Algorithmique

Pierron Théo

ENS Ker Lann

Table des matières

1 Tris **1**
1.1 Tri par insertion 1
1.1.1 Algorithme 1
1.1.2 Terminaison 1
1.1.3 Correction 2
1.1.4 Complexité 3
1.2 Tri fusion 3
1.2.1 Algorithme 3
1.2.2 Terminaison et correction 4
1.2.3 Complexité 4
1.2.4 Optimalité 4
1.3 Tri par tas 5
1.3.1 File de priorité 5
1.3.2 Implémentation avec des arbres 5
1.3.3 Implémentation des arbres en tableaux 6
1.3.4 Tri par tas 7

2 Algorithmes géométriques **9**
2.1 Appartenance d'un point à un polygone 9
2.1.1 Algorithme lent mais générique 9
2.1.2 Algorithme rapide pour les polygones convexes 9
2.2 Enveloppe convexe 11
2.2.1 Algorithme de Graham 11

3 Ensembles et tableaux associatifs **13**
3.1 Structures de données abstraites 13
3.2 Arbres binaires de recherche 13
3.2.1 Définition 13
3.2.2 Opérations 13
3.3 Arbres binaires de recherche équilibrés 14
3.4 Table de hachage 15

3.4.1 Adressage ouvert . . . 15
3.4.2 Table de hachage . . . 15
3.4.3 Exemple . . . 16

4 Graphes et parcours de graphes **17**
4.1 44 définitions de théorie des graphes . . . 17
4.1.1 Graphes . . . 17
4.1.2 Chemins, cycles . . . 17
4.1.3 Qualificatifs pour les graphes . . . 18
4.1.4 Structure de données . . . 18
4.2 Parcours en profondeur . . . 19
4.2.1 Généralités . . . 19
4.2.2 Application à la détection de cycles . . . 20
4.2.3 Tri topologique . . . 21
4.2.4 Composantes fortement connexes . . . 21
4.3 Parcours en largeur . . . 23
4.3.1 Algorithme . . . 23
4.3.2 Terminaison et complexité . . . 23
4.3.3 Correction . . . 24
4.3.4 Algorithme de Dijkstra . . . 24

5 Programmation dynamique **27**
5.1 Exemple : suite de Fibonacci . . . 27
5.1.1 Algorithme naïf . . . 27
5.1.2 Algorithme itératif . . . 27
5.1.3 Mémoïsation . . . 27
5.2 Sous-structures optimales . . . 28
5.2.1 Découpage des barres . . . 28
5.2.2 Sous-structures . . . 28
5.3 Plus court chemin . . . 29
5.3.1 Cas sans cycle . . . 29
5.3.2 Cycles positifs . . . 30
5.3.3 Algorithme de Bellman-Ford . . . 30
5.3.4 Algorithme de Floyd-Warshall . . . 33

6 Algorithmes gloutons **35**
6.1 Arbre couvrant de poids minimum . . . 35
6.1.1 Le problème . . . 35
6.1.2 Algorithme de Kruskal . . . 35
6.1.3 Algorithme de Prim . . . 41
6.2 Approximation d'une couverture d'ensemble . . . 41

7 Flots et programmation linéaire 43
7.1 Flots . . . 43
7.1.1 Définitions . . . 43
7.1.2 Échec d'une méthode gloutonne . . . 43
7.1.3 Réseaux résiduels, algorithme de Ford-Fulkerson . . . 44
7.1.4 Théorème du flot maximum . . . 45
7.2 Programmation linéaire . . . 46
7.2.1 Problème . . . 46
7.2.2 Réduction . . . 47
7.3 Introduction à l'algorithme du simplexe . . . 47
7.3.1 Principe général . . . 47

8 Des problèmes difficiles 49
8.1 Introduction : le problème de l'arrêt . . . 49
8.2 Classes de complexité P et NP . . . 50
8.2.1 Intuition . . . 50
8.2.2 Une machine pour modéliser un algorithme . . . 50
8.2.3 Difficulté . . . 51
8.3 Le pouvoir de la logique propositionnelle . . . 51
8.3.1 La logique propositionnelle . . . 51
8.3.2 La puissance du problème SAT . . . 51
8.3.3 3-SAT . . . 52
8.4 Montrer la NP-complétude par réduction . . . 52
8.4.1 Ensembles indépendants dans un graphe . . . 52
8.4.2 Le problème CLIQUE . . . 53

Chapitre 1

Tris

Le problème de tri est : étant donné un tableau T, il faut trouver une permutation des éléments de T qui soit triée.

1.1 Tri par insertion

1.1.1 Algorithme

Algorithme 1: insertion(T, i)

Entrées : Un entier i et un tableau T_0 tel que $T_0[1, \cdots, i-1]$ soit trié
Sorties : Un tableau T tel que $T[1, \cdots, i]$ soit une permutation triée de $T_0[1, \cdots, i]$

1 $el := T[i]$, $j := i - 1$ **tant que** $j > 0$ *et* $el < T[j]$ **faire**
2 $\quad T[j+1] := T[j]$
3 $\quad j := j + 1$
4 $T[j+1] := el$
5 **retourner** T

1.1.2 Terminaison

Théorème 1.1 *L'algorithme insertion termine.*

Démonstration. $j >$ décroit strictement. ■

Théorème 1.2 *L'algorithme tri_insertion termine.*

Démonstration. i est strictement croissant majoré par $|T| + 1$. ■

Algorithme 2: tri_insertion

Entrées : Un tableau T de longueur n
Sorties : Une permutation de T triée

1 **pour** $i = 2$ **à** n **faire**
2   insertion(T, i)
3 **retourner** T

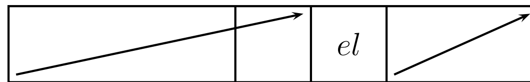
1.1.3 Correction

Théorème 1.3 *insertion fait bien ce qu'on lui demande de faire.*

Démonstration. On a l'invariant suivant : $T_0[1, i-1]$ a ses éléments dans l'ordre dans $T[1, j]$ puis dans $T[j+2, i]$ et si $j+2 \leqslant i$, $el < T[j+2]$.

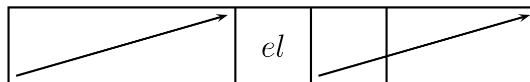
C'est bien vrai au début.

Si c'est vrai au début de la boucle, on a T de la forme :



j

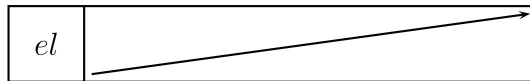
Après la boucle, on a T de la forme :



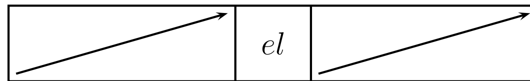
j

Et on a éxécuté la boucle car $el < T[j+2]$.

Quand la boucle est finie, T s'écrit si $j = 0$:



Et si $j > 0$,



∎

Théorème 1.4 *tri_insertion fait ce qu'on lui demande.*

Démonstration. On a clairement l'invariant : $T[1, i-1]$ est une permutation triée de $T_0[1, i-1]$.

On a donc le résultat. ■

1.1.4 Complexité

THÉORÈME 1.5 *Dans le pire des cas, insertion a une complexité en* $4i-1 = O(i)$ *et tri_insertion a une complexité en* $\sum_{i=2}^{n}(4i-1) = O(n^2)$.

1.2 Tri fusion

1.2.1 Algorithme

Algorithme 3: trifus

Entrées : Un tableau T de longueur n
Sorties : Un tableau trié, permutation de T

```
1  si n ⩾ 1 alors
2  |  retourner T
3  sinon
4  └  retourner fusion (trifus (T[1, n/2])) (trifus (T[n/2 + 1, n]))
```

Algorithme 4: fusion

Entrées : Deux tableaux triés T_1 et T_2
Sorties : Un tableau trié contenant les éléments des deux tableaux

```
1  si T1 = [] alors
2  |  retourner T2
3  sinon
4  |  si T2 = [] alors
5  |  |  retourner T1
6  |  sinon
7  |  |  si T1[1] ⩾ T2[1] alors
8  |  |  |  retourner T2[1] ::fusion(T1, T2[2, j])
9  |  |  sinon
10 └  └  └  retourner T1[1] ::fusion(T1[2, j], T2)
```

1.2.2 Terminaison et correction

<u>Théorème 1.6</u> *fusion termine et est correcte.*

Démonstration. Par récurrence sur $i + j$. H_0 est vraie.

Si H_n est vraie, si $a = 0$ ou $b = 0$, c'est vrai.

Sinon, on appelle fusion sur n qui termine via H_n.

De plus, si $A[1] < B[1]$, on a de plus $A[1] < A[2]$ donc c'est correct.

On fait de même si $A[1] > B[1]$. ■

<u>Théorème 1.7</u> *trifus termine et est correct.*

Démonstration. Par récurrence forte sur n.

H_0 est claire. Si H_n est vraie, les appels terminent et fusion aussi et ils sont de plus corrects et fusion aussi.

Donc H_{n+1} est vraie. ■

1.2.3 Complexité

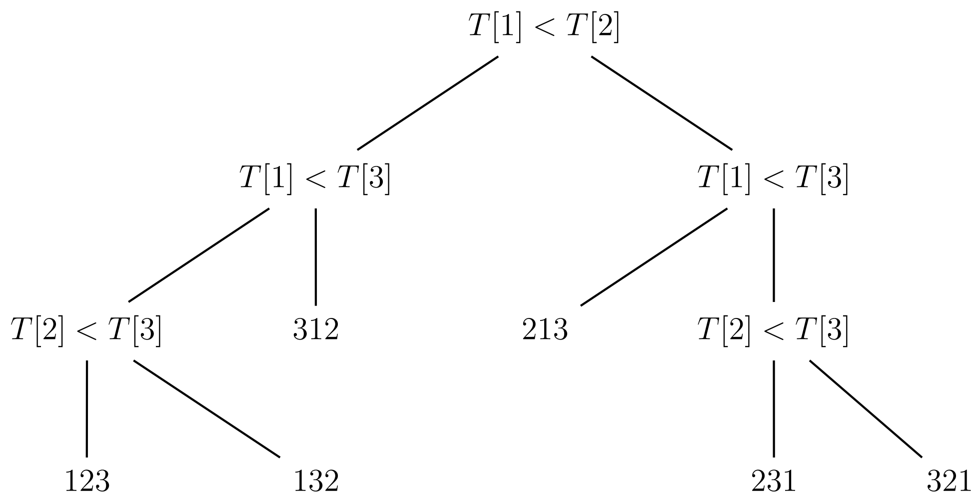
<u>Théorème 1.8</u> *fusion est en $O(a + b)$ et trifus est en $O(n \ln(n))$ dans le pire des cas.*

Démonstration. Si $n = 2^p$, $T_{2^p} = 2T_{2^{p-1}} + 2^p$ donc $T_{2^k} = (1 + k)2^k$. ■

1.2.4 Optimalité

<u>Théorème 1.9</u> *Un tri requiert au moins $\Theta(n \ln(n))$ opérations dans le pire des cas.*

Démonstration. On modélise l'algorithme par un arbre de décisions : par exemple, avec $n = 3$,



L'arbre a au moins $n!$ feuilles. Le nombre d'opérations est au plus la hauteur h de l'arbre.

