INF478
Résolution de Problèmes Algorithmiques

S. Oudot, A. Nolin

(polycopié repris de celui de C. Dürr)

Dernière mise à jour : 29 mai 2015

Table des matières

Chapitre 1

Introduction

1.1 Les concours de programmation

L'un des objectifs affichés de ce cours par le passé était de préparer les élèves à des concours de programmation. Mais pour être honnête, ce n'était qu'un prétexte pour leur donner une culture algorithmique et de bonnes pratiques de programmation. C'est pourquoi nous avons maintenant supprimé l'aspect "concours de programmation" du cours, le laissant au binet ACM. Toutefois, il est bon de rappeler comment fonctionnent ces concours en introduction, afin de vous donner une idée de la nature des exercices que nous considérerons tout au long du cours.

ACM est une société savante d'informatique (Association for Computing Machinery) et elle organise le concours ICPC (International Collegiate Programming Contest). Il y a d'abord une sélection par régions. Le concours de la région du sud-ouest de l'Europe s'appelle le concours régional SWERC, qui par exemple en novembre 2013 s'est tenu à Valencia.

Le concours s'organise par équipes de 3 personnes, un seul ordinateur, non connecté à Internet. On dispose de 5 heures pour résoudre un nombre maximum parmi 10 problèmes. Le premier critère de classement est le nombre de solutions soumises et acceptées (testées contre des jeux de test inconnus). Le deuxième critère est la somme pour chaque problème du temps écoulé entre le début du concours et le moment de la soumission acceptée. Pour chaque soumission erronée une pénalité de 20 minutes est ajoutée.

Le temps sur l'ordinateur est précieux, il faut arriver à écrire rapidement des programmes corrects, on n'a pas trop de temps pour les déboguer. Il faut que tout soit clair sur papier avant de saisir. Il y a plusieurs théories sur comment est formée une bonne équipe, en général il faut un bon codeur, un bon algorithmicien, et un bon débogueur. Et il faut bien s'entendre, même en situation de stress. Lors du concours on peut apporter 25 pages de code imprimé en 8 points pour référence. On dispose également de la document en ligne de l'API de Java et de STL de C++.

Il y a bien sûr plusieurs autres concours de programmation à part ICPC, et énormément de sites qui contiennent des annales. Le concours Google Code Jam a la particularité, qu'on peut programmer dans le langage de son choix, et *python* est très adapté pour ce type de programmation. Aussi c'est facile de participer, les premières étapes sont effectuées en individuel, devant une page web, et sans limite d'âge. Il y a aussi l'olympiade d'informatique *IOI*, petite soeur de l'olympiade de mathématiques, et le site *TopCoder*.

De manière générale il est intéressant de regarder des problèmes issus de ces concours, car c'est tout a fait le genre de problèmes que l'on retrouve dans la vraie vie. De plus, c'est le type de questions posées lors des entretiens d'embauche dans les entreprises high-tech, en particulier celles de la Sillicon Valley.

Les sites contenant des annales du concours ACM/ICPC sont classiquement *uva.onlinejudge.org* et *icpc.baylor.edu*. On a beaucoup de problèmes avec ces sites, notamment quand on soumet des solutions en Java, elles sont compilées avec une vieille version de *javac* qui rend la programmation pénible. On préfèrera nettement des sites comme *www.spoj.com* (Sphere Online Judge), qui

acceptent un grand nombre de langages et dont sont tirés la plupart des énoncés des TDs.
Les difficultés sont multiples dans ces problèmes. Découvrir la structure du problème, trouver le meilleur algorithme, connaître les astuces d'implémentations qui font la différence et aussi savoir organiser son programme de manière simple. L'exercice CheckTheCheck est très bon pour cela. Si vous commencez à écrire plein de traitements de cas, vous serez parti pour une séance de débogage longue.
En pratique, quand on soumet son code à un juge en ligne, ce dernier le teste sur des jeux de données qu'il ne divulgue pas et renvoie simplement l'une des réponses suivantes :

ACCEPT rien à dire.

PRESENTATION ERROR C'est presque un programme accepté, mais vous affichez des blancs ou retours chariot en trop ou en moins.

WRONG ANSWER Relisez l'énoncé, un détail a du vous échapper. Êtes-vous sûr d'avoir testé tous les cas limites ?

TIME LIMIT EXCEEDED Probablement vous n'avez pas implémenté l'algorithme le plus efficace pour ce problème, ou alors vous avez une boucle infinie quelque part.

COMPILATION ERROR Ah, c'est une erreur embêtante. Assurez-vous de bien compiler localement avec les mêmes options de compilation. Certains juges vous permettent en cliquant sur l'erreur d'avoir plus de détail.

RUNTIME ERROR En général, il s'agit soit d'une division par zéro, soit d'un dépassement de limite de tableau, d'un pop() sur une pile vide ou de l'appel d'une méthode depuis null. $try\{\ldots\}catch(\ldots)$ est très utile au débogage dans cette situation.

C'est malheureusement une très mauvaise manière de concevoir les choses. En effet, un débogage efficace suppose l'accès aux données sur lesquelles le programme plante. Dans la vraie vie, quand on développe un logiciel, on a une communauté d'utilisateurs qui fait remonter les problèmes avec des jeux de données élémentaires associés. C'est donc un non-sens de procéder comme décrit plus haut, mais c'est le jeu des concours de programmation. En pratique, on remarque que beaucoup d'étudiants se cassent les dents pour faire accepter leur code au juge parce qu'ils ne parviennent pas a trouver l'instance (non divulguée par le juge et correspondant souvent à un cas limite) sur laquelle leur code a planté. Pour éviter ce problème nous avons décidé de ne plus utiliser de juges en ligne et d'évaluer les codes des élèves sur des jeux de tests directement. Ainsi les élèves ont accès aux données et peuvent déboguer leurs codes plus efficacement. Ceci étant, nous fournissons aussi des liens vers les énoncés en ligne pour ceux d'entre vous qui désirent s'amuser (ou pas) en se frottant aux juges en ligne.

1.2 Le temps de calcul

En général vos programmes doivent résoudre chaque instance donnée en 2 secondes. En gros vous avez quelques millions d'opérations à votre disposition. Si la taille de l'entrée peut être 1.000.000 vous devez trouver un algorithme linéaire, où en $O(n \log n)$ à la limite. Un algorithme quadratique supporte des entrées de taille 1000 et un algorithme en $O(n^4)$ des entrées de taille 50. Ai-je mentionné que cette règle est grossière ?

Exercice écrivez un programme qui fait N divisions entières et testez combien vous pouvez en faire en 2 secondes. Faites-vous alors un tableau pour les limites N acceptables pour un algorithme dont la complexité serait $O(N)$, $O(N \log N)$, $O(N^2)$, $O(N^3)$, $O(N^4)$. En général, à l'aide des limites sur la taille de l'entrée données dans l'énoncé, on peut déjà avoir une première idée de l'algorithme à trouver.

1.2.1 La complexité des fonctions de la bibliothèque

Les méthodes de la JDK sont assez bien documentées mais malheureusement pas leur complexité. Voici un test effectué sur la recherche d'une sous-chaîne. Chercher si une chaîne aiguille de longueur m est une sous-chaîne dans une chaîne botteDeFoin de longueur n, prend un temps $O(nm)$ par un test naïf. Un meilleur algorithme est celui de Knuth-Morris-Pratt, qui fonctionne en O(n+m). Nous avons fait un test de vitesse pour savoir quel algorithme est implémenté dans la bibliothèque. Pour différentes valeurs de $n = 100, 200, \ldots, 20000$, nous avons cherché la sous-chaîne $a^{n/2}b$ dans la chaîne a^n, et mesuré le temps pour le faire. Dans la figure 1.1, vous voyez avec des «+» le temps pris par la recherche naïve, que nous avons implémentée et exécutée avec OpenJDK 1.6.0. La méthode indexOf de la classe String (notée par "×") est meilleure par une constante, ce qui est dû à l'implémentation de notre méthode naïve. En dessous avec des «*» le temps pris par la fonction strstr de la bibliothèque standard C. C'est elle qui semble implémenter l'algorithme de Knuth-Morris-Pratt. C'est choquant n'est-ce pas ? Nous avons fait un test similaire avec le compilateur gcj 4.2.4. La recherche naïve y est à la même vitesse qu'avec OpenJDK. Mais ce qui est étonnant c'est que l'exécution de indexOf est 5 fois plus lente, donc la bibliothèque gcj n'est pas très optimisée.[1]

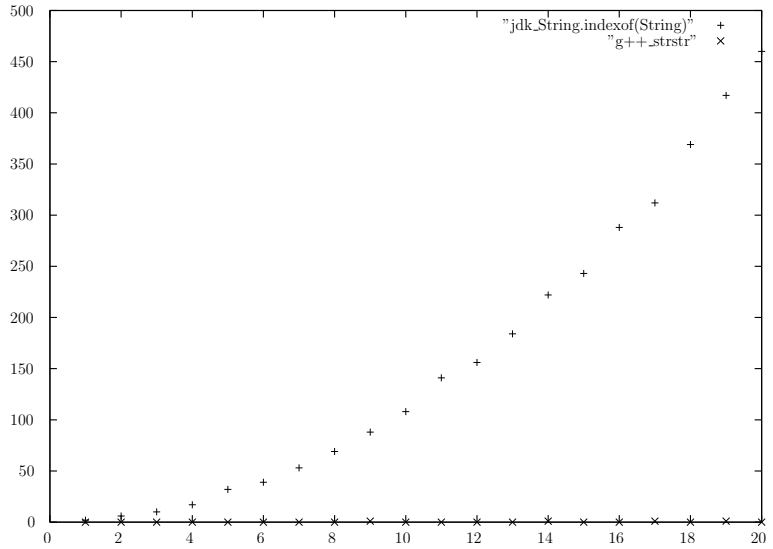


FIGURE 1.1 – tests de vitesse pour la recherche d'une sous-chaîne

1.3 Conseils personnels

Soyez organisés et systématiques. Séparez dans votre programme lecture de l'instance avec le calcul de la solution. Précisez en commentaire le nom du problème, idéalement avec son URL, et donnez la complexité de votre algorithme.

Bien lire l'énoncé

Quelle complexité est acceptable ? Analysez votre algorithme avant de l'implémenter.

Quelles promesse sur les entrées ? Ne déduisez pas des promesses à partir de l'exemple. Ne supposez rien : s'il n'est pas dit que le graphe est non-vide, il y aura probablement une instance avec un graphe vide, s'il n'est pas dit qu'une chaîne de caractère ne contient pas d'espace, attendez-vous à une instance avec une telle chaîne, etc.

[1] Cette expérience a été confirmé pour Oracle JDK 1.6.0_31.

Quel type de nombre choisir ? Entier ou flottant ? Est-ce que les nombres peuvent être négatifs ? Quelle est la borne sur la plus grande valeur intermédiaire dans vos calculs ? En conclusion faut-il int, long, double ou BigInteger ?

Bien planifier

Comprenez l'exemple. Faites des schémas. Découvrez des liens avec des problèmes connus. Comment exploiter les particularités des instances ?

Quand c'est possible utilisez la bibliothèque. Maîtrisez les arbres de recherche, le tri, les dictionnaires.

Séparez le programme en lecture, calcul, affichage. Les programmes des équipes de Varsovie sont toujours composés de trois fonctions.

Utilisez les mêmes noms de variables que dans l'énoncé. et de préférence des noms courts, mais parlant.

Déboguer

Faites des erreurs maintenant pour avoir les bons réflexes plus tard.

Produisez des jeux de tests pour les cas limites (wrong answer) et pour les grands instances (time limit ou runtime error).

Expliquer l'algorithme et commenter le programme à un co-équipier.

Soyez capable d'expliquer chaque ligne. N'écrivez rien que vous ne comprenez pas complètement.

Simplifiez votre implémentation. regroupez du code similaire

Prenez de la distance, en passant à un autre problème pour revenir avec un regard neuf.

1.4 Eclipse

Le concours met entre autres Eclipse et Emacs à votre disposition. Si jamais vous vous décidez pour Eclipse, voici quelques conseils.
Créez une fois pour toutes un *workspace*. C'est un répertoire où Eclipse va stocker ses paramètres et en général les programmes. Puis pour chaque programme, vous allez créer un projet. Vous pouvez soit laisser Eclipse créer le répertoire source dans le workspace, soit lui indiquer un répertoire que vous souhaitez utiliser et que vous aurez crée préalablement.
Si Eclipse ne connaît pas la classe Scanner, c'est probablement qu'il n'est pas configuré pour le bon environnement Java. Dans une console tapez `locate javac`, et notez le répertoire de la distribution Java par Sun, comme par exemple `/usr/lib/jvm/java-1.5.0-sun-1.5.0.15/bin`. Ensuite dans **Windows>Preferences>Java>Installed JREs**, ajoutez ce répertoire et cochez le.
Il est souvent plus pratique d'exécuter votre programme à partir de la console, pour passer un fichier en entrée, ou inversement enregistrer la sortie.

1.5 Afficher des messages pour déboguer

Pour déboguer votre programme, vous voudriez par fois afficher le contenu de certaines variables à différents endroits du code. Vous pouvez alors afficher sur la sortie erreur standard, qui est ignoré par le juge. C'est-à-dire que vous affichez avec *System.err.println*, au lieu de *System.out.println*. Attention néanmoins, l'affichage ralentit le code. Pensez à supprimer ou commenter ces affichages une fois devenus inutiles. Prenez aussi garde au fait que l'ordre

programmé de deux messages envoyés sur deux flux différents ne sera pas forcément respecté en pratique, selon quel programme en assure la gestion.

1.6 Lire l'entrée

On va utiliser la classe Scanner pour lire depuis l'entrée standard. On l'initialise avec Scanner in = new Scanner(System.in) ; in.useLocale(Locale.US). L'appel de useLocale est nécessaire pour que Scanner lise les flottants avec un point décimal et non une virgule, si jamais la machine est installée en français.

- in.hasNext() permet de savoir si la fin de l'entrée n'est pas encore atteinte (EOF).
- in.nextLine() lit toute une ligne et la retourne sans le retour chariot à la fin.
- in.next() lit un *mot* : une chaîne délimitée par des caractères blancs (espace, retour chariot, etc). Cette fonction ignore les caractères blancs initiaux.
- in.nextInt(), in.nextDouble() comme la fonction précédente, mais interprète le mot comme un nombre.

Si on veut ensuite de nouveau décomposer une ligne qu'on vient de lire par nextLine, alors il suffira de déclarer un nouveau scanner Scanner sin(s) ; et de lire du flux sin. Ceci est un peu mieux que de déclarer un StringTokenizer sur s, car Scanner permet de convertir en entiers et flottants.

1.7 Exemple

On veut lire une entrée qui est composée de plusieurs jeux de test. Chaque jeu de test commence avec deux entiers positifs n et m dans une seule ligne. Puis suivent n lignes de m caractères chacune, où les caractères sont soit ' ' soit '#'. Le dernier jeu de test est suivi de 00 dans une seule ligne.

```
int n, m;
Scanner in = new Scanner(System.in);
while (true) {
    n = in.nextInt();
    m = in.nextInt();
    if (n==0)
        return;
    in.nextLine(); // important: consommer le '\n' apres n m
    char[][] tab = new char[n][];
    for (int i=0; i<n; i++)
        tab[i] = in.nextLine().toCharArray();
    // traiter tab
    [...]
}
```

1.8 Lire des lignes

Dans certains cas, il faut lire des lignes entières dans une variable. Par exemple, dans l'exemple suivant on lit un graphe codé de la manière suivante : la première ligne contient un entier n qui est son nombre de sommets. Ensuite suivent n lignes, la i-ème ligne (en comptant de 0 à $n-1$) contient la liste des sommets j, tel qu'il existe un arc de i à j. Le piège avec ce format est qu'après lecture de n, la tête de lecture se trouve encore sur le retour charriot juste après le n. Il faut donc une lecture, pour consommer ce caractère.

```
class Liste extends ArrayList<Integer> {}
// ...{

class ReadLine {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        int n = in.nextInt();
        in.nextLine(); // consommer fin de ligne
        Liste[] A = new Liste[n];
        for (int i=0; i<n; i++) {
            A[i] = new Liste(); // creer nouvelle liste d'adj pour i
            Scanner l = new Scanner(in.nextLine());
            while (l.hasNext()) { // extraire voisins de la ligne
                int j = l.nextInt();
                A[i].add(j);
            }
        }
        // -- affichage des arcs
        for (int i=0; i<n; i++)
            for (Integer j: A[i])
                System.out.println(i+" -> "+j);
```

1.9 Quand la classe Scanner est trop lente

Le juge du site acmicpc-live-archive.uva.es utilise malheureusement une ancienne version de Java qui ne connait pas la classe Scanner. Parfois aussi quand il s'agit de lire plus que 10.000 nombres, la classe Scanner peut s'avérer trop lente.

Une solution est alors d'utiliser la classe BufferedReader, en combinaison avec *StringTokenizer* quand plusieurs nombres se trouvent sur une même ligne. Le code suivant illustre l'utilisation de ces classes.

```
    public static void main(String args[]) throws IOException {
        BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
        double bestX=-1, bestY=-1;
        int N = parseInt(in.readLine());
        while (N-->0) {
            StringTokenizer st = new StringTokenizer(in.readLine());
            double x = parseDouble(st.nextToken()), y = parseDouble(st.nextToken());
            if (x>bestX || x==bestX && y>bestY) {
                bestX = x;
                bestY = y;
            }
        }
        System.out.println(bestX+" "+bestY);
    }
```

Nous avons fait une expérience avec une lecture d'un fichier d'un million de lignes sur le code ci-haut. L'amélioration est dramatique pour la lecture de flottants, voir figure 1.2.

	Scanner	BufferedReader
int	0,6 sec	0,3 sec
double	9,7 sec	0,9 sec

FIGURE 1.2 – temps pour lire un million de nombres, en fonction du type et de la classe utilisée

1.10 Précision

Normalement on utilise int pour stocker un entier. Mais dans certains cas extrêmes ont veut utiliser un entier qui utilise moins d'espace ou qui a une capacité plus grande. Si même long ne vous suffit pas, alors utilisez double si la précision n'est pas importante, soit java.math.BigInteger. Les constantes de type long se notent avec un L à la fin, comme dans 922337203685477580 7L. Les constantes limites MIN_VALUE, MAX_VALUE sont définies dans les classes Integer,Long et Double.

type	nb octets	MIN_VALUE	MAX_VALUE
byte	1	-128	127
int	4	$-2^{31} \approx -2 \cdot 10^9$	$2^{31} - 1 \approx 2 \cdot 10^9$
long	8	$-2^{63} \approx -8 \cdot 10^{18}$	$2^{63} - 1 \approx 8 \cdot 10^{18}$
double	8	suffisant	suffisant

En C++ un bool a la même taille qu'un int et pas 1 octet comme on pourrait penser. En Java cela dépend de l'implémentation, il faut faire l'essai.
En python les entiers sont à précision arbitraire.

Chapitre 2

Informations pratiques

2.1 Les chaînes de caractères

Pour ajouter du texte à la fin d'une chaîne de manière répétée, il vaut mieux utiliser StringBuilder (ou StringBuffer) plutôt que String. Le code suivant va faire $O(m^2)$ opérations, car une nouvelle chaîne est à chaque fois créée à partir de l'ancienne, dont le contenu doit être recopié.

```
String s = "";
for (int i=0; i<m; i++)
    s += "*";
```

La classe StringBuilder se comporte comme la classe ArrayList<Character> (ou Vector<Character>), et le code suivant ne fera que $O(m)$ opérations. StringBuilder travaille en fait avec un tampon : quand ce dernier devient trop petit, un nouveau tampon deux fois plus grand est alloué, et le contenu y est recopié.

```
StringBuilder s = new StringBuilder();
for (int i=0; i<m; i++)
    s.append("*");
```

2.1.1 Assert

L'utilisation d'assert est recommandé à chaque fois que vous faites une hypothèse sur le domaine d'une variable. Par exemple si la fonction move(char c) n'est censée recevoir que les valeurs 'p','P','n','N' pour c vous pourrez écrire l'instruction assert "pPnN".contains(""+c); qui permet d'effectuer ce test et d'arrêter le programme avec un message parlant en cas d'échec — ce qui est préférable à un comportement non prévu. Pour utiliser cette fonction exécutez avec l'option -ea comme dans java -ea MaClasse. Attention : le juge du site *acm.zju.cn.edu* donne une erreur *Runtime*, quand on utilise *assert*, même si l'instruction n'est jamais exécutée.
En général ne faites aucune hypothèse sur l'entrée hormi ce qui est mentionné dans l'énoncé du problème. Votre programme doit pouvoir traiter des graphes vides, des mots vides, etc.
Maintenant si la condition testée par la commande assert n'est pas satisfaite lors d'une exécution sur le juge, vous ne le saurez jamais, car le juge ne vous montre pas l'affichage produit. Vous pourrez donc utiliser cette méthode de remplacement.

```
void myAssert(boolean expr) {
    if (!expr) {
        // provoquer time-limit exception,
        while (true) {}
        /* ou au choix :
            int i = 1/(int)expr; // div par 0 = runtime error
```

```
            System.out.print("blabla");System.exit(0); // wrong answer
        */
    }
}
```

2.2 Variables

Variables globales On vous a toujours appris qu'il fallait éviter d'utiliser des variables globales, pour éviter les effets de bords des méthodes. Mais l'utilisation de variables globales simplifie les signatures et les appels de fonctions. Ceci se défend quand ces fonctions dépendent de tout un environnement, par exemple d'un graphe ou d'un dictionnaire de mots. Dans ce cas il faut faire attention à initialiser correctement ces variables globales avant le traitement de chaque nouvelle instance.

Noms de variables Les noms de variables devraient à la fois refléter le type et la signification. Par exemple i est généralement accepté comme un indice entier. Ou nbTours est plus parlant que n. Par contre des noms longs, rendent le programme long à taper et à lire. Mais vous pouvez vous aider de la fonction qui complète un début de nom, M-\ dans Emacs ou ^N dans vim. Soyez consistant avec les noms même entre des programmes différents, par exemple utilisez toujours V pour la variables contenant les sommets. Ainsi vous pourrez réutiliser plus rapidement votre code.

ArrayList En remplacement des tableaux en Java il y a la classe générique ArrayList. Par exemple int[] tab peut être remplacé par ArrayList<Integer> tab. L'avantage de cette classe : on peut varier la taille du tableau, la méthode push_back permet d'ajouter un nouvel élément en fin de tableau. En interne la classe travaille avec un tableau d'une taille qui est une puissance de 2. Quand celle-ci est trop petite un deuxième tableau deux fois plus grand est alloué, et les données y sont recopiées. Donc n ajouts en fin de tableaux sur un vecteur initialement vide coûtent un temps $O(n)$. Il y a une autre classe similaire Vector, plus ancienne, qui est plus lente d'après la documentation et qu'on ne devrait pas utiliser.

Dictionnaires Parfois on veut avoir un tableau dont les indices ne sont pas des entiers mais par exemple des chaînes des caractères. La classe HashMap permet cela. Par exemple HashMap<String,Integer> tab déclare une classe dont les indices sont des chaînes et les valeurs des entiers. L'expression tab.get("aha") retourne la valeur associée à la clé "aha", qui est automatiquement converti de Integer en int. Par contre si cette clé est nouvelle, null est retourné.

des éléments complexes Quand les valeurs d'un dictionnaire sont d'un type plus complexe, comme `ArrayList<Integer>`, par exemple, on a un problème, car dans une classe générique comme *TreeMap*, le type des valeurs ne peut pas être un type générique. Il faut alors définir une classe qui hérite de ce type générique. Un exemple est donné dans le code suivant.

2.3 Anagrammes

On veut grouper tous les mots qui sont anagramme l'un de l'autre. Pour cela on calcule une signature pour chaque mot, tel que deux mots sont anagrammes l'un de l'autre si et seulement si ils ont la même signature. Cette signature est tout simplement un autre mot composé des mêmes lettres, mais ordonnées.

```
class ListeMots extends LinkedList<String> {}
```

```
class Anagramme {
    public static void main(String args[]) {
        Scanner in = new Scanner(System.in);

        TreeMap<String,ListeMots> dict = new TreeMap<String,ListeMots>();
        // --- lire les mots
        while (in.hasNext()) {
            String w = in.next();
            char c[] = w.toCharArray(); // calculer signature "caba"->"aabc"
            Arrays.sort(c);
            String sig = new String(c);
            if (dict.get(sig)==null) // nouveau groupe ?
                dict.put(sig, new ListeMots());
            dict.get(sig).add(w); // ajouter w a ce groupe
        }

        // ---- afficher les anagrames
        for(String sig: dict.keySet())
            if (dict.get(sig).size()>1) { // ignore mots sans anagramme
                for (String w: dict.get(sig))
                    System.out.print(w+" ");
                System.out.println();
            }
    }
}
```

Problèmes Anagram Groups [poj :2408], T9 [poj :1451]

2.4 Parcourir

Un conteneur (liste ou vecteur par exemple) peut être parcouru par une boucle for de la forme for(int i : L), où int est le type des éléments dans L (peut être un autre type bien sûr), et L est le conteneur, qui peut-être un ensemble, une liste, un tableau, etc.
Pour boucler sur les clés d'un dictionnaire, on procède ainsi :

```
        Map<String,Integer> tab = new HashMap<String,Integer>();

        [...]
        for(String s: tab.keySet())
            System.out.println("tab["+s+"]="+tab.get(s));
```

2.5 Trier

Pour trier un tableau vous disposez de la fonction Arrays.sort et pour trier un Vecteur utilisez Collections.sort. Dans les deux cas, il faut que les éléments dans le tableau implémentent l'interface Comparable. En particulier on peut trier un tableau de chaînes, de flottants, etc. Pour les autres classes il y a deux possibilités.

2.5.1 Trier des objets comparables

L'objet T doit implémenter l'interface `Comparable<T>` et alors disposer d'une méthode compareTo, qui retourne un entier négatif, zéro ou positif, suivant le résultat de la comparaison.

Notez l'utilisation de *signum*, plutôt qu'une simple conversion vers *int*, qui aura le risque de produire 0 pour des valeurs proche de 0.

```
class Ville implements Comparable<Ville> {
    int i;
    double x,y
    Ville (int _i, double _x, double _y) {
        i = _i;
        x = _x;
        y = _y;
    }

    public int compareTo(Ville v) {
        return (int)Math.signum(Math.hypot(x,y)−Math.hypot(v.x,v.y));
    }
}

//[...]
            Ville[] tab = new Ville[n];
//[...]
// les villes vont etre triees en fonction de leur distance au centre
            Arrays.sort(tab);
```

2.5.2 Trier des objets avec un comparateur

On peut aussi fournir une classe qui compare, comme dans cet exemple. Ceci a l'avantage qu'on peut effectuer deux tris sur deux critères différents.

```
class Point {
    int x,y;
    Point(int _x, int _y) {x=_x; y=_y;}
    public String toString() {return "("+x+","+y+")";}
}

class TestSort2 {

    public static void main(String[] args) {
        Vector<Point> v=new Vector<Point>();

        // lire v
        [...]

        Vector<Point> X=new Vector<Point>(v);
        Vector<Point> Y=new Vector<Point>(v);

        // trier X lexigraphiquement par (x,y)
        Collections.sort(X, new Comparator<Point>() {
                public int compare(Point p, Point q) {
                    if (p.x!=q.x) return p.x−q.x;
                    else return p.y−q.y;
                }
            });

        // trier Y lexigraphiquement par (y,x)
```

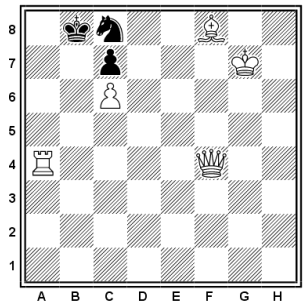


FIGURE 2.1 – Étant donnée un échiquier décrit par 8×8 caractères, savez vous déterminer si les blancs peuvent faire un échec et mat ?

```
Collections.sort(Y, new Comparator<Point>() {
        public int compare(Point p, Point q) {
            if (p.y!=q.y) return p.y−q.y;
            else return p.x−q.x;
        }
    });
```

Nous avons aussi fait un test d'un tri alternatif, qui utilise un TreeSet<String,Point>X. Pour chaque point p de v, on construit une chaîne s qui est composée des coordonnées (x,y) de p, en représentant chaque nombre avec le même nombre de caractères, de telle sorte que l'ordre lexicographique sur ces chaînes correspond sur l'ordre lexicographique sur les coordonnées. Ensuite on peut prendre les valeurs du dictionnaire dans l'ordre et les ajouter à X, et procéder pareil pour Y. Le code est un peu plus court, car on peut se passer des comparateurs, mais il est 5 fois plus lent.

2.6 Jeux de cartes

Voici un problème qui revient tout le temps, surtout dans des jeux. Imaginons un jeu de cartes, et dans l'entrée les cartes sont spécifiées avec deux caractères. Le premier donne le rang et le deuxième la couleur, par exemple 2D pour le rang 2 de la couleur carreau (D comme *diamond*). Mais en interne on préfère stocker rang et couleur comme des entiers, par exemple pour pouvoir calculer plus facilement la valeur d'une carte. Par exemple si les couleurs sont codées par les caractères 'C','D','H' et 'S' auquel on voudra attribuer les entiers de 0 à 3. On pourra faire la conversion avec l'instruction int i= "CDHS".indexOf(c), et l'inverse avec char c="CDHS".charAt(i).

2.7 Échecs

Pour simuler les mouvements des différents pions d'un échiquier on voudra éviter d'avoir un bout de code dédié à chaque type de pion, car ceci rendrait le programme long. On pourra se servir d'un tableau de directions dx,dy pour chaque type de pion, de telle sorte que le pion de type t positionné en (x, y) peut se déplacer en $(x + k \cdot \mathrm{dx}[t][i], y + k \cdot \mathrm{dy}[t][i])$ pour tout $i \geq 0$ tel que $\mathrm{dx}[t][i] \neq 0$ ou $\mathrm{dy}[t][i] \neq 0$ et tout $k \geqslant 1$ approprié.

2.8 Tableaux

La classe Arrays du package java.util contient des fonctions utiles pour manipuler des tableaux. Par exemple Arrays.fill pour affecter une valeur donnée à tous les éléments, ça évite d'avoir à

écrire une boucle. Aussi pour déboguer vous aurez envie d'utiliser Arrays.toString.

