TIPE - La Ville

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

Valentin FOULON - N° 29836

15 juin 2023

Table des matières

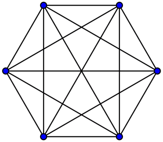
Choix du thème

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.

Définitions

Définition

Poids d'une arête : fonction $d : A \to \mathbb{N}, (u, v) \rightarrowtail d(u, v)$ 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

Objectifs

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.

Paramètres

Les unités sont arbitraires

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

Première approche : algorithme de Dijkstra

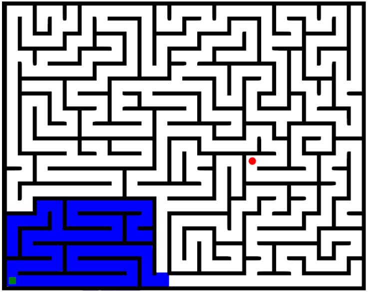
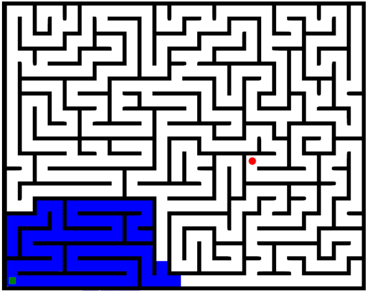
Dijkstra

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.

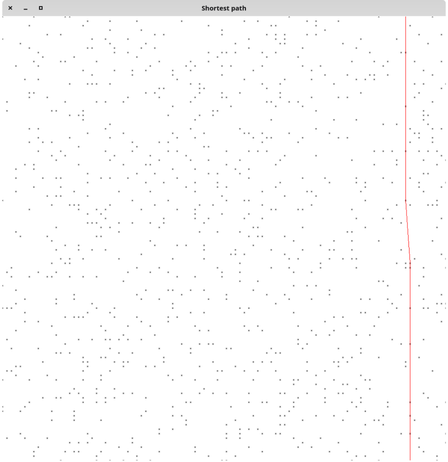
Choix et utilisation de l'algorithme

Dijkstra

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

Production de l'algorithme

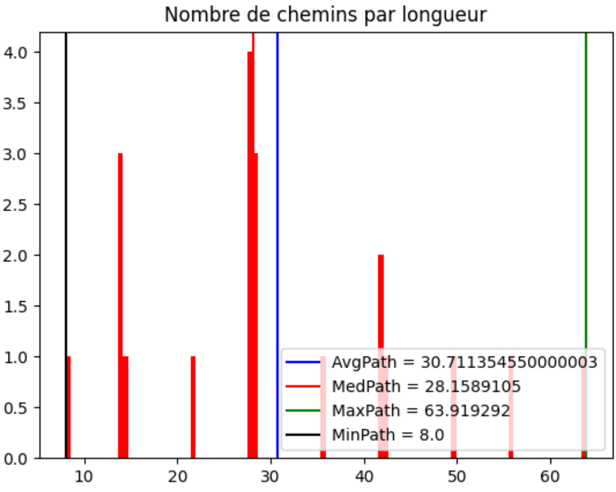


Cet algorithme crée des trajets avec peu de changements de direction.

Mesure des résultats

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



Problèmes

- La solution trouvée est alors optimale en terme de temps de trajet
- La capacité de transport entre les deux points vaut $(|S| - 2)*$ "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

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

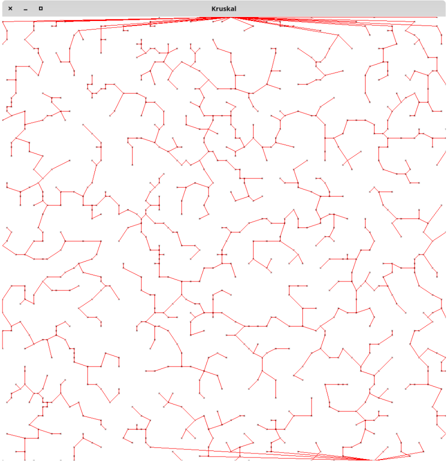
Un arbre couvrant de poids minimum est **un** arbre couvrant dont la somme des poids des arêtes de A' est la plus petite.

