THÉORIE ET ALGORITHMIQUE DES GRAPHES
LICENCE INFORMATIQUE

PHILIPPE LANGEVIN, IMATH, UNIVERSITÉ DE TOULON.

topo.tex	2020−11−30 14:50:30.265048877
planaire.tex	2020−11−24 07:58:40.809494050
coloriage.tex	2020−11−22 10:52:21.776549197
tag.tex	2020−11−22 10:15:48.322407589
disjoint.tex	2020−11−22 10:08:37.855317884
backtrack.tex	2020−11−22 09:35:44.726544762
connexe.tex	2020−11−19 23:10:47.635457249
acm.tex	2020−11−17 09:08:43.958209612
hamilton.tex	2020−11−11 16:14:51.584717928
implantation.tex	2020−11−07 13:30:19.290658198
euler.tex	2020−11−07 11:32:59.908852232
macros.tex	2020−11−03 09:24:31.616317745
reduction.tex	2020−10−29 08:34:45.168910063
np.tex	2020−10−29 08:34:45.159909984
rush.tex	2020−10−27 09:39:36.139463151

Date: dernière compilation :January 25, 2021.

Contents

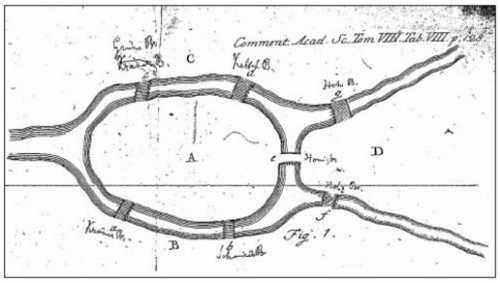


FIGURE 1. Peut-on faire une promenade passant une et une seule fois par chacun des sept ponts de la ville de Koenigsberg ?

1. Chemin eulérien

La théorie des graphes s'est développée au cours du XX^{e}, dans la note [16], 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}. 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.

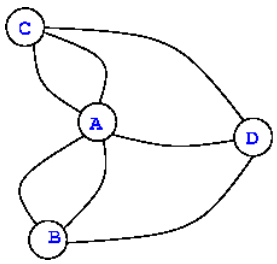


FIGURE 2. 2-graphe.

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

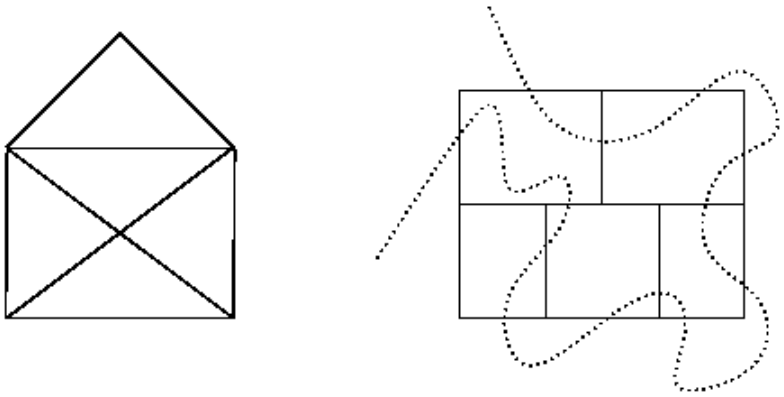


FIGURE 3. Peut-on tracer la petite maison d'un trait de crayon sans passer deux fois par le même segment? Peut-on tracer une courbe qui coupe une et une seule fois tous les segments de la figure?

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. 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 !

Proof. 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 [10] s'applique sans difficulté et donne :

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

□

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

Exercice 2. *De tout parcours, on peut extraire un parcours élémentaire ayant les mêmes extrémités. Détaillez cette affirmation !*

Exercice 3. *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 connexe est eulérien si et seulement si le nombre de sommets impairs est nul. Il est semi-eulérien si seulement si il possède exactement deux sommets impairs.*

Si un chemin passe par toutes les arêtes d'un sommet, ce sommet est une extrémité si et seulement son degré est impair.

On procède par induction sur le nombre d'arêtes.

