
ALGORITHMIQUE 2

PASCAL VANIER

TABLE DES MATIÈRES

Table des matières	iii
Liste des Symboles	v
Introduction	1
1 Complexité	3
1 Complexité dans le pire des cas	3
2 Compter approximativement et asymptotiquement	4
2.1 Notation de Landau : comparaisons asymptotiques	4
2.1.1 Domination	4
2.1.2 Même ordre	4
2.1.3 Négligeabilité	4
2.2 Quelques ordres de grandeur	5
3 Quelques exemples et quelques pièges	5
3.1 Tri insertion	5
3.2 Complexité cachée	6
2 Diviser pour régner	7
1 Tri Fusion	7
2 Recherche Dichotomique	8
3 Master Theorem	9
4 Multiplication Matricielle	10
4.1 Algorithme Naïf	10
4.2 Diviser pour régner sans astuce	11
4.3 Diviser pour régner avec astuce (Strassen)	11
5 Enveloppe Convexe	12
5.1 Marche de Jarvis	12
5.2 QuickHull	14
3 Graphes et parcours	17
1 Graphe	17
2 Stockage des graphes	19
2.1 Matrice d'adjacence	19
2.2 Listes d'adjacence	20
2.3 Tables de hachage	20
2.3.1 Tables de hachage	20
2.3.2 Graphes dans des tables	20
2.4 Tableau comparatif	21

3	Parcours en largeur	21
3.1	Complexité	22
4	Parcours en profondeur	24
4.1	Temps de parcours	24
4.2	Complexité	25
4.3	Version itérative	27
5	Tri topologique	27
4	Backtracking	29
1	Problème des n -Reines	29
1.1	Parcours des possibilités	30
1.1.1	Naïvement	30
1.1.2	En prenant en compte les contraintes	31
1.1.3	Arbre des possibilités	32
2	Sudoku	32
5	Programmation Dynamique	35
1	Plus longue sous-suite commune	37
1.1	Le problème	37
1.2	Trouver une formule de récurrence	37
1.3	Calcul	38
2	Floyd-Warshall : tous les plus courts chemins	40
2.1	Relation de récurrence grâce au nombre de sommets	41
2.2	Relation grâce aux sommets par lesquels passe un chemin	41
3	Memoïzation	42
4	Dominos sur une ligne $n \times 3$	43
6	Reductions et NP-complétude	49
1	Encodage	49
1.1	Encodages non équivalents	50
2	Problème	51
3	Réduction polynomiale	51
4	Exemples de réductions	52
4.1	$SAT \leq_P SUBSETSUM$	52
4.2	$SAT \leq_P 3SAT$	53
4.3	$SAT \leq_P LONGESTPATH$	54
5	NP-complétude	54
Bibliography		57
Index		59

LISTE DES SYMBOLES

$\mathcal{B}(z, n)$	Boule centrée en z de rayon $n : z + \llbracket -n, n \rrbracket^d$
\subset_{fin}	Sous-ensemble fini
e_i	Vecteur unité de la i -ème dimension $(0, \dots, 0, 1, 0, \dots, 0)$
$v_1 \cdot v_2$	Produit scalaire des vecteurs v_1 et v_2

INTRODUCTION

Le but de ce cours est de vous apprendre à construire des algorithmes, de préférence efficaces. Dans cette optique, nous allons vous construire une “boîte à outils” d’algorithmes et de techniques classiques. Ceux-ci ne seront pas toujours parfaitement adaptés aux problèmes que vous rencontrerez et il faudra donc savoir vous adapter en les modifiant, en utilisant plusieurs, en réfléchissant à vos propres solutions innovantes.

Nous allons donc aborder dans ce cours plusieurs grands paradigmes de l’informatique : diviser pour régner, les graphes, le backtracking ainsi que la programmation dynamique.

1

COMPLEXITÉ

La complexité d'un algorithme est une mesure de son efficacité, celle-ci peut être exprimée en terme de nombre d'étapes élémentaires, on parle alors de *complexité en temps* ou en terme d'espace utilisé, on parle alors de *complexité en espace*. La complexité d'un algorithme sera toujours exprimée en fonction de la taille n de son entrée, il s'agira donc d'une fonction de n .

1 Complexité dans le pire des cas

Il y a plusieurs manières de considérer les ressources utilisées par un algorithme ou un programme. Traditionnellement on s'intéresse à la complexité en temps dans le pire des cas, c'est à dire le nombre d'opérations ou d'étapes de calcul maximal pour une taille d'entrée donnée. Prenons la fonction suivante :

```
boolean contient_un(int[] tab){  
    for(int i=0; i<tab.length; i++){  
        if(tab[i]==1) return true;  
    }  
    return false;  
}
```

Cette fonction prend une entrée : tableau `tab` d'entiers. Ici la taille de l'entrée, n , est la longueur de `tab`.

Pour les tableaux qui contiennent un 1 à la première case, cette fonction s'arrêtera toujours en le même nombre d'étapes : un nombre constant d'étapes. En revanche, le pire cas possible est quand il n'y a aucun 1 ou alors quand un éventuel un est à la dernière case, dans ce cas le tableau entier est parcouru et le nombre d'étapes est linéaire en n .

Compter la complexité dans le pire des cas c'est ignorer tous les cas les plus favorables pour se concentrer sur celui qui prendra le plus de temps à être traité.

2 Compter approximativement et asymptotiquement

La manière dont nous venons de compter les opérations n'était pas très précise ("linéaire en n "), combien d'opérations précisément sont effectuées à chaque tour de boucle ? Combien d'opérations sont effectuées par `if(tab[i]==1)` ? La réponse précise n'est pas très importante, mais on voit aisément que l'on peut borner ce nombre par une constante. Dans toute la suite de ce cours nous compterons donc approximativement en utilisant la notation $\mathcal{O}(f(n))$ qui permet de s'affranchir des constantes les moins importantes.

La notation $g(n) = \mathcal{O}(f(n))$ exprime le fait que la fonction g croît au plus comme la fonction f , à une constante multiplicative près, formellement :

$$\exists n_0, K \quad n \geq n_0 \Rightarrow g(n) \leq Kf(n)$$

Cette notation est particulièrement appropriée pour compter la complexité dans le pire des cas étant donné qu'elle donne une *borne supérieure*. On notera que lorsque :

$$\lim_{n \rightarrow \infty} \frac{g(n)}{f(n)} < \infty,$$

on a alors $g(n) = \mathcal{O}(f(n))$.

2.1 Notation de Landau : comparaisons asymptotiques

La notation $g(n) = \mathcal{O}(f(n))$ dénote que la fonction g croît au plus comme la fonction f et fait partie d'une famille de notation appelée notation de Landau qui compare la croissance de fonctions.

2.1.1 ▶ Domination

Quand $g(n) = \mathcal{O}(f(n))$, on dira donc que la fonction f *domine* la fonction g .

2.1.2 ▶ Même ordre

Deux fonctions peuvent se dominer l'une et l'autre : $f(n) = \mathcal{O}(g(n))$ et $g(n) = \mathcal{O}(f(n))$ on dit dans ce cas qu'elles sont du *même ordre* et on note $g(n) = \Theta(f(n))$. Dans ce cas les fonctions croissent de manière relativement similaire.

2.1.3 ▶ Négligeabilité

Lorsqu'une fonction g croît bien moins vite qu'une fonction f , on notera $g(n) = o(f(n))$, formellement :

$$g(n) = o(f(n)) \quad \Leftrightarrow \quad \forall K, \exists n_0, \forall n > n_0, \quad |g(n)| \leq K|f(n)|.$$

On notera que lorsque :

$$\lim_{n \rightarrow \infty} \frac{g(n)}{f(n)} = 0,$$

on a alors $g(n) = o(f(n))$.

2.2 Quelques ordres de grandeur

En pratique, on peut traduire la complexité d'un algorithme directement en temps de calcul en supposant qu'un ordinateur effectue 1000000 d'opérations par seconde :

	n	$n \log n$	n^2	n^3	1.5^n	2^n	$n!$
10	< 1s	< 1s	< 1s	< 1s	< 1s	< 1s	< 4s
30	< 1s	< 1s	< 1s	< 1s	< 1s	18 min	10^{25} ans
50	< 1s	< 1s	< 1s	< 1s	11 min	36 ans	∞
100	< 1s	< 1s	< 1s	1s	12,9 ans	10^{17}	∞
1000	< 1s	< 1s	1s	18 min	∞	∞	∞
10000	< 1s	< 1s	2 min	12 jours	∞	∞	∞
100000	< 1s	2s	3 heures	32 ans	∞	∞	∞
1000000	1s	20s	12 jours	31710 ans	∞	∞	∞

Ce tableau est tiré de [TK05].

3 Quelques exemples et quelques pièges

3.1 Tri insertion

Le tri insertion est le tri consiste à chercher le plus petit élément du tableau et à le mettre à la place du premier élément puis à recommencer. Une des implémentations possibles est la suivante :

```
def tri(tab):
    for i in range(len(tab)):
        for j in range(i, len(tab)):
            if(tab[i] > tab[j]):
                c = tab[j]
                tab[j] = tab[i]
                tab[i] = c

    return tab;
```

La complexité du tri suivant est en $\mathcal{O}(n^2)$.

3.2 Complexité cachée

Certaines opérations peuvent avoir une complexité plus grande qu'il n'y paraît au premier abord, il faudra donc faire attention à bien les prendre en compte lorsque vous calculez la complexité d'une implémentation. Par exemple en python, les opérations suivantes sont plus coûteuses que leur brièveté pourrait le suggérer :

```
if c in S:  
    pass
```

La partie `c in S` est en $\mathcal{O}(\text{len}(S))$ quand `S` est un tableau. Si `S` est un dictionnaire, c'est à dire une table de hachage, l'opération sera en revanche en $\mathcal{O}(1)$.

Il faut faire attention au type des données pour calculer la complexité. Les opérations n'ont pas le même coût en fonction de la structure de données utilisée.

2

DIVISER POUR RÉGNER

Un des algorithmes de tri les plus connus et les plus simples est le tri fusion, celui-ci se base sur l'idée que lorsque l'on a deux tableaux déjà triés il est relativement facile de les fusionner pour obtenir un grand tableau trié. Cette technique qui consiste à subdiviser les données récursivement en petits paquets puis à les recombiner est appelée *diviser pour régner*, tout une classe d'algorithmes est basée sur cette technique. Nous allons voir ici quelques algorithmes de ce type et comment calculer leur complexité.

1 Tri Fusion

Commençons par un court rappel du fonctionnement du tri fusion. Ce tri est basé sur le fait que fusionner deux tableaux déjà triés peut se faire en temps linéaire, en voici une implémentation :

```
def fusion(tab1,tab2):
    n1,n2 = len(tab1), len(tab2)
    tab = []
    i1, i2 = 0, 0
    while len(tab)<n1+n2:
        if i1 >= n1:
            tab.append(tab2[i2])
            i2 += 1
        elif i2 >= n2:
            tab.append(tab1[i1])
            i1 += 1
        elif tab1[i1] <= tab2[i2]:
            tab.append(tab1[i1])
            i1 += 1
        else:
            tab.append(tab2[i2])
            i2 += 1
```

```

    return tab

def tri_fusion(tableau):
    n = len(tableau)
    if(n<=1):
        return tableau

    gauche = tri_fusion(tableau[:n//2])
    droite = tri_fusion(tableau[n//2:])
    return fusion(gauche,droite)

```

Ici la complexité de la fonction `fusion` est linéaire en la taille des tableaux `gauche` et `droite` : si `gauche` est de taille n_1 et `droite` est de taille n_2 , alors `fusion` a une complexité en $\mathcal{O}(n_1 + n_2)$.

La complexité de la fonction `tri_fusion` quand à elle peut être exprimée de manière récursive, notons-la $T(n)$:

$$T(2n) = \underbrace{T(n)}_{\text{tri_fusion de la première moitié}} + \underbrace{T(n)}_{\text{tri_fusion de la deuxième moitié}} + \underbrace{\mathcal{O}(n)}_{\text{fusion des deux moitiés}}$$

avec $T(1) = \mathcal{O}(1)$.

On peut alors en déduire une expression close pour $T(n)$, on supposera¹ que $n = 2^k$ (et donc que $k = \log n$) :

$$\begin{aligned}
T(2^k) &= 2 * T(2^{k-1}) + C * 2^k \\
&= 2^2 * T(2^{k-2}) + C * (2^k + 2 * 2^{k-1}) \\
&= 2^3 * T(2^{k-3}) + C * (2^k + 2 * 2^{k-1} + 2^2 * 2^{k-2}) * C \\
&= \dots \\
&= 2^k * T(1) + C * \underbrace{(2^k + 2 * 2^{k-1} + 2^2 * 2^{k-2} + \dots + 2^k * 2^0)}_{k * 2^k} \\
&= 2^k * (T(1) + k) \\
&= n * (T(1) + \log n) \\
&= \mathcal{O}(n \log n)
\end{aligned}$$

2 Recherche Dichotomique

Lorsqu'un tableau d'entiers est déjà trié, on peut appliquer l'algorithme naïf pour y chercher un élément : on le parcourt du début à la fin et on s'arrête si on le trouve ou bien si l'on a atteint la fin. Cependant, le fait que le tableau soit trié nous permet de chercher l'élément bien plus efficacement.

La première chose que l'on peut faire est de procéder par élimination en regardant l'élément central du tableau : si l'élément recherché est plus grand, on sait qu'il se

¹Notre but étant d'obtenir une expression asymptotique, ce n'est pas une supposition gênante.

trouvera forcément dans le demi tableau de droite, s'il est plus petit dans celui de gauche. En itérant l'élimination, on obtient l'algorithme de recherche dichotomique :

```
def recherche(tableau,element):
    def aux(debut,fin):
        if fin - debut <= 0:
            return False
        m = debut + (fin - debut)//2
        if tableau[m] == element:
            return m
        if fin-debut <=1:
            return False
        elif tableau[m] < element:
            return aux(m,fin)
        elif tableau[m] > element:
            return aux(debut, m)
    return aux(0,len(tableau))
```

La complexité de cet algorithme est en $\mathcal{O}(\log n)$ car chaque à chaque appel récursif, s'il restait m éléments à considérer on en élimine $m/2$, dit autrement, $\text{fin}-\text{debut}$ est divisé par deux. Il y a donc au plus $\log_2 n$ appels à `aux`.

