Introduction à l'algorithmique **Automne 2017**

IFT2125-6001 TA: Maëlle Zimmermann

Démonstration 11

1

Question: Donner un algorithme qui calcule la distance d'édition d, aussi appelée distance de Levenshtein, entre deux mots. Plus précisément, soient u et v deux mots, alors $d(u, v)$ est le nombre minimal de lettres qu'il faut supprimer, insérer ou substituer pour passer de u à v.

Par exemple, on peut passer de *table* à *stage* en 3 opérations:

$$\begin{aligned} table &\rightarrow tale && \text{(supprimer } b\text{)} \\ &\rightarrow stale && \text{(insérer } s\text{)} \\ &\rightarrow stage && \text{(substituer } l \text{ pour } g\text{)} \end{aligned}$$

Solution: Nous construisons un tableau D de taille $|u| + 1 \times |v| + 1$ tel que $D[i][j]$ est la distance entre les sous-mots $u[1...i]$ et $v[1...j]$.

Notons d'abord que $D[i][0] = i$ pour tout $0 \leq i \leq |u|$, et $D[0][j] = j$ pour tout $0 \leq j \leq |v|$. En effet, pour un mot vide ϵ, on a $d(x, \epsilon) = d(\epsilon, x) = |x|$ car l'édition minimale consiste dans le premier cas à effacer toutes les lettres de x ($|x|$ suppressions) et dans le deuxième cas à ajouter toutes les lettres de x au mot vide ($|x|$ insertions). Plus généralement, pour passer d'un mot $u[1...i]$ à un mot $v[1...j]$, il y a trois possibilités:

- Supprimer $u[i]$ puis passer de $u[1...i-1]$ à $v[1...j]$
- Passer de $u[1...i]$ à $v[1...j-1]$ puis insérer $v[j]$
- Passer de $u[1...i-1]$ à $v[1...j-1]$ puis substituer $u[i]$ pour $v[j]$ (si ces lettres sont différentes)

Cela donne la règle:

$$D[i][j] = \min(\underbrace{D[i-1][j]+1}_{\text{suppression}}, \underbrace{D[i][j-1]+1}_{\text{insertion}}, \underbrace{D[i-1][j-1]+b_{i,j}}_{\text{substitution}})$$

où $b_{i,j} = 0$ si $u[i] = v[j]$ et 1 sinon.

Voici une implémentation de l'algorithme, qui prend un temps dans $\Theta(|u||v|)$:

```
def d_tab(u, v):
   D = [[0]*(len(v) + 1) for _ in range(len(u) + 1)]

   for i in range(len(u)+1):
      D[i][0] = i

   for j in range(len(v)+1):
      D[0][j] = j

   for i in range(1, len(u) + 1):
      for j in range(1, len(v) + 1):
         b = 1 if u[i] != v[j] else 0
         D[i][j] = min(D[i-1][j] + 1, D[i][j-1] + 1, D[i-1][j-1] + b)
   return D

def d(u, v):
   return d_tab(u, v)[-1][-1]
```

2

Question: Soient $u = AGGAGGA$ et $v = AAGCTAAG$. Identifier tous les alignements entre u et v de coût $d(u, v)$, c'est-à-dire tous les alignements optimaux.

Solution: Le tableau des distances est:

	ϵ	A	A	G	C	T	A	A	G
ϵ	0	1	2	3	4	5	6	7	8
A	1	0	1	2	3	4	5	6	7
G	2	1	1	1	2	3	4	5	6
G	3	2	3	1	2	3	4	5	5
A	4	3	2	2	2	3	3	4	5
G	5	4	3	2	3	3	4	4	4
G	6	5	4	3	3	4	4	5	4
A	7	6	5	4	4	4	4	4	5

Ainsi la distance de Levenshtein entre les deux mots est de 5. pour trouver quel alignement et donc quelle séquence d'opérations donne ce coût, il faut remonter le tableau depuis la case `T[-1][-1]` en suivant les règles de l'algorithme.

