

Algorithmique

Notes de cours

Université Paris Diderot

Copyright © 2015 Sophie Laplante (Professeur), notes de Maxime Gourgoulhon (étudiant)

PUBLISHED BY CC

INFORMATIQUE.UNIV-PARIS-DIDEROT.FR

Licensed under the Creative Commons Attribution-NonCommercial 3.0 Unported License (the “License”). You may not use this file except in compliance with the License. You may obtain a copy of the License at <http://creativecommons.org/licenses/by-nc/3.0>. Unless required by applicable law or agreed to in writing, software distributed under the License is distributed on an “AS IS” BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the License for the specific language governing permissions and limitations under the License.

First printing, November 2015

Table des matières

	Union-Find
1	Graphes
1.1	Définition
1.2	Représentations
1.2.1	Matrice d'adjacence
1.2.2	Liste d'adjacence
1.3	Chemins
1.4	Cycle
1.5	Connexe
1.6	Arbres
1.7	Arbre couvrants
1.8	Graphes valués
1.9	Algorithme de Kruskal (1956)
2	Cycles et composantes connexes
2.1	Problèmes
2.2	Application : Kruskal
3	L'algorithme : Union-Find
3.1	Définition
3.2	Exemple
3.3	Applications

3.4	Idée de l'algorithme	14
3.5	Retour sur Kruskal	14
3.6	Implémentation en Python	14
3.7	Complexité	15
3.8	<i>UNION</i> : Maîtriser la hauteur des arbres	15
3.9	<i>FIND</i> : Compresser les chemins	15

II

Parcours d'arbres

4	Arbres binaires de recherche ABR	19
4.1	Complexité	19
4.2	Arbres binaires de recherche	19
4.3	Recherche dans un ABR	20
4.3.1	Preuve par récurrence sur $ A $	20
4.4	Insertion dans un ABR	21
4.4.1	Algorithme	21
4.4.2	Coût moyen d'insertion de n éléments dans un ABR vide	21

III

Parcours de graphes

IV

Algorithmes de graphes

5	Algorithme de Kruskal	27
5.1	Implémentation de l'algorithme de Kruskal	27
5.2	Correction de l'algorithme de Kruskal	27
5.2.1	Lemme d'optimalité	28
6	Algorithme de Prim	29
6.1	Implémentation de l'algorithme de Prim (version optimisée)	29
6.2	Complexité de l'algorithme de Prim	30
7	Les plus courts chemins : Dijkstra	31
7.1	Algorithme de Dijkstra	31
7.2	Preuve de l'algorithme de Dijkstra	32
8	Algorithme de Bellman-Ford	35
8.1	Idée	35
8.2	Implémentation en Python	36
8.3	Preuve	36
	Index	37

Union-Find

1	Graphes	7
1.1	Définition		
1.2	Représentations		
1.3	Chemins		
1.4	Cycle		
1.5	Connexe		
1.6	Arbres		
1.7	Arbre couvrants		
1.8	Graphes valués		
1.9	Algorithme de Kruskal (1956)		
2	Cycles et composantes connexes	11
2.1	Problèmes		
2.2	Application : Kruskal		
3	L'algorithme : Union-Find	13
3.1	Définition		
3.2	Exemple		
3.3	Applications		
3.4	Idée de l'algorithme		
3.5	Retour sur Kruskal		
3.6	Implémentation en Python		
3.7	Complexité		
3.8	<i>UNION</i> : Maîtriser la hauteur des arbres		
3.9	<i>FIND</i> : Compresser les chemins		

1. Graphes

1.1 Définition

Définition 1.1.1 — Graphe. Un graphe est constitué d'un ensemble de sommets S d'un ensemble d'arêtes $A \subseteq S \times S$ reliant les sommets. Les arêtes sont orientées (flèche) ou non orientées (segment).

Notation 1.1. On note un graphe $G = (S, A)$ en français, $G = (V, E)$ en anglais.

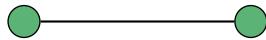


FIGURE 1.1 – Graphe non orienté



FIGURE 1.2 – Graphe orienté

Les sommets peuvent représenter des villes, des stations de métro, des sites sur des cartes géographiques, etc. Les arêtes peuvent symboliser des routes, des câbles, des liens, etc. Un graphe peut ainsi représenter des cartes géographiques, le web, les réseaux sociaux, des modèles 3D (animations), modèles physiques d'interactions...

1.2 Représentations

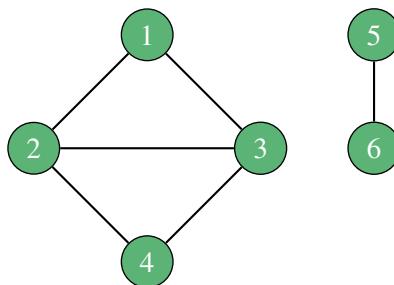


FIGURE 1.3 – Graphe non orienté

	1	2	3	4	5	6
1	0	1	1	0	0	0
2	1	0	1	1	0	0
3	1	1	0	1	0	0
4	0	1	1	0	0	0
5	0	0	0	0	0	1
6	0	0	0	0	1	0

1	2	3	
2	1	3	4
3	1	2	4
4	2 3		
5	6		
6	5		

FIGURE 1.4 – Matrice d’adjacence du graphe 1.3

FIGURE 1.5 – Liste d’adjacence du graphe 1.3

1.2.1 Matrice d’adjacence

Une matrice d’adjacence pour un graphe G à n sommets est une matrice de dimension $n \times n$ dont où l’élément à la position i, j vaut 1 s’il existe une arête reliant le sommet i au sommet j , 0 sinon.

1.2.2 Liste d’adjacence

Une liste d’adjacence est la liste des voisins de chaque sommets. C’est une représentation relativement compacte lorsqu’il y a peu d’arêtes.

1.3 Chemins

Définition 1.3.1 — Chemin. Un chemin de longueur k de u à v dans un graphe $G = (S, A)$ est une suite de sommets u_0, u_1, \dots, u_k telle que $u_0 = u, u_k = v$ et on ne passe que par des arêtes de G . $\forall i \in \llbracket 0, k \rrbracket \in A$

Définition 1.3.2 — Chemin simple. Un chemin simple est un chemin qui ne passe pas 2 fois par le même sommet.