3 Master Theorem

De manière plus générale, étant donné une formule de récurrence pour une complexité $T(n)$, il existe un théorème permettant d'obtenir directement la formule close pour la complexité :

Théorème 2.1

Master Theorem

Lorsque l'on a une formule de récurrence de la forme :

$$T(n) = aT\left(\frac{n}{b}\right) + \mathcal{O}(n^d),$$

en posant $c = \log_b a$ la formule close correspondante prend plusieurs formes selon les cas :

1. si $d > c$ alors

$$T(n) = \mathcal{O}(n^d),$$

2. si $d = c$ alors

$$T(n) = \mathcal{O}(n^c \log n),$$

3. si $d < c$ alors

$$T(n) = \mathcal{O}(n^c).$$

Notons qu'il existe de multiples versions du master theorem, dont certaines plus générales.

On peut appliquer le Master Theorem aux deux problèmes précédents en écrivant leurs relations de récurrence :

- Tri Fusion : $T(n) = 2T(n/2) + \mathcal{O}(n)$, on a donc $a = 2$, $b = 2$ et donc $c = 1$, on se trouve dans le deuxième cas et $T(n) = \mathcal{O}(n \log n)$.
- Recherche dichotomique : $T(n) = T(n/2) + \mathcal{O}(1)$, on a donc $a = 1$, $b = 2$ et $c = \log_2 1 = 0 = d$ et on est donc dans le deuxième cas avec $T(n) = \mathcal{O}(\log n)$.

4 Multiplication Matricielle

Étant donné deux matrices carrées A et B de taille $n \times n$, on rappelle que :

- leur somme $C = A + B$ est une matrice carrée de taille $n \times n$ où

$$C_{i,j} = A_{i,j} + B_{i,j}$$

- leur produit $C = AB$ est une troisième matrice carrée de taille $n \times n$ où

$$C_{i,j} = \sum_k A_{i,k}B_{k,j}.$$

leur somme est calculable trivialement en $\mathcal{O}(n^2)$ en appliquant directement la formule naïvement :

```
def matadd(A,B):
    n = len(A)
    C = [[0]*n for i in range(n)]
    for i in range(n):
        for j in range(n):
            C[i][j] = A[i][j] + B[i][j]
    return C
```

Le calcul du produit est un peu plus intéressant comme nous allons le voir.

4.1 Algorithme Naïf

On peut calculer naïvement ce produit en appliquant directement la formule :

```
def matmul_naif(A,B):
    n = len(A)
    C = [[0]*n for i in range(n)]
    for i in range(n):
        for j in range(n):
            for k in range(n):
                C[i][j] += A[i][k]*B[k][j]
    return C
```

l'algorithme de multiplication de matrices est alors en $\mathcal{O}(n^3)$.

4.2 Diviser pour régner sans astuce

On peut également essayer d'utiliser le paradigme diviser pour régner afin d'obtenir un algorithme plus efficace. On peut commencer par remarquer que chacune des matrices A et B peut être subdivisée en 4 sous-matrices :

$$A = \begin{pmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{pmatrix} \quad B = \begin{pmatrix} B_{1,1} & B_{1,2} \\ B_{2,1} & B_{2,2} \end{pmatrix}$$

où chaque sous-matrice est alors de taille $(n/2) \times (n/2)$, on peut alors exprimer le produit AB en fonction de ces sous-matrices exactement comme si c'étaient juste des coefficients² :

$$AB = \begin{pmatrix} A_{1,1}B_{1,1} + A_{1,2}B_{2,1} & A_{1,1}B_{2,1} + A_{1,2}B_{2,2} \\ A_{2,1}B_{1,1} + A_{2,2}B_{2,1} & A_{2,1}B_{2,1} + A_{2,2}B_{2,2} \end{pmatrix}$$

On peut alors itérer le processus jusqu'à atteindre une taille de matrice de 1. La formule de récursion obtenue en utilisant cet algorithme est la suivante :

$$T(n) = 8T(n/2) + \mathcal{O}(n^2),$$

en appliquant le Master Theorem, on obtient alors que $c = \log_2 8 = 3$ et l'on est dans le troisième cas $T(n) = \mathcal{O}(n^c) = \mathcal{O}(n^3)$.

On n'a donc rien gagné par rapport à l'algorithme naïf, voire même les constantes cachées derrière le $\mathcal{O}()$ nous ont fait perdre du temps.

4.3 Diviser pour régner avec astuce (Strassen)

Cependant, tout n'est pas perdu Volker Strassen en 1969 remarque que l'on peut exprimer les produits partiels qui forment l'étape de diviser pour régner à l'aide de seulement 7 sous-produits et non pas 8 en définissant les produits suivants :

$$\begin{aligned} M_1 &= (A_{1,1} + A_{2,2})(B_{1,1} + B_{2,2}) \\ M_2 &= (A_{2,1} + A_{2,2})B_{1,1} \\ M_3 &= A_{1,1}(B_{1,2} - B_{2,2}) \\ M_4 &= A_{2,2}(B_{2,1} - B_{1,1}) \\ M_5 &= (A_{1,1} + A_{1,2})B_{2,2} \\ M_6 &= (A_{2,1} - A_{1,1})(B_{1,1} + B_{1,2}) \\ M_7 &= (A_{1,2} - A_{2,2})(B_{2,1} + B_{2,2}) \end{aligned}$$

On peut alors exprimer AB de la manière suivante :

$$AB = \begin{pmatrix} M_1 + M_4 - M_5 + M_7 & M_3 + M_5 \\ M_2 + M_4 & M_1 - M_2 + M_3 + M_6 \end{pmatrix}$$

Comme nous n'avons plus que 7 sous produits, l'équation devient alors $T(n) = 7T(n/2) + \mathcal{O}(n^2)$ en appliquant le Master Theorem, on obtient $c = \log_2 7 > 2$, on a alors $T(n) = \mathcal{O}(n^{\log_2 7})$, ce qui nous donne une complexité inférieure à l'algorithme naïf!

²La preuve est laissée en exercice de L1.

5 Enveloppe Convexe

Le prochain exemple d'algorithme utilisant diviser pour régner que nous allons voir est un algorithme permettant de calculer l'*enveloppe convexe* d'un ensemble de points de \mathbb{R}^2 .

Définition 2.2

Un ensemble $A \subseteq \mathbb{R}^n$ est dit *convexe* quand étant donné $x, y \in \mathbb{R}^2$,

$$x, y \in A \Rightarrow \forall t \in [0, 1], tx + (1 - t)y \in A.$$

C'est à dire quand le segment $[x, y]$ est entièrement dans A .

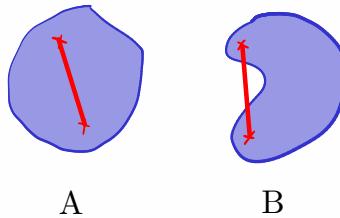


FIG. 2.1 : L'ensemble A est convexe, car pour tous les couples de points de l'ensemble que l'on peut choisir, le segment les reliant est dedans. L'ensemble B ne l'est pas le segment reliant les deux points de la figure n'y étant pas intégralement.

L'*enveloppe convexe* d'un ensemble A peut alors être définie comme étant le plus petit ensemble convexe qui contient A . Dans cette partie, on s'intéressera toujours à l'enveloppe convexe d'un ensemble fini de points : celle-ci sera toujours un polygone dont les coins sont des points de l'ensemble.

5.1 Marche de Jarvis

L'algorithme le plus intuitif pour calculer l'enveloppe convexe d'un ensemble de points est probablement la *marche de Jarvis*. Le principe est simple, on commence par prendre un point à une des extrémités, par exemple celui qui a la plus petite abscisse (et la plus grande ordonnée s'il y en a plusieurs). On parcourt ensuite sur les points restants et l'on prend le point qui maximise l'angle avec le segment le dernier segment que l'on vient de trouver jusqu'à retomber sur le premier point.

```
def marche_jarvis(points):
    if len(points) <= 3: # un triangle est sa propre enveloppe convexe
        return points
    x0, y0 = points[0]
```

```

for x,y in points:
    if x < x0:
        x0,y0 = x,y
enveloppe_convexe = [(x0,y0)]
xprev,yprev = x0,y0
while True:
    xmax,ymax = points[0]
    for i,(x,y) in enumerate(points):
        if (x,y) == (xprev,yprev):
            continue
        diff_pente = (y - yprev)*(xmax - xprev) - (ymax - yprev)*(x - xprev)
        if diff_pente >= 0:
            xmax,ymax = x,y

    enveloppe_convexe.append( (xmax,ymax) )
    if (xmax,ymax) == (x0,y0):
        break
    xprev,yprev = xmax,ymax
return enveloppe_convexe

```

La marche de Jarvis est en temps $\mathcal{O}(n^2)$, le pire des cas étant atteint lorsque tous les points sont dans l'enveloppe convexe.

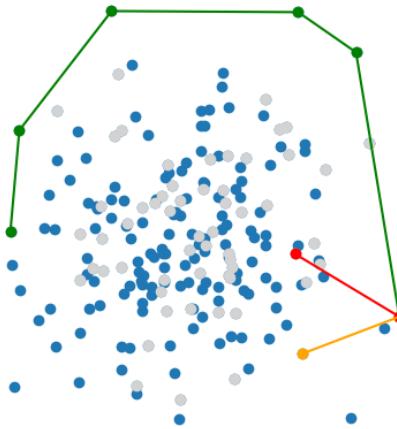


FIG. 2.2 : Une étape de la marche de Jarvis : en vert les éléments déjà sélectionnés comme faisant partie de l'enveloppe convexe, en orange le candidat actuel à l'ajout, en gris les points déjà considérés, en bleus ceux qui ne l'ont pas encore été et en rouge le segment en cours de comparaison : dans ce cas précis, il ne sera pas conservé car l'angle est supérieur à celui du segment orange.

5.2 QuickHull

L'algorithme QuickHull est un algorithme qui utilise le principe de diviser pour régner pour calculer l'enveloppe convexe. Le principe est de trouver le point le plus à gauche p_g et le point le plus à droite p_d . Il est garanti que ces éléments seront dans l'enveloppe convexe. Une fois ces éléments trouvés, ils permettent de diviser le plan en deux parties. Ensuite, de chaque côté du segment, on cherche le point p_l qui en est le plus éloigné, celui-ci sera également dans l'enveloppe convexe. Le triangle formé par ces trois points permet alors de diviser les points restants en trois sous-ensembles :

- Les points à l'intérieur du triangle, ils ne peuvent pas être dans l'enveloppe convexe.
- Les points à gauche du segment $[p_g, p_l]$: on y cherche le point le plus éloigné de $[p_g, p_l]$ qui nous permet d'obtenir un nouveau triangle sur lequel on recommence récursivement la procédure, ce traitement nous donne une liste L_g de sommets faisant partie de l'enveloppe convexe.
- Les points à droite du segment $[p_l, p_d]$, que l'on traitera récursivement de la même manière, ce qui nous donnera une liste L_d .

On peut alors fusionner les listes L_g et L_d en y intercalant les points aux bons endroits.

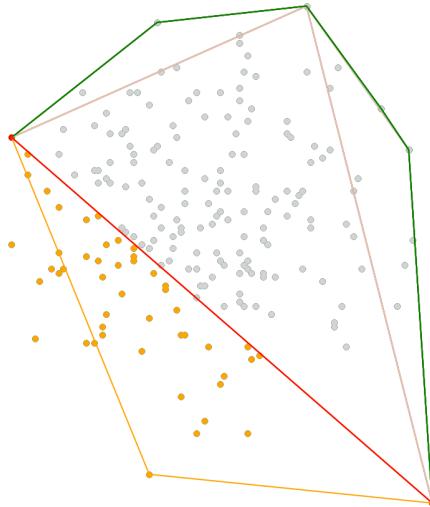


FIG. 2.3 : Lors de cette étape de QuickHull, les segments verts ont déjà été sélectionnés comme faisant partie de l'enveloppe convexe, les gris correspondent à des étapes précédentes et n'ont pas été retenus. Le segment rouge est le segment en cours de considération et les segments orange seront les prochains à l'être : ils relient le point le plus éloigné du segment rouge avec les extrémités de celui-ci.

On peut commencer par écrire une fonction `separer(p1, p2, pts)` qui permet

de séparer un ensemble de points en deux sous ensembles gauche et droite qui contiennent les points à gauche et à droite du segment défini par $[p_1, p_2]$. Elle renverra également les points de chacun de ces ensembles qui sont les plus éloignés du segment car cela peut être fait sans aucun coût supplémentaire. Cette fonction a une complexité en $\mathcal{O}(n)$.

```
def separer(p1,p2,pts):
    gauche, droite = [], []
    (x1,y1), (x2,y2) = p1, p2
    gmaxdist, dmaxdist = 0, 0
    gmax, dmax = p1, p1

    for x,y in pts:
        # on compare les pentes des segments [p1,(x,y)] et [p1,p2]
        # pour obtenir de quel côté se trouve (x,y)
        diff_pente = (y - y1) * (x2 - x1) - (y2 - y1) * (x - x1)
        # la valeur absolue correspond à la distance de (x,y) au
        # segment [p1,p2]
        distance = abs(diff_pente)
        if diff_pente >0:
            gauche.append((x,y))
            if distance > gmaxdist:
                gmaxdist = distance
                gmax = (x,y)
        elif diff_pente < 0:
            droite.append((x,y))
            if distance > dmaxdist:
                dmaxdist = distance
                dmax = (x,y)
    return (gmax,gauche), (dmax,droite)
```

On peut maintenant écrire notre algorithme dans la fonction `quickhull(points)`, qui assurera l'étape initiale de séparation de l'espace de points en deux ensembles séparés par le segment entre le point le plus à gauche et le point le plus à droite. Le reste du traitement sera assuré par la fonction auxiliaire `enveloppe(p1,p2,pts)`) qui se chargera de la partie récursive étant donné un segment et des points, ne garder que les points à la gauche du segment, chercher le point parmi eux le plus éloigné puis recommencer récursivement avec les deux segments nouveaux segments du triangle formé en ajoutant ce point au segment de départ.

