Compte rendu Projet Algorithme

Valérian THOMAS
Paul DUDNIC

Mai 2019

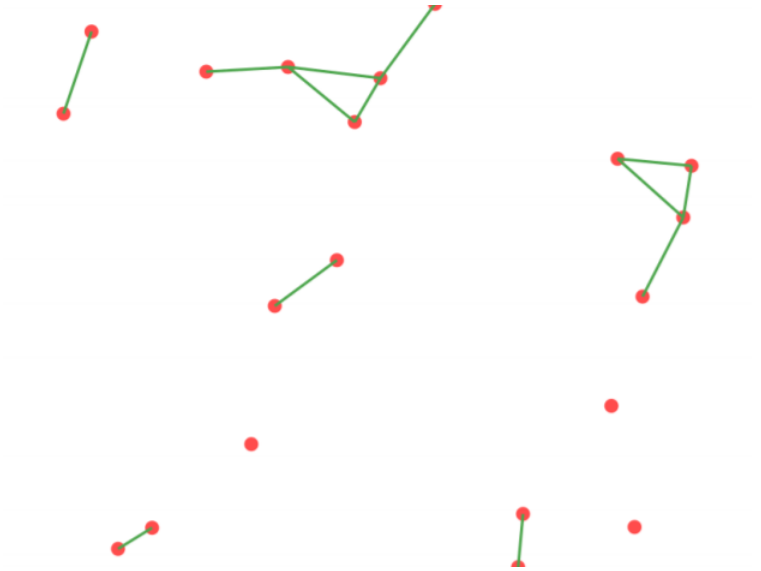


FIGURE 1 – Composantes connexes dans un graphe

1 Introduction

Le projet consistait en trouver un algorithme permettant de calculer la taille des composantes connexes d'un graphe de points puis de renvoyer la liste des tailles pas ordre décroissant.

Pour cela, nous avons implémenté des algorithmes utilisant trois méthodes

1. Un algorithme utilisant une méthode BFS
2. Un algorithme de croissance de couronne
3. Un algorithme quadrillant l'espace en petites sections

2 Algorithme naïf : BFS

Le principe de cet algorithme est le même que celui d'un parcours de graphe en largeur (BFS).

2.1 Pseudo-code

L'algorithme prend en entrée une liste de points *points* et une distance seuil *distance*.

Algorithme 1 : Algorithme BFS

```
1  Begin;
2  Initialiser listeGraphes ← listeVide;
3  tant que points n'est pas vide faire
4  |  Initialiser pileActuelle ← pileVide;
5  |  Initialiser grapheActuel ← listeVide;
6  |  pointEtudie ← points.pop();
7  |  Marquer pointEtudie;
8  |  Ajouter pointEtudie à pileActuelle;
9  |  tant que pileActuelle n'est pas vide faire
10 |  |  pointTeste ← pileActuelle.pop();
11 |  |  Ajouter pointTeste à grapheActuel;
12 |  |  point de points  si point est dans un carré de centre pointTeste de coté distance
   |  |  alors
13 |  |  |  si point est à une distance inférieure à distance de pointTeste alors
14 |  |  |  |  Ajouter point à pileActuelle;
15 |  |  |  finSi
16 |  |  finSi
17 |  |  point de pileActuelle  si point n'est pas marqué alors
18 |  |  |  Marquer point;
19 |  |  |  Enlever point de points;
20 |  |  finSi
21 |  finTantQue
22 finTantQue
23 Initialiser listeTaillesGraphes ← taille de chaque graphe de listeGraphes;
24 Trier listeTaillesGraphes par odre décroissant;
25 Renvoyer listeTaillesGraphes
```

2.2 Remarques sur l'algorithme

2.2.1 Marquage des points

Afin de ne pas tester tous les points de la liste *points* à chaque itération, on supprime au fur et à mesure les points qu'on ajoute aux graphes connexes. Cependant, pour éviter d'avoir des bugs dûs à une suppression d'un élément non existant de la liste, il faut marquer les points une fois visités et ne supprimer que les points n'étant pas marqués.