Exemples sur la figure 1.3 :

- 1, 3, 2, 4 est une chemin de longueur 3, de 1 à 4.
- 1, 4, 5 n’est pas un chemin.

1.4 Cycle

Définition 1.4.1 — Cycles. Pour $u \in S$, un cycle est un chemin de longueur supérieure ou égale à 3, de u à u , qui ne passe pas deux fois par le même sommet. Le graphe de la figure 1.3 possède des cycles.

1.5 Connexe

Définition 1.5.1 — Connexe. Un graphe est connexe si $\forall u, v \in S$ il existe un chemin de u à v dans G . Le graphe de la figure 1.3 n’est pas connexe. Le graphe de la figure 1.6 est connexe.

1.6 Arbres

Définition 1.6.1 — Arbre. Un arbre est un graphe connexe sans cycle.

1.7 Arbre couvrants

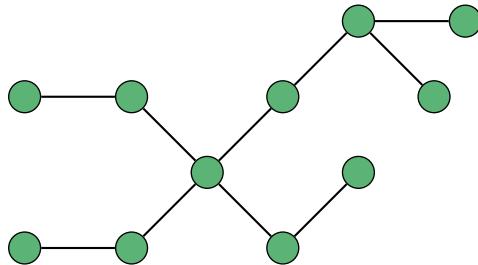


FIGURE 1.6 – Arbre

Définition 1.7.1 — Arbre couvrant. Un arbre $T = (S, A')$ couvre un graphe $G = (S, A)$ si $A' \subseteq A$.

Un graphe à n sommets peut admettre jusqu'à n^{n-1} arbres couvrants.

1.8 Graphes valués

Définition 1.8.1 — Graphe valué. Un graphe est valué (ou pondéré) lorsqu'on y associe une fonction de coût sur les arêtes : $w : S \times S \rightarrow \mathbb{R}^+$

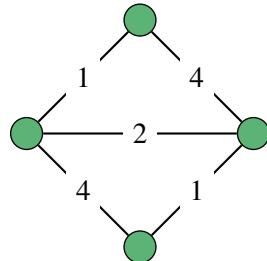


FIGURE 1.7 – Graphe non orienté

1.9 Algorithme de Kruskal (1956)

L'algorithme de Kruskal (1956) permet de trouver un arbre couvrant minimal dans un graphe valué $G = (S, A)$, $w : S \times S \rightarrow \mathbb{R}^+$.

1. Trier les arêtes par ordre de coût.
2. Parcourir toutes les arêtes (u, v) dans cet ordre. Si (u, v) ne forme pas de cycle, l'ajouter à l'arbre.

À ne pas confondre

Un **problème** décrit sur chaque donnée quel est le résultat attendu. ex : tri, recherche.

La **structure de données** décrit comment les données sont organisées et comment on y accède.

Un **algorithme** décrit le déroulement des étapes permettant de résoudre un problème.

Un **programme** (ou **implémentation**) est un choix des structures de données, des bibliothèques, du langage, de l'interface...

Exercices

Exercice 1 : Ordres de grandeur

Donner les relations (o, O, θ) entre les fonctions f et g suivantes :

1. $f(n) = 2n$ et $g(n) = 5n + 1$;
2. $f(n) = n^2$ et $g(n) = n^3$;
3. $f(n) = n^2$ et $g(n) = 2n^2 + 3n + 5$;
4. $f(n) = 8(\log n)^2$ et $g(n) = n - 2$;
5. $f(n) = \log(n^2)$ et $g(n) = \log n$;
6. $f(n) = 2^n$ et $g(n) = n^{10}$;
7. $f(n) = 2^{n+1}$ et $g(n) = 2^n$;
8. $f(n) = 2^{2n}$ et $g(n) = 2^n$.

Exercice 2 : Représentation des arbres

Un arbre (pas forcément binaire) peut être représenté de deux façons :

- par les prédecesseurs : chaque nœud contient un pointeur vers son père.
- par liste d'adjacence : chaque nœud contient la liste de tous ses fils.

1. Dessiner un arbre à 8 sommets et donner les deux façons de le représenter.
2. Proposer un algorithme qui transforme un arbre donné par liste d'adjacence en un arbre donné par prédecesseurs. Évaluer sa complexité.
3. Donner un algorithme qui transforme un arbre donné par prédecesseurs en un arbre donné par liste d'adjacence. Évaluer sa complexité.

Solution : <https://cloud.sagemath.com/projects/bb22f23b-6876-4930-9368-3ea2c32cf718/files/Tree.sagews>

2. Cycles et composantes connexes

2.1 Problèmes

Problème 2.1 Soit $G = (S, A)$ et $u, v \in S$.

Répondre *VRAI* si il existe un chemin de u à v dans G , *FAUX* sinon.

Problème 2.2 Soit $G = (S, A)$ et $u, v \in S$.

Est-ce qu'ajouter (u, v) à A crée un nouveau cycle ?

Proposition 2.1.1 Problème 2.1 \Leftrightarrow Problème 2.2

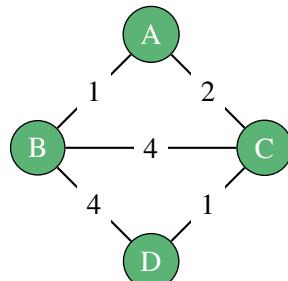
Si il existe un chemin de u à v : $u = u_1, u_2, \dots, u_k = v$ (qui ne passe pas 2 fois par le même sommet) alors il existe un cycle si on ajoute u, v .

Dans l'autre sens, si ajouter (u, v) crée un nouveau cycle, alors il existe un chemin. En effet, si le morceau cycle passe par u, v_1, \dots, v_k, u alors G contient le chemin u, v_k, \dots, v_1, v .

Définition 2.1.1 Une composante connexe d'un graphe $G = (S, A)$ est un sous-ensemble de sommets $S' \subseteq S$ maximal tel que $\forall u, v \in S'$, il existe un chemin de u à v dans G .