2.9 Coder un ensemble dans un entier

On peut utiliser les propriétés de la représentation binaire pour représenter des ensembles dans des entiers, pourvu que l'univers ne dépasse pas, disons 20 éléments. On associe à chaque entier de l'univers alors une position dans la représentation binaire de l'entier. Les opération d'intersection, d'union, etc., se traduisent alors naturellement en opérations binaires sur les entiers.

codage	interprétation
`1<<i`	$\{i\}$
`1<<n-1`	$\{0, 1, \ldots, n-1\}$
`s|t`	$S \cup T$, si s, t encodent les ensembles S, T
`s&t`	$S \cap T$
`s^t`	différence symmétrique entre S et T

```
static int card(int s) { // cardinalite
    int r=0;
    while (s!=0) {
        s = s&(s−1); // enleve le bit de poids plus faible
        r++;
    }
    return r;
}
```

Problèmes The Fewest Coins [poj :3260]

Chapitre 3

L'exploration exhaustive

3.1 L'exploration exhaustive

Pour certains problèmes il ne faut pas chercher un algorithme qui fonctionnera en temps polynomial dans le pire des cas, et résoudre ces problèmes avec une recherche exhaustive avec backtracking. Exemples :

Le problème des n reines Dans une échiquier $n \times n$ il faut placer n reines de telle sorte qu'elles soient toutes dans des colonnes, lignes, diagonales et anti-diagonales distinctes.

Décoder un texte chiffré avec un code à substitution Un code à substitution est une permutation sur les lettres de l'alphabet. Étant donné un texte chiffré, il faut retrouver le texte clair. On a la promesse que tous les mots clairs sont des mots d'un dictionnaire, qui est fourni lui-aussi.

Labyrinthe de miroirs Dans une grille il y a trois types de cases. Des murs absorbants, des miroirs réfléchissants, et des cases vides. On nous donne une configuration d'une grille, avec des murs sur les bords, sauf à deux endroits. Les miroirs peuvent en position "/" ou en position "\", et refléchissent une lumière horizontale de manière verticale et vice-versa. Il faut trouver une position pour les miroirs tel qu'un laser qui entre par un des trous du bord, ressortira par l'autre.

Sudoku Compléter une grille, tel que dans chaque ligne, chaque colonne et chaque bloc, chaque valeur soit représentée exactement une fois.

Le principe de l'exploration exhaustive est de parcourir en profondeur un arbre de toutes les possibilités, en rebroussant chemin quand on découvre qu'il n'y a pas de solution dans le sous-arbre. Concrètement sur l'exemple de Sudoku d'une grille 16×16 la recherche pourrait avoir la forme suivante.

```
// la matrice prise ligne par ligne
static char M[] = new char[81];

// N[p] contient la liste des cellules avec lesquelles p est en conflit
static int N[][] = new int[81][9+9+9-6];

static boolean solve(int p) {
    if (p>=81)
        return true;
    if (M[p] != '-')
        return solve(p+1);
    for (char v = '1'; v<='9'; v++) { // essayer toutes les val. poss.
```

```
            boolean ok=true;
            for (int q: N[p]) // est−ce qu'il y a conflit avec un voisin ?
                if (M[q]==v) {
                    ok = false;
                    break;
                }
            if (ok) {
                M[p] = v;
                if (solve(p+1)) // descente recursive
                    return true;
                M[p] = '−';
            }
        }
        return false;
    }
```

Pour pouvoir tester simplement si une case donnée peut accueillir une valeur donnée, nous avons construit un tableau N, tel que $N[p]$ contienne toutes les cases avec lesquelles p sera en conflit. Imaginez le problème Sudoku comme un problème de coloration de sommets d'un graphe, où N sera la liste d'adjacence, et deux sommets voisins doivent avoir des couleurs différents.
L'initialisation de N se fait comme suit.

```
        // initialiser N
        for(int p=0; p<81; p++) {
            int n=0;
            for (int q=0; q<81; q++) {
                int rp = p/9, rq = q/9; // ligne
                int cp = p%9, cq = q%9; // colonne
                int bp = 3*(rp/3)+cp/3; // bloc
                int bq = 3*(rq/3)+cq/3;
                if (p!=q && (rp==rq || cp==cq || bp==bq))
                    N[p][n++] = q;
            }
        }
```

Problèmes Crypt Kicker [UVA :843], Mirror Maze [Baylor :258]

3.2 Améliorations

On peut facilement produire une instance où le programme de résolution Sudoku ci-dessus sera trop lent. Comme il explore toutes les celulles dans un ordre fixé et toutes les valeurs dans un ordre fixé, si jamais dans la solution la première ligne consiste en les valeurs 987654321, alors le programme va en fait simuler un compteur sur cette ligne et nécessiter 10^9 itérations. Plusieurs solutions sont envisagables.

- Sélectionner un ordre aléatoire sur les cellules et parcourir les cellules dans cet ordre. Ceci va vous protéger des instances pire des cas avec grande probabilité.

- Brancher toujours sur la cellule avec un nombre minimum de valeurs possibles. Ainsi si la solution actuelle ne peut pas être complétée, on va construire un arbre petit, et se rendre compte rapidement de la situation. Ceci nécessite de maintenir un ensemble de valeurs admissibles par cellule, et de mettre à jour les celulles voisines lors d'une affectation.

3.3 Toutes les permutations

Parfois on veut tester tous les $n!$ permutations d'un tableau de taille n, pour trouver une solution. En C++ vous pouvez utiliser la fonction standard *next_permutation* pour itérer sur toutes les permutations. En Java ou en python il faut écrire une petite fonction, ce qui reste très simple.

```
permutation("", str);
{
    private static void permutation(String prefix, String str) {
        int n = str.length();
        if (n == 0) {
        if (test(prefix))
            return; // on a trouve une solution
        }
        else {
        for (int i = 0; i < n; i++)
            permutation(prefix + str.charAt(i), str.substring(0, i) + str.substring(i+1, n));
        }
    }
```

Chapitre 4

Les algorithmes gloutons

4.1 L'arbre de recouvrement minimal

Soit $G = (V, E)$ un graphe connecté non-dirigé. Une *forêt* dans G est un sous-graphe $F = (V, E')$ sans cycle. Un *arbre de recouvrement* est une forêt de G avec exactement une composante connexe. Étant données des poids $w : E \to \mathbb{N}$ un *arbre de recouvrement minimum* est un arbre de recouvrement dont le poids total des arêtes est minimum. Le problème MST consiste à trouver un tel arbre. Voici un algorithme découvert par Kruskal en 1956.
Algorithme glouton pour MST.

1. trier les arêtes par poids croissants.
2. Pour chaque arête de la liste en ordre croissant, l'ajouter à la solution, si elle ne forme pas un cycle avec les arêtes déjà ajoutées, sinon l'ignorer.

L'algorithme peut être arrêté dès que $n - 1$ arêtes ont été retenues, à cause du lemme suivant.

Lemme 1 *Soient $F = (V, E)$ un graphe non-dirigé, c son nombre de composantes connexes, $n = |V|$ et $m = |E|$. Alors F n'a pas de cycles si et seulement si $c + m = n$.*

La première partie coûte un temps $O(m \log m) = O(m \log n)$ sauf dans le cas particulier où les poids sont bornés par m, au quel cas on peut utiliser le tri par paniers en $O(m)$.
La deuxième partie est presque linéaire en n, si on utilise la structure de données union-find, mais pour l'instant voyons comment réaliser cette partie en $O(n \log n)$ étapes. On représente les composantes connexes par des listes chaînées. Chaque élément comporte un pointeur sur l'élément suivant et aussi sur la tête de la liste. Quand on veut savoir si deux éléments sont dans la même composante, il suffit de comparer leur tête de liste. Pour réunir deux composantes, il faut concaténer les listes, et alors il faut mettre à jour le pointeur sur la tête de liste de tous les éléments de la liste qu'on concatène. On a alors intérêt à concaténer toujours la liste la plus courte sur la plus longue, et pour cela il suffit de maintenir un compteur, qu'on stocke dans les têtes de liste. Chaque fois que le pointeur sur la tête de liste d'un élément est changé, cet élément se trouve dans une liste au moins deux fois plus grande. Le programme s'arrête quand une seule liste contient tous les sommets, on a alors effectué $O(\log n)$ opérations sur chaque élément.

4.2 Union-Find

Union-Find est une structure de données qui stocke un partitionnement d'un univers V, et qui permet les opérations suivants.

- find(v) retourne un élément canonique de l'ensemble contenant v. Pour tester si u et v sont dans le même ensemble il suffit de comparer find(u) avec find(v).

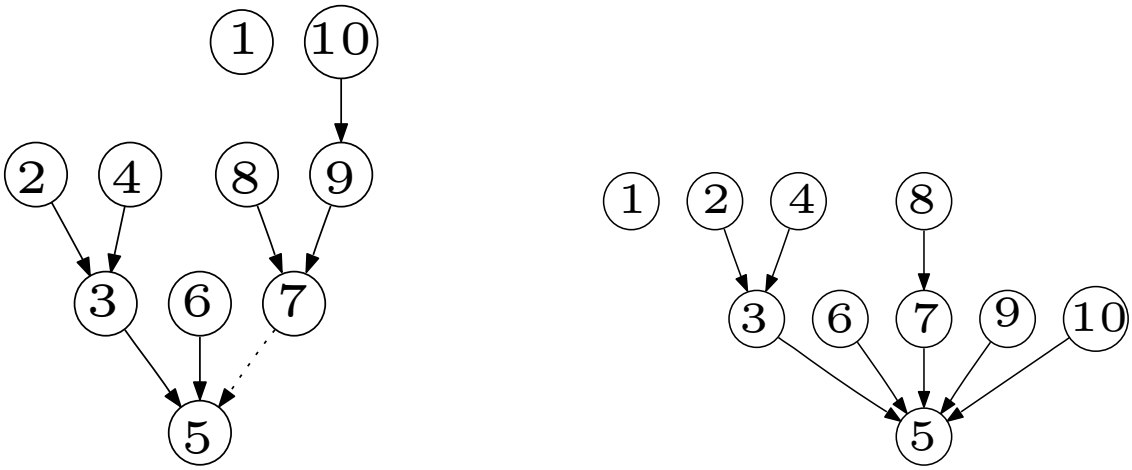


FIGURE 4.1 – Un graphe avec 3 composantes dont les éléments canoniques sont respectivement 1,5 et 7. Idée 1 : pour réunir deux composantes (ici 5 et 7), toujours attacher la plus petite à la plus grande. Idée 2 : lors du parcours d'un élément (ici 10) vers la racine, compresser le chemin.

- union(u,v) fusionne l'ensemble contenant u avec celui contenant v.

Nous organisons les éléments d'un ensemble en arbre dirigé, orienté vers un élément canonique. Chaque élément v a une référence parent[v] vers l'élément plus haut dans l'arbre. La racine v — l'élément canonique de l'ensemble — sera indiquée par une valeur spéciale dans parent[v] qu'on choisira 0, -1, où v lui-même, il suffit d'être consistant. Nous stockons aussi la taille de l'ensemble dans un tableau taille[v], où v est l'élément canonique. Il y a deux idées dans cette structure de données.

1. Lors de l'union on accroche le plus petit des arbres sous la racine du plus grand.
2. Lors du parcours d'un élément vers la racine, on profite pour compresser le chemin, c'est-à-dire on accroche tous ces éléments directement sous la racine.

Ceci nous donne :

```
int find(int v) {
  int pv = parent[v];
  // est-ce la racine ?
  if (pv==v)
    return v;
  return parent[v] =find(pv);
}

void union(int u, int v) {
  int ru = find(u);
  int rv =find(v);
  if (ru==rv) return; // meme composante, rien a faire
  if (size[ru]<size[rv]) {
    size[rv] += size[ru];
    parent[ru] = rv;
  }
  else {
    size[ru] += size[rv];
    parent[rv] = ru;
  }
```

```
}
```

C'est peut-être trop long à expliquer, mais n'importe quelle séquence de m opérations union ou find, sur un univers de taille n coûte un temps $O((m+n)\alpha(n))$, où α est l'inverse de la fonction de Ackerman, qui en pratique peut être considéré comme la constante 4.

Problèmes Networking [poj :1287]

4.3 Généralisation

Voici une approche plus générale qui englobe la plupart des algorithmes connus pour le problème d'arbre de recouvrement minimal.
Soit $G(V, E)$ un graphe connecté, non-dirigé avec une pondération des arêtes $w : E \to \mathbb{N}$.
Considérons les deux règles suivants pour colorer les arêtes, introduites par Tarjan en 1983.

Règle bleue Trouver une coupe (partition de V en deux ensembles disjoints X et $V - X$) qui n'est pas traversée par une arête bleue. Prendre une arête sans couleur de poids minimal traversant la coupe et la colorer en bleu.

Règle rouge Trouver un cycle (un chemin dans G qui commence et termine dans un même sommet) qui ne contient pas d'arête rouge. Prendre une arête sans couleur du cycle de poids maximal et la colorer en rouge.

L'algorithme glouton est juste une application répétée de la règle bleue, et sa correction est basée sur ce théorème.

Théorème 1 *Supposons qu'initialement toutes les arêtes soient sans couleur. Si les règles rouges et bleues sont appliquées dans un ordre arbitraire, jusqu'à ce qu'aucune ne puisse s'appliquer, alors l'ensemble final des arêtes bleues forme un arbre de recouvrement minimal.*

En fait le problème de l'arbre de recouvrement minimal se généralise en ce qui s'appelle *les matroïdes*. Ils forment exactement la classe des problèmes que l'algorithme glouton résout, voir cours INF550 "Conception et analyse des algorithmes".

Problèmes Minimal Coverage [uva :10020]

4.4 Tomographie discrète

Voici un problème qui peut être résolu par un algorithme glouton, mais pas dans le sens des matroïdes. Il s'agit de construire un graphe biparti $G(V, U, E)$ sous des contraintes de degrés données : On vous donne l'ensemble des sommets U, V et une spécification des degrés $d : U \cup V \to \mathbb{N}$. Le but est de trouver un ensemble d'arêtes $E \subseteq U \times V$ qui satisfait les degrés demandés. Formellement on peut décrire le problème comme suit.

Problème de Reconstruction de Matrices 0-1

Entrée $2n$ entiers $r_1, \ldots, r_n, c_1, \ldots, c_m$ pour $n \geqslant 1$

Sortie une matrice $M \in 0, 1^{n \times m}$ telle que pour tout $1 \leq i \leq n, 1 \leq j \leq m$,

$$r_i = \sum_{k=1}^{m} M_{\mathrm{ik}} \text{ et } c_j = \sum_{k=1}^{n} M_{\mathrm{kj}}.$$

Clairement la solution n'est pas forcément unique si elle existe, par exemple l'instance $1, \ldots, 1$ a $n!$ solutions, les matrices de permutations. Ce problème fut étudié en 1963 par Ryser qui donna une caractérisation exacte des projections admettant une solution. Il nous faut deux définitions, le *dual* et la *dominance* :
Soit $v \in \mathbb{N}^n$ un vecteur trié ($v_i \geq v_{i+1}$ pour tout $1 \leq i \leq n-1$) et satisfaisant $0 \leq v_i \leq n$ pour tout $1 \leq i \leq n$. Graphiquement on peut représenter un tel vecteur comme un tableau de n barres horizontales, alignées à gauche, et la i-ème barre est de longueur v_i, voir figure 4.2. Ce tableau peut aussi être vu comme l'union de n barres verticales, alignées en haut. La hauteur de la j-ème barre est alors $\bar{v}_j := |i : v_i \leq j|$. Le vecteur $\bar{v}$ est de nouveau un vecteur trié, et ses valeurs sont comprises entre 0 et n. Notons que si $u = \bar{v}$ alors $\bar{u} = v$, ceci motive le terme «*dual*».

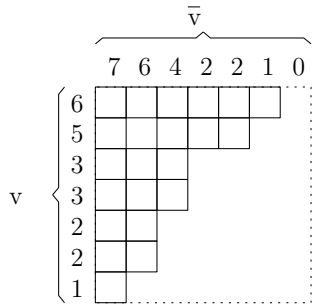


FIGURE 4.2 – Le dual d'une projection.

On définit un ordre partiel sur les vecteurs de même dimension, appelé *ordre de dominance*. Soient les vecteurs $u, v \in \mathbb{N}^n$. On dit que u domine v, noté $v \preceq u$, si pour tout $1 \leq i \leq n$ on a $\sum_{j=1}^{i} v_j \leq \sum_{j=1}^{i} u_j$.
Cet ordre a la jolie propriété $u \preceq v$ si et seulement si $\bar{v} \preceq \bar{u}$, ce qui n'est pas difficile à montrer.

Proposition 1 *Il existe une matrice 0-1 aux projections r, c si et seulement si $r \preceq \bar{c}$ (ou de manière équivalente $\bar{r} \preceq c$).*

Voici un algorithme glouton qui résout ce problème, et dont la correction est basée sur proposition précédente.

Algorithme glouton pour reconstruction de matrice 0-1 Initialiser M à 0. Pour chaque ligne (dans un ordre arbitraire), faire l'opération suivante tant que $r_i > 0$: choisir j tel que c_j est maximal. Placer $M_{ij} := 1$ et décrémenter r_i et c_j.
Pour la correction de l'algorithme, il suffit de montrer qu'à chaque opération $r \preceq \bar{c}$ reste satisfait.
Ce problème a également beaucoup de structure. Soit $\mathcal{M}$ l'ensemble des matrices qui sont une solution pour des projections (r, c) fixées. Alors $\mathcal{M}$ est connecté par l'opération élémentaire suivante. Pour une matrice $M \in \mathcal{M}$ soient les indices i_1, i_2, j_1, j_2. Si $M_{i1j1} = M_{i2j2} = 1$ et $M_{i1j2} = M_{i2j1} = 0$, alors l'échange de 0 et 1 dans ces quatre variables préserve les projections et résulte en une autre matrice $M' \in \mathcal{M}$. On peut obtenir n'importe quelle matrice de $\mathcal{M}$ par M avec simplement des opérations élémentaires. C'est dans ce sens que $\mathcal{M}$ est connecté.

Problèmes Fake scoreboard [Swerc2010]

4.5 File de priorité

Voici une structure de données qui maintient un ensemble, et permet les opérations suivantes. Il s'agit de classe PriorityQueue<E>.

offer(x) ajoute x à l'ensemble.

isEmpty() permet de savoir si l'ensemble est vide (vous l'aviez deviné, n'est-ce pas ?)

peek() retourne le plus grand élément de l'ensemble.

remove() retire le plus grand élément de l'ensemble.

L'opération peek se fait en temps constant. Tandis que les opérations offer et remove se font en temps $O(\log n)$ amorti, si n est la taille de l'ensemble. C'est beaucoup mieux que si on avait à trier à chaque appel de offer ou remove. La file de priorité est réalisée probablement — la documentation JDK ne le précise pas — par un *tas binomial*, structure de données inventée par Vuillemin en 1978. Les éléments de ce tas sont mis dans un container, généralement un vecteur.

4.6 Code de Huffman

Voici un exemple qui se sert d'une file de priorité pour construire un code qui permet de compresser un texte, dû à Huffman, 1952. Dans un tel code, chaque caractère est remplacé par une chaîne binaire, de sorte, (1) qu'aucune chaîne n'est préfixe d'une autre chaîne, et (2) que les caractères de grande fréquence sont associés à des chaînes courtes. On peut alors coder un texte par une séquence binaire qui est la concaténation des codes, et par la propriété (1) ce code peut être facilement décodé. Et par la propriété (2) cette séquence est de longueur minimale, et en général bien plus courte que le texte original.

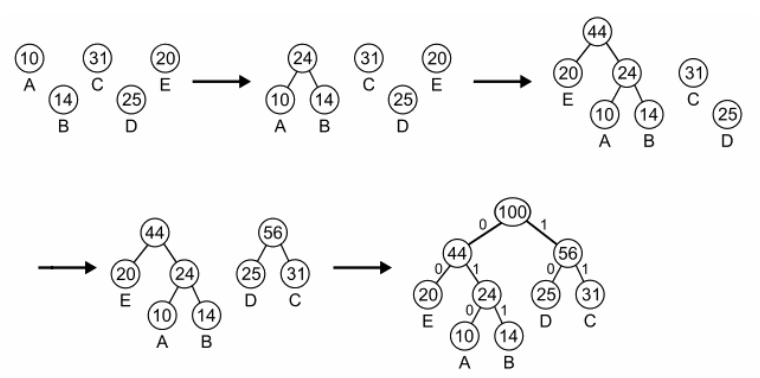


FIGURE 4.3 – Construction d'un code de Huffman. Dans les ronds, la fréquence des lettres du sous-arbre.

Pour construire un code de Huffman, on part d'une forêt, où chaque lettre est un arbre à un seul sommet étiqueté avec sa fréquence. Puis tant qu'il y a plus d'un arbre, on réunit les deux arbres de fréquences minimales, et on étiquette la nouvelle racine avec la somme des fréquences des deux sous-arbres. Placer les arbres dans une file de priorité est alors bien adapté.

4.6.1 Une implémentation

Voici une implémentation qui lit en entrée des instances spécifiant des fréquences de caractères, et qui produit pour chacune le code de Huffman. Par exemple sur l'entrée qui est donnée dans la colonne de gauche, le programme produira la sortie montrée dans la colonne de droite :

```
4
A 7                    A : 00
B 7                    C : 01
C 7                    B : 10
D 7                    D : 11
4
A 40                   D : 000
B 5                    C : 001
C 2                    B : 01
D 1                    A : 1
0
```

On représente les noeuds internes des arbres et les feuilles par un unique agrégat, qui contient fréq(uence), feuille, et deux fils. L'interprétation est que si fils est non-nul, alors feuille est le caractère attaché à la feuille. Sinon les deux fils contiennent les racines des deux sous-arbres gauche et droit.
Une difficulté technique est l'utilisation de la file de priorité. Normalement la méthode poll retourne et enlève le plus petit élément de la file, mais il faut lui dire comment comparer les noeuds. Pour cela il faut que la classe Noeud implémente Comparable<Noeud>, et il faut écrire la méthode compareTo, qui compare les fréquences de deux noeuds. Voici le code de cette implémentation.

```
class Noeud implements Comparable<Noeud> {
    int freq;
    char feuille;
    Noeud[] fils;

    Noeud(Noeud f0, Noeud f1) {
        fils = new Noeud[2];
        fils[0] = f0;
        fils[1] = f1;
        freq = f0.freq + f1.freq;
    }

    Noeud(char c, int f) {
        feuille = c;
        freq = f;
        fils = null;
    }

    public int compareTo(Noeud n) {
        return freq - n.freq;
    }

    void printTree() {printTree("");}

    void printTree(String prefix) {
        if (fils != null) {
            fils[0].printTree(prefix+"0");
            fils[1].printTree(prefix+"1");
        }
        else
            System.out.println(feuille+": "+prefix);
    }
}
```

```
class Huffman {
    /* lit <n> puis n lignes avec
       <caractere> <sa frequence>
       affiche un arbre de Huffman
    */
    public static void main(String[] args) {
        PriorityQueue<Noeud> q = new PriorityQueue<Noeud>();
        Scanner in = new Scanner(System.in);

        int n;
        while (true) {
            n = in.nextInt();
            if (n==0)
                return;
            // normalement la file est deja vide, mais soyons prudent
            q.clear();
            // lire l'instance
            while (n-->0)
                q.offer(new Noeud(in.next().charAt(0), in.nextInt()));

            // reunir les deux arbres de plus petite frequence
            Noeud n1 = q.poll();
            while (!q.isEmpty()) {
                Noeud n2 = q.poll();
                q.offer(new Noeud(n1, n2));
                n1 = q.poll();
            }
            n1.printTree();
            System.out.println();
        }
    }
}
```

Problèmes Huffman Trees [poj :1261]

4.7 Produit interne

Étant données deux séquences d'entiers positifs $x_1, \ldots, x_n$ et $y_1, \ldots, y_n$ on veut trouver une permutation σ qui minimise

$$\sum_{i=1}^{n} x_i y_{\sigma(i)}.$$

Par un argument d'échange on peut montrer qu'il faut toujours essayer de grouper le plus grand élément de x avec le plus petit élément de y, et itérer. Ainsi la solution optimale s'obtient en triant x en ordre croissant, y en ordre décroissant et en multipliant les éléments de même indice.

Problèmes minimum scalar product [gcj :2009]

4.8 Ordonnancement

Étant données deux séquences d'entiers positifs $w_1, \ldots, w_n$ et $p_1, \ldots, p_n$ on veut trouver une permutation σ qui minimise

$$\sum_{i=1}^{n} w_i \sum_{j:\sigma(j)\leq\sigma(i)} p_j.$$

Ceci modélise un problème d'ordonnancement. On dispose de n tâches, la tâche i a une durée p_i et un poids de priorité w_i. Il faut trouver un ordre d'exécution de ces tâches, $sigma(i)$ représente le rang dans cet ordre. Le temps de complétude d'une tâche i est la somme des durées des tâches qui précèdent i incluant la tâche i elle-même.

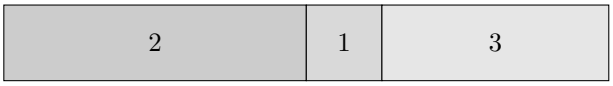


FIGURE 4.4 – Un ordonnancement optimal pour 3 tâches. La couleur représente le Smith-ratio.

Par un argument d'échange on peut montrer qu'il faut toujours ordonnancer les tâches en ordre de *Smith-ratio* décroissant, où le Smith-ratio de la tâche i est défini comme w_i/p_i (voir figure 4.4).

Chapitre 5

La programmation dynamique

Vous vous souvenez du problème *Containers*, où pour une séquence donnée il fallait trouver un partitionnement en un nombre minimal de sous-séquences décroissantes. Aujourd'hui nous allons passer en revue d'autres problèmes sur des séquences et qui se résolvent par de la programmation dynamique. La programmation dynamique est une des méthodes qui doivent faire partie de votre couteau suisse de programmeur. L'idée est de décomposer un problème en sous-problèmes, disons indexés par un paramètre k, de telle façon que calculer la solution optimale du sous-problème k puisse se faire à partir des solutions optimales pour $1, \ldots, k-1$ (ou un de leur sous-ensemble). Pour certains problèmes, k peut en fait être un vecteur de paramètres, muni d'un ordre partiel $\prec$ (lexicographique ou dominance) et la solution pour k dépend des solutions de certains p avec $p \prec k$. Euh, je suis sûr que cette explication est incompréhensible, alors voyons cette technique sur un problème concret.

5.1 La plus longue sous-séquence commune

Il s'agit du problème suivant. Étant données deux séquences $x, y \in \mathbb{N}^\star$, trouver une séquence de longueur maximale $s \in \mathbb{N}^\star$ qui est sous-séquence à la fois de x et de y. Rappel : on dit que s est sous-séquence de x s'il existe des indices $i_1 < \ldots < i_{|s|}$ tels que $x_{i_k} = s_k$ pour tout $k = 1, \ldots, |s|$. L'idée est que pour tout $i \leqslant |x|$ et pour tout $j \leqslant |y|$ on calcule la plus longue sous-séquence commune aux préfixes $x_1 \ldots x_i$ et $y_1 \ldots y_j$. Ceci nous donne $n \cdot m$ sous-problèmes pour $n = |x|$ et $m = |y|$. Et la solution optimale pour (i, j) peut calculée à partir des solutions pour $(i-1, j)$ et $(i, j-1)$, en temps constant. On peut alors résoudre la problème pour (n, m) en temps $O(nm)$, ce qui est quadratique en la taille de l'entrée, qui est $\Theta(n+m)$. L'algorithme est basé sur l'observation suivante.

Lemme 2 *Soit* $\mathrm{opt}(i, j)$ *la plus longue sous-séquence commune à* $x_1 \ldots x_i$ *et* $y_1 \ldots y_j$. *Si* $i = 0$ *ou* $j = 0$ *alors* $\mathrm{opt}(i, j)$ *est vide. Si* $x_i \neq y_j$ *alors* $\mathrm{opt}(i, j)$ *est le maximal entre* $\mathrm{opt}(i-1, j)$ *et* $\mathrm{opt}(i, j-1)$. *Si* $x_i = y_j$ *alors* $\mathrm{opt}(i, j)$ *est le maximal entre* $\mathrm{opt}(i-1, j)$ *et* $\mathrm{opt}(i, j-1)$ *et* $\mathrm{opt}(i-1, j-1).x_i$. *Ici le* . *représente la concaténation, et par* maximal *on entend la séquence de longueur maximale.*

Le problème peut se résoudre en temps $O(n + m \log m)$ si la 2ème séquence est triée, ou en temps $O(n+m)$ si les deux séquences sont triées. Savez-vous comment ?