2.2.2 "Carré de centre *pointEtudie* (l12)"

Calculer la distance entre deux points est une opération coûteuse. Il est plus rapide de vérifier tout d'abord si la distance entre *point* et *pointTeste* vaut la peine d'être calculée par deux simples inégalités. En notant x et y les coordonnées d'un point, on vérifie :

$$pointTeste.x - distance \leq point.x \leq pointTeste.x + distance$$
$$pointTeste.y - distance \leq point.y \leq pointTeste.y + distance$$

Si ces conditions sont vérifiées, on calcule la distance entre $pointTeste$ et $point$.

2.3 Variantes

Nous avons essayé plusieurs variantes pour cet algorithme, en utilisant différentes structures de données pour stocker la liste *points* : vecteur (liste python), liste chaînée, dictionnaire. Finalement la variante vectorielle s'est révélée être la plus rapide.

Une autre possibilité aurait été d'utiliser un parcours en profondeur des graphes (DFS). Nous n'avons cependant pas exploré cette piste, préférant travailler sur d'autres méthodes moins "naïves".

3 Algorithme de croissance de couronne

Lors d'une séance de TD, nous avons eu l'idée de penser à un algorithme à zone croissante.

3.1 Pseudo-code

L'algorithme prend en entrée une liste de points *points* et une distance seuil *distance*.

Algorithme 2 : Algorithme croissance de couronne

```
1  Début;
2  Initialiser Nuage ← listeVide;
3  tant que points n'est pas vide faire
4  |  Initialiser grapheActuel ← listeVide;
5  |  pointEtudie ← points.pop();
6  |  Marquer pointEtudie;
7  |  Appliquer RechercheGraphe au pointEtudie;
8  |  Ajouter Grapheretour à Nuage;
9  finTantQue
10 Initialiser listeTaillesGraphes ← taille de chaque graphe de Nuage;
11 Trier listeTaillesGraphes par ordre décroissant;
12 Renvoyer listeTaillesGraphes
```

On explicite ici l'algorithme de Recherche de Graphe à partir d'un point

Algorithme 3 : Algorithme Recherche Graphe

```
1  Début;
2  Initialiser Graphe ← [Point_etudie];
3  Initialiser point_restant ← listeVide;
4  Initialiser max ← −1;
5  Initialiser largeur ← d_seuil;
6  tant que max ≠ 0 faire
7  |  Prendre un point_calcul de la liste de points;
8  |  si point est dans un carré de centre pointEtudie de coté 2 ∗ largeur alors
9  |  |  si point est à une distance inférieure à distance d'un des points de Graphe alors
10 |  |  |  calculer la distance dist entre point_calcul et PointEtudie;
11 |  |  |  affecter à max ← max(max, dist);
12 |  |  |  Ajouter point à pileActuelle;
13 |  |  finSi
14 |  |  sinon
15 |  |  |  Ajouter point_calcul à point_restant;
16 |  |  finSi
17 |  finSi
18 |  sinon
19 |  |  Ajouter point_calcul à point_restant;
20 |  finSi
21 finTantQue
22 si max > 0 alors
23 |  largeur ← max + d_seuil
24 finSi
25 Renvoyer Graphe et point_restant
```

3.2 Remarques sur l'algorithme

3.2.1 Marquage des points

Le marquage des points ici est simplement le fait de soustraire à la liste de points notre point d'étude. Cela évite de faire trop de calculs.

3.2.2 carre de coté 2*largeur

Pour le carré de centre $Point\_Etudie$ et de coté $2 * Largeur$, on considère en effet au début que $Largeur = d\_seuil$. On appelle ici $largeur$ la distance entre le centre et un des coté du carré. C'est pourquoi on obtiens un carré de longueur $a = 2 * largeur$

3.3 Amélioration par changement de forme

3.3.1 Rectangle > carré ?

Une idée d'amélioration de ce programme fut de considérer non pas un carré centré sur notre point d'étude mais plutôt un rectangle centré sur ce point.