Toutes les détections d'ensembles à gauche ou à droite sont faites par la fonction `separer`.

```

def quickhull(points):
    def enveloppe(p1,p2,pts):
        if not pts:
            return []
        (gmax,gauge), (dmax,droite) = separer(p1,p2,pts)

        env_g = enveloppe(p1,gmax,gauge)
        env_d = enveloppe(gmax,p2,gauge)

        return env_g + [gmax] + env_d

    if len(points) <=3:
        return points

    (xg,yg), (xd,yd) = points[0], points[0]
    for x,y in points:
        if x < xg:
            xg,yg = x,y
        if x > xd:
            xd,yd = x,y

    (_, gauge), (_, droite) = separer((xg,yg), (xd,yd), points)
    env_g = enveloppe((xg,yg),(xd,yd), gauge)
    env_d = enveloppe((xd,yd),(xg,yg), droite)
    enveloppe = [(xg,yg)] + env_g + [(xd,yd)] + env_d + [(xg,yg)]
    return enveloppe

```

► Complexité

Dans le pire des cas, la séparation des $n - 2$ points fait un premier ensemble contenant un unique élément et un second avec $n - 3$ et l'algorithme fonctionne en $\mathcal{O}(n^2)$. En revanche si les points sont bien répartis dans l'espace³, l'algorithme sépare récursivement en ensembles qui font à peu près la même taille, si l'on se trouve dans cette situation, la relation de récurrence serait alors $T(n) = 2T(n/2) + \mathcal{O}(n)$ et le Master Theorem nous donne une complexité en $\mathcal{O}(n \log n)$.

³Par exemple selon une distribution normale. La complexité d'un algorithme en prenant en compte une distribution de probabilité pour les données en entrée est appelée complexité en moyenne. Vous verrez lors dans certains autres cours cette notion, elle est par exemple abordée dans le cours d'option *Algorithmique, Structures informatiques et Cryptologie*.

3

GRAPHES ET PARCOURS

1 Graphe

Les graphes sont un moyen de représenter des données et les relations qu'elles ont entre elles. Un graphe est donné par un ensemble de sommets et d'arêtes qui les relient.

Définition 3.1 Graphe

Un graphe est un couple (V, E) où V un ensemble de *sommets* et $E \subseteq V \times V$ un ensemble d'*arêtes*.

Un graphe est dit *non-orienté* quand pour toute arête $(i, j) \in E$, on a également $(j, i) \in E$. Par défaut, on considérera qu'un graphe est orienté.

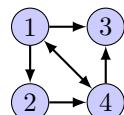
On représente en général les sommets par des ronds et les arêtes par des flèches les reliant :

Exemple 3.2. Le graphe $G_1 = (V, E)$ où

$V = \{1, 2, 3, 4\}$ et

$E = \{(1, 2), (1, 3), (1, 4), (2, 4), (4, 1), (4, 3)\}$

peut être représenté ainsi :



Vous rencontrez les graphes tous les jours, parfois explicitement comme par exemple pour les plans de transports en commun (voir [Figure 3.1](#)), parfois implicitement, comme par exemple pour les liens entre pages web (voir [Figure 3.2](#)).

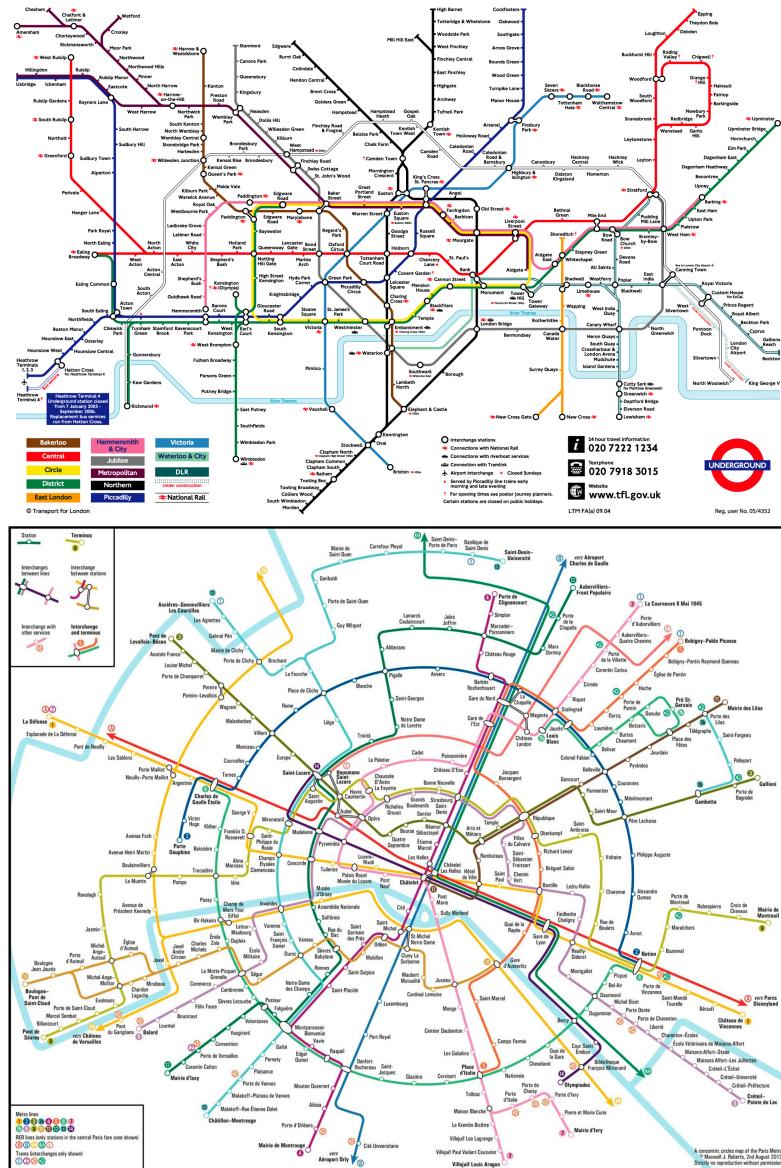


FIG. 3.1 : Le plan du métro est représenté sous forme de graphe : les sommets correspondent aux arrêts de métro et les arêtes sont les lignes les reliant.

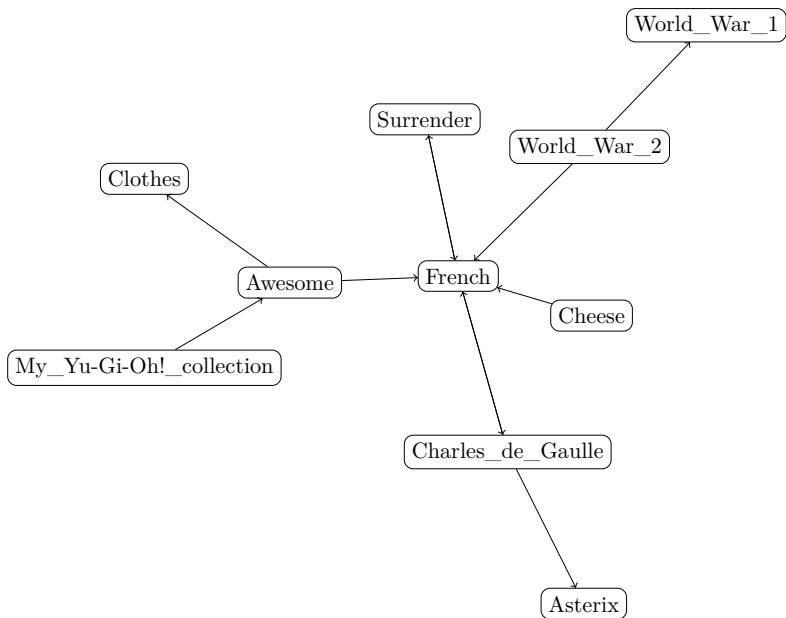


FIG. 3.2 : Un graphe des liens entre certaines des pages de [Uncyclopedia](#) : il y a une arête entre deux pages s'il existe un lien de l'une vers l'autre.

2 Stockage des graphes

Plusieurs opérations nous intéresseront sur les graphes :

- `arete(i, j)` : existe-t-il une arête du sommet i au sommet j ?
- `voisins(i)` : quels sont les voisins du sommet i (les sommets vers lesquels il y a une arête) ?
- `ajouter_arete(i, j)` : ajouter une arête de i vers j .
- `ajouter_sommet(i)` : ajouter un sommet i .
- `enlever_arete(i, j)` : enlever l'arête de i vers j .
- `enlever_sommet(i)` : enlever le sommet i .

Il existe plusieurs structures de données permettant de représenter des graphes, selon la représentation ces opérations n'auront pas la même complexité.

2.1 Matrice d'adjacence

Une des façons de stocker un graphe est sous la forme d'une matrice d'adjacence. Étant donné un ensemble de sommets V , on peut supposer sans perte de généralité qu'ils sont numérotés de 1 à $n = |V|$. On peut alors représenter sous la forme d'une matrice M le fait qu'il existe ou non une arête entre deux sommets : $M_{i,j} = 1$ s'il y a une arête de i vers j et $M_{i,j} = 0$ sinon. On appelle cette matrice, la *matrice d'adjacence*

Exemple 3.3. Si l'on reprend le graphe G_1 de l'[Exemple 3.2](#), alors sa matrice d'adjacence

est la suivante :

$$\begin{pmatrix} 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 \end{pmatrix}$$

En pratique, la matrice d'adjacence sera stockée sous la forme d'un tableau biddenionnel. Certaines opérations seront alors faciles (en $\mathcal{O}(1)$) comme vérifier si deux sommets sont voisins ou encore ajouter une arête. D'autres seront en revanche moins rapides, comme obtenir la liste des voisins (en $\mathcal{O}(|V|)$).

2.2 Listes d'adjacence

Un graphe peut également être représenté sous la forme de listes d'adjacence : pour chaque sommet, on stocke une liste des sommets vers lesquels il a une arête.

Exemple 3.4. Toujours pour le graphe G_1 de l'[Exemple 3.2](#), les listes d'adjacence seront les suivantes :

```
l[0] = [1,2,3]
l[1] = [3]
l[2] = []
l[3] = [0,1]
```

Dans le cas des listes d'adjacence, lister les voisins d'un sommet est facile, il suffit de parcourir sa liste d'adjacence. En revanche si l'on souhaite juste savoir s'il existe une arête entre deux sommets, la complexité ne sera plus en $\mathcal{O}(1)$ car il faudra parcourir la liste.

2.3 Tables de hachage

Il existe un bon compromis pour stocker les graphes de manière à ce que la plupart des opérations soient relativement efficaces, c'est de les stocker dans des tables de hachage.

2.3.1 ▶ Tables de hachage

Les tables de hachage ont une complexité dans le pire des cas qui n'est pas intéressante, mais lorsque l'on fait de nombreuses opérations le coût est *amorti*, c'est à dire que si l'on fait la moyenne des coûts sur toutes ces opérations ce coût amorti sera bien plus bas que dans le pire des cas. Ici n est le nombre d'éléments dans la table de hachage.

Opération	Coût amorti	Coût pire des cas
Insertion	$\Theta(1)$	$\mathcal{O}(n)$
Suppression	$\Theta(1)$	$\mathcal{O}(n)$
Parcours	$\Theta(n)$	$\mathcal{O}(n)$

2.3.2 ▶ Graphes dans des tables

Stocker un graphe avec des tables de hachage est relativement simple. Il suffit pour chaque sommet x d'avoir une table de hachage dans laquelle on a une clef y pour chaque arête (x, y) .

Exemple 3.5. Toujours pour le graphe G_1 de l'[Exemple 3.2](#), on pourra le représenter de la façon suivante :

```
G1 = {
    0 : {
        1 : True,
        2 : True,
        3 : True,
    },
    1 : {
        3 : True,
    },
    2 : {},
    3 : {
        0 : True,
        1 : True,
    },
}
```

2.4 Tableau comparatif

Voici un tableau qui compare les différentes opérations en fonction de la structure de données sous-jacente. On note n le nombre de sommets du graphe et m le nombre de voisins du sommet i .

	Matrice	Listes d'adjacence	Tables de hachage
arete(i,j)	$\mathcal{O}(1)$	$\mathcal{O}(m)$	$\Theta(1)^*$
voisins(i)	$\Theta(n)$	$\mathcal{O}(m)$	$\Theta(m)^*$
ajouter_arete(i,j)	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\Theta(1)^*$
ajouter_sommet(i)	$\mathcal{O}(n^2)^{**}$	$\mathcal{O}(1)$	$\Theta(1)^*$
enlever_arete(i,j)	$\mathcal{O}(1)$	$\mathcal{O}(m)$	$\Theta(1)^*$
enlever_sommet(i,j)	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$	$\Theta(n)^*$

3 Parcours en largeur

Peut-on rejoindre un sommet v en partant du sommet u ? En traversant combien d'arêtes? C'est à ces questions que l'on va pouvoir répondre en faisant un parcours en largeur.

L'idée du *parcours en largeur* est de commencer par explorer un sommet s puis d'explorer ses voisins directs, ensuite les voisins de ses voisins, etc... sans jamais repasser par un sommet que l'on a déjà vu. Cela revient à parcourir le graphe en cercles concentriques centrés en s .

Pour cela on utilise une *file*, une liste chaînée avec deux opérations :

- **enfiler** un élément, c'est à dire le mettre à la fin de la liste,
- **depiler** un élément, c'est à dire prendre le premier élément de la liste.

