# THÉORIE ET ALGORITHMIQUE DES GRAPHES
# LICENCE INFORMATIQUE

PHILIPPE LANGEVIN, IMATH, UNIVERSITÉ DE TOULON.

| | |
|---|---|
| tag.tex | 2021−11−10 11:47:44.603005308 |
| promenade.tex | 2021−11−10 11:48:04.996445904 |
| implantation.tex | 2021−11−10 11:43:17.496233025 |
| macros.tex | 2021−11−10 08:26:16.563371958 |
| euler.tex | 2021−11−10 08:18:12.246882641 |
| topo.tex | 2021−01−26 08:07:48.490984981 |
| reduction.tex | 2021−01−26 08:07:48.752990708 |
| np.tex | 2021−01−26 08:07:48.799991735 |
| planaire.tex | 2021−01−26 08:07:48.756990795 |
| hamilton.tex | 2021−01−26 08:07:48.800991757 |
| disjoint.tex | 2021−01−26 08:07:48.802991801 |
| coloriage.tex | 2021−01−26 08:07:48.803991822 |
| connexe.tex | 2021−01−26 08:07:48.802991801 |
| backtrack.tex | 2021−01−26 08:07:48.815992085 |
| acm.tex | 2021−01−26 08:07:48.816992107 |
| devoir.tex | 2020−12−12 09:39:33.405428396 |
| rush.tex | 2020−10−27 09:39:36.139463151 |

*Date*: dernière compilation :3 décembre 2021.

## Table des matières

## DEUXIÈME RÉCRÉATION.

# LE JEU DES PONTS ET DES ILES.

PARMI les divers travaux des mathématiciens sur cette branche de la science de l'étendue que l'on nomme *Géométrie de situation*, on rencontre, dès l'origine, un fameux Mémoire d'Euler, connu sous le nom de *Problème des Ponts de Kœnigsberg;* nous donnons, d'après les *Nouvelles Annales de Mathématiques,* un commentaire de cet opuscule, qui a paru en latin dans les *Mémoires de l'Académie des sciences de Berlin* pour l'année 1759, et qui a pour titre : *Solutio problematis ad Geometriam situs pertinentis.*

### LE MÉMOIRE D'EULER.

1° Outre cette partie de la Géométrie qui s'occupe de la grandeur et de la mesure, et qui a été cultivée dès les temps les plus reculés, avec une grande application, Leibniz a fait mention, pour la première fois, d'une autre partie encore très inconnue actuellement, qu'il a appelée *Geometria situs.* D'après lui, cette branche

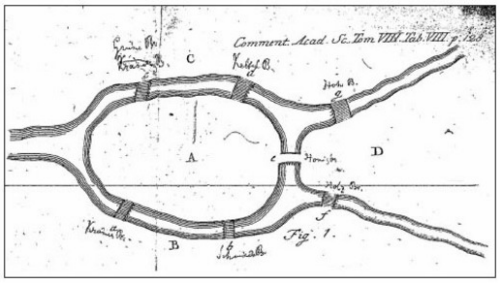



FIGURE 1. Peut-on faire une promenade passant une et une seule fois par chacun des sept ponts de la ville de Koenigsberg ?

## 1. Chemin eulérien

La théorie des graphes s'est développée au cours du XX^e siècle, dans la note [18], Las Vergnas nous rappelle que terminologie de graphe a été introduite par Sylvester en 1877, et que le premier livre sur la théorie des graphes a été écrit par D. König en 1936. La génèse de la théorie des graphes semble être une étude de Léonard Euler, un très célèbre mathématicien du XVIII^e siècle. Dans un article publié en 1736, il traite un problème devenu classique, illustré par la devinette : peut on faire une promenade passant une fois par chacun des sept ponts de la ville de Koenigsberg ? Il suffit de faire quelques essais pour se convaincre de l'impossibilité de réaliser une telle promenade. L'objectif de cette section est de dégager un résultat général.

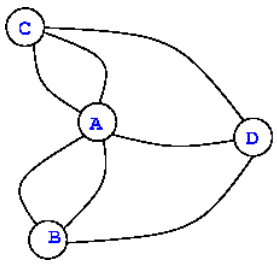



FIGURE 2. 2-graphe.

La question se traduit directement dans le langage de la théorie des graphes par : existe-il un chemin passant une et une seule fois par les arêtes du graphe ? Dans la terminologie de Claude Berge [1], il s'agit d'un 2-graphe car certains sommets sont reliés par au plus deux arêtes. Dans la suite, nous nous intéressons uniquement aux graphes simples : sans arête multiple et sans boucle.

1.1. **Terminologie.** Un graphe $\Gamma(S, A)$ est la donnée de deux ensembles finis : un ensemble de sommets $S$ et un ensemble d'arêtes $A$. Une arête est une paire de sommets, ce sont les extrémités de l'arête. Une arête $\{x, y\}$ est notée $xy$, et les sommets sont dits adjacents. Un chemin de

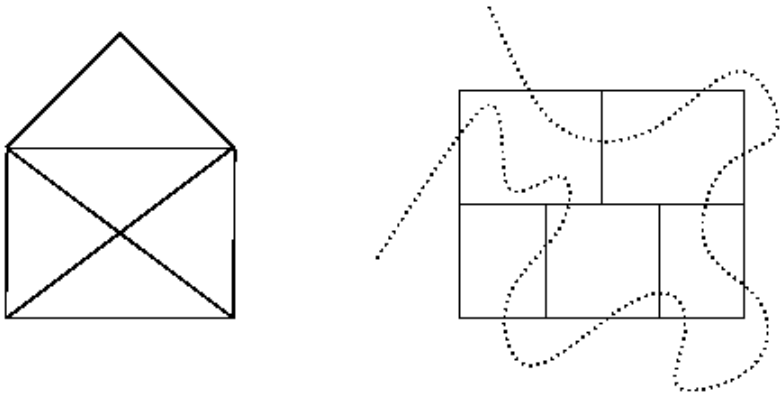


FIGURE 3. Peut-on tracer la petite maison d'un trait de crayon sans passer deux fois par le même segment ? Peut-on tracer une courbe qui coupe une et une seule fois tous les segments de la figure ?

longueur $n$ est une suite de sommets $x_0$, $x_1$, ..., $x_n$ tels que :

$$\forall i, \quad 0 \leq i < n \Longrightarrow x_i x_{i+1} \in A,$$

Les sommets $x_0$ et $x_n$ sont les extrémités du chemin, $x_0$ est l'origine et $x_n$ le sommet terminal. On parle de cycle quand $x_0 = x_n$. On dit que $x$ est connecté à $y$, et on note $x \rightsquigarrow y$ quand il existe un chemin d'extrémité $x$ et $y$. Il s'agit là d'une relation symétrique et transitive qu'il convient de prolonger par réflexivité. Un chemin est dit simple quand il ne passe jamais plus d'une fois par une même arête. Un chemin élémentaire ne passe pas deux fois par un même sommet. L'ensemble des sommets voisins d'un sommet $x$ :

$$\text{voisin}(x) = \{y \in S \mid xy \in A\}, \quad \deg(x) = \sharp\text{voisin}(x),$$

le degré d'un sommet est égal au nombre d'arêtes incidentes. Un sommet de degré pair est dit pair, un sommet de degré impair est dit impair. Un sommet de degré nul est dit isolé. Un sommet adjacent à tous les autres sommets est dit dominant.

Nous avons l'amusante relation des paires et de l'impair :

**Lemme 1** (parité des impairs)**.** *Dans un graphe,* $\Gamma(S, A)$ *:*

$$\sum_{x \in S} \deg(x) = 2|A|$$

*en particulier, le nombre de sommets impairs est toujours pair !*

*Démonstration.* Notons $A_s$ les arêtes incidentes au sommet $s$,

$$A = \bigcup_{s \in S} A(s).$$

Une triple intersection des $A_s$ est vide, une double intersection non vide est une arête du graphe. Le principe d'inclusion et d'exclusion [12] s'applique sans difficulté et donne :

$$|A| = \sum_{s \in S} \deg(s) - |A|.$$

□

**Exercice 1** (dénombrement des graphes)**.** *Dénomnbrer le nombre de graphes d'ordre 8.*

**Exercice 2.** *Prouver qu'un graphe d'ordre supérieur à 1, possède deux sommets de degré identique.*

**Exercice 3.** *De tout parcours, on peut extraire un parcours élémentaire ayant les mêmes extrémités. Détaillez cette affirmation !*

**Exercice 4.** *Dans un graphe acyclique ayant au moins une arête, il existe un sommet de degré 1. Expliquez !*

Nous arrivons au problème de décision qui nous importe.

**Problème 1** (chemin eulérien)**.** *Etant donné un graphe. Existe-t-il un chemin passant une et une seule fois par toutes les arêtes de ce graphe ?*

**Remarque 1.** *Dans la littérature, le graphe est dit eulérien quand il existe un cycle eulérien, semi-eulérien quand il existe un chemin eulérien non cyclique.*

À condition de poser $x \rightsquigarrow x$ pour tout sommet $x$, la relation $\rightsquigarrow$ est une relation d'équivalence : réflexive, symétrique et transitive. Les classes d'équivalence sont des composantes connexes. Le graphe est dit connexe quand il possède une seule composante connexe. Autrement dit, pour toute paire de sommets $x$ et $y$, il existe un chemin d'extrémité $x$ et $y$.

1.2. **Caractérisation.** Notons d'une part qu'un graphe eulérien sans point isolé est forcément connexe, et que d'autre part, deux sommets non isolés d'un graphe eulérien sont connectés.

**Théorème 1.** *Un graphe sans point isolé est eulérien si et seulement si il est connexe et tous ses sommets sont pairs. Un graphe sans point isolé est semi-eulérien si et seulement si il possède exactement 2 sommets impair.*

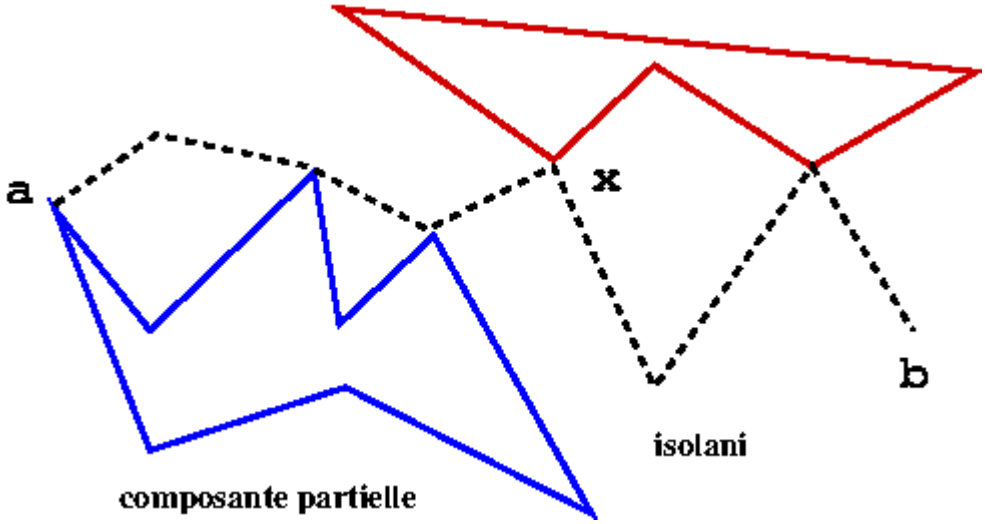



FIGURE 4. Les composantes connexes du graphe partiel qui ne sont pas des singletons forment des cycles eulériens.