On a donc $n! \leqslant f \leqslant 2^h$ donc $\log_2(n!) \leqslant h \leqslant n$.

Or $\ln(n!) \sim n\ln(n) - n + \ln(\sqrt{2\pi}) + \frac{\ln(n)}{2}$ donc $h = \Theta(n\ln(n))$. ■

1.3 Tri par tas

On veut un algorithme optimal et en place.

1.3.1 File de priorité

Définition 1.1 Une file est une structure de données avec les opérations `enfiler` (ajouter un élément) et `défiler` (enlever le plus vieil élément).

Une file de priorité est une structure de données munie des opérations `enfiler` (ajouter un élément avec une priorité) et `défiler` (l'élément de priorité maximale).

Le tri par tas consiste à construire une file de priorité et à défiler un à un les éléments.

Pour implémenter les files de priorité, on peut utiliser des tableaux : `enfiler` est en $O(1)$ et `défiler` en $O(n)$.

Si le tableau est trié, `enfiler` est $O(n)$ et `défiler` en $O(1)$.

1.3.2 Implémentation avec des arbres

Définition 1.2 Un arbre binaire est : l'arbre vide ($\varnothing$) ou $N(g, x, d)$ avec g et d deux arbres binaires.

Définition 1.3 La hauteur d'un arbre est définie inductivement par $h(\varnothing) = -1$ et $h(N(g, x, d)) = 1 + \max\{h(g), h(d)\}$.

On appelle racine d'un arbre son premier nœud et feuille un nœud tel que $g = d = \varnothing$.

Théorème 1.10 *On a $1 + h \leqslant n \leqslant 2^{h+1} - 1$, avec h la hauteur et n le nombre de nœuds.*

De même, $f \leqslant 2^h$ avec f le nombre de feuilles.

Définition 1.4 Un arbre est dit presque complet ssi tous les niveaux sont complets sauf éventuellement le dernier.

Proposition 1.1 Dans les arbres complets, $h \leqslant \log_2(n)$.

Définition 1.5 Un tas est un arbre binaire presque complet où chaque nœud a une valeur ajoutée plus grande que celle de ses fils.

1.3.3 Implémentation des arbres en tableaux

On représente un arbre par un tableau dans lequel tout élément en position i a son fils gauche en position $2i$ et son fils droit en position $2i + 1$.

Enfiler

On place le nouvel élément comme une feuille et on le fait remonter en permutant avec le nœud père si besoin est.

Défiler

On enlève la racine et on la remplace par une feuille. On fait descendre la feuille en permutant avec le maximum des nœuds fils g et d si besoin est.

Algorithme 5: défiler

Entrées : Un tas T
Sorties : Un tas issu de T, dans lequel on a enlevé le premier élément et cet élément

```
1 si T.taille= 0 alors
2 |  retourner Erreur
3 sinon
4 |  max := T[1]
5 |  T[1] := T[T.taille]
6 |  T.taille :=T.taille−1
7 |  tasser(T, 1)
8 |  retourner max,T
```

Algorithme 6: tasser

Entrées : Un tableau T et un indice i
Sorties : Un tas issu de T

```
1 i_max :=indicemax(T, i)
2 si i_max ≠ i alors
3 |  echange T[i] et T[i_max]
4 |  tasser(T, i_max)
```

La complexité de tasser est en $O(h) = O(\ln(n))$ et celle de défiler est donc en $O(\ln(n))$.

Algorithme 7: indicemax

Entrées : Un tableau T et un élément i
Sorties : L'indice de l'élément maximal entre $T[i]$, $T[2i]$ et $T[2i+1]$

```
1 si T[2i] > T[i] et 2i < T.taille alors
2 |   retourner 2i
3 sinon
4 |   si T[2i+1] > T[i] et 2i+1 < T.taille alors
5 |   |   retourner 2i+1
6 |   sinon
7 |   |   retourner i
```

1.3.4 Tri par tas

Algorithme 8: construire_tas

Entrées : Un tableau T
Sorties : Un tas issu de T

1 T.taille $:= |T|$
2 **pour** $i = \lfloor \frac{|T|}{2} \rfloor$ **à** 1 **faire**
3 | tasser(T, i)

Algorithme 9: tout_défiler

Entrées : Un tas T
Sorties : Un tableau trié issu de T

1 **pour** $i = |T|$ **à** 2 **faire**
2 | $T[i] :=$ défiler$(T[1, \cdots, i])$

Algorithme 10: tri

Entrées : Un tableau T
Sorties : Une permutation triée de T

1 construire(T)
2 tout_défiler (T)

La complexité de `construire` est en $O(n\ln(n))$ (en fait, c'est du $\ln(n)$: voir TD). Celle de `tout_défiler` est en $O(n\ln(n))$ donc on a un tri en $O(n\ln(n))$.

Chapitre 2

Algorithmes géométriques

2.1 Appartenance d'un point à un polygone

2.1.1 Algorithme lent mais générique

Définition 2.1 Un polygone P est décrit avec une séquence $(p_1, \cdots, p_n) \in (\mathbb{R}^2)^n$.

Les côtés en sont les segements $[p_i, p_{i+1}]$ et $[p_n, p_1]$.

On dit que P est simple ssi $S_i \cap S_j \neq \varnothing \Rightarrow |j-i| = 1$ et $\mathrm{Card}(S_i \cap S_j) = 1$.

Remarque 2.1 On ne s'intéresse qu'aux polygones simples.

Théorème 2.1 *Un polygone définit trois parties du plan : l'intérieur, l'extérieur et le contour.*

Théorème 2.2 *Soit P un polygone et $x \in \mathbb{R}^2$.*

Si x n'appartient pas au contour de P, x est à l'intérieur de P ssi le nombre de points d'intersection de P avec D où D est la demi-droite ouverte horizontale qui part de x vers la droite est impair.

Remarque 2.2 On a un problème avec les segments horizontaux et avec les points d'intersections de deux segments. On va donc ignorer les segments horizontaux et au lieu de considérer les $[p_i, p_{i+1}]$, on va considérer $[A, B[$ avec A les point le plus bas.

Théorème 2.3 *On a une complexité en $O(n)$.*

2.1.2 Algorithme rapide pour les polygones convexes

Définition 2.2 On dit qu'un polygone P est convexe ssi pour tout $a, b \in \mathring{P}$, $[a, b] \subset \mathring{P}$.

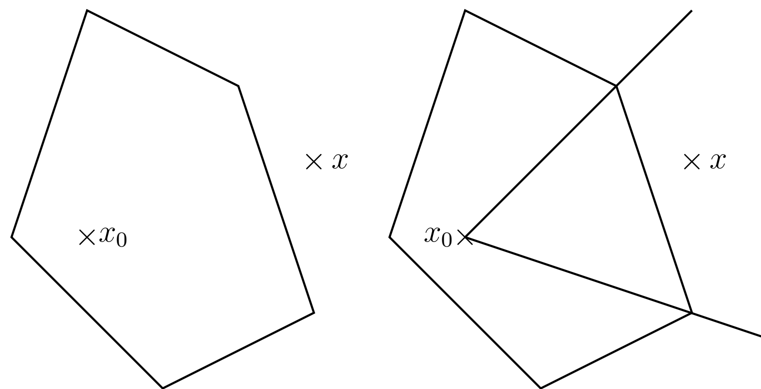
Algorithme 11: appartient

Entrées : Un polygone P et $x \in \mathbb{R}^2$
Sorties : La valeur booléenne de $x \in P$

```
1  si x ∈ P.contour alors
2  |  retourner Faux
3  sinon
4  |  D := demi-droite...
5  |  i := 0
6  |  pour [A, B[ ∈ P non horizontal faire
7  |  |  si D ∩ [A, B[ ≠ ∅ alors
8  |  |  |  i := i + 1
9  |  si i ≡ 0 mod 2 alors
10 |  |  retourner Faux
11 |  sinon
12 |  |  retourner Vrai
```

L'algorithme procède en 3 étapes :

- Choisir un point x_0 dans $\mathring{P}$.
- Trouver le secteur angulaire correspondant à x et x_0
- Tester si x et x_0 sont du même coté ou non.



On trouve un point à l'intérieur de P en temps constant (prendre un barycentre), on trouve le secteur angulaire en $O(\ln(n))$ car on peut faire une dichotomie. Dire si $x \in P$ prend alors un temps constant.

On a donc une complexité en $O(\ln(n))$.

2.2 Enveloppe convexe

2.2.1 Algorithme de Graham

Algorithme

Soit P un polygone. On pose q_0 le point le plus bas de P (et le plus à gauche, s'il y en a plusieurs) et $[q_1, \cdots, q_n]$ la liste triée de $P \setminus \{q_0\}$ selon les angles $(\widehat{\overrightarrow{p_0x}, \overrightarrow{p_0p_i}})$. On contruit ensuite l'enveloppe convexe de Q via :

Algorithme 12: Enveloppe convexe

Entrées : $[q_0, \cdots, q_n]$ triée
Sorties : L'enveloppe convexe de Q

```
1 S :=pile_vide
2 empile(S, p0),empile(S, p1),empile(S, p2)
3 pour i = 3 à n faire
4 |  tant que (S_sommet S_sous-sommet, S_sommet p_i) < π faire
5 |  |  dépiler(S)
6 |  empiler(S, p_i)
7 retourner S
```

Ligne 4 : **tant que** $(\widehat{\overrightarrow{S_{sommet}S_{sous-sommet}}, \overrightarrow{S_{sommet}p_i}}) < \pi$ **faire**

On a l'invariant de boucle suivant : « S contient l'enveloppe convexe de $p_0, \cdots, p_{i-1}$ ».

On a une complexité en $O(n)$ car tout point n'est défilé qu'une fois. La recherche de l'enveloppe convexe est donc en $O(n\ln(n))$.

Optimalité

On va effectuer une réduction : on va créer un algorithme de tri à partir d'un algorithme de recherche d'enveloppe convexe pour montrer que la complexité optimale est en $O(n\ln(n))$.

Pour trier le tableau T, on cherche l'enveloppe convexe de l'ensemble $T' = \{(x, x^2), x \in T\}$, on cherche le point q_i d'ordonnée minimale dans T' et on renvoie $T'[i, \cdots, n] :: T[1, \cdots, i-1]$.

Grâce à cet algorithme, on voit que la recherche d'enveloppe convexe est au moins en $O(n\ln(n))$ et la borne est atteinte par l'algorithme précédent.

Chapitre 3

Ensembles et tableaux associatifs

3.1 Structures de données abstraites

Définition 3.1 Un ensemble est une structure de données qui possède les opérations : ajouter un élément, elever un élément et tester l'appartenance.

Remarque 3.1 On peut ajouter, union, intersection cardinal, minimum,. . .

Définition 3.2 Un tableau associatif est une structure de données qui possède les opérations : ajouter un couple (clé,valeur), supprimer l'association d'une clé et rechercher la valeur d'une clé.

On va s'intéresser aux implémentations de ces structures.

3.2 Arbres binaires de recherche

On considère que l'ensemble des éléments admet un ordre total.

3.2.1 Définition

Définition 3.3 Un arbre binaire de recherche est un arbre binaire tel que pour tout nœud $N(G, x, D)$ de cet arbre, pour tout $n \in G$, $n \leqslant x$ et pour tout $n \in D$, $n \geqslant x$.