Choix et utilisation de l'algorithme

Kruskal

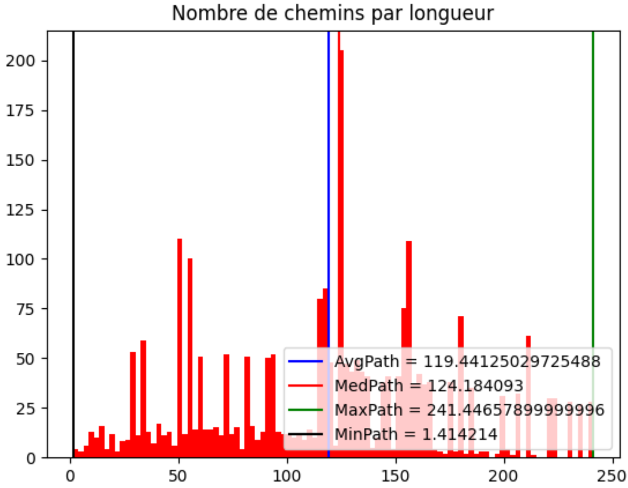
- Complexité : $O(|A| log |A|)$
- Algorithme correct : le sous-graphe renvoyé est un arbre couvrant de poids minimum

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.

Problèmes

- 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.

Idée

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

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

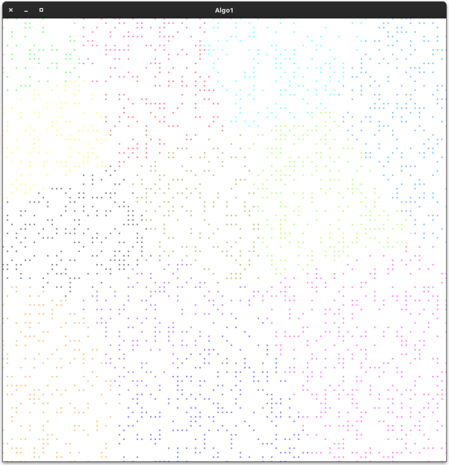
K-moyennes

- 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

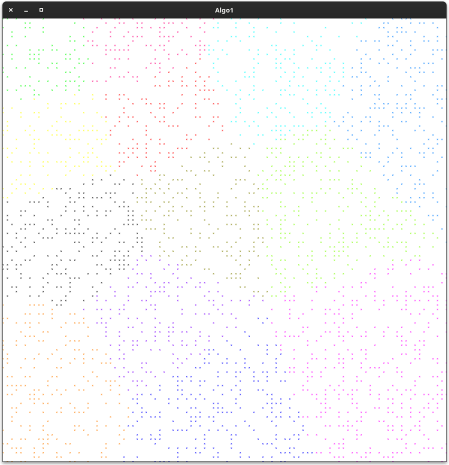
L'algorithme ne converge pas nécessairement vers une solution pour certaines combinaisons de jeux de données, valeurs de K, positions initiales des centroïdes.

