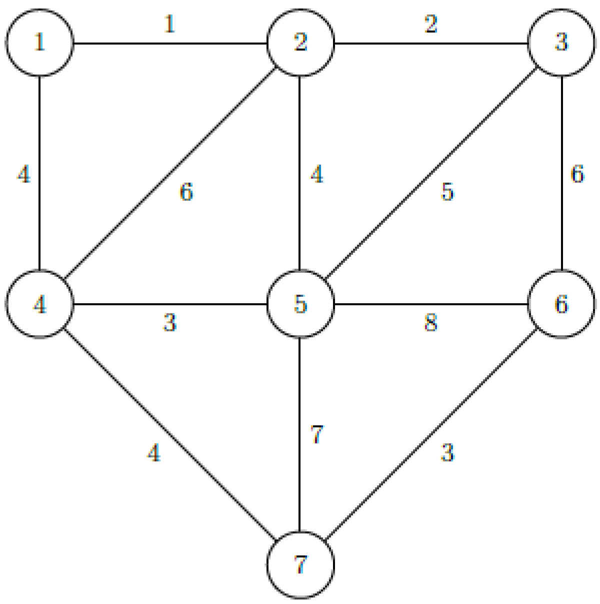
Introduction à l'algorithmique **Automne 2017**

IFT2125-6001 TA: Maëlle Zimmermann

Démonstration 6

1

Question: Exécuter l'algorithme de Prim sur le graphe suivant:



Solution: En exécutant l'algorithme, nous obtenons:

Itération	(u, v)	B
0	-	$\{1\}$
1	$(1, 2)$	$\{1, 2\}$
2	$(2, 3)$	$\{1, 2, 3\}$
3	$(1, 4)$	$\{1, 2, 3, 4\}$
4	$(4, 5)$	$\{1, 2, 3, 4, 5\}$
5	$(4, 7)$	$\{1, 2, 3, 4, 5, 7\}$
6	$(7, 6)$	$\{1, 2, 3, 4, 5, 7, 6\}$

Ainsi l'arbre sous-tendant de poids minimal est de poids 17.

2

Question: Montrer que l'algorithme de Prim peut, comme celui de Kruskal, être implémenté en utilisant des monceaux. Montrer qu'il prend alors un temps dans $\Theta(a \log n)$.

Solution: Considérons d'abord l'algorithme de Prim implémenté naïvement sans monceaux:

```
def Prim(V, E):
    def poids((u, v, c)): return c
    F = sorted(E, key = poids)
    T = []
    B = set(V[:1])

    # tant que tous les sommets ne sont pas couverts
    while len(B) != len(V)
        for (u, v, _) in F:
            if (u in B) != (v in B):
                break
        T.append((u, v))
        B.update([u, v])
    return T
```

Dans le pire cas, cet algorithme prend un temps dans $O(an)$ (boucle while exécutée exactement $n-1$ fois et parcours de F en entier qui contient a arêtes). Mais il existe une meilleure implémentation. Voici l'algorithme de Prim implémenté avec des monceaux:

```
def Prim_heap(V, E):
    if len(V) == 0:
        return []
    x = V[0] # sommet actuel
```

```python
T = [] # arbre partiel minimum
B = set(V[:1]) # sommets couverts par T
H = [] # monceau vide

# construire voisins des sommets
voisins = [[] for v in V]
for (u, v, c) in E:
   voisins[u].append((v, c))
   voisins[v].append((u, c))

# calcul de l'arbre
while len(T) < len(V) - 1:
# met tous les voisins de x dans le monceau
   for (y, c) in voisins[x]:
      heappush(H, (c, (x, y)))

   # retire l'arete au poids c minimum
   (c, (u, v)) = heappop(H)

   # continue de retirer jusqu'a ce que l'arete traverse B et V\B
   while (u in B) == (v in B):
      (c, (u, v)) = heappop(H)

   # update x, T et B
   x = u if u not in B else v
   T.append((u, v))
   B.add(x)
return T
```

Dans un monceau, les opérations push et pop prennent un temps dans $\Theta(\log k)$ et $\Theta(\log k)$ respectivement (où k est le nombre d'éléments dans le monceau). Posons $n = |V|$ et $a = |E|$, alors:

- La boucle qui construit les voisins est exécutée a fois donc prend un temps dans $\Theta(a)$.
- Chaque arête (u, v) est ajoutée au plus 2 fois dans le monceau, soit via u soit via v. Il y a donc au plus $2a$ opérations push qui nécessitent chacune un temps de $\Theta(\log a)$. Donc le temps total des ajouts est dans $\Theta(a \log a)$.
- Similairement, il y a au plus $2a$ opérations pop qui nécessitent chacune un temps de $\Theta(\log a)$. Donc le temps total des retraits est dans $\Theta(a \log a)$.
- Les opérations append et add sont exécutées autant de fois qu'il y a d'itérations de l'algorithme, donc au plus $n - 1$ fois.

Le temps d'exécution de l'algorithme est donc dans $\Theta(a \log a)$, ce qui s'écrit aussi $\Theta(a \log n)$ pour les mêmes raisons que lors de l'analyse de l'algorithme de Kruskal.