*Démonstration.* Tous les sommets sont sur une arête au moins. Comme un chemin eulérien passe par toutes les arêtes, il passe par tous les sommets. Le long d'un chemein, tous les sommets sont de degré pairs, sauf les deux éventuelles extrémités. Montrons par induction sur le nombre d'arêtes qu'un graphe connexe avec 0 ou 2 sommets impairs est eulérien. Supposons qu'il existe deux sommets impairs $a$ et $b$. Il existe un chemin $\mu$ qui relie $a$ et $b$. On construit un sous-graphe partiel par suppression des arêtes de ce chemin. Les composantes connexes de ce graphe partiel forment des sous-graphes dont les sommets sont tous pairs. Certaines de ces composantes sont réduites à un point, les autres forment des cycles eulériens.

Pour chacune de ces composantes, on choisit un sommet représentant le long du chemin $\mu$. On obtient un chemin eulérien en remplaçant un représentant $x$ de $\mu$ par le cycle eulérien $\mu_x$ du graphe partiel. □

**Exercice 5.** *Résoudre les deux puzzles* FIG. *(3).*

1.3. **Algorithme.** L'algorithme récursif `eulerien(s, G)` imprime un chemin eulérien d'origine $s$ dans le graphe $G$. Il suppose que le chemin existe, et que le sommet $s$ est bien de degré impair dans le cas non cyclique.

```
Eulerien( s : sommet, G : graphe )

   P  ← promenade( s, G )
   pour chaque sommet x  de P
        si degre( x ) > 0 alors
           Eulerien(x, G)
        sinon
           imprimer( x )
```

La routine `promenade(s, G)` construit une chaîne simple et maximale d'origine $s$. Au cours de cette balade, les arêtes intermédiaires sont retirées du graphe de sorte à garantir l'arrêt du code et la simplicité des chemins.

**Proposition 1.** *L'algorithme `eulerien(s, G)` est quadratique en l'ordre du graphe.*

*Démonstration.* □

## 2. Mise en oeuvre

2.1. **organisation.** En pratique, nous réaliserons une bibliothèque en langage `C`. Les sources des procédures et fonctions destinées à être utilisées dans des expériences numériques sont déposées dans un répertoire unique. Elles sont compilées en des fichiers objets et intégrées dans une bibliothèque `libgraphe.a`.

La hiérarchie FIG. (5) est un modèle d'organisation de répertoire et de fichiers. Il ne faut pas hésiter à se faciliter la tâche en ajoutant des vatiables et `alias` dans votre fichier `.bashrc` :

```
1 export GRAPHE="$HOME/cours/graphe"
2 alias mklib="make -C $GRAPHE/lib"
```

Les structures de données et algorithmes de bases seront implantés dans des fichiers séparés pour constituer une bibliothèque, et seront exploiter dans des exériences. Dans l'exemple, la structure de pile de sommets est implanter dans les fichiers `pile.[ch]`, utilisée dans le répertoire.

Les sources C seront déposées dans un répertoire unique piloté par un fichier `makefile` pour maintenir à jour une bibliothèque. Vous trouverez ci-dessous un exemple de fichier `makefile`. Pour ce cours, les sources C seront considérées comme acceptables à condition d'être compilées avec

```
1  SRC=$(wildcard *.c)
2  OBJ=$(SRC:.c=.o)
3  CFLAGS = -Wall -g
4  ifeq ($(shell hostname), imath02.univ-tln.fr)
5      CFLAGS = -O2
6  endif
7
8  libgraphe.a : $(OBJ)
9      ar -cr libgraphe.a *.o
10     #nm libgraphe.a
11 %.o:%.c
12     gcc $(CFLAGS) -c $*.c
13 joli :
14     indent -kr *.c
15 clean:
16     rm -f *~ *.o
17 proper:
18     rm -f *.a
```

LISTING 1. La bibliothèque statique `libgraphe.a`

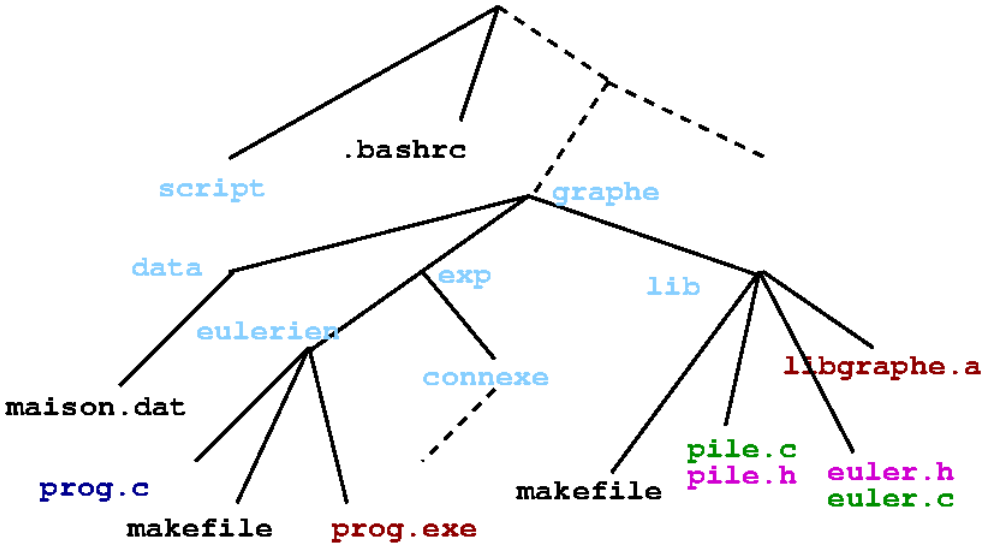



FIGURE 5. Hiérarchie des fichiers.

```
1 OPTIONS= -Wall -g
2 OBJET=graphe.o inout.o
3 libgraphe : $(OBJETS)
4 	ar crs libgraphe.a $(OBJETS)
5
6 graphe.o : graphe.c
7 	gcc $(OPTIONS) -c graphe.c
8 inout.o : inout.c
9 	gcc $(OPTIONS) -c inout.c
```

LISTING 2. Un premier makefile

l'option `-Wall`, sans erreur ni avertissement. L'option de débogage `-g` est indispensable pour pouvoir utiliser les débogueurs `gdb` et `valgrind`.

**Travaux-Pratiques 1** (Préliminaire)**.** *Préparer l'environnement de travail pour les séances à venir.*

- *ajouter l'alias* `mklib` *et la variable* `GRAPHE` *dans le fichier de commande* `.bashrc` *;*
- *mettre en place la hiérarchie de fichiers ;*

2.2. **Format de fichier source.** On choisit un format de fichiers assez sommaire pour représenter les graphes. Un modèle de fichier texte pour décrire le graphe de la maison à 5 sommets est le suivant :

```
#          0
#        /   \
#      1 --- 2
#      |   X |
#      3 --- 4
nbs=5
0 1
0 2
1 2
1 3
1 4
2 3
2 4
3 4
```

```
1  typedef struct {
2      int       nbs;   // ordre du graphe
3      int      **mat;  // matrice dadjacence
4      //et plus tard des champs...
5      char*    idt;    // identificateur
6      liste    *adj;   // liste d'adjacence
7  } graphe;
8
9  graphe initGraphe( int nbs );
10 graphe aleatoire ( float p, int n );
11 void   liberer ( graphe G );
12 int degre( int s, graphe G );
```

LISTING 3. graphe.h

2.3. **matrice d'adjacence.** Un graphe $G(S, A)$ d'ordre $s$ dont l'ensemble des sommets $S$ est l'intervalle $\{0, 1, \ldots, n-1\}$ est complètement défini par une matrice d'adjacence. Il s'agit de la matrice booléenne

$$A_{i,j} = 1 \Longleftrightarrow ij \in A.$$

**Travaux-Pratiques 2** (commun)**.** *Dans les fichiers* `graphe.[ch]`*, coder les fonctions :*

- `graphe initGraphe( int n )` *: allocation et initialisation des champs.*
- `graphe aleatoire( float p, int n )` *: construction d'un graphe,* $p$ *est la probabilité de connecter deux sommets.*
- `void liberer( graphe g)` *: libération de la mémoire.*
- `int degre( int s, graphe G )`*.*

2.4. **Entrée-sortie.** Une fonction d'entrée-sortie n'est pas toujours commode à écrire. La facilité d'emploi d'une fonction d'entrée sortie est proportionnelle au temps qui aura été consacré par le programmeur !

Le service minimum de la fonction `minilire` n'aura pas couté bien cher en temps de développement ! Elle impose un format de fichier pas trés agréable pour l'utilisateur. Pour analyser des fichiers plus complexes en terme d'entité lexicale, l'usage des automates est un moyen de bien faire. Pour gagner du temps, le développeur peut utiliser des outils de compilation comme `flex`, voire `bison`.

**Travaux-Pratiques 3** (entrée-sortie)**.** *Il s'agit d'implanter des fonctions d'entrées-sorties pour manipuler les petits graphes au format décrit*

```
1  void minilire ( FILE * src )
2  {
3      int nbs, x, y;
4      while ( 0 == fscanf(src, "nbs=%d", &nbs ) ) fgetc( src );
5      printf ("ordre=%d\n", nbs );
6      // allocation
7      while ( ! feof ( src) ) {
8          if ( fscanf (src, "%d%d", &x, &y ) )
9                  printf ("x=%d y=%d\n", x, y );
10         else fgetc ( src );
11     }
```

LISTING 4. Le service minimum en matière d'entrée/sortie !

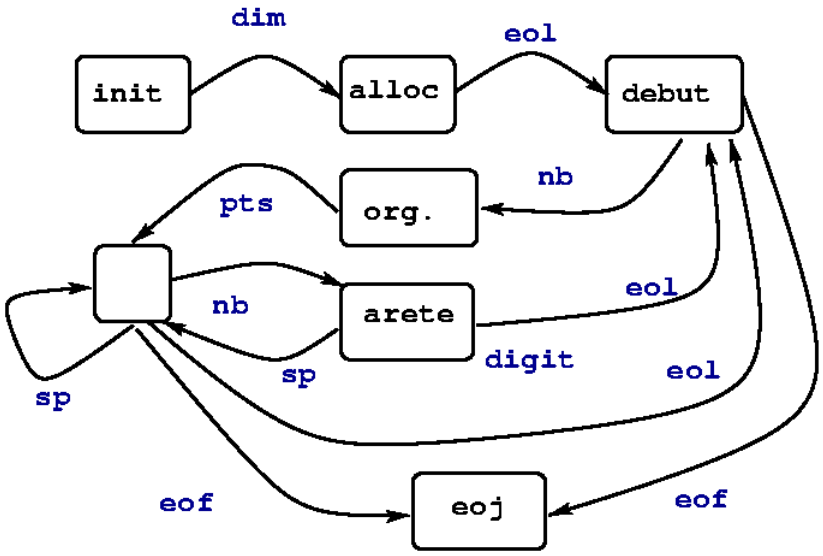



FIGURE 6. Automate pour la lecture d'un graphe au format xyz.

