Emmanuel Jeandel

ALGORITHMES ET PROGRAMMATION 4

TABLE DES MATIÈRES

PRÉLIMINAIRES

La plupart des notions abordées dans ce cours sont assez standard, et se retrouvent dans la plupart des livres d'algorithmique.

Les livres suivants sont accessibles à la bibliothèque universitaire de la Fac de Sciences (à l'exception du livre [3] qui est disponible à la BU de gestion au centre ville)

Bien que le cours ne les suive pas exactement, les livres [2] et [4] sont très adaptés.

[1] Bruno BAYNAT, Philippe CHRÉTIENNE, Claire HANEN, Salia KEDAD-SIDHOUM, Alix MUNIER-KORDON et Christophe PICOULEAU : *Exercices et problèmes d'algorithmique*. Dunod, 2010.

[2] Thomas H. CORMEN, Charles E. LEISERSON, Ronald L. RIVEST et Clifford STEIN : *Introduction à l'algorithmique*. Dunod, 2004.

[3] Michael T. GOODRICH et Roberto TAMASSIA : *Data Structures and Algorithms in Java*. Wiley, 2006.

[4] Robert SEDGEWICK et Kevin WAYNE : *Algorithms*. Addison-Wesley, 2011.

Quelques remarques cependant :

— Les éditions 3 et suivantes du livre [2] ne contiennent pas le chapitre sur les arbres binomiaux.

— Il existe plusieurs versions en français du livre [4]. Cependant, la majorité d'entre elles ne contiennent pas le chapitre sur l'algorithmique des graphes, qui est le chapitre le plus intéressant dans notre cas. C'est en particulier le cas de la version "Algorithmes en Java" de la bibliothèque. En revanche, la version "Algorithmes en C" (référence 005.133 SED) contient ces chapitres.

Les chapitres suivants contiennent, sauf exception, une implémentation en Python des algorithmes proposés. L'implémentation est donnée à titre d'information mais n'est pas toujours la meilleure implémentation possible, et peut quelquefois avoir une complexité supérieure à la complexité théorique indiquée.

Chapitre 1

TAS BINOMIAUX

Les tas binaires vus précédemment offrent la complexité suivante pour les opérations classiques, pour un tas contenant n éléments :

Opération	prototype	Complexité
Création à partir d'un tableau `T`	`makeheap(T)`	$O(n)$
Insertion de x avec clé k	`insert(H,x,k)`	$O(\log n)$
Suppression du Min	`extractMin(H)`	$O(\log n)$
Décrémenter la clé de x à la valeur k	`DecreaseKey(H,x,k)`	$O(\log n)$
Union de deux tas H_1, H_2 de n et m clés	`Union(`H_1`,`H_2`)`	$O(m \log n)$

Dans de nombreuses applications, il est nécessaire d'avoir une meilleure complexité des deux dernières opérations.

On introduit pour cela les tas binomiaux et tas de Fibonacci :

Prototype	Binaire	Binomiaux	Fibo
`makeheap(T)`	$O(n)$	$O(n)$	$O(n)$
`insert(H,x,k)`	$O(\log n)$	$O(1)$	$O(1)$
`extractMin(H)`	$O(\log n)$	$O(\log n)$	$O(\log n)$
`DecreaseKey(H,x,k)`	$O(\log n)$	$O(\log n)$	$O(1)$
`Union(`H_1`,`H_2`)`	$O(m \log n)$	$O(\log n + \log m)$	$O(1)$

La structure de données idéale est donc les tas de Fibonacci. Cependant ils sont assez difficiles à décrire et à implémenter (de sorte qu'ils ne sont pratiquement jamais utilisés). Nous nous contenterons dans ce cours des tas binomiaux, qui utilisent certaines des techniques présentes dans les tas de Fibonacci, mais qui sont bien plus simples à comprendre.

Une implémentation en Python est donnée à titre d'exemple.

A la fin de ce chapitre, nous n'aurons pas réussi à montrer toutes les bornes de complexité enoncées plus haut : En particulier, on montrera uniquement que l'insertion est en $O(\log n)$ pas qu'elle est en $O(1)$. Les techniques nécessaires pour obtenir (et surtout comprendre) ces meilleures complexités seront l'objet du chapitre suivant.

1.1 Définitions

Définition 1.1 *Arbre binomial*

Un arbre binomial de niveau 0 est une feuille.

Un arbre binomial de niveau k est un arbre dont la racine a k fils, chacun représentant un arbre binomial de niveau $k-1, k-2, \ldots 1, 0$ (dans cet ordre)

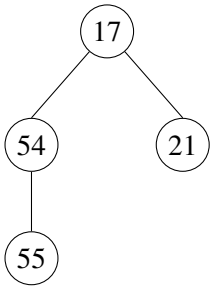
De plus un arbre binomial vérifie la propriété de *tas* : la clé d'un noeud est toujours inférieure à la clé de ses fils.

♦ **Exemple**

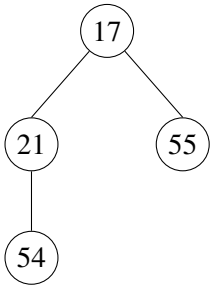
Voici un arbre binomial de niveau 0

Voici un arbre binomial de niveau 1 :

Voici un arbre binomial de niveau 2 :



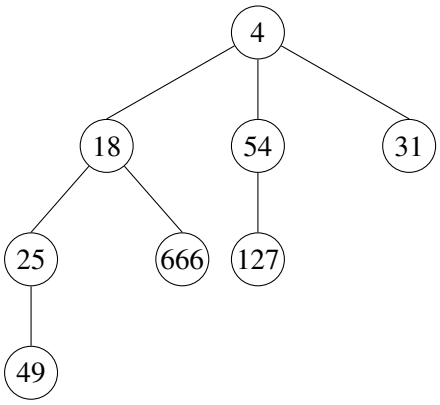
Voici un autre arbre binomial de niveau 2 avec les mêmes clés :



```
class Element:
    def __init__(self, key):
        self.key = key

class ArbreBinomial:
    def __init__(self,x):
        self.element = x
        self.element.ptr = self
        self.level = 0
        self.children = []
        self.parent = None
```

Implém 1.1 – Une implémentation possible d'un arbre binomial (dont les noeuds sont de type Element)

et un arbre binomial de niveau 3 :



On remarque que, par construction, un arbre binomial de niveau k contient 2^k éléments, et est de hauteur (distance de la racine à la feuille la plus profonde) exactement k.

Pour implémenter correctement les arbres binomiaux, on peut par exemple représenter les fils d'un noeud donné par une liste doublement chaînée, voir la figure 1.1

A noter que, suivant l'opération à effectuer, il peut être intéressant d'avoir les fils par ordre croissant de niveau, ou quelquefois par ordre décroissant, d'où l'intérêt de la liste doublement chaînée.

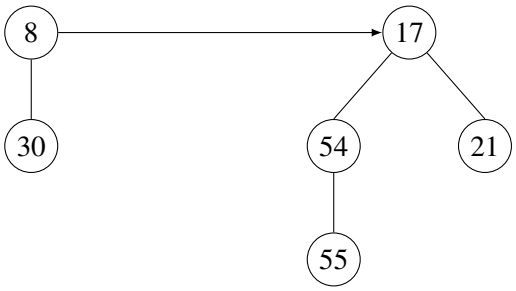
Définition 1.2 *Tas binomial*

Un tas binomial est un ensemble d'arbres binomiaux de niveau différents.

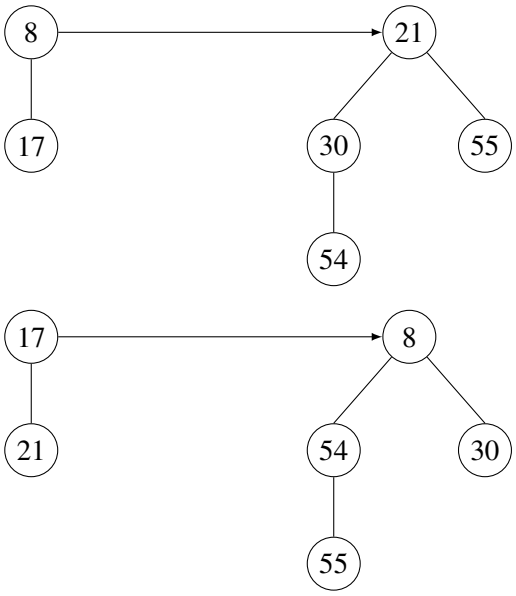
On le représente habituellement par une liste simplement chaînée, du plus petit au plus grand.

♦ **Exemple**

Voici un exemple de tas binomial avec 6 clés :



Voici deux autres tas binomiaux avec les mêmes clés :



Etant donné un ensemble de n clés, on peut trouver beaucoup de tas binomiaux différents possédant ces n clés. Cependant ils ont tous la même forme : Chaque arbre binomial contenant un nombre de clés qui est une puissance de 2, et deux arbres du même tas ayant un nombre de clés différents, on en déduit que les arbres constituant un tas binomial de n clés sont uniquement définis par la représentation de n en binaire.

Par exemple, si $n = 6$, alors $n = 4 + 2 = 2^2 + 2^1$, il faut donc un arbre binomial de niveau 2 et un de niveau 1. Pour $n = 21$, soit $n = 16 + 4 + 1 = 2^4 + 2^2 + 1$, il faut donc un arbre de niveau 4, un de niveau 2 et un de niveau 0.

On en déduit en particulier

Théorème 1.1

Un tas binomial de n clés contient au maximum $\log n$ arbres, de niveau entre 0 et $\log n$ (donc de hauteur entre 0 et $\log n$)

```
def minimum(H):
    indexmin = 0
    for i in range(len(H)):
        if H[indexmin].element.key > H[i].element.key:
            indexmin = i
    return H[indexmin].element
```

Implém 1.2 – Minimum dans un tas binomial

1.2 Opérations sur les tas binomiaux

1.2.1 Recherche du minimum

Par définition d'un arbre binomial, le minimum dans un arbre binomial est en racine. Dans un tas binomial, le minimum est donc sur l'une des racines. On en déduit :

Théorème 1.2 *Minimum d'un tas binomial*

On peut trouver le minimum dans un tas binomial de n clés en $O(\log n)$: Il suffit de regarder toutes les racines des arbres du tas.

On pourra se reporter à la figure 1.2 pour une implémentation possible

1.2.2 Union de deux tas binomiaux

Avant de regarder toutes les autres opérations, concentrons-nous sur l'opération d'union de deux tas binomiaux.

Commençons par le cas particulier où les deux tas sont réduits à un arbre binomial.

Si les deux arbres sont de niveau différents, il suffit de les mettre ensemble côte à côte pour faire l'union des deux tas.

Si on a deux arbres de même niveau, on ne peut pas les mettre côte à côte, puisque deux arbres d'un même tas doivent avoir des niveaux différents.

On effectue alors la *fusion* des deux arbres : Si $\texttt{T}_1$ et $\texttt{T}_2$ sont les deux arbres, et $\texttt{T}_1$ est l'arbre qui a la plus petite racine, on obtient un nouvel arbre en mettant $\texttt{T}_2$ comme fils de $\texttt{T}_1$.

Algorithme 1.3 *Fusion de deux arbres*

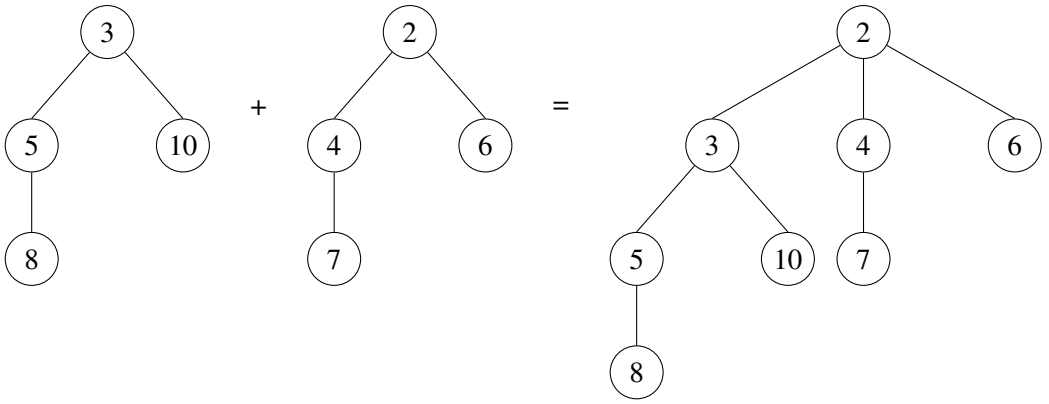
Si $\texttt{T}_1$ et $\texttt{T}_2$ sont deux arbres binomiaux de même niveau k, la fusion de $\texttt{T}_1$ et $\texttt{T}_2$, notée `Fusion(`$\texttt{T}_1$`, `$\texttt{T}_2$`)` est l'arbre binomial de niveau $k+1$ où l'arbre avec la plus petite racine est devenu père de l'autre. Cette opération prend un temps constant $O(1)$: il suffit de tester les deux racines, et d'ajouter un élément à la liste des fils d'un des arbres.

On pourra se reporter à la figure 1.3 pour une implémentation possible.

```
def fusion(A,B):
    """ Fusion de deux arbres binomiaux
    Renvoie le nouvel arbre obtenu """
    if A.element.key < B.element.key:
        A.children = [B]+A.children
        A.level+=1
        B.parent = A
        return A
    else:
        B.children = [A]+B.children
        B.level+=1
        A.parent = B
        return B
```

Implém 1.3 – Fusion de deux arbres binomiaux

♦ **Exemple**



Maintenant qu'on sait faire la fusion de deux arbres, l'union de deux tas est très simple. La meilleure façon de la comprendre est de la voir comme une addition en binaire. Par exemple, supposons avoir un tas binomial de 6 clés (soit un arbre de niveau 2 et un de niveau 1), et un tas binomial de 13 clés (soit un arbre de niveau 3, un de niveau 2 et un de niveau 0). Le nouveau tas aura 19 clés, soit un arbre de niveau 4, un de niveau 0 et un de niveau 1.

Ecrivons ce qui se passe en binaire :

$$\begin{array}{cccccc} & & 0 & 1 & 1 & 0 \\ + & {\scriptstyle 1} & 1_1 & 1 & 0 & 1 \\ \hline & 1 & 0 & 0 & 1 & 1 \end{array}$$

L'arbre de niveau 0 est donc celui correspondant au premier tas. L'arbre de niveau 1 est donc celui correspondant au deuxième tas. L'arbre de niveau 4 est obtenu en fusionnant d'abord les deux tas de niveau 2 (ce qui crée une retenue : un arbre de niveau 3), puis en fusionnant les deux arbres de niveau 3 obtenu pour avoir un arbre de niveau 4.

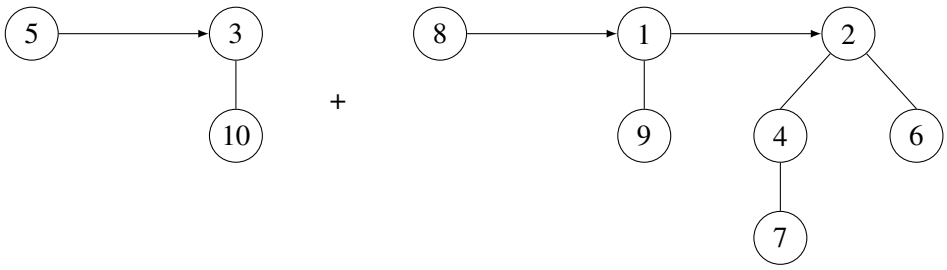
Algorithme 1.4

L'algorithme d'union de deux tas binomiaux fonctionne de la façon suivante :
— Soit $\mathtt{a}_1$ le premier arbre du tas $\mathtt{H}_1$ non traité
— Soit $\mathtt{a}_2$ le premier arbre du tas $\mathtt{H}_2$ non traité
— Soit `retn`, un arbre binomial "retenue", potentiellement inexistant.
L'algorithme fonctionne ainsi :
— Soit k le minimum entre le niveau de $\mathtt{a}_1$,$\mathtt{a}_2$ et la retenue (notons que la retenue, si elle existe, a toujours le niveau minimum)
  — Si un seul des 3 arbres a ce niveau, l'ajouter à la sortie. Dans tous les cas, il n'y a plus de retenue.
  — S'il y en a 2 ou 3, en fusionner deux pour créer une retenue, et ajouter le 3e (s'il existe) à la sortie.