Problème 2.3 Soient $G = (S, A)$ et $u, v \in S$, u et v sont-ils dans la même composante connexe ?

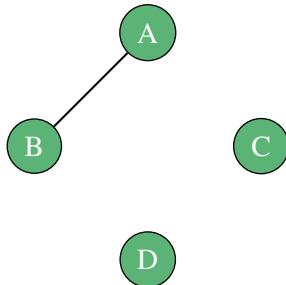
2.2 Application : Kruskal



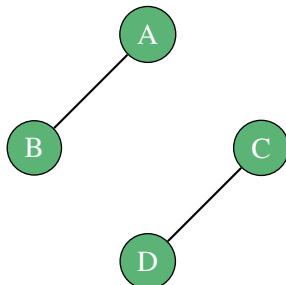
Trier les arêtes : $(A, B)(C, D)(A, C)(B, C)(B, D)$

Ajouter dans l'ordre sans créer de cycle :

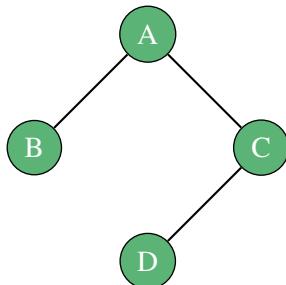
1. Ajouter (A, B) , 3 composantes connexes.



2. Ajouter (C, D) , 2 composantes connexes.



3. Ajouter (A, C) ? Ok, A et C ne sont pas dans la même composante connexe.



4. Ajouter (B, C) . Non.

Objectif : trouver un algorithme performant pour résoudre le problème 2.1.

3. L'algorithme : Union-Find

3.1 Définition

On démarre avec des éléments d'un univers U (ici les sommets d'un graphe). On admet 2 opérations :

- $UNION(u, v)$: associe u et v (ici, on ajoute une arête)
- $FIND(u, v)$: permet de savoir si u et v sont associés, transitive (ici, si il existe un chemin de u à v)

3.2 Exemple

Algorithm 1 Algorithme de Kruskal

```
for ( $u, v$ ) in  $A$  (par ordre croissant d'étiquette) do
    if not  $FIND(u, v)$  then
         $UNION(u, v)$ 
    else
        end
    end if
end for
```

3.3 Applications

- Segmentation d'image (combien de formes dans une image).
- Construction de labyrinthes.
- Kruskal, arbre couvrant minimal.
- Détection de cycles.

3.4 Idée de l'algorithme

Chaque élément connaît son "chef".

On note $Id[u]$ le chef de u .

Pour savoir si u et v sont associés, on note la "hiérarchie" de chef en chef. Si u et v ont le même "PDG", alors u et v sont associés.

Pour associer u et v : $UNION(u, v)$ = le chef du "PDG" de u devient le "PDG" de v .

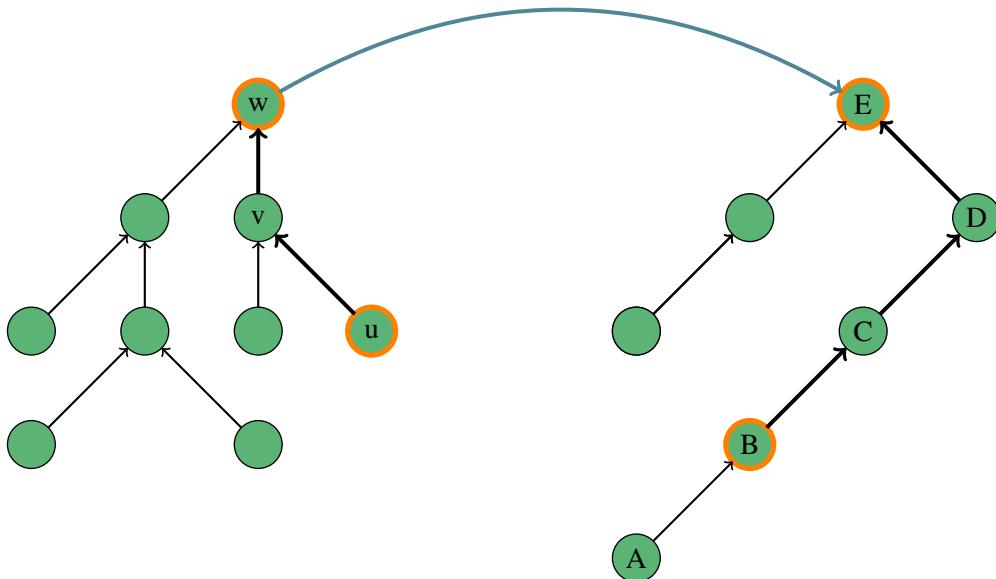


FIGURE 3.1 – $UNION(u, B)$

3.5 Retour sur Kruskal

On teste successivement les arêtes avec $FIND$, tant que la fonction renvoie $FAUX$ on utilise $UNION$, sinon on a fini.

3.6 Implémentation en Python

La classe UF en Python :

```

1  class UF:
2      def __init__(self, U):
3          self.Id = {}
4          for x in U:
5              self.Id[x] = x
6      def root1(self, u):
7          while self.Id[u] != u:
8              u = self.Id[u]
9          return u
10     def find1(self, u, v):
11         return self.root1(u) == self.root1(v)
12     def union1(self, u, v):
13         if not self.find1(u, v):
14             chefu = self.root1(u)
15             chefv = self.root1(v)
16             self.Id[chefu] = chefv

```

3.7 Complexité

- `root1` : $O(\text{hauteur de l'arbre})$
- `find1` : $2 \times \text{hauteur} + 1$
- `union1` : $4 \times \text{hauteur} + 1$

3.8 UNION : Maîtriser la hauteur des arbres

Pour améliorer la complexité on cherche à maîtriser la hauteur des arbres.

Suggestion : on raccroche le plus petit arbre au plus grand ; cela n'augmente pas la hauteur du nouvel arbre sauf si les deux sont de même hauteur.

On maintient un deuxième tableau, qui, pour chaque élément donne sa hauteur.