Au début cette file contiendra uniquement le sommet de départ, puis tant qu'elle contiendra des éléments on dépilera le premier et on mettra à jour sa distance puis on empilera ses voisins :



```

Entrées :  $G = (V, E)$  un graphe et  $s$  un sommet de départ
Sorties : Un dictionnaire contenant les distances à  $s$  des noeuds accessibles à
partir de  $s$ 
pour chaque  $v \in V$  faire
| dist[ $v$ ]  $\leftarrow \infty$ ;
fin
dist[ $s$ ]  $\leftarrow 0$ ;
 $F \leftarrow \text{File}(s)$  /*  $F$  est une file contenant juste  $s$ . */ *
tant que  $F$  non-vide faire
| /* visite d'un élément */
|  $v \leftarrow F.\text{depiler}()$  /* on prend le premier élément de  $F$ . */
| pour chaque  $(v, w) \in E$  faire
| | dw = dist[ $v$ ] + 1;
| | si dw < dist[ $w$ ] alors
| | | dist[ $w$ ]  $\leftarrow$  dw;
| | |  $F.\text{enfiler}(w)$ 
| | fin
| fin
fin
retourner dist

```

Algorithme 1 : Parcours en largeur

3.1 Complexité

Le parcours en largeur considère au plus chaque arête une seule fois : en effet dans la file si un sommet v est sorti avant un sommet w , alors $\text{dist}[v] < \text{dist}[w]$ donc un sommet exploré après un autre ne peut pas diminuer une distance déjà existante (autre que ∞). Chaque sommet est donc enfilé une fois au plus et les arêtes en sortant sont donc considérées une unique fois. La complexité du parcours en largeur est donc de $\mathcal{O}(|V| + |E|)$.

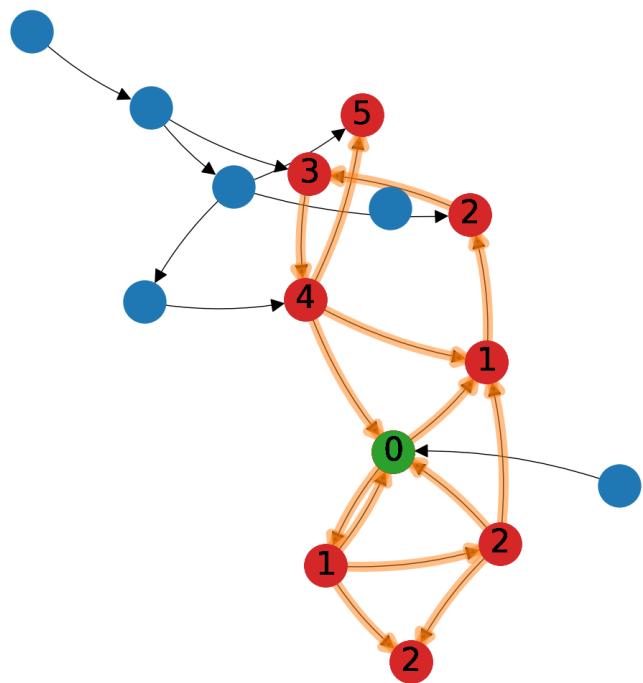


FIG. 3.3 : Un parcours en largeur en commençant par le sommet vert. Les sommets contiennent leur distance à ce sommet. Les sommets bleus sont des sommets vers lesquels il n'y a pas de chemin, les sommets inaccessibles.

4 Parcours en profondeur

Le parcours en largeur d'un graphe explore dans l'ordre d'abord les voisins, puis les voisins des voisins et ainsi de suite. Une autre manière de parcourir un graphe est de suivre des arêtes jusqu'à arriver à un cul de sac (un sommet dont tous les voisins ont déjà été visités), puis de reprendre au dernier embranchement et de recommencer. Le plus simple est de faire cela récursivement, comme pour un parcours d'arbre :

```

Entrées :  $G = (V, E)$  un graphe et  $s$  un sommet de départ
Sorties : Un dictionnaire contenant les sommets accessibles à partir de  $s$ 
Fonction ParcoursProfondeur( $G, s$ ) :
    visité  $\leftarrow \{\}$ ;
    pour chaque  $v \in V$  faire
        | visité[ $v$ ]  $\leftarrow$  Faux;
    fin
    Fonction Explorer( $v$ ) :
        | visité[ $v$ ]  $\leftarrow$  Vrai;
        | pour chaque  $w$  tel que  $(v, w) \in E$  faire
            |   | si visité[ $w$ ] = Faux alors
            |   |   | Explorer( $w$ )
            |   | fin
        | fin
    Explorer( $s$ );
    retourner visité

```

Algorithme 2 : Parcours en profondeur : exploration récursive

4.1 Temps de parcours

Les appels récursifs qui ont été faits pour notre phase d'exploration définissent un *arbre de parcours*. Pour reconstruire cet arbre on peut s'aider de temps de parcours, pour cela on va rajouter deux temps pour chaque sommet : quand a commencé sa visite (stocké dans `pre`) et quand on l'a finie (stocké dans `post`).

Entrées : $G = (V, E)$ un graphe et s un sommet de départ
Sorties : Un dictionnaire contenant les sommets accessibles à partir de s

Fonction ParcoursProfondeur(G, s) :

```

visité ← {};
post ← {};
pre ← {};
temps ← 0;
Fonction Explorer( $v$ ) :
    pre[ $v$ ] ← temps++;
    visité[ $v$ ] ← Vrai;
    pour chaque  $w$  tel que  $(v, w) \in E$  faire
        si visité[ $w$ ] = Faux alors
            |   Explorer( $w$ )
        fin
    fin
    post[ $v$ ] ← temps++;
    pour chaque  $v \in V$  faire
        visité[ $v$ ] ← Faux;
    fin
    Explorer( $s$ );
    retourner visité

```

Algorithme 3 : Parcours en profondeur : exploration récursive

On peut définir l'intervalle de visite de v à l'aide de ces temps : $[pre[v], post[v]]$. Les intervalles de visite de deux sommets u et v sont soit disjoints, soit l'un des deux intervalles est inclus dans l'autre. Dans ce dernier cas, le sommet dont l'intervalle est inclus est un descendant de l'autre sommet dans l'arbre de parcours.

4.2 Complexité

Chaque sommet est marqué comme visité lors de la fonction Explore, et cette fonction n'est appelée que sur les sommets qui n'ont pas encore été visités. La fonction Explore est donc appelée au plus $\mathcal{O}(|V|)$ fois et elle itère sur les arêtes sortant du sommet sur lequel elle a été appelée. On considère donc chaque arête une seule fois. La complexité est donc en $\mathcal{O}(|V| + |E|)$

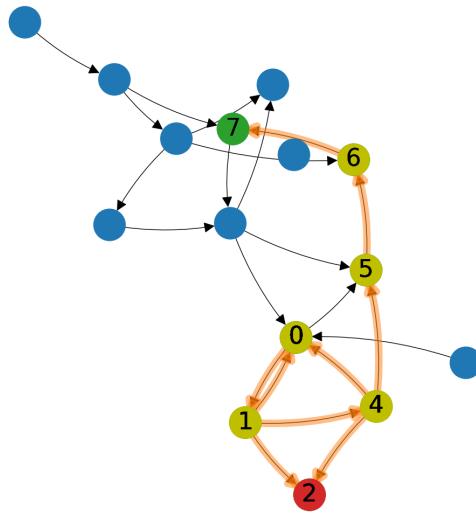


FIG. 3.4 : Un exemple d'étape intermédiaire du parcours en profondeur : en vert le sommet en cours de considération, en jaune les sommets dont le parcours a commencé mais n'est pas encore terminé, en rouge ceux dont le parcours est fini et en bleus les sommets non explorés.

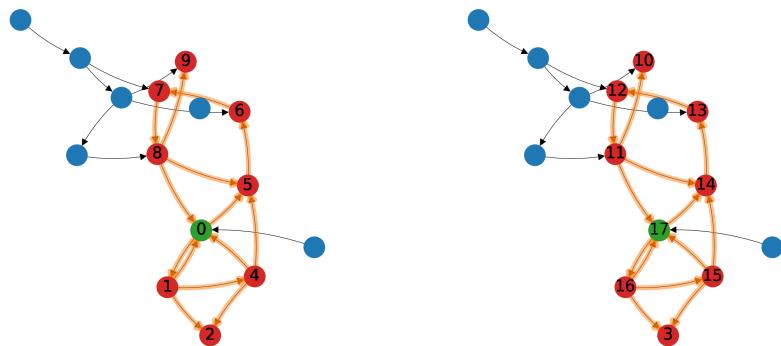


FIG. 3.5 : Une fois le parcours en profondeur fini, on a exploré tous les sommets accessibles, en rouge, à partir du sommet de départ, en vert. Sur la gauche les temps de début de visite sont inscrits sur les sommets et à droite les temps de fin de visite.

4.3 Version itérative

On peut transformer ce parcours récursif en un parcours itératif en utilisant quasi-méme algorithme que pour le parcours en largeur mais en utilisant une *pile*, une liste chaînée avec deux opérations :

- **empiler** un élément, c'est à dire le mettre au début de la liste,
- **depiler** un élément, c'est à dire prendre le premier élément de la liste.

Ainsi maintenant ce sera le dernier élément inséré qui sera le premier à ressortir.

```

Entrées :  $G = (V, E)$  un graphe et  $s$  un sommet de départ
Sorties : Un dictionnaire contenant les sommets accessibles à partir de  $s$ 
visité  $\leftarrow \{\}$ ;
pour chaque  $v \in V$  faire
|   visité[ $v$ ]  $\leftarrow$  Faux;
fin
 $F \leftarrow$  Pile( $s$ ) /*  $F$  est une pile contenant juste  $s$ . */
tant que  $F$  non-vide faire
|   /* visite d'un élément
|    $v \leftarrow F.\text{depiler}()$  /* on prend le premier élément de  $F$ .
|   si visité[ $v$ ] = Faux alors
|   |   visité[ $v$ ]  $\leftarrow$  Vrai;
|   |   pour chaque  $(v, w) \in E$  faire
|   |   |   si visité[ $w$ ] = Faux alors
|   |   |   |    $F.\text{empiler}(w)$  /* on met  $w$  au début de la pile. */
|   |   |   fin
|   |   fin
|   fin
fin
retourner visité

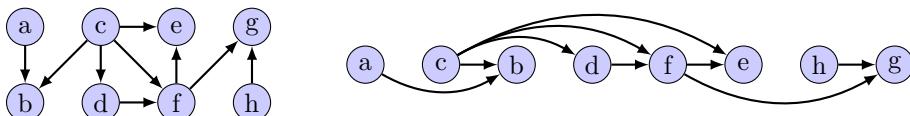
```

Algorithme 4 : Parcours en profondeur itératif.

5 Tri topologique

Lorsqu'un graphe orienté $G = (V, E)$ ne contient pas de cycle, on peut trouver un ordre \succ sur les sommets tel que $u \succ v$ uniquement s'il n'existe aucun chemin allant de v vers u . C'est à dire que l'on peut ordonner les sommets sur une ligne de manière à ce que toutes les arêtes soient orientées vers la droite.

Exemple 3.6. Étant donné le graphe ci-dessous à gauche, on peut voir un tri topologique à droite : toutes les arêtes pointent vers la droite et donc $a \succ c \succ b \succ d \succ f \succ e \succ h \succ g$.



Le tri topologique n'est pas unique ici : le sommet h aurait pu être placé n'importe où dans l'ordre sauf à la dernière position par exemple.

On peut montrer que lors du parcours en profondeur, si un sommet v est accessible à partir d'un sommet u alors on a nécessairement $\text{post}[v] < \text{post}[u]$. On peut donc obtenir un tri topologique des sommets en les ordonnant par post décroissant.

Attention, cette section n'est pas finie !

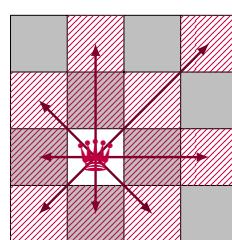
4

BACKTRACKING

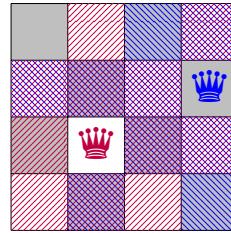
Le principe du backtracking est d'essayer toutes les solutions (en anglais “brute-force”) mais de manière un peu plus intelligente qu’en listant directement toutes les possibilités puis en les essayant. On va en quelque sorte faire du bourrinage presque intelligent. Pour illustrer le backtracking nous allons nous pencher sur le problème des n -reines dans un premier temps. Dans un second temps nous donnerons un algorithme pour résoudre le sudoku.

1 Problème des n -Reines

Sur un échiquier, on dit qu'une reine menace une case si celle-ci peut se déplacer dessus au prochain tour. Ainsi une reine menace les cases qui sont dans sur la même ligne, dans la même colonne ou dans une de ses diagonales :



On voit ici que l'on peut facilement placer une seconde reine sur cet échiquier 4×4 sans qu'elles ne se menacent mutuellement :



Le but est de réussir à mettre n reines dans une grille de taille $n \times n$.

1.1 Parcours des possibilités

1.1.1 ► Naïvement

Une première manière de faire serait de tester toutes les possibilités, par exemple avec un algorithme comme celui-ci :

```
def queens1(N):
    def recherche(positions):
        if len(positions)==N:
            return positions
        for i in range(N):
            for j in range(N):
                possible = True
                for ri,rj in positions:
                    if ((ri,rj) == (i,j)) # * il y a déjà une reine
                        or                                # en (i,j)
                    abs(ri-i) == abs(rj-j)# * il y a une reine sur
                        or                                # la diagonale
                    ri == i                            # * il y a une reine sur
                        or                                # la même ligne
                    rj == j):                          # * il y a une reine dans
                                                # la même colonne
                        possible = False
                if possible:
                    positions.append((i,j))
                    r = recherche(positions)
                    if r is not False:
                        return r
                    positions.pop() # on dépile la dernière possibilité
                                    # envisagée
    return False
return recherche([])
```

Cet algorithme teste toutes les possibilités pour chacune des n reines : pour placer la première reine, il peut la mettre sur n'importe quelle case (il y en a n^2), pour la seconde il peut essayer n'importe quelle case qui n'est pas menacée et ainsi de suite, dès qu'il n'y a plus aucune possibilité et que l'on a pas encore placé les n reines, on revient à notre dernier choix.