3.2.2 Opérations

Pour chercher i dans $N(G, x, D)$ on cherche i dans G si $i < x$ et dans D si $i > x$.

Pour ajouter i à $N(G, x, D)$, on cherche s'il y est. Dans ce cas, on ne fait rien. Sinon, on l'ajoute à G si $i < x$ et à D sinon.

Pour supprimer i à $N(G, x, D)$, on le supprime dans G si $i < x$, dans D si $i > x$ et si $i = x$, on renvoie $N(G', y, D)$ avec $y = \max G$ et $G' = G \setminus \{y\}$.

Ces trois opérations sont en $O(h)$ avec h la hauteur de l'arbre.

3.3 Arbres binaires de recherche équilibrés

Définition 3.4 On définit inductivement les arbres binaires de recherche équilibrés par :

- $\varnothing$ est équilibré
- $N(G, x, D)$ est équilibré ssi G et D le sont et $|h(G) - h(D)| \leqslant 1$.

Proposition 3.1 Si A est un arbre binaire de recherche équilibré, $\log_2(n+1) \leqslant h + 1 \leqslant 1.44 \log_2(n)$.

Remarque 3.2 On ne va pas calculer la hauteur à chaque fois, on la stocke dans A.h et on la met à jour au fur et à mesure.

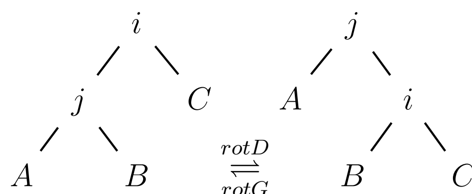
La fonction d'ajout ne change pas, mais on doit rééquilibrer après avoir ajouté un élément.

Algorithme 13: rééquilibrage

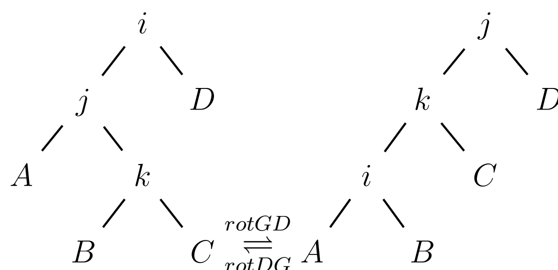
Entrées : Un arbre A
Sorties : Un arbre équilibré

```
1  si |G.h − D.h| ⩽ 1 alors
2      retourner A
3  sinon
4      si G.h = 2 + D.h alors
5          si G = ∅ alors
6              retourner Erreur
7          sinon
8              G := N(G', j, D')
9              si G'.h ⩾ D'.h alors
10                 retourner rotD(A)
11             sinon
12                 retourner rotGD(A)
13     sinon
14         on fait de même avec rotG et rotDG
```

avec :



et



Remarque 3.3 Il y a au plus un rééquilibrage effectué. En effet, à chaque rééquilibrage, h est constante.

L'algorithme de suppression est le même, sauf qu'on rééquilibre le résultat. Il peut y avoir plusieurs rééquilibrages.

3.4 Table de hachage

3.4.1 Adressage ouvert

Pour représenter $E \subset [\![0, M]\!]$, on peut utiliser un tableau de booléens à $M + 1$ cases.

L'ajout, la suppression et le test d'appartenance se font en temps constant, mais on a des problèmes pour M grand.

3.4.2 Table de hachage

Définition 3.5 Soit U un ensemble d'éléments et de clés.

Une table de hachage est un tableau T de M alvéoles. Pour placer les éléments, on utilise une fonction de hachage $h : U \to [\![1, M]\!]$ et on place l'élément e en $T[h(e)]$.

Remarque 3.4 Si h n'est pas injective, on parle de collision. Pour gérer ces colliusions, on utilise plutôt un tableau dont chaque élément est une liste.

Dans le pire des cas (h est constante) les complexités sont en $O(n)$ et dans le meilleur des cas (h injective), elles sont en $O(1)$.

Théorème 3.1 *La probabilité de trouver une fonction h injective est de $\frac{M!}{M^{|U|}(M-|U|)!}$.*

Démonstration. On a $M^{|U|}$ fonctions de hachage et $\frac{M!}{(M-|U|)!}$ sont injectives. D'où le résultat. ∎

<u>THÉORÈME 3.2</u> *Si $|U| \geqslant (N-1)M+1$, il existe une alvéole de N éléments.*

3.4.3 Exemple

On pose U l'ensemble des mots d'un roman de Victor Hugo et T un tableau à 30 cases.

- Posons $h : s \mapsto |s| \mod 31$. La répartition des mots n'est pas uniforme : la majorité des mots se situent au milieu du tableau.
- Posons $h : s \mapsto \sum_{i=0}^{3} s[i] \mod 31$. La répartition est plus uniforme mais il y a encore des écarts.
- Posons $h : s \mapsto \sum_{i=0}^{|s|} 19^i s[i] \mod 31$. La répartition est quasi-uniforme.

Chapitre 4

Graphes et parcours de graphes

4.1 44 définitions de théorie des graphes

4.1.1 Graphes

Définition 4.1 Un graphe orienté est un couple (S, A) où S est un ensemble de sommets et $A \subset S \times S$ est un ensemble d'arcs (orientés). Un arc orienté (s, t) se note $s \to t$.

On dit que $s \to t$ part de s et arrive à t.

Les sommets t tels que $s \to t$ sont appelés successeurs de s. Les sommets s tels que $s \to t$ sont appelés prédécesseurs de t.

Un arc $s \to t$ est dit incident pour s et t. Une boucle est un arc $s \to s$.

Le degré entrant d'un sommet est le nombre d'arcs qui y arrivent. Le degré sortant d'un sommet est le nombre d'arcs qui en partent.

Un graphe non orienté est un couple (S, A) où S est un ensemble de sommets et A un ensemble d'ensemble de deux sommets. Les éléments de A s'appellent des arcs non orientés (ou arêtes). Il n'y a pas de boucle.

Un arc non orienté $\{s, t\}$ se note $s - t$. Le degré d'un sommet s est le nombre d'arêtes incidentes à s.

4.1.2 Chemins, cycles

Définition 4.2 Un chemin de longueur n de s à t est une suite de sommets $s_0, \cdots, s_n$ avec $s_0 = s$, $s_n = t$ tels que pour tout i, $s_i \to s_{i+1}$.

Le chemin contient les sommets s_i et les arcs $s_i \to s_{i+1}$.

S'il existe un chemin de s à t, on dit que t est accessible depuis s. Il existe toujours un chemin de longueur 0 entre s et s. Si $0 \leqslant i \leqslant j \leqslant n$, $(s_i, \cdots, s_j)$ est un sous-chemin de $s_0, \cdots, s_n$.

Un chemin est dit élémentaire (ou simple) ssi tous les sommets du chemin sont distincts. Un circuit (ou cycle) est un chemin dont le premier et le dernier élément sont égaux qui contient au moins un arc. Pour les graphes non orientés, un chemin s'appelle aussi une chaîne.

Un chemin eulérien (resp. hamiltonien) est un chemin qui passe exactement une fois par chaque arc (resp. chaque sommet).

4.1.3 Qualificatifs pour les graphes

Définition 4.3 Un graphe simple est un graphe sans boucles. Un graphe sans cycles est acyclique.

Un graphe non orienté est connexe ssi tout sommet est accessible depuis tous les autres sommets. Une composante connexe d'un graphe est une classe d'équivalence pour la relation « est accessible depuis ». Un arbre est un graphe non orienté connexe et acyclique.

Un graphe orienté est fortement connexe ssi pour tout s, t, s est accessible depuis t et t est accessible depuis s. Une composante fortement connexe est une classe d'équivalnce pour cette relation.

Un graphe complet est un graphe tel que pour tout couple de sommets (s, t), on a $s \to t$.

Un graphe est dit planaire ssi il est dessinable dans le plan de manière à ce que deux arêtes ne se croisent pas.

Un graphe pondéré $G = (S, A)$ est un graphe muni d'une fonction de pondération $w : A \to \mathbb{R}$. $w(s \to t)$ s'appelle le poids de l'arc $s \to t$. Le poids d'un chemin est le somme des poids des arcs par lesquels il passe.

4.1.4 Structure de données

Un graphe peut être vu comme une structure de données avec les opérations :

- ajouter un sommet ou un arc
- supprimer un sommet ou un arc
- fusionner deux sommets
- déterminer les successeurs

Proposition 4.1 On peut représenter un graphe par une matrice d'adjacence $M \in \mathfrak{M}_n(\{0, 1\})$ avec $n = |S|$ et $M_{i,j} = 1$ ssi $i \to j$.

On peut aussi le représenter par un liste d'adjacence : on met les successeurs du sommet i dans L_i

4.2 Parcours en profondeur

4.2.1 Généralités

Algorithme

Algorithme 14: prévisite

Entrées : Un sommet s

1 $pre[s]$:=temps
2 temps :=temps+1

Algorithme 15: postvisite

Entrées : Un sommet s

1 $post[s]$:=temps
2 temps :=temps+1

Algorithme 16: explorer

Entrées : Un graphe G et un sommet s

1 $m[s]$:=vrai
2 prévisite(s)
3 **pour** *t successeur de s* **faire**
4   **si** $m[t]$ =*faux* **alors**
5   explorer(G, t)
6 postvisite(s)

Algorithme 17: Parcours en profondeur

Entrées : Un graphe G

1 **pour** $s \in S$ **faire**
2     $m[s]$:=faux
3 **pour** $s \in S$ **faire**
4   **si** $m.[s]$ =*faux* **alors**
5   explorer(s)

<u>Théorème 4.1</u> *On a un algorithme en $O(|A| + |S|)$.*

Démonstration. La boucle principale prend $O(|S|)$ opérations.
`explorer`(s) requiert Card $\underbrace{\{t, s \to t\}}_{E_t}$ opérations.

On a $O\left(\sum_{t \in S} \text{Card}(E_t)\right) = O(|A|)$.

D'où une complexité en $O(|S| + |A|)$. ∎

Proposition 4.2 Soient s et t deux sommets. Les intervalles $[pre[s], post[s]]$ et $[pre[t], post[t]]$ sont disjoints ou inclus l'un dans l'autre.

Classification des arcs

<u>Définition 4.4</u> Un arc $s \to t$ est dit :

- appelant ssi `explorer`(s) appelle directement `explorer`(t).
- avant ssi s est un ancêtre non père de t.
- arrière ssi s est un descendant de t.
- transverse sinon.

Lemme utile

Lemme 4.1.1
Soit $u \neq v \in S$.
`explorer`(u) appelle `explorer`(v) non directement ssi, au moment où u est découvert, il existe un chemin de u à v composé uniquement de sommets non vus.

Démonstration.

- $\Rightarrow$ `explorer`(u) appelle `explorer`(u_1) qui appelle ... qui appelle enfin `explorer`(v) donc pour tout i, $m[u_i]$ =faux et on a ledit chemin.
- $\Leftarrow$ On a un chemin $u \to u_1 \to \cdots \to v$. Soit i le plus petit indice tel que `explorer`(u_i) soit non appellé.
  Pendant l'appel `explorer`(u_{i-1}), $m[u_i]$ =faux et $u_{i-1} \to u_i$ existe donc `explorer`(u_{i-1}) n'est pas appellé, d'où la contradiction avec la minimalité de i. ∎

4.2.2 Application à la détection de cycles

Proposition 4.3 Un graphe orienté admet un cycle ssi un parcours en profondeur donne un arc arrière.

