Algorithmes de Kruskal et de Borůvka

Julie Stempel

Simon Lemal

11 décembre 2020

Table des matières

Introduction

Nous commençons par présenter la structure de donnée Union-Find. Cette structure est exploitée par les deux algorithmes. Nous présentons ensuite les algorithme de Kruskal et de Borůvka. Plus précisément, nous expliquons leur fonctionnement, étudions leur complexité et démontrons leur validité. Nous terminons par quelques exemples et des comparaisons des deux algorithmes sur de gros graphes.

1 Union-Find

Un Union-Find est une structure de données permettant de manipuler une partition d'un ensemble et de facilement la remplacer par une partition plus grossière. Pour implémenter cette structure, on représente chaque classe d'équivalence par un arbre enraciné. Pour stocker cette forêt, il suffit que chaque noeud stocke son noeud père (une racine est son propre père). Chaque racine représente donc une classe d'équivalence.

MakeSet La partition discrète peut être représentée comme suit : chaque sommet est son propre parent. Il y a alors autant de classes d'équivalence que de sommets.

```
class DisjointSet:
  def __init__(self, points):
    self.parent = {p: p for p in points}
    self.depth = {p: 0 for p in points}
```

Union On peut aussi efficacement remplacer deux classes par leur union : la racine de la première classe devient le parent de la racine de la seconde. Pour réduire la complexité, il est commode de minimiser la profondeur des arbres. Pour cela, on fait en sorte que la racine de l'arbre ayant la plus grande profondeur devienne le parent de la racine de l'autre arbre.

```
    def union(self, p, q):
    p, q = self.find(p), self.find(q)
    if p != q:
      if self.depth[p] > self.depth[q]:
        self.parent[q] = p
      elif self.depth[p] < self.depth[q]:
        self.parent[p] = q
      else:
        self.parent[q] = p
        self.depth[p] += 1
```

Find On peut facilement vérifier à quelle classe appartient un sommet : il suffit de trouver son plus lointain ancêtre. Si le sommet est une racine (i.e. est son propre père), il est le représentant d'une classe ; sinon, on recherche la classe à laquelle appartient son père. Pour réduire la complexité, on peut, si le sommet considéré n'est pas une racine, remplacer son père par le plus lointain ancêtre (et faire de même pour tous les ancêtres qu'il a fallut parcourir avant d'arriver au plus lointain). Ainsi, si on réutilise cette procédure sur un des sommets mis à jour, elle s'effectuera en temps constant.

```
6   def find(self, p):
      if self.parent[p] != p:
8       self.parent[p] = self.find(self.parent[p])
      return self.parent[p]
```

Complexité Pour une structure Union-Find correctement implémentée (i.e. qui met à jour les parents lors d'un **Find** et tente de minimiser la profondeur de l'arbre lors d'un **Union**), la complexité amortie des opérations **Union** et **Find** est $O(\log(n))$ où n est la taille de l'ensemble à partitionner.

2 Algorithme de Kruskal

Présentation de l'algorithme L'algorithme de Kruskal commence avec un graphe n'ayant aucune arête et ajoute les arêtes de poids minimal les une après les autres, tant qu'elles ne créent pas de cycle. Plus précisément, l'algorithme commence par ordonner les arêtes par poids croissants (`sorted` ligne 3). Il parcourt ensuite chacune des arêtes (`for` ligne 4). Si l'arête considérée ne crée pas de cycle, il l'ajoute au graphe ; sinon, il passe à l'arête suivante.

