

## TIPE - La Ville

Comment créer un réseau de transports publics dans une ville afin de minimiser les temps de déplacement ?

Valentin FOULON - N° 29836

August 14, 2023

# Table des matières

- 1 Choix du thème
- 2 Définitions
- 3 Objectifs du TIPE
- 4 Conditions d'exploitation
- 5 Première approche : algorithme de Dijkstra
- 6 Deuxième approche : algorithme de Kruskal
- 7 Troisième algorithme
- 8 Conclusion
- 9 Codes

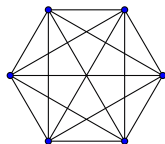
## Choix du thème

# Définitions

# Définitions

## Définition

Graphe : on note  $G = (S, A)$  un graphe ayant pour sommets l'ensemble  $S$  et pour arêtes l'ensemble  $A$ . On désigne par  $|A|$  et  $|S|$  les cardinaux respectifs de  $A$  et  $S$ .



## Remarque

On considère ici des graphes non orientés connexes : il n'y a pas de point isolé dans le graphe.

## Définition

Poids d'une arête : fonction  $d : A \rightarrow \mathbb{N}, (u, v) \mapsto d(u, v)$  utilisée pour les algorithmes de graphe. On utilisera ici la distance euclidienne entre deux sommets.

On représente les villes par des graphes non orientés connexes. Les centres d'intérêt sont des sommets de ce graphe. Pour chaque couple  $(u, v)$  de sommets dans le graphe, si la distance entre  $u$  et  $v$  est inférieure à une valeur donnée, il existe une arête entre  $u$  et  $v$ . On suppose que le graphe ainsi créé reste connexe.

## Objectifs du TIPE

Etudier différents algorithmes pour créer un réseau de transport public et les comparer

- Les algorithmes doivent produire des résultats efficaces
- Les algorithmes doivent être efficaces
  - ▶ On espère ici avoir une complexité inférieure à  $O(n^2)$
- Tous les points d'intérêt doivent être reliés entre eux
- La solution trouvée doit être réaliste



## Conditions d'exploitation

# Conditions d'exploitation

- On utilise l'exemple des métros car pas de contrainte de parcours
- Les trains circulent à vitesse constante (proportionnalité entre distance et temps de trajet)
- Le temps d'arrêt à une station est nul
- Le temps d'attente à une station est nul

Ces suppositions ne sont pas réelles mais influent peu sur le résultat.

Les unités sont arbitraires

- On utilise un plan de taille  $100 \times 100$
- Pour les tests on positionne au plus 1000 stations de manière aléatoire

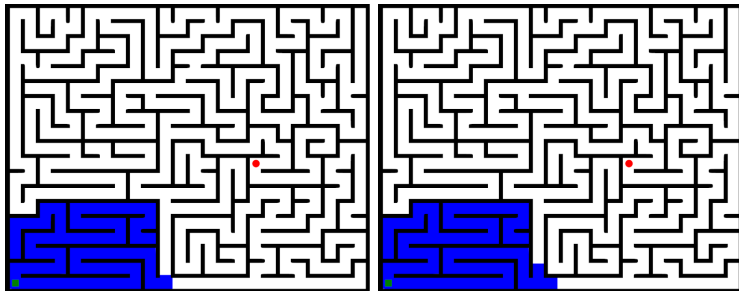
## Première approche : algorithme de Dijkstra

## Algorithme de Dijkstra

Algorithme de plus court chemin dans un graphe.

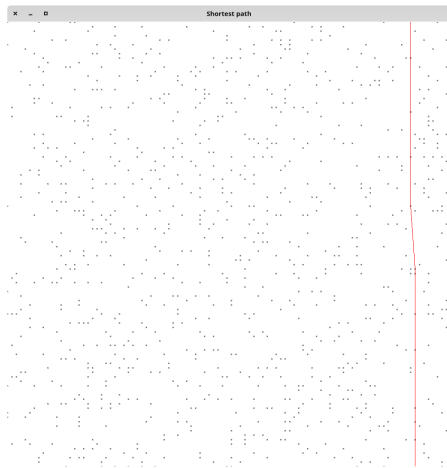
Objectif : trouver le plus court chemin entre les bords haut et bas du plan.