Démonstration.

$\Leftarrow$ Clair
$\Rightarrow$ S'il y a une boucle $u \to u$, on a un arc arrière.
Sinon, s'il y a un cycle $v_0 \to \cdots \to v_k \to v_0$, quand v_i est exploré (on note i l'indice tel que v_i est le premier exploré), il existe un chemin de v_i à v_{i-1} composé de sommets non vus.
Donc `explorer`(v_{i-1}) est appelé par `explorer`(v_i) donc $m[v_i]$ =vrai quand v_{i-1} est exploré donc $v_{i-1} \to v_i$ est un arc arrière. ∎

4.2.3 Tri topologique

Extensions linéaires

Définition 4.5 On dit que $\leqslant_l$ est une extension linéaire de l'ordre partiel $\leqslant$ ssi $\leqslant_l$ est un prolongement total de $\leqslant$.

THÉORÈME 4.2 DE SZPILRAJN *Tout ordre partiel admet une extension linéaire.*

Définition 4.6 Si G est acyclique, on définit $\leqslant_G$ par $s \leqslant_G t$ ssi il existe un chemin de s à t.

Proposition 4.4 $\leqslant_G$ est un ordre partiel.

Tri

On cherche une extension linéaire de $\leqslant_G$. On va réaliser un parcours en profondeur. L'extension linéaire est donnée par l'ordre décroissant des valeurs de $post[s]$.

THÉORÈME 4.3 *Ce principe est correct.*

Démonstration. Montrons que $(u \to v) \Rightarrow (post[u] > post[v])$.

Comme G est acyclique, il n'y a pas d'arc arrière dans le parcours en profondeur.

Quand `explorer`(u) est appelé, il y a trois possibilités :
- $u \to v$ est un arc avant : on a $pre[u] < pre[v] < post[v] < post[u]$.
- $u \to v$ est un arc appelant : on a $pre[u] < pre[v] < post[v] < post[u]$.
- $u \to v$ est un arc transverse : on a $pre[v] < post[v] < pre[u] < post[u]$.

∎

4.2.4 Composantes fortement connexes

Définition 4.7 Une composante fortement connexe est une classe d'équivalence de la relation $u\mathcal{R}v$ ssi il existe un chemin de u à v et de v à u.

Définition 4.8 On appelle graphe quotient de G le graphe $G' = (S', A')$ avec S' les composantes fortement connexes de G et A' l'ensemble des arêtes définies pas $C \overset{G'}{\to} D$ ssi $C \neq D$ et il existe $x, y \in C \times D$ tel que $x \overset{G}{\to} y$.

Proposition 4.5 G' est acyclique.

Algorithme de Kosaraju

Définition 4.9 Soit $G = (S, A)$ un graphe. on définit le tranposé de G, noté $G^t = (S^t, A^t)$ le graphe défini par $S^t = S$ et $A^t = \{(y, x), (x, y) \in A\}$.

On va effectuer un parcours en profondeur sur G et obtenir la liste L des sommets triés par ordre décroissant de $post_1[s]$ et on parcourt ensuite G^t avec une boucle principale qui parcours L.

Théorème 4.4 *Les conposantes fortement connexes sont les arbres issus du deuxième parcours.*

Démonstration. Soit $U \subset S$ non vide. On pose $pre_1(U) = \min_{s \in U} pre_1[s]$ et $post_1(U) = \max_{s \in U} post_1[s]$.

Lemme 4.4.1
Soit C et C' deux composantes fortement connexes distinctes, $x \in C$ et $y \in C'$.

Si $x \to y$ alors $post_1(C) > post_1(C')$.

Démonstration. Soit u le premier sommet visité de $C \cup C'$.

- Si $u \in C$, $post_1(C) = post_1[u]$ et pour tout $s \in C'$, $post_1[s] < post_1[u]$ donc $post_1(C') < post_1[u] < post_1(C)$.
- Si $u \in C'$, $post_1(C') = post_1[u]$ et pour tout $s \in C$, $post_1[s] < post_1[u]$ car u n'appelle pas de sommets de C. Donc $post_1(C) < post_1(C')$. ■

On va montrer par récurrence que pour tout k, les k premiers arbres obtenus pas le deuxième parcours sont des composante fortemant connexes de G.

P_0 est vraie et si P_k est vraie, notons C_x la composante fortement connexe qui contient x et A_x l'arbre issu de x dans le deuxième parcours.

- Si $y \in C_x$, `explorer`(y) appelle tous les `explorer`(s) avec $s \in C_x$ donc $y \in A_x$.
- S'il existe $y \in A_x \setminus C_x$, `explorer`(x) appelle indirectement `explorer`(y) don cil existe un chemin de x à y dans G^t. Dans ce chemin, on note s le premier sommet qui sort de G_x. Soit v le prédécesseur de s dans ce chemin.

Dans G, on a $s \to v$ donc $post_1(C') > post_1(C_x)$ avec C' la composante fortement connexe de s. Donc x n'est pas un sommet non vu tel que $post_1(x)$ soit maximal. Contradiction. Donc $A_x = C_x$ et P_{k+1} est vraie. ■

4.3 Parcours en largeur

4.3.1 Algorithme

Algorithme 18: Parcours en largeur

Entrées : Un graphe G et un sommet s

```
1  pour t ∈ S faire
2  └ d[t] := +∞
3  d[s] := 0
4  pred[s] := []
5  F :=créer_file([s])
6  tant que F ≠ ∅ faire
7  │ s' := F.défiler
8  │ pour t tel que s' → t faire
9  │ │ si d[t] = +∞ alors
10 │ │ │ d[t] := d[s'] + 1
11 │ │ │ enfile(F, t)
12 └ └ └ pred[t] := s'
```

4.3.2 Terminaison et complexité

THÉORÈME 4.5 *Chaque sommet est enfilé au plus une fois.*

Démonstration. Un sommet t qui a été dans la file est tel que $d[t]$ est fini. ■

THÉORÈME 4.6 *On a une complexité en $O(|S| + |A|)$.*

Démonstration. La boucle « tant que » a une complexité en $O(|S|)$.
Et la boucle « pour » a pour complexité $O(|A|)$. ■

4.3.3 Correction

<u>Théorème 4.7</u> *Pour tout t, $d[t]$ est la longueur d'un plus court chemin de s à t s'il en existe un.*

Démonstration. Notons $\delta(t)$ cette longueur si elle existe.

On a, par une récurrence claire, $\delta \leqslant d$.

Posons P_n : « Il y a un moment dans l'exécution tel que, pour tout $t \in S$,

- $\delta(t) \leqslant n \Rightarrow d[t] = \delta(t)$
- $\delta(t) > n \Rightarrow d[t] = +\infty$
- $\delta(t) = n$ ssi t est au sommet de la file ».

P_0 est claire. Si P_d est vraie, la file ne contient que des sommets de distance d. Le moment de P_{d+1} va être celui où la file contient les successeurs d'iceux.

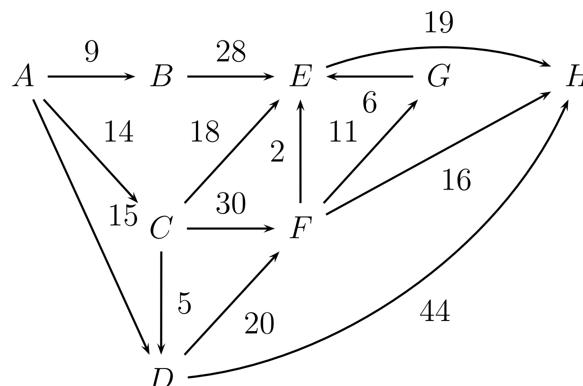
- Pour tout $t \in S$ tel que $\delta(t) \leqslant d+1$, si $\delta(t) \leqslant d$, $d[t] = \delta(t)$.
  Sinon, notons c un plus court chemin de s à t et u le prédécesseur de t dans c. Le sous-chemin $s \to u$ de c reste un plus court chemin et $\delta(u) = d$ donc, comme $d[t] = d[u] + 1$, $\delta(t) = d[t]$ et $t \in F_{d+1}$ car $u \in F_d$.
- Pour tout $t \in S$ tel que $\delta(t) > d+1$, $t \notin F_{d+1}$ sinon on aurait $\delta(t) = d[t] \leqslant d+1$. Donc t n'est pas mis à jour et $d[t] = +\infty$.
- Si $\delta(t) = d+1$, il existe $u \in F_d$ tel que $u \to t$ donc $t \in F_{d+1}$.
  Si $t \in F_{d+1}$, $d[t] = d[u] + 1 = d + 1$.

D'où P_{d+1}. ∎

4.3.4 Algorithme de Dijkstra

On s'intéresse aux plus courts chemins à partir d'une origine s où le graphe G est pondéré positivement.

Exemple :



Algorithme 19: Dijkstra

Entrées : Un graphe G et un sommet s
Sorties : Les longueurs $d[t]$ des plus courts chemins de s à t

```
1  pour t ∈ S faire
2      d[t] := +∞
3  d[s] := 0
4  pred[s] := []
5  F :=créer_file_priorité([s]) (triée avec d)
6  tant que F ≠ ∅ faire
7      u := F.défiler_min
8      pour t tel que u -p-> t faire
9          si d[t] > d[u] + p alors
10             d[t] := d[u] + p
11             pred[t] := u
12             MàJ(F)
```

A	B	C	D	E	F	G	H
0	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$
0	9	14	15	$+\infty$	$+\infty$	$+\infty$	$+\infty$
0	9	14	15	37	$+\infty$	$+\infty$	$+\infty$
0	9	14	15	32	44	$+\infty$	$+\infty$
0	9	14	15	32	35	$+\infty$	59
0	9	14	15	32	35	$+\infty$	51
0	9	14	15	32	35	46	51

THÉORÈME 4.8 *L'algorithme est correct.*

Démonstration. On a comme précédemment $\delta \leqslant d$.

On a aussi l'invariant : $\forall t \notin F$, $d[t] = \delta(t)$.

En effet, $d[s] = 0 = \delta(s)$. De plus, s'il n'y a pas de chemin de s à u, alors $+\infty = \delta(u) \leqslant d[u] = +\infty$.

Sinon, on a un plus court chemin c de s à u avant que u soit défilé. Soit y le premier sommet de c qui est dans F et x son prédécesseur.

Par l'invariant, $x \notin F$ donc $d[x] = \delta(x)$ donc $d[y] = \delta(y)$. De plus, comme u est sur le point d'être défilé, c'est le minimum de la file donc $d[u] \leqslant d[y] = \delta(y) \overset{\text{car } y \to u}{\leqslant} \delta(u) \overset{\text{lemme}}{\leqslant} d[u]$. Donc $d[u] = \delta(u)$. ∎

THÉORÈME 4.9 *L'algorithme effectue*

$$O(|S| + |S||\texttt{défiler_min}| + |\texttt{file_créer}| + |A||\texttt{MàJ}|)$$

opérations.

Remarque 4.1 *Avec une implémentation sous forme de tableau, on a une complexité en* $O(|S|^2 + |A|)$.

Sous forme d'un tas, c'est $O((|S| + |A|)\log(|S|))$.