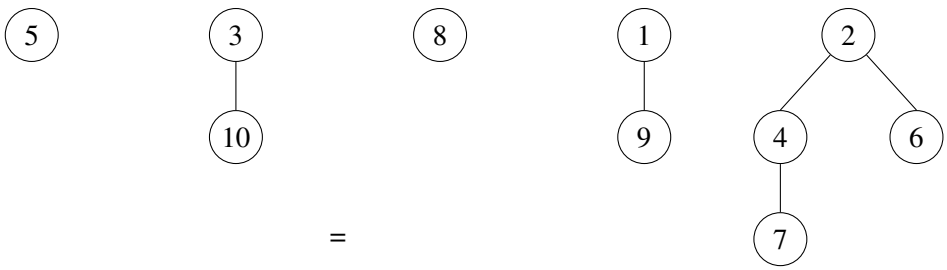
L'algorithme est présenté de façon détaillée à la figure 1.4

♦ **Exemple**

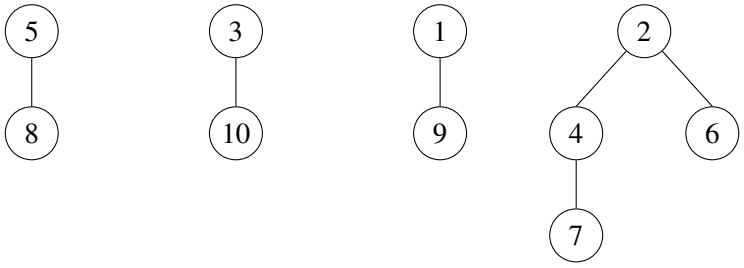
Quand on effectue l'algorithme sur papier, le plus simple est de commencer par transformer les deux tas en deux ensembles d'arbres. On fusionne ensuite les arbres de même niveau tant que c'est possible :



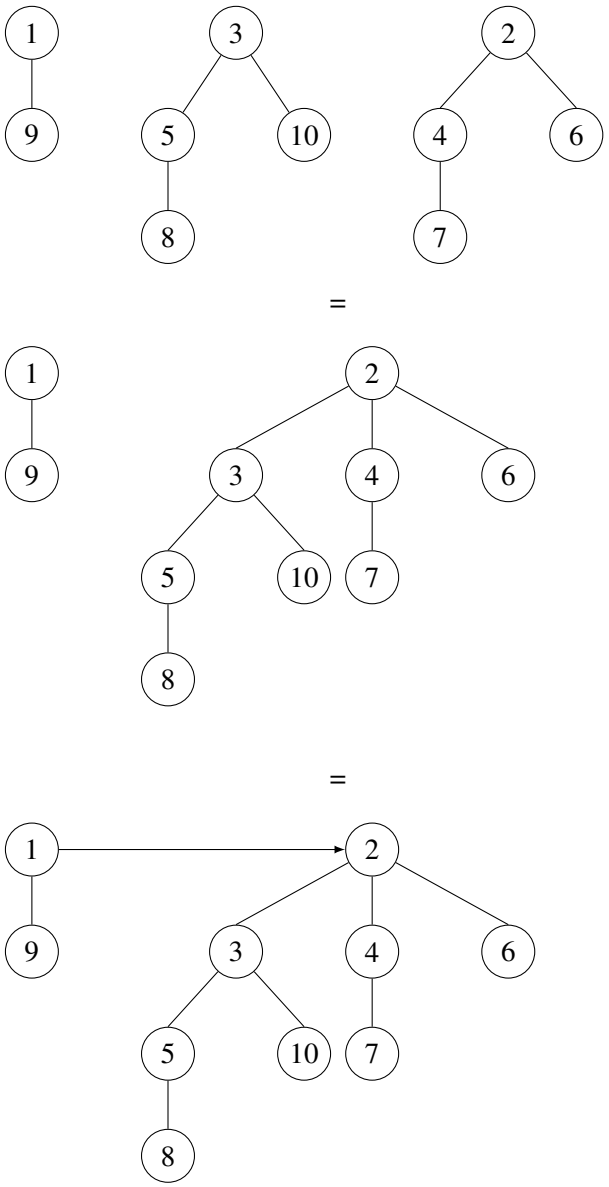
=



=



=



Il est facile de voir qu'en deux étapes de cet algorithme, on progresse dans au moins une des deux listes. Le nombre de passages de la boucle est donc proportionnel au nombre total d'arbres des deux tas binomiaux. Comme un tas de n noeuds contient au maximum $\log n$ arbres, le nombre de passages dans la boucle est donc $O(\log n + \log m)$, où n et m représentent le nombre de noeuds des deux tas. Chaque passage dans la boucle fait un nombre constant d'opérations, d'où :

Théorème 1.3

La complexité de l'opération d'union de deux tas binomiaux de taille n et m est $O(\log n + \log m)$

```
def union(H1, H2):
    """ Union de deux tas binomiaux
    Renvoie le nouveau tas obtenu"""
    retn = None
    output = []
    while len(H1)+len(H2) > 0:
        atraiter = []
        if retn is not None:
            atraiter = [retn]
            if len(H1) > 0 and H1[0].level == retn.level:
                atraiter += [H1.pop(0)]
            if len(H2) > 0 and H2[0].level == retn.level:
                atraiter += [H2.pop(0)]
            retn = None
        else:
            if len(H1) > 0:
                if len(H2) > 0:
                    if H1[0].level == H2[0].level:
                        atraiter = [H1.pop(0), H2.pop(0)]
                    elif H1[0].level > H2[0].level:
                        atraiter = [H2.pop(0)]
                    else:
                        atraiter = [H1.pop(0)]
                else:
                    atraiter = [H1.pop(0)]
            else:
                atraiter = [H2.pop(0)]
        if len(atraiter) > 1:
            retn = fusion(atraiter[0],atraiter[1])
            atraiter.pop(0)
            atraiter.pop(0)
        if len(atraiter) > 0:
            output += atraiter
    if retn is not None:
        output += [retn]
    return output
```

Implém 1.4 – Union de deux tas binomiaux

1.2.3 Opérations qui se déduisent de l'union

Une fois écrit l'algorithme d'union, il est très facile de coder les autres opérations : L'insertion n'est rien d'autre qu'un cas particulier de l'union :

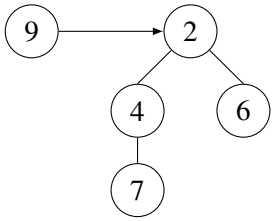
Algorithme 1.5 *Insertion*

L'opération d'insertion d'un élément dans un tas binomial `H` est équivalent à l'union d'un tas avec un élément avec le tas `H`

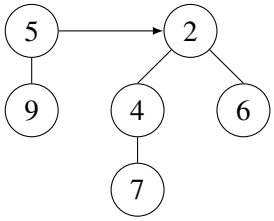
Sa complexité est donc $O(\log n)$ si `H` a n éléments.

♦ **Exemple**

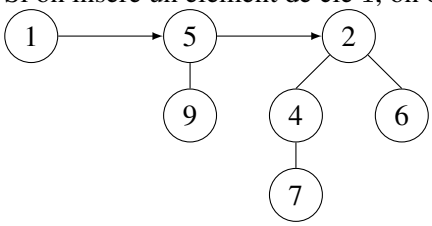
Insérer un élément de clé 5 dans le tas suivant :



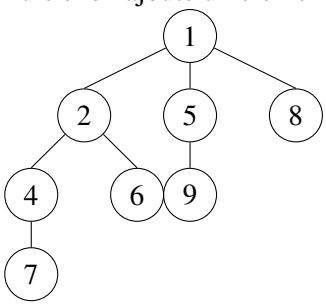
revient donc à faire l'union avec un tas contenant uniquement un seul arbre de niveau 0, constitué d'un seul élément de clé 5. Le résultat est donc :



Si on insère un élément de clé 1, on obtient alors :



Puis si on ajoute un élément de clé 8 :



Une implémentation possible est présentée en figure 1.5.

L'autre opération importante dans une structure de type tas est la suppression de l'élément minimum. L'élément de clé minimum est toujours racine d'un des arbres du tas. Si on supprime cet élément de l'arbre, on a remplacé l'arbre . . . par un tas ! L'extraction du minimum est donc équivalente à l'union de deux tas :

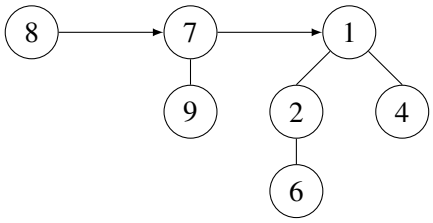
Algorithme 1.6 *Extraction du Minimum*

L'opération d'extraction du minimum dans un tas binomial $\mathtt{H}$ fonctionne ainsi :
- Trouver l'arbre $\mathtt{A}$ du tas qui contient le minimum en racine.
- Soit $\mathtt{H}_1 = \mathtt{H} \setminus \mathtt{A}$.
- Soit $\mathtt{H}_2$ la liste des fils de la racine de $\mathtt{A}$. Il s'agit bien d'un tas binomial
- Le résultat est alors l'union de $\mathtt{H}_1$ et de $\mathtt{H}_2$..

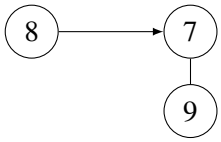
La complexité de cet algorithme est $O(\log n)$ si $\mathtt{H}$ a n éléments : La recherche du minimum coûte $\log n$: $\mathtt{H}_1$ et $\mathtt{H}_2$ étant des sous-tas de $\mathtt{H}$, ils ont moins de n éléments, donc leur union prend un temps inférieur à $O(\log n)$.

♦ **Exemple**

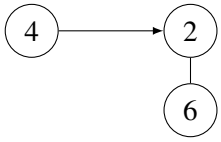
Essayons d'extraire le minimum du tas suivant :



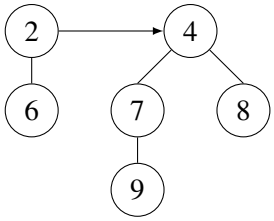
Le minimum est l'élément de clé 1, il faut donc faire la fusion entre le tas :



et le tas :



Le résultat est donc le tas :



L'algorithme est présenté à la figure 1.6

```
def insert(H,x):
    """ Insère dans le tas binomial l'éleéent x
    Renvoie le nouveau tas binomial
    """
    node = ArbreBinomial(x)
    H2 = [node]
    return union(H, H2)
```

Implém 1.5 – Insertion dans un tas binomial

```
def extractmin(H):
    """ Extrait le minimum d'un tas
    Renvoie le nouveau tas et l'élément minimum"""
    indexmin = 0
    for i in range(len(H)):
        if H[indexmin].element.key > H[i].element.key:
            indexmin = i
    A = H.pop(indexmin)
    res = A.element
    H2 = A.children
    for a in H2:
        a.parent = None
    H2.reverse()
    return (union(H,H2), res)
```

Implém 1.6 – Extraction du minimum dans un tas binomial

1.2.4 Opérations classiques

Il nous reste à parler de deux opérations, pour lesquelles nous utilisont uniquement la structure de tas des arbres binomiaux.

La première est l'opération de changement de clé DecreaseKey.

Algorithme 1.7

L'opération `DecreaseKey(H,x,k)` fonctionne de la façon suivante :

— D'abord trouver où se trouver `x` dans le tas (réalisable facilement si on garde un pointeur vers l'arbre binomial où on a inséré `x`)

— Changer la clé du noeud `x`.

— Tant que `x` a une plus petite clé que son père, échanger `x` et son père.

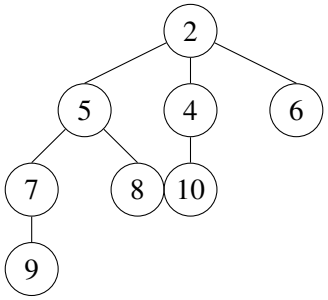
Cette opération a une complexité $O(\log n)$ (sans compter le temps pour trouver `x`)

```
def decreaseKey(H,x,k):
    """ Change la clé du noeud x pour la nouvelle valeur k """
    x.key = k
    n = x.ptr
    while n.parent is not None and \
          n.parent.element.key > n.element.key:
        n.parent.element,n.element=n.element,n.parent.element
        n.element.ptr,n.parent.element.ptr = n,n.parent
        n = n.parent
```

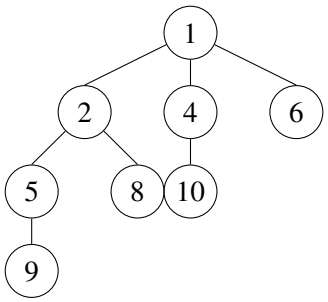
Implém 1.7 – Changement de clé (vers une clé plus petite)

♦ **Exemple**

Prenons le tas suivant :



Si on change la clé noeud de clé 7 pour qu'elle soit maintenant de 1, on obtient :



Une implémentation est proposée figure 1.7.

La dernière opération est l'opération de suppression d'un élément quelconque. On le simule aisément à l'aide des opérations précédentes

Algorithme 1.8

L'opération `delete(H,x)` fonctionne de la manière suivante :
- D'abord trouver où se trouver `x` dans le tas
- Effectuer `DecreaseKey(H, x, `$-\infty$`)` pour que `x` soit en racine.
- Faire `extractMin(H)`

Cette opération a une complexité $O(\log n)$ (sans compter le temps pour trouver `x`)

```
def delete(H,x):
    """ Supprime x du tas"""
    decreaseKey(H,x,-sys.maxint-1)
    return extractmin(H)[0]
```

Implém 1.8 – Suppression d'un élément

Conclusion

L'analyse des algorithmes présentés ci-dessus ne permettent pas d'obtenir les complexités énoncées en introduction : Nous avons montré que l'insertion est en $O(\log n)$ alors que nous l'avons annoncé en $O(1)$. Ce $O(1)$ indiqué est à comprendre dans un sens très particulier, qui fait l'objet du prochain chapitre.

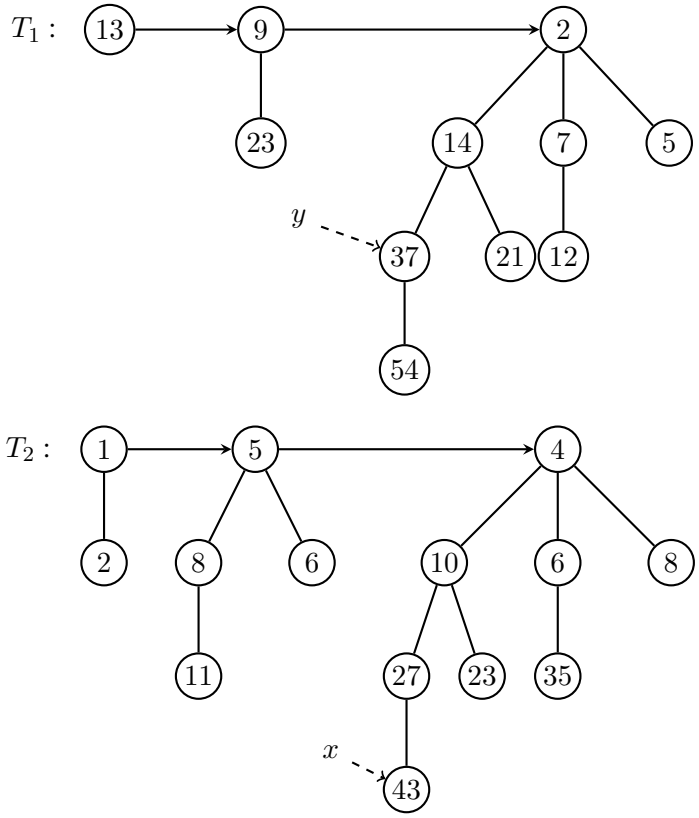
Exercices

(1 - 1) Quels sont les arbres binomiaux d'un tas binomial de $n_1 = 23$ clés, $n_2 = 36$ clés, $n_1 + n_2$ clés ? Commentez les ressemblances / différences du troisième tas avec les deux premiers.

(1 - 2) Donnez une représentation graphique complète d'un tas binomial contenant les clés : 4, 2, 12, 3, 5, 7, 5, 4, 10, 4, 3.

(1 - 3) Réalisez les opérations suivantes sur les tas T_1 et T_2 :

— `Union(T1, T2)`
— `ExtractMin(T1)`
— `DecreaseKey(T2, x, 2)`
— `Delete(T1, y)`



(1 - 4) Donner un exemple (pour n arbitrairement grand) pour lequel la fonction `ExtractMin` va nécessiter au moins $\log n$ étapes de temps, où n est le nombre de clés. Même question avec `DecreaseKey` et `Insert`.

(1 - 5) Dans cet exercice, on suppose donné deux tas binomiaux T_1 et T_2. On cherche à fusionner les deux tas et à trouver le minimum (sans l'enlever).

On propose deux algorithmes différents :

1. Trouver le minimum de T_1, le minimum de T_2, en déduire le minimum. Puis fusionner T_1 et T_2.
2. Fusionner T_1 et T_2, puis trouver le minimum.

Q 1) Expliquer intuitivement (sans preuve), pourquoi le premier algorithme fait toujours un peu plus d'opérations que le deuxième.

Q 2) Trouver un exemple (arbitrairement grand) où le deuxième algorithme est bien plus efficace.

(1 - 6) On suppose qu'un tas binomial de 10 clés contient tous les entiers de 1 à 10. Montrer qu'il y a exactement 8 positions où le nombre 8 peut se trouver. Pour cela :

— Indiquer les deux positions où il ne peut pas se trouver, et justifier pour chacune d'entre elles pourquoi c'est le cas

— Pour les 8 positions restantes, expliquer très brièvement (une seule explication pour les 8 positions à la fois) pourquoi le nombre 8 peut s'y trouver.

(1 - 7) Sous les mêmes hypothèses, trouver toutes les positions où le nombre 2 peut se trouver. Justifier.

(1 - 8) Donner la complexité de la recherche du maximum dans un tas binomial.

(1 - 9) Ecrire un algorithme de tri en utilisant la structure de tas binomial. Calculer sa complexité.