# Choix et utilisation de l'algorithme



- Complexité :  $O(|A|\log|A|)$
- Trouve le meilleur résultat à chaque fois
- Fonctionne car la fonction de poids est positive

# Production de l'algorithme

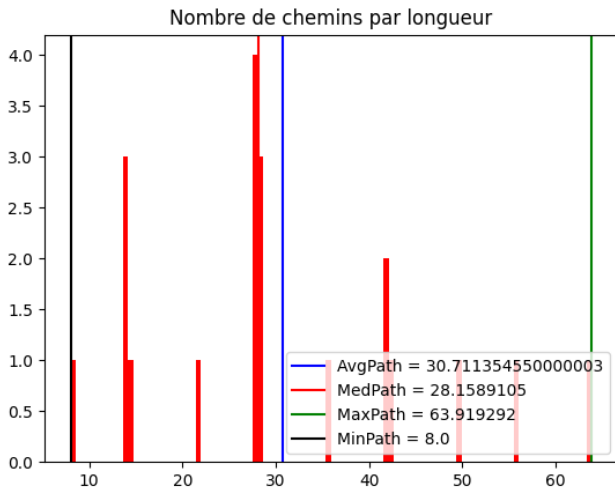


Cet algorithme crée des trajets avec peu de changements de direction.



Les différents algorithmes employés ici permettent de sauvegarder les graphes créés dans des fichiers. Un autre programme peut ensuite calculer les informations que l'on souhaite sur ces graphes : moyenne, médiane, longueur minimum, maximum.

# Résultat sur un jeu de données aléatoire



- La solution trouvée est alors optimale en terme de temps de trajet
- La capacité de transport entre les deux points vaut  $(|S| - 2) \times$   
"nombre de places dans une rame"
- Cet algorithme donne le plus court chemin entre deux points spécifiques du graphe, on peut ainsi l'appliquer sur chaque couple de sommets
- Mais il reste deux problèmes
  - ▶ Risque de lignes en double sur des portions de trajet
  - ▶ Beaucoup de lignes à créer

## Deuxième approche : algorithme de Kruskal

## Algorithme de Kruskal

Algorithme qui trouve un arbre couvrant de poids minimal dans un Graphe.

## Définition

Un arbre couvrant est un sous-graphe  $G' = (S, A')$  de  $G = (S, A)$  où  $A' \subset A$  et pour tous  $(u, v) \in A$ , il existe un chemin de  $u$  à  $v$  dans  $G'$ .

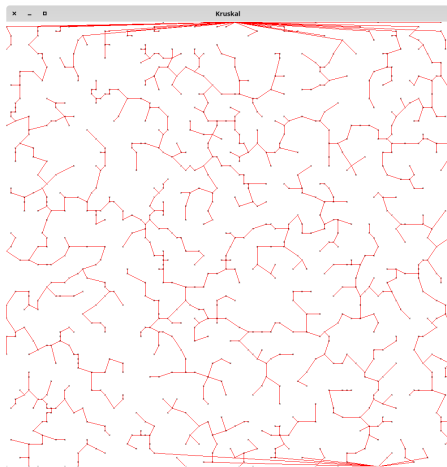
## Remarque

Un arbre couvrant de poids minimum est **un** arbre couvrant dont la somme des poids des arêtes de  $A'$  est la plus petite.

# Choix et utilisation de l'algorithme

- Complexité :  $O(|A|\log|A|)$
- Algorithme correct : le sous-graphe renvoyé est un arbre couvrant de poids minimum

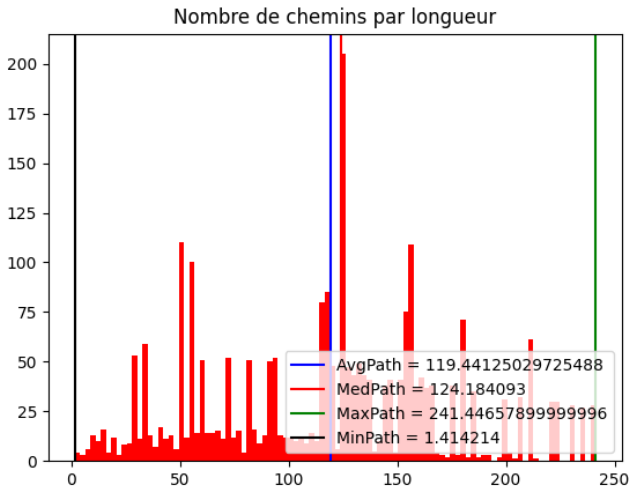
# Production de l'algorithme



Chaque sommet est relié à tous les autres par un chemin.



