

En ce qui concerne la compléxité temporelle de cet algorithme, l'existence de deux boucles imbriquées qui itèrent sur la liste des points, fait que la compléxité temporelle dans le pire des cas soit de $O(n^2)$. Cependant, en pratique, la présence de la variable *visited* réduit la compléxité temporelle puisqu'on itère une deuxième fois que si le point n'appartient pas à *visited*. L'optimisation faite nous a permis de gagner un peu de temps pour le Test0 mais l'algorithme plante toujours pour les autres tests.



La fonction implémentant cet algorithme se trouve aussi dans connectes.py.

2.2 2ème Approche : Les KD-Tree

Après avoir réussi seulement un seul des tests du prof avec la méthode naïve, nous avons chercher à obtenir davantage d'informations sur le sujet en question, on a ainsi découvert les K-D Tree.

C'est une structure de données de partition de l'espace permettant de stocker des points, et de faire des recherches (plus proche voisin, etc.) plus rapidement qu'en parcourant linéairement le tableau de points. Les arbres k-d sont des arbres binaires, dans lesquels chaque nœud contient un point en dimension k. Chaque nœud non terminal divise l'espace en deux demi-espaces. Les points situés dans chacun des deux demi-espaces sont stockés dans les branches gauche et droite du nœud courant.

Cette structure de donnée est principalement utilisée pour résoudre des problèmes liés à l'algorithme des K plus proches voisins (kNN en anglais).

La compléxité temporelle de l'algorithme des kNN en utilisant les K-D Tree, en prenant k=n et d=2 dans notre cas, est de O(nlog(n)). La complexité spatiale, quant à elle, est de $O(n^2)$ en moyenne, néanmoins, en pratique, elle est de O(nlog(n)). Ces deux complexités peuvent certainement être considérées comme attractives par rapport à ceux de la méthode naïve.

Malheuresement, on s'est rendu compte après avoir commencer le travail, que cette méthode pourrait nous coûter cher quant à la précision de nos résultats. Cet algorithme peut considérer un nuage de points comme étant une composante connexe alors qu'il ne l'est pas, ou, au contraire, peut ne pas prendre en compte une composante connexe dans le résultat retourné. Ce problème de précision nous a forcé à laisser tomber cette méthode.

2.3 3ème Approche : Une Méthode Performante

Cette algorithme consiste à trouver les potentiels voisins de chaque point et puis verifier s'ils sont vraiment voisins, au lieu de parcourir tous les points. Pour cela, on divise l'espace en sous espaces , en utilisant des quadrants, sous forme de carrés dont la taille des cotés est la distance d et pour chaque point on parcourt juste les points qui se trouvent dans le même quadrant et ceux qui se trouvent dans les quadrants adjacents.

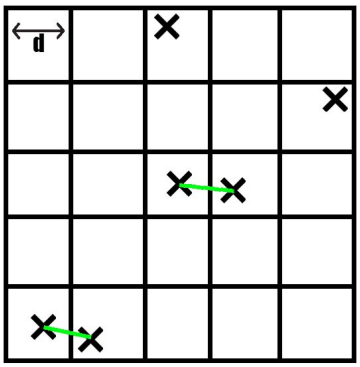


Figure 2: division de l'espace en carrés de cotês d

En essayant d'implémenter cet algorithme, on s'est rendu compte qu'on peut utiliser le fait que si on trouve un voison *point1* d'un point *point0*, on est sûr que tous les points appartenant au cercle de centre *point1* et de rayon d appartiennent à la même composante connexe du point *point0*.

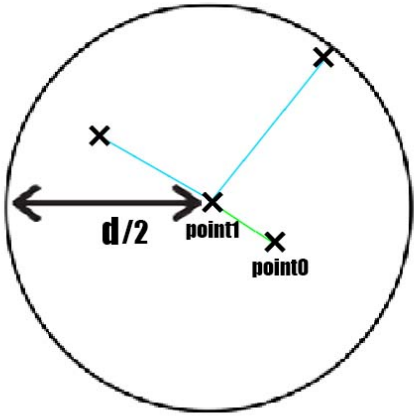


Figure 3:

Malheureusement, diviser l'espace en cercles est quasi-impossible donc on a décidé de diviser l'espace en carrés tel que chaque carré soit le plus grand carré qu'on peut insérer dans un cercle de diagonale d. Ainsi, on a divisé l'espace en carrés de coté $\frac{r}{\sqrt{2}}$.