Problèmes Common Subsequence [tju :1683]

5.2 La plus longue sous-séquence croissante

Étant donnés $x_1 \ldots x_n \in \mathbb{N}$ il s'agit de trouver une séquence d'indices $i_1 < \ldots < i_k$ telle que $x_{i1} < \ldots < x_{i_k}$. Pour cela on va pour chaque $i = 1, \ldots, n$ maintenir un ensemble de

sous-séquences du préfixe $x_1 \dots x_i$. Parmi toutes les séquences de même longueur on est intéressé à garder seulement une seule qui termine avec un élément minimal. On dit qu'elle est dominante. Par exemple pour $(1, 4, 6, 2)$ on gardera les séquences $(1), (1, 2), (1, 4, 6)$. Concrètement on maintient un tableau b tel que $b[u]$ soit le dernier élément de la plus longue sous-séquence de longueur u. Ces séquences sont chaînées de la fin vers le début, et le tableau p contient ces pointeurs. Notez que b est un tableau croissant. Maintenant quand on considère le prochain élément $x = x_{i+1}$ de la séquence, on cherche si on peut améliorer une des séquences. Soit o la longueur maximale des sous-séquences croissantes dans $x_1, \dots, x_i$. Si $x > b[o]$ alors on peut produire une nouvelle séquence de longueur $o + 1$ en ajoutant x à celle de longueur o. Sinon, soit u le plus petit indice tel que $b[u] \geqslant x$. Si l'inégalité est stricte on peut améliorer la u-ème séquence en ajoutant x à la $(u - 1)$-ème séquence. Ainsi on obtient une u-séquence qui termine par un élément strictement plus petit. La complexité est $O(n \log n)$ ou plus précisément $O(n \log k)$. Voici une implémentation de cet algorithme.

```
// retourne la longueur de la plus longue sous-sequence croissante strictement
static int find_lis(int []x) {
    int []b = new int[x.length+1];
    Arrays.fill(b, Integer.MAX_VALUE);
    int best=0; // valeur de l'optimum

    for (int xi: x) {
        /* recherche dichotomique: trouver le plus petit j
           tel que b[j] >= xi,
        */
        int bot, top, mid;
        for (bot=0, top=best; bot < top;) {
            int m = (bot + top) / 2;
            if (b[m] < xi) bot=m+1; else top=m;
        }
        b[bot] = xi; // on ameliore la fin de la plus longue sous-sequence de long bot
        if (bot==best) best = bot+1;
    }
    return best;
}
```

Problèmes Longest Ordered Subsequence [tju :1765]

5.3 La distance d'édition

Étant données deux chaînes de caractères x, y, on veut savoir combien d'opérations (insertion, suppression ou substitution) il faut pour transformer x en y. Cette distance est utilisée par exemple dans des correcteurs orthographiques. On va calculer un tableau $O[i, j]$ qui est la distance entre le préfixe de x de longueur i et le préfixe de y de longueur j. Alors pour démarrer la programmation dynamique on initialise $O[0, j] = j$ et $O[i, 0] = i$. Puis en général quand $i, j \geqslant 1$, on a trois possibilités sur les dernières lettres des préfixes. Soit x_i est supprimé. Soit y_j est inséré (à la fin). Soit x_i est remplacé par y_j, (s'il ne sont pas déjà égaux). Ceci donne la récursion suivante :

$$O[i, j] = \min \begin{cases} O[i-1, j-1] + \text{match}(x_i, y_j) \\ O[i, j-1] + \text{indel}(y_j) \\ O[i-1, j] + \text{indel}(x_i), \end{cases}$$

où match est une fonction qui retourne 0 si $x_i = y_j$ et 1 sinon, et indel retourne toujours 1. Ces fonctions codent le coût d'une substitution, insertion ou suppression d'une lettre, qui peut ainsi

être ajusté. Par exemple le coût de substitution pourrait dépendre de la distance des lettres sur le clavier. Voici ce que donne le calcul de la matrice O sur les chaînes $x =$ " THOU " et $y =$ " YOU ". La distance est donc de 2. La chaîne des transformations peut être reconstruite en partant de la case $O[|x|, |y|]$ et en remontant récursivement vers la case qui a réalisé le minimum (montré en gras). Ainsi on réalise que d'abord T est remplacé par Y et puis H est supprimé.

		T	H	O	U
	0	1	2	3	4
Y	1	**1**	**2**	3	4
O	2	2	2	**2**	3
U	3	3	3	3	**2**

Notez que les indices du tableau commencent à 0, et ceux des chaînes à 1. À tenir compte dans une implémentation.

5.4 Correcteur orthographique

Comment stocker les mots d'un dictionnaire pour réaliser un correcteur orthographique ? On pouvoir rapidement trouver un des mots du dictionnaire les plus proches d'un mot donnée. Pour cela ça serait idiot de stocker les mots du dictionnaire dans une table de hashage. On perderait alors toute l'information de proximité entre les mots.
Non, ce qui est beaucoup plus malin est de les stocker dans un arbre préfixe, appelé aussi *trie*. Dans un tel arbre, les arcs d'un noeud vers ses fils sont étiquetés par des lettres distincts. Chaque mot du dictionnaire correspond alors à un chemin de la racine vers un noeud de l'arbre. On doit alors marquer les noeuds pour distinguer ceux qui correspondent à des mots du dictionnaire de ceux qui sont seulement des préfixes stricts de mots du dictionnaires.

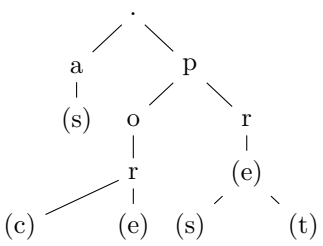


FIGURE 5.1 – un arbre préfixe pour les mots "as", "porc", "pore", "pré", "près", "prêt" (mais sans les accents). Pour l'illustration l'étiquette de l'arete est indiquée sur le noeud du fils. Les parenthèses marquent les noeuds noeuds correspondant à des mots du dictionnaire.

Voici la structure qui réalise cet arbre.

```
// noeud de l'arbre des prefixes
class Node {
    boolean isWord; // indique que le mot qui a mene' de la racine vers ce noeud est dans le
        dictionnaire
    Node son[];
    Node () {son = new Node[27];}
}
```

Avec une telle structure il est facile de chercher un mot du dictionnaire qui soit à distance d d'un mot donné. Il suffit de simuler les opérations d'édition à chaque noeud, et alors effectuer les appels récursifs de la recherche avec le paramètre $d - 1$.
Il existe une structure améliorée qui regroupe des noeuds à un seul fils. Elle porte le nom de *Patricia trie*, mais dépasse ce cours.

```
// ajoute un mot s dans l'arbre dont la racine est n, retourne le remplacement pour n
static Node add(Node n, String s) {
    if (n==null)
        n = new Node();
    if (s.equals(""))
        n.isWord = true;
    else {
        // --- decomposer s
        char c = s.charAt(0);
        String r = s.substring(1);
        if ('a'<=c && c<='z')
            n.son[c-'a'] = add(n.son[c-'a'], r);
    }
    return n;
}

// cherche un mot dans le dico n qui soit a distance au plus dist de s
static String find(Node n, String s, int dist) {
    if (dist<0 || n==null) return null; // cas de base
    if (s.length()==0)
        if (n.isWord)
            return "";
        else
            return null;
    // --- decomposer s
    char c = s.charAt(0);
    int i = c-'a';
    String r = s.substring(1);
    // --- chercher
    String f = find(n.son[i], r, dist); // direct
    if (f!=null) return c+f;
    for (c='a'; c<='z'; c++) {
        f = find(n.son[i], s, dist-1); // insertion
        if (f!=null) return c+f;
        f = find(n.son[i], r, dist-1); // substitution
        if (f!=null) return c+f;
    }
    return find(n, r, dist-1); // suppression
}


// trouve un mot le plus proche du dictionnaire
static String find(Node root, String s) {
    String f;
    // augmenter distance jusqu'a succes
    for (int dist=0; (f=find(root,s,dist))==null; dist++) {}
    return f;
}
```

5.5 Plus courte super-séquence commune

Entrée deux séquences $x \in \mathbb{N}^n$, et $y \in \mathbb{N}^m$.

Sortie une séquence z de longueur minimal, telque à la fois x et y sont sous-séquence de z.

Réduction à la distance d'édition restreint aux opérations d'insertion et de suppression. Les insertions transforment x en z et les suppressions z en y.

5.6 Tous les plus courts chemins par l'algorithme de Floyd

Étant donné un graphe $G(V, E)$ pondéré sur les arêtes $w : E \to \mathbb{N}$ on veut trouver pour tout $u, v \in V$ la longueur du plus court chemin de u à v. (Avec un peu de travail, mais pour le même prix, on peut trouver les plus courts chemins en plus de leur longueur).
L'idée est qu'on numérote les sommets de 1 à n, où $n = |V|$. Et pour chaque $k = 0, \ldots, n$ on calcule pour tout couple (u, v) la longueur du plus court chemin de u à v en ne passant que par des sommets intermédiaires de numéro inférieur ou égal à k. On maintient donc une matrice M de dimension $n \times n$, et qu'on note M_k pour la k-ème itération. Initialement elle vaut $M_0[u, v] = w(u, v)$ pour tout $(u, v) \in E$ et $M_0[u, v] = \infty$ pour tout $(u, v) \notin E$. Puis pour k on met à jour :

$$M_k[u, v] = \min\{M_{k-1}[u, v], M_{k-1}[u, k] + M_{k-1}[k, v]\}.$$

L'idée est qu'on vérifie si passer par k permet de raccourcir la distance de u à v. En pratique on peut faire ce calcul dans une seule matrice, qui sera égal à M_k à la k-ème itération. Et la valeur ∞ peut être représentée par la constante Integer. MAX_VALUE /2. On divise par deux pour éviter un débordement lors de l'addition.

5.7 Multiplication d'une chaîne de matrices

Soient n matrices $M_1, \ldots, M_n$, où la i-ème matrice a r_i lignes et c_i colonnes, avec $c_i = r_{i+1}$ pour tout $1 \leqslant i < n$. On veut calculer le produit $M_1 M_2 \ldots M_n$ avec le moins d'opérations possibles. Pour multiplier deux matrices M_i et M_{i+1}, on utilise l'algorithme des écoliers qui prend $r_i c_i c_{i+1}$ opérations. Il y a un algorithme plus efficace de Strassen, mais qui est trop compliqué pour nous. Le but est de trouver un ordre pour multiplier au moindre coût. La récursion est simple. La dernière multiplication multiplie le résultat de $M_1 \ldots M_k$ avec le résultat de $M_{k+1} \ldots M_n$ pour un certain $1 \leqslant k < n$. Soit $m(i, j)$ le coût minimal pour calculer $M_i \ldots M_j$. On a alors $m(i, i) = 0$ et si $i < j$

$$m(i, j) = \min_{i \leqslant k < j} (m(i, k) + m(k + 1, j) + r_i c_j c_k).$$

5.8 Bin packing

Soit k une petite constante. Étant donnés n_i objets de poids p_i pour chaque $1 \leqslant i \leqslant k$, ainsi qu'une capacité B, comment placer les objets en un nombre minimal de boîtes afin que le contenu d'aucune ne dépasse la capacité B ? Soit $\mathcal{T} := \{0, 1, \ldots, n_1\} \times \cdots \times \{0, 1, \ldots, n_k\}$, la famille de tous les vecteurs décrivant tous les sous-ensembles possibles d'objets. Pour simplifier on identifie les vecteurs de $\mathcal{T}$ avec les ensembles d'objets.
D'abord on calcule $\mathcal{C} \subseteq \mathcal{T}$ l'ensemble des vecteurs $(i_1, \ldots, i_k)$ tel que $i_1 p_1 + \ldots + i_k p_k \leqslant B$, la famille de tous les ensembles qui tiennent dans une boîte. Ceci se fait facilement en temps $O(|\mathcal{T}|) = O(n_1 \cdots n_k)$.
Pour calculer le nombre minimal de boîtes nécessaires pour placer l'ensemble $(i_1, \ldots, i_k)$, on suit la récursion suivante. Si $(i_1, \ldots, i_k) \in \mathcal{C}$ alors $b(i_1, \ldots, i_k) = 1$ par définition de b. Sinon, il faut plus qu'une boîte, et la première boite contient forcément un ensemble décrit par un vecteur $(j_1, \ldots, j_k) \in \mathcal{C}$. Donc dans le cas où $(i_1, \ldots, i_k) \notin \mathcal{C}$ on a

$$b(i_1, \ldots, i_k) := \min b(i_1 - j_1, \ldots, i_k - j_k) + 1$$

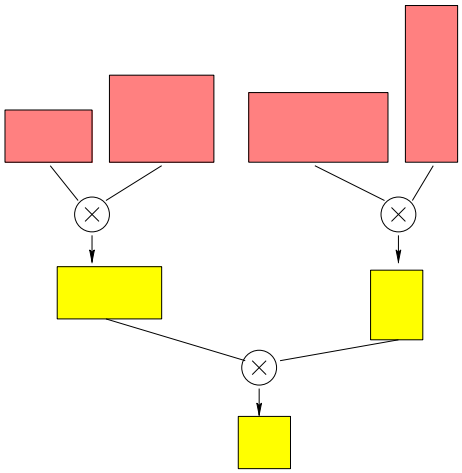


FIGURE 5.2 – Dans quel ordre faut-il multiplier une séquence de matrices afin de minimiser le nombre total d'opérations ?

où le minimum est pris sur $(j_1, \ldots, j_k) \in \mathcal{C} \cap \{0, \ldots, i_1\} \times \cdots \times \{0, \ldots, i_k\}$. Ces calculs se font en temps $O(|\mathcal{T}| \cdot |\mathcal{C}|) = O(|\mathcal{T}|^2)$.

5.9 Détails d'implémentation

Dans le cas de l'évaluation d'un programme dynamique calculant $O[i, k]$ à partir des valeurs $O[i-1, k']$ pour certains $1 \leqslant k' \leqslant$ MAXK, on peut calculer d'abord toutes les valeurs pour $i = 1$ puis pour $i = 2$ et ainsi de suite. Mais pour certains problèmes, par exemple *Bin-packing*, il est possible qu'on calcule des valeurs qui n'enteront pas dans la détermination de l'optimum. On pourrait alors calculer seulement les valeurs nécessaires et procéder comme pour la programmation récursive. On définirait alors une fonction d'accès $o(i, k)$ qui regarde dans la table O si la valeur $O[i, k]$ a déjà été calculé, et la calcule avec la formule de récurrence le cas échéant.
Une autre manière d'optimiser est d'économiser de l'espace. Vous avez remarqué qu'une fois toutes les valeurs pour $O[i, \cdot]$ calculées, on n'a plus usage des valeurs $O[i-1, \cdot]$ et des précédentes. Alors au lieu de déclarer `int O[MAX_I][MAX_K]` ; on pourrait déclarer `int O[2][MAX_K]` ; et calculer `O[i%2][k]` à partir de `O[(i-1)%2][.]`. Attention à ce que $i \geqslant 1$, car en C++ ou Java, le modulo d'un nombre négatif est négatif aussi.
Parfois on veut calculer la solution optimale en plus de sa valeur. Dans ce cas on peut maintenir en parallèle avec le tableau $O[i, j]$ un tableau $S[i, j]$ qui contient la solution optimale de valeur $O[i, j]$. Lors du choix optimal dans le programme il faudrait alors maintenir aussi S.

Chapitre 6

Ensembles

6.1 Subsetsum

Étant donnés n entiers positifs $x_0, \ldots, x_{n-1}$, on veut savoir s'il existe un sous-ensemble dont la somme vaut une valeur R donnée. Ce problème est NP-difficile lorsque R fait partie de l'entrée, mais il existe un algorithme dont la complexité est en $O(nR)$, alors si R n'est pas trop grand... Idée : par programmation dynamique. On maintient un tableau booléen, qui indique pour chaque $0 \leq s \leq R$ s'il existe un sous-ensemble des entiers $x_0, x_1, \ldots, x_{i-1}$ dont la somme vaut s. Pour $i = 0$, ce tableau n'est vrai qu'à l'indice 0, pour l'ensemble vide.

```
static boolean subsetsum(int x[], int R) {
    boolean b[] = new boolean[R+1];
    b[0] = true;
    for (int xi: x)
        for (int s=R; s>=xi; s--)
            b[s] |= b[s-xi];
    return b[R];
}
```

6.2 Partitionner un ensemble d'entiers le plus équitablement possible

Entrée : n entiers positifs $x_0, \ldots, x_{n-1}$.

Sortie : Un partitionnement de $\{0, \ldots, n-1\}$ en S et $\bar{S}$, tel que $\sum_{i \in S} x_i - \sum_{i \in \bar{S}} x_i$ soit le plus petit possible en valeur absolue.

Complexité $O(nR)$ pour $R = \sum_i x_i$.

Algorithme procéder comme pour Subsetsum, puis chercher dans le tableau de booléen t un indice s avec $t[s]$ vrai avec s le plus proche de $R/2$ en valeur absolue. Donc pour tout $d = 0, 1, 2, \ldots$ et $o = +1, -1$, considérer $t[R/2 + o * d]$. Il n'y a pas de risque de débordement de tableau, si $n > 0$.

6.3 Coin change

Cette fois-ci on veut savoir s'il existe une combinaison linéaire positive de $x_0, \ldots, x_{n-1}$ qui vaut R. On veut produire un montant R avec des pièces de monnaie qui valent respectivement

$x_0, \ldots, x_{n-1}$ centimes. Vous rigolez, mais en Birmanie il y avait des billets de 15, 35, 45, 75 et de 90 Kyats.
Pour ce problème on peut utiliser la même récurrence, mais il faut juste inverser l'ordre de la boucle. Ainsi une valeur x_i peut contribuer plusieurs fois à une somme.

```
static boolean coinchange(int x[], int R) {
    boolean b[] = new boolean[R+1];
    b[0] = true;
    for (int xi: x)
        for (int s=xi; s<=R; s++)
            b[s] |= b[s−xi];
    return b[R];
}
```

6.4 k-tuplet dont la somme vaut R

De nouveau on nous donne n entiers $x_0, x_1, \ldots, x_{n-1}$ et on veut en sélectionner k dont la somme vaut R, pour une constante k fixé.

6.4.1 $k = 2$

On cherche un x_i et un x_j tel que $x_i = R - x_j$. Ceci se fait en temps linéaire avec une table de hashage.

6.4.2 $k = 3$

Il suffit de trier l'entrée tel que $x_0 \leq \ldots \leq x_{n-1}$ en temps $O(n \log n)$. Puis pour chaque entier x_i, on cherche une intersection dans les listes $x_o + x_i, x_1 + x_i, \ldots, x_{n-1} + x_i$ et $R - x_{n-1}, \ldots, R - x_0$.
On peut faire cette recherche, avec la technique du *merge*. On utilise deux pointeurs, et on à chaque fois avance celui qui pointe sur la valeur la plus petite.
Faire attention à enlever $x_i + x_i$ de la première liste..
La complexité totale est donc en $O(n^2)$.
Cette technique se généralise à une complexité de $O(n^{k-1})$.

Problèmes 4 Values whose Sum is 0 [poj :2785]

6.5 Voyageur de commerce

Encore un autre problème NP-dur. On nous donne un graphe à n sommets, disons complet et pondéré sur les arêtes, et on veut trouver un cycle hamiltonien — c'est à dire un cycle qui passe par tous les sommets exactement une fois — qui soit de poids total minimal.
On peut résoudre ce problème par programmation dynamique en temps $O(n^2 2^n)$, ce qui est acceptable pour $n \leq 15$.
Pour l'algorithme suivant on fixe une origine, le sommet d'indice $n-1$. Pour chaque ensemble $S \subseteq \{0, 1, \ldots, n-2\}$, on calcule $O[S][j]$, la poids minimal d'un chemin débutant à l'origine, passant par tous les sommets de S puis terminant en j, avec $j \notin S$.
Pour le cas de base, $O[\emptyset][i]$ est juste la distance de $n-1$ vers i. Sinon $O[S][i]$ pour S non-vide est le minimum de

$$O[S \backslash \{j\}][j] + w(j, i)$$

pour tout j dans S, où $w(j, i)$ est le poids de l'arête (j, i).

Chapitre 7

Ordres partiels

Pour le tri topologique, voir section 12.5.

7.1 Largeur d'un ordre partiel

On nous donne un ordre partiel. C'est-à-dire un graphe dirigé $G(V, A)$ sans cycle. Une chaîne dans G est un chemin dirigé, et par extension l'ensemble des sommets qu'il couvre. Une anti-chaîne dans G est un ensemble de sommet S tel que qu'il n'existe pas $(u, v) \in S^2$ avec $(u, v) \in A$.
La largeur de l'ordre partiel est définie comme la taille de la plus grande anti-chaîne. Un problème est de calculer cette largeur.
Un autre problème est de trouver une décomposition du graphe en un nombre minimal de chaînes.
Pour une anti-chaîne S et une décomposition en chaînes D on a clairement que S intersecte chaque chaîne dans D que dans au plus un sommet. Mais chaque sommet $v \in S$ doit appartenir à un chemin de D, comme D partitionne le graphe. Donc $|S| \leq |D|$. Le théorème de Dilworth dit que les optimima à ces deux problèmes sont en fait égaux.
Comment calculer une décomposition minimale en chaînes ? Ce problème se réduit au problème de couplage maximum.

- Produire un graphe bi-parti $H(V, V', A')$, où V' est une copie de V, et $(u, v') \in A'$ ssi $(u, v) \in A$.
- Calculer un couplage maximal bi-parti dans H.
- Compter le nombre de sommets non-couplés dans U. C'est la taille de la plus grande anti-chaîne dans G, donc la largeur de l'ordre partiel G.

Problèmes StockCharts [GoogleCodeJam :2009], Taxi Cab Scheme [Onlinejudge :3126, poj :2060]

7.2 Ensemble initial de poids maximum

On nous donnee un graphe $G(V, A)$ avec un pondération $w : V \to \mathbb{R}$. Typiquement un ordre partiel, mais le problème est défini pour un graphe général. Et le but est de construire un ensemble initial de poids maximum.
Un ensemble S de sommets est initial si pour tout arc (i, j), on a $j \in S \Rightarrow i \in S$. Si on dessine le graphe avec les arcs de gauche vers la droite, un ensemble initial est en gros un ensemble gauche.

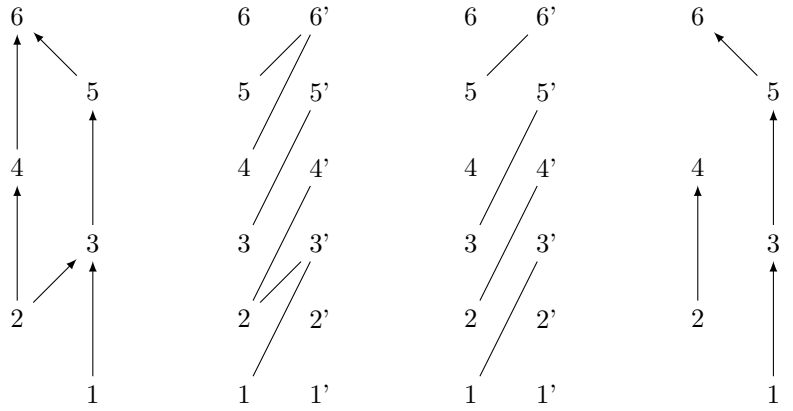


FIGURE 7.1 – De gauche à droite : un ordre partiel G, le graphe biparti associé H, un couplage maximum dans H, la décomposition en chaînes maximum dans G associée.

Maintenant on dispose d'un poids w sur les sommets, et on veut l'ensemble initial de poids total maximum.
Réduction à un problème de flot.
Soit P l'ensemble des sommets de poids positifs et N l'ensemble des sommets de poids négatifs ou nul.

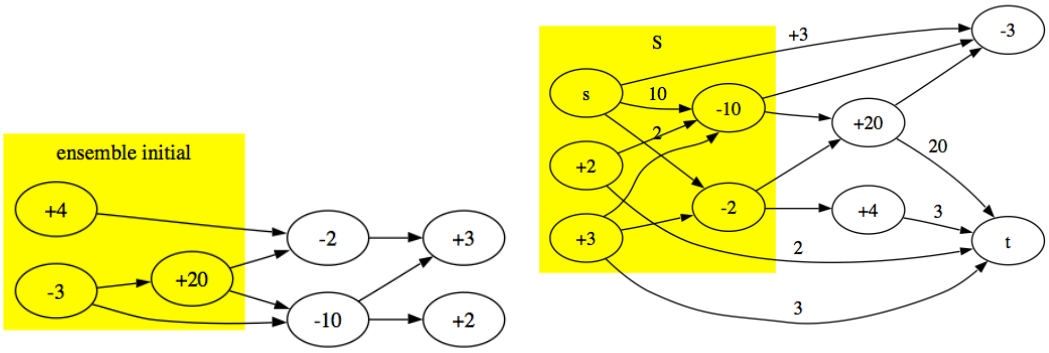


FIGURE 7.2 – (gauche) un ordre partiel et un ensemble initial de poids 21, (droite) Le graphe H et une coupe $(S, \bar{S})$.

On construit un graphe H, qui consiste en le graphe G avec les arcs renversés et de capacité infinie. Il y a aussi deux nouveaux sommets s, t. Il y a un arc (s, u) de capacite $w(u)$ pour tout u dans N et un arc (u, t) de capacité $-w(u)$ pour tout u dans P.
Soit $(S, \bar{S})$ une $s-t$-coupe de valeur finie dans H, avec S contenant s et $\bar{S}$ contenant t. Comme la valeur de la coupe est finie, il n'y a pas d'arc sortant de S dans H. Comme on a renversé les arcs, dans G, aucun arc n'entre en $S\backslash\{s\}$, qui est donc un ensemble initial pour G.
La valeur de la coupe est $w(P \cap \bar{S}) + |w(N \cap S)|$. Donc $w(P)$ moins la valeur de la coupe est

$$w(P \cap S) - |w(N \cap S)| = w(P \cap S) + w(N \cap S) = w(S).$$

Et en particulier si $(S, \bar{S})$ est la coupe minimale alors $S\backslash\{s\}$ est un ensemble initial de poids maximal.

Variantes

Un ensemble S est final, si pour tout arc (i, j) on a $i \in S \Rightarrow j \in S$. Si on veut calculer un ensemble final de poids maximum, il suffit de ne pas inverser les arcs. Si on veut des ensembles de poids minimaux, alors il suffit d'inverser les poids.

Chapitre 8

Intervalles

Une petite aide : l'intersection entre deux intervalles $[s_1, e_1), [s_2, e_2)$ est $[\max\{s_1, s_2\}, \max\{e_1, e_2\})$ et est vide si $e_1 \leq s_2 \vee ee_2 \leq s_1$.

8.1 Trouver une valeur commune à un nombre maximum d'intervalles

On vous donne n intervales semi-ouverts, c'est-à-dire de la forme $[s, t)$. Et il faut trouver un point x sur la ligne qui est contenu dans un nombre maximum d'intervalles.
En temps $O(n \log n)$ par balayage. On produit un tableau de couples d'entiers (x, d), où x est une extrémité d'un des intervalles données et $d \in \{-1, +1\}$ précise s'il s'agit du côté gauche $(+1)$ ou du côté droit (-1).
Il faut trier ce tableau dans l'ordre lexicographique, et par un balayage cumuler dans une variable o tous les d des couples (x, d) rencontrés. La valeur o après le traitement de (x, d) représente le nombre d'intervalles contenant x.

Variantes Si les intervalles sont fermés, alors il suffit d'ordonner les couples (x, d) en ordre lexicographique de $(x, -d)$.

8.2 Minimum hitting set

On vous donne n intervales. Et il faut trouver un nombre minimum de valeurs, tel que chaque intervalle en contienne au moins une.
En temps $O(n \log n)$ par balayage.

Observation On peut ignorer les intervalles qui contiennent d'autres intervalles (superset). Pour la solution on peut se restreindre à des valeurs qui sont les extrémités droits d'un intervalle. Trier les intervalles $[s_1, e_1], \ldots, [s_n, e_n]$ tel que $s_i <= s_{i+1}$. Ce n'est pas nécessaire de trier aussi par les extrémités e_i. On va produire une solution $t_1, \ldots, t_k$. Pour chaque i on a construit la solution pour les i premiers intervalles qui minimise k et qui maximise t_k. Cette solution est dominante.
Initialiser $k = 0$. Puis pour tout i :

- Si $k = 0$ ou $t_k < s_i$ alors $k := k + 1$ et $x_k := e_i$, car une nouvelle valeur est nécessaire pour couvrir l'intervalle i.
- Et si $e_i < t_k$ alors $t_k := e_i$, car on a trouvé un intervalle inclu.

```
class Intervalle implements Comparable<Intervalle> {
    double l, r;
    int d;
    static int c=0;
    Intervalle(double _l, double _r) {l=_l; r=_r; d=c++;}
    public int compareTo(Intervalle I) { // ordre lexicographique (l,r)
        int dl = (int)Math.signum(l-I.l);
        int dr = (int)Math.signum(r-I.r);
        return dl!=0 ? dl : d-I.d;
    }
}
```

```
        // -- parcours dans l'ordre
        Arrays.sort(I);
        int k=0; // nb intervalles crees
        double x=0; // position derniere antenne
        for (Intervalle i: I)
            if (k==0 || x<i.l) { // -- nouveau intervalle
                k++;
                x = i.r;
            }
            else if (x>i.r) // -- sous-intervalle
                x = i.r;

        return k;
```

Problèmes Radar Installation [Tianjin :1115]

8.3 Nombre maximum d'intervalles disjoints

Selectionner un sous-ensemble maximum d'intervalles deux à deux disjoints.
Algorithme en $O(n \log n)$ par balayage.
Même observation que pour le problème précédent. Sans perte de généralité une solution ne comporte pas d'intervalle qui inclu un autre. Donc si on ignore ces intervalles, l'instance peut être trié tel que les côtés gauches soient ordonnées et simulatément les côtés droits. Maintenant on peut se convaincre que sans perte de généralité une solution est dominante à gauche, c'est-à-dire elle comporte le premier intervalle, puis le prochain qui est disjoint avec le premier, et ainsi de suite. Une telle solution est trouvé par l'algorithme glouton.

```
        // -- parcours dans l'ordre
        Arrays.sort(I);
        int k=0; // nb intervalles retenus
        int x=Integer.MIN_VALUE; // cote droit dernier intervalle retenu
        for (Intervalle i: I)
            if (x<i.l) { // -- nouveau intervalle
                k++;
                x = i.r;
            }
            else if (x>i.r) // -- intervalle inclu
                x = i.r;
        return k;
```

8.4 Arbres de Fenwick

Cette structure permet de maintenir un tableau t de n valeurs et effectuer en temps $O(\log n)$ les opérations suivantes.

- changer $t[i]$ pour un indice i donné.
- calculer $t[1] + \ldots + t[i]$ pour un indice i donné.

Pour des raisons techniques les indices de t varient seulement de 1 à $n-1$, sans inclure 0.
Avec une petite modification cette structure peut également servir à effectuer les opérations suivantes en temps $O(\log n)$.

- ajouter une valeur d à tout un intervalle $t[a], t[a+1], \ldots, t[b]$ pour des indices a, b donnés.
- demander la valeur de $t[i]$ pour un indice i donné.

L'idée de la structure est de ne pas stocker le tableau t tel quel mais de stocker dans un tableau s des sommes d'intervalles dans t. Ainsi $s[i]$ stocke la somme de $t[j]$ sur $j \in I(i)$, où $I(i)$ est un intervalle défini ci-dessous. Ces intervalles sont organisés sous forme d'arbre par inclusion de la manière suivante (voir figure 8.1).

- L'entrée $s[i]$ pour $i = a10^k$ en décomposition binaire est responsable de la somme de t dans l'intervalle $I(i) = \{a0^k1, \ldots, i\}$. Ici a est un mot binaire.
- Le père de l'indice $i = a10^k$ est $i + 10^k$.
- Le prédécesseur de l'indice $i = a10^k$ est $j = a00^k$. L'intervalle $I(j)$ est celui qui précède $I(i)$.

Ainsi la somme du préfixe $t[1] + \ldots + t[i]$ avec $i = a10^k$ est $s[i]$ plus récursivement la somme du préfixe de $t[a0^{k+1}]$.

