TD : structure Union-Find et algorithme de Kruskal

Cette séance a pour objectif d'élaborer un algorithme pour calculer un arbre couvrant de poids minimal dans un graphe pondéré. On se servira pour cela d'une structure de donnée abstraite nommée *Union-Find* et qui est utilisée pour gérer les partitions d'un ensemble fini.

1 Structure de donnée Union-Find

La structure de donnée abstraite *Union-Find* permet de manipuler les partitions d'un ensemble fini de n éléments qu'on identifiera à $\{0, \dots, n-1\}$. Chaque ensemble de la partition traitée a un unique représentant. Les opérations possibles sont :

- `make_partition : int -> partition`
  `make_partition(n)` créé la partition de $\{0, \dots, n-1\}$ formée de singletons.
- `find : partition -> int -> int`
  `find(p, x)` renvoie le représentant de x
- `union : partition -> int -> int -> unit`
  `union(p, x, y)` fusionne dans la partition les deux parties contenant x et y respectivement, ne change pas la partition si x et y sont dans la même partie.

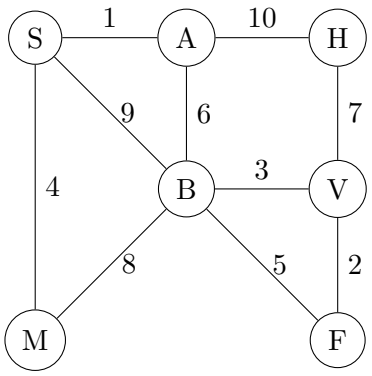
1. Proposer une structure de donnée concrète implémentant *Union-Find*. On se servira naïvement d'un tableau T dans lequel $T[i]$ contient le représentant de la partie de i.
2. Vérifiez sur un exemple choisi le bon fonctionnement de votre implémentation.
3. Étudier la complexité dans le pire cas de chacune des opérations.

2 Algorithme de Kruskal

Vous devenez ministre des transports dans un pays complètement dépourvu de réseau ferroviaire. Vous ordonnez une étude des coûts de construction des voies ferrées possibles entre les 7 principales villes

- Solitude (S)
- Aubétoile (A)
- Fordhiver (H)
- Blancherive (B)
- Vendeaume (V)
- Markarth (M)
- Faillaise (F)

et on vous apporte les résultats :



On vous confie la mission de connecter toutes ces villes au réseau ferroviaire mais hélas c'est la crise et il va falloir faire au mieux question trésorerie. Il s'agit donc de choisir un sous-ensemble d'arêtes de ce graphe tel que toutes les villes soient connectées et de coût minimal.

1. Représenter en Caml le graphe pondéré G dessiné ci-dessus, on choisira une représentation adaptée.

2.1 Étude de l'Algorithme

L'algorithme de Kruskal fonctionne ainsi :
- On commence par trier toutes les arêtes par poids croissant.
- On construit progressivement l'arbre couvrant T en y ajoutant les arêtes dans l'ordre de poids croissant. On n'ajoute pas d'arête si celle-ci créé un cycle.
- Pour détecter si l'ajout de (x, y) créé un cycle, il suffit de vérifier si x et y sont dans la même composante connexe de l'actuel T, pour cela on utilisera une structure *Union-Find* pour gérer les composantes connexes.
- Lorsque toutes les arêtes ont été considérées on retourne T.

1. Appliquer cet algorithme à notre exemple en gérant *à vue* les composantes connexes.

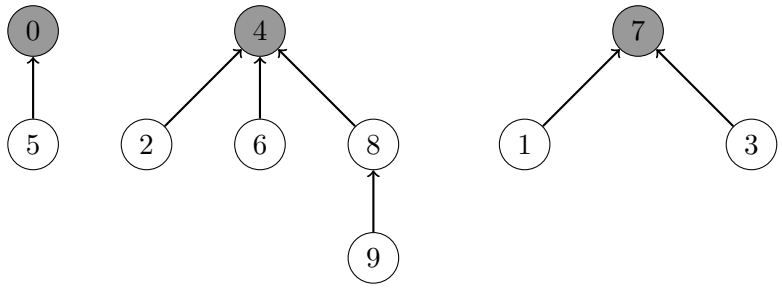
2.2 Programmation

1. Écrire une fonction `liste_aretes` qui prend en entrée un graphe et qui renvoie la liste des arêtes pondérés sous forme d'une liste de triplets (x, y, p_{xy}). On prendra à garde à éliminer les doublons (le graphe est non orienté).
2. Écrire la fonction `tri_aretes` qui trie la liste des arêtes par poids croissants. On utilisera un *tri fusion*.
3. Écrire en Caml l'algorithme de Kruskal et vérifiez son bon fonctionnement.

4. Indiquez la complexité de cet algorithme.

3 Amélioration de la structure Union-Find à l'aide de fôrets

Les graphes permettent une implémentation plus efficace de la structure *Union-Find*. On représente la partition par un graphe orienté où chaque composante connexe représente une partie sous forme d'un arbre. La racine de l'arbre est le représentant de la partie concernée. L'orientation des arcs permet de remonter à la racine depuis n'importe quel nœud.



Ainsi ce graphe représente la partition

$$\{\{0,5\},\{2,4,6,8,9\},\{1,3,7\}\}$$

avec pour représentants 0, 4 et 7.

1. Dans ce graphe quels sont les degrés sortants possibles ? En déduire une représentation adaptée à ce type de graphe.
2. Proposer une structure de donnée concrète implémentant *Union-Find*.
3. Vérifiez son bon fonctionnement à l'aide du même exemple que pour l'implémentation naïve.
4. Étudier la complexité dans le pire cas de chacune des opérations.
5. On peut rendre cette structure plus efficace en aplanissant les arbres. En effet lorsqu'on utilise `find` on peut faire pointer tous les nœuds rencontrés vers la racine. Réaliser cette optimisation.

Remarque : cette structure concrète est très élégante mais l'étude de sa complexité est ardue.