Au moment de faire l'union, on met à jour `size[u]`, `size[v]`.

```

1 from UF import UF
2 class UF2(UF):
3     def __init__(self, U):
4         super(UF2, self).__init__(U)
5         self.size = {}
6         for x in U:
7             self.size[x] = 0
8     def union2(self, u, v):
9         chefu = self.root1(u)
10        chefv = self.root1(v)
11        if self.size[chefu] > self.size[chefv]:
12            self.Id[chefv] = chefu
13        else:
14            self.Id[chefu] = chefv
15        if self.size[chefu] == self.size[chefv]:
16            self.size[chefv] += 1

```

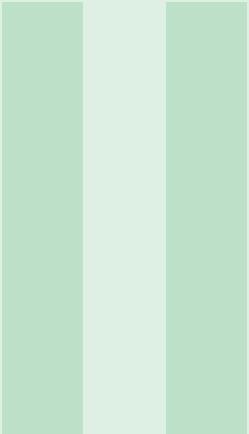
3.9 FIND : Compresser les chemins

Au moment de chercher le représentant d'un élément on compresse le chemin

```

1 from UF import UF
2 class UF3(UF):
3     def __init__(self, U):
4         super(UF2, self).__init__(U)
5         self.size = {}
6         for x in U: self.size[x] = 0
7     def find2(self, u, v):
8         return self.root2(u) == self.root2(v)
9     def root2(self, u):
10        if self.Id[u] != u:
11            self.Id[u] = self.root2(self.Id[u])
12        return self.Id[u]
13     def union2(self, u, v):
14         ru = self.root2(u)
15         rv = self.root2(v)
16         if self.size[ru] < self.size[rv]:
17             self.Id[ru] = self.Id[rv]
18         else:
19             self.Id[rv] = self.Id[ru]
20         if self.size[ru] == self.size[rv]:
21             self.size[ru] += 1

```

Parcours d'arbres

4	Arbres binaires de recherche ABR 19
4.1	Complexité
4.2	Arbres binaires de recherche
4.3	Recherche dans un ABR
4.4	Insertion dans un ABR

4. Arbres binaires de recherche ABR

4.1 Complexité

Pour un algorithme A on note $\text{coût}_A(x)$ le nombre d'opérations élémentaires que l'algorithme effectue sur une entrée x .

Complexité au pire cas : $\text{coût}_A(n) = \max \text{coût}_A(x)$, x de largeur n .

Coût amorti d'un algorithme A : on exécute $A(x_1)A(x_2)\dots A(x_n)$; coût amorti = $\frac{1}{n} \sum_{i=1}^n \text{coût}_A(x_i)$

Complexité en moyenne : "*définition mathématique formelle de, à la louche*", on a une distinction sur les entrées μ , $\text{coût}_A^\mu(x) = \sum_{x \sim \mu} \mu(x) \text{coût}_A(x)$.

4.2 Arbres binaires de recherche

Définition 4.2.1 — Arbre binaire. Un arbre binaire est soit vide, soit une racine avec un arbre binaire à gauche et un arbre binaire à droite.

En python :

```
1 class ArbreBinaire:
2     # par convention ArbreVide = None
3     def __init__(self, v, g=None, d=None):
4         self.val = v
5         self.gauche = g
6         self.droite = d
```

Définition 4.2.2 — Arbre binaire de recherche. Un arbre binaire de recherche (ABR) est un arbre étiqueté avec les propriétés suivantes :

1. $A.\text{val}$ est plus grand que toutes les valeurs de $A.\text{gauche}$.
2. $A.\text{val}$ est plus petit que toutes les valeurs de $A.\text{droite}$.

Définition 4.2.3 — Taille d'un arbre. On écrit $|A|$ la taille de l'arbre A son nombre de sommets.

$$|A| = \begin{cases} 0 & \text{si l'arbre est vide} \\ 1 + |A.gauche| + |A.droite| & \text{sinon} \end{cases}$$

Définition 4.2.4 — Profondeur d'un nœud. On définit la profondeur d'un nœud v dans l'arbre A comme la longueur du chemin de la racine de A jusqu'au sous arbre de racine v :

$$\text{Prof}_A(v) = \begin{cases} 0 & \text{si } A.val = v \\ 1 + \text{Prof}_{A.gauche}(v) & \text{si } A.val > v \\ 1 + \text{Prof}_{A.droite}(v) & \text{si } A.val < v \end{cases}$$

Définition 4.2.5 — Hauteur d'un arbre. On définit la hauteur d'un arbre A :

$$h(A) = \begin{cases} 0 & \text{si l'arbre est vide} \\ 1 + \max(h(A.gauche), h(A.droite)) & \text{sinon} \end{cases}$$

Remarque : $h(A) = \max \text{Prof}_A(v), v \in A$

4.3 Recherche dans un ABR

```

1 def TrouverABR(A, v):
2     if v == A.val: return A
3     elif v < A.val and A.gauche is not None:
4         return TrouverABR(A.gauche, v)
5     elif v > A.val and A.droite is not None:
6         return TrouverABR(A.droite, v)
7     return None

```

Proposition 4.3.1 $T(A, v) = \begin{cases} \text{Prof}_A(v) & \text{si } v \in A \\ h(A) & \text{si } v \notin A \end{cases}$

coût_{trouverABR}(A, v) $\leq T(A, v)$, \forall ABR A , \forall valeur v , $|A| \geq 1$

4.3.1 Preuve par récurrence sur $|A|$

Hypothèse d'induction

$H(n)$ = l'énoncé de la proposition 4.3 est vrai $\forall A, v |A| \leq n$

Initialisation ("Base")

$n = 1$: A a une seule valeur.

si $v \in A$ alors v se trouve à la racine, 1 comparaison.

si $v \notin A$ alors on fera 5 comparaisons.

Héritéité ("Pas d'induction")

Supposons $H(n)$ est vraie pour $n \leq 1$, montrons $H(n+1)$.

Soit A de taille $n+1$.

Si $v \in A$: Si $A.val = v$ on s'arrête après 1 comparaison.