	ε	A	A	G	C	T	A	A	G
ε	0	1	2	3	4	5	6	7	8
A	1	0	1	2	3	4	5	6	7
G	2	1	1	1	2	3	4	5	6
G	3	2	2	1	2	3	4	5	5
A	4	3	2	2	2	3	3	4	5
G	5	4	3	2	3	3	4	4	4
G	6	5	4	3	3	4	4	5	4
A	7	6	5	4	4	4	4	4	5

Le chemin du haut correspond à l'alignement de séquence suivant:

A	A	G	C	T	A	A	G	-
-	A	G	-	G	A	G	G	A

ce qui correspond à faire deux insertions (A et C), deux substitutions (G pour T et G pour A) et une suppression (A). En tout nous obtenons 6 chemins, et les 5 autres correspondent aux alignements de séquences suivants:

A	A	G	C	T	A	A	G	
A	G	G	A	G	G	A	-	

A	A	G	C	T	A	A	G	-
A	G	G	-	-	A	G	G	A

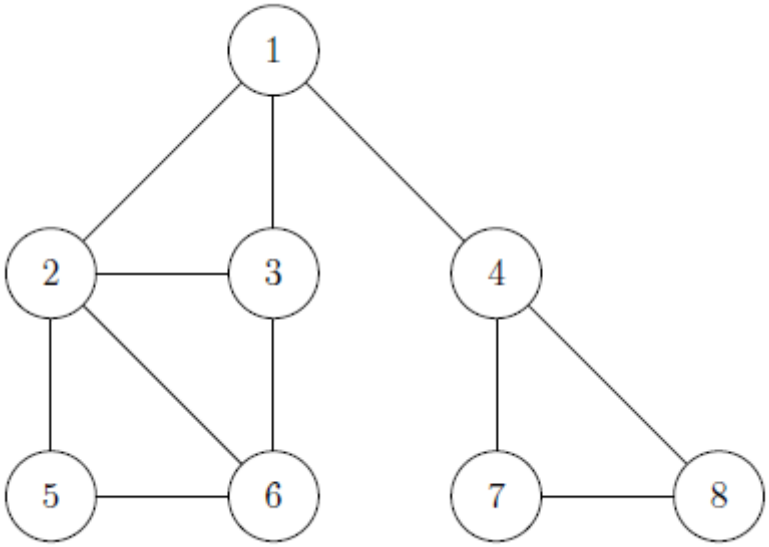
A	A	G	C	T	A	A	G	-
A	-	G	-	G	A	G	G	A

A	A	G	C	T	A	A	G	-
-	A	G	G	-	A	G	G	A

A	A	G	C	T	A	A	G	-
A	-	G	G	-	A	G	G	A

3

Question: Faire un parcours en profondeur sur le graphe suivant et trouver ses points d'articulation:



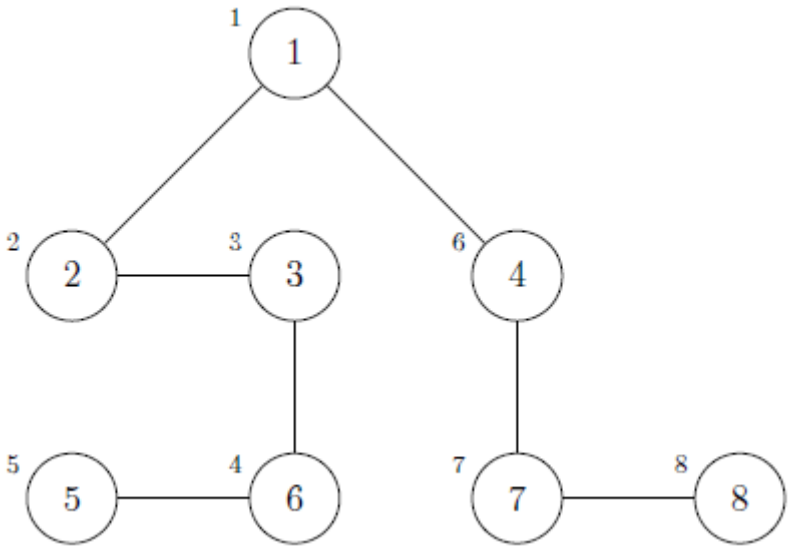
Solution: Voici une implémentation du parcours en profondeur avec numérotation `prenum` des sommets:

```
def dfs(V, E):
   # construire voisins des sommets
   voisins = {v:[] for v in V}
   for (u, v) in E:
      voisins[u].append(v)
      voisins[v].append(u)
   prenum = {v: 0 for v in V}
   i = 0
   # lancer la fouille en profondeur sur chaque sommet non visite
   for v in V:
      if prenum[v] == 0:
         i = _dfs(v, i, voisins, prenum)
   return prenum

def _dfs(v, i, voisins, prenum):
   i += 1
   prenum[v] = i
   for w in voisins[v]:
      if prenum[w] == 0:
```

```
            i = _dfs(w, i, voisins, prenum)
    return i
```