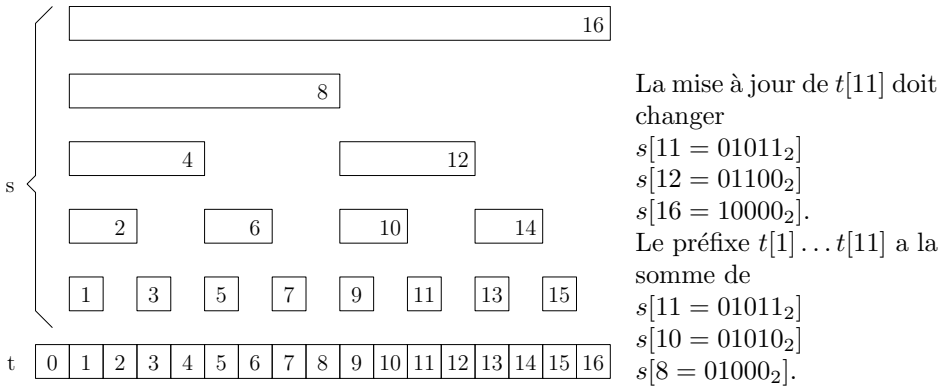


FIGURE 8.1 – Exemple d'un arbre Fenwick.

Une opération importante pour implémenter cette structure consiste à lire le bit de poids faible de i, c'est à-dire transformer $a10^k$ en 10^k. Cette opération est tout simplement `i&-i`.
Explication :

$$\begin{aligned} i &= a10^k \\ \bar{i} &= \bar{a}01^k \\ -i = \bar{i} + 1 &= \bar{a}10^k \\ i\& - i &= 10^k \end{aligned}$$

```
            sum += tree[idx];
            idx -= (idx & -idx);
        }
        return sum;
    }

    // retourne la somme de l'intervalle [a,b]
    static int intervalSum(int a, int b) {
        return prefixSum(b) - prefixSum(a-1);
    }

    // ajoute une valeur a un indice
    static void update(int idx ,int val) {
        assert(0<idx);
        while (idx < tree.length) {
            tree[idx] += val;
            idx += (idx & -idx);
        }
    }

    // variante:
    // cette fois ci on permet
    // - de modifier la valeur de tout un intervalle d'elements
    // - demander la valeur d'un element particulier
    // tout ca en temps O(log n)
    // l'idee est que dans le tableau on ne stocke que
    // les differences entre les elements successifs

    static void intervalUpdate(int a, int b, int d) {
        update(a, +d);
        update(b+1, -d);
    }

    static int read(int idx) {
        return prefixSum(idx);
    }
```

Chapitre 9

Tableaux, matrices

9.1 Somme d'un intervalle

On veut coder un tableau t de $n$entiers, tel qu'on puisse calculer en temps constant la somme pour un intervalle d'indice données. Pour cela on calcule la somme des préfixes dans un tableau s à $n+1$ entiers, tel que $s[j] := t[0] + \ldots + t[j-1]$. Dans ce cas la somme d'un intervalle d'indices $[a, b]$ se calcule par

$$s[b+1] - s[a] = t[a] + t[a+1] + \ldots + t[b].$$

Cette astuce se généralise bien aux matrices à deux ou plusieurs dimensions, et porte le nom d'*inclusion/exclusion.*

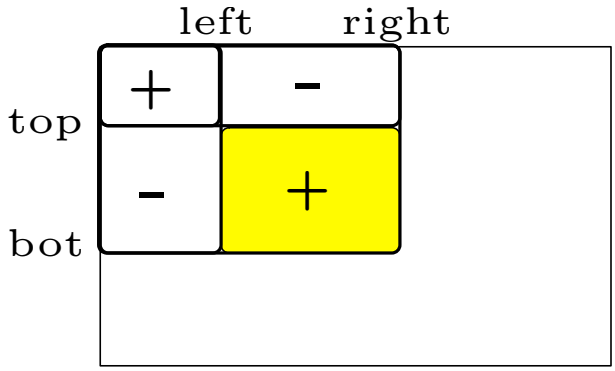


FIGURE 9.1 – Principe d'inclusion/exclusion.

Si $S[i0][j0]$ est la somme sur $M[i][j]$ avec $0 \leq i < i0$ et $0 \leq j < j0$, alors la somme de $M[i][j]$ avec $i \in [top, bot]$ et $j \in [left, right]$ s'écrit comme
$S[bot+1][right+1] - S[bot+1][left] - S[top][right+1] + S[top][left]$.

9.2 Maximum d'un intervalle

Cette-fois ci, les requêtes demandent le maximum du tableau dans un intervalle d'indices.
On effectue un précalcul en temps $O(n \log n)$ qui permet de répondre à ces requêtes en temps constant.
À partir du tableau t de n entiers, on construit une matrice s de dimension $n \times \lceil \log_2 n \rceil$. L'entrée $s[i][k]$ comporte le maximum sur $t[i], t[i+1], \ldots, t[i+2^k-1]$. La récurence pour s est simple. Le cas de base est $s[i][0] = t[i]$, et sinon $s[i][k+1] = \max\{s[i][k], s[i+2^k][k]\}$.

```
|i             j|
+--------+           de i à i+2^k
      +--------+     de j-2^k à j
```

Maintenant le maximum sur $t[i], t[i+1], \ldots, t[j]$ est $\max\{s[i][k], s[j-2^k][k]\}$ pour $k = \lfloor \log_2(j-i) \rfloor$.
En exercice : trouvez une structure de donnée qui se précalcule en temps $O(n)$ et permet de répondre aux requêtes en temps $O(\log n)$.

Variante : tous les intervales sont de taille k

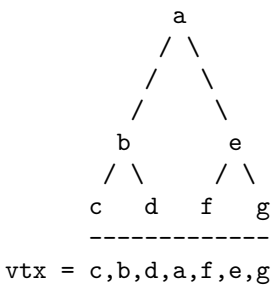
Quand toutes les requêtes concernent des intervalles de taille k on peut avoir une structure plus simple avec une complexité linéaire pour le précalcul, et constant pour les requêtes.
Diviser le tableau en blocs de taille k chacuns, sauf evt. le dernier. Un element i dans le tableau appartient à un bloc et y définit un prefixe et un suffixe. On va stocker dans un tableau *pre*, le maximum de ce préfixe, et dans un tableau *suf*, le maximum de ce suffixe.

```
tableau = |1 4 3 2|7 3 4 6|2 8 9 3|2 3|
indice i               ^
préfixe déf par i  +----+
suffixe déf par i      +--+

tab.pre=  |1 4 4 4|7 7 7 7|2 8 9 9|2 3|
tab.suf=  |4 4 3 2|7 6 6 6|9 9 9 3|3 3|

le max d'un intervalle est le max d'un suffixe et d'un préfixe
               +--+-----+
               i        i+k-1

int maxInt(int i) {return max(suf[i], pre[i+k-1]);}
```

Problèmes zju :1752 [quadtrees], lié zju :1610

9.3 Antécédent commun le plus bas

Le problème du *least common ancestor* est le suivant. On a un un arbre binaire, et on doit pouvoir répondre aux requêtes suivantes : pour deux sommets données, il faut trouver l'antécédant commun le plus bas dans l'arbre.
Précalcul en $O(n \log n)$ et requêtes en $O(1)$. Notez, qu'il y a une solution en $O(n)$, mais pour le concours un facteur $\log n$ est acceptable.
C'est une réduction au problème de requête du minimum dans un intervalle. Faire un parcours en profondeur de l'arbre. Inscrire dans l'ordre infixe les sommets rencontrés dans un tableau *vtx*, les niveaux correspondants dans un tableau *lev* et dans un tableau *idx* les positions associés aux noeuds.




```
vtx = c,b,d,a,f,e,g
```

```
lev = 2,1,2,0,2,1,2
idx = a:3, b:1, c:0, d:2, e:5, f:4, g:6
```

```
int lev[], vtx[], idx[];
// retourne la taille de l'arbre T
int makeLevelTab(Tree T) {return makeLevelTab(T,0,0);}
int makeLevelTab(Tree T, int level, int pos) {
    if (T==null) return 0;
    int left = makeLevelTab(T.left, level+1, pos);
    idx[T.root] = pos+left;
    vtx[pos+left] = T.root;
    lev[pos+left] = level;
    int right = makeLevelTab(T.right, level+1, pos+left+1);
    return left+1+right;
}
```

Problèmes Closest Common Ancestors [zju :1141]

Arbres non-binaires

Pour des arbres non-binaires, d'arité $d > 2$, c'est un petit peu plus compliqué. Chaque sommet génère alors $d + 1$ entrées dans les tableaux *vtx* et *lev* : une fois lors de la première visite, puis chaque fois qu'on a terminé d'explorer un de ses sous-arbres. À la place de *idx*, on note alors l'indice de la première occurrence dans *vtx* et l'indice de la dernière. Notons *first, last* ces deux tableaux. Alors l'intervalle de recherche devient *[min(first(u),first(v), max(last(u),last(v)]*.

9.4 Nombre d'intervalles contenant une valeur donnée

On nous donne n intervalles, et on veut pouvoir répondre rapidement aux requêtes suivante : étant donnée une valeur v combien d'intervalles contiennent v.
Précalcul en $O(nlogn)$, requêtes en $O(logn)$.

Précalcul trier les tableaux s et e comportant les extrémités respectives des n intervalles. Soit *int index(int t[], int v)* une fonction qui retourne le dernier indice $0 <= i < n$ tel que $t[i] <= v$ pour un tableau t trié.

Requête la réponse est le nombre d'intervalles $[s_i, e_i)$ avec $s_i \leq v$ moins le nombre d'intervalles $[s_i, e_i)$ avec $e_i \leq x$. Se résoud avec deux recherches dichotomiques, donc *index(s,v)-index(e,v)*.

9.5 Médian d'un tableau d'entiers

Entrée Un tableau de n nombres

Sortie le médian

Explication On veut trouver l'élément de rang $\lfloor n/2 \rfloor$. C'est ça le problème et l'algorithme se généralise naturellement à trouver l'élément de rang k.

Complexité linéaire

En C++ Il existe dans STL une méthode pour ce problème *nth_element*. Pour Java une telle fonction semble manquer dans JDK.

Algorithme en $O(n \log n)$ On pourrait trier le tableau puis lire l'élément à la k-ème position. Ça coûte $O(n \log n)$, mais on fait trop de travail. Si on regarde de près l'algorithme quicksort, on voit qu'on a pas besoin de faire toutes les étapes pour résoudre le problème.
Pour le concours, $O(n \log n)$ sera souvent suffisant. Mais pour les cas extrèmes, et pour les curieux...

Algorithme en $O(n)$ Supposons que tous les éléments du tableau soient distincts pour simplfier. Ce qu'on veut est une procédure qui permute les éléments du tableau T tel qu'au résultat tous les éléments de $T[1]$ à $T[k-1]$ soient plus petit que $T[k]$ et tous les éléments de $T[k+1]$ à $T[n]$ soient plus grand que $T[k]$. C'est plus faible que de trier complètement T.

Ingrédient 1 : s'inspirer de quicksort *Partition(a,b,T)* est une fonction qui choisit un pivot c avec $a \leq c \leq b$ et permute T entre les indices a et b inclus. Elle retourne ensuite le nouvel indice de $T[c]$.
On s'en sert dans *Select(a,b,k,T)* qui cherche dans $T[a...b]$ l'élément de rang k avec la promesse $a \leq k \leq b$. C'est facile, si $a = b$ on retourne $T[k]$. Sinon on calcule d=*Partition(a,b,T)* et on compare d avec k. Si $k = d$ on retourne $T[k]$. Si $k < d$ on retourne *Select(a,d-1,k,T)* et si $k > d$ on retourne *Select(d+1,b,k,T)*. Comme la récursivité est terminale, on peut faire tout en itératif.

Ingrédient 2 : choix du pivot Traditionellement pour Partition on cherche un pivot uniformément au hasard entre a et b. Si on fait ça, la complexité totale sera $O(n)$ en moyenne mais $O(n^2)$ en pire des cas. L'idée est donc de choisir un pivot de telle sorte que la taille des deux intervalles résultants soient au moins un cinquième de $b - a + 1$. Dans ce cas la complexité totale serait $O(n)$.
Grouper les éléments de $T[a..b]$ en paquets de 5 éléments, et au plus 5 pour le dernier paquet. Pour chaque paquet calculer le médian en temps constant. Soit M le tableau de ces $\lceil (b - a + 1)/5 \rceil$ medians. Recursivement appler *Select* pour trouver le médian de M, appelons le c. Maintenant c est un bon pivot, car il partitionnera $T[a...b]$ en deux parties donc chacun est au moins un cinquième de la taille de $T[a...b]$.

Implémentation

- Essayer d'abord avec un choix de pivot $c = \lfloor a + b)/2 \rfloor$.
- Si c'est trop long, essayez avec un choix aléatoire.
- Si c'est encore trop long, implémentez la recherche ci-haut.

Chapitre 10

Évaluer une expression

10.1 Construire l'arbre d'une expression

Dans l'exemple suivant on doit construire l'arbre d'une expression booléene. Il y a 26 variables $A, B, \ldots, Z$. La négation s'écrit en ajoutant le caractère #, le *et* logique par simple concaténation, et le *ou* logique est représenté par le +. On peut aussi utiliser des parenthèses. D'abord on définit une classe qui représente un neoud de l'arbre de l'expression.

```
class Expr {
    Expr x,y;
    char op;
    Expr(Expr _x, char _op, Expr _y) {
        x = _x;
        op=_op;
        y = _y;
    }
    int eval() { //−1=false, +1=true, 0=undefined
        switch (op) {
        case '+': return Math.max(x.eval(), y.eval()); // OR
        case '*': return Math.min(x.eval(), y.eval()); // AND
        case '−': return −x.eval(); // NOT
        default: // VAR
            return Main.val[op];
        }
    }
}
```

Puis on définit une grammaire

O = A | A + O
A = N | N A
N = V #*
V = *var* | (O)

où *var* représente un caractère entre 'A' et 'Z'. Cette grammaire est naturellement traduit en code. D'abord on dispose d'une chaîne s qu'on veut évaluer et une position p qu'on est en train de lire. La méthode *look* permet de lire ce qui se trouve à cette position, et *next* d'avancer la tête de lecture. Finalement *parse* initialise cette la lecture avec une chaîne source donnée.
Dans le code suivant chaque non-terminal de la grammaire correspond à une méthode, et le caractère retourné par *look* permet de déterminer avec certitude quelle règle doit être appliquée.

```
    static String s; // chaine representant une expr.
    static int p; // position dans s
```

```
static char look() {return p<s.length() ? s.charAt(p) : '\0'; }
static void next() {p++;}

static Expr parse(String _s) {
    s = _s;
    p = 0;
    return O();
}

static Expr O() {
    Expr e = A();
    while (look()=='+') {
        next(); // consommer le '+'
        Expr f = A();
        e = new Expr(e, '+', f);
    }
    return e;
}

static Expr A() {
    Expr f, e = N();
    while ((f=N())!=null)
        e = new Expr(e, '*', f);
    return e;
}

static Expr N() {
    Expr e = V();
    int p=0;
    if (e==null)
        return null;
    while (look()=='#') {
        next();
        p++;
    }
    if (p%2==1)
        return new Expr(e, '-', null);
    else
        return e;
}

static Expr V() {
    char c = look();
    if (c=='(') {
        next(); // (
        Expr e = O();
        next(); // )
        return e;
    }
    if ('A'<=c && c<='Z') {
        used[c] = true;
        next();
        return new Expr(null, c, null);
```

```
        }
        return null;
    }
```

10.2 Évaluer une expression arithmétique

Pour évaluer une expression arithmétique du genre "2+3*(10-1)" il faut procéder en deux étapes. D'abord il faut identifier les nombres, les variables et les opérateurs, puis il faut effectuer les opérations dans l'ordre imposé par leurs priorités. Dans l'exemple de cette section, on va considérer des expression simplifiées, où tous les nombres sont non-négatifs, et les seuls opérateurs sont l'addition et la multiplication (voir UVa 622 *Grammar Evaluation*). On a alors les règles suivantes. Il s'agit en fait de règles de réécriture. L'entier positif et les caractères (,),+,* sont appelés des *terminaux*, et ⟨expression⟩, ⟨component⟩, ⟨factor⟩ des *non-terminaux*. Les règles suivantes ont la bonne propriété que lors d'un unique parcours de gauche à droite, il suffit de regarder la prochaine unité lexicale pour savoir quelle règle doit être appliquée.

[entier positif]	:= [0-9]+
⟨expression⟩	:= ⟨component⟩ | ⟨component⟩ + ⟨expression⟩
⟨component⟩	:= ⟨factor⟩ | ⟨factor⟩ * ⟨component⟩
⟨factor⟩	:= [entier positif] | (⟨expression⟩)

L'idée est que pour créer une expression comme "7+3*(1+2)" on va appliquer successivement les règles :

⟨expression⟩
⟶ ⟨component⟩ + ⟨expression⟩
⟶ ⟨factor⟩ + ⟨component⟩
⟶ [entier positif] + ⟨factor⟩ * ⟨component⟩
⟶ 2 + [entier positif] * ⟨factor⟩
⟶ 2 + 3 * (⟨expression⟩)
...
⟶ 2 + 3 * (1 + 2)

Notez que l'arbre décrit par ces remplacements est aussi l'arbre syntaxique de l'expression. Notre but maintenant est d'effectuer l'opération inverse.

10.2.1 L'analyse lexicale

Un premier filtre découpe la chaîne en unités lexicales, appelées *tokens*. Dans l'exemple suivant, on ne distingue que les entiers, les identifiants, et caractères non-blancs. Voici le code de la classe MyScanner.

```
// le scanner decompose une chaine en un flux d'unites lexicales (tokens)
class MyScanner {
    // la source
    String s;

    // position de la tete de lecture
    int i;

    // retourne le caractere sous la tete de lecture
    char tete() {
        return i<s.length() ? s.charAt(i) : '\0';
    }
```

```
// saute les caracteres blancs
void sauteBlancs() {
    while (Character.isWhitespace(tete()))
        i++;
}

// se debrouiller pour que la tete soit toujours sur un non-blanc
MyScanner(String _s) {
    s = _s;
    i = 0;
    sauteBlancs();
}

// consomme un caractere, et attend que ce soit c
void next(char c) {
    if (tete() != c)
        abort("" + c + "' attendu");
    i++;
    sauteBlancs();
}

// consomme un entier [0-9]+
int nextInt() {
    if (! Character.isDigit(tete()))
        abort("entier attendu");
    int debut=i;
    while (Character.isDigit(tete()))
        i++;
    int fin =i;
    sauteBlancs();
    return Integer.parseInt(s.substring(debut, fin));
}

// consomme un identifiant [a-zA-Z][a-zA-Z0-9]*
String nextIdent() {
    if (! Character.isLetter(tete()))
        abort("identifiant attendu");
    int debut=i;
    if (Character.isLetter(tete())) {
        do {
            i++;
        } while (Character.isLetterOrDigit(tete()));
    }
    int fin =i;
    sauteBlancs();
    return s.substring(debut, fin);
}

// arrete tout avec un message d'erreur, indiquant la position de la tete
void abort(String msg) {
    System.err.println("Syntax error: "+msg);
    System.err.println(s);
    while (--i>0)
```

```
            System.err.print('␣');
        System.err.println("^");
        System.exit(1);
    }

}
```

10.2.2 L'analyse syntaxique

Pour évaluer l'expression, on va simplement se calquer sur la grammaire donnée. Chaque non-terminal correspond à une fonction, qui à partir d'un parser donné tente de reconnaître une partie de l'expression. Notez que l'arbre des appels de fonctions, correspond à l'arbre des expressions, et il est alors possible d'effectuer toutes sortes d'opérations sur cet arbre. Dans notre cas, on évalue l'expression reconnue, et on affecte sa valeur à une variable.

```
class Eval {
    // dictionnaire qui associe a des variables leur valeur
    static Map<String,Integer> var = new TreeMap<String,Integer>();

    static int factor(MyScanner s) {
        int v;
        if (Character.isDigit(s.tete()))
            return s.nextInt();
        if (Character.isLetter(s.tete())) {
            String id = s.nextIdent();
            if (!var.containsKey(id))
                s.abort("variable␣inconnue");
            return var.get(id);
        }
        s.next('('); // consommer le '('
        v = expression(s);
        s.next(')'); // consommer le ')'
        return v;
    }

    static int component(MyScanner s) {
        int v = factor(s);
        if (s.tete() == '*') {
            s.next('*');
            return v * component(s);
        }
        if (s.tete() == '/') {
            s.next('/');
            return v / component(s);
        }
        return v;
    }

    static int expression(MyScanner s) {
        int v = component(s);
        if (s.tete() == '+') {
            s.next('+');
            return v + expression(s);
        }
```

```
        if (s.tete() == '−') {
            s.next('−');
            return v − expression(s);
        }
        return v;
    }

    static void assignement(MyScanner s) {
        String id = s.nextIdent();
        s.next('='); // consommer le '='
        int v = expression(s);
        var.put(id, v);
        s.next('\0');
    }

    public static void main(String[] args) {
        String ligne;
        Scanner in = new Scanner(System.in);
        while (in.hasNext())
            assignement(new MyScanner(in.nextLine()));
        // afficher le contenu des variables
        for (String id: var.keySet())
            System.out.println(id+"\t="+var.get(id));
    }
}
```

Chapitre 11

Calculer

11.1 Les nombres premiers

- Des nombres premiers inférieurs ou égaux à n, il y en a $n/\ln n$ à la limite.
- Le crible d'Eratosthène : pour générer tous les nombres premiers inférieur ou égaux à n, partez d'un tableau avec les nombres $2, 3, \ldots, n$. Puis pour tout $i = 2, \ldots, n$: si i n'est pas barré, le marquer comme premier, et barrer tous les multiples de i dans le tableau. Une implémentation possible :

```
// au debut tout le monde est premier, sauf 0,1
for (int i=2; i<n; i++)
    p[i] = true;
for (int j=2; j<n; j++)
    if (p[j]) {
        // j est premier
        System.out.print(" "+j);
        for (int k=2*j; k<n; k+=j)
            p[k] = false; // k = multiple de j
    }
```

- Pour tester si un nombre n est premier, il suffit de vérifier qu'il ne peut être divisé par aucun des nombres premiers entre 2 et $\sqrt{n}$. Ceci donne lieu à un algorithme, où on génère une liste L de nombres premiers, et pour chaque i, on teste la primalité en divisant par les entiers de L jusqu'à $\sqrt{i}$ et on ajoute i à la fin de L, si i est premier. Cette méthode utilise moins d'espace que le crible d'Eratosthène, mais est bien moins efficace.

11.2 L'algorithme d'Euclide

C'est le premier algorithme qui fut inventé.

$$\operatorname{pgcd}(a,b) = \begin{cases} \operatorname{pgcd}(b, a \bmod b) & \text{si } b > 0 \\ |a| & \text{si } b = 0. \end{cases}$$

On a $\operatorname{pgcd}(a,b) = \operatorname{pgcd}(b,a)$ et $\operatorname{pgcd}(a,b) = \operatorname{pgcd}(-a,b)$. Pour l'analyse de sa complexité soit F_k le k-ème nombre de la suite de Fibonacci $(0, 1, 1, 2, 3, 5, 8, \ldots)$, où $F_k = F_{k-2} + F_{k-1}$ pour $k \geqslant 2$. Alors si $a > b \geqslant 0$ et $b < F_{k+1}$ alors il y a au plus k appels récursifs. Et cette limite est serré, car

$$\operatorname{pgcd}(F_{k+1}, F_k) = \operatorname{pgcd}(F_k, (F_{k+1} \bmod F_k)) = \operatorname{pgcd}(F_k, F_{k-1}).$$

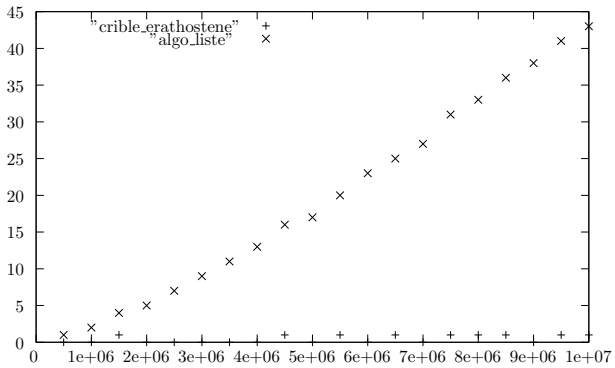


FIGURE 11.1 – Temps en secondes des 2 algorithmes en fonction de n.

La limite des appels est $O(\log F_k)$, où

$$\begin{aligned} F_k &= \frac{1}{\sqrt{5}}\left(\left(\frac{1+\sqrt{5}}{2}\right)^k - \left(\frac{1-\sqrt{5}}{2}\right)^k\right) \\ &\approx \frac{1}{\sqrt{5}}\left(\frac{1+\sqrt{5}}{2}\right)^k. \end{aligned}$$

Pour calculer le plus petit multiple commun, on peut se ramener à calculer $\text{ppmc}(x, y) = xy/\text{pgcd}(x, y)$.

11.3 Arithmétique modulo n

```
// calcule a ** b mod n
int exp(int a, int b, int n) {
    int res = 1;
    while (b != 0) {
        if (b%2 != 0)
            res = (res*a) % n;
        b = b/2;
        a = (a*a) % n;
    }
    return res;
}
```

On a

$$\begin{aligned} (x+y) \bmod n &= ((x \bmod n) + (y \bmod n)) \bmod n \\ (x \cdot y) \bmod n &= ((x \bmod n) \cdot (y \bmod n)) \bmod n. \end{aligned}$$

Pour calculer $a^b \bmod n$, on procède comme suit par la mise au carré répétée. Soit $b = b_0 2^0 + b_1 2^1 + \ldots b_k 2^k$. Alors

$$\begin{aligned} a^b &= a^{b_0 2^0 + b_1 2^1 + \ldots b_k 2^k} \\ &= a^{b_0 2^0} a^{b_1 2^1} \cdots a^{b_k 2^k} \\ &= (a^{2^0})^{b_0} (a^{2^1})^{b_1} \cdots (a^{2^k})^{b_k}. \end{aligned}$$

Donc il suffit de calculer itérativement les différentes puissances de a, et de les multiplier au résultat si le bit correspondant dans b est 1.

11.4 Résoudre un système d'équations linéaire

Considérons une marche aléatoire sur un graphe dirigé. Les arcs (u, v) sont étiquetés avec des probabilités p_{uv} de faire une transition de u à v en une étape. On suppose $\sum_v p_{uv} = 1$ pour tout u. On distingue un sommet u_0, qui est le point de départ au temps 0. Maintenant on vous demande de trouver pour chaque sommet v, le temps espéré d'atteindre v en partant de la position u_0. Soient x_v ces temps, alors $x_{u0} = 0$ par définition. Et sinon on a

$$x_v = \sum_u (1 + x_u) p_{uv}.$$

Donc on a n inconnus et n égalités qui forment un système d'équations linéaires. Pour le résoudre on le considère sous la forme $Mx = b$ où x est le vecteur colonne des n variables, b est la partie constante dans chacune des m égalités (dans notre cas $m = n$). M est la matrice qui contient les coefficients de chaque variable dans chaque égalité. Idéalement, si M était la matrice d'identité, alors b contiendrait tout simplement les valeurs pour x, la solution au système. On va transformer M dans cette forme par l'élimination de Gauss.
Invariant : au bout de k itérations, pour $j = 1, \ldots, k$, $M_{j,j} = 1$ et $M_{i,j} = 0$ pour $i = 1, \ldots, i-1, i+1, \ldots, m$. Pour l'itération $k+1$, on cherche dans la sous-matrice des lignes $k+1$ à m et des colonnes $k+1$ à n une entrée $M_{i'j'}$ non-nulle, de préférence la plus grande en valeur absolu, cela diminuera les erreurs d'arrondis. Si cette sous-matrice est nulle ou vide ($k = n$ ou $k = m$), alors de deux choses l'une. Soit il existe une ligne $i \geqslant k$ avec $b_i \neq 0$ et dans ce cas il n'y a pas de solution. Sinon une solution est de poser $x_i = b_i$ pour $i = 1, \ldots, k$ et $x_i = 0$ pour $i > k$.

$$M = \begin{pmatrix} 1 & 0 & 0 & \cdot & \cdot & \cdot \\ 0 & 1 & 0 & \cdot & \cdot & \cdot \\ 0 & 0 & 1 & \cdot & \cdot & \cdot \\ 0 & 0 & 0 & \cdot & \cdot & \cdot \\ 0 & 0 & 0 & \cdot & \cdot & \cdot \\ 0 & 0 & 0 & \cdot & \cdot & \cdot \end{pmatrix} \quad b = \begin{pmatrix} \cdot \\ \cdot \\ \cdot \\ \cdot \\ \cdot \\ \cdot \end{pmatrix}$$

FIGURE 11.2 – Structure de l'invariant pour $k = 3$

1. Maintenant supposons qu'il existe une entrée $M_{i'j} \neq 0$. On la ramène à $M_{k+1,k+1}$, en échangeant les lignes i' et $k+1$ (échanger aussi $b_{i'}$ et b_{k+1}) et en échangeant les colonnes j' et $k+1$ (échanger aussi $x_{j'}$ et x_{k+1}).
2. Ensuite on multiplie la ligne $k+1$ par $1/M_{k+1,k+1}$ pour mettre 1 en $M_{k+1,k+1}$.
3. Puis on soustrait cette même ligne des autres lignes $i \neq k+1$ avec coefficient $M_{i,k+1}$, ce qui va avoir pour effet de mettre 0 dans toute la colonne $k+1$, sauf en $M_{k+1,k+1}$ et ainsi préserver l'invariant. Ces opérations sur les lignes doivent inclure b.

