Travaux Pratiques : Arbre couvrant de poids minimum
Implémentation avec la structure Union-Find

Olivier Bodini

November 12, 2024

Contents

1 Introduction

Ce TP a pour objectif d'implémenter un arbre couvrant de poids minimum (ou Minimum Spanning Tree - MST) à l'aide de la structure de données **Union-Find**. Cette structure est efficace pour gérer les sous-ensembles dans des algorithmes de graphes, comme celui de *Kruskal*, qui construit un MST en ajoutant les arêtes les moins coûteuses tout en évitant les cycles.

2 Objectifs du TP

- Comprendre le concept d'un arbre couvrant de poids minimum.
- Implémenter la structure de données Union-Find (ou Find-Union).
- Appliquer l'algorithme de Kruskal pour construire un MST en utilisant Union-Find.

3 Notions théoriques

3.1 Arbre couvrant de poids minimum

Un arbre couvrant de poids minimum d'un graphe est un sous-graphe connexe acyclique (un arbre) qui relie tous les sommets du graphe avec un poids total minimum. Le poids est la somme des coûts des arêtes qui composent l'arbre.

3.2 Structure de données Union-Find

Union-Find est une structure de données pour gérer des ensembles disjoints. Elle prend en charge deux opérations principales :

- **Find**: Trouver le représentant (ou chef) d'un ensemble.
- **Union**: Fusionner deux ensembles.

Pour rendre ces opérations plus efficaces, on utilise les techniques de *compression de chemin* et *union par rang*.

4 Algorithme de Kruskal

L'algorithme de Kruskal est une approche par arêtes croissantes pour trouver un MST. Les étapes sont les suivantes :

1. Trier toutes les arêtes du graphe par poids croissant.
2. Initialiser une structure Union-Find pour gérer les sous-ensembles.
3. Parcourir les arêtes triées. Pour chaque arête, si les deux extrémités sont dans des ensembles différents (vérifié avec `Find`), on les relie dans le MST et on fusionne les ensembles (avec `Union`).
4. Répéter jusqu'à ce que le MST ait $n - 1$ arêtes.

5 Mise en œuvre

5.1 Structure Union-Find en pseudocode

Voici le pseudocode de la structure Union-Find :

Algorithm 1 Union-Find avec compression de chemin et union par rang

```
Fonction Find(x)
if x est son propre parent then
  return x
else
  return parent[x] = Find(parent[x])
end if
Procédure Union(x, y)
rootX = Find(x)
rootY = Find(y)
if rootX ≠ rootY then
  if rank[rootX] > rank[rootY] then
    parent[rootY] = rootX
  else if rank[rootX] < rank[rootY] then
    parent[rootX] = rootY
  else
    parent[rootY] = rootX
    rank[rootX] = rank[rootX] + 1
  end if
end if
```

5.2 Implémentation de l'algorithme de Kruskal

Listing 1: Implémentation de Kruskal en Python

```
class UnionFind:
    def __init__(self, n):
        self.parent = list(range(n))
        self.rank = [0] * n

    def find(self, x):
        if self.parent[x] != x:
            self.parent[x] = self.find(self.parent[x])
# Compression de chemin
        return self.parent[x]

    def union(self, x, y):
        rootX = self.find(x)
        rootY = self.find(y)
        if rootX != rootY:
            if self.rank[rootX] > self.rank[rootY]:
                self.parent[rootY] = rootX
            elif self.rank[rootX] < self.rank[rootY]:
                self.parent[rootX] = rootY
            else:
                self.parent[rootY] = rootX
                self.rank[rootX] += 1

def kruskal(n, edges):
    mst = []
    uf = UnionFind(n)
    edges.sort(key=lambda x: x[2])  # Tri des aretes par poids
    for u, v, weight in edges:
        if uf.find(u) != uf.find(v):
            uf.union(u, v)
            mst.append((u, v, weight))
    return mst
```

6 Exercices

1. **Exercice 1** : Implémentez la structure de données Union-Find en utilisant le pseudocode fourni.

2. **Exercice 2** : Testez votre implémentation en utilisant l'algorithme de

Kruskal sur différents graphes.

3. **Exercice 3** : Comparez les performances de votre solution avec et sans compression de chemin.

4. **Exercice 4** : Justifiez pourquoi l'algorithme de Kruskal produit toujours un arbre couvrant de poids minimum.