*dans (2.2) en vous appuyant sur l'automate de la figure* FIG. *(6) et l'analyseur lexical* LIST. *(7). Dans les fichiers* `inout.[ch]`*, coder les fonctions :*

- `graphe lireGraphe( char* nom )` *: lecture d'un graphe au format texte ;*
- `void ecrire( graphe G)` *: écriture au format texte ;*
- `void dessiner( graphe G )`.

```
1 graphe lireGraphe( char* nom );
2 void   ecrireGraphe( graphe G );
3 void   dessiner( graphe G );
```

LISTING 5. inout.h

```
graph maison {
0 -- 1; 0 -- 2; 1 -- 2; 1 -- 3;
1 -- 4; 2 -- 3; 2 -- 4; 3 -- 4;
}
```

LISTING 6. maison au format dot

```
DOT(1)   General Commands Manual                DOT(1)
NAME
       dot - filter for drawing directed graphs
       neato - filter for drawing undirected graphs
       ...
SYNOPSIS
       dot [options] [files]
       neato [options] [files]
```

*La commande externe* **dot** *( et ses cousines ) du projet graphiz permet de dessiner des beaux graphes à partir d'une source texte. Elle sera utilisée dans la fonction* **void dessiner(graphe G)** *pour construire image du graphe G via un appel de la fonction* **system** *de la* **glibc**.

```
SYSTEM(3) Manuel du programmeur Linux SYSTEM(3)
NOM
       system - Executer une commande shell
SYNOPSIS
      #include <stdlib.h>
       int system(const char *command);
DESCRIPTION
```

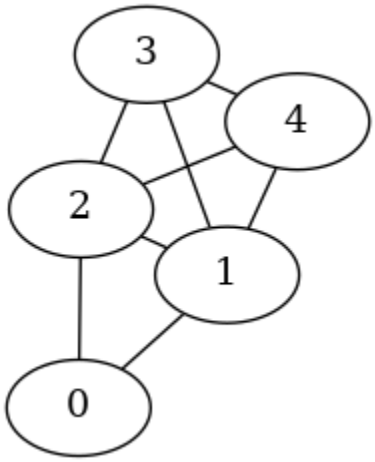



FIGURE 7. neato -Tpng maison.dot -o maison.png.

```
La fonction de bibliotheque system() utilise fork
(2) pour crer un processus fils qui execute la
commande d'interpreteur indiquee dans command...

```

## 3. PILE ET PROMENADE

On choisit la structure de pile de sommets pour mettre en oeuvre l'algorithme `eulerien`.

**Exercice 6.** *Comment adapter l'algorithme* `eulerien(G)` *pour une promenade à pile.*

3.1. **pile de sommets.** Les $n$ sommets d'un graphe d'ordre $n$ seront systématiquemnt les entiers de 0 à $n-1$. Une pile de sommets est donc une pile d'entiers.

**Exercice 7** (pile)**.** *Comment implanter la structure de pile sur la sytucture de* `liste` *?*

L'hypercube de dimension $r \geq 0$ est le graphe d'ordre $n = 2^r$ dont les sommets sont les entiers inférieurs à $n$, deux sommets sont adjacents si leurs représentations binaires diffèrent d'un chiffre.

**Exercice 8** (hypercube)**.** *Quels sont les paramètres de l'hypercube de dimension r ? L'hypercube est-il eulérien ? Ecrire une fonction* `graphe hypercube(int r )` *qui construit le graphe de l'hypercube de dimension r.*

```
1  int getoken( FILE *src )
2  {
3      int car;
4      car = fgetc( src );
5      if ( car == '#' ) do car = fgetc(src) ; while ( car != '\n
       ' && car != EOF );
6      if ( car == ' ' ) {
7              while ( car == ' ' ) car = fgetc( src );
8              ungetc( car , src );
9              car = ' ';
10     }
11     switch( car ) {
12         case '=' : fscanf(src, "%d", &value); return DIM;
13         case ':' : return PTS;
14         case '\n' : return EOL;
15         case ' ' : return SP;
16         case EOF : return EOF;
17     }
18     if ( isdigit ( car ) ) {
19             ungetc( car, src );
20             fscanf(src, "%d", & value ); return NB;
21     }
22     return SMILE;
23 }
```

LISTING 7. Un exemple d'analyseur lexical.

### 3.2. **Expérience eulérien.**

**Travaux-Pratiques 4** (commande `/usr/bin/time`)**.**
*Ecrire une commande* `eulerien.exe` *qui détermine un cycle eulerien dans l'hypercube de dimension* $r$*.*
*Utiliser la commande externe* `time` *pour déterminer la forme du temps de calcul en fonction de* $r$*.*
*Interpréter le résultat.*

3.3. **Liste d'adjacence.** En pratique, la représentation par matrice d'adjacence peut se révéler octophage et chronophage. Typiquement, lors de la détermination des voisins d'un sommet.

**Travaux-Pratiques 5.** *Dans le fichier* `graphe.[ch]`

```
pile promenade( sommet x , graphe G )
{
variable
  y : sommet
  P : pile
  initialiser(P)
  tantque    ( degre(x) > 0 )
    y ← un voisin de x
    empiler( y, P );
    deconnecter x et y
    x ← y;
}
retourner P
ftq
```

LISTING 8. Promenade dans $G$ à partir du sommet $x$

```
1  #ifndef PILE_H
2  #define PILE_H
3  typedef struct _liste_ {
4       int s;
5       struct _liste_  * svt;
6  } enrliste , * liste ;
7
8  void    empiler( int s, liste  *p );
9  int     depiler (  liste  *p );
10 #endif
```

LISTING 9. Structure de pile basée sur des listes chainées

— *coder la fonction* `void adjacence( graphe G)` *qui construit la table des listes d'adjacence.*
— *Adapter la procédure* `liberer( graphe G )`

## 4. Composante Connexe

4.1. **Connexité.** À condition de poser $x \rightsquigarrow x$ pour tout sommet $x$, la relation $\rightsquigarrow$ est une relation d'équivalence : réflexive, symétrique et transitive. Les classes d'équivalence sont des composantes connexes. Le

```
1 typedef struct {
2     int       nbs;    // ordre du graphe
3     int      **mat;   // matrice dadjacence
4     //et plus tard des champs...
5     char*    idt;     // identificateur
6     liste    *adj;    // liste d'adjacence
7 } graphe;
8
9 void conversion( graphe G );
```

LISTING 10. `graphe.h`

graphe est dit connexe quand il possède une seule composante connexe. Autrement dit, pour toute paire de sommets $x$ et $y$, il existe un chemin d'extrémité $x$ et $y$. Un parcours récursif des sommets d'un graphe permet de décider du caractère connexe d'un graphe.

**Problème 2** (connexe)**.** *Etant donné un graphe. Le graphe est-il connexe ?*

**Exercice 9** (abondance)**.** *Montrer que les graphes connexes sont plus nombreux que les graphes déconnectés.*

**Exercice 10** (Berge)**.** *(1) Quel est le nombre d'arêtes de $K_n$ ?*

*(2) Montrer dans un graphe à $p$ composantes connexes*

$$m \leq \frac{1}{2}(n-p)(n-p+1)$$

*(3) Montrer qu'un graphe possédant plus de $\frac{1}{2}(n-1)(n-2)$ arêtes est obligatoirement connexe.*

4.2. **Parcours récursif.** L'algorithme `parcours(s,G)` effectue un parcours récursif et en profondeur du graphe $G$ à partir du sommet $s$. Les sommets sont marqués au fur et à mesure de l'exploration du graphe. Le nombre de sommets marqués est maintenu dans la variable compteur. Le graphe est connexe quand tous les sommets sont marqués.

Les paramètres et variables locales d'une fonction sont stockés dans une zone particulière de la mémoire : la pile (stack). La taille de cette zone mémoire est souvent limitée et les appels récursifs peuvent conduire à un débordement de pile (stack overflow). La commande `ulimit` permet de connaître la taille par défaut de la pile d'un processus, elle permet de procéder à des réglages de cette ressource.

```
parcours( x, G )
  marquer[x] ← 1
  compteur++
  pour chaque voisin y de x dans G
       si marque[y] < 1 alors
                parcours( y , G )


connexe( graphe G )
  initialiser les marques
  compteur ← 0
  choisir un sommet s
  parcours( s , G )
  retourner compteur == ordre(G)
```

LISTING 11. Marquage des sommets par un parcours récursif en profondeur

**Exercice 11** (élimination de la récursivité)**.** *Comment utiliser explicitement une structure de de pile pour éliminer la récursivité de* LIST. *(11).*

**Exercice 12.** *Ecrire un algorithme pour dénombrer le nombre de composantes connexes d'un graphe.*

4.3. **Matrice d'ajacence.** Le temps de calcul pour l'identification des voisins d'un sommet dans le cas d'une implantation des graphes par matrice d'adjacence est proportionnel au nombre de sommet.

**Proposition 2** (parcours par matrice d'adjacence)**.** *Dans le cas d'une implantation par matrice d'adjacence, le temps de calcul de* `connexe(G)` *est quadratique en l'ordre $n$ du graphe :*

$$T(m,n) = \Theta(n^2)$$

*Démonstration.* Chaque sommet du graphe est visité une fois, le temps de calcul d'une visite est proportionnel à $n$. □

4.4. **Liste d'adjacence.** Dans le cas des graphes peu dense, il est préférable d'implanter les graphes au moyen de listes d'adjacence.

**Proposition 3** (parcours par liste d'adjcence)**.** *Dans le cas d'une implantation par liste d'adjacence, le temps de calcul de* `connexe(G)` *vérifie :*

$$T(m,n) = \mathrm{O}(m+n)$$

```
1
2  void parcours( int s,  graphe G )
3  {
4  int t;
5  marque[s] = 1;
6  for( t = 0; t < G.nbs; t++ )
7      if ( G.mat[s][t] && ! marque[t] ) {
8          parcours( t , G );
9      }
10 }
```

```
1  typedef struct _ls_ {
2       sommet num;
3       struct _ls_ * next;
4  } enrliste , *liste ;
5
6  typedef struct {
7       int      nbs;
8       liste   *mat;
9  } graphe;
10
11
12 void parcours( int s,  graphe G )
13 {
14 liste  aux;
15 marque[s] = 1;
16 aux = G[s];
17 while ( aux ){
18     if ( ! marque[ aux->num ] )
19         parcours( aux->num , G );
20     aux = aux->next;
21     }
22 }
```

LISTING 12. liste d'adjacence

*où $m$ est le nombre d'arêtes du graphe dont l'ordre $n$.*

*Démonstration.* La recherche des voisins d'un sommet est proportionnelle à son degré. □

**Exercice 13.** *Coder* `int adjacent( sommet x, y, graphe g)` *qui retourne 1 xy est une arête du graphe.*

**Exercice 14.** *Coder* `void relier( sommet x, y, graphe g)` *qui ajoute l'arête xy au graphe.*

**Exercice 15.** *Ecrire une fonction* `void freeGraphe( graphe g)` *qui libère la mémoire allouée au graphe passé en argument dans le cas d'une implantation par liste d'adjacence.*

**Travaux-Pratiques 6** (connexité)**.** *Dans un fichier* `connexe.c`*, implanter une fonction* `int connexe( graphe G, int mode )` *qui retourne le nombre de composante connexe du graphe $G$. Le paramètre permettra de décider du mode opératoire :*