Pour vérifier si une arête crée un cycle, il suffit de vérifier si ses deux extrémités appartiennent à la même composante connexe, ce qui peut être fait efficacement à l'aide d'un Union-Find. Initialement, il n'y a aucune arête, donc chaque sommet est une composante (pour cela, on utilise la procédure **MakeSet**, qui correspond à la ligne 2 dans notre implémentation). Lorsque l'on ajoute une arête, on utilise la procédure **Union** pour joindre les deux composantes connexes reliées par la nouvelle arête (`ds.union(u, v)`, ligne 7).

```
def kruskal(g):
2   ds = DisjointSet(g.nodes)
    edges = sorted(g.edges, key=lambda t: g[t[0]][t[1]]["weight"])
4   for u, v in edges:
      if ds.find(u) != ds.find(v):
6       g[u][v]["spanning"] = True
        ds.union(u, v)
8     else:
        g[u][v]["spanning"] = False
```

Remarquons que si le graphe fourni en entrée n'est pas connexe, l'algorithme fournit un arbre couvrant de poids minimal pour chaque composante connexe.

L'algorithme fonctionne également avec un multigraphe, à la différence près que `for u, v in g.edges` doit être remplacé par `for u, v, m in g.edges`, `g[u][v]` par `g[u][v][m]`, etc.

Complexité Dans la suite, E désigne le nombre d'arêtes et V le nombre de sommets. La ligne 2 s'exécutent en $O(V)$ et la ligne 3 en $O(E\log(E))$ (python utilise l'algorithme Timsort). Les lignes de 5 à 9 s'exécutent en $O(\log(V))$ (cf. section 1), donc les lignes de 4 à 9 s'exécutent en $O(E\log(V))$.
Ainsi, l'algorithme s'exécute en $O(E\log(E) + E\log(V))$. Comme $V - 1 \leq E \leq \frac{V(V-1)}{2}$, on a $O(\log(E)) = O(\log(V))$ donc l'algorithme s'exécute en $O(E\log(V))$.

Validité Montrons que si G est un multigraphe connexe et H est le sous-graphe fourni par l'algorithme de Kruskal, alors H est un arbre couvrant de poids minimal. D'une part, si H a deux composantes connexes, G possède une arête de poids minimal les joignant. Mais cette arête appartient forcement à H. D'autre part, si H possède un cycle, il existe une arête de poids maximal dans ce cycle. Mais lorsque cette arête a été considérée, ses deux extrémités appartenaient déjà à la même composante connexe, donc elle a forcement été ignorée par l'algorithme. Ainsi H est nécessairement un arbre couvrant.

On va montrer par récurrence qu'à chaque étape de l'algorithme, il existe un arbre couvrant contenant les arêtes sélectionnées. Initialement, aucune arête n'est sélectionnée donc c'est évident. Soit H' est un sous-graphe de G et T un arbre couvrant de poids minimal ayant H' comme sous-graphe. Soit uv une arête de poids minimal parmi les arêtes telles que $H' + uv$ n'a pas de cycle (i.e. une arête que l'algorithme est susceptible de choisir). Si uv est une arête de T, il est évident qu'il existe un arbre couvrant de poids minimal contenant $H' + uv$.
Sinon, $T + uv$ contient un cycle. Ce cycle contient une arête wt n'appartenant pas à H' (par choix de uv). $H' + wt$ ne contient pas de cycle car c'est un sous-graphe de T donc le poids de wt est plus grand que celui de uv (car uv est choisi minimal). Ainsi, $T + uv - wt$ est un arbre ($T + uv$ a exactement un cycle, contenant wt, donc $T + uv - wt$ n'a pas de cycle) couvrant de poids inférieur à celui de T. Comme T était choisi de poids minimal, $T + uv - wt$ est également de poids minimal. De plus, il contient toutes les arêtes de $H' + uv$.

Ainsi, à chaque étape, il existe un arbre couvrant de poids minimal contenant les arêtes choisies par l'algorithme. Lorsque l'algorithme s'arrête, on sait que le sous-graphe H contenant les arêtes choisies est un arbre couvrant qui est un sous-graphe d'un arbre couvrant de poids minimal. Il est donc lui même un arbre couvrant de poids minimal.