Si $A.val > v$: On fait 3 + $T(A.gauche, v)$ comparaisons (au plus)

$$T(A.gauche, v) = 5(\text{Prof}_{A.gauche}(v) \text{ par } H(|A.gauche|))$$

On doit montrer que $T(A.gauche, v) + 3 \leq T(A, v)$

$$\Leftrightarrow 5(\text{Prof}_{A.gauche}(v) + 1) + 3 \leq T(A, v) = 5(\text{Prof}_A(v) + 1)$$

$$\begin{aligned} \text{D'après la définition de } \text{Prof}_A(v), \text{ cas } v < A.val &= 5(\text{Prof}_{A.gauche}(v) + 1 + 1) \\ &= 5(\text{Prof}_{A.gauche}(v) + 1) + 5 \end{aligned}$$

Si $A.val < v$: On fait $T(A.droite, v) + 5$ comparaisons (au plus)

[...]

$$\begin{aligned}
 & \text{Si } v \notin A : \text{Si } A.\text{val} = v, \text{impossible} \\
 & \quad \text{Si } A.\text{val} > v : \#comparaisons = T(A.\text{gauche}, v) + 3 \\
 & \quad \leq 5h(A.\text{gauche}) + 3 \text{ par hypothèse d'induction} \\
 & \quad \leq 5(h(A) - 1) + 3 \text{ par définition de } h \\
 & \quad \leq 5h(A) \\
 & \quad \text{Si } A.\text{val} < v : \#comparaisons = T(A.\text{droit}, v) + 5 \\
 & \quad \leq 5h(A)
 \end{aligned}$$

4.4 Insertion dans un ABR

4.4.1 Algorithme

```

1 def InsererABR(A, v):
2     if v == A.val: raise Exception("v déjà dans l'arbre")
3     if v < A.val:
4         if A.gauche is not None:
5             InsererABR(A.gauche, v)
6         else: A.gauche = ABR(v, None, None)
7     if v > A.val:
8         if A.droite is not None:
9             InsererABR(A.droite, v)
10        else: A.droite = ABR(v, None, None)

```

On souhaite faire l'analyse amortie de cet algorithme, en supposant qu'on insère les éléments 1, 2, ... N dans un ordre aléatoire.

$$\text{coût Amorti} = \sum_{i=1}^N \text{coût}_{\text{InsererABR}}(\text{la } i\text{ème valeur})$$

On suppose qu'il reste à insérer les valeurs $x_1 < x_2 < \dots < x_N$ et on choisit d'insérer x_i avec une probabilité de $\frac{1}{N}$.

Si on choisit x_i , on aura $i - 1$ éléments à gauche et $n - i$ éléments à droite.

4.4.2 Coût moyen d'insertion de n éléments dans un ABR vide

Proposition 4.4.1 Le coût moyen d'insérer $x_1 < x_2 < \dots < x_N$ dans un ABR initialement vide, noté C_N est : (où on compte le nombre d'appels récursifs)

$$C_N = \begin{cases} 0 \text{ si } N = 0 \\ 1 \text{ si } N = 1 \\ \sum_{i=1}^N \frac{1}{N} [1 + N - 1 + C_{i-1} + C_{N-i}] \text{ sinon} \end{cases}$$

Car : coût(placer x_i) = 1

les $N - 1$ qui restent passent par la racine,
 $x_1 \dots x_{i-1}$ sont insérés à gauche,
 $x_{i+1} \dots x_N$ sont insérés à droite.

Proposition 4.4.2 $C_N = O(N \log N)$

Démonstration. Prenons $N > 1$

$$\begin{aligned}
 C_N &= N + \frac{1}{N} \sum_{i=1}^N (C_{i-1} + C_{N-i}) \\
 &= N + \frac{1}{N} \left(\sum_{i=1}^N C_{i-1} + \sum_{j=1}^N C_{N-j} \right) \\
 &= N + \frac{1}{N} 2 \sum_{k=0}^{N-1} C_k
 \end{aligned}$$

Télescopons :

$$\begin{aligned}
 NC_N &= N^2 + 2 \sum_{k=0}^{N-1} C_k \\
 (N-1)C_{N-1} &= (N-1)^2 + 2 \sum_{k=0}^{N-2} C_k \\
 NC_N - (N-1)C_{N-1} &= N^2 - (N-1)^2 + 2C_{N-1} \\
 NC_N &= 2N-1 + (N+1)C_{N-1} \\
 \frac{C_N}{N+1} &= \frac{2N-1}{N(N+1)} + \frac{C_{N-1}}{N}
 \end{aligned}$$

On pose : $t(N) = \frac{2N-1}{N(N+1)} \simeq \frac{2}{N}$

$$\begin{aligned}
 \frac{C_N}{N+1} &= t(N) + \frac{C_{N-1}}{N} \\
 \frac{C_N}{N+1} &= t(N) + t(N-1) + \frac{C_{N-2}}{N-1} \\
 &= t(N) + t(N-1) + t(N-2) + \frac{C_{N-3}}{N-2} \\
 &= \sum_{i=2}^N t(i) + \frac{1}{2} \\
 &\approx \sum_{i=1}^N \frac{2}{i} + \frac{1}{2} \\
 &\approx 2 \sum_{i=1}^N \frac{1}{i}
 \end{aligned}$$

$\frac{C_N}{N+1} \approx 2 * N \ln n$

On conclut que $C_N = 2(N+1)N \ln(N)$ et que le coût amorti $\frac{C_N}{N} = 2(N+1) \ln(N)$

■

IV Algorithmes de graphes

5	Algorithme de Kruskal	27
5.1	Implémentation de l'algorithme de Kruskal	
5.2	Correction de l'algorithme de Kruskal	
6	Algorithme de Prim	29
6.1	Implémentation de l'algorithme de Prim (version optimisée)	
6.2	Complexité de l'algorithme de Prim	
7	Les plus courts chemins : Dijkstra	31
7.1	Algorithme de Dijkstra	
7.2	Preuve de l'algorithme de Dijkstra	
8	Algorithme de Bellman-Ford	35
8.1	Idée	
8.2	Implémentation en Python	
8.3	Preuve	
	Index	37

5. Algorithme de Kruskal

Définition 5.0.1 — Graphe valué. Un graphe $G = (S, A)$ est valué (ou pondéré) si on y associe une fonction de coût sur ses arêtes : $w : A \rightarrow \mathbb{R}^+$.