*(1) parcours récursif représentation matricielle ;*

*(2) parcours récursif sur des listes d'adjacences ;*

*(3) parcours profondeur représentation matricielle ;*

*(4) parcours profondeur sur des listes d'adjacences ;*

*Utiliser l'outil de profilage* `gprof` *pour comparrer pour comparer les temps de calcul des différentes approches.*

4.5. **Profilage.** Le profilage d'un code consiste à mesurer le temps de calcul des différentes fonctions, pour d'éventuelles améliorations. La commande `gprof` rapporte le profilage d'un code compilé avec l'option `-pg` du compilateur `gcc`.

```
gcc -Wall -pg prog.c
./a.out 1000000
gprof -b a.out
```

LISTING 13. un exemple de profilage de code

```
1 #include <stdlib.h>
2
3 int P(int t[], int k, int n)
4 {
```

```
5     int i, res = 0;
6     for (i = 1; i < n; i++)
7     if (t[i] > k) res += i * k;
8     return res;
9  }
10
11 int Q(int t[], int k, int n)
12 {
13    int i, res = 0;
14    for (i = 0; i < n; i++)
15    if (t[i] > k) res += i * k;
16    return res;
17 }
18
19 void melange(int t[], int n)
20 {
21    int i, j, tmp, r;
22    for (r = 0; r < n; r++) {
23    i = random() % n;
24    j = random() % n;
25    tmp = t[i];
26    t[i] = t[j];
27    t[j] = tmp;
28    }
29 }
30
31 int main(int argc, char *argv[])
32 {
33    int n = atoi(argv[1]);
34    int *t = calloc(n, sizeof(int));
35    for (int i = 0; i < n; i++) t[i] = i;
36    P(t, n / 2, n);
37    melange(t, n);
38    Q(t, n / 2, n);
39    return 0;
40 }
```

```
Flat profile:
Each sample counts as 0.01 seconds.
```

```
  %   cumulative   self              self     total
 time   seconds   seconds    calls  s/c      s/c     name
 90.98      2.17     2.17        1  2.17     2.17  melange
  4.23      2.28     0.10        1  0.10     0.10  Q
  4.23      2.38     0.10                            main
  1.69      2.42     0.04        1  0.04     0.04  P


                     Call graph


granularity: each sample hit covers 2 byte(s) for 0.41% of 2.42 seconds

index % time    self  children    called     name
                0.10    2.32                 main [1]
                2.17    0.00       1/1       melange [2]
                0.10    0.00       1/1       Q [3]
                0.04    0.00       1/1       P [4]
-----------------------------------------------
                2.17    0.00       1/1       main [1]
[2]     90.0    2.17    0.00       1         melange [2]
-----------------------------------------------
                0.10    0.00       1/1       main [1]
[3]      4.2    0.10    0.00       1         Q [3]
-----------------------------------------------
                0.04    0.00       1/1       main [1]
[4]      1.7    0.04    0.00       1         P [4]
-----------------------------------------------
```

**Exercice 16.** *Analysez le profilage obtenu !*

4.6. **Classe polynomiale.**

**Problème 3** (problème du cycle eulérien)**.** *Etant donné un graphe. Existe-t-il un cycle eulérien ?*

Pour une instance donnée, la réponse est VRAI ou FAUX. Le problème du cycle eulérien est un exemple de problème de décision sur les graphes.

La théorie de la complexité a pour objectif de classifier les problèmes de décision. La classe **P** est constituée des problèmes de décision pour lesquels il existe un algorithme de résolution polynomial.

**Proposition 4.** *Le problème du cycle eulérien est dans la classe* **P***.*

*Démonstration.* □

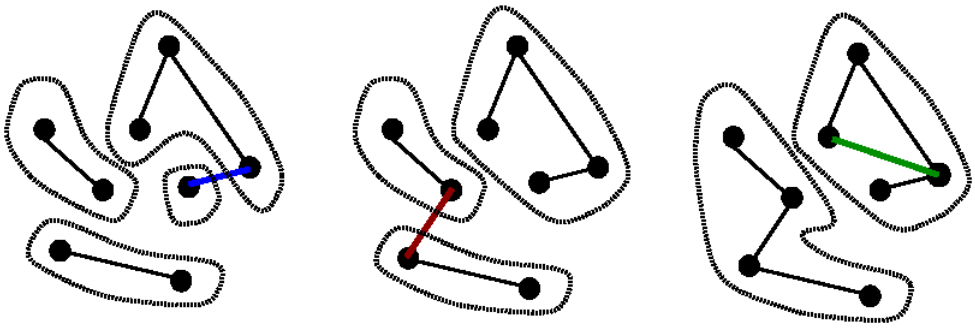


FIGURE 8. Évolution d'une structure d'ensembles disjoints.

## 5. Ensembles Disjoints

5.1. **Structure d'ensembles disjoints.** Les chances d'un graphe d'être connexe augmentent avec le nombre de ses arêtes. On souhaite réaliser une expérience numérique pour mieux comprendre la corrélation entre connexité d'un graphe et le nombre de ces arêtes, en fonction de son ordre. Pour l'ordre $n$, on part d'un graphe sans arête auquel on ajoute des arêtes aléatoires jusquà obtenir un graphe connexe. L'évolution des composantes connexes de la suite de graphes peut-être modélisée par la structure de donnée d'ensembles disjoints sur l'ensemble des $\{0, 1, \ldots, n-1\}$. Une structure d'ensembles disjoints sur des objets est une structure avancée munie de trois opérations caractéristiques ;

- `singleton(objet t)` : initialise l'ensemble réduit au singleton de $t$ ;
- `representant(disjoint x)` qui retourne le représentant de la classe de $x$.
- `union(disjoint x, y)` qui fusionne la classe de $x$ et la classe de $y$.

5.2. **Implantation par liste.** On implante la structure d'ensemble disjoint à l'aide d'une liste chaînée.

Dans cette version naïve, une succession de $m$ opérations dont $n$ opérations `singleton` peut coûter assez cher $\mathrm{O}(\ n + m^2\ )$. On peut y remédier par une heuristique simple et puissante. L'heuristique de union pondérée qui consiste à modifier les représentants du plus petit des ensembles. En pratique, il suffit d'ajouter un champ, maintenu à jour sur les représentants uniquement, pour stocker le nombre d'éléments des classes.

**Exercice 17.** *Implanter l'heuristique de l'union pondérée.*

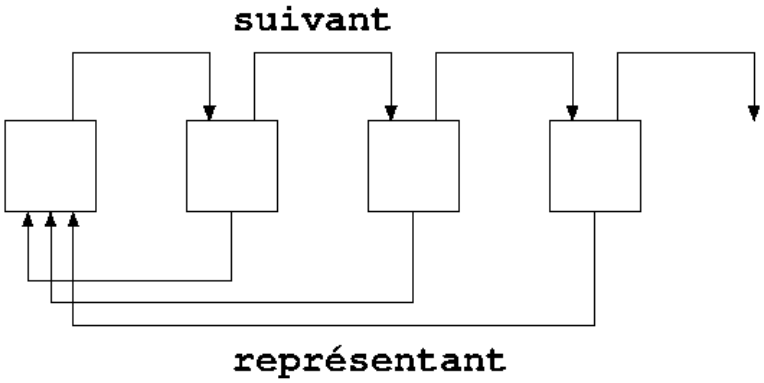



FIGURE 9. Représentation par liste chaînée.

```
1  typedef struct _edl_ {
2      struct _edl_ *rep;
3      struct _edl_ *svt;
4      int num;
5  } enrdisjoint , *disjoint ;
6
7  disjoint representant(disjoint x);
8
9  disjoint singleton(int v);
10
11 void reunion(disjoint x, disjoint y)
12 {
13     disjoint aux;
14     aux = x;
15     while (aux->next)
16       aux = aux->next;
17     aux = aux->next = y->rep;
18     while (aux) {
19         aux->rep = x->rep;
20         aux->svt = aux;
21     }
22 }
```

LISTING 14. implantation par liste chaînée.

**Proposition 5** (union pondérée)**.** *Le temps de calcul d'une succession de $m$ opérations dont $n$ opérations* `singleton` *est* $\mathrm{O}(\, m + n \log n \,)$.

*Démonstration.* □

5.3. **Forêt d'ensemble disjoint.** Dans la représentation par arbre, un objet d'une structure d'ensemble disjoint pointe uniquement dans la direction de son représentant.

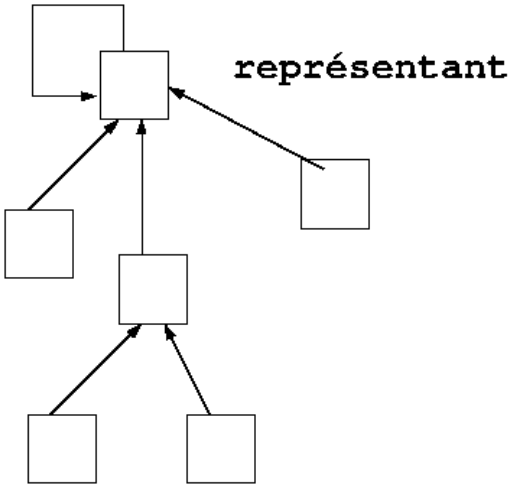



FIGURE 10. Représentation arbre.

La détermination d'un représentant est l'opération la plus coûteuse. Pour éviter des arbres trop profonds, lors d'une union, le représentant le moins haut est greffé sur le plus grand. C'est l'heuristique de l'union par rang. Au départ les singletons sont des arbres de rang 0. L'union de deux classes de même rang $r$ produit une classe de rang $r + 1$.

**Exercice 18.** *Implanter l'heuristique de l'union par rang.*

**Proposition 6** (union par rang)**.** *Le temps de calcul d'une succession de $m$ opérations dont $n$ opérations* `singleton` *est* $\mathrm{O}(\, m \log n \,)$.

*Démonstration.* Il suffit de vérifier que le nombre d'éléments d'une classe de rang $r$ est supérieur à $2^r$. □

Lors de la détermination d'un représentant, on peut changer le champ `rep` des noeuds visités pour qu'ils pointent directement vers leur chef de classe. Les chemins vers les représentants racourcissent au fur et à mesure des recherches de représentants. C'est l'heuristique de la compression de chemin, d'une redoutable efficacité !

**Exercice 19.** *Implanter l'heuristique de la compression de chemin.*

```
1  typedef struct _edl_ {
2      struct _edl_ *rep;
3      struct _edl_ *svt;
4      int num;
5  } enrdisjoint , *disjoint ;
6
7  disjoint representant(disjoint x);
8
9  disjoint singleton (int v);
10
11 void reunion(disjoint x, disjoint y)
12 {
13     disjoint aux;
14     aux = x;
15     while (aux->next)
16       aux = aux->next;
17     aux = aux->next = y->rep;
18     while (aux) {
19         aux->rep = x->rep;
20         aux->svt = aux;
21     }
22 }
```

LISTING 15. implantation par forêt d'arbre.

Le logarithme itéré est défini sur les nombre réels par la formule récursive :

$$\log^*(x) = \begin{cases} 0 & x \leq 0; \\ 1 + \log^*(\log x) & \text{x/gt0.} \end{cases}$$

**Exercice 20.** *Compléter la table :*

| $n$ | *1* | *2* | *4* | *8* | *16* | *32* |
|---|---|---|---|---|---|---|
| $2^n$ | *2* | *4* | *16* | | | |
| $\log^*(2^n)$ | *1* | *2* | *3* | | | |

**Proposition 7** (compression de chemin)**.** *Le temps de calcul d'une suite de $m$ opérations dont $n$ opérations* `singleton` *est* $\mathrm{O}(\, m \log^* n \,)$.