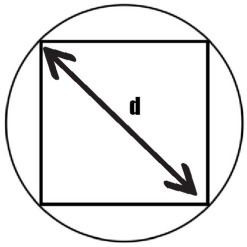


Figure 4:

Il suffit donc de diviser l'espace suivant cette convention. Le but de cette division est de minimiser le nombre de points qu'on doit visiter pour trouver les composantes connexes de notre graphe. Puisque si on a un point *point0* dans un quadrant a lié à un point *point1* dans un autre quadrant b, alors tous les points dans les deux quadrants a et b appartiennent à la même composante connexe .

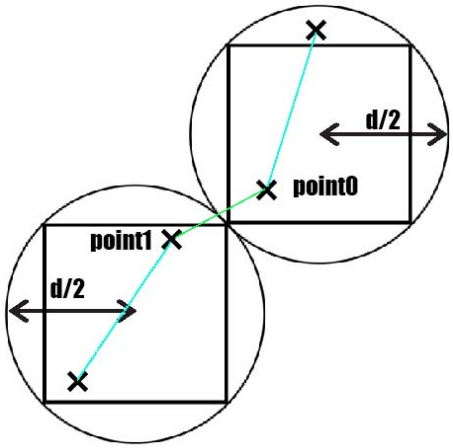


Figure 5:

Une fois l'espace divisé, on commence par parcourir les quadrants. Pour chaque quadrant, on calcule les distances entre les points de ce quadrant avec ceux des voisins.

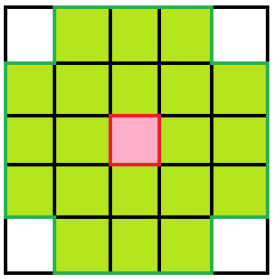


Figure 6: En vert : les voisins du quadrant en rose

Une fois qu'on trouve qu'un quadrant a et un quadrant b constituent une composante connexe on ajoute ce dernier dans une file pour étudier ses quadrants voisins après avoir fini l'étude des voisins du premier quadrant a. Après avoir terminer l'étude de a, on va étudier le quadrant qui est le premier élément de la file. A la fin, quand la file est vide, on obtient une composante connexe, puis on passe au quadrant suivant qui n'appartient à aucune composante connexe déja trouvée. Il suffit de stocker ces composantes dans une list et de calculer leurs tailles.

Avant d'avoir eu l'idée d'implémenter une file, nous avons eu l'idée de tenter un parcours en profondeur d'un graphe(V, E qui, pour chaque sommet s du graphe, i.e quadrant, va visiter l'ensemble des sommets accessibles depuis s, avec E représentant l'existence de deux points liées appartenants à deux quadrants distincts. Cependant, nous n'avons pas testé cet algorithme avec les tests du professeur, puisque, même en le testant localement, il mettait beaucoup de temps à s'éxécuter. La compléxité d'un parcours en profondeur d'un graphe au pire des cas est O($V+E$) et on sait que dans un graphe simple comlpet, E ¡ n^2. Ainsi, le parcours de graphe en profondeur est de O(n^2), sans prendre en compte, la complexité du reste de l'alogrithme (division de l'espace, etc.).

Soit m le nombre de quadrant et n le nombre de points.

Les structures de données utilisées dans l'algorithme final, qui se sert d'une file, sont

- d : Un dict qui va servir à la division de l'espace, dont les clés sont les quadrants et dont les valeurs sont des list de points appartenant au

quadrant clé. On ne fait qu'ajouter des éléments à ce dictionnaire et à ses lists, qui se font en O(1) pour chaque ajout. Le coût spatial est de O(n).

- composantes : Une list qui va contenir les composantes connexes, l'ajout d'une composante se fait en O(1). Son coût spatiale est de O(m).
- visited : Un set qui va contenir l'ensemble des quadrants déjà visités. On a choisi que cette variable soit un set pour les mêmes raisons que dans la méthode naïve (ajout d'élément en O(1), pas de doublons ce qui avantageux pour les tests d'appartenance, etc.). Son coût spatial est de O(m).
- file : Une list, qui va servir comme queue de priorités qui ordonne les quadrants qu'il faut étudier. On a choisi une list puisque dans un set il n'y a pas d'accès par index, ce qui est nécessaire pour notre algorithme. Le coût spatial est de O(m) au pire des cas.

La complexité temporelle de notre algorithme en se servant d'une file est dans le meilleur des cas, i.e le cas où les points appartiennent au même quadrant, est O(n).