Sous forme d'un tas de Fibonacci, c'est $O(|S|\log(|S|) + |A|)$.

Chapitre 5

Programmation dynamique

5.1 Exemple : suite de Fibonacci

5.1.1 Algorithme naïf

Algorithme 20: Algorithme naïf

Entrées : Un entier n
Sorties : fibo(n)

```
1 si n ⩾ 2 alors
2 | retourner 1
3 sinon
4 └ retourner fibo(n − 1)+fibo(n − 2)
```

Complexité exponentielle.

5.1.2 Algorithme itératif

Complexité linéaire.

5.1.3 Mémoïsation

Remarque 5.1 Dans le tri fusion, il n'y a pas de chevauchement des problèmes donc ces procédés ne sont pas utiles.

Algorithme 21: Algorithme itératif

Entrées : Un entier n
Sorties : Un tableau F tel que $F[n]$ =fibo(n)

```
1 si n ⩾ 2 alors
2 │ F[n] := 1
3 sinon
4 │ pour i ⩾ 3 faire
5 │ └ F[n] := F[n − 1] + F[n − 2]
6 retourner F
```

Algorithme 22: Algorithme mémoïsé

Entrées : Un entier n
Sorties : fibo(n)

```
1 si F[n] est défini alors
2 │ retourner F[n]
3 sinon
4 │ si n ⩾ 2 alors
5 │ │ F[n] := 1
6 │ sinon
7 │ └ F[n] :=fibo(n − 1)+fibo(n − 2)
8 └ retourner F[n]
```

5.2 Sous-structures optimales

5.2.1 Découpage des barres

Le problème est, étant donné un tableau P tel que $P[n]$ est le prix d'une barre de longueur n et un entier n_0, de savoir comment découper une barre de longueur n_0 pour gagner le plus d'argent.

Remarque 5.2 Tester toutes les solutions est inutile (on aurait une complexité exponentielle)

5.2.2 Sous-structures

Si on a une solution pour n_0, toute sous-découpe est une solution optimale pour sa longueur. On en déduit la relation de récurrence sur r_n (le prix

optimal d'une découpe d'une barre de longueur n) :

$$r_n = \max_{0 \leqslant i \leqslant n-1} (p_i + r_{n-i})$$

et $r_0 = 0$.

Cette formule nous conduit à l'algorithme :

Algorithme 23: Découpage des barres

Entrées : Un tableau P et un entier n
Sorties : r_n et le découpage associé

```
1 R[0] := 0
2 pour k = 1 à n faire
3     m := -∞
4     pour i = 1 à k faire
5         si m < p_i + R[k - i] alors
6             m := p_i + R[k - i]
7             decoupe[k] := i
8         R[k] := m
9 retourner R[n], decoupe
```

5.3 Plus court chemin

Cette fois, on s'autorise les poids négatifs.

Proposition 5.1 Soit G un graphe connexe sans cycles négatifs et s, t deux sommets. Il existe un plus out chemin de s à t.

Démonstration. $L_l = \{l(c), c \text{ chemin élémentaire de } s \text{ à } t\}$ est fini donc admet un minimum. ∎

5.3.1 Cas sans cycle

On note $\delta(t)$ la longueur d'un plus court chemin de s à t.

$\delta(t) = \min_{u, u \to t} (\delta(u) + poids(u, t))$.

Dans un tri topologique, si $u \to t$, $u \leqslant t$ donc chercher u crée un sous-problème.

Algorithme 24: Plus court chemin

Entrées : Un graphe G et un sommet s

Sorties : Un tableau d tel que pour tout sommet t, $d[t]$ est la longueur d'un plus court chemin de s à t

```
1 pour t ∈ S faire
2     d[t] := +∞
3 d[s] := 0
4 L :=tri_topo(G)
5 pour t ∈ L (dans l'ordre) faire
6     d[t] := min_{u→t}(d[u] + poids(u, t))
7 retourner d
```

5.3.2 Cycles positifs

Dans ce cas, le calcul des $\delta(u)$ n'est plus un sous-problème du calcul de $\delta(t)$.

On doit donc trouver d'autres sous-problèmes. Il y a deux méthodes :

- Si on a un plus court chemin à k étapes, le préfixe de $k-1$ étapes de ce chemin est un plus court chemin. On introduit alors l'algorithme de Bellman-Ford.
- Si $S = [\![1, n]\!]$, et si on a un plus court chemin élémentaire où les sommets sont inférieurs à k, soit tous les sommets intermédiaires sont inférieurs à $k-1$, soit il est de la forme $s \rightsquigarrow k \rightsquigarrow t$ avec des chemins dont les sommets sont inférieurs à $k-1$. Ce qui conduit à l'algorithme de Floyd-Warshall

5.3.3 Algorithme de Bellman-Ford

On note $\delta(k, t)$ la distance d'un plus court chemin de s à t en au plus k étapes.

On a la relation :

$$\delta(k, t) = \min\{\delta(k-1, t), \min_{u, u \to t}(\delta(k-1, u) + poids(u, t))\}$$

et $\delta(0, t) = 0$ si $t = s$ et $+\infty$ sinon.

On a donc complexité spatiale en $O(|S|^2)$. On va écrire une autre version où on ne retient pas toutes les valeurs.

On a une complexité spatiale en $O(|S|)$ et temporelle en $O(|S| + |A|)$.

Algorithme 25: Bellman-Ford

Entrées : Un graphe G sans cycles négatifs
Sorties : La distance minimale de s à t pour tout $t \in S$

```
1  pour t ∈ S faire
2  └ d[0, t] := +∞
3  d[0, s] := 0
4  pour k = 1 à |S| − 1 faire
5  │ pour t ∈ S faire
6  │ │ m := D[k − 1, t]
7  │ │ pour u tel que u → t faire
8  │ │ │ si m > D[k − 1, u] + poids(u, t) alors
9  │ │ └ └ m := D[k − 1, u] + poids(u, t)
10 └ └ D[k, t] := m
11 retourner D[|S| − 1, ·]
```

Algorithme 26: Bellman-Ford allégé

Entrées : Un graphe G sans cycles négatifs
Sorties : La distance minimale de s à t pour tout $t \in S$

```
1  pour t ∈ S faire
2  └ D[t] := +∞
3  d[s] := 0
4  pour k = 1 à |S| − 1 faire
5  │ pour u → t ∈ A faire
6  └ └ D[t] := min{D[u] + poids(u, t), D[t]}
7  retourner D
```

Remarque 5.3 On peut détecter les cycles négatifs en modifiant légèrement l'algorithme :

Algorithme 27: Détection de cycles négatifs

Entrées : Un graphe G

Sorties : Le booléen « il y a un cycle négatif »

```
1  pour t ∈ S faire
2  └ D[t] := +∞
3  d[s] := 0
4  pour k = 1 à |S| − 1 faire
5  │ pour u → t ∈ A faire
6  └ └ D[t] := min{D[u] + poids(u, t), D[t]}
7  pour u → t ∈ A faire
8  │ si D[t] > D[u] + poids(u, t) alors
9  └ └ retourner Frai
10 retourner Faux
```

Théorème 5.1 *S'il n'y a pas de cycles négatifs, alors $D[t] = \delta(t)$ pour tout $t \in S$.*

Démonstration. On montre les invariants $D[t] \geqslant \delta(t)$ et à la fin du k-ème passage, $D[t] \leqslant \delta(k, t)$.

À la fin, on a donc $\delta(t) \leqslant D[t] \leqslant \delta(|S| - 1, t)$. ∎

Théorème 5.2 *L'algorithme de détection de cycles négatifs fonctionne.*

Démonstration. S'il n'y a pas de cycles négatifs, $D(t) = \delta(t)$ et on n'a jamais $D(t) > D(u) + poids(u, t)$.

S'il y a un cycle $v_0, \cdots, v_k = v_0$ tel que $\sum_{i=0}^{k-1} poids(v_i, v_{i+1}) < 0$, supposons que l'algorithme ne dise rien.

$$\begin{aligned} D[v_0] &\leqslant D[v_{k-1}] + poids(v_{k-1}, v_0) \\ &\leqslant D[v_{k-2}] + poids(v_{k-1}, v_0) + poids(v_{k-2}, v_{k-1}) \\ &\cdots \\ &\leqslant D[v_0] + \sum_{i=0}^{k-1} poids(v_i, v_{i+1}) \end{aligned}$$

D'où la contradiction. ∎

5.3.4 Algorithme de Floyd-Warshall

On note $\delta(k,x,y)$ la longueur d'un plus court chemin de x à y où les sommes intermédiaires sont inférieurs à k.

On a la relation de récurrence :

$$\delta(k,x,y) = \min\{\delta(k-1,x,y), \delta(k-1,x,k) + \delta(k-1,k,y)\}$$

et $\delta(0,x,y) = poids(x,y)$ si $x \neq y$ et 0 sinon.

On donne directement l'algorithme à complexité spatiale réduite :

Algorithme 28: Floyd-Warshall allégé

Entrées : Un graphe G sans cycles négatifs
Sorties : La distance minimale de s à t pour tout $t \in S$

```
1  pour x, y ∈ S² faire
2      si x = y alors
3          D[x, y] := 0
4      sinon
5          D[x, y] := poids(x, y)
6  pour k = 1 à n faire
7      pour x ∈ S faire
8          pour y ∈ S faire
9              D[x, y] := min{D[x, y], D[x, k] + D[k, y]}
10 retourner D
```

La complexité spatiale est en $O(|S|^2)$ et la temporelle en $O(|S|^3)$.

Chapitre 6

Algorithmes gloutons

Définition 6.1 On appelle algorithme glouton un algorithme qui, à chaque étape, choisit le meilleur choix (maximum local) dans l'espoir d'obtenir un maximum global.

Par exemple, Dijkstra est un algorithme glouton. Rendre la monnaie utilise un algorithme glouton, contrairement aux (bonnes) IA d'échecs.

6.1 Arbre couvrant de poids minimum

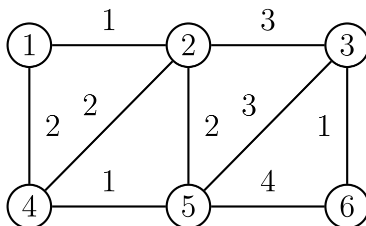
6.1.1 Le problème

À un graphe connexe pondéré non orienté $G = (S, A)$, on veut associer un arbre couvrant de poids minimal $T = (S', E)$ qui est un sous-graphe de G tel que $\sum_{(a,b)\in A} \text{poids}(a, b)$ est minimal. On trouve des applications pour des réseaux et des circuits électriques.

6.1.2 Algorithme de Kruskal

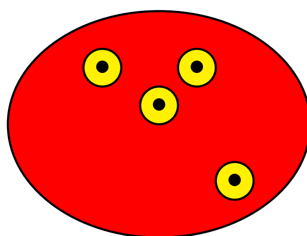
Structure de données

Tant qu'on n'a pas un arbre, on ajoute l'arête la plus légère si elle ne crée pas de cycles.