L'idée est donc de modifier la condition $IsinCarre$ par $IsinBox$. Pour cela,on change :

$$pointEtude.x - distance \leq point.x \leq pointEtude.x + distance$$
$$pointEtude.y - distance \leq point.y \leq pointEtude.y + distance$$

En un test du type :

$$pointEtude.x - distance_x \leq point.x \leq pointEtude.x + distance_x$$
$$pointEtude.y - distance_y \leq point.y \leq pointEtude.y + distance_y$$

où distance_x et distance_y représentent respectivement la demi-largeur et demi-longueur de notre rectangle. On augmentera donc ces distances de manières indépendantes. Cependant on considérera toujours notre $PointEtudie$ comme centre du rectangle. Par conséquent si on augmente la distance entre le centre et le coté vertical supérieur, il en sera de même avec son homologue inférieur. Il en va de même pour les cotés horizontaux.

Ainsi en observant la position du point, on peut réduire la surface de notre carré a celle d'un rectangle et donc considérer moins de points inutile par la suite.

3.3.2 Rectangle centré ou non centré ?

Une dernière idée d'amélioration est de ne plus considérer que notre rectangle est centré sur le point Étudie. Cela revient à modifier de manière indépendante chacune des distances entre les cotés et le centre.

Cela revient aussi à modifier une nouvelle fois notre test en implémentant :

$$pointEtude.x - distance_{xsup} \leq point.x \leq pointEtude.x + distance_{xinf}$$
$$pointEtude.y - distance_{ygauche} \leq point.y \leq pointEtude.y + distance_{ydroite}$$

Cela a à nouveau pour but de réduire la surface de considération au maximum sans nécessiter trop de calculs. A l'aide des graphes de performances, nous pourrons alors vérifier si nos hypothèses sont justes et si il est intéressant d'implémenter l'une ou l'autre de ces variantes.

4 Algorithme Quadrillage de l'espace

Dans cet algorithme on va quadriller l'espace et travailler sur des petites cases. On détermine tout d'abord les composantes connexes dans chaque case de l'espace puis on fusionne les composantes connexes des différentes cases si nécessaire.

La version implémentée travaille sur des cases carrés de côté $distance/\sqrt{2}$ (donc de diagonale *distance*) de manière à ce que tous les points d'une même case soient à une distance inférieure à *distance* les uns des autres. On a ainsi la certitude que tous les points d'une même case appartiennent à la même composante connexe, ce qui évite de devoir calculer les composantes connexes dans chaque case. Il ne reste alors qu'à fusionner les composantes connexes entre elles.

Si un point d'une case est à une distance inférieure à *distance* d'un point d'une autre case alors on peut fusionner les composantes connexes des deux cases.

4.1 Pseudo-code

L'algorithme prend en entrée une liste de points *points* et une distance seuil *distance*.

Algorithme 4 : Algorithme Quadrillage

```
1  Debut;
2  Créer listeCases de la bonne taille (dépend de distance);
3  pour chaque case de listeCases faire
4  |  Initialiser case ← listeVide;
5  finPour
6  pour chaque point de points faire
7  |  Ajouter point à la liste corréspondant à la case à laquelle il appartient;
8  finPour
9  Initialiser dictGraphes ← dictionnaireVide;
10 pour chaque caseActuelle de listeCases faire
11 |  si caseActuelle n'est pas vide alors
12 |  |  si caseActuelle n'appartient à aucun graphe alors
13 |  |  |  Créer une liste contenant caseActuelle;
14 |  |  |  Ajouter cette liste à dictGraphes[caseActuelle];
15 |  |  |  pour chaque caseTeste proche de caseActuelle faire
16 |  |  |  |  Appliquer FusionGraphes(caseTeste, caseActuelle, dictGraphes);
17 |  |  |  finPour
18 |  |  finSi
19 |  finSi
20 finPour
21 Initialiser listeTaillesGraphes ← le nombre de points de chaque graphe de dictGraphes
     Trier listeTaillesGraphes par odre décroissant;
22 Renvoyer listeTaillesGraphes
```

On utilise une fonction auxiliaire *FusionGraphes* définie ci-dessous.