Malheureusement avec cette manière de faire la complexité est très élevée, on peut borner le nombre de choix de positions pour la i ème reine par n^2 et donc le temps mis

par l'algorithme :

$$T(n) \leq \underbrace{n^2}_{\substack{\text{vérification qu'une position est valide}}} \cdot \underbrace{\prod_{1 \leq i \leq n} n^2}_{\substack{\text{nombre de positions}}} \leq n^{2n+2}$$

C'est tout de même très élevé ! On peut toutefois s'autopersuader de l'efficacité de l'algorithme en se disant que l'algorithme s'arrête dès qu'il a trouvé une possibilité et qu'on s'est donc épargné un certain nombre d'étapes de calcul par rapport à tester tous les n -couples de coordonnées (i, j) représentant les n reines.

1.1.2 ► En prenant en compte les contraintes

En y regardant de plus près, on se rend compte que ce n'est pas optimal : chaque ligne d'une solution doit contenir exactement une reine par ligne et on peut alors écrire un algorithme où l'on n'a que $n - i$ choix pour la i ème reine :

```
def queens2(N):
    def recherche(positions):
        i = len(positions)
        if i==N:
            return positions
        for j in range(N):
            possible = True
            for ri,rj in enumerate(positions):
                if(
                    abs(ri-i) == abs(rj-j) # * il y a une reine sur
                or                                # la diagonale
                    rj == j):                      # * il y a une reine dans
                                            # la même colonne
                possible = False
            if possible:
                positions.append(j)
                r = recherche(positions)
                if r is not False:
                    return r
                positions.pop() # on dépile la dernière possibilité
                                 # envisagée
    return False
return recherche([])
```

La complexité de cet algorithme est plus facile à borner finement, dans le sens où à la i ème reine on a déjà occupé i lignes, mais aussi i colonnes :

$$T(n) \leq n^2 \cdot \prod_{1 \leq i \leq n} (n - i) = n^2 n! = o(n^{2n+2})$$

ce qui est un peu plus intéressant¹.

La raison pour laquelle l'autre algorithme est beaucoup moins efficace est qu'il regardera plusieurs fois les mêmes positions : pour la i ème reine, on considèrera toutes les lignes comme des possibilités et donc on pourra réexaminer plusieurs fois une même position avant d'arriver à une solution.

¹Souvenez-vous que $n! = o(n^n)$ et donc c'est aussi négligeable face à n^{2n} .

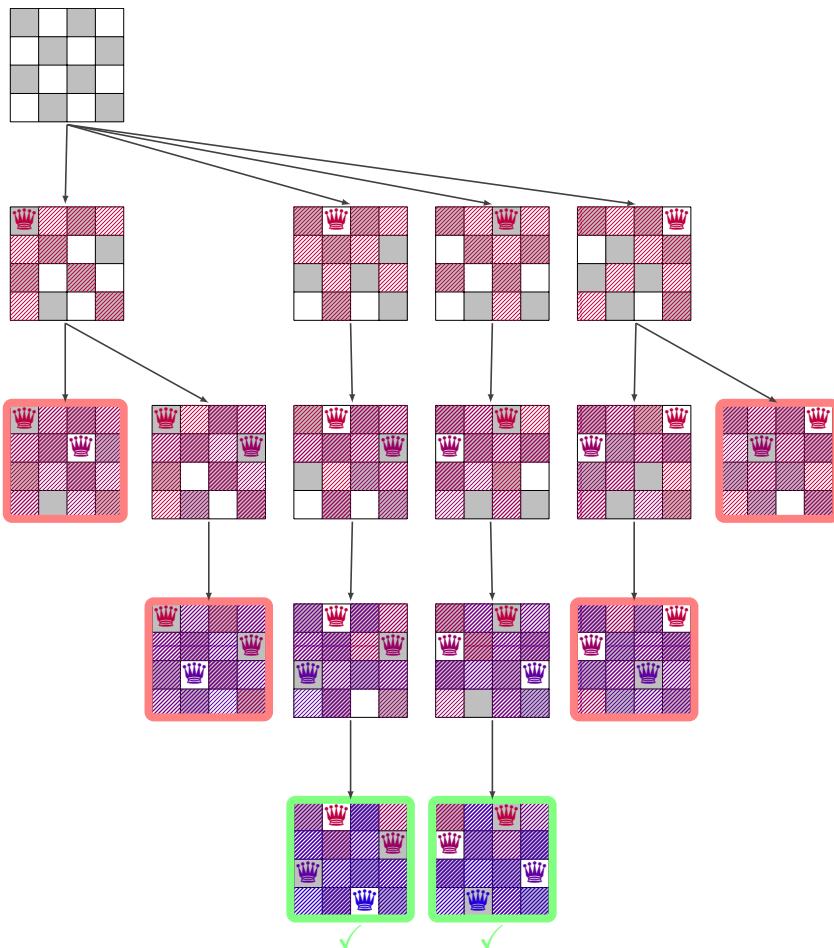


FIG. 4.1 : L’arbre de décision pour le problème des 4 reines, on utilise le fait que chaque ligne doit contenir une reine pour limiter le nombre de décisions.

1.1.3 ▶ Arbre des possibilités

Ce que nous avons fait implicitement dans ces algorithmes c'est un parcours en profondeur de l'arbre des positions possibles pour chaque reine. Sur la Figure 4.1 on peut voir l'algorithme queens2 déroulé pour le problème des 4 reines. On notera que dès qu'une contrainte ne peut pas être satisfaite avec les choix déjà effectués on cesse d'explorer son sous-arbre. Ceci permet de réduire les possibilités, on a ici 6 branches alors que $4! = 24$.

2 Sudoku

Un sudoku est une grille 9×9 remplie à l'aide de chiffres allant de 1 à 9, le but est de compléter les cases manquantes en respectant certaines contraintes :

- Chaque ligne doit contenir exactement une fois chaque valeur.

- Chaque colonne doit contenir exactement une fois chaque valeur.
- Les sous-carrés 3×3 doivent chacun contenir exactement une fois chaque valeur.

Voici un exemple de grille partiellement remplie avec les subdivisions en carrés 3×3 :

	4		5	3			8	
				9				
	1		8		7		5	
		2		7		5	3	9
	5				3		1	
			9					
	9				8	4		
6			3					8
5				1				

On propose de résoudre ce problème à l'aide de backtracking. Pour cela on va parcourir les cases une à une et choisir une des valeurs possibles avant de continuer, si on trouve une contradiction à un moment, on revient à la dernière case où l'on avait des choix :

```
def solve(self, i=0, j=0):
    i = i + j // 9
    j = j% 9
    if i>=9: # toutes les cases ont été remplies
        return self.sudoku
    if self.has_value(i, j):
        return self.solve(i, j+1)
    for val in range(1,10):
        if self.is_valid(i, j, val):
            self.set(i, j, val)
            if self.solve(i, j+1) is not False:
                return self.sudoku
            self.reset(i, j)
    return False
```

Ici, on explore une branche de l'arbre des possibilités uniquement si celle-ci ne provoque pas immédiatement une contradiction. On évite ainsi d'explorer tout le sous-arbre de possibilités qui était en dessous.

5

PROGRAMMATION DYNAMIQUE

La programmation dynamique est en quelque sorte le principe de mémoriser les résultats intermédiaires qui seront utilisés plusieurs fois. Nous pouvons illustrer ce principe à l'aide de la suite de Fibonacci.

La suite de Fibonacci est définie récursivement de la manière suivante :

$$\begin{aligned}F_0 &= 1 \\F_1 &= 1 \\F_{n+2} &= F_{n+1} + F_n\end{aligned}$$

Une manière de la calculer serait donc d'appliquer directement la définition pour obtenir un algorithme :

```
def FiboRec(n):
    if n<=1:
        return 1
    else:
        return FiboRec(n-1) + FiboRec(n-2)
```

Cet algorithme va faire de nombreux appels récursifs, comme on peut le voir sur la [Figure 5.1](#) : pour chaque terme F_i avec $i \leq n - 2$ le calcul aura lieu au minimum deux fois. Par exemple lorsque l'on appelle `FiboRec(n)`, les sous-arbres d'appels récursifs pour calculer `FiboRec(n-2)` sont effectués 2 fois, ceux pour $n - 4$ le sont au moins 4 fois et ainsi de suite. On peut en déduire qu'on a au moins $2^{\frac{n}{2}}$ appels récursifs pour un appel à `FiboRec(n)`. On peut également montrer qu'il y en a au plus 2^n : on a un arbre d'appels récursif de profondeur au plus n avec deux sous-arbres au plus à chaque noeud. Le calcul de F_n se fait donc en temps exponentiel avec cet algorithme¹.

¹Si l'on y regarde de plus près, le calcul se fait en réalité en temps $\Theta(F_n)$

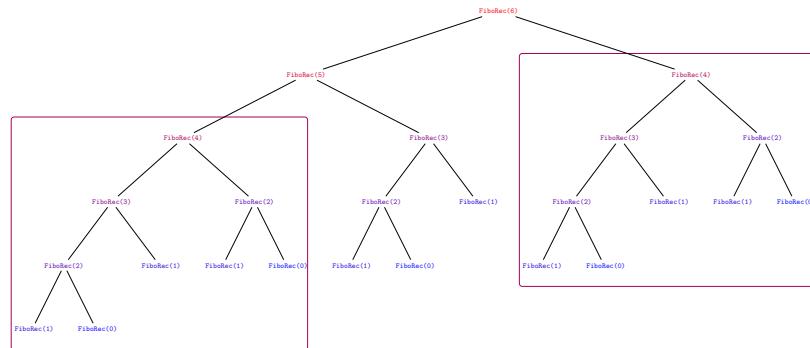
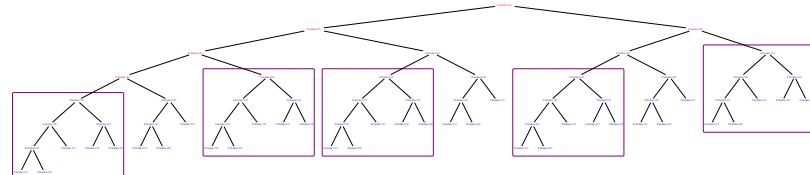
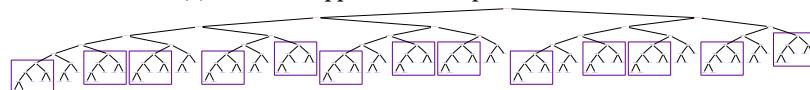
(a) Arbre des appels récursifs pour $\text{FiboRec}(6)$.(b) Arbre des appels récursifs pour $\text{FiboRec}(8)$.(c) Arbre des appels récursifs pour $\text{FiboRec}(10)$.

FIG. 5.1 : Lorsque l'on appelle FiboRec , on effectue de nombreuses fois les mêmes calculs : pour $n = 6$ on recalcule 2 fois F_4 , pour $n = 8$ on le fait 5 fois et pour $n = 10$ on le fait 13 fois.

On se rend assez facilement compte qu'on peut éviter la redondance de ces appels récursifs en stockant tous les termes intermédiaires de la suite et en les calculant dans le bon ordre, du plus petit au plus grand :

```
def FiboIter(n):
    F = [1] * (n+1)
    for i in range(2,n+1):
        F[i] = F[i-1] + F[i-2]
    return F[n]
```

On a maintenant un algorithme qui calcule F_n en $\Theta(n)$. La programmation dynamique va correspondre à cette méthode générique qui consiste en deux étapes :

1. Trouver une formule récursive.
2. Comprendre comment la calculer efficacement.

Dans le cas présent, trouver la formule récursive était très facile, étant donné qu'elle correspondait exactement à la définition de la suite de Fibonacci. Cela ne sera pas le cas en général et établir la formule récursive sera l'étape la plus importante et la plus difficile. Dans la suite de ce chapitre, nous allons nous intéresser à plusieurs problèmes pour lesquels on va établir la formule de récursion et comment la calculer.

1 Plus longue sous-suite commune

1.1 Le problème

Un problème fréquent en analyse d'ADN est de devoir trouver la meilleure manière d'aligner deux brins pour pouvoir les comparer. Pour cela on utilise souvent leur plus longue sous-suite commune, c'est à dire la plus grande suite de lettres non consécutives commune.

Par exemple, ABCD et ACBAD ont :

- 5 sous-suites de longueur 2 en commun : AB, AC, AD, BD, CD,
- 2 sous-suites de longueur 3 en commun : ABD, ACD,
- aucune sous-suite commune de longueur 4 ou plus.

Il y a donc deux plus longues sous-suites communes ABD et ACD.

1.2 Trouver une formule de récurrence

On notera dorénavant la longueur de la plus longue sous-suite commune entre deux mots w_1 et w_2 par PLSC(w_1, w_2). On peut d'ailleurs remarquer que PLSC(w_1, ε) = PLSC(ε, w_2) = 0 où ε est le mot vide.

Maintenant, étant donné deux mots non vides $w_1 = m_1a$ et $w_2 = m_2b$ on va pouvoir construire une formule récursive qui dépendra de leurs deux dernières lettres a et b grâce à la disjonction de cas suivante :

1. Quand elles sont égales ($a = b$), la plus longue sous-suite commune est celle de m_1 et m_2 à laquelle on ajoute cette lettre.
2. Quand elles ne le sont pas, soit la plus longue sous-suite commune est celle de m_1a et de m_2 , c'est à dire que b n'est pas dans la sous-suite commune la plus longue,

3. soit c'est celle de m_1 et de m_2b , c'est à dire que a n'est pas dans la sous-suite commune la plus longue.

On peut établir donc la relation suivante :

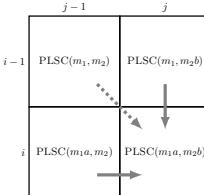
$$\text{PLSC}(m_1a, m_2b) = \max \begin{cases} 1 + \text{PLSC}(m_1, m_2) & \text{si } a = b \\ \text{PLSC}(m_1a, m_2) \\ \text{PLSC}(m_1, m_2b) \end{cases}$$

1.3 Calcul

Il nous reste maintenant à comprendre comment calculer notre formule. On remarque que la formule de récurrence se ramène toujours à des cas où les mots considérés sont des préfixes des mots initiaux, on va donc construire un tableau de taille $(m+1) \times (n+1)$ où m et n sont les longueurs des deux mots où la case (i, j) contiendra la longueur de la plus longue sous-suite commune entre le préfixe de longueur i du premier mot et le préfixe de longueur j du deuxième mot.