Supposons qu'il existe deux sommets impairs a et b. Il existe un chemin μ qui relie a et b. On construit le graphe partiel par suppression des arêtes de ce chemin. Les composantes connexes de ce graphe partiel

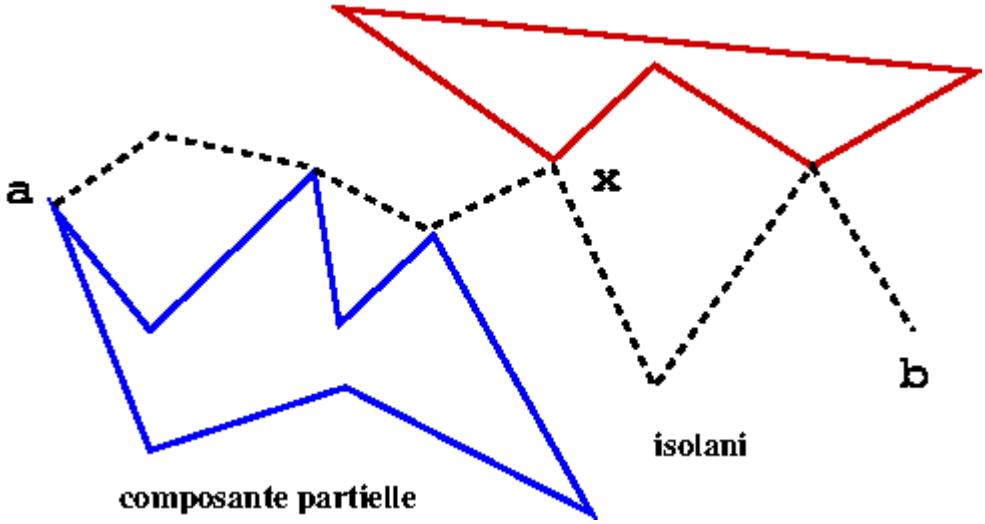


FIGURE 4. Les composantes connexes du graphe partiel qui ne sont pas des singletons forment des cycles eulériens.

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 μ. On obtient un chemin eulérien en remplaçant un représentant x de μ par le cycle eulérien μ_x du graphe partiel.

Exercice 4. *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.*

Proof. □

2. Mise en oeuvre

2.1. **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 2
1 : 0 2 3 4
2 : 0 1 3 4
4 : 1 2 3
```

2.2. **pile de sommets.** Nous envisageons de construire une bibliothèque en langage C pour coder des expériences numériques sur les graphes. FIG. (5) est un modèle d'organisation des fichiers dans les répertoires.

Les n sommets d'un graphe d'ordre n seront systématiquemnt les entiers de 0 à $n-1$.

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

Pratique 1 (pile de sommet)**.** *Implanter les piles sur des listes de sommets.*

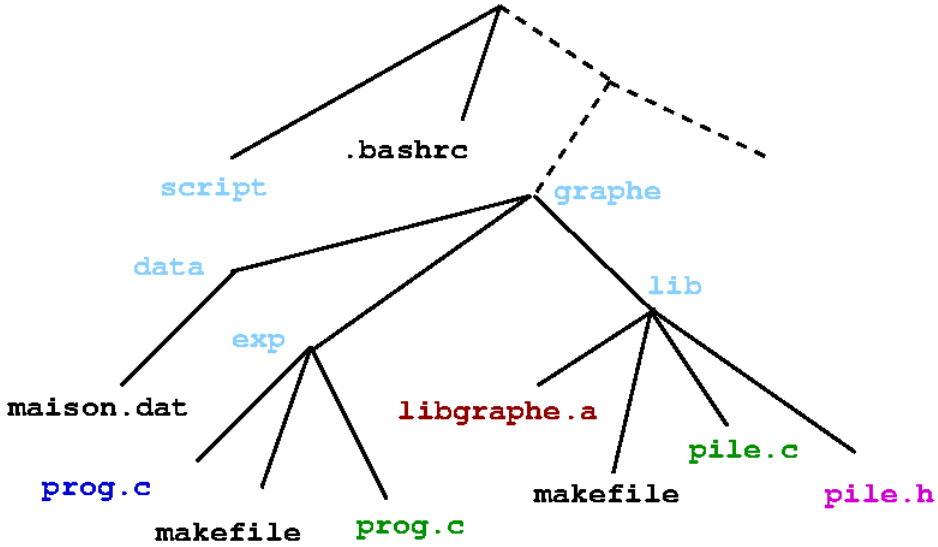


FIGURE 5. Hiérarchie des fichiers.