(1 - 10) Pour préparer le cours suivant : Quelle est la complexité totale de l'insertion de n clés dans un tas binomial initialement vide ?

Chapitre 2

COMPLEXITÉ AMORTIE

Dire que la complexité d'une opération sur une structure de données de taille n est en $O(\log n)$ signifie que le temps d'exécution ne dépasse jamais $O(\log n)$ unités de temps. En général, on est même capable de prouver que la complexité est en $\Theta(\log n)$ ce qui signifie qu'on peut trouver des exemples où le temps est effectivement de l'ordre de grandeur de $\log n$ et non pas plus petit.

Cependant, la donnée de la complexité est en générale insuffisante pour savoir si une structure données va être efficace en pratique. Revenons sur l'exemple précédent, et effectuons k fois consécutivement l'opération sur la même structure de données de taille n. Si l'opération est en $O(\log n)$, le temps nécessaire pour effectuer ces k opérations est donc en $O(k \log n)$. Il arrive toutefois régulièrement que le temps, en pratique, soit plus faible, par exemple de l'ordre de grandeur de $O(k + \log n)$, voire moins : C'est le cas par exemple si le pire cas, conduisant à une complexité de l'ordre de grandeur $\log n$, ne peut se produire qu'une fois lors d'une même exécution de l'algorithme.

La notion de complexité amortie permet de formaliser ce phénomène.

Définition 2.1

Soit T une structure de données disposant des opérations op_1, op_2. On dira que la complexité amortie des opérations op_1 et op_2 est $\hat{c}_1$ et $\hat{c}_2$ si, en partant de la structure vide, exécuter k_1 fois l'opération op_1 et k_2 fois l'opération op_2 aura un coût total de $k_1\hat{c}_1 + k_2\hat{c}_2$.

Tout se passe donc comme si, en moyenne temporelle, chaque opération op_1 avait un coût de $\hat{c}_1$, et chaque opération op_2 avait un coût de $\hat{c}_2$: Si certains des appels à op_1 peuvent avoir un coût supérieur à $\hat{c}_1$, ils seront compensés (*amortis*) par des appels à op_2 ou d'autres appels à op_1.

La définition se généralise bien entendu à une structure de données disposant de plus que deux opérations.

Dans la suite, nous examinerons une structure de données très simple.

La structure est un tableau de p cases, représentant un nombre écrit en binaire (initialement égal à 0). La seule opération possible est l'opération `incr()` qui incrémente le nombre écrit en binaire.

Le code de la fonction `incr()` est illustré à la figure 2.1.

Dans le pire des cas, par exemple quand le tableau est rempli de 1, cet algorithme va lire tout le tableau. Sa complexité est donc $O(p)$. Cependant, avant de pouvoir arriver à ce pire cas, il faut avoir avoir appelé la fonction `incr()` un grand nombre de fois (de l'ordre de 2^{p-1}). Sur 2^{p-1} appels consécutifs, un seul aura cette complexité maximale. On va démontrer, de plusieurs manières différentes que la complexité amortie de l'opération d'incrémentation est $O(1)$: Si on fait K incrémentations, le coût total est $O(K)$ (et non pas $O(Kp)$)

```
int incr(){
  int i = 0;
  while (i < p && T[i] == 1)
  {
    T[i] = 0;
    i++;
  }
  if (i < p)
    T[i] = 1;
}
```

FIGURE 2.1 – Code de la fonction d'incrémentation

2.1 Calcul mathématique

La première méthode pour prouver que la fonction `incr()` est en $O(1)$ amorti consiste à calculer exactement le coût de K opérations consécutives.

Il existe plusieurs façons de faire un calcul exact. Le premier procède ainsi :

— Si le nombre est impair, une seule opération sera effectuée
— Si le nombre est multiple de 2 mais pas de 4, deux opérations seront effectuées
— Si le nombre est multiple de 4 mais pas de 8, trois opérations seront effectuées
— ..

On rappelle que le compteur part initialement de la valeur 0. Quand on exécute K fois l'instruction `incr()`, on aura donc :

— au maximum $K/2$ fois une seule opération effectuée
— au maximum $K/4$ fois deux opérations effectuées
— au maximum $K/8$ fois trois opérations effectuées
— ...

Le temps total pour effectuer toutes les opérations est donc au maximum de

$$\frac{K}{2} + \frac{2K}{4} + \frac{3K}{8} + \frac{4K}{16} + \ldots = K(\sum_{i=1}^{\infty} \frac{i}{2^i})$$

Pour finir l'analyse, il faut savoir calculer le terme de droite. On peut montrer qu'il vaut 2 par des raisonnements mathématiques assez simple.

On peut également essayer de faire l'analyse du temps de calcul d'une autre façon. Le temps de calcul est proportionnel au nombre de cases du tableau visité.

— La première case est toujours visitée
— La deuxième case est visitée une fois sur 2
— La troisième case est visitée une fois sur 4

Donc le nombre total de cases visitées est inférieur à

$$K + \frac{K}{2} + \frac{K}{4} + \ldots = K(1 + \frac{1}{2} + \frac{1}{4} + \ldots) = 2K$$

On a donc démontré que le temps pour effectuer K opérations est de l'ordre de K. On peut donc dire que l'opération `incr` est en temps $O(1)$.

2.2 Méthode comptable

La méthode comptable utilise une analogie économique : Au lieu de parler de temps d'exécution (mesuré par exemple en secondes), on va parler du coût d'exécution (mesuré en euros).

Pour l'exemple du compteur qui peut s'incrémenter, on va estimer, comme précédemment, que le coût d'exécution est égal au nombre de cases visitées.

L'idée de la méthode comptable est de considérer le coût amorti d'une opération comme la surcharge du coût de certaines opérations, de sorte qu'à certaines étapes de l'exécution, on dispose d'un *capital*, qui va nous permettre d'avoir à payer moins que prévu pour certaines opérations.

L'idée est de montrer qu'à chaque étape, on a suffisament de capital pour payer les opérations chères (longues)

Dans l'exemple de l'incrémentation, on va procéder de la façon suivante. On va supposer qu'un accès à une case du tableau coûte 1 euro. On va montrer que si on fait payer 2 euros à toutes les instructions `incr`, le capital (la différence entre ce qu'on a payé, et le coût réel) est toujours positif, de sorte que n opérations auront toujours un coût inférieur à $2n$.

A noter qu'on a donc surchargé le coût de l'opération dans le cas où la première case du tableau est un 1 (Le coût aurait du être seulement de 1 euro, et on a payé 2) et qu'on l'a au contraire sous-estimé dans les autres cas.

Dans le cas où la première case du tableau est un 1, on paye 2 euros plutôt qu'un euro. Il faut donc comprendre à quoi correspond cet euro supplémentaire, et quelle future opération il sert à payer.

Avec un peu d'intuition, on finit par réaliser la chose suivante : A chaque incrémentation, une case du tableau passe de la valeur 0 à la valeur 1 et plusieurs cases passent de la valeur 1 à la valeur 0. Mais, pour qu'une case passe de la valeur 1 à la valeur 0, il a fallu qu'auparavant elle soit passée de la valeur 0 à la valeur 1.

On en déduit donc d'où provient ce coût de 2 euros : On paye 1 euro pour le changement d'une case de 0 en 1, et 1 euro d'avance pour le futur changement de cette même case de 1 en 0.

A chaque étape, on a donc assez de 2 euros pour payer, puisque tous les changements de 1 en 0 ont été payés en avance lors des étapes précédentes.

Si on veut essayer de rendre le raisonnement précédent rigoureux, on peut essayer de calculer le capital exact qu'on a à chaque étape. Dans cet exercice, il est facile à déterminer : il est exactement égal au nombre de cases du tableau qui sont à la valeur 1. En effet, supposons que le capital soit égal, à un moment donné, au nombre q de cases du tableau qui sont à la valeur 1 et effectuons une incrémentation :

- L'opération nous coûte exactement $k+1$ euros, où k est le nombre de cases à 1 qu'on a visité.
- Le nombre de cases à 1 du tableau passe alors de q à $q-k+1$.
- Comme on a payé 2 euros, notre capital diminue de $k+1-2$ euros, il passe donc de q à $q-(k-1) = q-k+1$.
- Donc à l'étape suivante, le capital est toujours bien égal au nombre de cases du tableau qui sont à la valeur 1.

A noter qu'en fait, le capital est légèrement supérieur : c'est dû au fait que, lorsque le tableau est entièrement rempli de 1, on le transforme en un tableau entièrement rempli de 0,ce qui coûte p euros et non pas $p+1$ euros. On gagne donc 1 euro toutes les 2^p étapes.

Remarquons que, contrairement à la méthode précédente, la méthode comptable n'est pas automatique, et nécessite de trouver, d'une façon ou d'une autre, les bons paramètres pour rendre l'analyse pertinente.

2.3 Méthode du potentiel

La méthode du potentiel est très similaire à la méthode comptable, avec une différence majeure : Dans la méthode comptable, on définit a priori les coûts amortis de chacune des opérations, et on vérifie que le capital est bien toujours positif. Dans la méthode du potentiel, on définit d'abord le capital, et on en déduit les coûts amortis des opérations.

La méthode du potentiel associe, à chaque étape, à la structure de données un *potentiel* ϕ tel que

— ϕ est toujours positif

— ϕ est nul au départ.

Définition 2.2

> On appelle coût amorti de l'opération op la somme $\hat{c}_{op}$ du coût réel c_{op} de l'opération et de la différence de potentiel qu'elle occasionne (potentiel après l'opération - potentiel avant l'opération)

Théorème 2.1

> $\hat{c}$ est bien un coût amorti au sens de la définition initiale

En effet, considérons une suite de n opérations sur une structure de données initialement vides. Notons c_i le coût de la $i$ème opération, et ϕ_i le potentiel à l'issue de la i-ème opération.

Le coût total des n opérations est donc $c_1 + c_2 + \dots c_n$.

Le coût amorti est

$$\hat{c}_1+\hat{c}_2+\dots \hat{c}_n = (c_1+\phi_0-\phi_1)+(c_2+\phi_1-\phi_2)+\dots+(c_n+\phi_{n-1}-\phi_n) = c_1+c_2\dots+c_n+\phi_0-\phi_n \leq c_1+\dots c_n$$

Il est donc bien inférieur au coût réel.

Revenons maintenant à l'exemple de l'incrémentation. Pour commencer l'analyse, il faut définir le potentiel de la structure. Vu les calculs fait précédemment, il parait raisonnable de choisir comme potentiel ϕ le nombre de cases du tableau qui valent 1. Dans ce cas :

— ϕ est bien nul au départ

— ϕ est bien toujours positif.

Prenons maintenant une opération d'incrémentation à une étape donnée. Son coût c est exactement $k + 1$, où k est le nombre de 1 visités dans le tableau. Le coût $\hat{c}$ est donc :

$$\hat{c} = k + 1 + \phi_{apres} - \phi_{avant} = k + 1 + (\phi_{avant} - k + 1) - \phi_{avant} = 2$$

On retrouve bien que le coût amorti est 2.

Remarquons de nouveau que la méthode du potentiel nécessite un apriori sur le résultat : il faut avoir une idée de la bonne fonction potentiel à utiliser. Une bonne façon de penser est de considérer le potentiel comme une entropie, représentant l'aspect chaotique du système. Au départ, le système est parfait, et les opérations peu coûteuses le rendent plus chaotique. Les opérations coûteuses, au contraire, rendent le système plus ordonné, et donc leur coût est compensé par la décroissance de l'entropie.

2.4 Application - Compteur binaire

On revient sur l'exemple du compteur binaire représenté par un tableau de taille p mais on a maintenant 2 opérations :

— Incrémentation

— Décrémentation

On reprend l'analyse du potentiel. Si on prend le même potentiel, rien ne change pour l'analyse de l'incrémentation, elle est donc toujours de coût amorti 2.

Passons à la décrémentation. Le coût de la décrémentation est $k+1$ où k est le nombre de 0 visités dans le tableau. Le potentiel ϕ varie alors de ϕ_{avant} à $\phi_{avant} - 1 + k$ (puisque tous les 0 sont transformés en 1, et le 1 est transformé en 0)

Le coût d'une décrémentation est donc

$$\hat{c} = k + 1 + \phi_{apres} - \phi_{avant} = k + 1 + k - 1 = 2k \leq 2p$$

On en déduit donc que l'incrémentation est en $O(1)$ et la décrémentation en $O(p)$.

En utilisant exactement la même analyse, on montre que :

Théorème 2.2

> Dans un tas binaire muni des opérations d'insertion et extraction du minimum, le coût amorti de l'insertion est $O(1)$ et le coût amorti de l'extraction du minimum est $O(\log n)$.

2.5 Tableau de taille variable

On cherche maintenant à analyser un tableau de taille variable : C'est à dire un tableau dont la taille initiale est 1 et qui est muni des deux opérations suivantes :

— `get(i)` qui donne la valeur de la $i$ème case

— `insert(x)` qui insère x à la fin du tableau. Si le tableau est plein, on crée un nouveau tableau dont la taille est le double de la taille courante, et on recopie dans le tableau.

Il est clair que toute la complexité est dans l'opération d'insertion, en particulier dans le cas où le tableau est plein. Cependant, il est clair que cette situation n'arrive pas souvent. On va donc démontrer, en suivant les 3 méthodes précédentes :

Théorème 2.3

> Les opérations get et insert sont de coût amortis $O(1)$.

Pour analyser le coût des opérations, on va supposer que la lecture/écriture d'une case du tableau est de coût 1, et on néglige le coût de création d'un tableau. On a donc

— L'opération get est de coût 1 (1 lecture)

— L'opération insert est de coût 1 s'il n'y a pas de recopie et de coût $2n+1$ sinon, où n est la taille du tableau avant insertion (n recopies (donc n lectures et n écritures), puis insertion de l'élément (1 écriture))

Méthode mathématique Supposons avoir fait p insertions et q get. Les q opérations get coûtent q en tout, soit 1 chacune.

Si on a fait p insertions, on a un coût total de p, auquel il faut ajouter :

— 1 recopie d'un tableau de taille 1 qui nous a coûté 2
— 1 recopie d'un tableau de taille 2 qui nous a coûté 4
— 1 recopie d'un tableau de taille 4 qui nous a coûté 8
— ...

Comme on a p éléments dans le tableau, on a donc fait au maximum $\log p$ recopies.

Le coût total de toutes les recopies est donc au maximum :

$$2 + 4 + 8 + 16 + \ldots + 4 \times 2^{\log p} = 2(1 + 2 + 4 + 8 + \ldots + 2^{\log p}) = 2(2^{\log p+1} - 1) = 4p - 2$$

L'opération get a donc un coût amorti de 1 et l'opération insert de $1 + 4 = 5$.

Méthode comptable Pour la méthode comptable, on peut essayer de comprendre d'où vient ce 5 : Il veut dire qu'on paie 4 euros de trop sur chaque opération, mis à part sur les opérations de recopies. Oublions donc les opérations d'insertion et regardons uniquement les opérations de recopies.

Plaçons nous lors d'une opération de recopie d'un tableau de taille n. Si on suppose que tout s'est bien passé avant, notre capital était positif avant l'ajout des $n/2$ derniers éléments. Lorsqu'on a ajouté les $n/2$ autres éléments, qui ont rempli le tableau, on a à chaque fois payé 5 alors qu'on avait besoin que de 1. Donc notre capital à augmenté de $4 \times n/2 = 2n$. Ca tombe bien, c'est exactement le capital dont nous avons besoin pour recopier les n éléments, donc tout va bien.

Méthode potentiel Définissons le potentiel ϕ d'un tableau comme :

$$\phi = 2occupe(T) - 2libre(T) + 2$$

Le potentiel est donc toujours positif, puisque le tableau est toujours à moitié rempli (sauf au début, d'où le $+2$), et au départ il vaut 0.

Une opération `get` a un coût de 1 et ne change pas le potentiel :

$$\hat{c}_{\texttt{get}} = 1 + \phi_{apres} - \phi_{avant} = 1$$

Donc une opération `get` a un coût amorti de 1.

Regardons maintenant une opération `insert` :

— Si l'insertion n'occasionne pas de redimensionnement, on paie 1 pour l'insertion, et la différence de potentiel est de 4 (une case occupée de plus, une case libre de moins), on obtient donc un cout total de
$$\hat{c}_{\texttt{insert}} = 1 + \phi_{apres} - \phi_{avant} = 5$$
— Si l'insertion occasionne un redimensionnement. Soit n la taille du tableau avant redimensionnement. On a donc :
  — Coût de l'insertion $1 + 2n$.
  — Potentiel avant insertion : $2n + 2$ (le tableau est rempli)
  — Potentiel après insertion : 6 (le tableau fait $2n$ cases et $n + 1$ cases sont occupées)

  Donc un coût amorti de
$$\hat{c}_{\texttt{insert}} = 2n + 1 + \phi_{apres} - \phi_{avant} = 2n + 1 + 6 - (2n + 2) = 5$$

Dans tous les cas, on obtient un nombre inférieur à 5. Le coût amorti vaut donc bien 5.