Résultat sur un jeu de données aléatoire



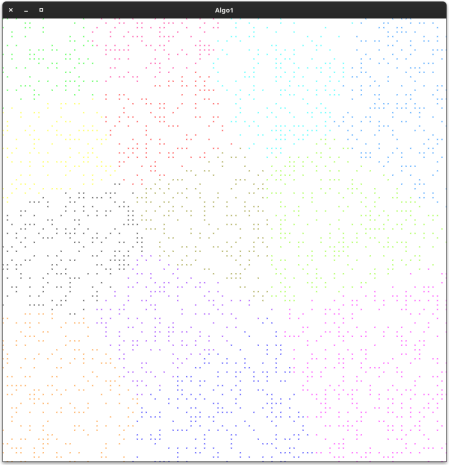
1e itération

Résultat sur un jeu de données aléatoire



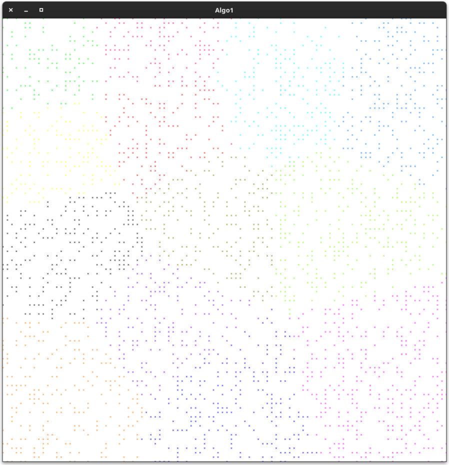
2e itération

Résultat sur un jeu de données aléatoire



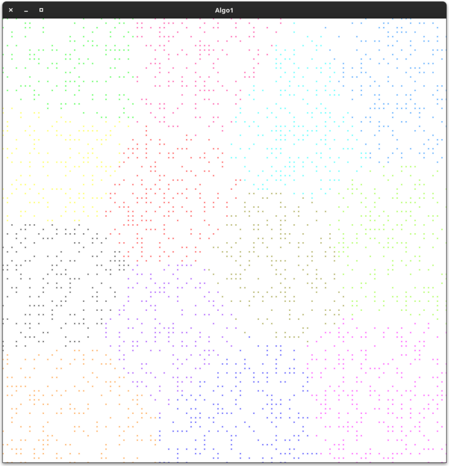
3e itération

Résultat sur un jeu de données aléatoire



4e itération

Résultat sur un jeu de données aléatoire



Après plusieurs itérations, convergence.

Utilisation

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.

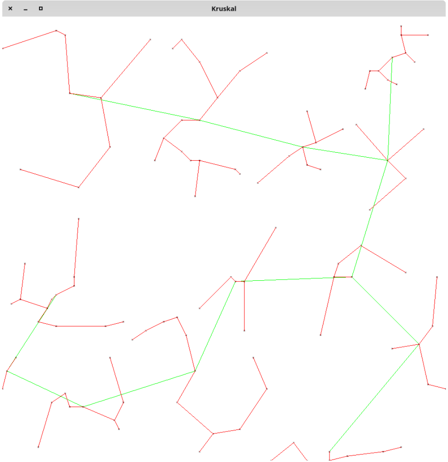
- Si on considère des quartiers d'une ville, on peut relier les clusters directement entre eux
- Si on considère une ville et son agglomération, on relie chaque cluster au cluster de la ville principale