*Démonstration.* Hors programme. □

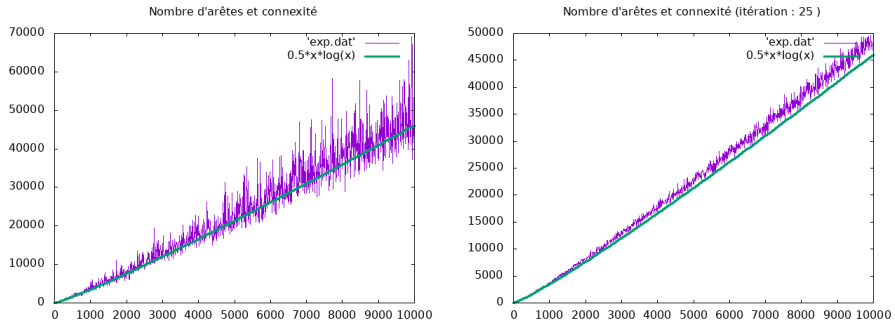



FIGURE 11. Nombre d'arêtes et connexité.

```
#!/bin/bash
if [ ! -f sigma.dat ] ; then
    for(( n=1; n < 1024; n++ )); do
        ./sigma.exe $n
    done > sigma.dat
fi
gnuplot <<< EOJ
   set term png
   set output 'sigma.png'
   plot 'sigma.dat', x*log(x)
EOJ
#$
```

LISTING 16. script à compléter

5.4. **Expérience Numérique.** Il s'agit de réaliser l'expérience numérique pour mettre en évidence une fonction de seuil $\sigma$ qui permette d'évaluer les chances pour un graphe d'ordre $n$ d'être connexe en considérant uniquement le nombre de ces arêtes. On procéde de manière empirique pour déterminer la valeur de $\sigma(n)$ en partant d'un graphe d'ordre $n$ dont tous les sommets sont isolés, on ajoute des arêtes aléatoirement jusqu'à obtenir un graphe connexe, $\sigma(n)$ correspond au nombre d'arêtes.

Dans une expérience numérique, il est souvent préférable de séparer les sources écrites en langage C qui calculent des données, des scripts qui exploitent les données. Ainsi, il sera possible d'utiliser le jeu de commandes offertes par le système d'exploitation pour mettre en forme

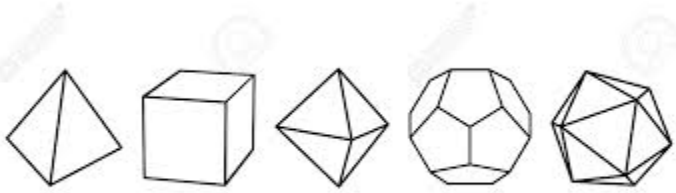


FIGURE 12. Les cinq solides de Platon : tétraèdre, hexaèdre, octoèdre, dodécaèdre et icosaèdre sont-ils Hamiltonien ?

le résultat d'une expérience numérique en évitant d'inutiles répétitions de calculs.

**Exercice 21** (exemple de script)**.** *Que fait le script plotcov.sh ?*

**Travaux-Pratiques 7** (scriptage empirique)**.** *Il s'agit de mettre en place une expérience numérique pour déterminer la fonction empirique $\sigma$ décrite plus haut.*

*(1) Implanter la structure d'ensembles disjoints dans des fichiers sources* `disjoint.[ch]` *pour la bibliothèque des graphes.*

*(2) Écrire une commande* `sigma.exe` *pour réaliser un calcul empirique de $\sigma(n)$. La commande lit $n$ sur la ligne de commande avant d'afficher $n$ et $\sigma(n)$ sur la même ligne, deux nombres séparés par un espace.*

*(3) Compléter le script* LIST. *(16) pour pour dessiner et identifier avec* `gnuplot` *le graphe de la fonction empirique.*

## 6. Graphe Hamiltonien

6.1. **Le voyage autour du monde.** En 1856, l'astronome-mathématicien irlandais W. Hamilton, célèbre pour sa découverte du corps des quaternions, introduit le problème de l'existence d'un chemin fermé suivant les arêtes d'un polyèdre passant une et une seule fois par tous les sommets. En particulier, dans le jeu du voyage autour du monde, schématisé par un dodécaédre régulier (12 faces pentagonales) dont les sommets sont des villes, il s'agit de trouver une route fermée tracée sur les arêtes qui passe une et une seule fois par chaque ville.

Le dodécaédre est un graphe planaire apparaît hamiltonien dans sa représentation planaire. Le jeu se corse quand on cherche à compter le nombre de cycles hamiltoniens.

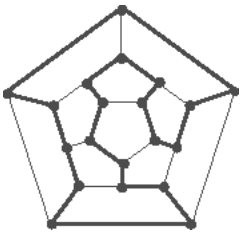


FIGURE 13. Une route autour du monde.

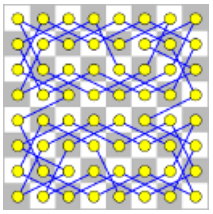


FIGURE 14

Notons que le contemporain de Philidor, Léonard Euler (encore lui) s'intéresse à des questions similaires. Dans "La Solution d'une question curieuse qui ne paraît soumise à aucune analyse", le mathématicien exibe la solution symétrique ci-contre. Il faut attendre une approche astucieuse de B. D. MacKay (1997) pour compter le nombre de tour de cavalier sur un échiquier 8x8 :

$$13267364410532 = 2 \times 2 \times 1879 \times 4507 \times 391661$$

**Exercice 22.** *Utiliser la relation d'Euler $f - m + n = 2$ pour dénombrer les nombres de faces, arêtes et sommets des cinq polyèdres de Platon.*

**Problème 4** (chemin Hamiltonien)**.** *Etant donné un graphe. Existe-t-il un chemin élémentaire passant par tous les sommets du graphe ?*

**Remarque 2.** *Dans la littérature, le graphe est dit hamiltonien quand il existe un cycle hamiltonien, semi-hamiltonien quand il existe un chemin hamiltonien non cyclique.*

### 6.2. **Deux conditions suffisantes.**

**Proposition 8** (G. A. . Dirac, 1952)**.** *Un graphe d'ordre $n$ dont les sommets sont de degré supérieur ou égal $\frac{n}{2}$ est hamiltonien.*

*Démonstration.* □

**Théorème 2** (Ø. Öre, 1961)**.** *Un graphe d'ordre $n$, dans lequel toute paire $\{x, y\}$ de sommets non adjacents vérifie :*

$$\deg(x) + \deg(y) \geq n,$$

*possède un cycle Hamiltonien.*

*Démonstration.* □

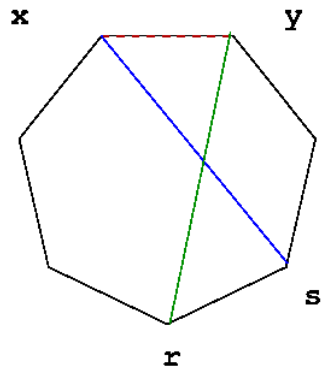



FIGURE 15

*Démonstration.* Par l'absurde, montrons qu'un graphe non hamiltonien ne peut pas vérifier les hypothèses du théorème. Quitte à ajouter des arêtes, on peut supposer le graphe non hamiltonien maximal. L'ajout d'une nouvelle arête $xy$ ($x \neq y$) produisant un graphe hamiltonien, le graphe est obligatoirement semi-hamiltonien : il existe un chemin $x \rightsquigarrow y$ hamiltonien, reliant les deux sommets. Un argument de dénombrement permet alors d'affirmer que sur ce chemin, il existe deux sommets adjacents $r$ et $s$ tel que

$$xs \in A \quad \text{et} \quad ry \in A.$$

d'où l'on tire un cycle hamiltonien, et une contradiction. □

.

Le problème sonne comme celui du chemin eulérien, l'objectif de cette section est de se convaincre qu'ils sont en fait de difficulté sans doute très différente.

6.3. **Code de Gray.** La distance de Hamming entre deux mots binaires de $m$ bits $x = (x_m \cdots x_2 x_1)$ et $y = (y_m \cdots y_2 y_1)$ compte le nombre de différence entre les composantes :

$$\mathrm{d_H}(x, y) = \sharp\{i \mid x_i \neq y_i\}.$$

L'hypercube de dimension $m$ est un graphe dont les sommets sont les mots de $m$-bits, deux sommets $x$ et $y$ sont adjacents si et seulement si $\mathrm{d_H}(x, y) = 1$.

**Exercice 23** (hypercube)**.** *Montrer que les hypercubes de dimension* 2, 3 *et* 4 *sont des graphes Hamiltonien. Montrer par induction sur la dimension qu'un hypercube est hamiltonien.*

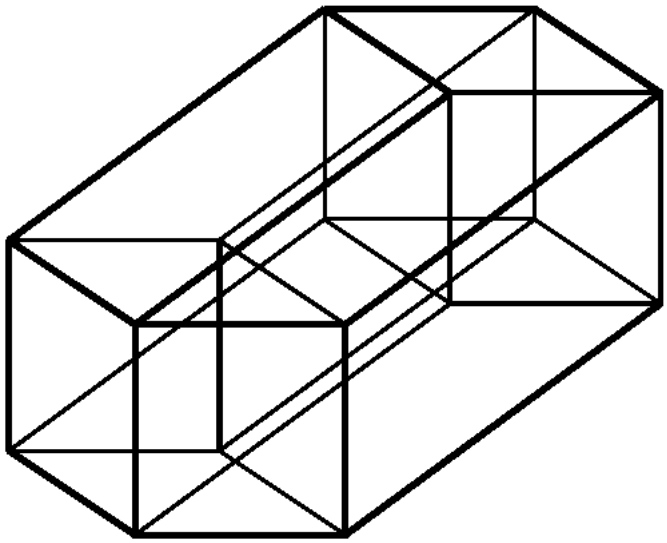


FIGURE 16
L'hypercube de dimension 4.

```
1  enumGray:
2  .LFB7:
3   .cfi_startproc
4   pushq %rbp
5   .cfi_def_cfa_offset 16
6   .cfi_offset 6, -16
7   movq %rsp, %rbp
8   .cfi_def_cfa_register 6
9   subq $32, %rsp
10  movl %edi, -20(%rbp)
11  movl -20(%rbp), %eax
12  movl $1, %edx
13  movl %eax, %ecx
14  sall %cl, %edx
15  movl %edx, %eax
16  movl %eax, -12(%rbp)
17  movl $0, -4(%rbp)
18  movl $1, -8(%rbp)
19  jmp .L5
20  .L6:
21  rep bsfl -8(%rbp), %eax
22  movl %eax, -16(%rbp)
23  movl -16(%rbp), %eax
24  movl $1, %edx
25  movl %eax, %ecx
26  sall %cl, %edx
27  movl %edx, %eax
28  xorl %eax, -4(%rbp)
29  movl -4(%rbp), %eax
30  movl -20(%rbp), %edx
31  movl %edx, %esi
32  movl %eax, %edi
33  call pbits
34  addl $1, -8(%rbp)
35  .L5:
36  movl -12(%rbp), %eax
37  cmpl %eax, -8(%rbp)
38  jb .L6
39  nop
40  nop
41  leave
42  .cfi_def_cfa 7, 8
43  ret
44  .cfi_endproc
```

```
45 .LFE7:
46 .size enumGray, .-
     enumGray
```

LISTING 17. La procédure `enumGray` assemblée par `gcc` option -S du compilateur.