Le parcours en profondeur donne l'ordre de visite `prenum` des sommets: $1, 2, 3, 6, 5, 4, 7, 8$. Graphiquement, nous obtenons l'arbre sous-tendant suivant:



Rappelons qu'un sommet v est un point d'articulation d'un graphe connexe non dirigé si le graphe obtenu en retirant v et ses arêtes incidentes n'est pas connexe.

Ayant fait un parcours en profondeur sur un graphe G, nous pouvons trouver ses points d'articulation à partir de l'ordre `prenum` et de l'arbre sous-tendant T. Pour chaque sommet v, nous calculons une nouvelle valeur `highest[v]`. Afin de calculer cette valeur nous faisons un parcours post-ordre de T et calculons `highest[v]` comme étant le minimum parmi

(a) `prenum[v]`,

(b) `prenum[w]` pour toute arête $(v, w) \in E(G)$ telle que $(v, w) \notin E(T)$,

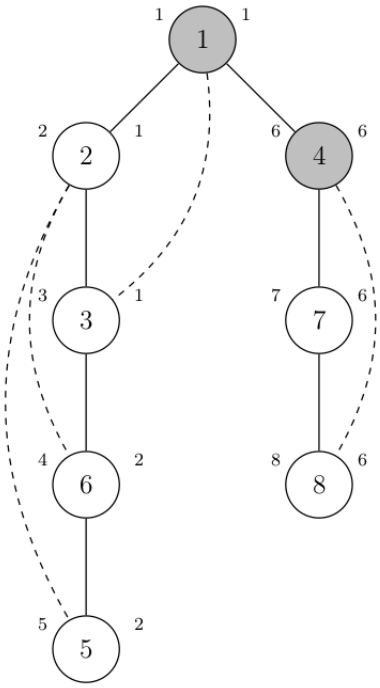
(c) `highest[w]` pour tout enfant w de v.

Soit v un sommet, alors v est un point d'articulation de G si et seulement si

- v est la racine de T et possède au moins 2 enfants, ou bien

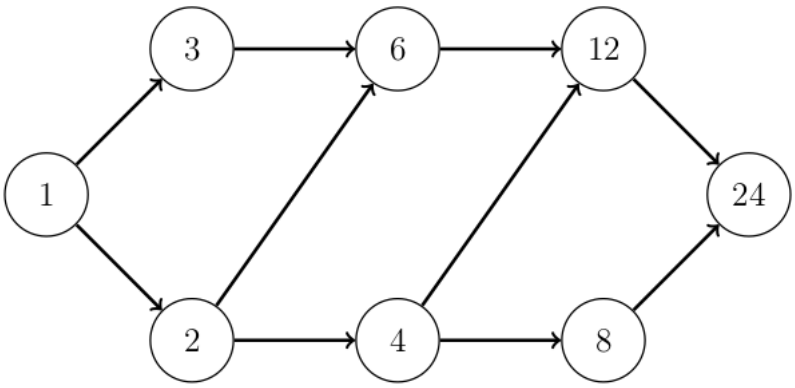
- v n'est pas la racine de T et possède un enfant w tel que `highest[w]` $\geq$ `prenum[v]`.

Les points d'articulation du graphe sont donc les sommets 1 et 4. Voici, graphiquement, l'arbre sous-tendant après le calcul de `highest`.



4

Question: Faire le tri topologique des sommets du graphe orienté et acyclique suivant.



Solution: Il suffit d'effectuer un parcours en profondeur du graphe et de numéroter chaque sommet à la toute fin de son exploration, puis d'inverser l'ordre obtenu.

L'ordre `postnum` obtenu après le parcours en profondeur est: 24, 8, 12, 4, 6, 2, 3, 1.

Inverser cet ordre trie les sommets topologiquement: 1, 3, 2, 6, 4, 12, 8, 24.