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

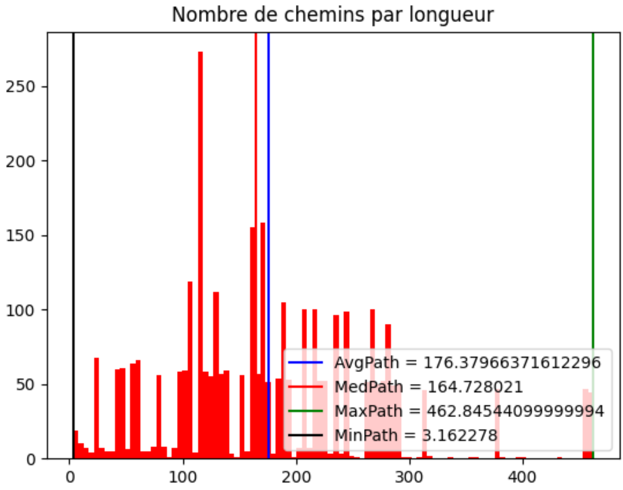
Avantages

- 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

shortestpath.c

```c
void dijkstra(int initial, Stack** stack, bool w) {
  float dist[NMAX];
  int prev[NMAX];
  bool processed[NMAX];
  for (int i = 0; i < NMAX; i++) {
    dist[i] = MAXLENGTH * NMAX;
    prev[i] = 0;
    processed[i] = false;
  }
  dist[initial] = 0;
  for (int i = 0; i < NMAX; i++) {
    int u = minimum(NMAX, dist, processed);
    processed[u] = true;
    for (int v = 0; v < NMAX; v++) {
      if (!processed[v] && dist[u] + graphe[u][v] < dist[v]
        && graphe[u][v] != MAXLENGTH * NMAX && dist[u] !=
        MAXLENGTH * NMAX) {
        dist[v] = dist[u] + graphe[u][v];
        prev[v] = u;
```

```
      }
    }
  }
  for (int i = 0; i < NMAX; i++) {
    int pre = i;
    while (pre != 0) {
      if (i == NMAX - 1) {
        Stack* new = malloc(sizeof(Stack));
        new->val = pre;
        new->next = *stack;
        *stack = new;
      }
      pre = prev[pre];
    }
  }
}
```

kruskal.c

```
Solution* auxiliary(Edgelist* edges, htbl** table) {
    Solution* solution = NULL;
    for (int i = 0; i < edges->s; i++) {
        edge a = edges->list[i];
        htbl* v1 = find(table[a.s1]);
        htbl* v2 = find(table[a.s2]);
        if (v1 != v2) {
            unite(v1, v2);
            Solution* l2 = malloc(sizeof(Solution));
            l2->val = a;
            l2->next = solution;
            solution = l2;
        }
    }
    return solution;
}

Solution* kruskal(Edgelist* edges) {
    int x = edges->s;
```

kruskal.c

```c
    htbl** table = malloc(x * sizeof(htbl*));
    assert(table != NULL);
    for (int i = 0; i < x; i++) {
        table[i] = malloc(sizeof(htbl));
        addElt(i, table[i]);
    }
    quicksort(edges, 0, edges->s - 1);
    return auxiliary(edges, table);
}
```

kavg.c

```c
void classify(Station* stations, StackCouple* sets[], int
    count) {
  for (int i = 0; i < count; i++) {
    double min = INT_MAX;
    int minset = 0;
    for (int j = 0; j < K; j++) {
      double d = distance(stations[i].x, stations[i].y, sets
    [j]->val.x0, sets[j]->val.y0);
      if (d < min) {
        min = d;
        minset = j;
      }
    }
    push_alt(&sets[minset], stations[i].x, stations[i].y, i)
    ;
  }
}

void getSetAvg(StackCouple* sets[]) {
```

kavg.c

```c
  for (int i = 0; i < K; i++) {
    double sumX = 0;
    double sumY = 0;
    int count = 0;
    StackCouple* s = sets[i];
    s = s->next;
    while (s != NULL) {
      sumX += s->val.x0;
      sumY += s->val.y0;
      count++;
      s = s->next;
    }
    assert(count != 0);
    sumX /= count;
    sumY /= count;
    freeSet(sets[i]->next);
    sets[i]->next = NULL;
    sets[i]->val.x0 = sumX;
    sets[i]->val.y0 = sumY;
```

kavg.c

```
    }
}
```

graph.py

```python
def calculatePath(node1, node2, array):
    if node1 == node2:
        return 0
    x = 0
    for elt in array:
        if elt['s1'] == node1:
            if not Vus[elt['s2']]:
                Vus[elt['s2']] = True
                x = calculatePath(elt['s2'], node2, array)
                if x != None:
                    x = x + elt['w']
                    break
        if elt['s2'] == node1:
            if not Vus[elt['s1']]:
                Vus[elt['s1']] = True
                x = calculatePath(elt['s1'], node2, array)
                if x != None:
                    x = x + elt['w']
                    break
```

graph.py

```python
    return x

for edge1 in edges:
    for edge2 in edges:
        if edge1 != edge2:
            Vus = [False] * (max(edges) + 1)
            path = calculatePath(edge1, edge2, X)
            print(edge1, edge2, path)
            if path != None and path != 0:
                paths.append(path)
# print(paths)
```