```
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, pile *p );
9  int    depiler (  pile *p );
10 #endif
```

LISTING 1. Structure de pile basée sur des listes chainées

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

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

Exercice 5. *Coder les fonctions* `graphe initGraphe( int n )`, `void libGraphe( graphe g)` *et* `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

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

LISTING 2. Un premier makefile

```
1  typedef struct {
2      int       **mat;
3      int        nbs;
4          liste     lst ;
5  } graphe;
6
7  graphe initGraphe( int nbs );
8  graphe lireGraphe( FILE *src );
9  void freeGraphe( graphe G );
10 int degre( int s,  graphe G );
```

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`.

Exercice 6. *Implanter une fonction* `graphe lireGraphe(FILE *src)` *pour analyser un graphe au format décrit dans (2.1) en vous appuyant sur l'automate de la figure* FIG. *(6) et l'analyseur lexical* LIST. *(4).*

```
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 3. Le service minimum en matière d'entrée/sortie!

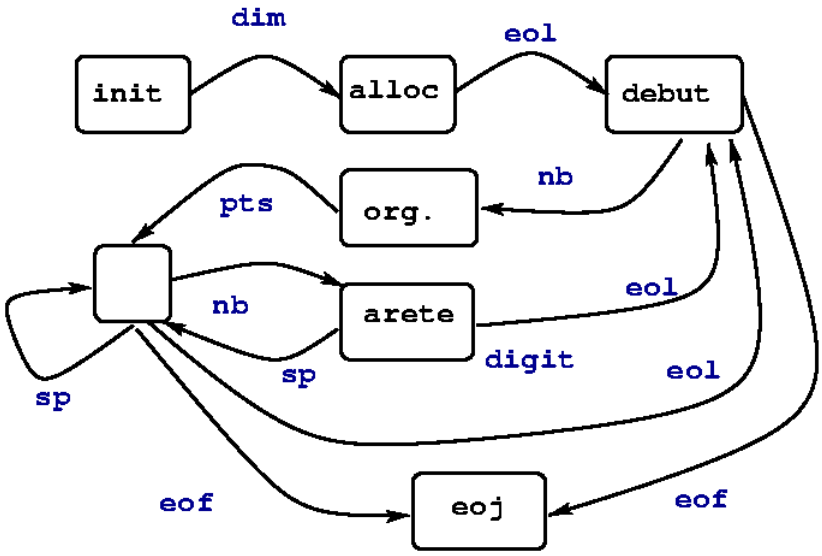


FIGURE 6. Automate pour la lecture d'un graphe au format xyz.

Pratique 2 (format `dot`, lancer une commande)**.** *La commande* `dot` *(et ses cousines) du projet graphiz permet de dessiner des beaux graphes à partir d'une source texte. Implanter une procédure* `void dotify(graphe G)` *qui affiche une image du graphe G via un appel de la fonction* `system` *de la* `glibc`*.*

```
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 4. Un exemple d'analyseur lexical.

2.5. **bibliothèque.** En pratique, il convient de réaliser une bibliothèque. Les sources des procédures et fonctions destinées à être utilisées dans des exprériences numériques sont déposées dans un répertoire. Elles sont compilées en des fichiers objets et intégrées dans une bibliothèque.

N'hésitez pas à vous faciliter la tâche en ajoutant des `alias` dans votre fichier `.bashrc`:

2.6. **Expérience numérique.** Les sources des expériences numériques sont déposées dans un répertoire séparé. Les chemins passés en option permettront au compilateur `gcc` retrouver les fichiers de définition (`.h`) et la bibliothèque `libgraphe.a`.