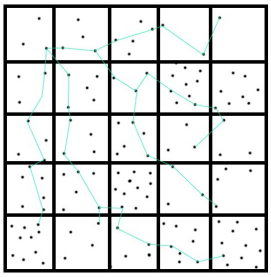


Figure 7: En bleu le chemin suivi lors de l'algorithme

Et au pire des cas c'est O(n^2) : à chaque quadrant on parcourt les 20 quadrants voisins et on ne trouve aucune liaison entre ces quadrants .

On a la probabilité suivante : P(qu'un point appartient à un quadrant i) = $\frac{surfaceduquadrant}{surfacetotale}$.

On pose N_i la variable aléatoire qui donne le nombre de points appartenant au quadrant i . N_i suit une loi binomiale de paramètre n et p tel que p est la probabilité calculée avant.

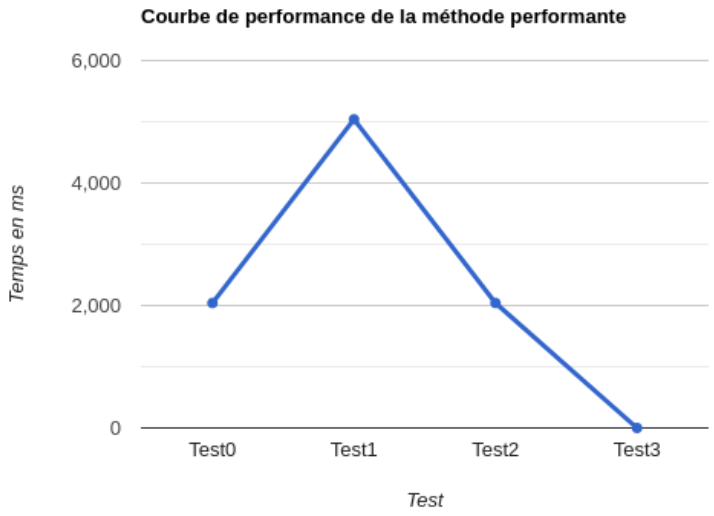
Donc ,en moyenne , chaque quadrant contient E(N_i) points. Cette espérance est égale à $n \times p$,on multiplie cela par le nombre de quadrants qu'on visite dans l'algorithme .

Donc $C(n) = \frac{surfacetotale}{surfaceduquadrant} \times 20 \times (np)^2$

Finalement $C(n) = 20 \times n^2 p$

En ce qui concerne, la complexité spatiale de l'algorithme, elle est en O(n) dans tous les cas :

- Dans le meilleur des cas, tous les points apartiennent à l'unique quadrant, le coût spatial est le coût du dict d donc O(n).
- Dans le pire des cas, chaque point est seul dans un quadrant, il y a donc n quadrants donc le coût spatial est de O(n).



Il est facile de remarquer qu'avec cet algorithme, on réussi finalement les Test1 et Test2, mais toujours pas le Test3. Comme on ignore les conditions d'entrées de ces tests, on peut supposer que les conditions d'entrées notre algorithme se limite aux entrées du Test3. Cet algo est sûrement plus performant au niveau temporel que la méthode naïve.

2.4 Conclusion :

En conclusion, les deux algorithmes sont basés sur des idées différentes pour résoudre le même problème, et ont des avantages et des inconvénients en termes de complexité spatiale et temporelle. Au niveau des compléxités spatiales, aucun des deux à un avantage assez important sur l'autre, puisqu'ils sont tous les deux en O(n). O(n) est considérée comme une très bonne complexité car elle signifie que l'espace mémoire nécessaire à l'exécution de l'algorithme est linéaire par rapport à la taille de l'entrée.

D'autre part, en ce qui concerne les compléxités temporelles, il est clair que la méthode performante a un avantage énorme par rapoprt à la méthode naïve. Même si au pire des cas, les deux sont en $O(n^2)$, dans certains cas, la compléxité de la méthode performante peut atteindre O(n) alors qu cela n'est jamais le cas pour l'autre. En générale, pour des données assez nombreuses, on va préferer travailler avec l'algorithme performant.

Aux niveaux personnels, nous avons appris beaucoup de chose en faisant ce projet. Notamment, la découverte de la structure K-D Tree, qui aurait du être très utile pour la résolution de ce problème si on a avait le droit d'avoir une précision inférieur à 100% . Egalement, le travail en binôme nous a permis de partager les idées et les connaissances de façon à comprendre qu'il n'y a pas toujours une seul méthode qui est juste et qui fonctionne. De plus, au niveau pratique, le projet nous a donné la chance d'observer de proche comment le choix des structures de données utilisées, l'emplacement des conditions "if" etc. jouent un rôle d'une grande importance dans la performance de l'algorithme.