Algorithme 5 : Fonction FusionGraphes(caseTeste, caseActuelle, dictGraphes)

```
1  Debut;
2  si caseTeste n'est pas vide alors
3      si caseTeste et caseActuelle n'appartiennent pas au même graphe alors
4          pour chaque point1 de caseActuelle faire
5              pour chaque point2 de caseTeste faire
6                  si point2 est à une distance inférieure à distance de point1 alors
7                      si caseTeste n'appartient à aucun graphe alors
8                          Ajouter caseTeste à dictGraphes[caseActuelle]
9                      finSi
10                     sinon
11                         pour chaque case de dictGraphes[caseTeste] faire
12                             Ajouter case à dictGraphes[caseActuelle]
13                         finPour
14                         Supprimer dictGraphes[caseTeste];
15                     finSi
16                     Sortir de la fonction;
17                 finSi
18             finPour
19         finPour
20     finSi
21 finSi
```

4.2 Remarques sur l'algorithme

4.2.1 "Proximité" des cases

Au lieu de tester l'ensemble des cases du quadrillage (ce qui résulterait en une complexité en n^2 en fonction du nombre n de cases), on se limite aux cases proches *ie* les cases dont les points peuvent être à une distance inférieure à *distance* de ceux de la case étudiée. Il s'agit des cases :

— en contact avec la case étudiée (contact par les coins aussi)

— en contact avec une case elle même en contact avec la case étudiée

On se limite ainsi à un carré de 25 cases centré autour de la case étudiée. En pratique on étudie encore moins de cases. En effet, dans l'implémentation Python on parcourt les cases ligne par ligne, ce qui nous permet de ne tester que les cases dont l'ordonnée est inférieure ou égale à celle de de la case étudiée.

4.2.2 Structure de donnée utilisée

Nous avons utilisé un dictionnaire pour stocker la liste des cases ainsi que la liste des graphes. Pour la liste des cases, nous aurions pu utiliser d'autres structures comme un tableau (l'indice de chaque case devrait alors être calculé par une formule), la structure de dictionnaire étant simplement plus pratique grâce à l'indexation par clé (ici les coordonnées (x, y) de chaque case). Cependant concernant la liste des graphes connexes, la structure de dictionnaire nous permet de rajouter et de supprimer de composantes connexes en complexité $O(log(n))$ et donc de réduire le temps de calcul par rapport à une liste Python.

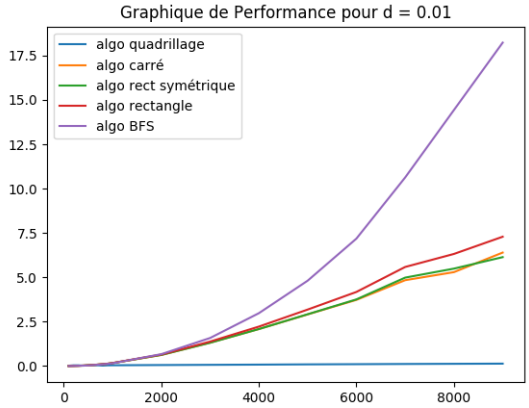
4.3 Variantes

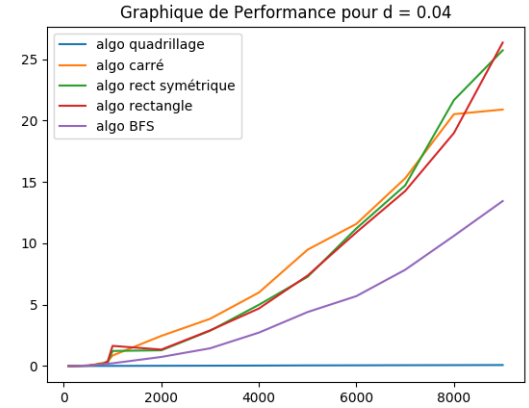
Une variante de cet algorithme à laquelle nous avons pensé est celle de l'équivalent récursif : diviser l'espace en quatre sections récursivement jusqu'à ce que la taille de chaque case soit inférieure ou égale à $distance/\sqrt{2}$ et de fusionner les composantes connexes des quatre cases. Nous n'avons cependant pas eu le temps d'explorer cette piste.

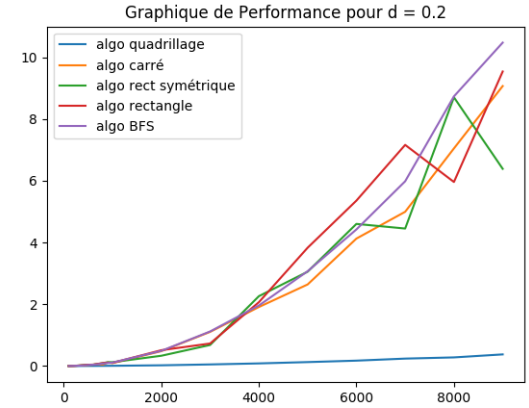
5 Graphes de Performances et choix de l'algorithme

Les différents graphiques ont en abscisse le nombre de points du graphe et en ordonnée le temps d'exécution de l'algorithme en secondes.