Si le graphe est éparse, comme dans le cas d'une grille ou d'un cycle, alors il n'y a qu'un nombre limité d'entrées non-nulles par ligne et par colonne. Si on organise bien son programme et qu'on exploite la structure particulière, alors on peut réduire le temps de calcul de $O(n^2 m)$ à $O(n)$.

```
class GaussJordan {

    /* elimination de gauss−jordan,
```

```
 * tente de resoudre Mx=b. En cas de succes retourne alors
 * la solution, sinon retourne null
 *
 * Si vous voulez seulement savoir s'il existe une solution,
 * alors vous pourriez inclure b en M comme une derniere colonne,
 * et rendre le programme plus simple.
 */
static double[] solve(double[][] M, double[] b) {
    int m = M.length; // nombre d'equations
    int n = M[0].length; // nombre de variables
    int K = Math.min(n, m);

    // x contains the variable indices
    int[] x = new int[n];
    for (int j=0; j<n; j++)
        x[j] = j;

    // invariant: la sousmatrice k*k des k premiers lignes et
    // colonnes est la matrice d'identite
    for (int k=0; k<K; k++) {
        // trouver i,j>=k qui maximize | M[i][j] |
        int imax=k, jmax=k;
        for (int i=k; i<m; i++)
            for (int j=k; j<n; j++)
                if (Math.abs(M[i][j]) > Math.abs(M[imax][jmax])) {
                    imax = i;
                    jmax = j;
                }

        // est-ce que les lignes k,..,n-1 sont tous 0 ?
        if (M[imax][jmax]==0.0) {
            for (int i=k; i<m; i++)
                if (b[i]!=0.0)
                    return null; // on a trouve une contradiction
            // on a trouve une solution,
            // et la matrice a le rang k en fait eu lieu de K
            K = k;
            break;
        }

        // echanger lignes imax et k
        double[] tmp = M[imax];
        M[imax] = M[k];
        M[k] = tmp;

        double t0 = b[imax];
        b[imax] = b[k];
        b[k] = t0;

        // echanger colonnes jmax et k
        for (int i=0; i<m; i++) {
            t0 = M[i][jmax];
            M[i][jmax] = M[i][k];
            M[i][k] = t0;
```

```
            }

            int t1 = x[jmax];
            x[jmax] = x[k];
            x[k] = t1;

            // diviser ligne k par M[k][k]
            double div = M[k][k];
            for (int j=k; j<n; j++)
                M[k][j] /= div;
            b[k] /= div;

            // soustraire ligne k des lignes i avec coefficient adapte
            for (int i=0; i<m; i++)
                if (i!=k) {
                    double fact = M[i][k];
                    for (int j=k; j<n; j++)
                        M[i][j] -= fact * M[k][j];
                    b[i] -= fact * b[k];
                }
        }
        // recopier la solution a partir de b
        double[] sol = new double[n];
        for (int j=0; j<n; j++)
            sol[j] = 0.0;
        for (int j=0; j<K; j++)
            sol[x[j]] = b[j];
        return sol;
    }

    [...]
}
```

11.5 Multiplier des matrices circulaires

Imaginez des cellules sur un cercle, où chaque cellule contient un nombre. Soit $x_0, \ldots, x_{n-1} \in \mathbb{N}$ la configuration initiale. Maintenant en une itération, chaque cellule i met à jour son nombre en remplaçant x_i par $Lx_{i-1} + Cx_i + Rx_{i+1}$, où la soustraction, l'addition sur i se fait modulo n. L, C, R sont des constantes qui font partie de l'instance du problème. Ces mises à jour se font simultanément sur toutes les cellules, qui calculent leur valeur en fonction des valeurs de l'itération précédente. Maintenant on vous donne un entier m et on vous demande de calculer la configuration au bout de m itérations. Donc il faut calculer le produit matriciel

$$\underbrace{\begin{pmatrix} C & R & 0 & 0 & L \\ L & C & R & 0 & 0 \\ 0 & L & C & R & 0 \\ 0 & 0 & L & C & R \\ R & 0 & 0 & L & C \end{pmatrix} \cdots \begin{pmatrix} C & R & 0 & 0 & L \\ L & C & R & 0 & 0 \\ 0 & L & C & R & 0 \\ 0 & 0 & L & C & R \\ R & 0 & 0 & L & C \end{pmatrix}}_{m} \begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ x_4 \end{pmatrix}.$$

D'une part si m peut être assez grand, on aura envie d'utiliser la mise en carrée répétée de la matrice $n \times n$, au lieu de faire m multiplications, voir section 11.3. Mais d'autre part la matrice est très régulière, chaque ligne est égale à la ligne précédente décalé à droite de manière

circulaire. On dit que la matrice est *circulaire*. Or le produit de deux matrices circulaires est aussi circulaire, c'est facile à vérifier. En utilisant une représentation compacte, on peut effectuer une multiplication en temps $O(n^2)$ au lieu de $O(n^3)$.
Donc si $[a_0, a_1, \ldots, a_{n-1}]$ représente la matrice circulaire dont la première ligne est $(a_0 \ldots a_{n-1})$, alors le produit sera

$$[a_0, a_1, \ldots, a_{n-1}] \cdot [b_0, b_1, \ldots, b_{n-1}] = [c_0, c_1, \ldots, c_{n-1}],$$

avec $c_i = \sum_j a_j b_{i-j}$. Attention il faut coder $(i-j) \bmod n$ par `(i+n-j)%n`, car en Java le modulo d'un entier négatif est aussi négatif.

Problèmes Top Secret [swerc :2008, uva :12183]

11.6 Le test de Freivalds

Étant donnée trois matrices A, B, C carrés de dimension $n \times n$, on veut décider si $AB = C$. Un algorithme naïf prendra un temps $O(n^3)$, le temps de multiplier A avec B.
Freivalds a montré en 1979 que si A et B étaient différentes, alors $Av \neq Bv$ avec probabilité au moins $1/p$ lorsque v est un vecteur aléatoire dont chaque coordonnée est dans $\mathbb{Z}_p$. Pour nous p sera donnée par la taille d'un mot machine, par exemple 2^{32}, ce qui donnera une probabilité d'erreur suffisament petite.
La complexité de ce test est alors $O(n^2)$ seulement pour vérifier $A(Bv) = Cv$.

Problèmes Comparing answers [Swerc2010]

11.7 Les chiffres dans les chiffres et les lettres

Entrée $n+1$ entiers $a_1, \ldots, a_n, K$ pour n petit, disons $n \leq 10$.

Sortie Une expression arithmétique valant le plus proche de K et n'utilisant chacun des entiers au plus une fois, et utilisant les opérations +,-,* et /. Ici la division n'est autorisé que si la divison est sans reste.

Complexité $O(K^2 2^{2n})$ par exploration exhaustive

Algorithme Soit $E[S,k]$ une expression qu'on peut former avec tous les éléments de $\{a_i | i \in S\}$ et qui soit de valeur exactement k, et $E[S,k]$=null si une telle expression n'existe pas. Alors

```
E = tableau de dimension 2^n.
-- Les éléments de E sont des dictionaires associant des entiers aux expressions.

Pour tout i=0,..,n-1:
  E[{i}][a[i]] = nouvelle Expression(a[i])
Pour tout ensemble S:
  E[S] = nouveau dictionaire vide
  Pour toute partition de S en S1,S2:
    pour toute clé k1 dans S1:
      pour toute clé k2 dans S2:
        E[S][k1+k2] = nouvelle Expression(E[S][k1], '+', E[S][k2])
        E[S][k1*k2] = nouvelle Expression(E[S][k1], '*', E[S][k2])
        E[S][k1-k2] = nouvelle Expression(E[S][k1], '-', E[S][k2])
        si k2 divise k1:
```

```
          E[S][k1/k2] = nouvelle Expression(E[S][k1], '/', E[S][k2])
-- lire la solution
trouver le couple S,k qui maximise k et tel que E[S][k] ne soit pas null
retourner E[S][k]
```

Chapitre 12

Exploration de graphes

Pour les composantes connexes voir 4.2.

12.1 Structures de données pour représenter des graphes

En général on représente les graphes par le graphe orienté associé, où chaque arête (u, v) est représentée par les arcs (u, v) et (v, u).
La plupart du temps on préfère la représentation des graphes par liste d'adjacence, qui est facile à manipuler — facilite le parcours des voisins d'un sommet — et efficace si le graphe est *éparse*, c'est à dire contient seulement $o(|V|^2)$ arêtes. On liste pour chaque sommet $v \in V$ tous les voisins de v dans un ordre arbitraire. Si le graphe est pondéré, alors on a envie de mettre dans une liste les couples $(u, w(v, u))$.
En général on vous donne le nombre de sommets en début de l'entrée. On aimerait alors identifier les sommets par les entiers de 0 à $n - 1$. On pourrait imaginer stocker le graphe dans ArrayList<Integer>[] V, de sorte que V[u] contienne les voisins de u. Mais il y a un problème énervant. Enfin, je ne sais pas pourquoi je dis ça, tous les problèmes sont énervants par définition. En Java c'est très difficile de créer un tableau dont les éléments sont d'un type générique. Une manière de contourner ce problème énervant, serait de définir une classe class Sommet extends ArrayList<Integer>. Mais il y a un autre petit problème, il faut alors définir le champ serialVersionUID.
Donc voici la représentation que je propose. On va identifier les sommets par les entiers. Si on veut attacher des étiquettes aux sommets, on va définir un tableau à part. Ensuite si on veut travailler par matrice d'adjacence, on va définir un tableau de booléens à deux dimensions. Si on veut travailler par liste d'adjacence, deux possibilités s'offrent à vous. Si vous connaissez le degré d'avance de chaque sommet, on pourrait — si la place le permet — de nouveau définir un tableau à deux dimensions, de telle sorte que ADJ[U] contient les voisins de u. Donc ce serait un tableau avec $|V|$ lignes, mais chaque ligne a une longueur différente suivant le degré. Cette solution est envisageable en Java, mais pas en C++. Dans le cas où vous ne connaissez pas les degrés en avance, vous pourriez représenter les extrémités des arcs par liste d'adjacence, où toutes les listes sont stockées dans un même tableau. Ici :

- arc[i] est l'extrémité du i-ème arc.
- Le premier arc avec origine u a l'indice ADJ[U].
- Le deuxième NEXT[ADJ[U]], le troisième NEXT[NEXT[ADJ[U]], jusqu'à l'indice -1 qui indique la fin de la liste.

Cette représentation a l'avantage d'être compacte en mémoire.

```
static int n, m; // nombre de sommets, nombre d'arcs deja ajoutes
```

```
static int[] arc, adj, next;

static void init(int _n, int maxm) { // maxm = 2*|E|
    n = _n;
    arc =new int[maxm];
    adj =new int[n];
    next=new int[maxm];
    m = 0;
    Arrays.fill(adj, −1);
}

static void addArc(int u, int v) {
    arc[m] = v;
    next[m] = adj[u];
    adj[u] = m;
    m++;
}
```

Boucler sur tous les voisins de u s'écrit alors :

```
for (int i=adj[u]; i!=−1; i=next[i]) {
    int v = arc[i];
    // [...]
}
```

12.2 Explorations

Vous connaissez quatre algorithmes d'exploration de graphes, qui ont tous la forme suivante. S est l'ensemble des sommets déjà parcourus, A est un arbre qui couvre S, et Q l'ensemble des arêtes à explorer. Un détail technique : Un arbre est généralement représenté comme un ensemble d'arêtes, mais pour $S = \{v_0\}$, on va dire que l'arbre qui couvre S est (v_0, v_0).

> Étant donné $G(V, E)$.
> Soit $S = \emptyset \subseteq V, A = \emptyset \subseteq E$ et $Q = \emptyset \subseteq E$.
> Ajouter (v_0, v_0) à Q pour un sommet $v_0 \in V$.
> Tant que Q est non-vide :
> sortir un élément (u, v) de Q
> si $v \notin S$:
> ajouter v à S et (u, v) à A
> pour tous les voisin w de v tel que $w \notin S$:
> ajouter (v, w) à Q
> S est une composante connexe de G et A un arbre qui couvre S.
> Si $S \neq V$, alors recommencer la procédure sur le graphe induit par $V \backslash S$.

1. Si Q est une pile, alors vous avez le parcours en profondeur.
2. Si Q est une file FIFO, alors vous avez le parcours en largeur.
3. Supposons qu'une pondération des arêtes $w : E \to \mathbb{Z}$ soit donnée. Si Q est une file de priorité et que l'élément (u, v) extrait de Q soit celui qui minimise le poids $w(u, v)$, alors vous avez l'algorithme de Prim pour calculer l'arbre de recouvrement minimum.
4. Maintenant supposez que l'algorithme maintient une fonction $d : S \to \mathbb{N}_0$, qui initialement vaut $d(v_0) = 0$, et que la pondération soit non-négative, i.e. $w : E \to \mathbb{N}_0$. Et si Q est de

nouveau une file de priorité, où l'élément (u, v) extrait de Q soit celui qui minimise $d(u) + w(u, v)$, alors vous avez l'algorithme de DIJKSTRA pour calculer les plus courts chemins de v_0 vers tous les autres sommets du graphe, et d est la fonction de distance.

En général on préfère implémenter le parcours en profondeur plutôt que le parcours en largeur, car on peut se passer de maintenir une pile. Celle-ci sera implicite par la pile des appels récursifs. Par contre c'est intéressant de faire le parcours en largeur pour trouver un chemin entre deux sommets dans un graphe infini. Ce parcours a la propriété sympathique que si u est plus proche de v_0 que v alors u va être visité plus tôt que v. C'est donc intéressant quand on veut découvrir en priorité les chemins courts en terme de nombre d'arêtes (voir section 16.3).

12.3 Propriétés du parcours en profondeur

Le parcours en profondeur classe les arcs en quatre catégories.

1. Les *arcs de liaison*, sont les arcs suivis par le parcours.

2. Les *arcs retour* sont les arcs (u, v) qui relient un sommet u à un ancêtre de v. Les boucles (u, u) sont considérés comme des arcs retour.

3. Les *arcs avant* sont les arcs (u, v) qui relient un sommet u à un descendant de v, et qui ne sont malgré tout pas des arcs de liaison.

4. Les *arcs couvrants* sont tous les autres arcs.

Théorème 2 *Soit d_u la date du début du traitement du sommet u par le parcours en profondeur et f_u la date de fin de traitement. Alors seulement une des trois conditions suivantes est vérifiée :*

1. *Si $[d_u, f_u] \subseteq [d_v, f_v]$ alors v est un descendant de u dans l'arbre produit.*

2. *Si $[d_u, f_{u]} \supseteq [d_v, f_v]$ alors u est un descendant de v.*

3. *Si $[d_u, f_u]$ et $[d_v, f_v]$ sont disjoints, alors ni u ni v sont descendants l'un de l'autre.*

Théorème 3 *Lors d'un parcours en profondeur, un sommet v est descendant d'un sommet u si et seulement si en début de traitement de u (au temps d_u), il existe un chemin de u à v composé seulement de sommets* blancs *(pas encore visités).*

12.4 Bipartition d'un graphe

Un graphe $G(V, E)$ est biparti si les sommets peuvent être colorés en noir et blanc, tel qu'il n'existe pas d'arête entre deux sommets de même couleur.

Théorème 4 *Un graphe $G(V, E)$ est biparti si et seulement s'il n'existe pas de cycle de longueur impaire.*

C'est facile de décider si un graphe est biparti en temps linéaire $O(|V| + |E|)$. Tout simplement faites un parcours en profondeur avec un paramètre *profondeur*. Il y a un cycle de longueur impaire si vous tombez sur une arête dont les extrémités ont la même parité de profondeur. Sinon la parité de la profondeur donne une bipartition du graphe. En général il y a 2^t colorations bipartitions possibles où t est le nombre de composantes connexes.

12.5 Le tri topologique

Étant donné un ordre partiel sur V, qu'on peut voir comme un arbre orienté sans cycle $G(V, A)$ on veut trouver un raffinement total de cet ordre. En fait on veut trouver une numérotation des sommets de V en $v_1, \ldots, v_n$, tel que si $(v_i, v_j) \in A$ alors $i < j$.

1. Faire un parcours en profondeur de $G(V, A)$. Noter $f[v]$ la date de fin de traitement du sommet v.

2. L'ordre topologique est la liste des sommets V en ordre décroissant de $f[v]$.

Concrètement on va insérer v en début d'un liste à la fin de son traitement. Ce même algorithme permet aussi de savoir si un graphe est acyclique.

Lemme 3 *Un graphe orienté G est acyclique si et seulement si un parcours en profondeur ne génère aucun arc retour.*

Problèmes Ordering Tasks [uva :10305]

12.6 Les composantes fortement connexes

Un ensemble $U \subseteq V$ d'un graphe orienté est fortement connecté si pour tout $u, v \in U$ il existe un chemin (orienté) de u à v et aussi de v à u. Voici un algorithme de **Tarjan** pour calculer les composantes fortement connexes d'un graphe. Les composantes peuvent être reliés par des arcs. Mais bien sûr si on contracte les arcs en des sommets simples, on obtient un graphe acyclique. On note $A^T := (v, u)|(u, v) \in A$ l'inversion des arcs.

1. Faire un parcours en profondeur de $G(V, A)$. Noter $f_v[v]$ la date de fin de traitement du sommet v.

2. Faire un parcours en profondeur de $G(V, A^T)$, en choisissant les racines v dans l'ordre de f_v décroissant.

3. Chaque arbre rencontré dans ce deuxième parcours est une composante fortement connexe.

La correctude de l'algorithme repose sur l'idée que si on associe à chaque composante fortement connexe C l'entier $F(C) := \max_{u \in C} f_u$, alors F donne un ordre topologique sur le graphe induit par les composante fortement connexe dans $G(V, A^T)$. Ainsi chaque arbre produit dans le deuxième parcours reste à l'intérieur d'une composante, car les seuls arcs sortants mènent à des composantes déjà parcourues.
Voici une implémentation de cet algorithme. Chaque sommet comporte un champs comp. Ce champs a une double fonction, il contient -1 quand le sommet n'a pas encore été visité, et pour les sommets déjà visités, il contiendra un numéro d'ordre : l'heure du début de traitement et enfin il devrait à la fin indiquer le numéro de la composante fortement connecté. Pour être précis, le numéro d'une composante sera l'indice du premier sommet visité, l'*élément canonique*. Pour ne pas confondre avec les numéros de composantes, l'heure commencera à $-(|V| + 1)$, et sera incrémenté à chaque début de traitement d'un sommet.
On maintient aussi une pile aTraiter, qui contient tous les sommets pour lesquels on n'a pas encore établi leur composante. La méthode sccp(u), retourne le plus petit numéro d'ordre de tous les sommets qu'on peut atteindre à partir de u, en ignorant les composantes déjà trouvés. C'est ici qu'intervient le fait que tout numéro d'ordre est plus petit qu'un numéro de composante. Si le minimum sur sccp(v) sur tous les voisins de u est justement u, alors on sait que u est l'élément canonique de sa séquence, et tous les sommets sur la pile aTraiter, qui ont été mis depuis le début de traitement appartiennent à la composante.

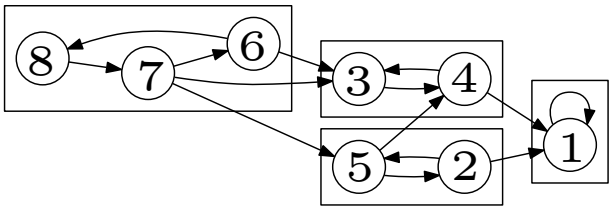


FIGURE 12.1 – Les composantes fortement connexes d'un graphe. Les sommets u sont étiquetés par la date de fin de traitement f_u.

```
static int n; //nb literaux, variable xi=2*i, literal non(xi)=2*i+1
                                        //pour un literal y, non(y)=y^1
static boolean adj[][]; // adj[x][y] = il existe un arc x=>y

static int o[]; // o[i] = i-eme sommet traite par dfs
static boolean deja[]; // marqueur pour dfs
static int time;
static int sccp[]; // sccp[u]=numero composante fort. connex. de u

// parcours dfs normal, noter dans o l'ordre fin visite
static void dfs(int u) {
    deja[u] = true;
    for (int v=0; v<n; v++)
        if (adj[u][v] && !deja[v])
            dfs(v);
    o[time++] = u;
}

// parcours dfs en inversant les arcs, mettre tout le monde dans meme comp.
static void dfsInv(int u, int comp) {
    deja[u] = true;
    sccp[u] = comp;
    for (int v=0; v<n; v++)
        if (adj[v][u] && !deja[v])
            dfsInv(v, comp);
}

// calculer les composantes fortement connexes, et retourner leur nombre
static int sccp() {
    time = 0;
    deja = new boolean[n];
    o = new int[n];
    for (int u=0; u<n; u++)
        if (!deja[u])
            dfs(u);

    int comp=0;
    deja = new boolean[n];
    sccp = new int[n];
    for (int i=n-1; i>=0; i--) {
        int u = o[i];
        if (!deja[u])
```

```
            dfsInv(u, comp++);
        }
        return comp;
    }
```

12.7 Résoudre une formule booléenne 2-SAT

Beaucoup de problèmes peuvent se modéliser comme un problème de satisfaction d'une formule booléene. Le problème de satisfaction est le suivant. On dispose de n variables booléennes. L'ensemble des variables et de leur négation est appelé l'ensemble des littéraux. Une clause est une disjonction de plusieurs littéraux. Une clause est satisfaite si un des littéraux est vrai. Une formule est la conjonction de plusieurs clauses. Une formule est satisfaite si tous les clauses sont satisfaites. Le but est de trouver une assignation aux variables qui satisfait la formule.
Une formule est appelé 2-SAT si chaque clause contient au plus deux littéraux. Un des résultats fondamentaux est qu'on peut vérifier en temps linéaire si une formule 2-SAT est satisfiable, alors qu'en général, on ne sait pas faire beaucoup mieux que de tester les 2^n assignations possibles.
Peut-être vous vous rappelez que $x \vee y$ est équivalent à $\overline{x} \Rightarrow y$ voir $\overline{y} \Rightarrow x$. On associe a une formule le graphe des implications, où les sommets sont les littéraux, et les arcs les implications. Ce graphe présente une forte symétrie.

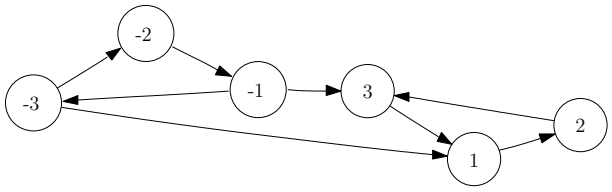


FIGURE 12.2 – Le graphe associé à la formule $(\overline{x_1} \vee x_2) \wedge (\overline{x_2} \vee x_3) \wedge (\overline{x_3} \vee x_1) \wedge (x_1 \vee x_3)$ où par exemple -2 représente $\overline{x_2}$

Théorème 5 *Une formule 2-SAT est satisfiable si et seulement si dans le graphe d'implication il n'existe pas de chemin de x à $\overline{x}$ et de $\overline{x}$ à x pour une variable x.*

Les implications étant transitives, il est clair que si $x \Rightarrow \overline{x} \Rightarrow x$ est impliqué par la formule, alors elle ne peut pas être satisfiable. Supposons que la formule soit satisfiable. Alors les composantes fortement connexes viennent par paires, une contient toutes les négations des littéraux de l'autre. Et tous les littéraux dans une même composante doivent avoir la même valeur booléene. Ainsi pour trouver une assignation qui satisfait la formule, il suffit de poser à vrai une composante dont aucun arc n'en sort. Il y en a forcement une, car le graphe des composantes est sans cycle. Ensuite on pose à faux la composante opposée, on enlève ces deux composantes du graphe et on recommence.
Pour faire cela efficacement, notons que l'algorithme qui trouve les composantes fortement connexes, les trouve dans l'ordre topologique inverse. Il suffit alors d'y ajouter le traitement suivant. Quand une composante a été trouvée, si le littéral associé au sommet canonique n'a pas encore une valeur, alors on peut lui affecter la valeur `true`, ainsi qu'a tous les littéraux de cette composante, et affecter FALSE à la composante opposée. Voici une implémentation possible.

```
    static int nc; // nb composantes fortement connexes
    static int val[]; // valeur d'une comp. +1=vrai, −1=faux, 0=indef.
    static int neg[]; // neg[j]=composante opposee de j

    // retourne true si la formule est satisfiable
    // dans ce cas, pour chque literal x sa valeur est val[sccp[x]]
```

```
static boolean satisfiable() {
    nc = sccp();
    neg = new int[nc];

    // est-ce qu'il existe une variable x et sa negation non(x)
    // qui soient dans la meme composante ?
    for (int x=0; x<n; x+=2)
        if (sccp[x] == sccp[x^1])
            return false;
        else {
            neg[sccp[x]] = sccp[x^1];
            neg[sccp[x^1]] = sccp[x];
        }

    // on utilise le fait que les composantes portent un numero dans
    // l'ordre topologique inverse
    val = new int[nc];
    for (int j=0; j<nc; j++)
        if (val[j]==0) {
            val[j] = -1;
            val[neg[j]] = +1;
        }
    return true;
}
```

Problèmes Manhattan [uva :10319]

12.8 Points d'articulations et ponts

- Un *point d'articulation* d'un graphe non-orienté connexe est un sommet dont la suppression déconnecte le graphe.
- Un *pont* est une arête dont la suppression déconnecte le graphe.
- Les arêtes qui ne sont pas des ponts se partionnent en *composantes biconnexes*. Une telle composante est un ensemble maximal d'arêtes, tel que dans le graphe induit (restreint à cet ensemble) pour tout couple de sommets u, v il existe deux chemins sommet-disjoints de u à v.

Un ingrédient important pour la détermination des objets précédents, est le calcul du *low point*. Qu'est-ce que c'est. Un premier parcours DFS construit des arbres couvrants chaque composante connexe, et partionne les arêtes en arcs de liaison (ceux de l'arbre), arcs de retour et arcs en avant. C'est expres qu'on utilise maintenant des arcs dirigés, l'orientation (u, v), indique que lors du traitement de u un voisin v de u a été considéré. Alors si on note d_u la date de la première visite de u, un arc (u, v) est un arc de retour si $d_u > d_v$ et que (v, u) n'est pas un arc de liaison. Maintenant low[u] est défini comme la valeur minimale d_v, pour tout arc retour (w, v) où w est un sommet du sous-arbre enraciné en u, donc où w peut être obtenu par u seulement avec des arcs de liaison.
Une fois calculé ces valeurs *low*, on peut trouver les points d'articulations, les ponts et les composantes bi-connexes par un deuxième parcours en profondeur. L'algorithme est basé sur ces observations.

1. Un sommet u qui est une racine de l'arbre DFS est un point d'articulation si et seulement si elle possède au moins deux fils dans l'arbre. Notez que chaque fils v satisfait low[v] $\geq d_u$.

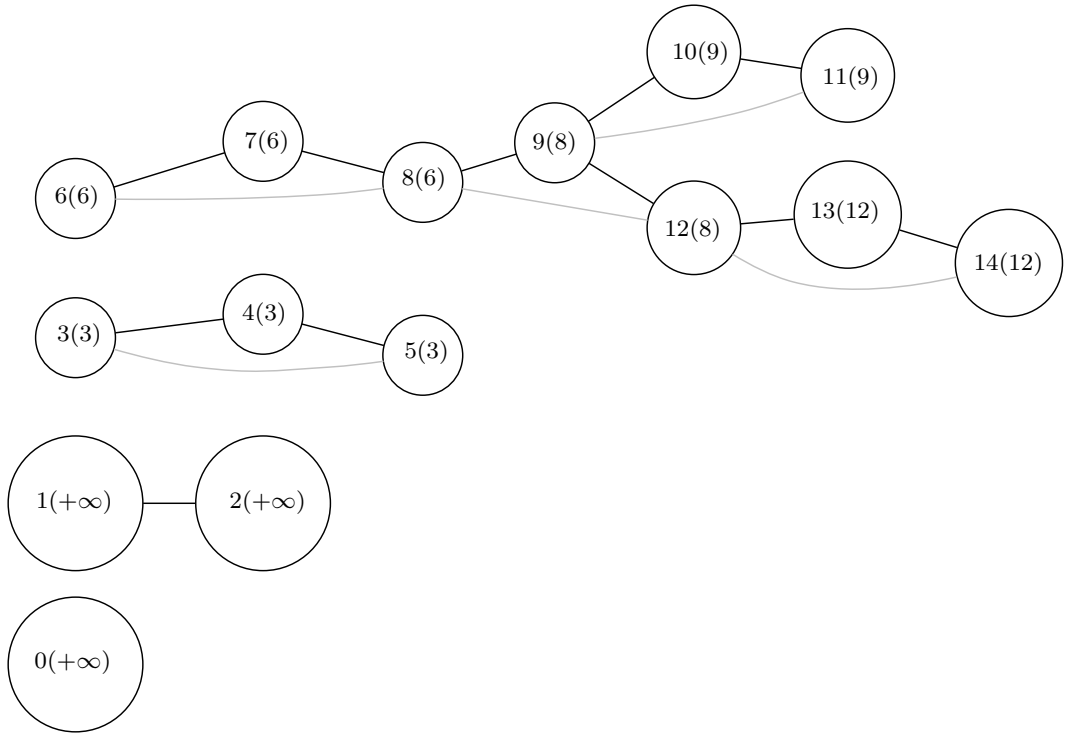


FIGURE 12.3 – En gras les arêtes de liaison. Les sommets u sont étiquetés par d_u suivi de low$[u]$ entre parenthèses..

2. Un sommet u qui n'est pas une racine de l'arbre DFS est un point d'articulation si et seulement si elle possède au moins un fils v avec low$[v] \geq d_u$.

3. Une arête (u, v) est un pont si et seulement si — à échange de u et v près — si (u, v) est une arc de laison et low$[v] \geq d_v$.

Pour déterminer les composantes bi-connexes, il suffit alors d'appliquer les définitions ci-dessus pour déterminer à quel moment débute une nouvelle composante bi-connexe.