Exercices

(2 - 1) On rappelle qu'une pile `P` est une structure de données disposant des opérations suivantes :
— `push(P,x)` qui ajoute en tête de la pile `P` l'élément `x`
— `pop(P)` qui supprime l'élément en tête de pile et renvoie sa valeur.
— `empty(P)` qui teste si la pile est vide.
On suppose dans toute la suite avoir une implémentation des piles de sorte que les 3 opérations précédentes fonctionnent chacune en une unité de temps exactement.
On cherche à implémenter à l'aide des piles la structure de données File, qui dispose des opérations suivantes :
— `enfile(F,x)` : ajoute l'élément `x` en dernière position de la file `F`
— `defile(F)` : supprime et renvoie l'élément en première position de la file
— `vide(F)` qui teste si la file est vide.

Q 1) Expliquer comment simuler cette structure de données en utilisant 2 piles $\texttt{P}_1$ et $\texttt{P}_2$.

Dans toute la suite, on utilise l'implémentation proposée à la question précédente.

Q 2) Soit p_1 et p_2 le nombre d'éléments dans les piles $\texttt{P}_1$ et $\texttt{P}_2$ à un moment donné de l'algorithme. Estimez la complexité des opérations `enfile` et `defile` en fonction de p_1 et p_2.

Q 3) Donner un exemple de `n` opérations, partant d'une file initialement vide, de sorte que l'une de ces `n` opérations mette un temps proportionnel à n.

On va maintenant montrer que la complexité amortie des opérations `enfile`, `defile` est en fait respectivement de `4` et `2`.

Q 4) Justifier cette complexité amortie par la méthode comptable.

Q 5) On considère comme potentiel ϕ la quantité $3p_1$. Calculer la complexité amortie des deux opérations `enfile` et `defile` en utilisant ce potentiel.

On cherche maintenant à implémenter une structure de données `S` avec les 5 opérations suivantes :
— `enfiletête(S,x)` : ajoute l'élément `x` en première position de `S`
— `enfilequeue(S,x)` : ajoute l'élément `x` en dernière position de `S`
— `defiletête(S)` : supprime et renvoie l'élément en première position de `S`
— `defilequeue(S)` : supprime et renvoie l'élément en dernière position de `S`
— `vide(S)` qui teste si `S` est vide.

Q 6) Expliquer pourquoi l'implémentation précédente ne permet pas d'obtenir de bonnes complexités pour les nouvelles opérations. En particulier, donner un exemple de n opérations consécutives dont le temps total d'exécution est de l'ordre de $O(n^2)$.

Q 7) Trouver comment simuler intelligemment cette structure de données en utilisant 2 piles $\texttt{P}_1$ et $\texttt{P}_2$, et quelques piles auxiliaires qui ne servent que provisoirement.

Q 8) Montrer que toutes les opérations sont en complexité amortie $O(1)$. On utilisera la méthode du potentiel après avoir réfléchi à un potentiel adapté.

(2 - 2) *(Compteurs)*

On cherche à implémenter un compteur (qui peut être positif ou négatif), initialement à 0 sur lequel les trois opérations suivantes sont possibles :

— `inc()` qui incrémente le compteur
— `dec()` qui le décrémente
— `zero()` qui teste si le compteur est nul.

Le compteur peut prendre des valeurs très grandes, de sorte qu'il est nécessaire de le représenter en binaire.

Q 1) On suppose dans cette question, et cette question seulement, que le compteur est toujours positif. Montrer qu'une implémentation triviale du compteur c par un tableau `T` où `T[i]` est la valeur du `i`-ème bit de c en binaire, n'est pas efficace. Donner en particulier une série de n opérations pour laquelle le temps total pour les n opérations est de l'ordre de $n \log n$.

On décide de représenter le compteur par deux tableaux `P` et `N`, de sorte que le compteur est représenté par la différence des deux nombres en binaire représentés par les tableaux. On ajoute la contrainte suivante : Il est impossible pour `P[i]` et `N[i]` de valoir tous les deux 1.

Q 2) Ecrire les opérations `inc()` et `dec()`

Q 3) Ecrire l'opération `zero()`

Q 4) Montrer que la complexité amortie des 3 opérations est $O(1)$.

Chapitre 3

UNION-FIND

On s'intéresse dans ce chapitre à une structure de données qui permet de représenter des partitions des entiers de 0 à $n-1$, c'est à dire une division des entiers de 0 à $n-1$ en plusieurs ensembles, et munies des deux opérations suivantes :

— `Find(i)` Renvoie l'ensemble où se trouve l'entier `i`

— `Union(i,j)` Fais la fusion des ensembles où se trouvent i et j.

Au départ, tous les entiers de 1 à n sont dans des ensembles différents. Il y a bien entendu une opération de création de la structure, dont nous ne parlerons pas.

Pour l'opération `Find`, la façon dont l'ensemble est représentée importe peu. La seule chose qui importe est que `Find(i) = Find(j)` si et seulement si `i` et `j` sont dans le même ensemble.

♦ **Exemple**

Prenons $n = 6$.

Au départ, nous avons donc 6 ensembles

Après l'opération `Union(1,3)` on obtient :

Après l'opération `Union(2,5)` on obtient :

Après l'opération `Union(4,3)` on obtient :

Après l'opération `Union(3,5)` on obtient :

L'intérêt de cette structure de données n'est pas forcément évident a priori. Nous verrons des exemples d'application dans le chapitre suivant.

```
def create(n):
    return [i for i in range(n)]

def find(T,i):
    return T[i]

def union(T,i,j):
    label = T[i]
    tochange = T[j]
    for l in range(len(T)):
        if T[l] == tochange:
            T[l] = label
```

Implém 3.1 – Une implémentation possible d'union find

3.1 Implémentations tableau/liste

Une première implémentation naïve utilise un tableau `T`.

`T[i]` représente l'ensemble auquel appartient `i`.

Au départ, on a donc `T[i] = i`. `Find(i)` est alors égal à `T[i]`. La seule opération compliquée est l'opération d'union : `Union(i,j)` doit chercher tous les éléments du tableau dont la valeur est là même que celle de `T[i]` et la mettre à `T[j]`.

♦ **Exemple**

Au départ :

i	0	1	2	3	4	5
T[i]	0	1	2	3	4	5

Après `Union(1,3)`

i	0	1	2	3	4	5
T[i]	0	1	2	1	4	5

Après `Union(2,5)`

i	0	1	2	3	4	5
T[i]	0	1	2	1	4	2

Après `Union(4,3)` :

i	0	1	2	3	4	5
T[i]	0	4	2	4	4	2

Après `Union(3,5)` :

i	0	1	2	3	4	5
T[i]	0	4	4	4	4	4

On trouvera une implémentation à la figure 3.1.

Il est facile d'estimer la complexité des opérations :

Théorème 3.1

La complexité des opérations Find et Union est respectivement $O(1)$ et $O(n)$.

On comprend assez aisément comment optimiser cette structure de données : On perd beaucoup de temps à chercher tous les éléments dans le tableau qui ont la même valeur que `T[i]`.

Pour résoudre le problème, on va complexifier un peu la structure de données : On représente les ensembles par des listes (simplement chaînées par exemple), et `T` est maintenant un tableau de pointeurs : `T[i]` est uin de pointeurs vers la liste correspondant à l'ensemble.

La fonction Find ne change pas. La fonction Union `(i,j)` fusionne les listes `T[i]` et `T[j]` et change les listes `T[k]` pour tous les `k` égaux à `T[j]`, donc tout ceux de la liste `T[j]`.

♦ **Exemple**

Initialement ;

i	0	1	2	3	4	5
T[i]	l_0	l_1	l_2	l_3	l_4	l_5

l_0 : | 0 | l_1 : | 1 | l_2 : | 2 | l_3 : | 3 | l_4 : | 4 | l_5 : | 5 |

Après `Union(1,3)` :

i	0	1	2	3	4	5
T[i]	l_0	l_1	l_2	l_1	l_4	l_5

l_0 : | 0 | l_1 : | 1 | 3 | l_2 : | 2 | l_4 : | 4 | l_5 : | 5 |

Après `Union(2,5)`

i	0	1	2	3	4	5
T[i]	l_0	l_1	l_2	l_1	l_4	l_2

l_0 : | 0 | l_1 : | 1 | 3 | l_2 : | 2 | 5 | l_4 : | 4 |

Après `Union(4,3)` :

i	0	1	2	3	4	5
T[i]	l_0	l_4	l_2	l_4	l_4	l_2

l_0 : | 0 | l_2 : | 2 | 5 | l_4 : | 4 | 1 | 3 |

Et après `Union(3,5)` :

i	0	1	2	3	4	5
T[i]	l_0	l_4	l_4	l_4	l_4	l_4

l_0 : | 0 | l_4 : | 4 | 1 | 3 | 2 | 5 |

Un exemple d'implémentatione est présenté à la figure 3.2

La complexité des opérations n'a a priori pas changé : Dans le pire cas possible, on se retrouve à changer beaucoup d'éléments du tableau, ce qui coûte cher.

On peut cependant trouver une heuristique assez simple qui va nous donner une bonne complexité *amortie*. L'heuristique est la suivante : Quand on fusionne deux listes, on fusionne toujours la petite liste à la grande liste et non pas le contraire.

On va maintenant démontrer que la complexité amortie est bonne (en supposant que la concaténation de deux listes se fait en temps 1)

```
def create(n):
    return [[i] for i in range(n)]

def find(T,i):
    return T[i]

def union(T,i,j):
    first = T[i]
    second = T[j]
    if first != second:
        first += second
        for k in second:
            T[k] = first
```

Implém 3.2 – Une autre implémentation possible d'union find

Théorème 3.2

Si on fusionne toujours la petite liste à la grande liste, la complexité amortie de Find est $O(1)$ et celle de Union est de $O(\log n)$ (où n est le nombre d'éléments) : m opérations Union et k opérations Find ont une complexité de $O(k + m \log n)$

Preuve : On va utiliser une méthode inspirée de la méthode mathématique vu au chapitre précédent. Le lecteur avisé pourra essayer de trouver une preuve en utilisant la fonction potentiel en utilisant le potentiel suivant :

$$l \log n - \sum_k a_k \log a_k$$

où l est le nombre de liste, et a_k le nombre d'éléments dans la k-ème liste. Le raisonnement reste cependant non trivial avec la fonction potentiel.

On raisonne donc autrement. Il est clair que la complexité de la fonction Union est directement proportionnel au nombre d'éléments du tableau qui changent de valeur.

Supposons faire m opérations unions et k opérations Find. Comme les opérations Find ne changent rien à la structure de données, on se concentre sur les m opérations union. Pour calculer la complexité de ces opérations, il suffit donc de compter, pour chaque case qui a changé de valeur, combien de fois elle a changé de valeur.

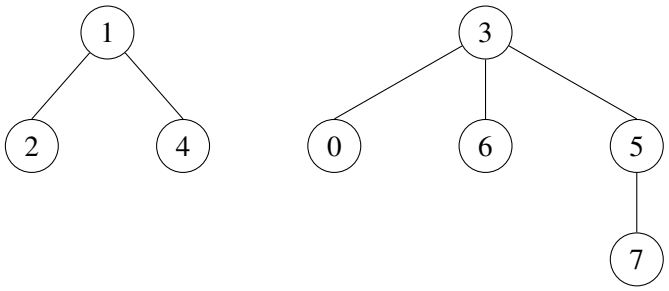
Un raisonnement rapide montre d'abord qu'au plus $2m$ cases du tableau ont changé : A chaque étape d'une union, on fusionne au maximum 2 listes, donc le nombre d'éléments initiaux impactés par une opération de fusion est au maximum de $2m$.

Prenons maintenant un de ces éléments. A chaque fois qu'il a changé de valeur, la taille de l'ensemble dans laquelle il se trouve a au minimum doublé. Par conséquent, chaque case du tableau ne peut changer de valeur qu'au plus $\log n$ fois.

Le nombre de changements de valeur d'une case du tableau en m opérations est donc au plus $m \log n$, ce qui nous donne bien une complexité amortie de $\log n$. ■

3.2 Implémentation arborescente

On peut encore diminuer la complexité de ces opérations avec une autre structure de données. On représente maintenant chaque ensemble par un arbre. Par exemple, si à un moment de la vie de la structure, les entiers $1, 2, 4$ se trouvent ensemble, et les entiers $0, 3, 5, 6, 7$ se trouvent ensemble, on pourra avoir par exemple :



Contrairement à la représentation classique d'un arbre, où chaque noeud a une liste de ses fils, on va utiliser une représentation dans l'autre sens : Chaque noeud a uniquement un pointeur vers son père.

Dans ce cas, toute la structure de données peut être donnée par un tableau : `T[i]` est le numéro du noeud qui est le père de i (avec `T[i] = i` si `i` n'a pas de père).

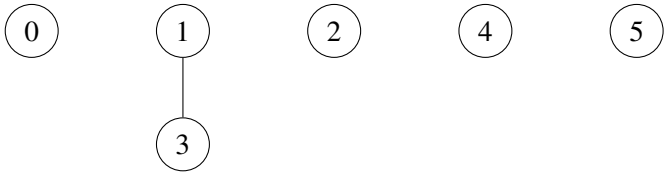
L'exemple ci-dessus peut par exemple s'écrire

i	0	1	2	3	4	5	6	7
T[i]	3	1	1	3	1	3	3	5

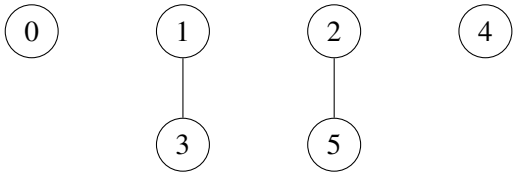
Une fois défini la structure de données les opérations sont simples : L'opération `Find(i)` prend le père de `i` jusqu'à tomber sur une racine. L'opération `Union(i,j)` commence par trouver la racine des deux arbres correspondant à `i` et `j`. S'ils sont distincts, on rend l'un fils de l'autre.

♦ **Exemple**

Après `Union(1,3)` :



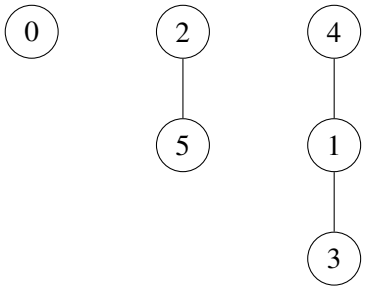
Après `Union(2,5)` :



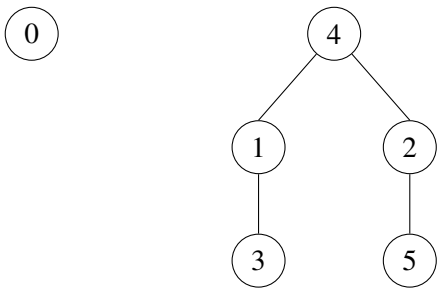
Après `Union(4,3)` :

```
def create(n):
    return [i for i in range(n)]

def find(T,i):
    while T[i] != i:
        i = T[i]
    return i

def union(T,i,j):
    first = find(T,i)
    second = find(T,j)
    if first != second:
        T[second] = first
```

Implém 3.3 – Implémentation arborescente d'union find



Et après `Union(3,5)` :



On trouvera une implémentation à la figure 3.3

Cette structure de données reste cependant peu efficace : On peut facilement trouver des exemples où l'arbre est très déséquilibré : Dans ce cas, la fonction `Find` n'est plus de complexité $O(1)$ mais maintenant de complexité $O(n)$.

Théorème 3.3

Pour l'implémentation précédente de la structure de données Union-Find, la complexité de Find et de Union sont en $O(n)$.

On peut cependant utiliser la même astuce que précédemment :

Théorème 3.4

Considérons l'implémentation arborescente d'Union-Find, avec l'heuristique :
— On relie toujours l'arbre le plus petit à l'arbre le plus grand
Alors Union et Find ont une complexité $O(\log n)$.

Preuve : La complexité de l'Union étant du même ordre de grandeur de deux Find, concentrons nous sur Find. La complexité de Find est directement reliée à la hauteur des arbres obtenus.

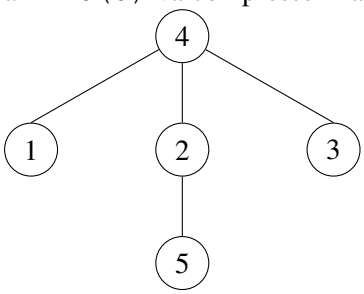
Pour calculer cette hauteur, on introduit le rang $R(v)$ d'un noeud v. Le rang d'une feuille est 0, et le rang d'un noeud est un de plus que le rang de ses fils. Il est clair que le rang de la racine d'un arbre n'est rien d'autre que la hauteur de l'arbre.

La remarque importante est la suivante : Si un noeud v est de rang r alors il a 2^r descendants (lui compris). La preuve est pas récurrence : C'est vrai pour $r = 0$. Si on prend un noeud v de rang r, il a un fils w de rang $r - 1$. A un moment donné de l'exécution, w a été ajouté comme fils de v. Si cette situation arrive c'est que v et w étaient tous deux racine d'una rbre. A ce moment là, w avait déjà 2^{r-1} descendants (une fois qu'il n'est plus racine, il ne peut plus gagner de descendants). Comme on a ajouté w à v et non le contraire, c'est que l'arbre correspondant à v avait au moins autant de noeuds que l'arbre correspondant à w. L'arbre correspondant à v avait donc au moins 2^{r-1} descendants à ce moment là, ce qui en fait donc au moins 2^r une fois qu'on lui ajoute w.