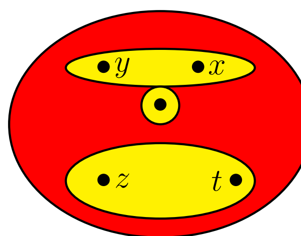


Pour cela, on va utiliser une structure de données abstraite pour représenter une partition d'un ensemble. Cette structure s'appelle Union-Find et ses opérations :

- `creer_union_find` : crée la partition $\bigcup_{s \in S} \{\{s\}\}$:
- `find` : Trouve la composante de x



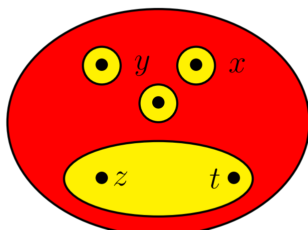
(a) `creer_union_find`



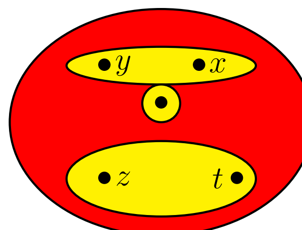
(b) `find`

On a `find`(x) = `find`(y) mais `find`$(x) \neq$ `find`(z)

- `union` : fusionne les composantes de x et y.



(c) Avant



(d) Après `union`(x, y)

Algorithme

<u>Théorème 6.1</u> *À la fin, E contient les arêtes d'un ACM.*

Démonstration. On va démontrer l'invariant : « il existe T ACM tel que $E \subset T$ et pour tout $x, y \in S$, x et y sont reliés pas un chemin de E ssi x et y sont dans la même composante dans CC. »

Initialisation : $E = \varnothing$ et il existe un ACM.

Conservation : Supposons qu'à une certaine étape, on ait l'invariant.

On ajoute une arête $\{x, y\}$.

Algorithme 29: Algorithme de Kruskal

Entrées : Un graphe $G = (S, A)$ pondéré non orienté
Sorties : Un ACM (arbre couvrant minimal)

```
1 CC := creer_union_find(S)
2 E := ∅
3 L := tri(A) par cordre croissant des poids.
4 pour {x, y} ∈ L (dans l'ordre) faire
5 |  si CC.find(x) ≠ CC.find(y) alors
6 |  |  E := E ∪ {{x, y}}
7 |  |  CC.union(x, y)
8 retourner E
```

Si $\{x, y\} \in T$, alors c'est bon. Sinon, il existe un chemin c de x' à y' avecc x' dans la composante de x et y' dans celle de y.

On définit T' par $T' = T \cup \{\{x, y\}\} \setminus \{\{u, v\}\}$ où $\{u, v\}$ est une arête de poids maximal dans c.

T' est connexe, T' et T ont même nombre d'arêtes ($|S| - 1$) donc c'est un arbre. De plus, $poids(T') = poids(T) + poids(\{x, y\}) - poids(\{u, v\}) \leqslant poids(T)$.

Donc T' est un ACM.

On a donc $E \subset T'$. Mais E est connexe. En effet, si $x, y \in S$, on sait que x et y sont reliés dans G : $x = x_1 \leftrightarrow x_2 \leftrightarrow \cdots \leftrightarrow x_n \leftrightarrow y$.

Si $\{x_i, x_{i+1}\} \notin E$, on avait x_i et x_{i+1} dans la même composante donc on a un chemin de x_i à x_{i+1} dans E. Donc on a un chemin de x à y dans E. La réciproque est claire. ∎

Complexité

Complexité : $|$`creer_union_find`$| + O(A\ln(A)) + O(A(|$`find`$| + |$`union`$|))$.

On peut utiliser des arbres : chaque partie de la partition est représentée par un arbre. On oriente les flèches : elles pointent vers le père. De plus, la racine est reliée à elle-même.

Algorithme 30: `creer_union_find`

Entrées : $S = \{s_1, \cdots, s_n\}$
Sorties : Union Find

```
1 retourner Les arbres à un seul sommet : [s_1, ..., s_n]
```

Algorithme 31: `find`

Entrées : x
Sorties : La racine de l'arbre qui contient x

```
1 si x = x.succ alors
2 |   retourner x
3 sinon
4 |_  retourner find(x.succ)
```

Algorithme 32: `union`

Entrées : x, y
Sorties : Un arbre qui contient x et y et tous les éléments des arbres qui les contenait.

```
1  r_x := find(x)
2  r_y := find(y)
3  G' := graphe formé de la réunion des arbres contenant x et y
4  si r_x = r_y alors
5  |_  Ne rien faire
6  si h(r_x) > h(r_y) alors
7  |   Ajouter l'arête r_y → r_x à G'
8  sinon
9  |_  Ajouter l'arête r_x → r_y à G'
10 Mettre à jour les hauteurs.
11 retourner G'
```

Proposition 6.1 Il y a au moins 2^k nœuds dans un arbre de hauteur k.

Démonstration. Faire une étude de cas. ∎

Corollaire 6.1 *S'il y a n éléments dans la structure, il y a au plus $\frac{n}{2^k}$ nœuds de hauteur k.*

Corollaire 6.2 *Les hauteurs des arbres de la forêt sont inférieures à $\log_2(n)$.*

Avec $O(B) = O(S) + O(A \ln A)$, on a :

Implémentation	Tableau	Arbres
|`creer_ union_ find`|	$O(S)$	$O(S)$
|`find`|	$O(1)$	$O(\log_2(n))$
|`union`|	$O(S)$	$O(\log_2(n))$
|Kruskal|	$O(B) + O(AS)$	$O(B) + O(A \log_2(S))$

Comme $|A| \geqslant |S| - 1$ (G connexe), on a une complexité de $O(A \log_2 A)$. Mais on peut faire encore mieux.

Amélioration

Quand on cherche un représentant avec `find`, on est idiots. On peut en profiter pour réduire le chemin d'un nœud à sa racine. On parle de compression de chemins.

On remplace l'implémentation de `find` par :

Algorithme 33: `find`

Entrées : x
Sorties : La racine de l'arbre qui contient x

1 **si** $x \neq x.succ$ **alors**
2     $x.succ :=$ `find`$(x.succ)$
3 **retourner** $x.succ$

Les rangs sont les hauteurs des arbres s'il n'y avait pas compression des chemins.

Les rangs sont entre 0 et $\ln n$.

On va découper l'intervalle $[1, \log_2(n)]$ en sous-intervalles de la forme $[k+1, 2^k]$.

Définition 6.2 On définit $\ln^*(n)$ le plus petit nombre k tel que :

$$\underbrace{\log_2(\log_2(\cdots(\log_2}_{k \text{ fois}}(n)\underbrace{)\cdots)}_{k \text{ fois}} \leqslant 1$$

Exemples : $\ln^*(2^{2^2}) = 3$ et $\ln^*\left(2^{2^{2^{2^2}}}\right) = 5$.

On va distribuer $n\ln^*(n)$ aux nœuds au cours de l'algo. On va montrer que `find` prend $O(n\ln^*(n))$ opérations plus une quantité de temps telle que l'exécution de m fois `find` aie une complexité en $O(m\ln^*(n)) + O(n\ln^*(n))$.

Un nœud reçoit de l'argent quand il cesse d'être racine (et son rang r ne sera plus modifié). Si $r \in [k+1, 2^k]$, on donne 2^k à ce nœud.

Le nombre de nœuds de rang k est au plus $\frac{n}{2^k}$.

Le nombre de nœuds de rang strictement supérieur à k est au plus $\frac{n}{2^{k+1}} + \frac{n}{2^{k+2}} + \cdots = \frac{n}{2^k}$.

Donc le nombre de nœuds de rang entre $k+1$ et 2^k est au plus $\frac{n}{2^k}$. L'argent distribué est majoré par :

$$\sum_{j=1}^{\ln^*(n)} \frac{n}{\underbrace{2^{2^{\cdot^{\cdot^{\cdot^2}}}}}_{j}} \underbrace{2^{2^{\cdot^{\cdot^{\cdot^2}}}}}_{j} = n\ln^*(n)$$

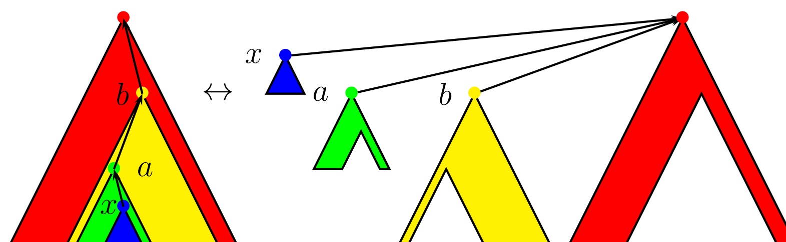
Regardons un appel de `find` :

- Soit les rangs de y et $y.pred$ ne sont pas dans le même intervalle, on comptabilise une opération pour chacun.
- Soit les rangs sont dans le même intervalle, alors y paie.
- À chaque fois qu'un nœud paie, on le connecte à la racine dont le rang est stristement plus grand que le rang $y.pred$.

Si le rang de $y \in [k+1, 2^k]$, alors, au bout de 2^k paiement, y et $y.pred$ auront des rangs dans des intervalles différentes et donc y ne paiera plus.

Appliquer m fois `find` coûte $O(m\ln^*(n))$ opérations et $O(n\ln^*(n))$ opérations.

En fait, on transforme :



Bilan : Dans le cas général, on a une complexité en $O(A\ln(A))$. Si le tri de A a déjà été fait alors la complexité est en $O(A\ln^*(S))$.

6.1.3 Algorithme de Prim

On utilise une stratégie gloutone : on ajoute à E (arbre qui grossit pendant l'algorithme) un arc $e \in E \to x \notin E$ de poids minimum.

Algorithme 34: Algorithme de Prim

Entrées : Un graphe $G = (S, A)$
Sorties : Un ACM de G

```
1  pour s ∈ S faire
2  │ cout[s] :=+∞
3  └ pred[s] :=∅
4  u0 := un sommet de G
5  cout[u0] :=0
6  F := filedepriorité (S,cout)
7  tant que F ≠ ∅ faire
8  │ t := F.défilemin
9  │ pour (u,p) tel que t → u ∈ A faire
10 │ │ si cout[u]> p alors
11 │ │ │ cout[u] :=p
12 │ │ │ Màj(F)
13 └ └ └ pred[u] :=t
14 retourner pred
```

6.2 Un algortihme glouton approximant une couverture d'ensemble

On a un ensemble de villes et on veut placer des écoles acvec deux contraintes : chaque école est dans une ville et il doit y avoir une école à moins de 10km de chaque village.

On peut écrire un algorithme exact mais sa complexité va être exponentielle.

Si on utilise une statégie gloutonne, on a un algortihme approximatif mais efficace. À chaque fois, on choisit un ensemble S_j avec le plus d'éléments non couverts.

THÉORÈME 6.2 *Si* $\operatorname{Card} B = n$ *et si la couverture optimale en contient* k*, alors l'algorithme glouton renvoie au plus* $k\lceil\ln(n)\rceil$ *sous-ensembles.*

Algorithme 35: Couverture

Entrées : Un ensemble B et $\mathcal{E} = \{S_1, \cdots, S_n\} \subset \mathscr{P}(B)$

Sorties : $J \subset [\![1, n]\!]$ tel que $\bigcup_{j \in J} S_j = B$ et $\text{Card}(J)$ est minimal

```
1 L := []
2 C := ∅
3 tant que C ≠ B faire
4     choisir j ∈ [[1, n]] tel que Card(S_j ∩ C^c) soit maximal.
5     si S_j ⊂ C alors
6         retourner Impossible
7     C := C ∪ S_j
8     L := j :: L
9 retourner L
```

Démonstration. On note n_t le nombre d'éléments non couverts au t^{e} tour de boucle.