5.1 Comparaisons des 5 algorithmes





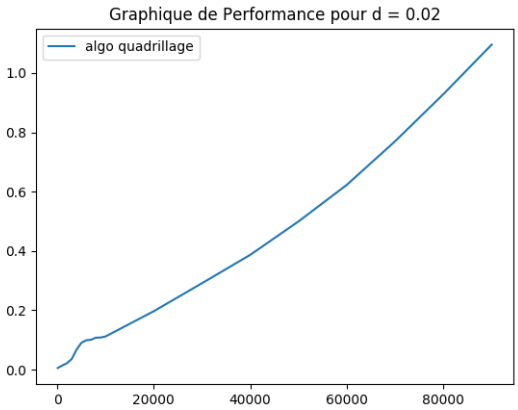


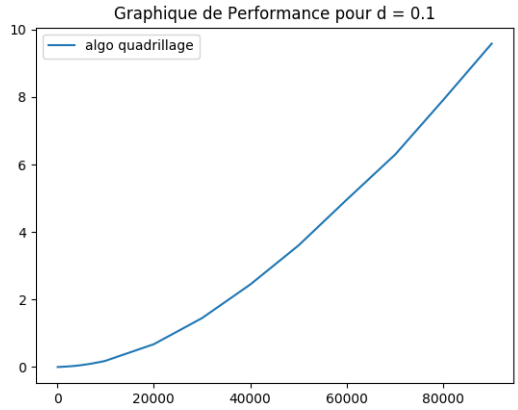
A l'issue de ces graphes, nous pouvons constater 3 états de faits :

1. En règle général l'algorithme BFS est moins bons que les autres, surtout à partir de $d > 0.2$
2. Malgré nos prévisions, il semble que ce soit en général l'algorithme de croissance de couronne avec un carré qui soit le plus performant
3. Il est indéniable que parmi tous nos algorithmes, le quadrillage est de loin le plus rapide.

Avec ces résultats il est donc intéressant d'observer les courbes de notre algorithme de quadrillage. Toujours en entrée un liste de N points, N variant entre 10 et 100 000 points.

5.2 Performances de notre meilleur algorithme : Quadrillage





Par une recherche de complexité, nous n'avons pu obtenir que l'information suivante : $O(log(n)) < C(n) < O(nlog(n))$

6 Conclusion

6.1 Finalité

Au fil de notre projet, nous avons découvert, codé et implémenté différentes manières de résoudre le problème posé. Nous avons ainsi découvert que certaines solutions n'étaient pas viables, ou moins bonnes que d'autre. Cela nous a permit de modifier notre manière de penser et de coder.

De plus, l'ensemble des TD en algorithme nous ont laissés bon nombre de pistes afin d'optimiser chacun de nos codes comme l'utilisation d'ABR ou de Tas. Bien que nous n'ayons pas utilisé ces solutions, nous avons tout de même pris compte des conseils donnés.

En finalité de ce projet, nous avons donc réussi à implémenter un algorithme fonctionnel, réalisant les trois premiers niveau de test soumis à notre encontre et réalisant un temps correct. De plus, en comparant avec les courbes de complexité connue comme $nlog(n)$ et $log(n)$ nous avons pu déterminer que notre algorithme admet une complexité $C(n)$ telle que $O(log(n)) < C(n) < O(nlog(n))$.

6.2 Aller plus loin ...

Pour aller plus loin dans notre optimisation, il serait intéressant d'utiliser les structures vues en cours, en TD et en examen, par exemple par l'utilisation d'arbres ou de tas. Les structures abordées dans l'exercice I de l'examen d'algorithmique nous intéressent particulièrement : elles nous permettraient de réduire les coûts de fusion de graphes, d'ajout de points dans un graphe, ou de recherche du graphe auquel appartient un point.