3 Algorithme de Borůvka

Présentation de l'algorithme L'algorithme commence avec un graphe sans arête. À sa première itération, il sélectionne, pour chaque sommet, l'arête de poids minimal. Après cette étape, certains sommets sont connectés entre eux, mais le graphe peut ne pas être connexe. Il va alors considérer chaque composante connexe comme un sommet et réappliquer la même procédure. Autrement dit, il va chercher, pour chaque composante, l'arête de poids minimal qui quitte cette composante. En répétant ces étapes jusqu'à n'avoir plus qu'une seule composante, le graphe obtenu est un arbre couvrant de poids minimal.

Comme pour l'algorithme de Kruskal, nous utilisons un Union-Find pour stocker la donnée des différentes composantes connexes.

En pratique, plutôt que de remplacer chaque composante connexe par un sommet, on parcourt toutes les arêtes et on supprime celles qui ont leurs deux extrémités dans une même composante connexe (c'est ce qui est fait dans le `if` ligne 9). On profite également de cette itération pour trouver l'arête de poids minimal de chaque composante (ce qui est réalisé par le `else` ligne 14 et l'initialisation des mémoires ligne 5). Une fois ces arêtes trouvées, on les ajoute à l'arbre couvrant (`for` ligne 21).

```
def boruvka(g):
  ds = DisjointSet(g.nodes)
  edges = list(g.edges())
  while edges:
    lightest_edges = dict()

    for i in reversed(range(len(edges))):
```

```
8         u, v = edges[i]
          if ds.find(u) == ds.find(v):
10          edges.pop(i)
            if g[u][v].get("spanning") is None:
12            g[u][v]["spanning"] = False

14        else:
            if not lightest_edges.get(ds.find(u)) or \
16              lightest_edges[ds.find(u)][2] > g[u][v]["weight"]:
              lightest_edges[ds.find(u)] = (u, v, g[u][v]["weight"])
18
            if not lightest_edges.get(ds.find(v)) or \
20              lightest_edges[ds.find(v)][2] > g[u][v]["weight"]:
              lightest_edges[ds.find(v)] = (u, v, g[u][v]["weight"])
22
      for u, v, w in lightest_edges.values():
24      g[u][v]["spanning"] = True
        ds.union(u, v)
```

Le `reversed` dans le `for` ligne 7 permet d'éviter les soucis d'indice lorsque l'on supprime un élément de la liste `edge`. Si l'on parcourait les indices dans l'ordre croissant, à chaque fois que l'on supprimerait un élément, tous les éléments suivants seraient décalés.

À nouveau, si le graphe fournit en entrée n'est pas connexe, l'algorithme va fournir un arbre couvrant minimal pour chaque composante.

Comme l'algorithme de Kruskal, l'algorithme de Borůvka reste valide pour un multigraphe, moyennant quelques modifications en python.

Complexité Il est évident que les lignes 2 et 3 s'exécutent en $O(V)$ et $O(E)$ respectivement.

Si le graphe fournit en entrée est connexe, à chaque itération du `while` (ligne 4), chaque composante connexe est connectée à au moins une autre, donc leur nombre diminue au moins de moitié. Ainsi, après k itérations, il reste au plus $\frac{V}{2^k}$ composantes. Comme l'algorithme s'arrête lorsqu'il n'en reste qu'une, on en déduit que le `while` est exécuté au plus $\log_2(V)$ fois (donc $O(\log(V))$ fois). En particulier, l'algorithme se termine toujours.

Le premier `for` (ligne 7) s'exécute $O(E)$ fois. La ligne 9 ne contribue qu'à une terme $O(V)$ sur l'ensemble des itérations `for`. En effet, si on commence par un **Find** qui demande beaucoup d'appels récursifs, beaucoup de parents vont être mis à jour et les **Find** suivants se feront en temps constant. Les lignes de 14 à 21 s'exécutent en temps constant (puisque les parents ont été mis à jours, les **Find** s'exécutent en $O(1)$). Ainsi les lignes de 7 à 21 ont une complexité en $O(E+V) = O(E)$.