La case $(0, 0)$ correspondra aux deux préfixes vides, leur plus longue sous-suite commune est de longueur 0, il en ira de même pour les cases $(i, 0)$ et $(0, j)$ qui correspondront à la plus longue sous-suite en commun entre m_1 (resp. m_2) et le préfixe vide de m_2 (resp. m_1).

Maintenant pour calculer la valeur de la case (i, j) il suffit de regarder ses cases voisines² :



On obtient donc l'algorithme suivant :

```
def plsc(mot1,mot2):
    n1, n2 = len(mot1), len(mot2)
    t = [[0]*(n2+1) for i in range(n1+1)]
    for i in range(1, n1+1):
        for j in range(1, n2+1):
            if mot1[i-1] == mot2[j-1]:
                t[i][j] = 1 + t[i-1][j-1]
            t[i][j] = max(t[i][j],
                           t[i-1][j],
                           t[i][j-1])
    return t[n1][n2]
```

Cet algorithme nous permet d'obtenir la longueur de la plus longue sous-suite commune, on peut cependant aisément le modifier pour qu'il reconstruise une des solutions et la renvoie :

²Pour les algorithmes sur les mots, par tradition on représentera les tableaux avec i qui incrémentera en descendant et j en allant à droite.

	a	l	g	o	r	i	t	h	m	e
r	0	0	0	0	0	0	0	0	0	0
y	0	0	0	0	0	1	1	1	1	1
t	0	0	0	0	0	1	1	2	2	2
h	0	0	0	0	0	1	1	2	3	3
m	0	0	0	0	0	1	1	2	3	4
e	0	0	0	0	0	1	1	2	3	4
										5

(a) Tableau pour $m_1 = rythme$ et $m_2 = algorithme$, il n'y a qu'une plus longue sous-suite commune : *rthme*. une sous-suites la plus longue correspond à un chemin valide de la case en bas à droite et allant à la case en haut à gauche qui "remontent" le long du calcul. En particulier un déplacement en diagonale n'est autorisé que si les lettres correspondantes sont égales et un déplacement horizontal ou vertical n'est autorisé que si la valeur des cases est la même.

	t	r	o	u	v	e	r	e
t	0	0	0	0	0	0	0	0
r	0	1	2	2	2	2	2	2
o	0	1	2	3	3	3	3	3
u	0	1	2	3	4	4	4	4
b	0	1	2	3	4	4	4	4
a	0	1	2	3	4	4	4	4
d	0	1	2	3	4	4	4	4
o	0	1	2	3	4	4	4	4
u	0	1	2	3	4	4	4	4
r	0	1	2	3	4	4	5	5

(b) Tableau pour $m_1 = troubadour$ et $m_2 = trouvere$, dans ce cas il y a plusieurs sous-suites, chacune correspond à un chemin valide différent.

FIG. 5.2 : Pour le calcul de PLSC (m_1, m_2), on commence par la case en haut à gauche en parcourant ligne par ligne. Si les prefixes correspondant à la case en cours de considération finissent par la même lettre, on prend en compte la case dans la diagonale en considération pour le calcul du max, sinon on ne regarde que les cases au dessus et à gauche.

```

def plsc2(mot1,mot2):
    n1, n2 = len(mot1), len(mot2)
    t = [[0]*(n2+1) for i in range(n1+1)]
    for i in range(1, n1+1):
        for j in range(1, n2+1):
            if mot1[i-1] == mot2[j-1]:
                t[i][j] = 1 + t[i-1][j-1]
            t[i][j] = max(t[i][j],
                           t[i-1][j],
                           t[i][j-1])
    i,j = n1,n2
    s = ""
    while (i,j) != (0,0):
        if j>0 and i>0 and mot1[i-1] == mot2[j-1]:
            s = mot1[i-1] + s # O(|s|)
            i,j = i-1,j-1
        elif i>0 and t[i][j] == t[i-1][j] :
            i,j = i-1,j
        else:
            i,j = i,j-1

    return (t[n1][n2],s)

```

La construction des tableaux et la reconstruction d'une solution sont montrés en Figure 5.2.

2 Floyd-Warshall : tous les plus courts chemins

La prochaine application de la programmation dynamique que nous allons regarder est le problème du calcul des plus courts chemins dans un graphe pour tous les couples de sommets. Nous allons modifier notre définition de graphe pour y ajouter une notion de poids sur les arêtes :

Définition 5.1

Graphe pondéré

Un *graphe pondéré* est un graphe $G = (V, E)$ auquel on ajoute une fonction $\delta : E \rightarrow \mathbb{R}^+$ donnant les poids des arêtes. On notera $G = (V, E, \delta)$.

L'algorithme de parcours en largeur présenté en Section 3 ne permet pas de trouver le plus petit chemin dans un graphe pondéré : ajouter un sommet u avant un sommet v dans la file ne garantit plus que u soit accessible par un chemin plus court que v et c'est exactement ce qui faisait fonctionner l'algorithme. On peut remplacer la file par une file de priorité, et l'on obtient alors l'algorithme de Dijkstra. Vous verrez cet algorithme en cours de théorie des graphes au second semestre.

Nous allons voir deux manières d'établir la relation de récurrence.

2.1 Relation de récurrence grâce au nombre de sommets

Une première manière de voir cette relation est en se focalisant sur le nombre de sommets : si l'on connaît déjà les plus courts chemins sur un graphe à k sommets, que se passe-t-il si on lui ajoute un $(k + 1)$ -ème sommet u_{k+1} ?

- Le chemin le plus court de u_{k+1} vers u_i avec $i \leq k + 1$ peut passer par n'importe laquelle des arêtes sortantes de u_{k+1} , on a donc :

$$d_{k+1}(u_{k+1}, u_i) = \min_{j \leq k} (\delta(u_{k+1}, u_j) + d_k(u_j, u_i) \mid (u_{k+1}, u_j) \in E)$$

- De la même manière, le chemin le plus court de u_i vers u_{k+1} peut passer par n'importe laquelle des arêtes entrantes de u_{k+1} , on a donc :

$$d_{k+1}(u_i, u_{k+1}) = \min_{j \leq k} (\delta(u_j, u_{k+1}) + d_k(u_i, u_j) \mid (u_j, u_{k+1}) \in E)$$

- Le chemin le plus court pour aller de u_i à u_j peut maintenant éventuellement être raccourci en passant par k :

$$d_{k+1}(u_i, u_j) = \min(d_k(u_i, u_j), d_{k+1}(u_i, u_k) + d_{k+1}(u_k, u_j))$$

On obtient alors l'algorithme suivant en supposant que la méthode est dans une classe `GraphePondere` qui a une méthode `poids(u, v)` renvoyant le poids de l'arête allant de `u` vers `v` quand elle existe et l'infini sinon :

```
def FloydWarshall(self):
    d = {}
    sommets = [u for u in self.noeuds()]
    for k in range(len(sommets)):
        for i in range(k+1):
            d[k,i] = self.poids(k,i)
            d[i,k] = self.poids(i,k)
        for i in range(k+1):
            for j in range(k):
                d[i,k] = min(d[i,k], d[i,j] + d[j,k])
                d[k,i] = min(d[k,i], d[k,j] + d[j,i])
        for i in range(k):
            for j in range(k):
                d[i,j] = min(d[i,j], d[i,k] + d[k,j])
    return d
```

L'algorithme obtenu est en $\mathcal{O}(n^3)$

2.2 Relation grâce aux sommets par lesquels passe un chemin

Une autre manière de construire une relation de récursion est de regarder les sommets par lesquels passe un chemin du sommet u_i vers le sommet u_j . Initialement, en ne regardant que les chemins ne passant par aucun sommet intermédiaire, ceux-ci correspondent exactement aux arêtes. On peut ensuite s'intéresser aux chemins les plus courts ne passant que par le sommet u_1 , on a alors :

$$d(u_i, u_j) = \min(\delta(u_i, u_j), \delta(u_i, u_1) + \delta(u_1, u_j))$$

On peut généraliser cette relation assez facilement. Supposons que $d_k(u_i, u_j)$ est le poids du chemin le plus court ne passant que par les sommets $\{u_1, \dots, u_k\}$, on obtient la relation suivante entre d_{k+1} et d_k :

$$d_{k+1}(u_i, u_j) = \min(d_k(u_i, u_j), d_k(u_i, u_{k+1}) + d_k(u_{k+1}, u_j))$$

Cette relation se traduit par l'algorithme suivant qui est également en $\mathcal{O}(n^3)$:

```
def FloydWarshall2(self):
    d = {(u,v) : self.poids(u,v) for u in self.noeuds()
          for v in self.noeuds()}
    sommets = [u for u in self.noeuds()]
    n = len(sommets)
    for k in range(n):
        for i in range(n):
            for j in range(n):
                d[i,j] = min(d[i,j], d[i,k] + d[k,j])

    return d
```

3 Memoïzation

Dans toutes les implémentations précédentes de la programmation dynamique, nous avons explicitement cherché un ordre optimal de calcul, parfois on peut s'en abstenir en faisant usage d'une mémoire auxiliaire. L'idée est de ne calculer qu'une seule fois chaque appel récursif en stockant son résultat en mémoire la première fois qu'on le fait. Par exemple pour le calcul de la suite de Fibonacci, on peut modifier le programme FiboRec de la manière suivante :

```
def FiboMem(n):
    mem = {}
    def fib(n):
        try:
            return mem[n]
        except:
            if n <=1 :
                return 1
            r = fib(n-1) + fib(n-2)
            mem[n] = r
            return r
    return fib(n)
```

Ici, la fonction auxiliaire `fib` stocke les résultats déjà calculés dans `mem`, ainsi pour une même valeur de n , on ne fera qu'une seule fois les sous-appels récursifs. Le nombre d'appels récursifs obtenus de cette manière est $\mathcal{O}(n)$, on peut les voir en [Figure 5.3](#).

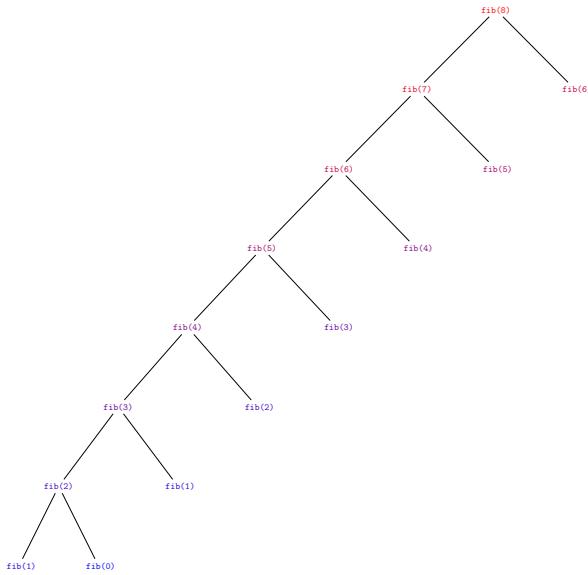


FIG. 5.3 : Les appels récursifs ayant lieu dans la fonction auxiliaire `fib`. Il y en a un nombre linéaire en n .

4 Dominos sur une ligne $n \times 3$

On souhaite connaître le nombre de manières dont on peut remplir une ligne $3 \times n$ avec des dominos de taille 1×2 et 2×1 . En [Figure 5.5](#) on peut voir deux dominos placés sur une ligne 10×3 .

Ici on peut essayer d'exprimer le nombre de manières de remplir la ligne $n \times 3$ en fonction du nombre de manières de remplir une ligne plus petite. Pour certains cas on y parvient mais pas dans d'autres, comme on peut le voir en [Figure 5.7](#). Ici la solution consiste à avoir plusieurs relations récursives dépendantes les unes des autres. On va pour cela regarder la manière de remplir plusieurs formes, montrées sur la ?? :

- $R(n)$ comptera le nombre de manières de remplir le rectangle $3 \times n$,
- $L_1(n)$ comptera le nombre de manières de remplir le rectangle $3 \times (n - 1)$ avec une case en plus en bas à droite,
- $L_1(n)$ comptera le nombre de manières de remplir le rectangle $3 \times (n - 1)$ avec deux cases empilées verticalement en bas à droite,
- $\Gamma_1(n)$ comptera le nombre de manières de remplir le rectangle $3 \times (n - 1)$ avec une case en plus en haut à droite,
- $\Gamma_2(n)$ comptera le nombre de manières de remplir le rectangle $3 \times (n - 1)$ avec deux cases empilées verticalement en haut à droite.
- Il y a en théorie deux autres cas possibles, mais on peut les écarter en notant un fait simple : quand on colorie les cases à la manière d'un damier un domino doit couvrir une case noire et une case blanche, or dans ces deux cas il n'est jamais possible d'obtenir le même nombre de cases noires et de cases blanches, quelle que soit la valeur de n et on ne peut donc pas les résoudre.

On peut maintenant établir nos relations de récurrence, en commençant par $R(n)$

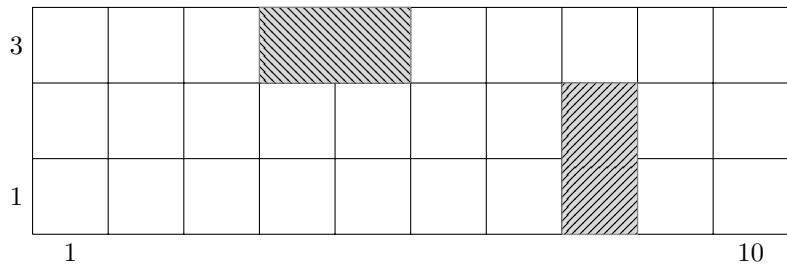


FIG. 5.4 : Deux dominos placés sur la grille 10×3 , un horizontalement, un verticalement.