# Résultat sur un jeu de données aléatoire



Une grande partie des chemins a une longueur faible.

- Cet algorithme fournit un unique chemin entre chaque couple de sommets sauf entre les sommets haut et bas
- La capacité de transport entre les deux points vaut le nombre de places dans une rame
- Certains chemins semblent donc absurdes et beaucoup trop longs entre certains couples de sommets
- On peut alors ajouter des arêtes avec l'algorithme de Dijkstra vu avant
- Beaucoup de lignes différentes à créer

## Troisième algorithme

## Résumé de ce qu'on a vu

Les deux algorithmes vu précédemment ne suffisent pas à répondre à la question. De plus les villes se développent en agglomérations, certaines lieux peuvent devenir des hubs.

Utiliser l'algorithme des K-moyennes afin de regrouper des centres d'intérêt entre eux, relier les groupes entre eux puis relier les centres d'intérêt dans ces groupes.

## K-moyennes

Algorithme de regroupement de points en  $K$  groupes appelés clusters afin de minimiser la distance entre les points d'un même groupe.

## Remarque

On utilise ici de l'algorithme de Lloyd, ou algorithme naïf.

# Choix et utilisation de l'algorithme

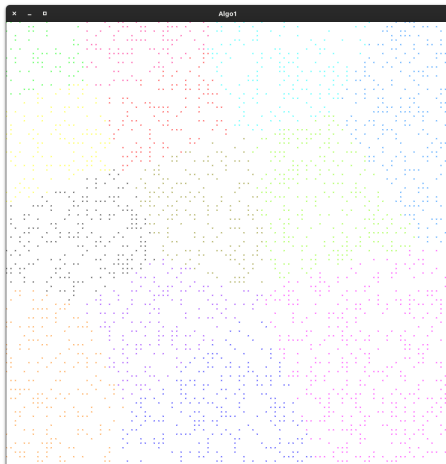
- Complexité :  $O(|S|Ki)$ ,  $i$  étant le nombre d'itérations avant d'obtenir une solution
- L'algorithme produit une solution souvent proche de l'optimale

## Attention

L'algorithme ne converge pas nécessairement vers une solution pour certaines combinaisons de jeux de données, valeurs de  $K$ , positions initiales des centroïdes.