Définition 5.0.2 — Arbre couvrant minimal. Soit $G = (S, A)$ et w une fonction de coût, trouver $A' \subseteq A$ tel que $T = (S, A')$ soit un arbre couvrant de cout minimal $\text{coût}(T) = \sum_{(u, v) \in A'} w(u, v)$

5.1 Implémentation de l'algorithme de Kruskal

```
1 from UF import UF
2 def Kruskal (G, w):
3     T = [] ; cout = 0;
4     A = G.A # l'ensemble des arretes de G
5     S = sorted(A) # A trie selon w croissant
6     for (u, v) in S:
7         if not UF.find(u, v): # si (u, v) ne forme pas de cycle
8             T.append((u, v))
9             cout += w[(u, v)]
10            UF.union(u, v)
11    return (T, cout)
```

Complexité : $m \log m$ avec $m = \text{nombre d'arêtes}$

5.2 Correction de l'algorithme de Kruskal

Définition 5.2.1 — Arbres couvrants minimaux. $ACM(G, w) =$ l'ensemble des arbres couvrants minimaux de $G = (S, A) : T' = (S, A')$ avec $A' \subseteq A$.

Définition 5.2.2 — Compatibilité. Pour toute solution partielle $T' \subseteq G$, on dit que T' est compatible avec $ACM(G, w)$ et on note $T' \sqsubseteq ACM(G, w)$ si $\exists T \in ACM(G, w)$ tel que $T' \subseteq T$.

5.2.1 Lemme d'optimalité

Soit G connexe, $G = (S, A)$ et $w : A \rightarrow \mathbb{R}^+$. Si $T' \subseteq ACM(G, w)$, $T' = (S, A')$ et $S = S_1 \cup S_2$ $S_1 \cap S_2 = \emptyset$ tels que $A' \cap (S_1 \times S_2) = \emptyset$, alors on peut ajouter $(u, v) \in S_1 \times S_2$, de coût minimal parmi les arêtes $S_1 \times S_2 \cap A$. Autrement dit, $T \cup \{(u, v)\} \subseteq ACM(G, w)$

Preuve

On a $G = (S, A)$ connexe, $w \leftarrow \mathbb{R}^+$ $S = S_1 \cup S_2$ $S_1 \cap S_2 = \emptyset$

$T' \subseteq T \in ACM(G, w)$

$T' \cap S_1 \times S_2 = \emptyset$

Soit (u, v) **de moindre coût** parmi $A \cap S_1 \times S_2$

On veut montrer que $T' \cup \{(u, v)\} \subseteq ACM(G, w)$

— **cas 1** : si $(u, v) \in T$ alors $T' \cup \{(u, v)\} \subseteq ACM(G, w)$

— **cas 2** : si $(u, v) \notin T$, comme G est connexe, T est un *ACM*.

T est connexe, $(u', v') \in S_1 \times S_2 \cap T$ car il existe un chemin de u à v dans T .

On a pris (u, v) **de poids minimal**; donc $w(u, v) \leq w(u', v')$.

Regardons $\tilde{T} = T \setminus \{(u', v')\} \cup \{(u, v)\}$

1. coût de T : coût dans S_1 + coût dans S_2 + $w(u', v')$

coût de \tilde{T} : coût dans S_1 + coût dans S_2 + $w((u, v))$ or $(u, v) \leq w(u', v')$

Donc coût de $\tilde{T} <$ coût de T

2. Si T est un arbre couvrant, \tilde{T} l'est aussi.

Donc $\tilde{T} \in ACM(G, w)$ et $T' \cup \{(u, v)\} \subseteq ACM(G, w)$

Dans le cas de l'algorithme de Kruskal, supposons qu'à une itération donnée, on ait choisi les arêtes T' et on a montré $T \subseteq ACM(G, w)$.

Posons S_1 la composante connexe qui contient u et $S_2 = S \setminus S_1$.

(u, v) que choisit Kruskal est celle de poids minimal parmi les arêtes qui restent qui ne forme pas de cycle, comme $u \in S_1$, forcément $v \notin S_1$. Donc on peut l'ajouter d'après le lemme d'optimalité.

6. Algorithme de Prim

On fait pousser un arbre T .

À chaque instant T est un arbre.

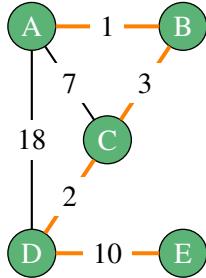
On ajoute l'arête de coût minimal parmi les arêtes qui sortent de l'arbre.

Pour trouver la meilleure on utilise une file de propriété (un tas min).

6.1 Implémentation de l'algorithme de Prim (version optimisée)

On met les sommets dans le tas et on note le prédécesseur du sommet (arête) qui donne la meilleure priorité. En cours d'algorithme la priorité des sommets dans le tas peut changer.

```
1 from FilePriorite import FilePriorite
2 def Prim(G, w, s):
3     inf = float("inf")
4     H = FilePriorite()
5     T = []
6     s.priorite = 0;    H.inserer(0, s)
7     for u in G.sommets:
8         u.pred = None
9         if u != s :
10             H.inserer(inf, u)
11             u.priorite = inf
12     while not H.is_empty():
13         p, u = H.pop_min()
14         if u.priorite != inf:
15             if u.pred is not None: T.append((u.pred, u))
16             for v in u.voisins:
17                 if v.idx is not None: #v n'est pas sorti du tas
18                     if w[(u, v)] < v.priorite:
19                         v.priorite = w[(u, v)]
20                         H.diminuer_cle(v.idx, v.priorite)
21                         v.pred = u
22     return T
```