**Exercice 24** (code de Gray)**.** *Le code* LIST. *(18) affiche un cycle Hamiltonien de l'hypercube de dimension m. Il est utilise la fonction* `ctz` *qui compte les zéros de poids faible d'un entier. Une opération sur la plupart des architectures. Retrouver son code opération dans le code mnémonique assembleur* LIST. *(17) de la fonction* `enumGray` *?*

## 7. Backtracking sur l'échiquier

Les algorithmes de backtracking parcourent l'arbre de décision d'un problème pour la recherche d'une solution. Rarement trés efficaces, ils permettent de résoudre les instances de petites tailles des problèmes difficiles de la théorie des graphe : cycle hamiltonien, clique maximale, couverture, ensemble stable etc...

7.1. **Les reines de l'échiquier.** De combien de façons peut-on placer $n$ dames ( reine ) sur un échiquier sans qu'aucune de ces dames n'en contrôle une autre ? Le problème n'est pas facile à appréhender du point de vue mathématiques. Il semblerait que le grand Gauss n'ait pas réussi à trouver la totalité des 92 solutions pour l'échiquier standard. En convenant de noter $\mathfrak{Q}(n)$ le nombre de solutions, personne ne sait prouver, par exemple, que pour $n$ assez grand $Q$ est croissante. De la célèbre encyclopédie des nombres en ligne OEIS de N. J. A. Sloane, nous tirons les première valeurs :

| $n$ | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|
| $\mathfrak{Q}(n)$ | 1 | 0 | 0 | 2 | 10 | 4 | 40 | 92 |

La monotonie de $Q$ est conjecturale mais notons le théorème de Pauls :

**Problème 5** (Pauls, E. 1874)**.** *Pour* $n > 3$, $Q(n) > 0$.

Le nombre de solutions est comparable au nombre de permutations. On rappelle que $log(n!)$ est equivalent à $nlog(n) - n$. Le graphique FIG. (18), suggère l'équivalence $log(Q(n)) \sim 0.5n \log(n)$.

```c
#include <stdio.h>
#include <stdlib.h>

void pbits( int x, int m )
{
    while ( m--) {
        printf (" %d ", x & 1);
        x >>=1;
    }
    putchar('\n');
}

void enumGray( int  dimen )
{ int limite  = 1 << dimen;
    unsigned int x = 0, v;
    unsigned int i = 1;
    while ( i  < limite  ){
        v = __builtin_ctz  ( i  );
            x ^= 1 << v;
        pbits(x , dimen);
        i  = i + 1;
    }
}

int main( int argc, char* argv[] )
{
    enumGray( 3  );
}
```

LISTING 18. Énumération de Gray des mots binaires

Un algorithme suffisant pour résoudre les instances de petites tailles est décrit par LIST. (19).

**Exercice 25.** *Compléter la fonction* `accept`

**Travaux-Pratiques 8** (profilage de l'arbre de récursion)**.** .

*(1) Implanter l'algorithme* `Reine`*.*

*(2) Faire des mesures de temps de calcul.*

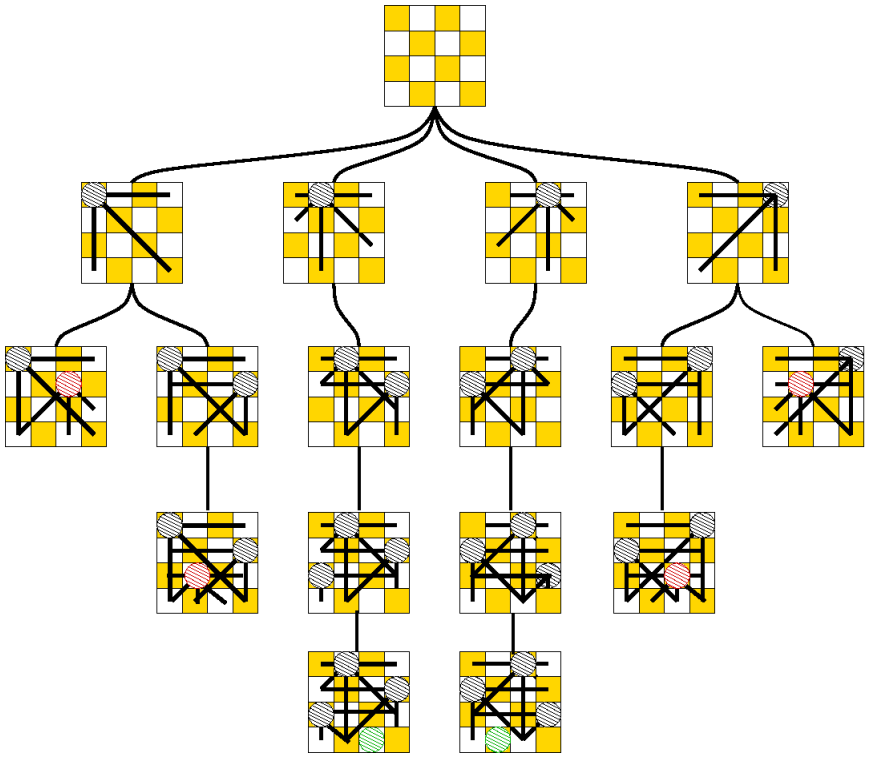


FIGURE 17. L'arbre de backtracking pour $Q(4) = 2$.

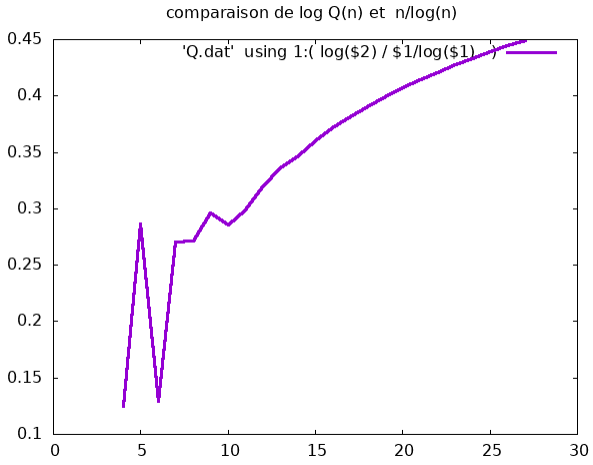



FIGURE 18. Comparaison avec la fonction $n \log(n)$.

*(3) Par extrapolation, quelles instances pourraient être traitées en 1 heure, 1 jour, 1 mois et 1 an ?*

```
placeDame( r : entier, pos : table )
si r > n alors
 traitement( pos )
 retourner
pour chaque colonne j
   si ( accept( r, j, pos ) ) alors
  pos[r] ← j
  place(r + 1, pos )
```

LISTING 19. Une première solution algorithmique

(4) *Modifier les codes pour dessiner l'histogramme du nombre de noeuds visités par l'algorithme en fonction des niveaux.*

(5) *Modifier les codes pour dessiner l'arbre de récursion.*

7.2. **Bit programming.** Pour manipuler des ensembles de petite taille, la représentation des ensembles par des mots de 64 bits donne des codes compacts. En langage C, les codes les plus courts sont souvent les plus efficaces ! Dans cette représentation, les entiers de 64 bits représentent des parties de $\{0, 1, \ldots, 63\}$. Il convient de définir le type :

```
1 typedef unsigned long long int ensemble;
```

L'entier 1777 représente l'ensemble $\{0, 4, 5, 6, 7, 9, 10\}$ car

```
1 bc <<< 'obase=2; 1777'
2 11011110001
```

Les entiers 0 et 1 représentent respectivement l'ensemble vide et le singleton de 0. La réunion de deux ensembles $X$ et $Y$ s'écrit $X|Y$, l'intersection devient $X\&Y$, et bien sûr, $\sim X$ le complémentaire de $X$. Comment écrire plus simplement :

(1) $$(X|Y)\&(\sim (X\&Y)) = ?$$

```
#include <stdio.h>
#include <stdlib.h>
typedef unsigned long long int ullong;
int  n, count = 0;
void qbit( int pos, int left , int right , int row )
{
ullong x, y;
if ( ! row ){ count++; return; }
row--;
left  <<=1; right>>=1;
x = pos & (~left) & (~right);
while ( x ) {
        y   = x;
        x  &= (x - 1 );
        y   = x ^ y;
        qbit( pos & (~y), left  | y, right | y , row );
}
}
int main( int argc, char* argv[] )
{
int n  = atoi( argv[1] );
ullong t = 1;
t = (t << n) - 1;
qbit ( t, 0, 0, n);
printf ("\nQ(%d)=%d\n", n, count );
return 0;
}
```

```
int mystibit( ensemble X )
{ int r;
  for ( r = 0; X & (X-1) ; r++);
  return r;
}
```

**Exercice 26** (job). *Une fois compris le résultat de la fonction* `mystibit`, *vous pourrez postuler sur des emplois de bit-programming !*

**Exercice 27** ( décryptage ). *Décrypter la source du* LIST. *(??).*

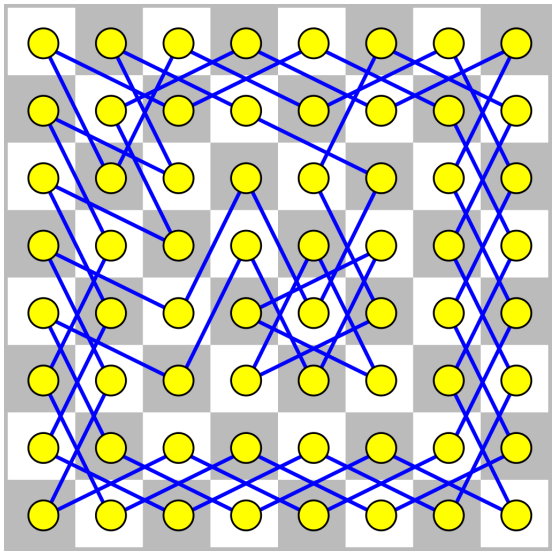


FIGURE 19. Un remarquable tour de cavalier dû à Euler.

```
hamiltonien( s : sommet, k : entier )
si k = n alors
 traitement()
 retourner
pour chaque voisin t de  s
   si libre[ t ] alors
 libre[t]  ← faux
 soluce[k] ← t
 hamiltonien( t, k+1 )
 libre[t]  ← vrai
```

**Exercice 28** (ctz). *Il est possible de flirter davantage avec la machine en utilisant les opérations* CTZ *qui présentent sur la plupart des architectures. Étudier la question avec le manuel de* `gcc`.

### 7.3. Cycle Hamiltonien.

**Exercice 29.** *Implanter* `hamilton(int s, int k, ullong libre )` *pour compter le nombre de chemin hamiltonien dans un graphe.*

**Travaux-Pratiques 9.** *Écrire un programme pour trouver un tour de cavalier sur l'échiquier n par n. L'heuristique de la contrainte maximale permet d'éviter l'enlisement de la recherche ! Dessiner les plus beaux cycles !*

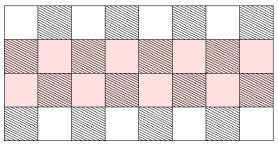


FIGURE 20. Le graphe du cavalier dans l'échiquier $4 \times n$ n'est pas hamiltonien.

```
maxstable( S, A )
  si A est vide alors retourner card(S)
  v ← un sommeton isole
  g = maxstable( S - v, A' );
  d = maxstable( S - voisin(v), A'' )
  retourner max( g, d + 1 )
```

**Exercice 30.** *Montrer qu'il n'existe pas de cycle Hamiltonien dans le graphe du cavalier d'un échiquier* $4 \times n$.