Dans une situation optimale, $\mathcal{S} = (S_j)_{j \in J}$, il y a un ensemble S_j avec au moins $\frac{n_t}{k}$ éléments non couverts.

Par l'absurde, à l'étape t l'algortihme va choisir un ensemble S_j tel que $\text{Card}(S_j \cap \mathcal{C}_t^c) \geqslant \frac{n_t}{k}$.

Donc $n_{t+1} \leqslant n_t - \frac{n_t}{k} = n_t(1 - \frac{1}{k})$ donc $n_t \leqslant n(1 - \frac{1}{k})^t < n\mathrm{e}^{-\frac{t}{k}}$.

Et $n\mathrm{e}^{-\frac{t}{k}} \leqslant 1 \Rightarrow t \geqslant k\lceil \ln(n) \rceil$. ■

Chapitre 7

Flots et programmation linéaire

7.1 Flots

7.1.1 Définitions

Définition 7.1 Un réseau de flot est un graphe orienté irréflexif, antisymétrique, pondéré positivement pas des entiers dans lequel on distingue dux sommets : une source s et un puits t tel que pour tout sommet v on a un chemin de s à v et de v à t.

On note $c(u, v)$ la capacité de l'arête (u, v). (Si cette arête n'existe pas, c'est 0)

Définition 7.2 Un flot de $G = (S, A)$ est une fonction $f : S^2 \to \mathbb{R}^+$ telle que :

- $\forall u, v \in S$, $0 \leqslant f(u, v) \leqslant c(u, v)$.
- Loi de Kirchhoff : $\forall u \in S \setminus \{s, t\}$, $\sum\limits_{v \in S} f(v, u) = \sum\limits_{v \in S} f(u, v)$.

Définition 7.3 On appelle valeur du flot f la valeur $|f| = \sum\limits_{v \in S} f(s, v) - \sum\limits_{v \in S} f(v, s)$.

Le problème du flot maximum est, étant donné un réseau G, de trouver un flot f tel que $|f|$ est maximal.

7.1.2 Échec d'une méthode gloutonne

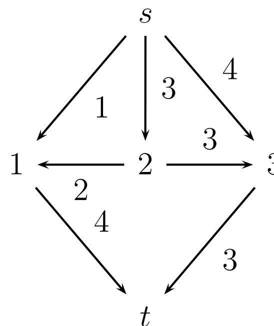
Exemple :

Algorithme 36: Problème du flot

Entrées : Un réseau G

Sorties : Un flot f tel que $|f|$ est maximal

1 f :=flot_nul
2 **tant que** ∃ *un chemin simple de s à t qui améliore le flot* **faire**
3 augmenter f
4 **retourner** f



était censé ne pas faire marcher l'algorithme. Pas de bol, l'algorithme est juste sur ce graphe.

7.1.3 Réseaux résiduels, algorithme de Ford-Fulkerson

Algorithme

Définition 7.4 On définit le graphe résiduel de $G = (S, A)$ et f le graphe G_f où les sommets sont S et où la fonction de poids est :

$$c_f(u,v) = \begin{cases} c(u,v) - f(u,v) & \text{si } (u,v) \in A \\ f(v,u) & \text{si } (v,u) \in A \\ 0 & \text{sinon} \end{cases}$$

Démonstration

THÉORÈME 7.1 *On a l'invariant : « f est un flot ».*

THÉORÈME 7.2 *L'algorithme termine.*

Démonstration. La valeur du flot $|f|$ augmente strictement et est majorée par $|f^*|$ où f^* est un flot maximal.

Algorithme 37: Ford-Fulkerson

Entrées : Un graphe G et les sommets s et t
Sorties :

1 f :=flot nul
2 **tant que** $\exists$ *un chemin simple c de s à t dans G_f* **faire**
3 $\quad c_{\min} = \min\{c_f(u,v), u \to v \in c\}$
4 $\quad$ **pour** $u \to v \in c$ **faire**
5 $\quad\quad$ **si** $u \to v \in A$ **alors**
6 $\quad\quad\quad f(u,v) := f(u,v) + c_{\min}$
7 $\quad\quad$ **sinon**
8 $\quad\quad\quad f(v,u) := f(v,u) - c_{\min}$
9 **retourner** f

En effet, $|f_{nouveau}| = \sum_{v \in S} f_{nouveau}(s,v) - \sum_{v \in S} f_{nouveau}(v,s)$.

Si $s \to x$, $f_{nouveau}(s,x) = f(s,x) + c_{\min}$ et si $x \to s$, $f_{nouveau}(x,s) = f(x,s) - c_{\min}$ et on a bien $|f_{nouveau}| = |f| + c_{\min}$. ■

Complexité

On a une complexité en $O(|f^*|(S+A))$.

Remarque 7.1 L'algorithme peut être amélioré en choisissant à chaque fois un plus court chemin en nombre d'arêtes dans G_f. Cette variante s'appelle l'algorithme de Edmonds-Karp *qui a une complexité en $O(SA^2)$. Il y en existe d'autres encore meilleurs.*

7.1.4 Théorème du flot maximum

Définition 7.5 Une coupe (E,T) d'un réseau (G,s,t) est une partition de G telle que $s \in E$ et $t \in T$.

Définition 7.6 Soit f un flot et (E,T) une coupe.

Le flot net $f(E,T)$ à travers la coupe (E,T) est défini par :

$$f(E,T) = \sum_{(u,v) \in E \times T} f(u,v) - \sum_{(u,v) \in E \times T} f(v,u)$$

On définit aussi la capacité de coupe par $c(E,T) = \sum_{(u,v) \in E \times T} c(u,v)$.

Lemme 7.2.1
$|f| = f(E, T)$

Démonstration. C'est grâce à Kirchhoff. ■

THÉORÈME 7.3 *Soit G un réseau et f un flot. On a équivalence entre :*
- *f est un flot maximal dans G*
- *Il n'existe pas de chemin de s à t dans le graphe résiduel G_f.*
- *Il existe une coupe (E, T) tel que $|f| = c(E, T)$.*

Démonstration.
1 ⇒ 2 Si 2 est faux, il y a un chemin améliorant et Ford-Fulkerson trouve un meilleur flot que f donc 1 est faux.
3 ⇒ 1 Pour tout flot g, on a $|g| \leqslant c(E, T) = |f|$ donc f est maximal.
2 ⇒ 3 Supposons qu'il n'existe pas de chemin améliorant.
Posons $E = \{u \in S, \exists c \in G_f, s \xrightarrow{c} u\}$ et $T = S \setminus E$. C'est bien une coupe.
Montrons que $|f| = c(E, T)$.
On a en fait $c(E, T) = f(E, T)$ puisque

$$f(E, T) = \sum_{(u,v)\in E\times T} f(u, v) - \sum_{(u,v)\in E\times T} f(v, u)$$

Soit $u \in E$ et $v \in T$ tel que $u \to v \notin G_f$.
Si $u \to v \in G$, $f(u, v) = c(u, v)$ et $f(v, u) = 0$
Si $v \to u \in G$, $f(u, v) = 0 = c(u, v)$ et $f(v, u) = 0$.
Si $u \to v \notin G$ et $v \to u \notin G$, $f(u, v) = f(v, u) = 0 = c(u, v)$.
Donc $f(E, T) = \sum_{(u,v)\in E\times T} c(u, v) = c(E, T)$. ■

COROLLAIRE 7.1 *Ford-Fulkerman est correct.*

7.2 Programmation linéaire

7.2.1 Problème

Un problème de programmation linéaire consiste à maximiser ou minimiser une fonction linéaire soumise à un nombre fini de contraintes linéaires.

Exemple : maximiser $2x+3y-5z$ sachant que $x = y+2$ et $y \leqslant z+34x-3$.

Il se peut que le problème n'ait pas de solutions : maximiser $x + y$ avec $x \geqslant 2$ et $y \leqslant 5$.

Il se peut que l'ensemble des contraintes soit inconsistant : maximiser x avec $x \leqslant 1$ et $x \geqslant 2$.

Il se peut que le maximum ne soit pas atteint : maximiser x pour $x < 1$.

7.2.2 Réduction

Définitions

On dit qu'il existe une réduction polynômiale d'un problème P_1 vers un problème P_2 ssi il existe une fonction f calculable en temps polynômial tel que si x est une entrée de P_1, alors $f(x)$ est une entrée de P_2 et un algorithme qui résoud P_2 pour l'entrée $f(x)$ permet de résoudre P_1 pour l'entrée x.

Exemples

Trouver le plus court chemin revient à maximiser une fonction d_u avec les contraintes $d_s = 0$ et $d_v \leqslant d_t + poids(v,t)$.

Trouver un flot maximal revient à maximiser $|f|$.

7.3 Introduction à l'algorithme du simplexe

7.3.1 Principe général

Si un problème linéaire a une solution, on peut se ramener au problème suivant appellé problème canonique :

- Entrée : $c \in \mathbb{R}^n$, $A \in \mathfrak{M}_{k,n}(\mathbb{R})$ et $b \in \mathbb{R}_+^n$.
- Sortie : $x \in \mathbb{R}^n$ tel que $c^t x$ soit maximal sous les contraintes $Ax \leqslant b$ et $x \geqslant 0$.

Exemple du chocolatier

Il vend des chocolats simples (1 euro), des pyramides (6 euros) et des pyramides de luxe (13 euros)

Au maximum, il peut vendre 200 boites simples, 300 pyramides, pas plus de 400 boîtes en tout et le nombre de pyramides plus trois fois le nombre de pyramides de luxe est au plus 600.

On doit maximiser $x_1 + 6x_2 + 13x_3$ avec les contraintes :

$$\begin{cases} x_1 \leqslant 200 \\ x_2 \leqslant 300 \\ x_1 + x_2 + x_3 \leqslant 400 \\ x_2 + 3x_3 \leqslant 600 \end{cases}$$

Les contraintes définissent une région de réalisabilité qui est convexe en tant qu'intersection de convexes. Si le maximum est atteint, il l'est sur un des sommets.

L'algorithme du simplexe parcourt les sommets à la recherche du maximum en améliorant à chaque fois la fonction objectif.

Résolution

Au début, on est en $x_1 = x_2 = x_3 = 0$.

Itération : on choisit x_e nulle en choisissant e minimal et x_e qui fait augmenter l'objectif et on choisit x_s de la base à faire sortir (on choisit celui avec la contrainte la plus forte).