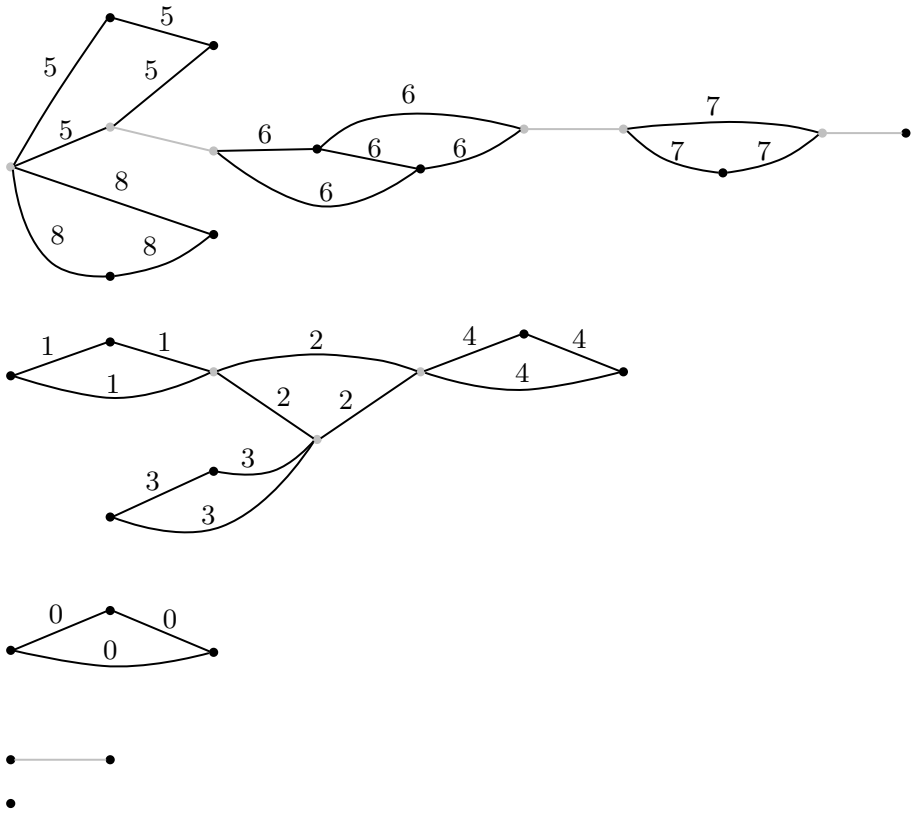


FIGURE 12.4 – Les arêtes sont étiquetées par un identifiant de composante biconnexe. Les pont et points d'articulations sont grisés.

```
static void solve() {
    dfs1();
    dfs2();
}

// premier parcours DFS pour calculer low
static void dfs1() {
    d = new int[V];
    T = new int[V];
    low = new int[V];
    Arrays.fill(d,NEW);
    time = 0;
    for (int u=0; u<V; u++)
        if (d[u]==NEW)
            dfs1(u, NEW);
    // initier un nouvel arbre pour composante connexe contenant u
}

// p = pere de u dans l'arbre. NEW si u = racine
static void dfs1(int u, int p) {
    T[u]=p; // noter le pere du u (optionnel)
    d[u]=time++; // marquer ordre premiere visite
    // low = destination minimale arete retour
```

```
        low[u]=Integer.MAX_VALUE;
        for (int v:N[u])
            if (d[v]==NEW) {
                dfs1(v, u);
                if (low[v]<low[u])
                    low[u] = low[v];
            }
            else
                if (v!=p && d[v]<low[u])
                    low[u] = d[v];
    }

    static void dfs2() {
        B = 0; // nb composantes bi−connexes
        deja = new boolean[V];
        isArtPoint = new boolean[V];
        bcc = new int[V][V];
        for (int u=0; u<V; u++)
            if (!deja[u])
                dfs2(u, NEW, NEW);
    }

    // c = numero de composante, si c=NEW alors composante encore vide
    // p = pere de u dans l'arbre DFS, si p=NEW alors u est une racine
    static void dfs2(int u, int p, int c) {
        deja[u] = true;
        int nbSonsDisc = 0; // nombre de fils qui seraient deconnectes sans u
        for (int v: N[u])
            if (!deja[v]) {
                if (low[v]>=d[v]) { // (u,v) est un pont
                    bcc[u][v] = bcc[v][u] = BRIDGE;
                    nbSonsDisc++;
                    dfs2(v, u, NEW);
                    // ici commence une nouvelle comp.
                }
                else if (low[v]>=d[u] || p==NEW) {
                    // nouvelle composante bi−connexe
                    int b = bcc[u][v] = bcc[v][u] = B++;
                    nbSonsDisc++;
                    dfs2(v,u,b);
                } else { // meme composante
                    assert(c!=NEW && p!=NEW);
                    bcc[u][v] = bcc[v][u] = c;
                    dfs2(v,u,c);
                }
            }
            else if (v!=p && d[v]<d[u]) { // arete retour
                assert(c!=NEW && p!=NEW);
                bcc[u][v] = bcc[v][u] = c;
            }
        isArtPoint[u] = p==NEW && nbSonsDisc>=2 || p!=NEW && nbSonsDisc>=1;
    }
```

Chapitre 13

Plus courts chemins

Les algorithmes présentés ne calculent que la distance à partir de la source. Pour obtenir le plus court chemin qui réalise cette distance, maintenez un tableau $p : V \to V \cup \perp$, initialisé à $\perp$. Et quand vous mettez à jour $d[v]$, mettez aussi à jour $p[v] := u$. Dans ce cas $p[v]$ pointe sur le prédécesseur dans le plus court chemin.

13.1 Labyrinthes

Un labyrinthe est en général un sous-graphe de la grille, parfois avec des étiquettes sur les sommets ou sur les arcs ou arêtes. Exploitez la structure de la grille pour simplifer votre code. Pour traiter les problèmes de bord, évitez des traitement de cas avec pleins de *if* ou de *switch*.

```
int x2[] = {x−1, x, x+1, x };
int y2[] = {y, y−1, y, y+1};

for (int i=0; i<4; i++) {
    int vx = x2[i], vy=y2[i];
    if (0<=vx && vx<n && 0<=vy && vy<m && grille[vx][vy]!=MUR) {
        //[...]
    }
}
```

13.2 Arêtes sans poids

Quand il s'agit seulement de trouver le plus court chemin entre deux positions, un parcours BFS fera l'affaire. Il prend un temps linéaire. Il parcourt le graphe niveau par niveau, en cercles concentriques atour de la source. D'abord ceux de distance 1, puis ceux de distance 2, etc. La file contient des sommets dont il faut encore explorer le voisinage.

```
    int E[][]; // la liste d'adjacence

    int n; // nb de sommets

    int s; // la source
    int d[]; // distance
    boolean visited[]; // marque pour parcours BFS

    void BFS() {
```

```
        d = new int[n];
        Arrays.fill(d,Integer.MAX_VALUE);
        d[s] = 0;

        visited = new boolean[n];
        Queue<Integer> Q = new LinkedList<Integer>();
        Q.add(s);

        while (!Q.isEmpty()) {
            int u = Q.poll();
            if (!visited[u]) {
                visited[u] = true;
                for (int v: E[u]) {
                    d[v] = Math.min(d[v], d[u]+1);
                    Q.add(v);
                }
            }
        }
    }
```

Problèmes Traffic Jam [tju :2272]

13.3 Poids arêtes dans $\{0, 1\}$

Mais imaginons qu'il s'agit de trouver un chemin qui utilise un nombre minimum de portes, dans un labyrinthe composé de vide, de murs infranchissables et de portes. Alors on veut trouver un plus court chemin dans un graphe dont les arêtes sont étiquetés par 0 ou 1, et la longueur d'un chemin est la somme des poids des arêtes.
Ce problème se résoud de nouveau en temps linéaire avec une modification du parcours BFS. Au lieu d'utiliser une file avec les sommets dont il faut encore explorer le voisinage, on utilise une file à deux bouts (double ended queue) Q.
Quand on explore un nouveau voisin v d'un sommet u, avec $w(u, v) = 1$, on ajoute v en fin de file, comme pour un BFS normal. Mais quand $w(u, v) = 0$ on l'ajoute en tête de file. L'invariant est que la file contient des sommets à distance d ou $d + 1$ de l'orgine pour un entier d, et en tête de file se trouvent tous les sommets à distance d. Ce parcours a alors quelque chose d'un parcours BFS (File) et d'un parcours DFS (pile). En effet considérez le graphe par niveau, où un niveau est composé de tous les sommets de même distance de l'origine.
Cet algorithme parcourt alors les sommets d'un même niveau comme un parcours DFS, tout en posant dans la file des sommets du niveau suivant.

Problèmes Nemo [zju :2210]

13.4 DAG — graphe sans cycle dirigé

Se résoud par programation dynamique. Supposons que les sommets soient indicés de 0 à $n - 1$, tel que chaque arc (u, v) satisfait $u < v$. Alors on veut calculer la distance de chaque noeud à partir d'une source unique s. Clairement $d(s) = 0$, et sinon $d(v) = 1 + \min d(u)$ où le minimum est pris sur tous les arcs (u, v) et vaut $+\infty$ s'il n'y a pas d'arc.

1. Faire un tri topologique de $G(V, A)$.
2. Initialiser $d : V \to \mathbb{Z}$ à ∞ partout sauf pour $d[v_0] = 0$.

3. Pour tout $(u, v) \in A$, dans l'ordre topologique sur u, $d[v] > d[u] + w(u, v)$, alors poser $d[v] := d[u] + w(u, v)$.

13.5 Un DAG de composants

Imaginez que le labyrinthe comporte des clés et des portes qui ne s'ouvrent qu'avec certaines clés. Où que le labyrinthe possède une troisième dimension, et que certaines cases sont des ouvertures où on tombe vers un étage inférieur.
Tout ces cas se réduisent à un graphe qui est partitioné en composantes, et il y un ordre sur les composantes, tel que que tout arc entre un sommet de composante i vers un sommet de composante j satisfait $i \leq j$. L'indice de la composante pourrait être l'ensemble des clés ramassées, ou l'étage pour les examples cités.
Le graphe induit sur les composantes est alors un DAG (directed acyclic graph). Les plus courts chemins dans les DAG se calculent par programmation dynamique en temps linéaire. On peut alors utiliser cette propriété pour combiner cette programmation dynamique avec l'algorithme de plus court chemin au sein des composantes.

13.6 Poids non-négatifs

L'algorithme de Dijkstra calcule les plus courts chemin d'une source vers tous les sommets et a besoin que le poids des arcs soit non-négatif. Un implémentation brutale est en $O(n^2)$, et l'utilisation d'une file de priorité ramène à $O(m \log n)$, où n est le nombre de sommets et m le nombre d'arêtes.
Il fonctionne comme suit. Il maintient un ensemble de sommets S qui contient l'ensemble des sommets v, pour les quels on a déjà calculé le plus court chemin de la source à v. Initialement S ne contient que la source elle même. Plus précisément il maintient un arbre de plus courts chemin avec la source comme racine et qui couvre S.
Puis on s'intéresse aux arêtes du bord de S. Plus précisément on considère les arcs (u, v) avec u dans S et v pas dans S. Ces arcs définissement des chemins : un arc (u, v) définit un chemin de la source vers u, suivi de cet arc. Ce chemin a un poids et on définit la priorité de l'arc (u, v) comme l'inverse de ce poids. Maintenant considérons un arc (u, v) qui a la plus grande priorité, donc qui définit un chemin de poids minimal. Appelons P ce chemin.
Il se trouve que ce chemin est un plus court chemin vers v. Pourquoi ? N'importe quel chemin de la source vers v doit quitter à un moment donné l'ensemble S, et par le choix de l'arc (u, v), rien que cette partie est déjà aussi lourde que P. Est-ce que vous voyez où dans cette argumentation intervient l'hypothèse que les poids sont non-négatifs ? On peut donc ajouter l'arc (u, v) à l'arbre des plus courts chemin, et ainsi ajouter v à S et itérer.
La difficulté technique dans l'implémentation de l'algorithme de Dijkstra est le choix d'une bonne structure de données pour la file de priorité qui nous permettera de trouver l'arc le plus prioritaire.

File de priorité avec des arcs

On peut tout simplement maintenir une file de priorité avec tous les arcs (u, v) de la forme $u \in S$ et $v \notin S$. Quand on enleve un arc (u, v) de cette file, pour être strict il faudrait enlever aussi tous les autres arcs de la forme (u', v) à partir du moment où on ajoute v à S. Mais on peut être paresseux, et laisser ces arcs dans la file. Au moment de l'extraction, il faudra juste ignorer les arcs dont la destination est déjà dans S. Pour intier l'algorithme il faudrait mettre dans la file soit tous les arcs quittant la source, soit un arc fictif qui relie la source à la source.
La file de priorité sera alors de type *PriorityQueue⟨Arc⟩* et il faut définir une classe Arc, ou alors *PriorityQueue⟨Arc⟩* et il faut stocker un tableaux de tous les arcs identifiés par des entiers.

```
static int n; // nombre de sommets
```

```
static int d[]; // degree sortant des sommets
static int adj[][]; // adj[u][i] = indice du i-'eme arc sortant de u

static int m; // nombre d'arcs au total
static int arcSrc[], arcDest[];
static int arcWght[];
[...]

static int dist[];
// calcule les plus courts chemin par Dijktra

static long dijkstra(int source, int destination) {
    dist = new int[n];
    Arrays.fill(dist, Integer.MAX_VALUE/2); // -- div 2 = 'eviter d'ebord. lors d'une add.
    dist[source] = 0;
    PriorityQueue<Integer> Q = new PriorityQueue<Integer>(m, new Comparator<
        Integer>() {
            public int compare(Integer I, Integer J) {
                int i = I.intValue(), j=J.intValue();
                int wi = dist[arcSrc[i]] + arcWght[i];
                int wj = dist[arcSrc[j]] + arcWght[j];
                return wi-wj;
            }});
[...]
```

File de priorité avec des sommets

C'est plus simple d'utiliser une file de priorité avec des sommets, ce qui peut être un peu plus simple à implémenter qu'une file avec des arcs. Dans ce cas on maintient un tableau *dist*. Pour les sommets dans S ce tableau contiendra la distance à partir de la source. Pour les sommets v qui ne sont pas dans S, il contiendra le minimum entre $\text{dist}[u] + w(u, v)$ sur tous les arcs (u, v) avec $u \in S$ et $+\infty$ si un tel arc n'existe pas. Il suffit alors de munir la file de priorité avec un comparateur, qui compare les sommets en fonction des valeurs dans ce tableau.
Au moment d'ajouter un sommet v dans S, il faudrait alors mettre à jour le tableau *dist* pour tous les voisins v' de v qui ne sont pas encore dans S. Et là il aura un problème avec la file de priorité, car en changeant ces valeurs on pertube la représentation interne de la file. Une file est représentée par un tas, qui est un arbre binaire avec certaines conditions d'ordre entre père et fils, et un changement de valeur violerait ces conditions. La solution est alors de supprimer ces sommets de la file, changer leur valeur *dist*, puis les reinsérer. La complexité du programme ne se verra pas augmenté, le travail passé par arc est toujours d'ordre $O(\log n)$.
Par contre la classe *PriorityQueue⟨Integer⟩* ne permet pas la suppression d'un élément qui ne soit pas le sommet du tas. On peut alors utiliser comme solution de rechange la classe *TreeSet⟨Integer⟩*, qui permet aussi d'accéder au sommet de poids minimal en temps $O(\log n)$, et qui permet la supression de n'importe quel de ses éléments.

```
static void Dijkstra(int source) {
    dist = new int[n];
    // infini/2 pour eviter debordement lors d'une addition
    Arrays.fill(dist,Integer.MAX_VALUE/2);
    dist[source] = 0;

    boolean deja[] = new boolean[n];

    // cet ordre retire voisin de S a plus courte distance
```

```
        TreeSet<Integer> Q
            = new TreeSet<Integer>(new Comparator<Integer>() {
                    public int compare(Integer u, Integer v) {
                        return dist[u]−dist[v];
                    }
                });

        Q.add(source);

        while (!Q.isEmpty()) {
            int u = Q.pollFirst();
            if (!deja[u]) {
                deja[u] = true; // ajouter u a S
                for (int i: adj[u]) { // mettre a jour d pour les voisins
                    int v = arcDest[i];
                    if (!deja[v]) {
                        int w = arcWght[i];
                        int alt = dist[u]+w;
                        if (alt<dist[v]) { // amelioration
                            Q.remove(v);
                            dist[v] = alt;
                            Q.add(v);
                        }
                    }
                }
            }
        }
    }
```

Nos expériences ont montré que les deux implémentations ont des temps d'exécution très similaires, avec un avantage de 15% pour la file de priorité avec sommets. Le graphe de test était une clique de $n-1$ sommets, chaque arête de poids aléatoire entre 1 et 100, et un sommet supplémentaire relié directement à la source avec un poids 1000. La source était un sommet de la clique. Ainsi on espère maximiser la taille et le nombre d'opérations sur la file de priorité.

Problèmes War [tju :3410], Almost the shortest route [tju :2459]

13.7 Distance euclidienne

Imaginez que le graphe est planaire, dont les sommets sont sous forme de points dans le plan et le poids d'une arête est donnée par la distance euclienne — ou distance de Manhattan par exemple — entre le points. Imaginez qu'on veuille pas calculer la distance d'une source s vers tous les sommets v, mais vers une distination t donnée.
Dans ce cas, on peut rafiner l'agorithme de Dijstra. Au moment de retirer de la file de priorité un sommet u qui minimise d_u, s'il y a plusieurs candidats on va préférer celui qui minimise aussi la distance euclidienne vers t. Cet algorithme porte le nom de A^* et est très efficace.

13.8 Poids arêtes arbitraires

Par l'algorithme de Bellman-Ford en $O(mn)$. On relaxe les distances, c'est-à-dire on teste pour chaque arc (u, v), si l'utilisation de l'arc peut diminuer la distance de s à v, donc on compare la valeur actuelle de d_v avec $d_u + w(u, v)$ et on met à jour d_v. Ceci est fait pour chaque arc, et repété au plus $n-1$ fois, le nombre d'arc maximal d'un plus court chemin.

```
// retourne vrai s'il n'y a pas de cycle negatif
boolean BellmanFord() {
    d = new int[n];
    // infini/2 pour eviter debordement lors d'une addition
    Arrays.fill(d,Integer.MAX_VALUE/2);
    d[s] = 0;

    for (int i=1; i<n; i++)
        for(int u=0; u<n; u++) // pour chaque arc (u,v)
            for (int v: E[u])
                d[v] = Math.min(d[v], d[u]+w[u][v]); // relaxer
    // detecter cycle negatif
    for(int u=0; u<n; u++) // pour chaque arc (u,v)
        for (int v: E[u])
            if (d[v] > d[u]+w[u][v])
                return false; // on pourrait relaxer a l'infini
    return true;
}
```

13.9 Tous les plus courts chemins

Il y quelques semaines nous avons vu l'algorithme par programmation dynamique **Floyd-Warshall**, qui calcule tous les plus courts chemins dans un graphe. Rappel : $M_k[u, v]$ est la longueur du plus court chemin de u à v en n'utilisant que des sommets intermédiaires d'indice $\leqslant k$. La matrice M_k est calculée en temps $O(|V|^2)$ en fonction de M_{k-1}, ce qui donne un algorithme en temps $O(|V|^3)$. Il existe un algorithme de **Johnson**, pour des graphes éparses qui a la complexité $O(|V|^2 \log |V| + |V| \cdot |A|)$. Mais il est long à implémenter, donc on ne le présente pas.

13.10 Plus long chemin dans un DAG

Normalement le problème de trouver le plus long chemin entre deux sommets dans un graphe est NP-dur, mais dans le cas d'un graphe acyclique sans cycle, ce problème est de nouveau de complexité linéaire. On utilise tout simplement la même récurrence que pour le plus court chemin dans un DAG, en remplacant le min par max.

13.11 Plus long chemin dans un arbre

Comme beaucoup de problèmes sur des arbres, on peut appliquer la programmation dynamique, en raisonant sur les sous-arbres.
Par programmation dynamique en temps lineaire. On fixe une racine, ce qui a pour effet d'orienter les arêtes de l'arbre.
Alors pour chaque sommet v soit le sous-arbre avec v comme racine. On note $b[v]$ le plus long chemin dans ce sous-arbre terminant en v et $t[v]$ le plus long chemin dans ce sous-arbre sans restriction.
Cas de base : si v n'a pas de fils $b[v] = t[v] = 0$.
Sinon :

$$b[v] = 1 + \max b[u], \text{ sur les fils } u \text{ de } v$$
$$t[v] = \max\{\max t[u_1], \max b[u_1] + 2 + b[u_2]\}, \text{ sur les fils } u_1, u_2 \text{ de } v$$

Ici la 2ème option est seulement possible si v a 2 fils bien sûr.
Le programme peut se passer de tests sur le nombre de fils, en utilisant -1 comme valeur par default, donc tout s'annulera bien.

Piège S'il y a de l'ordre du million de sommets dans l'arbre, et alors un parcours DFS est hors d'atteinte par limite sur la pile des appels. Il faut alors réaliser un DFS avec une pile explicite.

Problèmes Labyrinth [Tianjin :1056]

13.12 Chemin qui minimise le poids maximum des sommets intermédiaires

Les sommets sont munis d'un poids, et le coût d'un chemin de s à t ici est le poids maximum des sommets intermediaires du chemin.

En $O(n^3)$

Se résoud en temps $O(n^3)$ par Floyd-Warshall. Il suffit de trier les sommets par ordre croissante de poids. L'algorithme calcule une matrice booléenne, dont l'entrée $M_k[i][j]$ est *vrai* s'il existe un chemin de i à j n'utilisant que des sommets intermédiaires entre 1 et $k-1$.

$$M_0[i,j] = \text{vrai s'il } i = j \text{ ou s'il existe une arête } (i,j)$$
$$M_k[i,j] = M_{k-1}[i,j] \vee M_{k-1}[i,k] \wedge M_{k-1}[k,j]$$

Implémentation en pratique on n'a pas besoin de n matrices, car la matrice M_k ne dépend que de M_{k-1}, donc une seule matrice suffira qu'on fera évoluer dans une boucle sur k.

En $O(n \log n + m)$

Par réduction au problème du chemin qui minimise le poids maximum des arêtes. Il suffit de définir le poids d'une arête comme le minimum entre le poids des sommets reliés par l'arête.
Dans un premier temps on trie les sommets en poids croissants en temps $O(n \log n)$. Puis pour chaque sommet u dans cet ordre, on ajoute les arêtes (u, v) dans une structure union-find.

13.13 Chemin qui minimise le poids maximum des arêtes

Ici les arêtes sont munis d'un poids w, et le coût d'un chemin de s à t est le maximum du poids de ses arêtes.
Algorithme en $O(|E|log|V|)$: Trier les arêtes. Partir du graphe vide, puis ajouter les arêtes un par un dans l'ordre croissant de poids, jusqu'à ce que s et t soient dans la même composante connexe.
Une structure pour maintenir les composantes connexes est *union-find*.

Chapitre 14

Couplage maximal

14.1 Couplage maximal biparti

Une approche naturelle pour aborder un problème d'optimisation est de partir d'une solution naïve, et de chercher à l'améliorer jusqu'à ce qu'elle soit optimale. C'est exactement le cas des algorithmes présentés aujourd'hui.
Le problème du couplage maximal est le suivant. Étant donnée un graphe biparti $G(U, V, E)$ on veut trouver un ensemble maximal $M \subseteq E$ tel que dans $G(U, V, M)$ chaque sommet ait au plus un voisin. Si c'est le cas, on appelle un tel sommet *couplé*, sinon c'est un sommet *libre*. Un tel ensemble M est appelé un *couplage*.

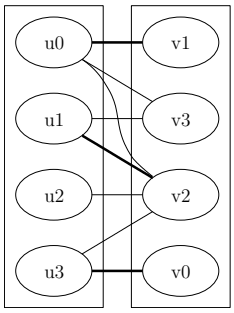


FIGURE 14.1 – En gras, un couplage. Le chemin (u_2, v_2, u_1, v_3) est un chemin augmentant.

Pour un couplage M fixé, un *chemin augmentant* $C \in E^*$ est un chemin qui relie deux sommets libres et utilise de façon alternée des arêtes de M et des arêtes en dehors de M. La différence symétrique $M \oplus C := (M - C) \cup (C - M)$ est de nouveau un couplage, mais de cardinalité augmenté de 1, d'où l'adjectif *augmentant*. Un cycle alternant est un chemin d'un sommet u vers lui même qui utilise de façon alternée des arêtes de M et en dehors de M.
La différence symétrique entre deux couplages est composée de cycles alternants et de chemins augmentants. Si les couplages n'ont pas la même cardinalité, clairement il y au moins un chemin augmentant. (Est-ce que vous voyez l'analogie avec les matroïdes ?) C'est pourquoi l'algorithme suivant est correct.

1. Partir de la solution triviale $M = \emptyset$.
2. Tant qu'il existe un chemin augmentant C, poser $M := M \oplus C$.

Un tel chemin augmentant peut être trouvé simplement par un parcours en profondeur. C'est tout bête, on part d'un sommet fictif, qui est attaché à tous les sommets libres dans U. Comme il y a au plus n itérations de la boucle, la complexité est $O(nm)$ pour $n = \min |U|, |V|$ et $m = |E|$.

Ceci peut être diminué à $O(\sqrt{n}m)$ en trouvant simultanément plusieurs chemins augmentants avec un unique parcours. C'est l'algorithme de HOPCROFT-KARP mais on ne le présente pas ici.

14.1.1 Détails techniques

Dans l'implémentation qui suit, $n_0 = |U|$ et $n_1 = |V|$. Les tableaux m_0 et m_1 contiennent le couplage, et un sommet $u \in U$ libre sera marqué par $m_0[u] =$ LIBRE. On a représenté le graphe par une matrice d'adjacence.

```
class MaxBipartiteMatching {

    // M = matrice d'adjacence
    // n0,n1 = nb sommets a gauche et a droite
    // m0[i] est le sommet a droite auquel i est couple'
    // m0[i]==LIBRE si i n'est pas couple'
    boolean[][] M;
    int n0, n1;
    int[] m0, m1;
    static final int LIBRE=-1;

    // marquage des sommets deja visites pour le dfs
    boolean[] deja;

    MaxBipartiteMatching(int _n0, int _n1) {
        n0 = _n0;
        n1 = _n1;
        M = new boolean[n0][n1];
        m0 = new int[n0];
        m1 = new int[n1];
        deja = new boolean[n0];
        Arrays.fill(m0, LIBRE);
        Arrays.fill(m1, LIBRE);
    }

    // retourne si un chemin augmentant a pu etre trouve,
    // dans ce cas ce chemin est applique au couplage
    // commence a explorer de i qui est a gauche
    boolean dfs(int i) {
        assert (!deja[i]);
        deja[i] = true;
        for (int j=0; j<n1; j++)
            if (M[i][j] && (m1[j]==LIBRE ||
                            !deja[m1[j]] && dfs(m1[j]))) {
                m0[i] = j; // coupler i avec j
                m1[j] = i;
                return true;
            }
        return false;
    }

    // cherche un chemin augmentant a partir de u
    // retourne 1 si succes, 0 sinon
    boolean cheminAugmentant() {
        Arrays.fill(deja, false);
```

```
        for(int i=0; i<n0; i++) {
            if (m0[i]==LIBRE)
                if (dfs(i))
                    return true;
        }
        return false;
    }

    // trouve un couplage maximal et retourne sa cardinalite
    int maxBipartiteMatching() {
        int val=0;
        // partir du couplage trivial vide
        Arrays.fill(m0, LIBRE);
        Arrays.fill(m1, LIBRE);
        // chercher a augmenter tant que possible
        while (cheminAugmentant())
            val ++;
        return val;
    }

    public static void main(String[] args) {
        int i,j, n0, n1;
        Scanner in = new Scanner(System.in);
        n0 = in.nextInt();
        n1 = in.nextInt();
        MaxBipartiteMatching g = new MaxBipartiteMatching(n0, n1);

        // lire les aretes
        while (in.hasNext()) {
            i = in.nextInt();
            j = in.nextInt();
            assert(0<=i && i<n0 && 0<=j && j<n1);
            g.M[i][j] = true;
        }
        System.out.println("max␣matching␣=␣"+g.maxBipartiteMatching());
        for (i=0; i<n0; i++)
            if (g.m0[i] != LIBRE)
                System.out.println("␣"+i+"␣--␣"+g.m0[i]);
    }
}
```

Problèmes Crime Wave [onlinejudge :5584]

14.2 Couplage planaire

Étant donnés n points blancs et n points noirs dans le plan, le but est de trouver un couplage parfait (de cardinalité n) entre les points blancs et noirs, de sorte que si on relie les points des couples par des segments, alors il n'y aura pas de croisement. Soient deux couples $(a_1, b_1), (a_2, b_2)$ dont les segments se croisent. Si on change le couplage en $(a_1, b_2), (a_2, b_1)$, alors la longueur totale des segments diminue (bien que le nombre de croisements puisse augmenter). Ceci montre deux choses : (1) Que le couplage parfait à coût minimal est sans croisement, où le coût est défini comme la longueur des segments. Et (2) qu'un décroissement répété va converger

vers une solution, mais sans garantie sur le temps de convergence, semble-t-il. À essayer en premier, car plus simple à implémenter que la méthode hongroise.

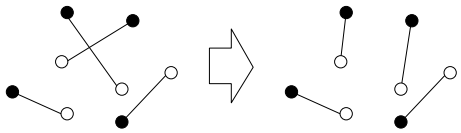


FIGURE 14.2 – Décroiser diminue la longueur totale des segments.

14.2.1 En temps $O(n^3)$

Réduire vers le problème de couplage parfait à coût total minimal dans un graphe biparti, où le poids d'une arête correspond à la longueur du segment entre les points.

14.2.2 En temps $O(n \log n)$

Utilise le théorème du sandwich au jambon. Celui-ci dit que si on dispose de n points noir et n points blancs dans le plan, alors il existe une droite telle que de chaque côté il y ait au plus $n/2$ points noirs et au plus $n/2$ points blancs. Un algorithme de Matousek, Lo, and Steiger de 1994 permet de trouver cette droite en temps $O(n)$. Par contre il est long à implémenter, je déconseille pour les concours.
En quoi cela nous aide ? Avec la promesse qu'il n'y ait pas trois points co-linéaire, on obtient la propriété suivante. Soit n est pair, et la ligne séparatrice ne traverse aucun point, soit n est impair et la ligne séparatrice traverse exactement un point blanc et un point noir. Dans cas il est correct de les coupler. Et dans tous les cas on peut itérer sur les deux sous-instances indépendantes crées par la ligne. Cette décomposition récursive a une profondeur $O(\log_2 n)$, donc l'algorithme final est bien en $O(n \log n)$.

Chapitre 15

Couplage biparti à profit maximal

Maintenant on dispose d'une pondération $w : E \to \mathbb{R} \cup \{\infty\}$ sur les arêtes d'un graphe biparti complet $G(U, V, E)$ avec $|U| = |V|$. Un couplage M est parfait, si tout sommet est couplé dans M. Le problème consiste à trouver un couplage parfait de profit total maximal.

Notez que si $|U| \neq |V|$ on peut toujours ajouter des sommets bidons et si le graphe n'est pas complet, on peut toujours ajouter des arêtes de poids zéro. Si on veut en fait le couplage parfait de poids minimal, il suffit de multiplier les poids par -1.