On en déduit maintenant le théorème : Un noeud d'un arbre ne peut avoir que n descendants en tout, donc est de rang au plus $\log n$. Donc les arbres ont une profondeur au plus $\log n$, et Union et Find sont bien en $O(\log n)$. ■

On peut faire encore mieux, en rendant l'arbre le plus plat possible, puisque c'est la hauteur de l'arbre qui donne la complexité des algorithmes.

Pour cela, on introduit une nouvelle heuristique : A chaque fois qu'on appelle la fonction Find, on en profite pour *compresser* l'arbre, en reliant tous les noeuds parcours à la racine. Si on essaie sur l'exemple précédent, la seule différence est tout à la fin : Lorsqu'on exécute `Union(3,5)`, l'appel à `Find(3)` va compresser l'arbre pour que 3 soit un fils de 4 avant l'union. Le résultat sera donc :



Il faut bien remarquer que, maintenant, l'opération Find n'est plus anodine : Si on fait Find, on peut aplatir la structure de l'arbre.

Une implémentation possible est présentée à la figure 3.4

On va maintenant démontrer qu'avec cette heuristique, la structure de données est plus efficace. Mais avant, nous avons besoin de deux définitions.

```
def find(T,i):
    j = i
    while T[j] != j:
        j = T[j]
    root = j
    while T[i] != i:
        k = T[i]
        T[i] = root
        i = k
    return root
```

Implém 3.4 – Compression de chemins

Définition 3.1 $\log^\star$

On note $\log^\star n$ le nombre de fois où il faut appliquer $\log$ pour avoir une valeur inférieure à 1. On note $2^{\uparrow k}$ pour 2 puissance 2 puissance 2 puissance... k fois.

On a par exemple :

— $2 \uparrow 3 = 16$

— $\log^\star 16 = 3$

— $\log^\star 17 = 4$

— $\log^\star x = k$ pour k entre $2^{\uparrow k-1} + 1$ et $2^{\uparrow k}$

— $\log^\star 10^{19000} = 6$

En pratique, vu le dernier résultat, cela veut dire que le $\log^\star$ de tout nombre raisonnable sur un ordinateur est toujours inférieur à 6.

Théorème 3.5

Considérons l'implémentation arborescente d'Union-Find, avec les heuristiques suivantes :

— On relie toujours l'arbre le plus petit à l'arbre le plus grand

— On compresse les chemins

Alors Union, Find, et MakeSet, ont une complexité amortie $O(\log^\star n)$: Si on fait k opérations find et m opérations union, le temps total est $O((n + 2m + k) \log^\star n)$.

En théorie, ce n'est donc pas une structure de données avec des opérations en temps constant $(O(1))$ mais en pratique c'est tout comme.

Preuve : On va réutiliser la notion de rang introduite précédemment. La difficulté est qu'un noeud peut perdre des fils, mais on peut par exemple définir le rang d'un noeud comme un de plus que le maximum des rangs des noeuds qui sont un jour l'un de ses fils. On se convainc assez facilement qu'on obtient la même notion de rang que précédemment.

En particulier :

- $R(v) \leq n$ (on a même $R(v) \leq \log n$)
- A n'importe quel instant, les rangs sont décroissants dans l'arbre

- Si un noeud est de rang r, il a eu un jour au moins 2^r descendants.
- Le nombre de sommets de rang r est au maximum $n/2^r$.

Si on regarde maintenant la nouvelle heuristique, on a de plus :

- Quand un noeud change de parent, c'est vers un parent de rang strictement supérieur.

Supposons faire k opérations. On va compter le nombre d'opérations réalisées. On suppose qu'on a transformé préalablement les union en find + link, donc on ne compte dans la suite que les find.

Chaque find part d'un noeud u et compresse le chemin $u_1 \dots u_p$ en reliant tout à la racine v.

Le nombre total d'opérations est égal au nombre total de changements de parents pour les sommets u. On sépare ces changements en deux bouts.

- Les changements concernant des sommets u qui ont même $\log^\star$ que leur père (à ce moment de l'algo)
- Les changements concernant des sommets u qui ont un $\log^\star$ différent de leur père (à ce moment de l'algo)

On va compter les deux changements de manière très différente.

Commençons par les changements du premier type, concernant des sommets qui ont même $\log^\star$ que leur père.

— Regardons un sommet dont le $\log^\star$ vaut k. Comme il y a au plus $n/2^r$ sommets de rang r, le nombre de sommets dont le $\log^\star$ du rang est k est au maximum :

$$\sum_{i=2^{\uparrow(k-1)}+1}^{2^{\uparrow k}} n/2^i \leq \frac{2n}{2^{\uparrow k}}$$

Chacun de ces sommets changera de père au maximum $2^{\uparrow k}$ fois avant que celui-ci n'ait un $\log^\star$ strictement plus grand. En tout, ça fait donc n changements au maximum pour les sommets dont le $\log^\star$ vaut k.

— Comme le $\log^\star$ peut valoir au maximum $\log^\star n$, ca nous fait donc au total au maximum $n \log^\star n$ opérations du premier type.

Comptons maintenant les changements du deuxième type, concernant des sommets qui ont un $\log^\star$ différent de celui de leur père.

— Chaque Find ne peut en faire que $\log^\star n$ au maximum.

— Donc en tout on en fait au maximum $k \log^\star n$

Soit au total $O((n+k)\log^\star n)$. ■

Exercices

(3 - 1) *(Résumé du cours)*

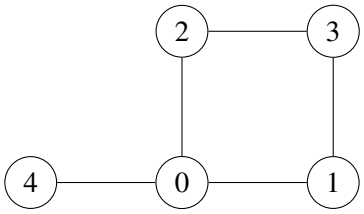
Q 1) Représenter la partition suivante avec la structure tableau/liste, et la structure arborescente. Pour chacune des deux structures, donner deux façons *différentes* de représenter la partition (on veut donc 4 représentations)

Q 2) Pour les quatre représentations de la question précédente, expliquez comment elles évoluent si on effectue l'opération $Union(2, 4)$, en suivant les heuristiques vues en cours.

(3 - 2) *(Graphes)*

Q 1) Rappeler les différents moyens de coder un graphe non orienté dans un langage de programmation

Q 2) Représenter le graphe suivant :



Q 3) Estimer l'usage en mémoire des différents codages pour un graphe avec n sommets et m arêtes.

Q 4) On rappelle qu'un graphe non orienté est connexe si pour tout sommet u et v, il y a un chemin de u à v. Ecrire un algorithme pour tester si un graphe est connexe en utilisant la structure de données Union-Find. Calculer sa complexité, suivant (a) l'implémentation de union-find (b) l'implémentation des graphes.

Q 5) Connaissez vous un autre algorithme pour tester si un graphe est connexe ? Si oui, quel est sa complexité ? Si non, allez rechercher vos cours de L2 ou d'IUT, puis recommencez la question.

(3 - 3) *(Inférence de types)*

On considère un langage de programmation très rudimentaire où il y a deux types de variables : entiers (int) et chaînes de caractères (str), et où les seuls opérations possibles sont :

— `a = b`

— `a = b + c` (on peut additioner deux entiers, ou additioner deux chaînes, mais pas un entier et une chaîne)

Voici un fragment P_1 de programme

```
a = 3
b = a + 5
c = d
f = e
f = "toto"
g = h
```

L'inférence de types consiste à trouver automatiquement le type de toutes les variables d'un programme.

Dans le cas où il n'est pas possible de donner précisément le type des variables, on regroupe ensemble les variables qui ont le même type, auquel on donne un nom spécial (dans la suite de la forme `'a`, `'b`, etc).

Par exemple, la sortie de l'algorithme d'inférence sur le programme P_1 ci dessus est :

```
a,b: int
e,f: str
c,d: 'a
g,h: 'b
```

ce qui signifie :

— Les variables `a` et `b` sont de type entier (int)
— Les variables `e` et `f` sont de type chaîne (str)
— Les deux variables `c` et `d` sont d'un type inconnu, mais qui est le même pour les deux variables
— Les deux variables `g` et `h` sont d'un type inconnu, mais qui est le même pour les deux variables

Q 1) Inférer les types sur le fragment de programme suivant. Il n'est pas demandé de justifier.

```
a = 3
b = c
d = a + c
g = f + f
e = "toto" + f
```

Q 2) Donner l'exemple d'un programme pour lequel l'inférence de types n'est pas possible.

Q 3) Concevoir un algorithme pour inférer les types. L'algorithme devra donner le type de chaque variable (comme dans l'exemple précédent) ou dire qu'il y a un problème de typage. On détaillera précisément ce que doit faire l'algorithme sur chaque ligne d'un programme.

Q 4) Expliquer ce qu'il faut ajouter à l'algorithme pour qu'il puisse traiter des programmes qui contiennent des fonctions. On expliquera en particulier comment effectuer l'inférence de types sur le programme suivant :

```
def fn(b, x):
  a = x + 1
  c = "toto" + b
  return a

def gn(b,x):
  return fn(x, b+1)
```

Q 5) Reformuler le problème comme un problème de graphes, et constater qu'il est très proche de l'exercice précédent.

Chapitre 4

ARBRE COUVRANT DE POIDS MINIMUM

Le but de ce chapitre est de répondre au problème suivant : On se donne un graphe non orienté, sur lequel chaque arête e est munie d'un poids $p(e) > 0$, qui peut représenter par exemple le coût de construction de l'arête e. On cherche à choisir certaines des arêtes, de sorte que tous les sommets soient connectés, et en minimisant la somme des poids des arêtes choisies.

Théorème 4.1

Une solution au problème est toujours un arbre. On l'appelle un *arbre couvrant de poids minimal* (ACPM)

Preuve : Rappelons qu'un arbre est un graphe connexe sans cycles. Remarquons d'abord qu'une solution du problème est nécessairement connexe par hypothèse.

De plus, elle est sans cycle : Si on prend un cycle et qu'on enlève n'importe quelle arête du cycle, on obtient une structure qui est toujours connexe, couvrante, et qui est de poids strictement plus petit (puisqu'on a enlevé une arête de poids strictement positif).

Dans une solution minimale, il n'y a donc pas de cycles. ■

Définition 4.1

Dans toute la suite, on note n le nombre de sommets et m le nombre d'arêtes du graphe.

4.1 Un meta-algorithme

Les deux algorithmes donnés plus loin pour le problème sont basés sur la même idée : On ajoute petit à petit des arêtes et on s'arrête quand on a obtenu un arbre couvrant de poids minimal.

Pour justifier les algorithmes, nous avons besoin d'un ou deux théorèmes.

Théorème 4.2

Si tous les poids sont distincts, il y a un seul ACPM.

Preuve : Soit T et T' deux ACPM. Soit e l'arête de poids minimale qui est présente dans l'un des arbres et pas l'autre. Disons qu'elle est présente dans T et pas dans T'.

Si on ajoute e à T', on crée un cycle. Soit e' une des arêtes du cycle qui est dans T' mais pas dans T (elle ne peuvent pas être toutes dans T puisque T n'a pas de cycle). Alors $T' \cup \{e\} \setminus \{e'\}$ est un arbre couvrant dont le poids est strictement plus petit que celui de T' contradiction. ■

Théorème 4.3

Soit U et V une partition des sommets en deux parties,
Soit e est l'(une) arête de poids minimal qui relie un sommet de U à un sommet de V.
Alors tout ACPM doit utiliser l'arête e (ou une autre arête de U à V de même poids).

Preuve : Soit T un ACPM. Supposons que T n'utilise pas l'arête e. On ajoute e à T. On obtient alors autant d'arêtes que de sommets, on a donc créé un cycle dans le graphe. Ce cycle utilise nécessairement l'arête e, et doit utiliser une autre arête e' qui relie U à V.

Considérons alors $T' = T \cup \{e\} \setminus \{e'\}$.

On se convainc assez facilement que T' est toujours un arbre couvrant. De plus le poids total T' est inférieur au poids total de T puisque $p(e) \leq p(e')$. Par minimalité de T, on a donc $p(e) = p(e')$. ■

Définition 4.2

Un *embryon* d'ACPM est un ensemble d'arêtes S qu'on peut compléter en un ACPM. Par exemple $S = \emptyset$ est un embryon.

Toute la suite est basée sur le théorème suivant :

Théorème 4.4

Soit S un embryon d'ACPM.
Soit U et V une partition des sommets en deux parties, de sorte qu'aucune arête de S n'ait une extrêmité dans U et l'autre dans V. Soit e l'(une) arête de poids minimal qui relie un sommet de U à un sommet de V.
Alors $S \cup \{e\}$ est un embryon d'ACPM.

Preuve : Même principe que la démonstration précédente. Soit T un ACPM qui complète S. On ajoute e à T, on en déduit une autre arête entre U et V qu'on peut supprimer de T et qui est nécessairement de poids plus grand que e. ■

En particulier, on ne crée pas de cycle en suivant cette méthode.

Algorithme 4.3

Le meta-algorithme suivant permet de trouver un ACPM.
On part de $S = \emptyset$. Tant que S n'a pas $n - 1$ arêtes :

- Trouver une coupe U, V du graphe qui ne contient aucune arête de S.
- Trouver l'arête de poids minimal e entre U et V
- Ajouter e à S

Il ne s'agit pas d'un algorithme à proprement parler, puisqu'il est nécessaire d'expliquer comment trouver (choisir) la coupe pour obtenir un algorithme. Il existe plusieurs possibilités, qui mènent à plusieurs algorithmes.

4.2 Algorithme de Prim

Algorithme 4.4

L'algorithme de Prim fonctionne de la manière suivante :
— On part d'un sommet u quelconque et on pose $S = \emptyset$.
— Tant qu'il reste un sommet non visité, on cherche l'arête e de poids minimal entre un sommet visité et un sommet non visité. On ajoute e à S, et l'autre extrêmité de l'arête aux sommets visités.

Théorème 4.5

L'algorithme de Prim renvoie bien un ACPM.

Preuve : L'algorithme de Prim est bien une instance du meta-algorithme précédent : Il revient à considérer à chaque étape la coupe où on met d'un côté les sommets visités L et de l'autre les sommets non visités. ■

Il reste à réfléchir à une implémentation possible de cet algorithme. Pour obtenir une bonne complexité, on doit être capable très facilement de trouver l'arête de poids minimal entre les sommets visités et les sommets non visités. Pour cela, associons à chaque sommet u non visité le poids de l'arête minimal le reliant à un sommet visité.

A chaque étape, on doit donc :

— Trouver et supprimer le sommet minimal
— Mettre à jour les poids des sommets voisins du sommet u.

Une structure de données de type file de priorités (tas binaire, binomial ou autre) est parfaitement adaptée :

Algorithme 4.5

L'algorithme de Prim s'implémente à l'aide d'un tas H. La clé associée à un sommet v est le poids de l'arête minimale le reliant à un sommet visité.
A chaque étape de l'algorithme
— On utilise `ExtractMin(H)` pour trouver le sommet u à visiter.
— On ajoute l'arête correspondante à S
— On met à jour H :
  — Pour tout voisin v de u non visité :
    — Si le poids de l'arête e' entre u et v est inférieure à la clé associée à v, alors changer cette clé à l'aide de `DecreaseKey`

Théorème 4.6

L'algorithme précédent
— Appelle n fois `ExtractMin`
— Appelle au maximum m fois `DecreaseKey`

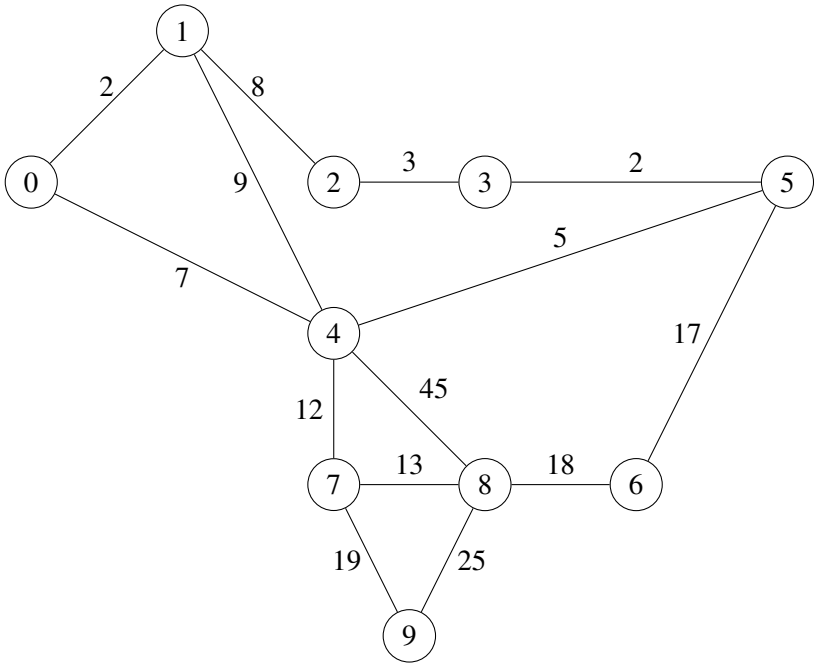
Toutes les autres opérations ont un temps cumulé de $O(n+m)$ si le graphe est implémenté sous forme de liste d'adjacence.