- On va choisir de faire entrer x_1. On fait sortir $\varepsilon_1 = 200 - x_1$.
  On est donc amenés à maximiser $200 - \varepsilon_1 + 6x_2 + 13x_3$ avec :

$$\begin{cases} x_1 & = 200 - \varepsilon_1 \\ \varepsilon_2 & = 300 - x_2 \\ \varepsilon_3 & = 200 + \varepsilon_1 - x_2 - x_3 \\ \varepsilon_4 & = 600 - x_2 - 3x_3 \end{cases}$$

- On fait rentrer x_2 et sortir ε_3.
  On maximise donc $1400 + 5\varepsilon_1 - 6\varepsilon_3 + 7x_3$ avec :

$$\begin{cases} x_1 & = 200 - \varepsilon_1 \\ \varepsilon_2 & = 100 - \varepsilon_1 + \varepsilon_3 + x_3 \\ x_2 & = 200 + \varepsilon_1 - \varepsilon_3 - x_3 \\ \varepsilon_4 & = 400 - \varepsilon_1 + \varepsilon_3 - 2x_3 \end{cases}$$

- On fait rentrer x_3 et sortit ε_2.
  On maximise $2800 + 12\varepsilon_1 - 13\varepsilon_3 - 7\varepsilon_2$ avec :

$$\begin{cases} x_1 & = 200 - \varepsilon_1 \\ \varepsilon_2 & = 300 - x_2 \\ x_3 & = 200 + \varepsilon_1 - \varepsilon_3 - x_2 \\ \varepsilon_4 & = -3\varepsilon_1 + 3\varepsilon_3 + 2x_2 \end{cases}$$

- On fait rentrer ε_1 et sortir ε_4.
  On maximise $800 - \varepsilon_3 - 4\varepsilon_4 + x_2$ avec :

$$\begin{cases} x_1 & = 200 - \varepsilon_3 + \frac{\varepsilon_4}{3} - \frac{2x_2}{3} \\ \varepsilon_2 & = 300 - x_2 \\ x_3 & = 200 - \frac{\varepsilon_4}{3} - \frac{x_2}{3} \\ \varepsilon_1 & = \varepsilon_3 - \frac{\varepsilon_4}{3} + \frac{2x_2}{3} \end{cases}$$

- On fait rentrer x_1 et sortir x_2.

Chapitre 8

Des problèmes difficiles

Dans les chapitres précédents, on a vu beaucoup d'algorithmes en temps polynômial. Mais ce n'est pas toujours le cas : il existe des problèmes pour lesquels il n'y a pas d'algorithmes pour les résoudre. On les dits indécidables.

8.1 Introduction : le problème de l'arrêt

<u>Théorème 8.1</u> *On ne peut pas décider de la terminaison d'un algorithme.*

Démonstration. Supposons qu'on ait un algorithme `termine` qui, étant donné un algorithme, renvoie le booléen « l'algortihme termine ».

Posons alors l'algorithme :

Algorithme 38: paradoxe

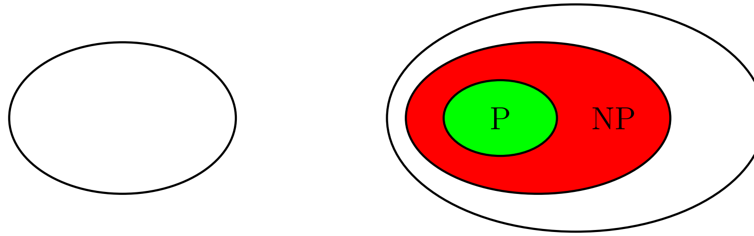
Entrées : Un algorithme z

1 **si** *termine*(z) **alors**
2   boucler

Si on considère `paradoxe(paradoxe)`, on obtient que cet algorithme termine ssi il ne termine pas.

D'où la contradiction. ■

On va donc classer les problèmes :



Indécidables Décidables

8.2 Classes de complexité P et NP

8.2.1 Intuition

Définition 8.1 On appelle P la classe des problèmes de décision ie tels qu'il existe un algorithme qui résout le problème.

On appelle NP la classe des problèmes donc on peut écrire un algorithme qui vérifie en temps polynômial si une solution est correcte ou non.

8.2.2 Une machine pour modéliser un algorithme

Un problème de décision est équivalent à un ensemble de mots.

Un algorithme est équivalent à une machine à états qui possède un ruban de travail.

Définition 8.2 (Machine de Türing) Soit Σ un alphbet fini.

Une machine de Türing est un triplet $M = (Q, \delta, s)$ avec Q un ensemble d'états, $\delta \subset (Q \times \Sigma) \times (Q \cup \{oui\}) \times \Sigma \times \{-1, 0, 1\}$ un ensemble de transitions et $s \in Q$ un état initial.

On dit qu'une machine de Türing est déterministe ssi pour tout $(q, a) \in Q \times \Sigma$, $\text{Card}\{(q', b, d), (q, a, q', b, d) \in \delta\} \leqslant 1$.

Remarque 8.1 Quand la machine est déterministe, le temps peut être représenté linéairement. Dans le cas contraire, on le représente par un arbre.

Définition 8.3 On appelle configuration un triplet (q, k, w) avec q un état, $k \in \mathbb{N}$ et w un mot sur Σ^*.

Définition 8.4 On note $(q, k, w) \to (q', k', w')$ ssi il existe $(q, \sigma, q', \rho, dir) \in \delta$ tel que :

- $w[k] = \sigma$,
- w' et w ont les même caractères, sauf le k-ème : $w'[k] = \rho$,
- $k' = k + dir$,
- $k' \geqslant 0$.

On notera $c \to^t c'$ pour dire qu'on peut passer de la configuration c à c' en t étapes et $c \to^* c'$ pour dire qu'on peut passer de c à c' en un nombre fini d'étapes.

Définition 8.5 Soit L un langage.

M décide L en temps polynômial ssi il existe un polynôme P tel que :

- Pour tout $x \in \Sigma^*$ et $t \in \mathbb{N}$, $(s, 0, x) \to^t (q, k, w)$ avec $t \leqslant P(|x|)$.
- $x \in L$ ssi il existe une exécution $(s, 0, x) \to^* (oui, k, w)$.

Définition 8.6 On appelle P la classe des langages tels qu'il existe une machine M déterministe qui décide de L en temps polynômial.

On appelle NP la classe des langages tels qu'il existe une machine M qui décide de L en temps polynômial.

8.2.3 Difficulté

Définition 8.7 Un problème (ie un langage) est dit NP-dur ssi pour tout langage $L' \in NP$, il existe une réduction polynômiale de L' dans L, c'est-à-dire qu'il existe $f : \Sigma^* \to \Sigma^*$ calculable en temps polynômial telle que pour tout $x \in \Sigma^*$, $x \in L' \quad \text{ssi} \quad f(x) \in L$.

Définition 8.8 On dit que L est NP-complet ssi $L \in NP$ et L est NP-dur.

8.3 Le pouvoir de la logique propositionnelle

8.3.1 La logique propositionnelle

Définition 8.9 On définit les formules propositionnelles inductivement :

- Les variables en sont.
- Si P en est une, $\neg P$ l'est aussi.
- Si P et Q en sont, $P \wedge Q$ et $P \vee Q$ en sont aussi.

Définition 8.10 On dit qu'une formule est satisfiable ssi on peut assigner des valeurs de vérité aux variables qui rendent la formule vraie.

8.3.2 La puissance du problème SAT

Le problème SAT consiste à déterminer la satisfiabilité d'une formule passée en argument.

Application

Certains problèmes : coloriage de carte, sudoku, ..., s'encodent dans SAT. Ces problèmes sont NP.

NP-complétude

<u>Théorème 8.2</u> *SAT est NP-complet.*

Démonstration. On admet que SAT est NP-complet. Mais on va quand même montrer que SAT est NP et NP-dur.

SAT est clairement NP : il suffit de générer toutes les possibilités et de les vérifier.

Pour montrer que SAT est NP-dur, on prend un problème L qui est NP.

On veut trouver f tel que $x \in L$ ssi $f(x)$ est une formule satisfiable.

En fait, on prend $f(x)$ qui encode toutes les contraintes d'une éxécution qui réussit d'une machine de Türing qui décide L avec x écrit sur le ruban dans la configuration initiale. ∎

8.3.3 3-SAT

<u>Définition 8.11</u> On appelle 3-SAT le problème SAT restreint aux formes normales conjonctives où chaque terme de la conjonction est une disjonction d'au plus 3 éléments.

<u>Théorème 8.3</u> (Admis) 3-*SAT est NP-complet.*

8.4 Montrer la NP-complétude par réduction

<u>Théorème 8.4</u> *Si L est NP-dur et L se réduit à L' alors L' est NP-dur.*

Démonstration. Clair ∎

8.4.1 Ensembles indépendants dans un graphe

<u>Définition 8.12</u> Le problème ENS INDEP est le suivant : on a un graphe $G = (S, A)$ non orienté et $k \in \mathbb{N}$ et on cherche la valeur du booléen « il existe un ensemble S' de taille k tel que $A \cap (S' \times S') = \varnothing$.

<u>Théorème 8.5</u> *ENS INDEP est NP-complet.*

Démonstration. Si on a G et k, on choisit un ensemble S' de taille k et on vérifie en temps polynômial que $A \cap (S' \times S') = \varnothing$. ∎

<u>Théorème 8.6</u> *ENS INDEP est NP-dur.*

Démonstration. On va réduire 3-SAT à ENS INDEP ie construire f qui, à toute formule normale conjonctive à au plus trois éléments φ, associe une instance de ENS INDEP.

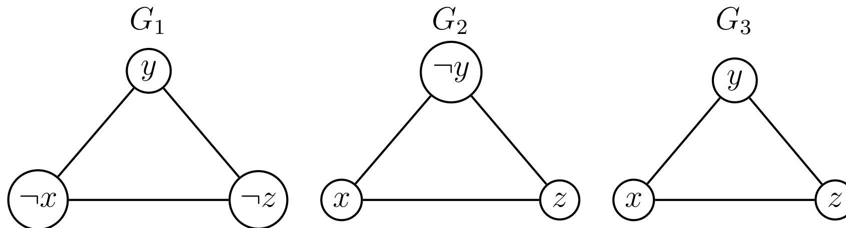
$f(\varphi)$ est la donnée d'un graphe G et d'un entier.

On va construire le graphe G comme suit :

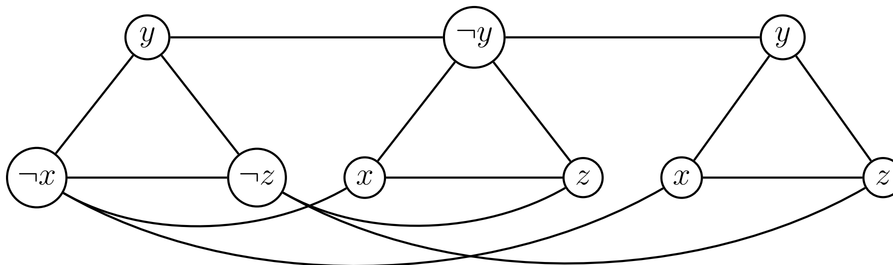
- On prend les graphes G_1, G_2 et G_3 correspondant aux disjonctions
- On relie p et $\neg p$ pour toute variable p.

Exemple :

$\varphi = (\neg x \vee y \vee \neg z) \wedge (x \vee \neg y \vee z) \wedge (x \vee y \vee z)$.



D'où le graphe total :



On a φ satisfiable ssi il existe $S' \subset S$ de taille k, $A \cap (S' \times S') = \varnothing$. ∎

8.4.2 Le problème CLIQUE

Définition 8.13 Étant donné un graphe non orienté et $k \in \mathbb{N}$, on cherche s'il existe une clique de taille k dans G ie un S' de taille k tel que pour tout $s, t \in S' \times S'$, $s - t$.

Théorème 8.7 *CLIQUE est NP-complet.*