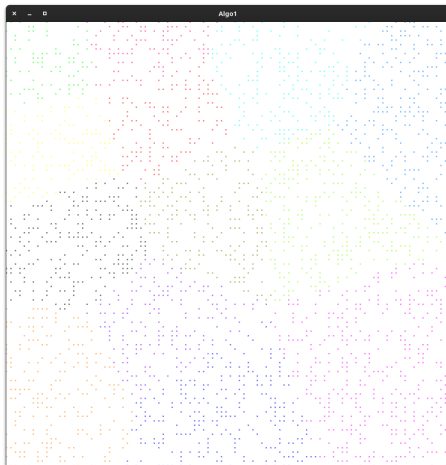


# Résultat sur un jeu de données aléatoire



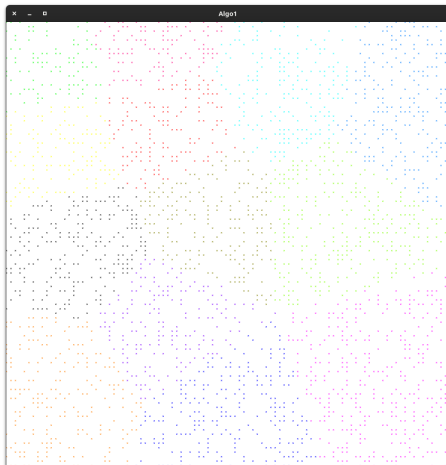
1e itération

# Résultat sur un jeu de données aléatoire



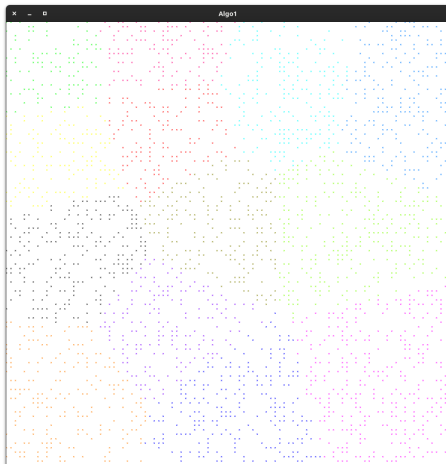
2e itération

# Résultat sur un jeu de données aléatoire



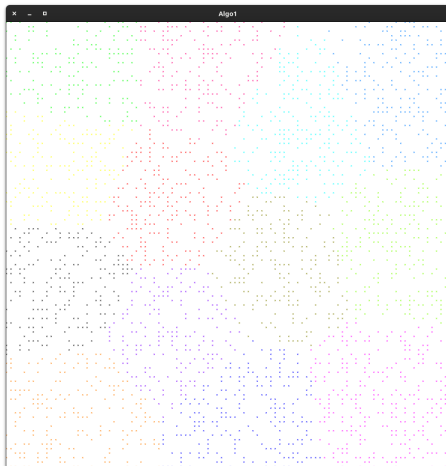
3e itération

# Résultat sur un jeu de données aléatoire



4e itération

# Résultat sur un jeu de données aléatoire



Après plusieurs itérations, convergence.

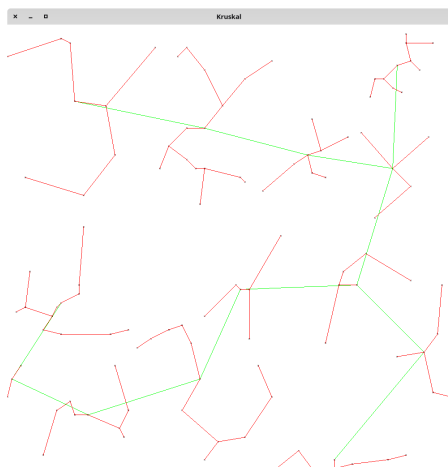
On applique l'algorithme des K-moyennes sur la ville en donnant à l'algorithme une valeur de K cohérente avec ce que l'on veut. On peut ensuite appliquer l'un des deux algorithmes précédents sur chaque cluster puis relier les cluster entre eux.

- Si on considère des quartiers d'une ville, on peut relier les clusters directement entre eux
- Si on considère une ville et son agglomération, on relie chaque cluster au cluster de la ville principale

Complexité totale de l'algorithme :  $O(K(|S|i + |A|\log|A|))$ .

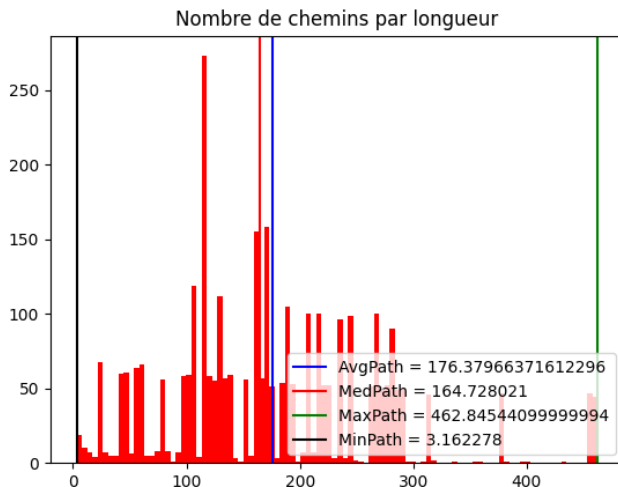
- Création de chemins plus courts avec un réseau reliant les clusters entre eux
- Réduction de la complexité du problème en créant le réseau sur des ensembles plus petits
- Adaptation à des villes dont les centres d'intérêt ne sont pas répartis de façon homogène dans l'espace, ou des agglomérations