Avec une implémentation des tas sous forme de tas binomiaux, la complexité est donc de $O((n+m)\log n)$. On peut descendre à $O(n\log n+m)$ avec des tas de Fibonacci.

Une implémentation possible est décrite à la figure 4.1

♦ **Exemple**

Considérons le graphe suivant :



Au départ, on insère tous les noeuds (à l'exception du noeud initial 0) dans le tas avec priorité $+\infty$. Puis on change la priorité des voisins du noeud 0 pour qu'elle soit égale au poids de l'arête correspondante. On retient dans le tableau parent d'où provient l'arête qui a obtenu le minimum :

Noeud	1	2	3	4	5	6	7	8	9
Priorité	2	$+\infty$	$+\infty$	7	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$
Parent	0	-1	-1	0	-1	-1	-1	-1	-1

On retire le noeud de plus petite priorité, qui est le noeud 1. On ajoute donc l'arête $(0,1)$ à notre arbre couvrant.

On met à jour les voisins de 1. La priorité de 2 passe à 8. La priorité du noeud 4 ne change pas.

Noeud	2	3	4	5	6	7	8	9
Priorité	8	$+\infty$	4	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$
Parent	1	-1	0	-1	-1	-1	-1	-1

On retire le noeud de plus petite priorité, qui est le noeud 4. On ajoute donc l'arête $(1, 4)$ à notre arbre couvrant.

On met à jour les voisins de 4. Les trois noeuds 5, 7 et 8 changent de valeur.

Noeud	2	3	5	6	7	8	9
Priorité	8	$+\infty$	5	$+\infty$	12	45	$+\infty$
Parent	1	-1	4	-1	4	4	-1

On retire le noeud de plus petite priorité, soit le noeud 5. On ajoute l'arête $(4, 5)$ à notre arbre couvrant.

On met à jour les voisins de 5, les noeuds 3 et 6 changent de valeur.

Noeud	2	3	6	7	8	9
Priorité	8	2	17	12	45	$+\infty$
Parent	1	5	5	4	4	-1

On continue de même en supprimant le noeud de plus petite priorité, soit le noeud 3, ce qui ajoute l'arête (5,3) et change la priorité du noeud 2.

Noeud	2	6	7	8	9
Priorité	3	17	12	45	$+\infty$
Parent	3	5	4	4	-1

Puis on supprime le noeud 2, ce qui ajoute l'arête (3,2)

Noeud	6	7	8	9
Priorité	17	12	45	$+\infty$
Parent	5	4	4	-1

Puis on supprime le noeud 7, ce qui ajoute l'arête (4,7) et change la priorité des noeuds 8 et 9.

Noeud	6	8	9
Priorité	17	13	19
Parent	5	7	7

Puis on supprime le noeud 8, ce qui ajoute l'arête (8,7). La priorité des noeuds 6 et 9 ne changent pas.

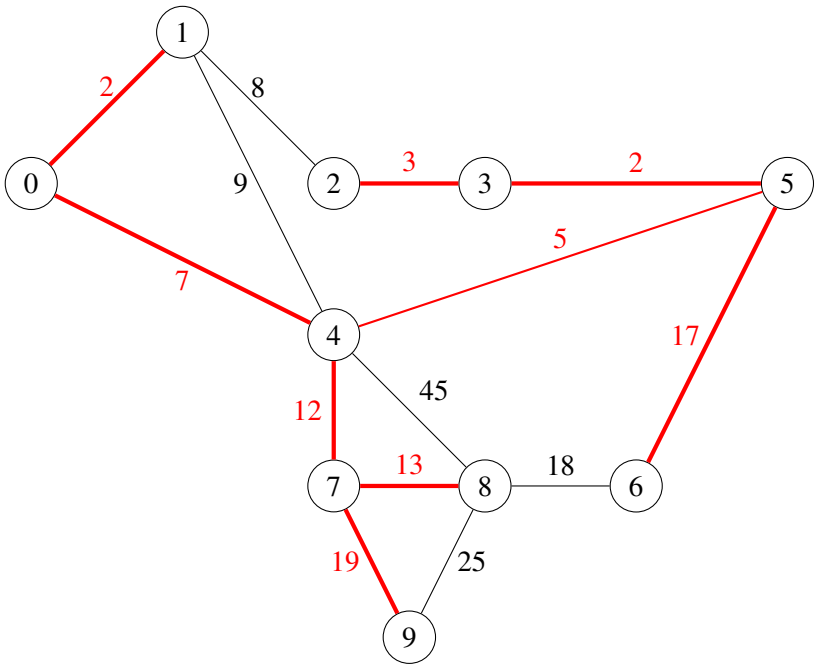
Noeud	6	9
Priorité	17	19
Parent	5	7

On supprime alors 6, ce qui ajoute l'arête $(5, 6)$.

Noeud	9
Priorité	19
Parent	7

Et enfin on ajoute l'arête $(7, 9)$.

Le résultat est donc le suivant :



4.3 Algorithme de Kruskal

Algorithme 4.6

L'algorithme de Kruskal ajoute les arêtes du graphe une par une à l'ensemble S, par ordre croissant, sans créer de cycles.

Théorème 4.7

L'algorithme de Kruskal crée bien un ACPM.

Preuve : On va montrer qu'à chaque étape de l'algorithme, on a bien un embryon S de ACPM. Plaçons nous à un moment où on ajoute l'arête e, de u à v. Soit U les sommets connectés à u et V les autres. Notons que v ne fait pas partie de U, sinon l'arête e créerait un cycle.

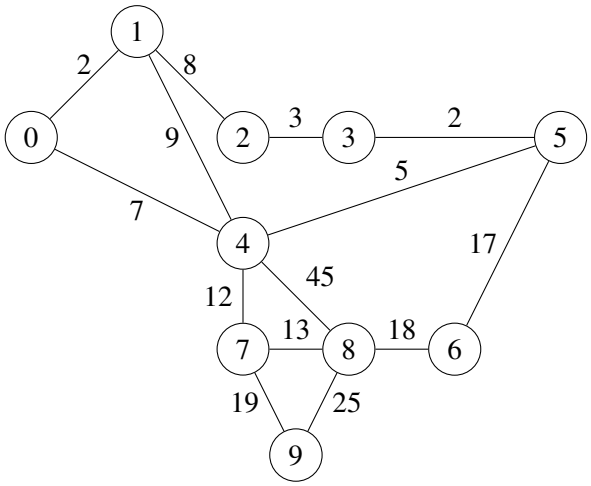
Toutes les arêtes examinées avant e sont soit dans S, soit créent un cycle. Donc en particulier aucune des arêtes examinées avant e n'est entre U et V. Donc e est clairement la plus petite des arêtes de U à V, donc $S \cup \{e\}$ est un embryon d'ACPM.

Il reste à prouver qu'à la fin S est bien un arbre couvrant, ce qui est clair. ■

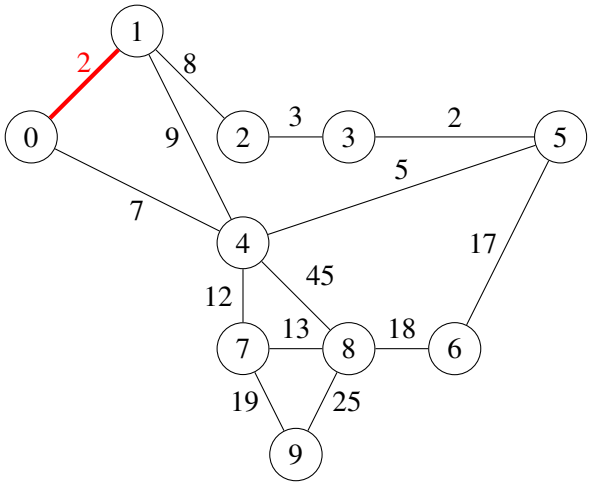
```
import binomial as heap
""" Algorithme de Prim
Le graphe a n sommets, numerotés de 0 à n - 1
Il est représenté par une liste d'adjacence:
adj[v] est une liste de couples (voisin, poids)
Le tas binomial contiendra des noeuds, initialement de priorité +∞
parent est le noeud pour laquelle la priorite est obtenue"""
class Node(heap.Element):
    def __init__(self,name):
        heap.Element.__init__(self,sys.maxint)
        self.name = name
        self.parent = -1

visited = [ False for i in range(n)]
# Initialisation du tas
H = []
nodes = [Node(i) for i in range(n)]
for node in nodes:
    H = heap.insert(H, node)


# On visite le sommet 0 et on traite ses voisins.
visited[0] = True

for (v,weight) in adj[0]:
    heap.decreaseKey(H,nodes[v], weight)
    nodes[v].parent = 0

S = []
#Algorithme principal
for i in range(n-1):
    (H,elem) = heap.extractmin(H)
    v = elem.name
    u = elem.parent
    S.append((u, v))
    visited[v] = True
    for (w,weight) in adj[v]:
        if not visited[w]:
            if nodes[w].key > weight:
                heap.decreaseKey(H,nodes[w], weight)
                nodes[w].parent = v

print S
```

Implém 4.1 – Algorithme de Prim

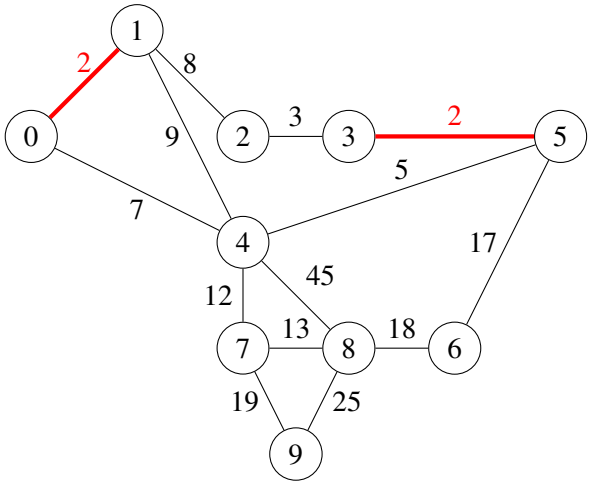
♦ **Exemple**



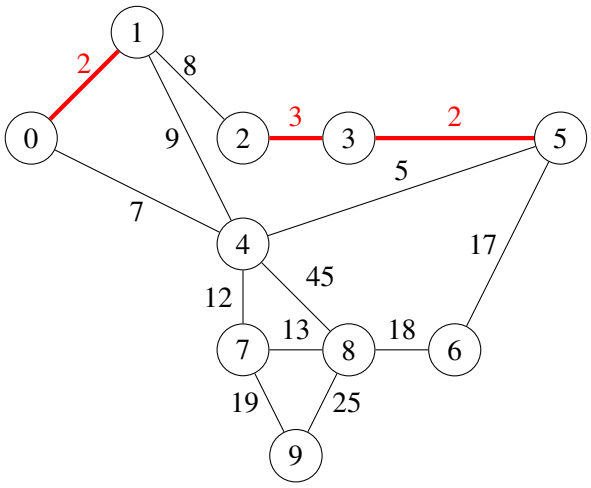
On commence par l'arête la plus petite, celle de 0 à 1.



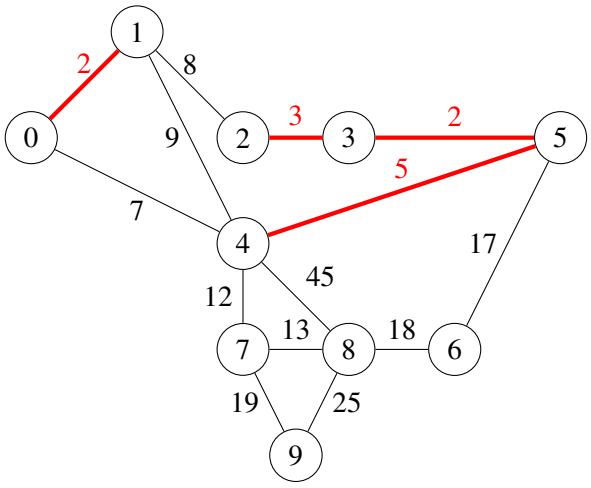
On ajoute ensuite l'arête la plus petite, soit celle de 3 à 5.



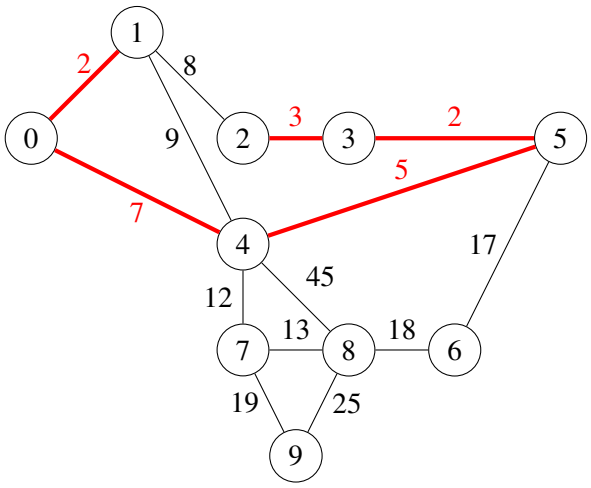
On ajoute ensuite l'arête la plus petite, soit celle de 2 à 3 :



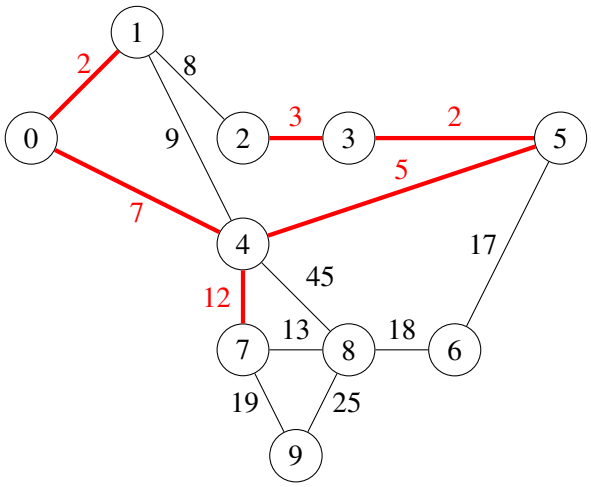
On ajoute ensuite l'arête la plus petite, soit celle de 4 à 5 :



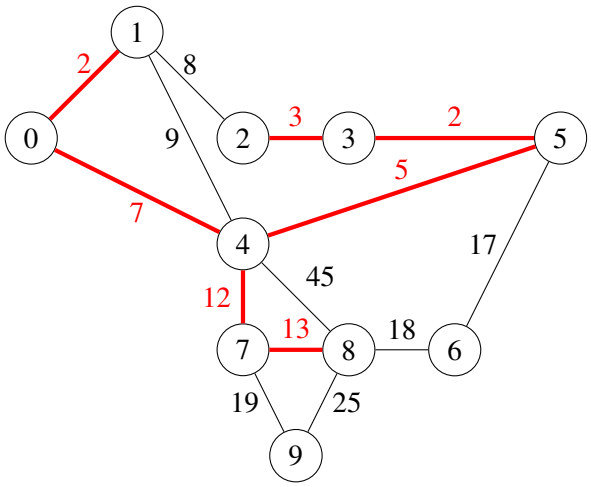
On ajoute ensuite l'arête la plus petite, soit celle de 0 à 4 :



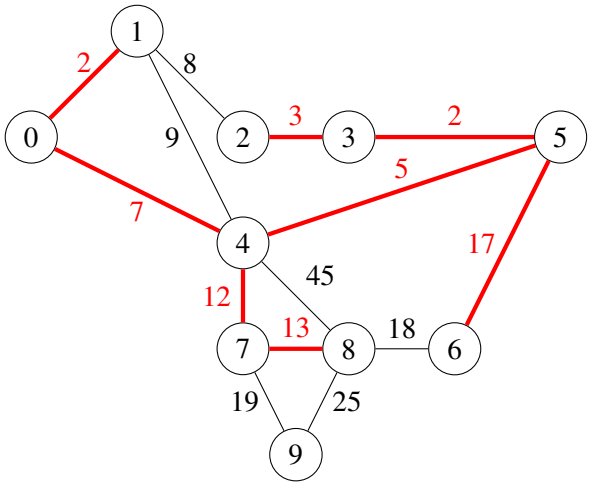
On ne peut pas ajouter l'arête de 1 à 2, car elle crée un cycle, de même que celle de 1 à 4. La prochaine arête est donc celle de 4 à 7 :



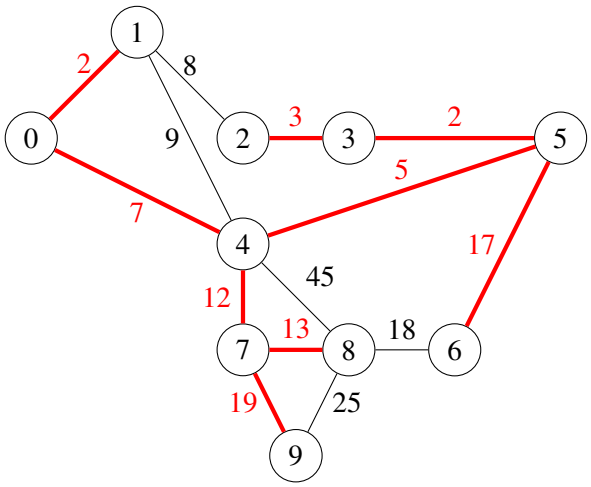
On ajoute ensuite l'arête de 7 à 8 :



Puis l'arête de 5 à 6 :



On ne peut pas ajouter l'arête 18, puisqu'on créerait un cycle, donc la précédente arête à ajouter est l'arête de 7 à 9 :



Il reste à réfléchir à l'implémentation. Nous avons principalement besoin d'une structure de données qui nous permet de savoir si ajouter une arête va créer un cycle ou non : La structure Union-Find est faite pour ça :

Algorithme 4.7

L'algorithme de Kruskal s'implémente en utilisant une structure de données Union-Find. L'algorithme s'écrit alors ainsi :

— Trier les arêtes par ordre croissant. Soit L la liste des arêtes.