Le deuxième `for` (ligne 23) a une complexité en $O(V)$ donc au final, l'algorithme s'exécute en $O(E\log(V))$.

Validité Si G est un multigraphe connexe, H est le sous-graphe renvoyé par l'algorithme. Il est évident que H est un arbre (car l'algorithme ignore les arêtes créant des cycles) connexe (car l'algorithme s'arrête lorsqu'il n'y a plus qu'une composante).

Comme pour l'algorithme de Kruskal, on va montrer qu'à chaque étape, il existe un arbre couvrant contenant les arêtes sélectionnées par l'algorithme. C'est trivial si aucune arête n'est sélectionnée. Soit H' un sous-graphe de G inclus dans un arbre couvrant T de poids minimal. Soit C un composante connexe de H' et uv une arête de poids minimal telle que $u \in C, v \notin C$. Si

uv appartient à T, il est évident que $H' + uv$ est inclus dans un arbre couvrant de poids minimal. Sinon, $T + uv$ contient un cycle. Ce cycle contient une arête wt de T telle que $w \in C$, $t \notin C$. Comme uv est choisi de poids minimal, $T + uv - wt$ est de poids inférieur à celui de T. Cependant, c'est un arbre couvrant (de poids minimal car T est de poids minimal). Il est évident que wt n'appartient pas à H' puisque w et t appartiennent à des composantes connexes différentes. Ainsi $H' + uv$ est inclus dans un arbre couvrant de poids minimal ($T + uv - wt$). La conclusion est alors immédiate.

4 Comparaison

Bien que en théorie, les deux algorithmes aient la même complexité asymptotique, en pratique, l'algorithme de Kruskal est environ trois fois plus rapide que l'algorithme de Borůvka. Il n'est cependant pas flagrant, avec les données dont nous disposons, que le temps de calcul soit en $O(E \log(V))$.

$\#E$	$\#V$				
	10	31	100	316	1000
1000	6.436×10^{-3}	5.018×10^{-3}	4.445×10^{-3}	5.260×10^{-3}	8.089×10^{-3}
	1.172×10^{-2}	1.447×10^{-2}	1.979×10^{-2}	2.337×10^{-2}	2.262×10^{-2}
1300	8.700×10^{-3}	5.329×10^{-3}	6.155×10^{-3}	6.477×10^{-3}	8.890×10^{-3}
	1.593×10^{-2}	1.941×10^{-2}	2.873×10^{-2}	3.838×10^{-2}	2.984×10^{-2}
1600	9.254×10^{-3}	5.663×10^{-3}	6.538×10^{-3}	8.126×10^{-3}	1.194×10^{-2}
	1.939×10^{-2}	2.344×10^{-2}	3.355×10^{-2}	4.943×10^{-2}	3.884×10^{-2}
1900	1.008×10^{-2}	6.873×10^{-3}	9.628×10^{-3}	1.004×10^{-2}	1.207×10^{-2}
	2.222×10^{-2}	2.830×10^{-2}	4.051×10^{-2}	4.660×10^{-2}	4.697×10^{-2}
2200	1.288×10^{-2}	8.479×10^{-3}	8.894×10^{-3}	1.051×10^{-2}	1.538×10^{-2}
	2.618×10^{-2}	3.638×10^{-2}	4.690×10^{-2}	5.785×10^{-2}	6.034×10^{-2}
2500	1.256×10^{-2}	8.811×10^{-3}	1.194×10^{-2}	1.341×10^{-2}	1.581×10^{-2}
	2.742×10^{-2}	4.136×10^{-2}	5.701×10^{-2}	6.872×10^{-2}	7.965×10^{-2}
2800	1.325×10^{-2}	1.013×10^{-2}	1.174×10^{-2}	1.426×10^{-2}	1.857×10^{-2}
	3.146×10^{-2}	5.032×10^{-2}	6.264×10^{-2}	7.678×10^{-2}	9.015×10^{-2}
3100	1.409×10^{-2}	1.335×10^{-2}	2.068×10^{-2}	2.451×10^{-2}	3.146×10^{-2}
	3.438×10^{-2}	5.233×10^{-2}	1.229×10^{-1}	1.333×10^{-1}	9.924×10^{-2}
3400	1.490×10^{-2}	1.487×10^{-2}	1.660×10^{-2}	1.690×10^{-2}	2.704×10^{-2}
	3.317×10^{-2}	5.014×10^{-2}	7.361×10^{-2}	9.501×10^{-2}	1.079×10^{-1}
3700	1.582×10^{-2}	1.415×10^{-2}	1.660×10^{-2}	1.942×10^{-2}	2.074×10^{-2}
	3.962×10^{-2}	6.209×10^{-2}	8.572×10^{-2}	1.067×10^{-1}	1.233×10^{-1}
4000	1.809×10^{-2}	1.496×10^{-2}	2.870×10^{-2}	3.145×10^{-2}	4.098×10^{-2}
	4.325×10^{-2}	1.255×10^{-1}	2.267×10^{-1}	1.657×10^{-1}	1.942×10^{-1}