Exemple

Tas :	$(A, 0)$	(B, ∞)	(C, ∞)	(D, ∞)	(E, ∞)
	<u>$(A, 0)$</u>	<u>(B, ∞)</u>	<u>(C, ∞)</u>	<u>(D, ∞)</u>	<u>(E, ∞)</u>
		<u>$(B, 1)$</u>	<u>$(C, 7)$</u>	<u>$(D, 18)$</u>	<u>(E, ∞)</u>
			<u>$(C, 3)$</u>	<u>$(D, 18)$</u>	<u>(E, ∞)</u>
				<u>$(D, 3)$</u>	<u>(E, ∞)</u>
					<u>$(E, 10)$</u>

6.2 Complexité de l'algorithme de Prim

Initialisation : $n \times \text{insérer} = O(n \log n)$

Boucle while : n fois

Corps de la boucle : $\text{pop} + \text{for} \times \text{update} = \log n + \deg(u) \times \log n \approx n \log n + 2m \log n$

7. Les plus courts chemins : Dijkstra

Communisme vs Capitalisme

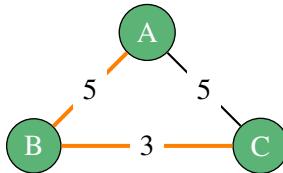


FIGURE 7.1 – Arbre Couvrant minimal, coût = 8

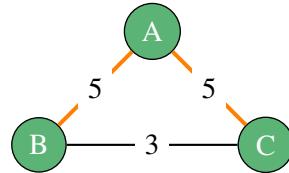


FIGURE 7.2 – Plus courts chemins depuis A

Étant donné un graphe valué G, w et un sommet de départ s . On veut calculer les plus courts chemins de s à v dans $G \forall v \in S$.

Définition 7.0.1 — Notations. Si $C = (s = u_0, u_1, u_2, \dots u_k = v)$ est un chemin de s à v dans G , on écrit :

- $s \xrightarrow{C} v$
- $w(C) = \sum_{i=1}^k w(u_{i-1}, u_i)$
- $\delta(s, v) = \min\{w(C) : s \xrightarrow{C} v\}$

Proposition 7.0.1 On peut toujours trouver des chemins C_u $s \xrightarrow{C_u} u$ tel que l'union des C_u forme un arbre.

Preuve Supposons qu'on ait 2 sommets u, v des plus courts chemins C_u, C_v dont l'union forme un cycle.

Les segments C_1, C_2 $x \xrightarrow{C_1} y$ $x \xrightarrow{C_2} y$ ont le même coût, sinon supposons $w(C_1) < w(C_2)$ alors il y a un chemin plus court de s à v qui passe par C_1 ; C_2 n'est pas un plus court chemin, impossible.

7.1 Algorithme de Dijkstra

On modifie l'algorithme de Prim, la notion de priorité change, maintenant c'est la distance à s .

améliorer la priorité \equiv améliorer la distance à s

```

1 from FilePriorite import FilePriorite
2 def Dijkstra(G, w, s):
3     inf = float("inf")
4     H = FilePriorite()
5     T = []
6     s.priorite = 0;    H.inserer(0, s)
7     for u in G.sommets:
8         u.pred = None
9         if u != s:
10            H.inserer(inf, u)
11            u.priorite = inf
12     while not H.is_empty():
13         p, u = H.pop_min()
14         if u.priorite != inf:
15             if u.pred is not None: T.append((u.pred, u))
16             for v in u.voisins:
17                 if v.idx is not None: #v n'est pas sorti du tas
18                     if u.priorite + w[(u, v)] < v.priorite:
19                         v.priorite = u.priorite + w[(u, v)]
20                         H.diminuer_cle(v.idx, v.priorite)
21                         v.pred = u
22     return T

```

7.2 Preuve de l'algorithme de Dijkstra

Définition 7.2.1 Notons $u.prio^t$ la valeur de $u.prio$ à la $t^{\text{ième}}$ itération.

Théorème 7.2.1 À l'itération t , si $u \notin \text{tas}$ alors $u.prio^t = \delta(s, u)$.

On utilise sans démontrer trois propriétés :

- (P1) S'il n'y a pas de chemin de s à u alors $u.prio^t = \infty \forall t$
- (P2) "Borne supérieure" : $\forall u, \forall u.prio^t \geq \delta(s, u)$ et $u.prio^t \geq u.prio^{t+1}$
- (P3) Si $s \xrightarrow{G}^* u \xrightarrow{} v$ est un plus court chemin de s à v et $u.prio^t = \delta(s, u)$, alors $\forall t' > t v.prio^{t'} = \delta(s, v)$

Preuve du théorème

Supposons qu'à l'itération t , u sorte du *tas*.

Regardons T^t l'ensemble des arêtes déjà choisies.

Soit C un plus court chemin de s à u , et x, y la première arête de C telle que $(x, y) \notin T^t$.

$$s \xrightarrow{C_1 \subseteq T} x \xrightarrow{\notin T} y \xrightarrow{C_2} u$$

On va montrer que $y = u$.

(I) Montrons que : $y.prio^t = \delta(s, y)$

(*) On utilise le fait que sur un plus court chemin de u_0 à u_k les segments initiaux de u_0 à u_i (avec $0 \leq i \leq k$) sont aussi des plus courts chemins

On a que $x \in T^t$, supposons qu'il a été ajouté à l'instant $t' < t$.

Par hypothèse d'induction : $x.prio^{t'} = \delta(s, x)$

À l'itération t' , y est encore dans le *tas* et y est voisin de x , et on aura posé $y.prio^{t'} \leq x.prio^{t'} + w(x, y)$ or $s \xrightarrow{}^i x \xrightarrow{}^1 y$ est un plus court chemin (par (*)).