— Pour tout `(u,v)` dans L :

— Si `Find(u)` $\neq$ `Find(v)` alors ajouter l'arête `(u,v)` à S et faire `Union(u,v)`.

La complexité de l'algorithme est $O(m \log m + (n+m) \log^\star n)$, le premier terme correspondant au tri, et le deuxième à l'utilisation de la structure de données Union-Find vu précédemment.

La complexité de cet algorithme est donc essentiellement concentrée dans le tri des arêtes.

On trouvera une implémentation à la figure 4.2.

A noter qu'un graphe connexe avec n sommets possède entre $n-1$ et $n(n-1)/2$ arêtes. Donc $\log m$ est du même ordre de grandeur que $\log n$.

Donc les deux algorithmes ont la même complexité, soit $O(m \log n)$. Cependant :

— Si $m > n \log n$, l'algorithme de Prim implémenté avec des tas de Fibonacci, a une meilleure complexité : $O(m)$.

— Si les arêtes sont déjà triées par odre croissant, l'algorithme de Kruskal a une meilleure complexité si $m < n \log n$.

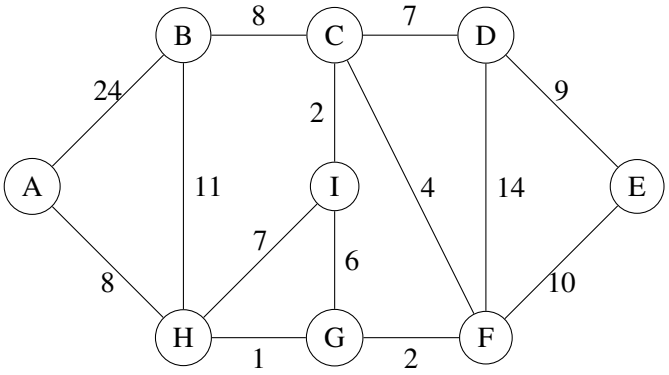
```
import unionfind as UF
## Algorithme de Kruskal
# Phase 1: Tri des arêtes
# On commence par générer la liste (poids(u,v), u, v)
edges = sum([[(weight,u,v)
              for (v,weight) in adj[u]]
              for u in range(n)], [])
edges.sort()
# Phase 2:
uf = UF.create(n)
S = []
for (k,u,v) in edges:
    if UF.find(uf, u) != UF.find(uf, v):
        UF.union(uf, u,v)
        S.append((u,v))
print S
```

Implém 4.2 – Algorithme de Kruskal

Exercices

(4 - 1) *(Algos vus en cours)*

Appliquer les algorithmes de Prim et de Kruskal au calcul d'un arbre couvrant de poids minimum (ACPM) du graphe suivant :



Rappel : Prim ajoute des sommets petit à petit à l'arbre en ajoutant une plus petite arête. Kruskal ajoute des arêtes de la plus petite à la plus grande sans jamais créer de cycles.

(4 - 2) *(Algorithme dual)*

Q 1) Soit G un graphe connexe, dont tous les poids des arêtes sont distincts, et $\mathcal{C}$ un cycle du graphe. Soit (u, v) l'arête de plus grand poids du cycle. Montrer qu'aucun arbre couvrant de poids minimum n'utilise l'arête (u, v).

Q 2) Que se passe-t-il si les poids des arêtes ne sont pas distincts ?

Q 3) Concevoir un nouvel algorithme, du même style que Kruskal, pour trouver l'arbre couvrant de poids minimum utilisant la propriété précédente. Appliquez-le à l'exemple du premier exercice.

Q 4) Comment implémenter votre algorithme ?

(4 - 3) *(Modifications locales)*

Q 1) Soit G un graphe connexe et T un arbre couvrant de G de poids minimal. On ajoute une nouvelle arête au graphe. Donner un algorithme pour trouver un ACPM du nouveau graphe G'. Calculer sa complexité. Appliquez à l'exemple lorsqu'on ajoute une arête de A à C de poids 5.

Q 2) Soit G un graphe connexe, et T un arbre couvrant de G de poids minimal. On ajoute un nouveau sommet au graphe. Donner un algorithme pour trouver un ACPM du nouveau graphe G'. Appliquez à l'exemple lorsqu'on ajoute le sommet J, relié à B, C, D avec poids $7, 6, 4$. Calculer sa complexité.

Q 3) Soit G un graphe connexe et T un arbre couvrant de G de poids minimal. On supprime une arête de G. Donner un algorithme pour trouver un ACPM du nouveau graphe G', s'il existe. Appliquer à l'exemple lorsqu'on supprime l'arête entre C et I. Calculer sa complexité.

Chapitre 5

CHEMIN DE COÛT MINIMAL

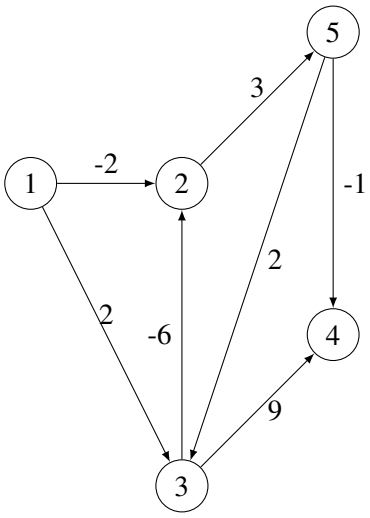
Dans ce chapitre, on cherche à résoudre le problème suivant : Etant donné un graphe où les arêtes e sont orientées et sont munies d'un coût $c(e)$, on cherche le chemin de coût minimum entre deux sommets u et v, ou plus généralement entre un sommet u et tous les autres sommets v.

Définition 5.1

Soit u un sommet privilégié.

On note $d[v]$ le chemin de coût minimal entre u et v. Par convention $d[v] = +\infty$ s'il n'y a pas de chemin entre u et v.

Le problème est rendu difficile par des arêtes dont le coût est négatif. Considérons ainsi le graphe suivant :



Le chemin de coût minimal de 1 à 4 n'existe pas : en effet si on emprunte une fois le cycle $2 \to 5 \to 3 \to 2$, on obtient un coût de -1. Donc si je pars de 1, que j'emprunte n fois le cycle, et que je finis en prenant les arêtes $2 \to 5 \to 4$, j'obtiens un coût total de $-2 - n + 3 - 1 = -n$, qui est donc aussi petit qu'on veut.

Cette situation arrive précisément à cause d'un cycle négatif :

Théorème 5.1

Si le graphe contient un cycle de coût total négatif, alors la fonction d n'est pas toujours définie.

Si le graphe ne contient pas de cycle de coût total négatif, alors d est toujours définie : dans ce cas le chemin de coût minimal de u à v ne contient jamais de cycle. Dans ce cas on a alors $d[u] = 0$.

Preuve : S'il n'y a pas de cycle de coût total négatif, alors tout chemin qui passe par un cycle a un coût supérieur au chemin plus court qui n'emprunte pas le cycle. On en déduit que le minimum est atteint pour un chemin qui ne contient pas de cycle. Il y a un nombre fini de tels chemins, donc le minimum ne peut pas voir $-\infty$. ■

Théorème 5.2

S'il n'y a pas de cycle négatif alors :

- $d[u] = 0$
- $d[w] = \min_{v|(v,w)\in E} d[v] + c(v, w)$ où v parcourt tous les voisins du sommet w.

Algorithme 5.2

Le méta-algorithme suivant résout le problème du chemin de coût minimum dans un graphe sans cycle négatif.

- On part avec $d[u] = 0$ et $d[v] = +\infty$ pour tous les autres sommets.
- Tant qu'il existe une arête (v, w) telle que $d[w] > d[v] + c(v, w)$, faire $d[w] = d[v] + c(v, w)$

Si on garde en mémoire , pour chaque sommet w, le sommet $v = p[w]$ pour lequel on a l'égalité $d[w] = d[v] + c(v, w)$, alors on peut retrouver le chemin le plus court de u à w en suivant le tableau p.

Preuve : On peut montrer facilement qu'au cours de l'algorithme $d[v]$ correspond toujours au coût d'un chemin de u à v. De plus comme $d[v]$ ne fait que diminuer à chaque étape de l'algorithme, on passe à chaque fois à un chemin de coût plus petit, donc l'algorithme s'arrête

On suppose maintenant que l'algorithme s'est arrêté. Soit $u = u_1, u_2 \dots u_n = v$ le chemin de coût minimal de u à v.

Vu le fonctionnement de l'algorithme, on a pour tout i $d[u_i] \leq d[u_{i-1}]+c(u_{i-1}, u_i)$. On en déduit que $d[v] \leq c(u_1, u_2) + c(u_2, u_3) + \dots c(u_{n-1}, u_n)$. Donc $d[v]$ est plus petit que la valeur du chemin de coût minimal. Comme il correspond au coût d'un chemin, il correspond bien au coût du chemin de coût minimal. ■

5.1 Coûts positifs - Algorithme de Dijkstra

On commence par le cas simple où les coûts sont positifs. Dans ce cas, il est certain qu'il n'existe pas de cycle négatif.

L'algorithme de Dijkstra calcule $d[v]$ pour tous les sommets v dans un ordre bien particulier.

Algorithme 5.3

L'algorithme de Dijkstra fonctionne de la manière suivante.

— On maintient une liste des sommets visités. Au départ, aucun sommet n'est visité. $d[u] = 0$, $d[v] = +\infty$ pour les autres sommets.

— Tant qu'il reste un sommet non visité, on visite le sommet v qui a la plus petite valeur de $d[v]$. On met ensuite à jour les valeurs $d[w]$ pour les voisins de v : Si $d[v] + c(v, w) < d[w]$ alors on change la valeur de $d[w]$

Théorème 5.3

L'algorithme de Dijkstra est correct.

Preuve : On note $g[v]$ la valeur du plus court chemin de u à v. On veut démontrer que $d[v] = g[v]$ pour tout v.

On va démontrer que, si $u = u_1, u_2, u_3, \dots u_k$ est un chemin de coût minimal, alors pour tout i, $d[u_i] = g[u_i]$, ce qui prouve le résultat.

Si ce n'est pas le cas, l'un des $d[u_i]$ n'a pas la bonne valeur. Quitte à prendre un chemin plus court, on peut supposer que c'est u_k.

Au moment où u_k est visité, le sommet u_{k-1} n'a pas encore été visité. Sinon, $d[u_k]$ aurait la bonne valeur.

Il existe donc un sommet u_i qui n'a pas été visité au moment où u_k est visité, on prend le i minimum. Au moment où u_k est visité, les sommets $u_1, u_2, \dots u_{i-1}$ ont été visités, mais pas u_i.

Par hypothèse, $d[u_j] = g[u_j]$ pour tous les $j < k$ à la fin de l'algorithme. Or $d[w]$ ne change plus une fois que le sommet w a été visité.

Donc à ce moment là :

— $d[u_i] = g[u_i]$. En effet u_{i-1} a été visité, donc $d[u_{i-1}] = g[u_{i-1}]$ et la visite de u_{i-1} va mettre la bonne valeur dans $d[u_i]$

— $d[u_k] > g[u_k]$ puisqu'on n'a pas la bonne valeur

— $d[u_k] \leq d[u_i]$ puisque c'est u_k qui sort et non pas u_i.

— $g[u_k] \geq g[u_i]$ car les arêtes sont positives.

On a donc $d[u_i] = g[u_i] \leq g[u_k] < d[u_k] \leq d[u_i]$ une contradiction. ■

Il reste à réfléchir à l'implémentation. Comme pour l'algorithme de Prim, l'utilisation d'une file

de priorité semble évidente

Algorithme 5.4

L'algorithme de Dijkstra s'implémente à l'aide d'un tas H. La clé $d[v]$ associée à un sommet v est le coût d'un chemin de u à v.

A chaque étape de l'algorithme

— On utilise `ExtractMin(H)` pour trouver le sommet u à visiter.

— On met à jour H :

— Pour tout voisin v de u non visité :

— Si $d[u]+c(u,v) > d[v]$, changer la valeur de $d[v]$ à l'aide de `DecreaseKey`

Théorème 5.4

L'algorithme précédent

— Appelle n fois `ExtractMin`

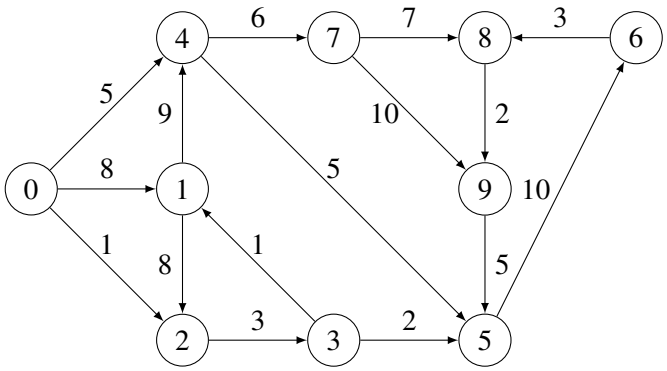
— Appelle au maximum m fois `DecreaseKey`

Toutes les autres opérations ont un temps cumulé de $O(n+m)$ si le graphe est implémenté sous forme de liste d'adjacence.

Avec une implémentation des tas sous forme de tas binomiaux, la complexité est donc de $O((n+m)\log n)$. On peut descendre à $O(n\log n+m)$ avec des tas de Fibonacci.

♦ **Exemple**

Considérons le graphe suivant :



Au départ, on insère tous les noeuds dans le tas avec priorité $+\infty$, sauf le noeud de départ (ici 0 avec priorité 0.

Noeud	0	1	2	3	4	5	6	7	8	9
Priorite	0	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$
Parent	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1

On retire le noeud de priorité minimum, soit 0. On met à jour ses trois voisins. (Les colonnes en gris correspondent aux noeuds visités, et ne sont donc plus dans la liste de priorité)

Noeud	0	1	2	3	4	5	6	7	8	9
Priorite	0	8	1	$+\infty$	5	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$
Parent	-1	0	0	-1	0	-1	-1	-1	-1	-1

On retire le noeud de priorité minimum, soit 2. On met à jour son voisin 3, qui passe donc a une priorité de $p[2] + 3 = 1 + 3 = 4$.

Noeud	0	1	2	3	4	5	6	7	8	9
Priorite	0	8	1	4	5	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$
Parent	-1	0	0	2	0	-1	-1	-1	-1	-1

On retire le noeud de priorité minimum, soit 3. On met à jour son voisin 5, qui passe donc a une priorité de $p[3] + 2 = 4 + 2 = 6$.

Noeud	0	1	2	3	4	5	6	7	8	9
Priorite	0	5	1	4	5	6	$+\infty$	$+\infty$	$+\infty$	$+\infty$
Parent	-1	3	0	2	0	3	-1	-1	-1	-1

On retire le noeud de priorité minimum, soit 1. Son voisin 4 ne change pas de valeur, puisqu'on obtiendrait une valeur de $p[1] + 9 = 14 > 5$

Noeud	0	1	2	3	4	5	6	7	8	9
Priorite	0	5	1	4	5	6	$+\infty$	$+\infty$	$+\infty$	$+\infty$
Parent	-1	3	0	2	0	3	-1	-1	-1	-1

On retire le noeud de priorité minimum, soit 4. Son voisin 5 ne change pas de valeur, au contraire de son voisin 7 :

Noeud	0	1	2	3	4	5	6	7	8	9
Priorite	0	5	1	4	5	6	$+\infty$	11	$+\infty$	$+\infty$
Parent	-1	3	0	2	0	3	-1	4	-1	-1

On continue ainsi le déroulement de l'algorithme.

Noeud	0	1	2	3	4	5	6	7	8	9
Priorite	0	5	1	4	5	6	16	11	$+\infty$	$+\infty$
Parent	-1	3	0	2	0	3	5	4	-1	-1

Noeud	0	1	2	3	4	5	6	7	8	9
Priorite	0	5	1	4	5	6	16	11	18	21
Parent	-1	3	0	2	0	3	5	4	7	7

Noeud	0	1	2	3	4	5	6	7	8	9
Priorite	0	5	1	4	5	6	16	11	18	21
Parent	-1	3	0	2	0	3	5	4	7	7

Noeud	0	1	2	3	4	5	6	7	8	9
Priorite	0	5	1	4	5	6	16	11	18	20
Parent	-1	3	0	2	0	3	5	4	7	8

Noeud	0	1	2	3	4	5	6	7	8	9
Priorite	0	5	1	4	5	6	16	11	18	20
Parent	-1	3	0	2	0	3	5	4	7	8

On déduit du résultat que le chemin de coût minimal entre 0 et 9 est de cout 20. Pour savoir par où il passe, on suit le tableau parent : Le père de 9 est 8, dont le père est 7, dont le père est 4, dont le père est 0. Le chemin de coût minimal est donc $0 \to 4 \to 7 \to 8 \to 9$

Une implémentation de l'algorithme est donné en figure 5.1.