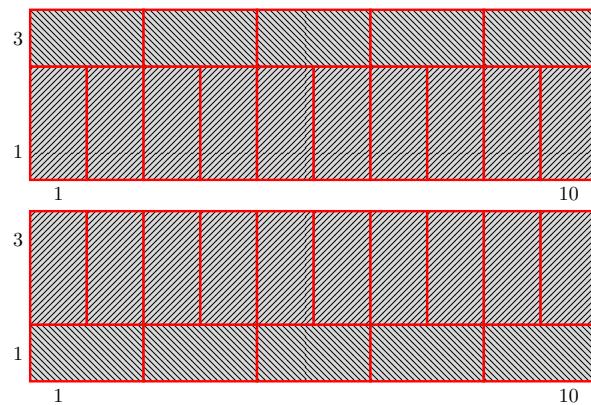


FIG. 5.5 : Deux manières de remplir la grille 10×3 .

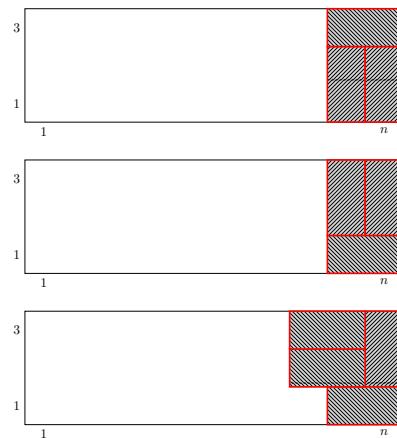


FIG. 5.6 : On peut essayer de se ramener à un cas plus petit en plaçant quelques dominos : dans les deux premiers placements cela fonctionne car la forme obtenue est bien un rectangle $(n - 2) \times 3$, cependant dans le dernier cas on obtient une autre forme.

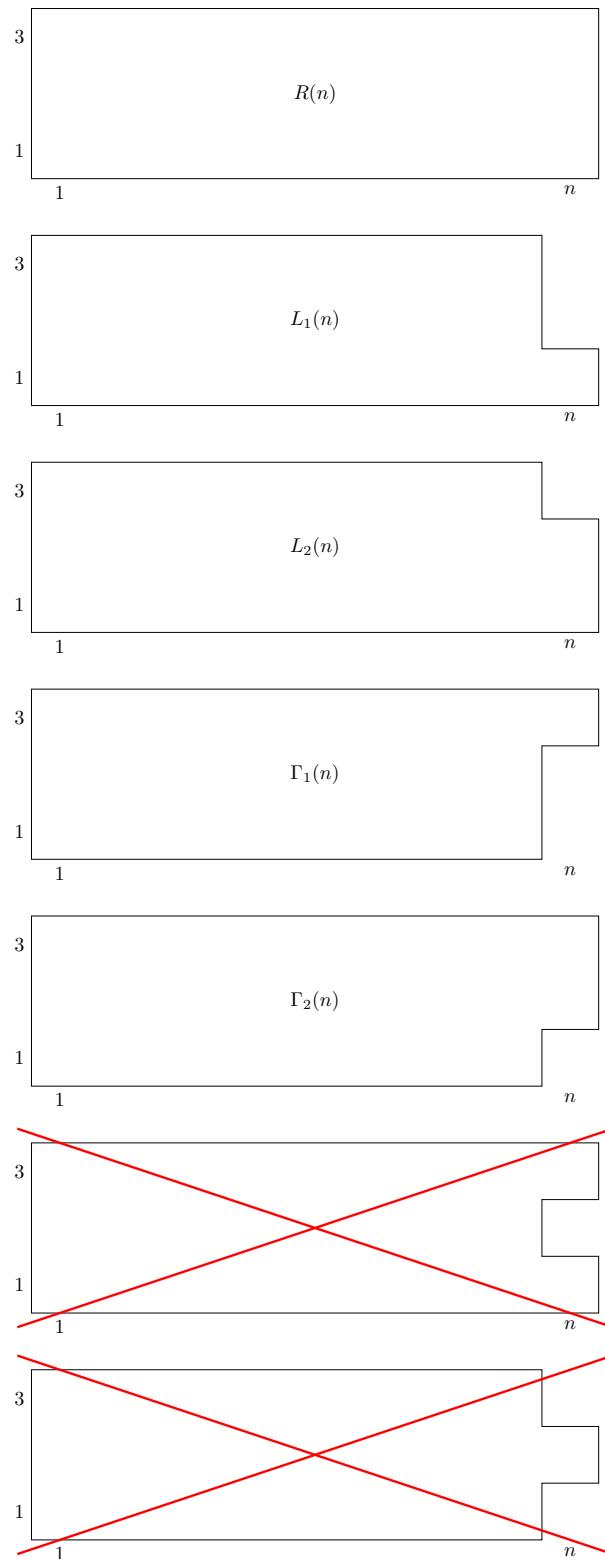
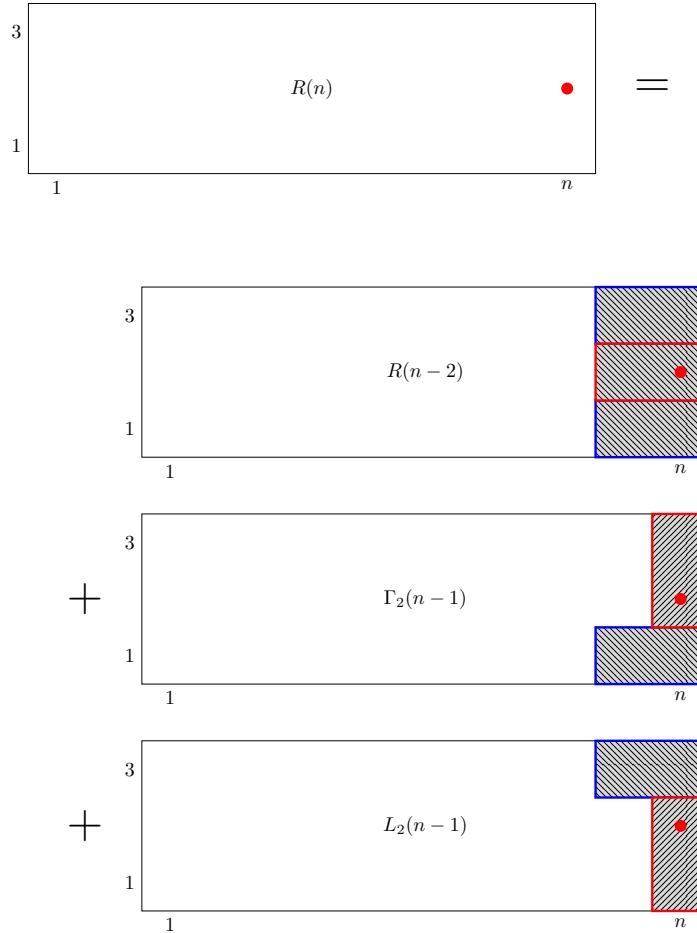


FIG. 5.7 : On va compter le nombre de manière de remplir avec des dominos ces formes, les deux dernières ne peuvent pas avoir lieu en plaçant des dominos.

que l'on peut exprimer en fonction de L_2 , Γ_2 et R sur des valeurs plus petites. Pour cela il suffit de se rendre compte qu'il n'y a que trois possibilités pour couvrir le point rouge, illustrées par les dominos en rouge. Ces dominos forcent les dominos bleus :



On peut donc poser la relation suivante :

$$R(n) = R(n-2) + \Gamma_2(n-1) + L_2(n-1)$$

Avec des raisonnements similaires on obtient les relations :

- $L_2(n) = R(n-1) + \Gamma_1(n-1)$
- $\Gamma_2(n) = R(n-1) + L_1(n-1)$
- $L_1(n) = \Gamma_2(n-1)$
- $\Gamma_1(n) = L_2(n-1)$

On peut également noter que $\Gamma_2 = L_2$ et que $\Gamma_1 = L_1$ par symétrie, on peut donc simplifier les relations en ne conservant que les deux quantités Γ_2 et $R(n)$:

- $R(n) = R(n-2) + 2 * \Gamma_2(n-1)$
- $\Gamma_2(n) = R(n-1) + \Gamma_2(n-2)$

On a les cas de base suivants :

- $R(0) = 0, R(1) = 0$
- $\Gamma_2(1) = 1$

Maintenant que tout ce travail est fait, le calcul de $R(n)$ est relativement simple : on peut remplir deux tableaux au fur et à mesure contenant les valeurs intermédiaires $R(i)$ et $\Gamma_2(i)$ pour tous les $i \leq n$.

6

REDUCTIONS ET NP-COMPLÉTUDE

Dans les problèmes que nous avons regardé précédemment et en TD, certains se ressemblaient étrangement. Nous allons ici voir la notion de *réduction* : comment on peut parfois résoudre un problème facilement si l'on sait en résoudre un autre. Autrement dit, si en résolvant un problème A on sait résoudre un problème B , alors B est plus “facile” que A .

Pour pouvoir parler de réduction nous allons d'abord devoir définir précisément ce qu'est un problème.

1 Encodage

Prenons un exemple que nous avons abordé en TP : le problème de la satisfiabilité d'une formule booléenne sous forme normale conjonctive.

Une formule booléenne $\phi(x_1, \dots, x_k)$ sous forme normale conjonctive est une conjonction de clauses C_i constituées uniquement de disjonctions de variables de leurs négations parmi x_1, \dots, x_k , par exemple :

$$\phi(x_0, \dots, x_8) = \underbrace{(x_1 \vee x_4 \vee \neg x_6)}_{C_1} \wedge \underbrace{(\neg x_2 \vee \neg x_4 \vee x_8)}_{C_2} \wedge \underbrace{(\neg x_3)}_{C_3}$$

Ici ϕ est une formule sur 8 variables x_1, \dots, x_8 , avec trois clauses C_1, C_2, C_3 .

Une formule ϕ est dite *satisfiable* si on peut assigner des valeurs à x_1, \dots, x_8 de manière à ce que la formule ϕ soit vraie. Pour l'exemple précédent, les valeurs $x_3 = 0, x_1 = 1, x_4 = 0$ rendent la formule vraie (quelles que soient les valeurs de x_2, x_5, x_6, x_7, x_8) et la formule est donc satisfiable.

Quelle que soit la structure de données avec laquelle on stocke notre formule, ultimement celle-ci sera représentée par une suite de 0 et de 1 en mémoire. La manière de coder une formule par une suite de 0 et 1 est ce que l'on appellera un encodage.

Voici par exemple plusieurs manières de stocker une formule, chacune d'entre elles définit un encodage différent :

- Sous forme de chaîne de caractères : $(x_1 \vee x_4 \vee \neg x_6) \wedge (\neg x_2 \vee \neg x_4 \vee x_8) \wedge (\neg x_3)$
- Sous forme d'une liste de table de hachage : chaque clause est représentée par une table de hachage de ses variables où la valeur associée est si la variable est sous forme négative ou non.
- Sous forme d'une liste de liste : chaque clause est représentée par une liste de couples où chaque couple contient un nom de variable et si celle-ci est sous forme négative ou non.

Néanmoins l'encodage lui-même, c'est à dire la traduction de la formule en mot binaire, importe peu tant que celui-ci est raisonnable, c'est à dire que celui-ci peut être calculé facilement à partir de la formule.

Définition 6.1 *Encodage*

Un *encodage* est une fonction $e : O \rightarrow \{0, 1\}^*$ qui transforme un objet¹ parmi un ensemble d'objets O en mot formé de 0 et de 1.

Deux encodages seront dits *équivalents* si l'on peut passer de l'un à l'autre facilement :

Définition 6.2 *Encodages équivalents*

Étant donné deux encodages $e_1 : O \rightarrow \{0, 1\}^*$ et $e_2 : O \rightarrow \{0, 1\}^*$, on dira qu'ils sont équivalents s'il existe deux fonctions $t_{1 \rightarrow 2}$ et $t_{2 \rightarrow 1}$ telles que :

- $t_{2 \rightarrow 1}(e_2(o)) = e_1(o)$ et $t_{1 \rightarrow 2}(e_1(o)) = e_2(o)$ pour tout objet $o \in O$.
- $t_{1 \rightarrow 2}$ et $t_{2 \rightarrow 1}$ sont calculables en temps linéaire.

Les fonctions $t_{i \rightarrow j}$ peuvent être vues comme des transformations d'un encodage à un autre. Le fait de les prendre calculables en temps linéaire garantit que l'on ajoute pas trop de complexité en passant d'un encodage à un autre.

Les trois encodages pour une formule énoncés précédemment sont équivalents.

1.1 Encodages non équivalents

Il existe bien entendu des encodages qui ne sont pas équivalents pour un même objet. Pour les entiers par exemple on peut :

- les encoder sous forme binaire, c'est à dire sous la forme de leur écriture en base deux. Par exemple 111 en binaire correspond à l'entier 7.
- les encoder en unaire, c'est à dire sous forme d'une suite de 1 dont la longueur est exactement l'entier. Par exemple 7 s'encode par 1111111.

Ces deux encodages ne sont pas équivalents, un entier N se code en $\log N$ bits en binaire et en N bits en unaire. Ainsi on pour passer de l'écriture binaire à l'écriture unaire il faut un temps exponentiel rien que pour écrire les bits.

2 Problème

Jusqu'à présent, nous n'avons pas défini formellement ce qu'était un problème. Contrairement aux TDs/TPs où nous avons vu de nombreux exercices dont le but était de trouver la plus petite solution, de compter le nombre de solutions, etc..., nous nous intéresserons ici uniquement aux questions fermées, c'est à dire celles dont la réponse est Oui ou Non.

Étant donné un ensemble d'objets O et un encodage de ces objets e sous forme de mots, un problème sera donc de savoir quels sont les mots de $\{0, 1\}^*$ dont la réponse est Oui.

Définition 6.3

Problème

Un *problème* est un ensemble (ou langage) $P \subseteq \{0, 1\}^*$.

Trouver un algorithme résolvant un problème P revient donc à trouver un algorithme A qui prend en entrée un mot $m \in \{0, 1\}^*$ et qui renvoie Oui si $m \in P$ et Non sinon.