7.4. **Stable-Clique-Domination.** Dans un graphe, un ensemble stable est une partie de l'ensemble des sommets composées de sommets deux à deux non adjacents. On fera attention au vocabulaire : la recherche d'un stable maximal est une opération aisée alors que la recherche d'un sensemble stable de taille maximale est un problème d'optimisation difficile. Le problème de décision associé est NP-complet. Les petites instances pourront être résolues par l'algorithme LIST. (**??**).

**Exercice 31.** *Montrer que le temps de calcul de l'algorithme est exponentiel dans le pire des cas.*

**Exercice 32.** *Implanter* `int maxstab( graphe g)` *pour déterminer la taille maximale d'un stable.*

**Exercice 33.** *Une clique dans un graphe est un ensemble de sommets deux à deux adjacents. Comment réduire le problème de la recherche d'une clique à la recherche d'une partie stabel ?*

## 8. ARBRE COUVRANT MINIMAL

8.1. **Arborescence.** En théorie des graphes, un arbre est un graphe connexe acyclique c'est-à-dire sans cycle. Un ensemble d'arbre est une forêt...

**Théorème 3** (Big Equivalence on Tree)**.** *Soit* $G(S, A)$ *un graphe d'ordre* $n$ *à* $m$ *arètes. Les 6 assertions qui suivent sont équivalentes :*

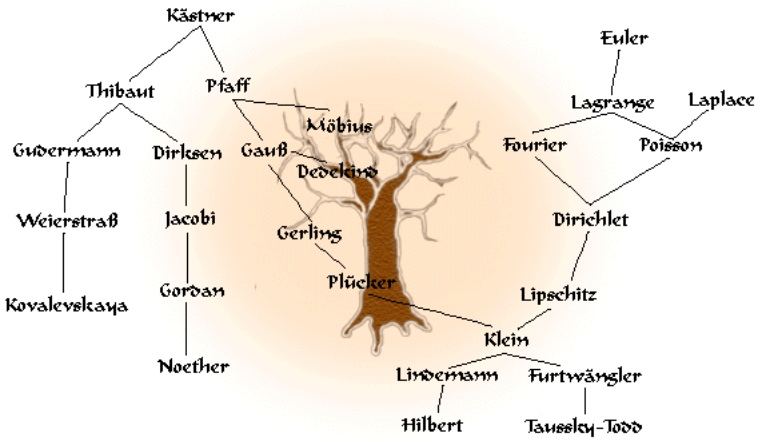



FIGURE 21. La décoration de l'arbre *mathematics genealogy project* inspire le respect mais ce n'est pas un arbre au sens de la théorie des graphes.

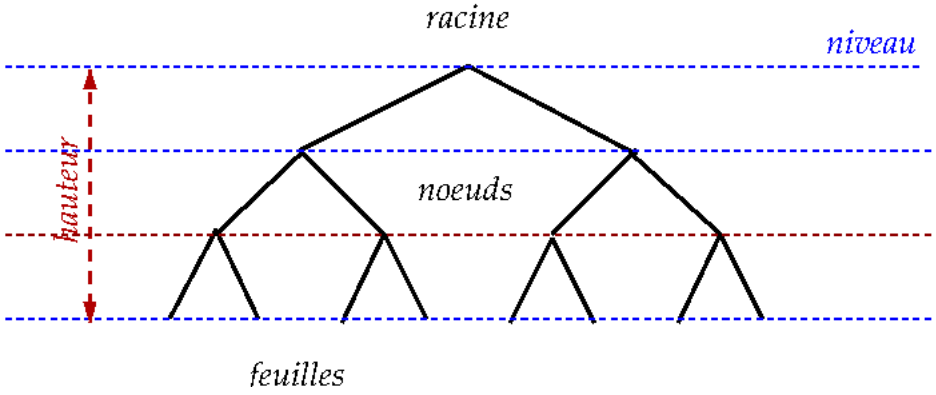



FIGURE 22. Un arbre binaire parfait de hauteur 4 à 15 noeuds organisés sur 4 niveaux et 8 feuilles au niveau 3. Conformément à l'usage algorithmique, la racine est en l'air sur le niveau 0 !

*(1) $G$ est connexe et acyclique ;*

*(2) $G$ est acyclique et $m = n - 1$ ;*

*(3) $G$ est connexe et $m = n - 1$ ;*

*(4) $G$ est acyclique et l'ajout d'une arête définit un et un seul cycle ;*

*(5) $G$ est connexe et la suppression d'une arête déconnecte le graphe ;*

*(6) deux sommets quelconques sont reliés par un et un seul chemin.*

```
PPV( t : tableau )
objet a, b, c, d
si ( t = 2 ) alors
    si t[0] > t[1] alors t[0]⟷t[1]
    retourner (t[0], t[1] )
    fsi
fsi
(a, b) ← PPV ( gauche t )
(c, d) ← PPV ( droite t )
si a < c alors
 retourner c < b ? (a,c) : (a,b)
sinon
 retourner a < d ? (c,a) : (c,d)

fsi
```

LISTING 20. Diviser pour régner

*Démonstration.* Les plus jolies preuves sont souvent les plus courtes ! Montrons par exemple, que (1) implique (3). On procède par induction sur le nombre $n$ de sommets. Le graphe possède obligatoirement une feuille, c'est-à-dire un sommet de degré 1. Le graphe induit par suppression de ce sommet est un arbre d'ordre $n-1$ et possède (induction) $n-2$. □

Un arbre couvrant d'un graphe est un graphe partiel qui est lui même un arbre.

**Proposition 9.** *Un graphe possède un arbre couvrant si et seulement s'il est connexe.*

**Exercice 34** (plus grandes valeurs)**.** *Tout le monde connaît l'algorithme de sélection qui procède en $n-1$ comparaisons pour déterminer la plus grande valeur d'un tableau de $n$ objets.*

*(1) Montrer qu'il est impossible de trouver le maximum en moins de $n-1$ comparaisons.*

*(2) Déduire qu'il faut au plus $2n-3$ comparaisons pour déterminer les deux plus grandes valeurs.*

*(3) L'algorithme `PGV( t )` fondé sur le principe algorithme "diviser pour régner" détermine les deux plus grandes valeurs de $t$. Combien de comparaisons sont réalisées pour une table de $n$ objets ?*

(4) *En s'inspirant des tournois de tennis, il n'est pas si difficile de se convaincre que* $c(n) := n - 1 + [\log_2 n] - 1$ *suffisent pour résoudre le problème des deux plus grandes valeurs. Comment ?*

*Le logicien Charles Dodgson (Lewis Carroll) a pointé la difficulté de cette question en affirmant par une démonstration incomplète qu'il est impossible de déterminer les 2 plus grandes valeurs en moins de* $c(n)$ *comparaisons.*

8.2. **Graphe pondéré.** Un graphe muni d'une application réelle définie sur ses arêtes est un graphe pondéré. Notons $w\colon A \to \mathbb{R}$ cette application, on dit que le graphe est pondéré par $w$. On définit alors le coût d'un chemin $\mu$ de longueur : $x_0$, $x_1$, …, $x_l$ par

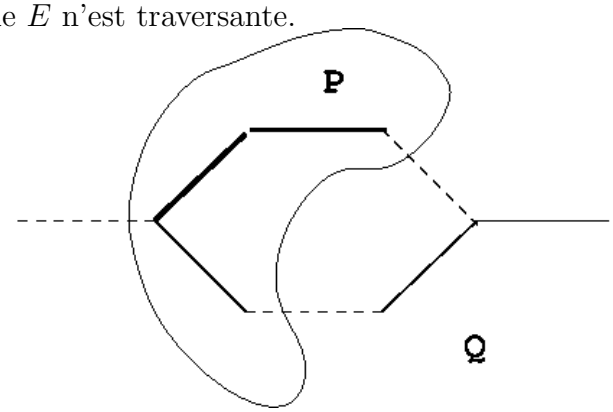
$$\mathrm{wt}(\mu) = \sum_{i=1}^{l} w(x_{i-1}x_i)$$

Plus généralement, le coût d'un graphe partiel est la somme des poids de ses arêtes.

**Problème 6** (arbre couvrant minimal)**.** *Étant donné un graphe pondéré, déterminer le coût minimal de ses arbres couvrants.*

Il s'agit d'un problème d'optimisation de complexité polynomiale, parmi les solutions classiques, les algorithmes de Prim, Kruskal sont détaillés dans cette note.

8.3. **Stratégie glouton.** Une coupure d'un graphe est une partition de l'ensemble des sommets en deux ensembles $P$ et $Q$. Une arête $xy$ est dite traversante si les deux sommets $x$ et $y$ ne sont ni dans $P$ ni dans $Q$. Une coupure respecte un ensemble d'arêtes $E$ si aucune des arêtes de $E$ n'est traversante.




**Lemme 2.** *(arête sûre) Soit* $E$ *une partie de l'ensemble des arêtes d'un arbre couvrant minimal* $Y$*. Si* $xy$ *est une arête traversante d'une*

```
KRUSKAL( G, w )
pour chaque sommet s
   singleton(s)
r ← 0
pour chaque arete xy par ordre croissant
     si representant(x) != representant(y)
 alors union(x, y )
     fsi
     r ← r + w(xy)
retourner r
```

LISTING 21. algorithme de Kruskal

*coupure du graphe respectant $E$ alors il existe un arbre couvrant minimal passant par $xy$ et toutes les arêtes de $E$.*

*Démonstration.* On considère une extension couvrante de l'arbre $Y$. Pour des raisons de connexité, il passe forcément par une arête traversante $ab$. L'ajout de l'arête $xy$ produit un et un seul cycle qui passe par $ab$, arête que l'on peut supprimer en conservant un arbre couvrant, qui reste de poids minimal. □

Les algorithmes de Prim et Kruskal utilisent une stratégie glouton fondée sur le lemme l'arête sûre pour contruire un arbre couvrant minimal d'un graphe.

8.4. **Algorithme de Kruskal.** L'algorithme de Kruskal (1956) s'appuie sur la structure d'ensembles disjoints pour construire un arbre couvrant minimal. Les arêtes de l'arbre sont découvertes de proche en proche. Les arêtes sont parcourues par poids croissant. Dans cet ordre, une arête $xy$ reliant deux classes distinctes est ajoutée à l'arbre couvrant. L'opération dominante est de trier les arêtes.

**Théorème 4.** *Le temps de calcul de l'algorithme de Kruskal sur un graphe connexe d'ordre $n$ et $m$ arêtes est* $\mathrm{O}(m \log m + m \log n)$.

*Démonstration.* L'algorithme commence par trier $m$ arêtes, d'où le terme $m \log m$. Le second terme correspond au calcul d'au plus $m$ de représentants d'une struture d'ensembles disjoints. □

8.5. **Algorithme de Jarnik-Prim.** L'algorithme de Prim s'appuie sur une file de priorité.

```
JARNIK-PRIM( G, w )
r ← 0
initialiser une file de priorite
p ← ordre( G )
tant que  p > 1
 s = prioritaire()
 r ← r + cle [ s ]
 pour chaque t voisin de s
       misajour( s, t)
 p ← p-1
retourner r
```

LISTING 22. algorithme de Jarnik-Prim

**Théorème 5.** *Le temps de calcul de l'algorithme de Prim sur un graphe connexe d'ordre $n$ et $m$ arêtes est* $\mathrm{O}(n \log n + m)$.