5.2 Cas général - Algorithme de Bellman-Ford

L'algorithme de Dijkstra ne fonctionne pas s'il y a des arêtes négatives (pourquoi ?). L'algorithme de Bellman-Ford peut dans ce cas être utilisé. Il permet de plus de trouver un cycle de coût négatif s'il y en a un.

Algorithme 5.5

L'algorithme de Bellman-Ford calcule un chemin de coût minimal dans un graphe avec des poids négatifs de la façon suivante :

- On part avec $d[u] = 0$, $d[v] = +\infty$ pour les autres sommets.
- On exécute n fois l'algorithme suivant :
  - Pour toute arête (u, v), si $d[v] > d[u] + c(u, v)$, alors $d[u] = d[v] + c(u, v)$

Si l'algorithme s'est stabilisé, il contient les bonnes valeurs. Si l'algorithme ne s'est pas stabilisé (si le tableau d a changé à la dernière étape), alors le graphe contient un cycle négatif.

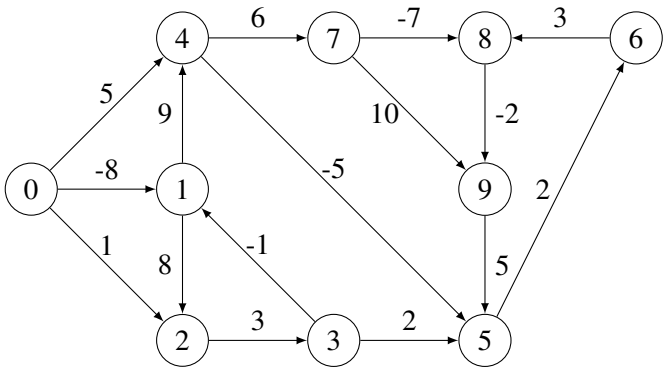
La complexité de l'algorithme de Bellman-Ford est $O(nm)$: n étapes, chacune examinant toutes les arêtes.

Preuve : On démontre assez facilement par récurrence sur k qu'au bout de k étapes de l'algorithme, $d[v]$ est plus petit que le chemin de coût minimal de longueur au plus k. S'il n'y a pas de cycles, le chemin de coût minimal est de longueur au plus $n - 1$, donc la dernière étape ne changera pas d.

Réciproquement, si l'algorithme s'est stabilisé, il est clair qu'il ne peut pas avoir de cycle négatif. ■

♦ **Exemple**

Considérons le graphe suivant :



```
import binomial as heap
"""Algorithme de Dijkstra
Le graphe a n sommets, numérotés de 0 à n - 1
Le graphe est donne par liste d'adjacence
adj[v] est une liste de couples (voisin, cout)
Le tas contiendra des noeuds, initialement de priorité +∞
parent est le noeud pour laquelle la priorité est obtenue"""
class Node(heap.Element):
    def __init__(self,name):
        heap.Element.__init__(self,sys.maxint)
        self.name = name
        self.parent = -1

visited = [ False for i in range(n)]

# Initialisation du tas
H = []
nodes = [Node(i) for i in range(n)]
nodes[0].key = 0
for node in nodes:
    H = heap.insert(H, node)

#Algorithme principal
for i in range(n):
    (H,elem) = heap.extractmin(H)
    v = elem.name
    u = elem.parent
    visited[v] = True
    for (w,cost) in adj[v]:
        if not visited[w]:
            if nodes[w].key > cost+nodes[v].key:
                heap.decreaseKey(H,nodes[w], cost+nodes[v].key)
                nodes[w].parent = v

for u in nodes:
    print u.name, u.key
```

Implém 5.1 – Algorithme de Dijkstra

Voici une exécution possible de l'algorithme de Bellman-Ford. Ici, on a fait une légère variante, plus facile à faire à la main (mais moins efficace en pratique) : On recalcule le nouveau tableau d à partir de l'ancien tableau d

Etape	0	1	2	3	4	5	6
0	0	0	0	0	0	0	0
1	$+\infty$	-8	-8	-8	-8	-8	-8
2	$+\infty$	1	0	0	0	0	0
3	$+\infty$	$+\infty$	4	3	3	3	3
4	$+\infty$	5	1	1	1	1	1
5	$+\infty$	$+\infty$	0	-4	-4	-4	-4
6	$+\infty$	$+\infty$	$+\infty$	2	-2	-2	-2
7	$+\infty$	$+\infty$	11	7	7	7	7
8	$+\infty$	$+\infty$	$+\infty$	4	0	0	0
9	$+\infty$	$+\infty$	$+\infty$	21	2	-2	-2

Commentons trois étapes. D'abord l'étape 1. Pour recalculer la nouvelle valeur de $d[0]$, on regarde les sommets qui pointent vers 0. Il n'y en a pas, donc la valeur de $d[0]$ ne change pas. Pour recalculer la nouvelle valeur de $d[1]$, on regarde les sommets qui pointent vers 1. Il y en a deux : le sommet 3 et le sommet 0. Le sommet 3 a une valeur de $+\infty$, ce qui fait une valeur de $+\infty + -1 = +\infty$. Le sommet 0 a une valeur de 0, ce qui fait donc une valeur de $0 - 8 = -8$. Le sommet 1 passe donc à -8. De même pour les autres sommets.

Maintenant l'étape 2. On peut vérifier que $d[0]$ et $d[1]$ ne changent pas. Pour recalculer la nouvelle valeur de $d[2]$, on regarde les sommets qui pointent vers 2. Il y en a 2 : le sommet 0 et le sommet 1. Si on utilise le sommet 0, on obtient une valeur de $d[0] + 1 = 1$. Si on utilise le sommet 1, on obtient une valeur de $d[1] + 8 = 0$. Donc $d[2]$ passe à 0.

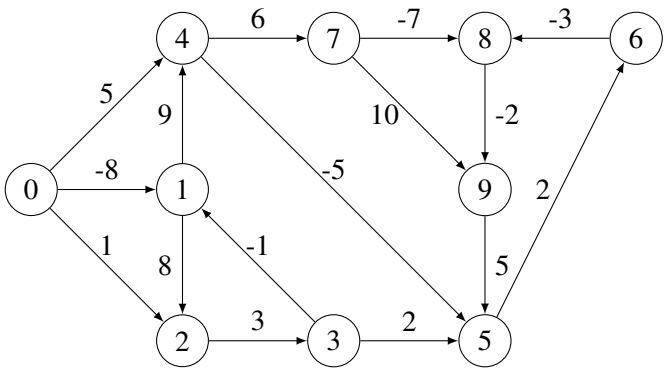
On va maintenant aller un peu plus vite et regarder comment trouver ce qui change à l'étape 5. Les seuls sommets qui peuvent changer de valeur à l'étape 5 sont des voisins des sommets qui ont changé de valeur à l'étape 4. A l'étape 4, seul les sommet $6, 8, 9$ ont changé de valeur. Les seuls sommets qui peuvent changer de valeur à l'étape 5 sont donc leurs voisins, soit les sommets 8 et 9.

Le sommet 8 peut changer de valeur du fait que le sommet 6 a changé de valeur : La nouvelle valeur possible pour $d[8]$ est donc $d[6] + 3 = -2 + 3 = 1$. Elle est supérieure à la valeur actuelle, donc $d[8]$ ne change pas de valeur

Le sommet 9 peut changer de valeur du fait que le sommet 8 a changé de valeur : La nouvelle valeur possible pour $d[9]$ est donc $d[8] - 2 = 0 - 2 = -2$. Elle est inférieure à la valeur actuelle, donc $d[9]$ change de valeur.

Le calcul a stabilisé au bout de 8 étapes, il n'y a donc pas de cycles négatifs.

♦ **Exemple**



Une arête a changé de coût par rapport à l'exécution précédente.

Dans ce cas le déroulement est le suivant :

Etape	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0	0
1	$+\infty$	-8	-8	-8	-8	-8	-8	-8	-8	-8	-8
2	$+\infty$	1	0	0	0	0	0	0	0	0	0
3	$+\infty$	$+\infty$	4	3	3	3	3	3	3	3	3
4	$+\infty$	5	1	1	1	1	1	1	1	1	1
5	$+\infty$	$+\infty$	0	-4	-4	-4	-4	-6	-6	-6	-6
6	$+\infty$	$+\infty$	$+\infty$	2	-2	-2	-2	-2	-4	-4	-4
7	$+\infty$	$+\infty$	11	7	7	7	7	7	7	7	7
8	$+\infty$	$+\infty$	$+\infty$	4	-1	-5	-5	-5	-5	-7	-7
9	$+\infty$	$+\infty$	$+\infty$	21	2	-3	-7	-7	-7	-7	-9

A la fin, le tableau ne s'est pas stabilisé, il y a donc un cycle négatif. Pour trouver le cycle, prenons un sommet qui a changé de valeur à la dernière étape : le sommet 9. S'il a changé de valeur, c'est à cause du changement du sommet 8 à l'étape 9. Si 8 a changé de valeur, c'est à cause du sommet 6 qui a changé de valeur à l'étape 8. Si 6 a changé de valeur à l'étape 8, c'est à cause de 5 qui change de valeur à l'étape 7. Si 5 change de valeur à l'étape 7, c'est à cause de 9 qui change de valeur à l'étape 6. On a donc trouvé le cycle : $9 \rightarrow 5 \rightarrow 6 \rightarrow 8 \rightarrow 9$.

Pour trouver le cycle dans une implémentation, on utilisera comme précédemment le tableau parent : En remontant en arrière à partir d'un sommet qui a changé de valeur à la dernière étape, on retrouve le cycle.

Une implémentation possible de l'algorithme de Bellman-Ford est proposée à la figure 5.2

```
d = [sys.maxint for i in range(n)]
parent = [-1 for i in range(n)]
d[0] = 0

i = 0
modified = 0
#modified est le numero d'un sommet qui change de valeur
#il vaut -1 si aucun sommet ne change de valeur
while i < n and modified != -1:
    modified = -1
    for u in range(n):
        for (v,cost) in adj[u]:
            if d[u] + cost < d[v]:
                d[v] = d[u]+ cost
                parent[v] = u
                modified = u
    i = i + 1

if modified != -1:
    print "Cycle négatif"
    print u
    u = modified
    while parent[u] != modified:
        u = parent[u]
        print u
else:
    for u in range(n):
        print d[u]
```

Implém 5.2 – Algorithme de Bellman-Ford

Exercices

(5 - 1) **Q 1)** Vrai ou Faux : Si j'augmente toutes les arête d'un graphe de la même constante c, alors je ne change pas le plus court chemin entre u et v ?

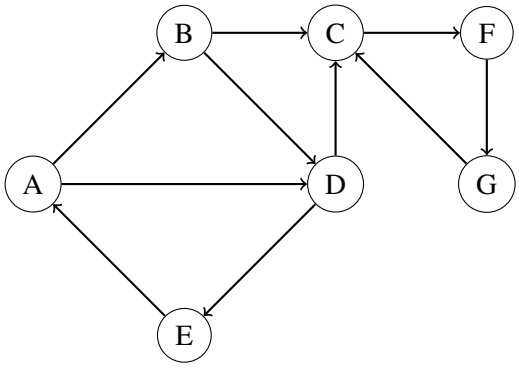
Q 2) Vrai ou Faux : Si je multiplie le poids de toutes les arête d'un graphe de la même constante $c > 0$, alors je ne change pas le plus court chemin entre u et v.

Q 3) Mêmes questions pour l'arbre couvrant de poids minimal.

Q 4) Donner un exemple avec une arête négative (mais pas de cycle négatif) où l'algorithme de Dijkstra se trompe.

(5 - 2) *(Cycles de taille fixée)*

On se donne un graphe orienté G et on cherche à trouver les sommets u qui appartiennent à des cycles de taille p. On s'intéresse à des cycles non élementaires, c'est à dire qu'on a le droit de passer plusieurs fois par le même sommet dans le cycle.



Dans l'exemple ci-dessus et si $p = 7$, on observe que les sommets A, B, C, D appartiennent à un cycle de taille p : le cycle A, B, D, E, A, D, E. Les sommets C, F ou G n'appartiennent pas à un cycle de taille 7.

Si u est un sommet, on note $N(u)$ ses voisins. Par exemple $N(D) = \{E, C\}$. Pour résoudre ce problème, on introduit un tableau $T[u, v, k]$, où u et v représentent des sommets et k un entier inférieur ou égal à p.

On veut remplir le tableau T de sorte que $T[u, v, k] = 1$ ssi il existe un chemin de u à v passant par exactement k sommets.

Q 1) Si on arrive à remplir ce tableau, comment peut-on trouver les sommets qui appartiennent à un cycle de taille p ?

Q 2) Expliquer comment remplir $T[u, v, 0]$ et $T[u, v, 1]$.

Q 3) Exprimer $T[u, v, k]$ en fonction de $T[\cdot, \cdot, k/2]$. Distinguer le cas pair et le cas impair.

Q 4) En déduire un algorithme pour remplir le tableau T puis trouver les sommets qui n'appartiennent pas à un cycle. Déterminer sa complexité en temps, mais aussi en mémoire, en fonction du nombre n de sommets, du nombre m d'arêtes et du nombre p.

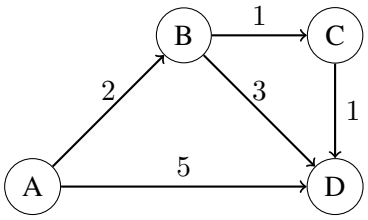
Q 5) On possède une machine possédant un processeur de dernière génération, qui peut effectuer 10^{11} instructions par seconde, et qui dispose de 8 Gigabytes de mémoire. On cherche à trouver les sommets appartenant à des cycles de taille 10 sur un graphe avec 10^5 sommets et 10^6 arêtes. Expliquer pourquoi cet algorithme n'est pas utilisable sur cet ordinateur. (Note : il y a à peu près 10^5 secondes dans une journée)

On fixe maintenant un sommet u et on considère le tableau $S[v, k]$ qui vaut 1 si et seulement si il y a un chemin de u à v de longueur k.

Q 6) Expliquer brièvement comment remplir le tableau S. En déduire un nouveau algorithme de complexité en temps $O(pn^3)$ et de complexité en mémoire $O(n)$ pour déterminer les sommets qui sont dans des cycles de taille p. Est-ce que cet algorithme est utilisable sur la machine ?

(5 - 3) *(Dijkstra avec dates)*

On considère un réseau de voies ferrées comme représenté ci-dessous :



Les poids indiqués sur les arêtes représentent le temps qu'il faut pour le train pour relier les deux villes. Il faut ainsi 5 heures pour aller de A à D en empruntant la ligne ferrée les reliant.

Le tableau ci-dessous recense, pour chacune des cing lignes, les horaires de trains :

$A \to B$	$A \to D$	$B \to D$	$B \to C$	$C \to D$
10 :00	15 :00	11 :00	12 :00	10 :00
11 :00	18 :00	14 :00	15 :00	16 :00
12 :00		17 :00	17 :00	18 :00

Un passager, partant de la ville A à $10 : 00$, veut aller à la ville D. On cherche à calculer le parcours en train le moins long pour y aller. Par exemple A pourrait décider de prendre le train de $10 : 00$ jusque B (en arrivant à 12 :00), puis le train de $14 : 00$ pour arriver en D à 17 :00.

On suppose que toutes les informations sont regroupées dans deux fonctions :

— $Duree(u, v)$: Donne la durée du trajet entre la ville u et la ville v en empruntant la voie ferrée de u à v. (Donc $Duree(A, B) = 2$, $Duree(A, D) = 5$)

— $Prochain(u, v, h)$ qui renvoie l'heure de départ du premier train partant après l'heure h de la ville u en destination de v. Par exemple, $Prochain(A, D, 16) = 18$

On suppose que si on arrive à 11h dans une ville, et que le prochain train part à 11h, alors il est possible de prendre ce train.

Q 1) Expliquer comment adapter l'algorithme de Dijkstra pour trouver le parcours en trian le moins long de A à D. On demande en particulier :

— De réecrire complètement l'algorithme de Dijkstra ;

— D'expliquer son fonctionnement ;

— De l'exécuter sur l'exemple.

(5 - 4) *(Chemins les plus courts entre tous les sommets)*

Q 1) Ecrire un algorithme qui détermine, simultanément pour tous les couples u et v, le poids du plus court chemin entre u et v. On remplira pour cela un tableau $T[u, v]$ qui contient le poids du plus court chemin entre u et v.

On cherche à obtenir un meilleur algorithme. Pour cela on suppose que les sommets du graphe sont numérotés de 1 à n et on définit le tableau $T[i, j, k]$ qui vaut le poids du plus court chemin entre i et j qui ne passe que par des sommets intermédiaires dont le numéro est inférieur à k.

Q 2) Que vaut $T[i, j, 0]$?

Q 3) Trouver comment calculer $T[i, j, k + 1]$ en fonction de $T[\cdot, \cdot, k]$

Q 4) En déduire un algorithme pour trouver tous les plus courts chemins. Déterminer sa complexité.