Kuhn a découvert en 1955, qu'on pouvait augmenter le coût d'un couplage M en appliquant un cycle alterné de poids négatif. C'est similaire à l'approche de la section précédente : Le poids d'un cycle alterné C est défini comme le poids total des arêtes dans $C - M$ moins le poids total des arêtes dans $C \cap M$. Clairement $M \oplus C$ est encore un couplage et son poids est la somme des poids de M et de C. Ce problème a reçu beaucoup d'attention, et a culminé dans ce qui s'appelle l'algorithme hongrois, ou l'algorithme de Kuhn-Munkres.

15.1 Arbre alterné

On note $w(M) := \sum_{(u,v)\in M} w(u, v)$ le poids d'un couplage M. On a déjà vu les cycles et les chemins alternants. Maintenant on définit aussi un *arbre alternant* : c'est un arbre enraciné en un sommet libre, où chaque chemin vers une feuille est alternant.

L'algorithme qu'on va présenter maintenant est un algorithme primal-dual, ça veut dire qu'il augmente une solution partielle en même temps qu'il diminue une solution pour le problème dual. Cette deuxième solution permet à la fois de déterminer comment améliorer la première solution et de détecter quand l'optimum est atteint. Mais voyons concrètement cette approche.

15.2 Problème dual d'étiquetage

Un *étiquetage de sommets* est une fonction $\ell : U \cup V \to \mathbb{Z}$. Il est *valide* si

$$\ell(u) + \ell(v) \geqslant w(u, v) \; \forall u \in U, v \in V. \tag{15.1}$$

Le problème dual au problème de couplage parfait à profit maximal consiste à trouver un étiquetage valide de somme totale minimale. Pensez à l'analogie avec les flots et les coupes, ou mieux avec le problème de plus court chemin et le problème dual de trouver des étiquettes de distances sur les sommets.

15.3 Graphe d'égalité

Pour certaines des arêtes on aura en fait égalité, et le *graphe d'égalité* $G(U, V, E_\ell)$ est le graphe composé justement de ces arêtes, où

$$E_\ell = (u, v) : \ell(u) + \ell(v) = w(u, v).$$

Théorème 6 (Kuhn-Munkres) *Si ℓ est valide et M un couplage parfait dans E_ℓ alors M est un couplage parfait de poids maximal.*

Démonstration Soit M' un couplage parfait arbitraire dans G (pas forcément dans E_ℓ). Comme tout sommet est couvert exactement une fois par M' on a

$$w(M') = \sum_{(u,v)\in M'} w(u, v) \leq \sum_{(u,v)\in M'} \ell(u) + \ell(v) = \sum_x \ell(x),$$

donc $\sum \ell(x)$ est une borne supérieure sur le poids de tout couplage parfait. Maintenant soit M un couplage parfait dans E_ℓ. Alors

$$w(M) = \sum_{(u,v)\in M} w(u, v) = \sum_{(u,v)\in M} \ell(u) + \ell(v) = \sum_x \ell(x).$$

Donc $w(M') \leqslant w(M)$ et M est en effet un couplage parfait de poids maximal. □

Ce théorème transforme un problème d'optimisation avec des poids à un problème purement combinatoire de recherche de couplage parfait. Et ça on sait faire. Notre algorithme aura donc la structure suivante.

1. Partir de n'importe quel étiquetage ℓ valide et un couplage M dans E_ℓ. On choisit simplement
   $$\forall u \in U : \ell(u) = \max_{v\in V} w(u, v) \forall v \in V : \ell(v) = 0.$$
   Clairement un tel étiquetage satisfait
   $$\forall (u, v) \in U \times V : w(u, v) \leq \ell(u) + \ell(v).$$
   Pour le couplage initial, ne nous fatiguons pas et posons $M = \emptyset$.
2. Tant que M n'est pas parfait, trouver un chemin augmentant pour M dans E_ℓ, et augmenter M. Cette recherche peut changer ℓ.

Maintenant on va détailler comment trouver un chemin augmentant. La procédure qui réalise cela va parfois changer l'étiquetage, mais tout en préservant que M est un couplage dans E_ℓ. Quand le processus termine, alors M sera un couplage parfait dans E_ℓ pour un étiquetage valide ℓ. Donc par le théorème Kuhn-Munkres, M sera en fait un couplage parfait de poids maximal dans G. Elle n'est pas belle la vie ?

15.4 Arbre alternant

Un arbre alternant est un arbre avec comme racine un sommet libre $u_0 \in U$, qui a la propriété que sur chaque chemin de la racine vers une feuille les arêtes sont de manière alternée dans M et pas dans M. On note A l'arbre, S les sommets de U couverts par l'arbre et T les sommets de V couverts par l'arbre. Notez que u_0 est l'unique sommet libre dans S, et que les sommets libres dans V sont des feuilles. Un chemin de u_0 vers une feuille libre est un chemin augmentant, et c'est ce qu'on veut trouver avec l'aide de cet arbre.
Pour construire un tel arbre, on commence par choisir un sommet libre $u_0 \in U$. S'il y en a pas, alors M est déjà un couplage parfait. On définit le voisinage d'un sommet $x \in U$ et d'un ensemble de sommets $S \subseteq U$ par

$$\Gamma_\ell(u) := \{v | (u, v) \in E_\ell\}, \Gamma_\ell(S) := \bigcup_{u\in S} \Gamma_\ell(u).$$

1. On initialise $S = \{u_0\}, T = \{\}, A = \{u_0\}$.

2. répéter en boucle

   (a) Si $\Gamma_\ell(S) = T$, améliorer l'étiquetage tel que $\Gamma_\ell(S) \supset T$ strict et que le graphe d'égalité est préservé pour les arêtes entre S et T et entre $\bar{S}$ et $\bar{T}$.

   (b) Maintenant il existe une arête $(u, v) \in E_\ell$ avec $u \in S, v \notin T$. Ajouter (u, v) à A et v à T. Si v est libre alors on a trouvé et chemin augmentant et on termine la construction de l'arbre. Sinon v est couplé dans M à un sommet disons $x \in U$, et on ajoute aussi (x, v) à A et x à S.

Clairement la boucle préserve l'invariant que les arêtes de M vont soit de S à T soit de $\bar{S}$ à $\bar{T}$. Et donc M reste un couplage dans E_ℓ, après chaque modification de ℓ. Il reste à expliquer comment améliorer l'étiquetage.

15.5 Améliorer l'étiquetage

Par *marge d'une inégalité* on entend la différence entre la partie gauche et la partie droite.

Lemme 4 *Soient $S \subseteq U$ et $T = \Gamma_\ell(S)$ avec $T \neq V$. Soit la marge (slack)*

$$\alpha_\ell = \min_{u \in S, v \notin T} \ell(u) + \ell(v) - w(u, v)$$

et

$$\ell'(x) := \begin{cases} \ell(x) - \alpha_\ell & \text{si } x \in S \\ \ell(x) + \alpha_\ell & \text{si } x \in T \\ \ell(x) & \text{sinon .} \end{cases}$$

Alors ℓ' est valide et

i. si $(u, v) \in E_\ell$ et $u \in S, v \in T$ alors $(u, v) \in E_{\ell'}$,

ii. si $(u, v) \in E_\ell$ et $u \notin S, v \notin T$ alors $(u, v) \in E_{\ell'}$.

iii. Il existe une arête $(u, v) \in E_{\ell'}$ avec $u \in S$ et $v \notin T$.

Le fait que ℓ' soit valide, découle du fait que les seules inégalités de (15.1) qui voient leur marge diminuer sont ceux pour $u \in S, v \notin T$, et justement on a choisi α_ℓ assez petit pour ne violer aucune de ces inégalités. Mais α_ℓ a été choisi assez grand, pour qu'une des inégalités devienne juste (tight), ce qui montre le point *iii*.

15.6 La méthode hongroise

On peut maintenant détailler un peu plus l'algorithme. À tout moment, on aura un étiquetage ℓ valide, un couplage M, une racine u_0, un arbre alterné enraciné en u_0, et les ensembles $S \subseteq U, T \subseteq V$ couverts par l'arbre. De plus on aura l'invariant que chaque arête $(u, v) \in M$ satisfait $u \in S, v \in T$ ou $v \notin S, v \notin T$.

1. Partir de n'importe quel étiquetage ℓ valide et un couplage M dans E_ℓ.

2. Si M est parfait, et bien, c'est parfait et on arrête. Sinon choisir un sommet libre $u_0 \in U$. Poser $S = \{u_0\}$ et $T = \emptyset$.

3. Si $\Gamma_\ell(S) = T$, mettre à jour les étiquettes (ce qui force $\Gamma_\ell(S)$ à devenir différent de T) :

$$\alpha_\ell = \min_{u \in S, v \notin T} \ell(u) + \ell(v) - w(u,v)$$

$$\ell'(x) := \begin{cases} \ell(x) - \alpha_\ell & \text{si } x \in S \\ \ell(x) + \alpha_\ell & \text{si } x \in T \\ \ell(x) & \text{sinon .} \end{cases}$$

4. Si $\Gamma_\ell(S) \neq T$, choisir $v \in \Gamma_\ell(S) - T$, et $u \in S$ tel que $\ell(u) + \ell(v) - w(u,v) = 0$.

   (a) Si v est libre alors le chemin de u_0 à u dans l'arbre suivi de (u,v) est un chemin augmentant. On augmente M, et on recommence au point 2.

   (b) Si v est couplé, avec disons x alors, on complète l'arbre alternant avec (u,v) et (v,x)

$$S = S \cup \{x\}, T = T \cup \{v\}.$$

   Recommencer au point 3.

15.7 Un exemple

Considérons l'exemple suivant, avec $n = 3$. Voici la matrice des poids du graphe biparti. Les lignes correspondent aux sommets de U et les colonnes aux sommets de V.

$$\begin{pmatrix} 5 & 1 & 9 \\ 0 & 7 & 8 \\ 2 & 3 & 6 \end{pmatrix}$$

L'algorithme sera déroulé pendant le cours. L'optimum est 18, soit par les arêtes de poids $5, 7, 6$ soit par les arêtes de poids $9, 7, 2$.

15.8 La correction

- On peut toujours trouver un étiquetage valide et un couplage vide initial.
- Si $\Gamma_\ell(S) = T$, on a vu qu'on peut toujours mettre à jour les étiquettes pour créer un nouvel étiquetage valide ℓ'. Le lemme 4 nous assure que toutes les arêtes dans $S \times T$ et $\overline{S} \times \overline{T}$ qui étaient dans E_ℓ seront encore dans $E_{\ell'}$. En particulier, ceci nous garantit que le couplage courant M reste un couplage dans $E_{\ell'}$, ainsi que l'arbre alternant construit jusqu'à maintenant.
- Si $\Gamma_\ell(S) \neq T$, alors on peut par définition toujours augmenter l'arbre alternant en choisissant des sommets $u \in S$ et $v \notin T$ tel que $(u,v) \in E_\ell$. Notez qu'à un moment donné, le v choisit doit être libre, et dans ce cas on augmente M.

15.9 La complexité

Dans chaque phase de l'algorithme, $|M|$ augmente de 1 donc il y a au plus $|U|$ phases. Et combien dure chacune des phases ?
En fait on va maintenir un tableau marge avec pour tout $v \notin T$,

$$\text{marge}_v = \min_{u \in S} \ell(u) + \ell(v) - w(u,v).$$

- Initialiser toutes les marges en début de phase coûte $O(|V|)$.

- À l'étape 4, on doit mettre à jour la marge de tous les sommets quand un sommet entre en S. Ceci prend un temps $O(|V|)$. Comme seulement $|V|$ sommets peuvent bouger de $\overline{S}$ à S, ceci coûte $O(|V|^2)$ par phase.
- Dans l'étape 3, $\alpha_\ell = \min \text{marge}_v$ et peut donc être calculé en temps $O(|V|)$ à partir des marges. Ceci est fait au plus $|V|$ fois par phase (voyez-vous pourquoi ?), et donc coûte au total aussi $O(|V|^2)$ par phase. Après avoir calculé α_ℓ, on doit mettre à jour toutes les marges. On peut faire cela en temps $O(|V|)$ en posant

  $$\forall v \notin T : \text{marge}_v = \text{marge}_v - \alpha_\ell.$$

  Comme ceci n'est fait que $O(|V|)$ fois, le temps total par phase est $O(|V|^2)$.

Il y a $O(|V|)$ phases qui durent chacune $O(|V|^2)$ la complexité totale est donc de $O(|V|^3)$.

```
class KuhnMunkres {

    // ---------------------------------- le graphe

    // taille du graphe
    int n;

    // poids des aretes
    double[][] w;

    // ---------------------------------- le couplage

    //couplage : m0[i] = sommet j dans V auquel i est couple, m1=inverse
    int[] m0, m1;
    final static int LIBRE=-1;

    // cardinalite du couplage
    int m;

    // l'arbre alternant
    // racine est un sommet libre de S
    int racine;
    // path : T -> V definit avec m0 l'arbre alternant
    // path[v] = predecesseur de v dans l'arbre
    int[] path;

    // ---------------------------------- l'etiquetage, etc.

    //etiquetage
    double[] l0, l1;

    //ensembles S,T
    boolean[] S, T;

    // slack en anglais,
    // pour tout v pas dans T :
    // margeVal[v] = min l0[u]+l1[v]-w[u][v] sur u dans S
    // et marge0[v] est u qui minimise
    double[] margeVal;
    int[] marge0;
```

```
KuhnMunkres(int _n) {
    n = _n;
    w = new double[n][n];
    m0 = new int[n];
    m1 = new int[n];
    path = new int[n];
    l0 = new double[n];
    l1 = new double[n];
    S = new boolean[n];
    T = new boolean[n];
    margeVal = new double[n];
    marge0 = new int[n];
}

// ---------------------------------- parties de l'algorithme

// chercher dans U un sommet non couple'
int sommetLibre() {
    for(int u=0; u<n; u++)
        if (m0[u]==LIBRE)
            return u;
    throw new Error("recherche_de_sommet_libre_sur_couplage_parfait");
}

// creer un etiquetage trivial initial
void initEtiquettes() {
    double sup;
    // etiquetage initial
    for (int u=0; u<n; u++) {
        sup = w[u][0];
        for (int v=1; v<n; v++)
            if (sup<w[u][v])
                sup = w[u][v];
        l0[u] = sup;
    }
    Arrays.fill(l1, 0.0);
    // et couplage vide
    Arrays.fill(m0, LIBRE);
    Arrays.fill(m1, LIBRE);
    m = 0;
}

// chercher a augmenter le couplage
void augmenterCouplage() {
    int y0=0, y1, next;
    double a=0.0, marge_uv;
    do {
        // chercher (y0,y1) dans S*!T avec la plus petite marge
        y1 = LIBRE;
        for (int v=0; v<n; v++)
            if (! T[v]) {
                if (y1==LIBRE || a > margeVal[v]) {
                    y1 = v;
```

```
                    y0 = marge0[v];
                    a = margeVal[v];
                }
            }

        // si a>0, alors N_l(S)=T et
        // on peut ameliorer l par a = marge[y]
        if (a>0) {
            for (int u=0; u<n; u++)
                if (S[u])
                    l0[u] -= a;
            for (int v=0; v<n; v++)
                if (T[v])
                    l1[v] += a;
                else
                    margeVal[v] -= a;
            a = 0;
        }

        // desormais a==0 et N_l(S) =/= T, et y1 est de N_l(S)-T.
        // on augmente l'arbre
        if (m1[y1]!=LIBRE) {
            int u = m1[y1];
            // on ajoute un nouveau sommet dans S, il faut mettre a jour la marge
            S[u] = true;
            for (int v=0; v<n; v++)
                if (! T[v]) {
                    marge_uv = l0[u] + l1[v] - w[u][v];
                    if (margeVal[v]> marge_uv) {
                        margeVal[v] = marge_uv;
                        marge0[v] = u;
                    }
                }
        }
        T[y1] = true;
        path[y1] = y0;
    } while (m1[y1]!=LIBRE);

    // y1 est libre, on peut augmenter le couplage
    // remonter le chemin de y1 vers la racine
    do {
        y0 = path[y1];
        next = m0[y0];
        // changer le couplage
        m1[y1] = y0;
        m0[y0] = y1;
        // iterer plus loin sur le chemin vers la racine
        y1 = next;
    } while (y0!=racine);
    m++;
}

void solve() {
    initEtiquettes();
```

```
        // chercher a rendre le coupage parfait
        while (m<n) {
            // chercher sommet libre u et poser S={u}, T={}
            Arrays.fill(S, false);
            Arrays.fill(T, false);
            racine = sommetLibre();
            S[racine] = true;
            // intialiser la marge
            for(int v=0; v<n; v++) {
                margeVal[v] = l0[racine]+l1[v]-w[racine][v];
                marge0[v] = racine;
            }
            augmenterCouplage();
        }
    }

    // ------------------------------------ programme principal

    public static void main(String[] args) {
        int u,v, n;
        double wuv;
        Scanner in = new Scanner(System.in);
        in.useLocale(Locale.US);

        n = in.nextInt();
        KuhnMunkres M = new KuhnMunkres(n);

        while (in.hasNext()) {
            u = in.nextInt();
            v = in.nextInt();
            wuv = in.nextDouble();
            M.w[u][v] = wuv;
        }
        M.solve();

        // afficher solution
        double total = 0.0;
        for (u=0; u<n; u++) {
            System.out.println(""+u+" -- "+M.m0[u]
                             +" ["+M.w[u][M.m0[u]]+"]");
            total += M.w[u][M.m0[u]];
        }
        System.out.println("\ttotal weight="+ total);
    }
}
```

Chapitre 16

Flot maximal

Un problème de flot est donné par un graphe $G(V, A)$ une source $s \in V$ et un puits $t \in V$, ainsi qu'une capacité $c : A \to \mathbb{N}$. Plus généralement les capacités peuvent être des rationnels non-négatifs. On suppose que pour chaque arc il y a les deux arcs (u, v) et (v, u) dans le graphe. Un flot est une fonction $f : A \to \mathbb{Z}$, qui satisfait

$$\forall (u, v) \in A : f(u, v) = -f(v, u). \tag{16.1}$$

et qui respecte les capacités

$$\forall e \in A : f(e) \leqslant c(e), \tag{16.2}$$

et qui respecte la conservation des flots en tout sommet, hormis la source et le puits,

$$\forall v \in V - s, t : \sum_{u:(u,v)\in E} f(u, v) = 0. \tag{16.3}$$

La valeur du flot est la quantité qui sort de la source, $\sum_v f(s, v)$. Le but est de trouver un flot maximal.

16.1 s-t-coupe minimale

Une s-t-coupe est un ensemble $S \in V$, qui déconnecte source et puits, $s \in S$ et $t \notin S$. La valeur de la coupe est la capacité totale des arcs entre S et $V - S$, donc $c(S) := \sum$c(e) où la somme est prise sur tout $e \in A \cap S \times \overline{S}$. Parfois on définit la coupe aussi par l'ensemble de ses arcs du bord, ce qui est équivalent.
Clairement tout flot doit traverser la coupe, et la valeur de tout coupe est donc une borne supérieure sur la valeur de tout flot. On dit que le problème de coupe est le dual du problème de flot. Cela vous rappelle-t-il quelque chose ? Mais il y a plus fort :

Théorème 7 ***(max-flow-min-cut)*** *La valeur du flot maximal est égal à la valeur de la coupe minimale.*

Démonstration La preuve est la suite de plusieurs observations simples.

1. Pour un flot f la quantité de flot qui sort d'une coupe S, à savoir $f(S) := \sum_{u\in S, v\notin S} f(u, v)$ est la même pour tout S. Tout simplement car pour tout $w \notin S, w \neq t$ on a par (16.3) puis

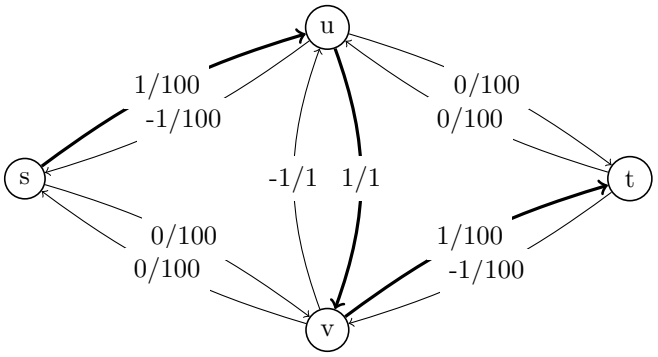


FIGURE 16.1 – Sur cette instance de flot maximum, l'algorithme de Ford et Fulkerson peut itérer un nombre de fois qui est linéaire dans la capacité maximale des arcs. Après avoir augmenté le flot de 1 le long du chemin $s-u-v-t$, il pourra augmenter le flot de 2 le long du chemin $s-v-u-t$, puis de nouveau de 2 le long du chemin $s-u-v-t$ et ainsi de suite.

par (16.1)

$$\begin{aligned} f(S) &= \sum_{u\in S, v\notin S} f(u,v) = \sum_{u\in S, v\notin S} f(u,v) + \sum_{v} f(w,v) \\ &= \sum_{u\in S, v\notin S, v\neq w} f(u,v) + \sum_{u\in S} f(u,w) + \sum_{v\in S} f(w,v) + \sum_{v\notin S} f(w,v) \\ &= \sum_{u\in S\cup w, v\notin S\cup w} f(u,v) = f(S\cup w). \end{aligned}$$

2. Par (16.2) on a $f(S) \leqslant c(S)$, c'est-à-dire que le flot qui sort de S n'est jamais plus que la valeur de S. Ceci prouve déjà une moitié du théorème, à savoir que la valeur du flot maximal est au plus la valeur de la coupe minimale.

3. Maintenant si pour un flot f donné, on a $f(S) < c(S)$ pour toute coupe S, alors il existe un chemin augmentant. Pour cela tout simplement posons $S = s$ et $P = \emptyset$. Puisque $f(S) < c(S)$ il existe une arrête (u, v) avec $u \in S, v \notin S, f(u, v) < c(u, v)$. Ajoutons (u, v) à P. Si $v = t$, alors P est un chemin augmentant, et sinon on ajoute v à S et on recommence.

□

16.2 Graphe résiduel

Pour un flot donné, soit G_f le graphe résiduel, qui est définit comme cela : On fixe f. Maintenant certains arcs sont saturés, certains ont encore un restant de capacité. Le graphe résiduel est le graphe avec ces restants de capacité. Donc G_f a les mêmes sommets et arcs, mais la capacité résiduelle $r(u, v) = c(u, v) - f(u, v)$. Notez que dans le graphe résiduel certains arcs ne deviennent plus franchissable, si la capacité est atteinte $f(u, v) = c(u, v)$, mais de nouveaux arcs franchissables vont apparaître, car par exemple ici $r(v, u) = -f(v, u) = c(u, v)$. Traverser ensuite cet arc, traduit le fait qu'on veuille diminuer le flot $f(u, v)$ pour le réorienter ailleurs.

16.3 Chemin augmentant

Un chemin augmentant est un chemin de s à t dans G_f qui n'utilise que des arcs de capacité résiduelle non-nulle. Clairement s'il existe un tel chemin, alors le flot n'est pas maximal et peut

être augmenté le long du flot par la capacité résiduelle minimale des arcs du chemin. Ceci donne l'algorithme suivant.

Ford-Fulkerson

1. Commencer avec le flot vide $f = 0$.
2. Tant qu'il existe des chemins augmentants (qu'on trouve par DFS), améliorer f.

Le problème avec cet algorithme est qu'il est seulement pseudo-polynomial. C'est-à-dire si les capacités sont entières, la seule garantie qu'on a, est que le flot augmente au moins de 1 à chaque itération, et la complexité dépend alors de la capacité maximale du graphe donné.
Ce qui est étonnant c'est que si on applique d'abord les chemins augmentants les plus courts, alors le même algorithme aura une complexité indépendante de la capacité maximale. L'idée est que la longueur du plus court chemin augmentant augmente strictement tous les $|E|$ itérations au plus. Ceci est du aux observations suivantes.

- Soit L_f le graphe par niveau, où s est au niveau 0, tous les sommets atteignables de s par un arc non saturé forment le niveau 1 et ainsi de suite.
- Un plus court chemin de s à t est forcément un chemin dans ce graphe. Si on augmente le flot le long de ce chemin un des arcs devient saturé.
- Une augmentation le long d'un chemin peut rendre non saturé des arcs, mais seulement des arcs vers des niveaux inférieurs. Donc la distance de s à v dans L_f ne peut pas diminuer, et par symétrie la distance de v à t aussi ne peut pas diminuer.
- Au bout d'au plus $|E| + 1$ itérations un arc (u, v) et sont inverse (v, u) ont été saturés. Ceci prouve que la distance de s à t a strictement augmenté.
- Comme il n'y a que n niveaux, il n'y a que $|V| \cdot |E|$ itérations au total.
- La recherche d'un plus court chemin se fait par BFS en temps $O(|V|)$. La complexité totale est $O(|V| \cdot |E|^2)$.

Il existe des algorithmes plus performants, mais plus long à implémenter. Par exemple l'algorithme des préflots-pousse (preflow-push), est un algorithme primal-dual et tourne en $O(|V|^2\sqrt{|E|})$.

Edmond-Karp

1. Commencer avec le flot vide $f = 0$.
2. Tant qu'il existe des chemins augmentants, améliorer f, avec un plus court chemin augmentant (qu'on trouve par BFS).

16.4 L'algorithme de Dinic

L'idée de l'algorithme de Dinic, qui a été trouvé en même temps que le précédent est la suivante. Plutôt que de trouver un ensemble de chemins augmentant un par un, jusqu'à ce que la distance entre s, et t augmente, on trouve un tel flot avec un seul parcours. Ceci ramène la complexité à $O(|V|^2|E|)$.
Imaginez une fonction DINIC(U,VAL) qui tente de faire passer un flot de u à t dans le graphe par niveaux. La restriction est que ce flot ne dépasse pas VAL. La fonction retourne alors la valeur de ce flot, qui pourra être augmenté le long du chemin de s à u, par les appels successifs qui ont mené à u. Concrètement pour *pousser* un flot de valeur val à t, on parcourt tous les voisins v dans le graphe par niveau, et récursivement on tente de passer un flot de v à t. La somme est alors ce qu'on a pu *pousser* de u à t.

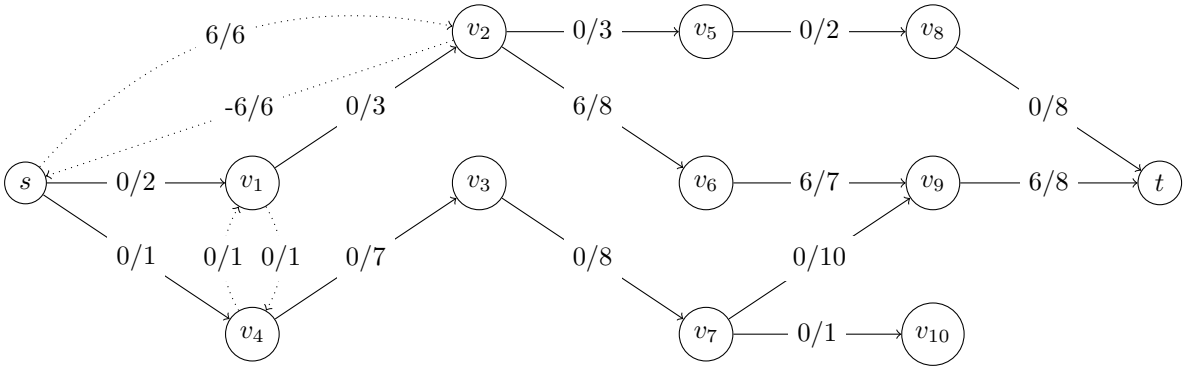


FIGURE 16.2 – Un graphe par niveau, les arcs sont étiquetés par la valeur du flot suivie de la capacité. Lors d'une recherche d'un plus court chemin on ignore les arcs saturés et les arcs qui ne mènent pas vers le niveau suivant. À titre d'exemple seulement certains de ces arcs sont montrés.

Cette fonction DINIC(U,VAL) détecte alors si plus aucun flot ne passe de u à t, donc si t devient inaccessible à partir de u. Dans ce cas elle enleve u du graphe par niveau, tout simplement en lui indiquant un niveau fictif -1. Ainsi les appels suivants ne tenteront plus de passer un flot à travers u. Clairement au bout de $O(n)$ itérations, s et t sont déconnectés, et on recalcule un nouveau graphe par niveaux.
Même si vous utilisez une liste d'adjacence pour le graphe, il est pratique d'utiliser des matrices pour la capacité résiduelle et le flot. On fera bien attention à ce que chaque opération préserve la symétrie $f(u,v) = -f(v,u)$.

```
static int n; // nb sommets
static int C[][]; // capacite initiale
static int F[][]; // flot
static int adj[][]; // adj list
static int l[]; // niveau (l=level)

// initialiser les tableaux
static void init(int _n) {
    n = _n;
    C = new int[n][n];
    F = new int[n][n];
    l = new int[n];
}


// calcule et retourne flot maximal (qui sera dans F)
static int dinic(int s, int t) {
    Queue<Integer> q = new LinkedList<Integer>();
    int total = 0;
    while (true) {
        q.offer(s);
        Arrays.fill(l,-1); // construire graphe des niveaux par BFS
        l[s]=0;
        while (!q.isEmpty()) {
            int u = q.remove();
            for (int v: adj[u]) {
                if (l[v]==-1 && C[u][v] > F[u][v]) {
```

```
                    l[v] = l[u]+1;
                    q.offer(v);
                }
            }
        }
        if (l[t]==−1) // plus de chemin s−t dans ce graphe
            return total;
        total += dinic(s,t, Integer.MAX_VALUE);
    }
}

// trouver un flot bloquant et retourne sa valeur
// suppose que dans p il y a le chemin de s a u
static int dinic(int u, int t, int val) {
    if (val<=0)
        return 0;
    if (u==t) // −− augment
        return val;
    int a=0, av; // −− advance
    for (int v: adj[u])
        if (l[v]==l[u]+1 && C[u][v] > F[u][v]) {
            av = dinic(v, t, Math.min(val−a, C[u][v]−F[u][v]));
            F[u][v] += av;
            F[v][u] −= av;
            a += av;
        }
    if (a==0) // −− retreat
        l[u] = −1;
    return a;
}
```

Chapitre 17

Géométrie

17.1 La bibliothèque java.awt.geom

Un grand avantage de Java par rapport à C++ est l'existence de la bibliothèque java.awt.geom. On y définit des points, des segments, des cercles et des rectangles. Utilisez Point pour des points à coordonnées entières et Point2D.Double pour des coordonnées flottantes. Pour les autres classes il n'y a que des variantes flottantes : Line2D.Double, Ellipse2D.Double (plus général qu'un cercle), Rectangle2D.Double. Ces objets ont toutes sortes de méthodes pour mesurer des distances, détecter les intersections, etc.
Une opération qui revient souvent dans les algorithmes de géométrie, est la suivante. Étant donnés 3 points A, B, C est-ce que C est à droite de la droite $\overline{AB}$ par rapport à l'orientation $A \to B$, sur la droite, ou à gauche ? En Java cela se note comme new Line2D.Double(A,B).relativeCCW(C), et retourne $1, 0$ ou -1 suivant la position relative de C.
Le même test peut aussi être réalisé par la comparaison suivante des tangeants.

```
// does the walk i->j->k make a strict left turn?
static boolean leftturn(int i, int j, int k) {
    return (x[i]-x[k])*(y[j]-y[k]) - (y[i]-y[k])*(x[j]-x[k]) > 1e-6;
}
```

17.2 Nombre de points entiers dans un polygone

Entrée Un polygone défini par les points $(x_0, y_)), \ldots, (x_{n-1}, y_{n-1}) \in \mathbb{Q}^2$.