*Démonstration.* L'extraction d'un élément prioritaire, la mise à jour d'une file de priorité sont de complexité $\mathrm{O}(\log n)$. Le second terme correspond à la découverte des voisins. □

## 9. Coloriage des sommets

9.1. **Problème de coloration.** Une $k$-coloration des sommets d'un graphe est une application $f :: S \to \{1, 2, \ldots, k\}$ tel que :

$$\forall x, y \in S, \quad xy \in A \Longrightarrow f(x) \neq f(y).$$

On peut noter $\gamma_G(k)$ le nombre de colorations. Le plus petit entier $k$ tel que $\gamma_G(k) > 0$ est le nombre chromatique du graphe $G$, souvent noté $\chi_G$.

Il n'est pas difficile de voir qu'un arbre est 2-colorable. Le nombre chromatique du triangle est 3, celui du cycle de longueur 4 est 2. Plus généralement,

**Proposition 10** (graphe bicolore)**.** *Un graphe est bicolore si et seulement s'il ne possède pas de cycle de longueur impair.*

*Démonstration.* Un bon exercice ! □

**Problème 7** ($K$-coloriage)**.** *Etant donné un graphe, une constante $K > 2$. Le graphe est-il $K$-coloriable.*

Pour $K > 2$, le problème du $K$-coloriage est NP-complet.

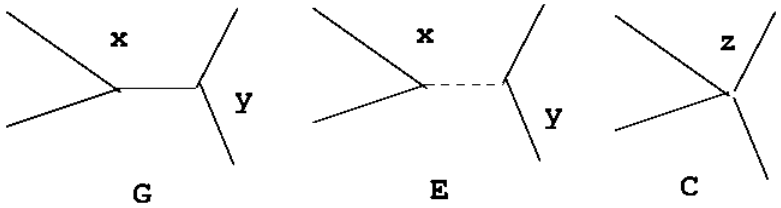



FIGURE 23. La transformation de Birkhoff. De gauche à droite, les graphe $G$, $E$ et $C$.

9.2. **Coloriage glouton.**

**Théorème 6** (Majoration)**.** *Pour tout graphe $G$, $\chi_G \leq \Delta_G + 1$.*

*Démonstration.* On construit une $k$-coloration avec $k \leq \Delta_G + 1$ en colorant les sommets dans un ordre arbitraire ave la plus petite couleur disponible. □

9.3. **Polynôme chromatique.**

**Théorème 7** (Birkhoff)**.** *L'application $k \mapsto \gamma_G(k)$ est polynomiale de degré égale à l'ordre du graphe.*

*Démonstration.* On raisonne par induction. Partant d'une arête $xy$, on considère deux graphes. Le graphe $E$ obtenu par suppression de l'arête $xy$, et le graphe $C$ obtenu par contraction de l'arête $xy$ en un sommet $z$. Toutes les $k$-colorations de $G$ sont des $k$-colorations de $E$. Les colorations de $E$ qui ne sont pas des colorations de $G$ s'identifient aux colorations de $C$ :

$$\gamma_G(t) = \gamma_E(t) - \gamma_C(t). \tag{2}$$

On remarque que pour un graphe sans arête d'ordre $n$, le nombre de $k$-colorations est $k^n$. □

**Exercice 35.** *Le polynôme chromatique d'un graphe d'ordre $n$ est unitaire de coefficient constant nul, à coefficient entiers. Il se décompose*

$$\gamma_G(t) = t(t-1)\cdots(t-k+1)P(t)$$

*où $P$ est un polynôme sans racines entières et $k$ le nombre chromatique de $G$.*

**Exercice 36.** *Déterminer le polynôme chromatique d'un arbre d'ordre $n$, d'un cycle d'ordre $n$.*

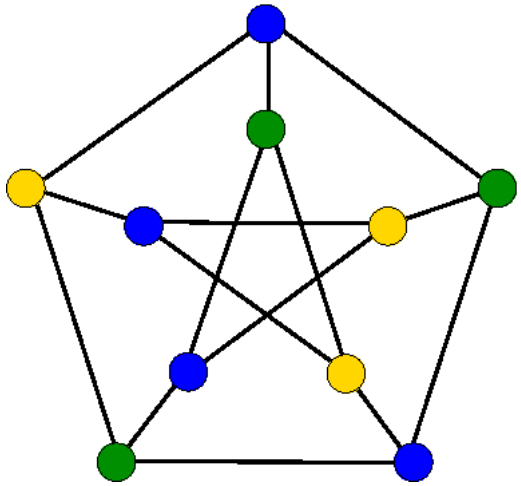


FIGURE 24. Le polynôme chromatique du graphe de Petersen est : $t^{10}-15t^9+105t^8-455t^7+1353t^6-2861t^5+4275t^4-4305t^3+2606t^2-704t$

```
coloriage( s, k )
si s == nbs  alors
   compteur += 1
   retourner
fsi
pour chaque couleur c < k
   si acceptable( s, c ) alors
      clr[s] ← c
        coloriage( s + 1 )
      clr[s] ← 0
   fsi
```

9.4. **Coloriage.** L'algorithme de backtracking pour colorier les $n$ sommets d'un graphe possède exactement la même structure que l'algorithme utilisé pour résoudre le problème des reines. Pour colorier les sommets avec $k$ couleurs, on commence par marquer les sommets avec la couleur nulle, ils sont transparents. Ensuite, les sommets sont coloriés par ordre croissant. La fonction `accept(s,c)` vérifie s'il est possible de colorier le sommet $s$ avec la couleur $c$.

**Exercice 37** (acceptable)**.** *Compléter l'algorithme de coloriage.*

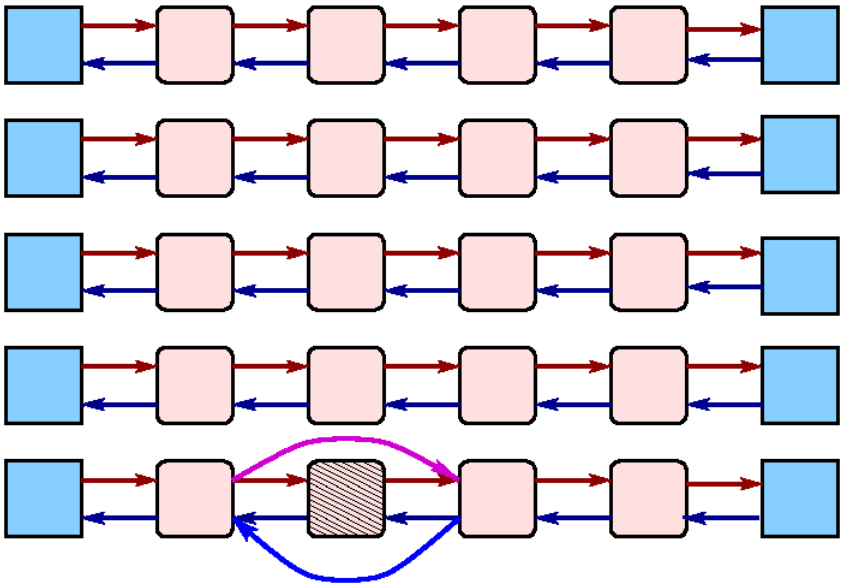


FIGURE 25. Un réseau de liens dansant pour le coloriage du graphe maison.

**Travaux-Pratiques 10.** *On peut utiliser des piles d'ensembles pour suivre l'évolution des contraintes.*

*(1) Implanter l'algorithme du calcul des colorations d'un graphe.*

*(2) Vérifier que le polynôme chromatique de la maison à 5 sommets est :*

$$t^5 - 8t^4 + 23t^3 - 28t^2 + 12t.$$

## 9.5. lien dansant.

**Exercice 38.** *Comment coder simplement l'initiatisation du réseau de liens ?*

**Travaux-Pratiques 11.** *Utiliser la technique des liens dansants sur une liste doublement chaînée pour décider k-colorabilité d'un graphe.*

## RÉFÉRENCES

[1] Claude Berge. *Graphes*. $\mu_B$. Dunod, Paris, third edition, 1983.

[2] T. H. Cormen, C. Leiserson, R. Rivest, and X. Cazin. *Introduction à l'algorithmique*. Science informatique. Dunod, 1994.

[3] Lucas Édouard. Récréations mathématiques—tome 1—les traversées. les ponts. les labyrinthes, 1882. http://gallica.bnf.fr.

[4] Noam Elkies and Richard Stanley. The mathematical knight, 200x. http://www-math.mit.edu/~rstan/papers/knight.pdf.

```
1  typedef struct _dl_ {
2      struct _dl_ *prd, *svt;
3      int clr ;
4      int hit ;
5  } enrdl, *dl;
6
7  enrdl  ** infos ;
8
9  void initdl ( int n, int k );
10 void refroidir ( int s, int k );
11 void rechauffer( int s, int k );
```

LISTING 23. structure pour les liens dansants


[5] Donald E. Knuth. *Art of Computer Programming, Volume 1 : Fundamental Algorithms (3rd Edition) (Art of Computer Programming Volume 1)*. Addison-Wesley Professional, 3 edition, November 1997.

[6] Donald E. Knuth. *Art of Computer Programming, Volume 2 : Seminumerical Algorithms (3rd Edition) (Art of Computer Programming Volume 2)*. Addison-Wesley Professional, 3 edition, November 1997.

[7] Donald E. Knuth. *Art of Computer Programming, Volume 3 : Sorting and Searching (3rd Edition) (Art of Computer Programming Volume 3)*. Addison-Wesley Professional, 3 edition, November 1997.

[8] Donaldi E. Knuth. Dancing links, 2000. http://arxiv.org/abs/cs/0011047.

[9] Philippe Langevin. Le tournoi de nimbad à euphoria, 19xy. http://langevin.univ-tln.fr/problemes/NIMBAD/marienbad.html.

[10] Philippe Langevin. Une brève histoire des nombres, 2003. http://langevin.univ-tln.fr/notes/rsa/rsa.pdf.

[11] Harold. F. Mattson. *Discrete Mathematics with Applications*. Wiley international editions. Wiley, 1993.

[12] Herbert J. Ryser and Paul Camion. *Méthodes Combinatoires*. Monographie Dunod. Dunod, 1969.

[13] Robert Sedgewick. Permutation generation methods, x. https://www.cs.princeton.edu/~rs/talks/perms.pdf.

[14] Robert Sedgewick and Jean-Michel Moreau. *Algorithmes en langage C*. I.I.A. Informatique intelligence artificielle. Dunod, 1991.

[15] Neil. J. A. Sloane. On-line encyclopedia of integer sequences, 1964. https://oeis.org/?language=french.

[16] Peter Wegner. A technique for counting ones in a binary computer. *Communications of the ACM*, 3(5), 1960.

[17] Herbert S. Wilf. *Algorithms and complexity*. Prentice-Hall international editions. Prentice-Hall, 1986.

[18] Jacques Wolfmann and Michel Las Vergnas. Matériaux de mathématiques discrètes pour le musée de la villette, 19xy.