TABLE 1 – Temps nécessaire aux algorithmes de Kruskal et de Borůvka, en secondes

Cependant, le calcul de la complexité a été effectué dans le cas d'un graphe simple. Si on considère un multigraphe, rien ne change pour l'algorithme de Borůvka qui reste en $O(E \log(V))$. Par contre, l'hypothèse $E \leq \frac{V(V-1)}{2}$ faite lors de l'étude de la complexité de l'algorithme de Kruskal n'est plus valide. Ainsi, dans le cas des multigraphes, l'algorithme de Kruskal a une complexité asymptotique en $O(E \log(E))$, ce qui est moins efficace que l'algorithme de Borůvka.

Ainsi, bien que l'algorithme de Kruskal semble bien plus efficace, lorsque l'on considère

des multigraphes pour lesquels $\log(E) \gg \log(V)$, l'algorithme de Borůvka est préférable. Nous n'avons pas réussi à mettre ce comportement en évidence en pratique.

Références

[1] *Amortized analysis*. URL : https://en.wikipedia.org/wiki/Amortized_analysis (visité le 10/11/2020).

[2] *Boruvka's algorithm*. URL : https://en.wikipedia.org/wiki/Bor%C5%AFvka's_algorithm (visité le 25/10/2020).

[3] *Boruvka's algorithm | Greedy Algo-9*. URL : https://www.geeksforgeeks.org/boruvkas-algorithm-greedy-algo-9/ (visité le 10/11/2020).

[4] *Disjoint-set data structure*. URL : https://en.wikipedia.org/wiki/Disjoint-set_data_structure (visité le 10/11/2020).

[5] *Kruskal's algorithm*. URL : https://en.wikipedia.org/wiki/Kruskal's_algorithm (visité le 25/10/2020).

[6] *Kruskal's Minimum Spanning Tree Algorithm | Greedy Algo-2*. URL : https://www.geeksforgeeks.org/kruskals-minimum-spanning-tree-algorithm-greedy-algo-2/ (visité le 20/11/2020).

[7] Edmond La Chance. « Algorithmes pour le problème de l'arbre couvrant minimal ». Mémoire. Université du Québec à Chicoutimi, 2014. URL : https://constellation.uqac.ca/2899/.

[8] *Minimum spanning tree - Kruskal's algorithm*. URL : https://cp-algorithms.com/graph/mst_kruskal.html (visité le 10/11/2020).

[9] *NetworkX 2.5 documentation*. URL : https://networkx.org/documentation/stable/ (visité le 11/11/2020).

[10] *Python 3.8.6 Documentation*. URL : https://docs.python.org/3.8/ (visité le 10/11/2020).

[11] *Timsort*. URL : https://en.wikipedia.org/wiki/Timsort (visité le 10/11/2020).

[12] *Union-find*. URL : https://fr.wikipedia.org/wiki/Union-find (visité le 10/11/2020).