Cette définition formelle de problème sera pratique à manipuler pour faire des preuves dans la suite, mais quand on décrit un problème il n'est souvent pas pratique de décrire l'encodage précisément, juste de savoir qu'il existe et de connaître sa taille.

Voici quelques exemples de problèmes :

- SAT : soit ϕ une formule sous forme normale conjonctive, est-elle satisfiable ?
- SUBSETSUM : soit $S \subseteq \mathbb{N}$ un ensemble d'entiers et K un entier, est-ce qu'il existe un sous-ensemble de S dont la somme est K ? On peut ici trouver un encodage dont la taille est en $\mathcal{O}(|S| \log M)$ où M est l'entier le plus grand de S .
- TILING : soit T un ensemble de tuiles carrées dont les quatre bords sont colorés et N un entier. Existe-t-il un moyen de tuiler un carré $N \times N$ à l'aide des tuiles de T de manière à ce que les bords de tuiles se touchant soient de la même couleur ? On dispose d'un nombre illimité de copies de chaque tuile et on ne peut pas les tourner.
- PLSC : soit m_1 et m_2 deux mots sur un alphabet Σ et un entier K , existe-t-il une sous-suite commune de longueur au moins K ?

3 Réduction polynomiale

On peut comparer deux problèmes à l'aide de la notion de *rédiction* :

Définition 6.4**Réduction polynomiale**

Etant donné un problème P_1 et un problème P_2 , on dit que P_1 se réduit à P_2 et l'on note $P_1 \leq_P P_2$ quand il existe une fonction f calculable en temps polynômial telle que :

$$m \in P_1 \Leftrightarrow f(m) \in P_2$$

En pratique, cela veut dire que si l'on sait résoudre P_2 , alors on sait résoudre P_1 avec au plus un coût polynômial supplémentaire. En particulier :

- S'il existe un algorithme polynomial résolvant P_2 , alors il existe un algorithme polynomial résolvant P_1 .
- S'il n'existe pas d'algorithme sous-exponentiel résolvant P_1 , alors il n'existe pas d'algorithme sous-exponentiel résolvant P_2 .

Les réductions sont transitives, c'est à dire que si $P_1 \leq_P P_2$ et $P_2 \leq_P P_3$, alors $P_1 \leq_P P_3$.

4 Exemples de réductions

Nous allons maintenant voir quelques exemples de réductions polynomiales.

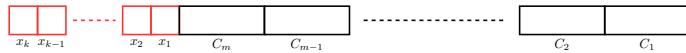
4.1 $\text{SAT} \leq_P \text{SUBSETSUM}$

Pour réduire SAT à SUBSETSUM, il faut que l'on trouve un moyen d'exprimer une formule ϕ comme un ensemble d'entier S et un objectif K de manière à ce que l'on puisse trouver un sous-ensemble de S dont la somme vaut K si et seulement si ϕ est satisfiable.

Prenons donc une formule : $\phi(x_1, \dots, x_k) = C_1 \wedge \dots \wedge C_m$

On va construire des entiers N_{x_i} et $N_{\bar{x}_i}$ qui correspondront respectivement à choisir $x_i = 1$ et $x_i = 0$. Il nous faudra donc faire en sorte de ne pas pouvoir utiliser les deux à la fois pour atteindre l'objectif K .

Pour cela, nos entier N_{x_i} seront divisés en plusieurs parties :



La première partie en gris nous permettra de nous assurer que l'on a choisi une (et une seule) valeur pour chaque x_i . La seconde nous permettra de nous assurer que chaque clause est bien rendue vraie par au moins une des valeurs des x_i choisies. Formellement :

$$N_{x_i} = 10^{mT+i} + \sum_{j=1}^m 10^{Tj} * \begin{cases} 1 & \text{si } x_i \text{ apparaît dans } C_j \\ 0 & \text{sinon} \end{cases}$$

$$N_{\bar{x}_i} = 10^{mT+i} + \sum_{j=1}^m 10^{Tj} * \begin{cases} 1 & \text{si } \bar{x}_i \text{ apparaît dans } C_j \\ 0 & \text{sinon} \end{cases}$$

où T est un entier qui sera choisi plus tard.

$$N_{x_4} \quad \begin{array}{|c|c|} \hline 0 & 0 \\ \hline \end{array} \cdots \cdots \begin{array}{|c|c|c|c|c|} \hline 0 & 1 & 0 & 0 & 0 \\ \hline x_5 & x_4 & x_3 & x_2 & x_1 \\ \hline \end{array} \cdots \cdots \begin{array}{|c|c|c|c|c|c|} \hline 1 & 0 & 1 & 1 & 0 & 0 \\ \hline C_6 & C_5 & C_4 & C_3 & C_2 & C_1 \\ \hline \end{array}$$

$$N_{\bar{x}_4} \quad \begin{array}{|c|c|} \hline 0 & 0 \\ \hline \end{array} \cdots \cdots \begin{array}{|c|c|c|c|c|} \hline 0 & 1 & 0 & 0 & 0 \\ \hline x_5 & x_4 & x_3 & x_2 & x_1 \\ \hline \end{array} \cdots \cdots \begin{array}{|c|c|c|c|c|c|} \hline 0 & 0 & 0 & 0 & 0 & 1 \\ \hline C_6 & C_5 & C_4 & C_3 & C_2 & C_1 \\ \hline \end{array}$$

FIG. 6.1 : Un exemple pour N_{x_4} et $N_{\bar{x}_4}$ où l'on suppose que x_4 apparaît dans C_3, C_4 et C_6 et \bar{x}_4 apparaît dans C_1 .

Pour forcer à choisir une et une seule valeur dans l'objectif, il faut donc que les chiffres correspondant à la partie rouge soient tous 1. Il nous faut donc $K = 1 \dots 1 \times 10^{mT} + q$ où q est la partie correspondant aux C_i .

Il faut maintenant compléter l'objectif de manière à forcer les parties correspondant aux C_i à être toutes positives. On ne peut pas simplement choisir qu'à chaque case noire correspondent un 1, car cela signifierait que l'on force à ce qu'exactement un des littéraux de la clause la rende positive. La somme de N_{x_i} et $N_{\bar{x}_i}$ pouvant rendre une clause vraie et permettant d'atteindre la partie déjà définie de l'objectif aura une valeur comprise entre 1 et k , le nombre de variables. Comme nous ne pouvons choisir qu'un seul objectif, nous allons ajouter des entiers de complétion : $E_j = \{2 * 10^{Tj}, 3 * 10^{Tj}, \dots k * 10^{Tj}\}$.

La somme de ces entiers vaut $\sum_{i=2}^k i * 10^{Tj} = \frac{k(k+1)}{2} - 1$ et nous allons donc fixer comme objectif

$$K = \sum_{i=1}^k 10^{mT+i} + \sum_{j=1}^m 10^{Tj} * \frac{k(k+1)}{2}$$

où l'on choisit $T = \lceil \log_{10} \frac{k(k+1)}{2} \rceil + 1$.

Cet objectif ne peut être atteint que si la valeur correspondant à chaque C_j est au moins de 1 et il peut être atteint pour n'importe quelle valeur entre 1 et k .

Pour résumer, notre ensemble d'entiers final est

$$S = \bigcup_{1 \leq i \leq k} (\{N_{x_i}\} \cup \{N_{\bar{x}_i}\}) \cup \bigcup_{1 \leq j \leq m} E_j.$$

Le processus de construction de cet ensemble d'entiers peut bien être fait en temps polynomial et on a donc une réduction prouvant que si l'on peut résoudre SUBSETSUM, alors on peut résoudre SAT.

4.2 SAT \leq_P 3SAT

On peut définir le problème de la satisfiabilité pour les formules sous forme normale conjonctive ayant au plus 3 littéraux par clause, on le nomme 3SAT.

On peut montrer relativement aisément que SAT se réduit à 3SAT. En remarquant qu'étant donné une formule

$$x_{i_1}^{s_1} \vee x_{i_2}^{s_2} \vee x_{i_3}^{s_3} \vee \dots \vee x_{i_k}^{s_k},$$

celle-ci est satisfiable uniquement si celle-ci l'est :

$$(y \vee x_{i_1}^{s_1} \vee x_{i_2}^{s_2}) \wedge (\bar{y} \vee x_{i_3}^{s_3} \vee \dots x_{i_k}^{s_k})$$

où y est une nouvelle variable. En effet, si $y = 1$ alors il faut que l'un des $x_{i_3}^{s_3}, \dots, x_{i_k}^{s_k}$ soit vrai, et si $y = 0$ alors il faut que soit $x_{i_1}^{s_1} = 1$ soit $x_{i_2}^{s_2} = 1$.

On peut de cette manière transformer une formule de SAT en une formule de 3SAT en temps linéaire en multipliant au plus sa taille par $\frac{3}{2}$ et on a donc une réduction polynomiale.

4.3 $SAT \leq_P LONGESTPATH$

Le problème du plus long chemin dans un graphe consiste à trouver le plus long chemin qui ne passant au maximum qu'une fois par sommet, pour le transformer en problème de décision, on peut le transformer en la question "Étant donné un graphe G et un entier k , le plus long chemin de G est-il de longueur au moins k ?". On peut montrer que ce problème est au moins aussi difficile que SAT.

Pour cela on va construire un graphe G_ϕ étant donné une formule ϕ sous forme normale conjonctive. L'idée est de faire un graphe en forme de "serpent" où chaque variable booléenne x_i sera représentée par deux chemins, comme dans la Figure 6.2. Ces chemins seront ensuite reliés entre eux (on ajoute une arête entre le sommet o_i et le sommet o_{i+1}). On ajoute ensuite une dernière ligne de sommets qui correspondront

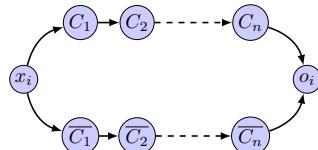


FIG. 6.2 : Deux chemins possibles pour une variable x_i , pour chaque clause de ϕ , on ajoute un noeud dans chacun des chemins.

à regarder si chaque clause sera satisfaite en choisissant des valeurs pour chaque x_i , voir la Figure 6.3.

5 NP-complétude

Vous verrez l'an prochain² une définition formelle de la classe de complexité **NP**. Pour ce cours, nous nous contenterons de la définition informelle suivante.

Définition 6.5

classe NP

Un problème P est dans la **classe NP** s'il existe une fonction $f : \{0, 1\}^* \times \{0, 1\}^* \rightarrow \{0, 1\}$, calculable en temps polynomial, telle que :

$$\exists y, f(x, y) = 1 \Leftrightarrow x \in P$$

On peut voir la fonction f comme une fonction de vérification d'une solution y . Intuitivement, un problème est dans **NP** s'il existe un algorithme qui étant donné

²Si vous restez, ou si votre nouvelle université/école a un cours de complexité.

une solution potentielle peut vérifier que c'est vraiment une solution. Un exemple serait pour SAT : on peut vérifier en temps linéaire si étant donné une assignation des variables x_i la formule est satisfaite par cette assignation.

La difficulté dans un problème **NP** n'est donc pas de vérifier une solution mais bien de la trouver.

En pratique, il existe des problèmes qui sont dans **NP** et qui sont maximalement difficiles, c'est à dire qu'en les résolvant on peut résoudre n'importe quel problème de **NP**. On appelle ces problèmes **NP-complets** :

Définition 6.6

NP-complet

Un problème **NP-complet** est un problème dans **NP** auquel se réduisent tous les problèmes de **NP** :

$$P_1 \text{ est NP-complet} \Leftrightarrow \forall P_2 \in \mathbf{NP}, P_2 \leq_P P_1$$

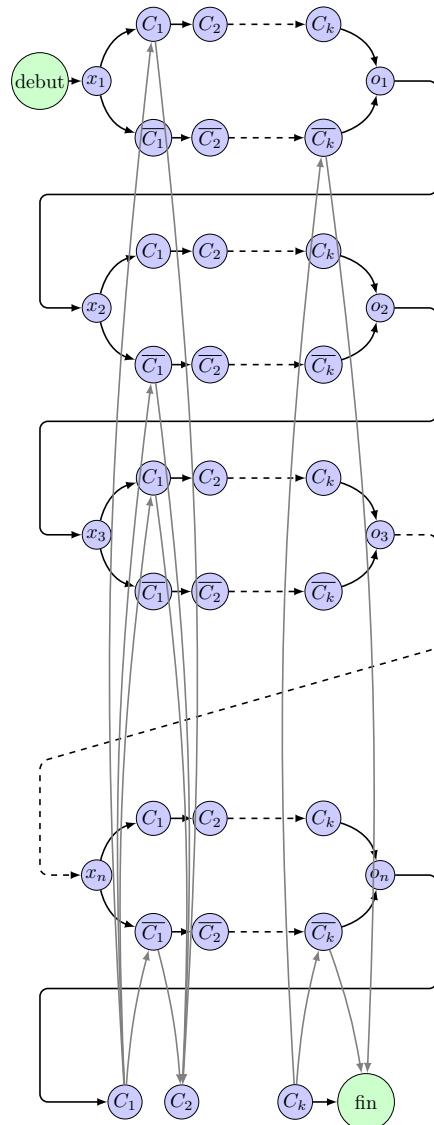


FIG. 6.3 : Dans la construction finale, la dernière ligne fait correspondre les littéraux à chaque clause.

BIBLIOGRAPHIE

- [TK05] Eva TARDOS et Jon KLEINBERG. *Algorithm Design*. en. Upper Saddle River, NJ : Pearson, mars 2005 (cf. p. 5).

INDEX

NP-complet, 55

classe **NP**, 54

convexe, 12

encodage

équivalence, 50

encodage, 50

enveloppe convexe, 12

file, 21

formule satisfiable, 49

Graphé

Arêtes, 17

matrice d'adjacence, 19

non-orienté, 17

Sommets, 17

graphe

pondéré, 40

Landau

domination, 4

même ordre, 4

marche de Jarvis, 12

parcours

en largeur, 21

en profondeur

arbre de parcours, 24

pile, 27

problème, 51

réduction, 51