# Production de l'algorithme final sur un jeu de données fixé





# Résultat de l'algorithme final



## Conclusion

# Codes

```
1     s2->val = li1->val;
2     s2->next = copyEdgeList(li1->next);
3     return s2;
4 }
5 Stack* copyStack(Stack* li1) {
6     if (li1 == NULL) return NULL;
7     Stack* s2 = malloc(sizeof(Stack));
8     s2->val = li1->val;
9     s2->next = copyStack(li1->next);
10    return s2;
11 }
12 int edgeListLength(Solution* s1) {
13     Solution* sln = copyEdgeList(s1);
14     if (sln == NULL) {
15         free(sln);
16         return 0;
17     }
18     int ttl = 1 + edgeListLength(sln->next);
19     free(sln);
```

```
20     return ttl;
21 }
22 Edgelist* convert(Solution* s1) {
23     Edgelist* sln = malloc(sizeof(Edgelist));
24     sln->s = edgeListLength(s1);
25     sln->list = malloc(sln->s * sizeof(edge));
26     Solution* sln2 = copyEdgeList(s1);
27     for (int i = 0; i < sln->s; i++) {
28         sln->list[i] = sln2->val;
29         sln2 = sln2->next;
30     }
31     free(sln2);
32     return sln;
33 }
```

```
1     free(sln);
2     return 0;
3 }
4 int ttl = 1 + edgeListLength(sln->next);
5 freeEdgeList(sln);
6 return ttl;
7 }
8 Edgelist* convert(Solution* s1) {
9     Edgelist* sln = malloc(sizeof(Edgelist));
10    sln->s = edgeListLength(s1);
11    sln->list = malloc(sln->s * sizeof(edge));
12    Solution* sln2 = copyEdgeList(s1);
13    for (int i = 0; i < sln->s; i++) {
14        sln->list[i] = sln2->val;
15        sln2 = sln2->next;
16    }
17    freeEdgeList(sln2);
18    return sln;
19 }
```

```
20 void print_edgelist(Edgelist* c) {  
21     for (int i = 0; i < c->s; i++) {  
22         printf("%d %d %lf\n", c->list[i].s1, c->list[i].s2,  
23             c->list[i].w);  
24     }  
25     printf("\n");  
26 }  
27 Solution* mapToS(Solution* sln) {  
28     Solution* t = copyEdgeList(sln);  
29     Solution* s = t;
```

```
1      while (s != NULL) {
2          count++;
3          s = s->next;
4      }
5      fprintf(fd, "%d\n", NMAX);
6      s = sets[j];
7      while (s != NULL) {
8          int id = s->val.id;
9          int x = s->val.x0;
10         int y = s->val.y0;
11         fprintf(fd, "%d %d %d\n", id, x, y);
12         s = s->next;
13     }
14     for (int i = 0; i < NMAX - count; i++)
15     {
16         fprintf(fd, "%d %d %d\n", -1, 0, 0);
17     }
18     fclose(fd);
19 }
```



```

19         char* outFile;
20         asprintf(&outFile, "%s%d.txt", outScheme
, 0);

21         fd = fopen(outFile, "w");
22         fprintf(fd, "%d\n", NMAX);
23         for (int j = 0; j < K; j++) {
24             int id = sets[j]->val.id;
25             int x = sets[j]->val.x0;
26             int y = sets[j]->val.y0;
27             fprintf(fd, "%d %d %d\n", id, x, y);
28         }
29         for (int i = 0; i < NMAX - K; i++) {
30             fprintf(fd, "%d %d %d\n", -1, 0, 0);
31         }
32         fclose(fd);
33     }
34     break;
35 }
36 case SDLK_RETURN: {

```

37

```
getSetAvg(sets);
```

```
1         print(edge1, edge2, path)
2         if path != None and path != 0:
3             paths.append(path)
4 paths = np.array(paths)
5 Y = np.array(list(map(lambda elt : elt['w'], X)))
6 ya = Y.mean()
7 print("avgEdge =", ya)
8 ys = Y.std()
9 print("stdEdge =", ys)
10 yM = Y.max()
11 print("maxEdge =", yM)
12 ym = Y.min()
13 print("minEdge =", ym)
14 yS = np.median(Y)
15 print("medEdge =", yS)
16 xa = np.mean(paths)
17 print("avgPath =", xa)
18 xs = np.std(paths)
19 print("stdPath =", xs)
```

```
20 xM = np.max(paths)
21 print("maxPath =", xM)
22 xm = np.min(paths)
23 print("minPath =", xm)
24 xS = np.median(paths)
25 print("medPath =", xS)
26 plt.hist(Y, bins=100, color='b')
27 plt.axvline(x=ya, color='blue', label='AvgEdge = '+str(ya))
28 plt.axvline(x=yS, color='red', label='MedEdge = '+str(yS))
29 plt.axvline(x=yM, color='green', label='MaxEdge = '+str(yM))
30 plt.axvline(x=ym, color='black', label='MinEdge = '+str(ym))
```