$y.prio^{t'} \leq \delta(s, x) + w(x, y) = \text{longueur du chemin} = \delta(s, y)$ et par P2 $y.prio^t = \delta(s, y)$

(II) On montre que $C_2 = \emptyset$

Donc on conclut que $y = u$, et par (I) $u.prio^t = y.prio^t = \delta(s, y) = \delta(s, u)$

À l'itération t , $y \in T^t$, mais y a une priorité $\neq \infty$ donc y est dans le tas, tout comme u qui a été choisi.

$$\begin{aligned} u.prio^t &\leq y.prio^t \quad \text{car } u \text{ sort du tas} \\ &= \delta(s, y) \quad (\text{II}) \\ &\leq \delta(s, u) \quad \text{car } y \text{ précède } u \text{ sur un plus court chemin} \\ &\leq u.prio^t \quad (\text{P2}) \end{aligned}$$

\Rightarrow toutes ces quantités sont égales $\Rightarrow y = u$

8. Algorithme de Bellman-Ford

Cet algorithme permet de trouver les plus courts chemins depuis un sommet source donné. Il est cependant moins performant que Dijkstra, mais est distribuable, chaque sommet du réseau connaît ses voisins, propage les résultats des calculs intermédiaires à ses voisins.

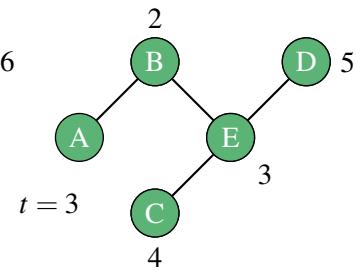
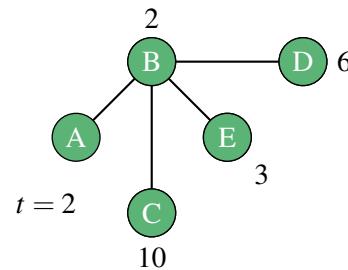
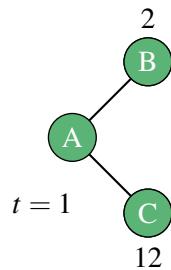
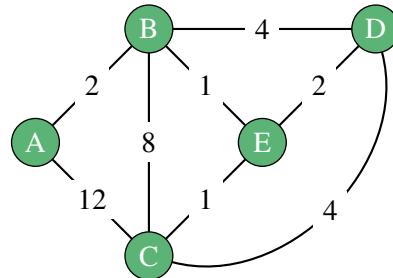
8.1 Idée

Commencer avec $d(s) = 0$ et $d(u) = \infty \forall u \neq s$.

À chaque itération, on parcourt toutes les arêtes du graphe ; $(u, v) \in A : \text{si } d(u) + w(u, v) < d(v)$

On pose $d(v) = d(u) + w(u, v)$ et $\text{pred}(v) = u$.

Exemple



On remarque qu'à l'itération t on trouve des plus courts chemins qui parcourent au plus t arêtes.

Un plus court chemin de s à u parcourt au plus $n - 1$ arêtes. Pour que l'algorithme soit correct, il suffit de montrer qu'à l'itération t on a calculé les plus courts chemins passant par au plus t arêtes et répéter $n - 1$ fois pour avoir tous les plus courts chemins de s aux autres sommets.

8.2 Implémentation en Python

```

1 def Bellman_Ford(G, w, s):
2     inf = float("inf")
3     for u in G.sommets:
4         if u != s: u.dist = inf
5         u.pred = None
6     s.dist = 0
7     for t in range(1, len(G.sommets)):
8         for u in G.sommets:
9             for v in u.voisins:
10                 if v.dist > u.dist + w[(u, v)]:
11                     v.dist = u.dist + w[(u, v)]
12                     v.pred = u
13     return [(u.pred, u) for u in G.sommets if u != s]

```

Complexité : $n \times m$

8.3 Preuve

Définition 8.3.1 On note :

- $\delta^t(u, v)$ = longueur du plus court chemin de u à v en passant par au plus t arêtes.
- $v.dist^t$ = valeur de $v.dist$ à l'itération t .

Proposition 8.3.1 $\forall v \in S \quad d^t(v) = \delta^t(s, v)$

Démonstration. Par récurrence sur t .

Lorsque $t = 0$, un seul chemin de longueur 0 de s à s .

À l'itération $t > 0$, par hypothèse d'induction, $\forall x \quad x.dist^{t-1} = \delta^{t-1}(s, x)$

Soit C_v un plus court chemin de s à v parcourant au plus t arêtes. Si C_v a moins de t arêtes alors on l'a découvert à une itération précédente $t' < t$ donc $v.dist^{t'} = \delta^{t-1}(s, v)$. Si C_v a t arêtes, le chemin $s \rightarrow x$ est un plus court chemin qui parcourt au plus $t - 1$ arêtes (sinon on trouverait un meilleur chemin de t arêtes vers v).

On peut appliquer l'hypothèse d'induction $x.dist^{t-1} = \delta^{t-1}(s, x)$.

À l'itération t on considère x et son voisin v et on posera :

$$\begin{aligned}
 v.dist &\leq x.dist + w(x, v) \\
 &= \delta^{t-1}(s, x) + w(x, v) \\
 &= \delta^t(s, v)
 \end{aligned}$$

■

Index

A	Cycle	8
ACM	27	
Algorithme de Kruskal.....	27	
Algorithme de Prim	29	
Arbre	8	
Arbre binaire	19	
Arbre binaire de recherche	19	
Arbre couvrant	8, 9	
Arbre couvrant minimal.....	27	
B	Dijkstra	31
Bellman-Ford (algorithme).....	35	
C	FIND	13
Chemin	8	
Chemin simple.....	8	
Coût amorti	19	
Compatibilité	27	
Complexité.....	19	
Complexité au pire cas	19	
Complexité en moyenne	19	
Composante connexe	11	
Connexe	8	
D	Graphe	7
Dijkstra	31	
F	Graphe pondéré.....	9, 27
FIND	13	
G	Graphe valué	9, 27
Graphe	7	
Graphe pondéré.....	9, 27	
Graphe valué	9, 27	
H	Hauteur (Arbre).....	20
Hauteur (Arbre).....	20	
I	Insertion (ABR).....	21
Insertion (ABR).....	21	

K

Kruskal 27

L

Liste d'adjacence 8

M

Matrice d'adjacence 8

P

Prim 29

Problème 9

Profondeur d'un nœud (Arbre) 20

R

Recherche dans un ABR 20

S

Structure de données 9

U

UNION 13