Sortie Le nombre de points $(x, y) \in \mathbb{N}^2$ contenu dans le polygone.

Complexité linéaire

Algorithme En trois étapes.

- calculer A, la surface du polygone
- calculer b, le nombre de points entier sur le contour du polygone
- utiliser le théorème de Pick pour calculer la solution, donc $A + 1 - b/2$

La surface se calcule par la formule

$$A = \sum_{i=0}^{n-1} (x_i y_{i+1} - x_{i+1} y_i).$$

Le nombre de points entiers du contour, se calcule indépendement pour chaque segment $(x_i, y_i) - (x_{i+1}, y_{i+1})$. En effet pour un segment de la forme $(0,0) - (x,y)$ — on a translaté vers l'origine — le nombre de points entiers traversés par le segment, excepté l'origine est $x == 0?y : (y == 0?x : pgcd(x,y))$.

17.3 Union de rectangles

On vous donne n rectangles rectilinéaires et il faut calculer la surface de l'union.
Chaque rectangle est décrit par deux points opposés (x_1, y_1) et (x_2, y_2) avec $x_1 \leq x_2, y_1 \leq y_2$.
Prendre l'ensemble de tous les coordonnées x_1, x_2, et les trier. Idem pour les coordonnées y_1, y_2.
Ces ensembles forment une grille, d'à peu près $2n \times 2n$ cellules.

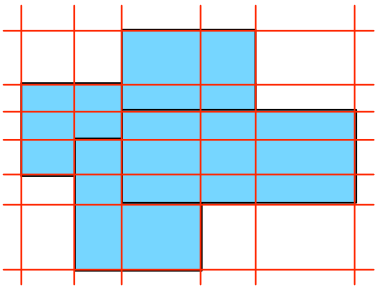


FIGURE 17.1 – Union de rectangles

Maintenir un tableau de booléens, qui indique pour chaque cellule si elle est couverte ou pas.
Chaque rectangle génère $2n \cdot 2n$ mises à jour au plus.
La complexité est en $O(n^3)$

17.4 Combien rectangles peut-on former

On vous donne n points. Parfois 4 points sont les coins d'un rectangles. Calculez combien de rectangles on peut former avec ces n points ?
Construire un dictionnaire qui associe à (c, d) tous les points (a, b) tel que d est la distance entre a et b et c est le point au milieu entre a et b. Alors tous les couples de points qui ont la même clé forment un rectangle. Et il y a au plus n couples avec la même clé.

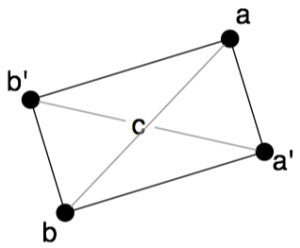


FIGURE 17.2 – propriété comune aux paires de coins opposés

Attention : arrondir c et d à ϵ près pour éviter les erreurs d'arrondi.
Complexité en $O(n^2)$.

Problèmes Finding Rectangles [poj :1314], Fortune at El Dorado [poj :2899]

17.5 Balayage

Une technique importante en géométrie algorithmique est le balayage. Nous allons en voir plusieurs applications.

17.6 Plus grand rectangle sous un histogramme

On nous donne un histogramme sous forme d'un tableau d'entiers non-négatifs $x_0, \ldots, x_{n-1}$. Le but est de placer un rectangle sous ce histogramme qui maximise sa surface. C'est-à-dire qu'il faut trouver un intervale $[a, b]$ qui maximise $l \times h$, où la largeur $l = b - a + 1$ et la hauteur $h = \min a \leq i \leq b x_i$.
L'algorithme est en temps linéaire et par balayage. Pour chaque préfixe du tableau on maintient une collection de rectangles dont on a pas encore déterminé le côté droit.
Donc ces rectangles sont déterminé par un couple d'entiers (a, h), où a est le côté gauche et h la hauteur. On stocke ces entiers sur une pile ordonnée par h.

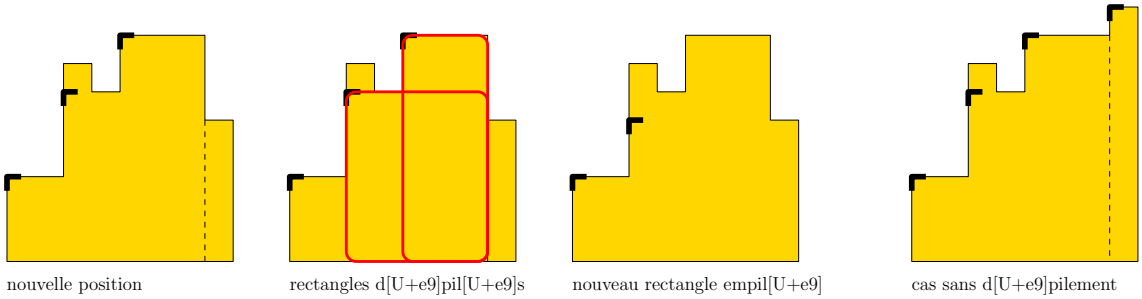


FIGURE 17.3 – Traitement effectué lors du balayage pour le problème du plus grand rectangle sous un histogramme

Maintenant pour chaque valeur x_i, il est possible qu'on a trouvé maintenant le côté droit de certains rectangles. Notamment c'est le cas de tous les rectangles codés (a, h) sur pile avec $h > x_i$. La largeur d'un tel rectangle est $i - a$. Mais cette valeur peut aussi créer un nouveau couple (a', h') avec $h' = x_i$ et a' est soit la valeur a du dernier rectangle qui a été dépilé., ou $a' = i$, s'il n'y a pas eu de dépilement.

```
// suppose que le dernier element de x est 0 pour vider la pile
static int largestRect(int x[]) {
    int n = x.length;
    int a[] = new int[n];
    int h[] = new int[n];
    int s=0; // hauteur de la pile
    int best = 0;

    for (int i=0; i<n; i++) { // balayage
        a[s] = i; // initialiser debut rectangle
        while (s>0 && h[s-1] > x[i]) {
            s--; // depiler
            int area = (i-a[s]) * h[s]; // ferme un rectangle
            if (area>best) best=area;
        }
        h[s] = x[i]; // empiler nouveau couple
        s++;
    }
    return best;
```

```
}
```

Problèmes Tianjin :1800

17.7 Plus grand rectangle monochromatique dans une image

On reçoit en entrée une image binaire et il faut trouver le plus grand rectangle dans cette image qui ne contient que des 1. Ce problème se réduit au problème de trouver un plus grand rectangle sous un histogramme : pour chaque ligne i, maintenez dans une variable $x[j]$ le nombre de 1 au dessous de la case (i, j). La mise à jour est facile, si le pixel en (i, j) est 1, alors $x[j] = x[j] + 1$, sinon $x[j] = 0$. Ces variables décrivent un histogramme. Donc on peut résoudre ce problème en temps linéaire.

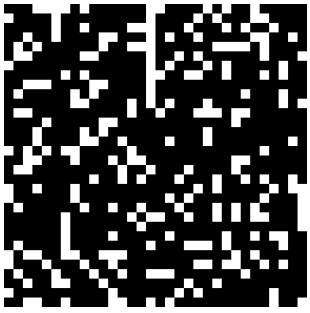


FIGURE 17.4 – Étant donnée une grille $n \times m$ de 0 (blanc) et de 1 (noir), savez-vous trouver en temps linéaire le plus grand rectangle composé seulement de 1 ?

Problèmes Largest Submatrix of All 1's [poj :3494]

17.8 Les points les plus proches

Étant donnés n points dans le plan, on veut trouver la paire la plus proche. L'algorithme naïf prendrait un temps $O(n^2)$. Il existe plusieurs algorithmes en temps $O(n \log n)$. On va en voir un qui utilise le balayage.
Supposons que les points soient triés par leurs coordonnées x, et qu'on les parcourt de gauche à droite. Supposons qu'on considère le point k, et que la plus courte distance trouvée jusqu'à maintenant est h, voir figure 17.5. On va maintenir une structure de donnée avec tous les points parmi $1, \ldots, k-1$ dont la coordonnée x est au moins $x_k - h$. Les points de cette structure sont triés par leur coordonnée y. Par recherche dichotomique on trouve tous les points i tel que $y_k - h < y_i < y_k + h$. Puis parmi eux, on cherche si la plus courte distance h peut être améliorée. L'observation importante est que cette zone ne contient que 6 points, car ils sont à distance au moins h les uns des autres et le rectangle ne fait que $h \times 2h$. Il en suit que le traitement d'un point ne prend que $O(\log n)$ temps, donc la complexité totale est $O(n \log n)$.
Un détail sur l'implémentation. On stocke les points dans un tableau P de taille n, dont les éléments sont de type Point2D. On maintient une structure B, pour stocker les points de la zone grise claire. En général on pré-calcule deux tableau X,Y tel que X[i] est l'indice du point avec la i-ème plus petite coordonnée x. Ne pas trier directement P a l'avantage de pouvoir afficher les indices des points, tel qu'ils étaient donnés en entrée. On maintient l'indice dans X du point le plus à gauche dans la zone claire. Comme ça on peut facilement déterminer les points à enlever de B quand on traite un nouveau point.

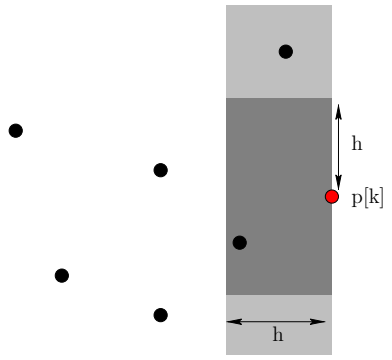


FIGURE 17.5 – Balayage : on maintient une structure avec les points de la zone gris clair et on cherche un plus proche voisin dans la zone gris foncé.

Pour représenter B on a besoin d'un arbre binaire de recherche, augmenté d'un chaînage entre les éléments. En fait on veut pouvoir insérer un segment s et accéder au prédécesseur de s dans B pour tester leur intersection. En C++ on pourrait utiliser la structure set de la STL, qui dispose un itérateur sur les éléments. La méthode find retourne un itérateur sur un élément de B, et permet ensuite d'atteindre le prédécesseur. En Java on pourrait utiliser la structure TreeSet. La méthode SUBSET retourne tous les éléments compris entre deux bornes données.

```
class ClosestPoints {
    public static void main(String args[]) {
        Scanner in = new Scanner(System.in);
        in.useLocale(Locale.US);
        // -- lire tous les points
        int n = in.nextInt();
        assert(n>=2);
        Point2D.Double tab[] = new Point2D.Double[n];
        for (int i=0; i<n; i++)
            tab[i] = new Point2D.Double(in.nextDouble(), in.nextDouble());

        long start = System.currentTimeMillis();
        // -- trier par X
        Arrays.sort(tab, new CompareXY());
        // -- dist=meilleure distance vue jusqu'alors
        double dist = tab[0].distance(tab[1]);
        Point2D.Double sol0 = tab[0], sol1 = tab[1]; // points realisant la distance

        // -- B=bande verticale triee par Y
        TreeSet<Point2D.Double> B
            = new TreeSet<Point2D.Double>(new CompareYX());
        // pour les classes des comparateurs voir chapitre 1

        // -- balayer de gauche a droite
        for (Point2D.Double p: tab) {
            // -- avancer le balai 'a p
            // -- comparer p avec les points de B au dessus et en dessous
            Point2D.Double low = new Point2D.Double(p.getX(), p.getY()-dist);
            Point2D.Double up = new Point2D.Double(p.getX(), p.getY()+dist);
            for (Point2D.Double q: B.subSet(low, up)) {
                if (q.getY() < p.getY()-dist)
                    B.remove(q); // -- point en dehors de la bande
                else {
                    double d = p.distance(q);
                    if (d<dist) { // -- on a trouve mieux
                        dist = d;
                        sol0 = q;
                        sol1 = p;
                    }
                }
            }
            // -- mettre a jour la bande
            B.add(p);
        }
        long end = System.currentTimeMillis();
        System.out.println("closest pair "+sol0+","+sol1+
                           " at distance "+dist);
        System.out.println("computation time (w/o reading) "+(end-start)+"ms");
    }
}
```

Problèmes Quoit Design [Zhejiang :2107]

17.8.1 Une approche différente en temps espéré linéaire

Voici un tout autre algorithme randomisé dont la complexité en moyenne est linéaire. Par contre nos expériences ont montré qu'il était un peu plus lent que l'algorithme précédent.
L'idée est qu'on parcourt les points un par un. On définit une grille de côté $(d/2) \times (d/2)$, et on place un par un les points dans cette grille. L'invariant à maintenir est qu'à tout moment, la plus petite distance entre les points déjà placés dans la grille est au moins d. Donc il y aura au plus un point par cellule, ce qui facilite le choix de la structure de données. Pour chaque nouveau point p on va comparer la distance entre p et les autres points q déjà placés, mais il suffit de considérer les points q dans les $5 \cdot 5$ cases autour de celle de p. Donc ce test coûte un temps constant. Si on trouve une distance d' inférieure à d, alors on pose $d = d'$ et on recommence tout.
Pour la complexité en moyenne l'argument clé est que si les entrées sont permutées uniformément au hasard, alors au moment du traitement du i-ème point $(3 \leq i \leq n)$, on améliore la distance d avec probabilité $1/(i-1)$. Donc au total la complexité espéré est de l'ordre de $\sum_{i=1}^{n} i/(i-1)$, donc linéaire en n.

```
class Cell {
    int a,b;
    Cell(int _a, int _b) {a=_a; b=_b;}
    public int hashCode() {
        return a*1001 + b;
    }
    public boolean equals(Object obj) {
        Cell c = (Cell)obj;
        return c.a==a && c.b==b;
    }
}

class Main {
    static int n;
    static double x[], y[];

    [...]

    static double dist(int i, int j) { // distance euclidienne
        return Math.hypot(x[i]-x[j], y[i]-y[j]);
    }

    static HashMap<Cell,Integer> H;

    static double d;

    static boolean improve() {
        H = new HashMap<Cell,Integer>();
        for (int i=0; i<n; i++) {
            // -- (a,b) coordonnees de la cellule du i-eme point
            int a = (int)Math.floor(x[i]*2/d);
            int b = (int)Math.floor(y[i]*2/d);
            // -- chercher dans les 25 cases autour
            for (int a2=a-2; a2<=a+2; a2++)
                for (int b2=b-2; b2<=b+2; b2++) {
                    Cell k2 = new Cell(a2,b2);
                    if (H.containsKey(k2)) {
                        int j = H.get(k2);
                        double d2 = dist(i,j);
```

```
                        if (d2<d) {
                            d = d2;
                            return true;
                        }
                    }
                }
            Cell k = new Cell(a,b);
            H.put(k, i);
        }
        return false;
    }

    static double solve() {
        d = dist(0,1);
        while (d>0 && improve()) {
            /* do nothing */
        }
        return d;
    }
```

17.9 Intersection de segments

Étant données n segments droits il s'agit de trouver tous les points d'intersection. L'idée est qu'on balaye une droite verticale de gauche à droite et on maintient une structure de donnée avec tous les segments qui l'intersectent. Les points d'intersection ne sont pas intéressants. Ce qui importe est l'ordre de ces points selon l'axe y. Soit une liste $L = s_1, \ldots, s_k$, qui donne l'ordre de ces segments. On maintient une file de priorité avec des évènements qui vont arriver dans le futur. Chaque évènement est associé à un point, et c'est celui de la coordonnée x la plus petite qui est le prochain extrait de la file. Il y a trois types d'évènements.

1. insertion d'un segment dans la liste, associé au point le plus à gauche du segment.
2. suppression d'un segment de la liste, associé au point le plus à droite du segment.
3. changement d'ordre de deux segments adjacents dans la liste, associé à leur point d'intersection.

La clef de cet algorithme est qu'il n'est pas nécessaire de considérer les intersections de deux segments qui ne sont pas adjacents dans la liste : Avant que s_1 intersecte s_3, p_s va intersecter un des deux segments.

1. Si on insère s avant s', alors il faut tester l'intersection entre s et s' et dans ce cas ajouter un évènement de changement d'ordre.
2. Si on supprime s entre s' et s'', alors il faut tester l'intersection entre s' et s''.
3. Si on change d'ordre s et s', alors il faut tester l'intersection entre le nouveau prédécesseur de s et le nouveau successeur de s'.

```
//ecole polytechnique - modex acm - c.durr (2008)

/* lit une suite de n segments et affiche le nombre de segments qui
   intersectent, disons k. utilise un algorithme de balayage pour cela.
   Complexite est O(nlogn + k).
*/
```

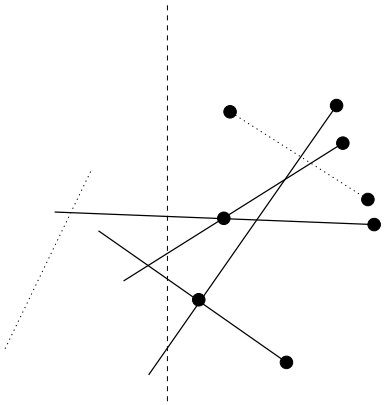


FIGURE 17.6 – Les segments en pointillés ne sont pas considérés. Les points représentent les évènements intéressants.

```
import java.util.*;
import java.awt.geom.*;


class Balai implements Comparator<Line2D.Double> {

    static double xt; // position balai

    /* calcule l'intersection de la droite x=xt (parallele a l'axe y)
       et du segment s (vue comme une droite)
       [...]
    */
    static double y(Line2D.Double s, double xt) {
        return s.y1 + (s.y2-s.y1)*(xt-s.x1)/(s.x2-s.x1);
    }

    /* on trie les segments par rapport a l'intersection de la droite
     * x=xt (parallele a l'axe y). Le premiers segments sont ceux
     * dont l'intersection est haute. On a la promesse que les droite
     * ne sont pas paralleles a l'axe y. Si deux segments
     * intersectent au meme endroit, on decale un peu la droite 'a
     * x=xt-1.
     */
    static int compareTo(Line2D.Double a, Line2D.Double b) {
        double diff = y(a,xt) - y(b,xt);
        return (int)Math.signum(diff!=0 ? diff : y(a,xt-1)-y(b,xt-1));
    }
}

//------------------------------------ evenements

class Event implements Comparable<Event> {
    static final int ARRIVEE=0, INTERSECT=1, DEPART=2;
    int event;
    double x; // heure de l'evenement
    Line2D.Double seg; // segment de l'evenement
```

```
    Event(int _event, double _x, Line2D.Double _seg) {
        event = _event;
        x = _x;
        seg = _seg;
    }

    // les evenements seront dans l'ordre du temps x et en priorite ARRIVE..
    public int compareTo(Event e) {
        int dx = (int)Math.signum(x - e.x);
        if (dx!=0)
            return dx;
        else
            return event-e.event;
    }
}

//-------------------------------------- Main

class SegmIntersect {

    static double xt; // position du balai

    PriorityQueue<Event> q= new PriorityQueue<Event>();

    TreeSet<Line2D.Double> b= new TreeSet<Line2D.Double>(new Balai());

    // retourne le point d'intersection ou null
    // considere les segments comme des droites
    Point2D.Double intersect(Line2D.Double a, Line2D.Double b) {
        if (!a.intersectsLine(b))
            return null;
        double adx = a.x2-a.x1;
        double ady = a.y2-a.y1;
        double bdx = b.x2-b.x1;
        double bdy = b.y2-b.y1;
        double x = (b.y1-a.y1+ady*a.x1/adx-bdy*b.x1/bdx)/(ady/adx - bdy/bdx);
        double y = ady*x/adx + a.x1;
        return new Point2D.Double(x,y);
    }

    // est-ce que le segment de a va intersecter avec le segment d'avant ?
    void detect(Arbre a) {
        if (a==null || a.prec==null)
            return;
        Point2D.Double p = intersect(a.prec.seg, a.seg);
        if (p==null)
            return;
        // intersection dans le futur ?
        if (p.x > Arbre.xt)
            q.add(new Event(Event.INTERSECT, p.x, a.seg));
    }

    void print(String msg, Line2D.Double s) {
```

```
        System.out.println(msg+"\t"+s.x1+" "+s.y1+" "+s.x2+" "+s.y2);
    }

    void add(Line2D.Double seg) {
        print("add",seg);
        racine = Arbre.add(racine, seg, null, null);
        Arbre a = Arbre.find(racine, seg);
        detect(a);
        detect(a.succ);
    }

    void remove(Line2D.Double seg) {
        print("del",seg);
        Arbre a = Arbre.find(racine, seg).succ;
        racine = Arbre.remove(racine, seg);
        detect(a);
    }

    void intersect(Line2D.Double seg) {
        Arbre a = Arbre.find(racine, seg);
        print("inter ",seg);
        System.out.println(""+a);
        print("  avec",a.prec.seg);
        // echanger dans l'arbre
        Line2D.Double tmp = a.seg;
        a.seg = a.prec.seg;
        a.prec.seg = tmp;
        detect(a.succ);
        detect(a.prec);
    }

    int count(Vector<Line2D.Double> all) {
        int countIntersect = 0;
        // creer la file d'evenements
        for (Line2D.Double l : all) {
            q.add(new Event(Event.ARRIVEE, Math.min(l.x1, l.x2), l));
            q.add(new Event(Event.DEPART, Math.max(l.x1, l.x2), l));
        }
        // traiter les evenements
        while (! q.isEmpty()) {
            Event e = q.poll();
            Arbre.xt = e.x; // on avance au moment de l'evenement
            Arbre.print(racine, "");
            System.out.println("-----");
            switch (e.event) {
            case Event.ARRIVEE:
                add(e.seg); break;
            case Event.DEPART:
                remove(e.seg); break;
            case Event.INTERSECT:
                countIntersect++;
                intersect(e.seg); break;
            }
        }
```

```
        return countIntersect;
    }


    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        in.useLocale(Locale.US);
        SegmIntersect s = new SegmIntersect();
        Vector<Line2D.Double> all = new Vector<Line2D.Double>();
        while (in.hasNext()) {
            double x1 = in.nextDouble();
            double y1 = in.nextDouble();
            double x2 = in.nextDouble();
            double y2 = in.nextDouble();
            all.add(new Line2D.Double(x1,y1,x2,y2));
        }
        System.out.println(""+s.count(all));
    }
}
```

17.10 Diagramme de Voronoï

Étant donné n points on veut calculer un diagramme de Voronoï, c'est-à-dire une séparation du plan en zones (cellules), où chaque à point donné on associe une cellule qui contient tous les points qui lui sont les plus proches.

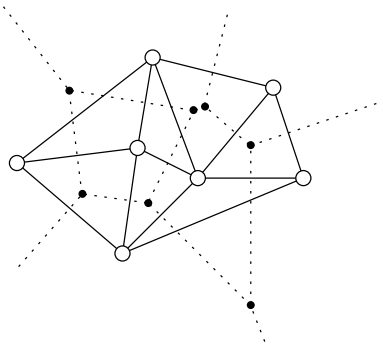


FIGURE 17.7 – En pointillé le diagramme de Voronoï définit par les points blancs. En noir les arêtes de la triangulation de Delaunay. Les points noirs sont les médiatrices des triangles et aussi les sommets du diagramme de Voronoï.

Ce problème peut aussi être résolu par un algorithme de balayage, par l'algorithme de Fortune en $O(n \log n)$. Le dual du diagramme de Voronoï est la triangulation de Delaunay. C'est un graphe dont les sommets sont les points donnés. On peut montrer que l'arbre de recouvrement qui recouvre tous les points est un sous-graphe de la triangulation de Delaunay. Ainsi on peut par exemple résoudre le problème de l'arbre de recouvrement minimal euclidien en temps $O(n \log n)$.

17.11 Enveloppe convexe

Il n'est pas possible en général de calculer l'enveloppe convexe en temps $o(n \log n)$. Pour s'en convaincre, soit une séquence de n nombres $a_1, \ldots, a_n$. Le calcul de l'enveloppe convexe des

points $(a_1, a_1^2), \ldots, (a_n, a_n^2)$ va les mettre dans l'ordre. Donc si on pouvait faire ce calcul en moins de $n \log n$ opérations, cela donnera un algorithme de tri de même complexité. Pour faire cet argument on devrait le formuler dans un modèle de calcul particulier.

L'algorithme d'Andrew qu'on décrit maintenant est essentiellement le balayage de Graham, mais ne traite pas les points suivant leur angle autour d'un point de repaire, mais en fonction de leur coordonnée x. Nous décrivons seulement comment obtenir la partie du haut de l'enveloppe convexe. Initialiser $L = \{\}$. Pour tout point p dans l'ordre des coordonnées x, faire l'opération de mise à jour suivante. Tant que $|L| \geqslant 2$ et le triangle formé des deux derniers points de L et de p est orienté normal, enlever le dernier point de L. Puis ajouter p à L.

```
/** Considere le Triangle a-b-c
    et retourne une valeur positive si oriente normal
    retourne une valeur negative si oriente dans le sens contraire
    et zero si a,b,c sont co-lineaires.
*/
static double cross(Point2D.Double a,
                    Point2D.Double b,
                    Point2D.Double c) {
    return (b.getX() - a.getX()) * (c.getY() - a.getY())
        - (b.getY() - a.getY()) * (c.getX() - a.getX());
}

/** Calcule l'enveloppe convexe. Va changer l'ordre de t.
 */
static Vector<Point2D.Double> convexHull(Point2D.Double t[]) {
    Arrays.sort(t, new Comparator<Point2D.Double>() {
            public int compare(Point2D.Double a, Point2D.Double b) {
                int dx = (int)Math.signum(a.getX()-b.getX());
                int dy = (int)Math.signum(a.getY()-b.getY());
                return dx!=0 ? dx : dy;
            }
        });
    // -- parties haute et base de l'enveloppe
    Point2D.Double [] top = new Point2D.Double[t.length];
    Point2D.Double [] bot = new Point2D.Double[t.length];
    int ntop = 0, nbot = 0; // -- longueurs respectives
    for (Point2D.Double p: t) {
        while (ntop>=2 && cross(top[ntop-2], top[ntop-1], p)>=0)
            ntop--;
        top[ntop++] = p;

        while (nbot>=2 && cross(bot[nbot-2], bot[nbot-1], p)<=0)
            nbot--;
        bot[nbot++] = p;
    }
    // enlever premier element de top et dernier de bot
    // concatener bot et top, c'est l'enveloppe convexe
    Vector<Point2D.Double> ret = new Vector<Point2D.Double>();
    for (int i=0; i<nbot-1; i++)
        ret.add(bot[i]);
    for (int i=ntop-1; i>0; i--)
        ret.add(top[i]);
    return ret;
```

```
    }
```

Chapitre 18

Tester

18.1 Préparer un fichier de test

Dans les énoncés seul un petit exemple est donné, mais en général on indique combien de mots, ou nombres votre programme doit pouvoir traiter. À vous de créer une instance de taille maximale pour tester votre programme sur le dépassement des limites de tableaux et sur le temps d'exécution.
En général vous avez le shell bash. Vous pouvez toujours vérifier avec echo $SHELL, et lancer bash le cas échéant.

18.2 Les wildcards

Le shell remplace * par tous les noms de fichiers de votre répertoire. Pour éviter cela on peut écrire * entre double guillemets ou le précéder d'un backslash.

18.3 Petits exemples

Faire une boucle est simple avec la commande for.

```
for i in {1..5}
do
  echo ‘for j in {1..$i}; do echo $j ; done‘
done
```

produit

```
1
1 2
1 2 3
1 2 3 4
1 2 3 4 5
```

La commande entre les simples guillemets obliques est exécutée et remplacé par sa sortie. Ainsi on peut donner en paramètre à une commande la sortie d'une autre commande. Ne confondez pas avec les guillemets simples non-obliques ’ qui servent à construire des chaînes de caractères où les variables (débutant par $) ne sont pas remplacées par leur valeur.
Pour donner en *entrée* à une commande la sortie d'une autre, utilisez le *pipe*, comme dans

```
(for i in {1..10}; do echo $RANDOM; done) | sort -n
```

ce qui produit 10 entiers au hasard en ordre croissant.
La variable $RANDOM est substituée par un entier aléatoire positif. Un réel entre 0 et 9.999 pourrait être généré comme suit :

```
echo "$(( RANDOM % 10)).$(( $RANDOM % 1000 ))"
```

Ici le shell interprète tout ce qui se trouve entre $((et)) comme une expression arithmétique.

18.4 Générer des mots au hasard

Le fichier /usr/share/dict/words contient une liste de mots anglais. La commande sort -R permet de les mettre dans un ordre qui n'est pas lexicographique. La commande head -10 permet de ne garder que les 10 premières lignes de l'entrée standard.

```
durr@dell-gris:~$ sort -R /usr/share/dict/words | head -10
morgue's
abrogations
hurler
watered
condole
colloquia
Eugene
refinery's
Ford's
restitution's
```

Et pour ne pas avoir des mots avec l'apostrophe on peut utiliser grep -v, où le -v inverse la sélection. Pour mettre tout en majuscules on peut utiliser tr (comme translate).

```
durr@dell-gris:~$ sort -R /usr/share/dict/words | grep -v "'" | head -10 | \
   tr [:lower:] [:upper:]
EDDY
REPROACHFULLY
CAVING
OFFAL
CRUCIFYING
TRANSMIGRATING
CLERGYWOMAN
PHIALLED
HERMITAGE
NAYSAYER
```