```
SRC=$(wildcard *.c)
OBJ=$(SRC:.c=.o)
CFLAGS = -Wall -g
ifeq ($(shell hostname), imath02.univ-tln.fr)
    CFLAGS = -O2
endif

libgraphe.a : $(OBJ)
      ar -cr libgraphe.a *.o
      #nm libgraphe.a
%.o:%.c
      gcc $(CFLAGS) -c $*.c
joli :
      indent -kr *.c
clean:
      rm -f *~ *.o
proper:
      rm -f *.a
```

LISTING 5. La bibliothèque statique `libgraphe.a`

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

Exercice 7. *Coder la commande* `eulerien.exe` *qui affiche un chemin eulérien du graphe correspondant au nom de fichier passé en argument.*

Pratique 3 (commande `/usr/bin/time`)**.** *Réaliser une exprérience numérique pour déterminer un cycle euliérien d'un graphe aléatoire d'ordre n dont tous les sommets sont pairs. Utiliser la commande externe* `time` *pour déterminer la forme du temps de calcul.*

3. Composante Connexe

3.1. **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 graphe est dit connexe quand il possède une seule composante connexe.

```
1  SHELL=/bin/bash
2  OPT=-Wall -g
3  LIB=-L$(GRAPHE)/lib
4  INC=-I$(GRAPHE)/lib
5  all   :  eulerien.exe
6
7  eulerien.exe :  eulerien.c
8        gcc $(OPT) $(INC) $(LIB)  $< -o $@ -lgraphe
9
10 eulerien :  eulerien.exe
11       ./eulerien.exe $(GRAPHE)/data/eulerien.dat
```

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

Exercice 9 (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.*

3.2. **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 6. Marquage des sommets par un parcours récursif en profondeur

Exercice 10 (élimination de la récursivité)**.** *Comment utiliser explicitement une structure de de pile pour éliminer la récursivité de* LIST. *(6).*

Exercice 11. *Ecrire un algorithme pour dénombrer le nombre de composantes connexes d'un graphe.*

3.3. **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 :*

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

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

Exercice 12. *Ecrire une fonction* `void freeGraphe( graphe g)` *qui libère la mémoire allouée au graphe passé en argument.*

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

```
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 7. liste 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 :*

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

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

Proof. 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.*

3.5. **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***.*

Proof. Pour décider si un graphe est eulérien, il suffit de vérifier d'une part la connexité du graphe, et d'autre part, que tous les sommets sont de degrés pairs. Les deux opérations sont réalisables en temps quadratique. □

4. Graphe Hamiltonien

4.1. **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.

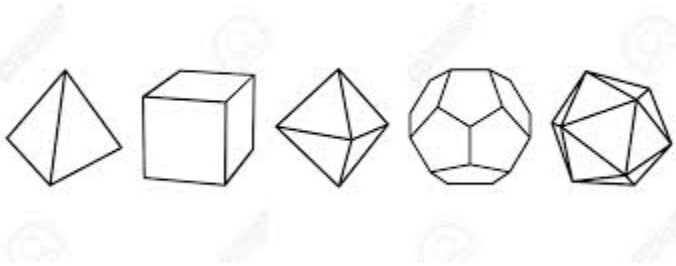


FIGURE 7. Les cinq solides de Platon : tétraèdre, hexaèdre, octoèdre, dodécaèdre et icosaèdre sont-ils Hamiltonien?

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.

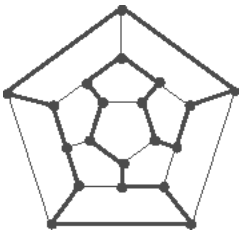


FIGURE 8. Une route autour du monde.

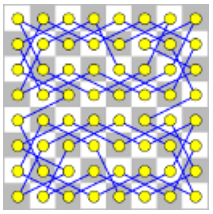


FIGURE 9

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 16. *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 ?*

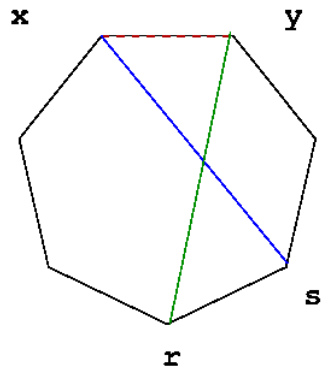


FIGURE 10

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.*

4.2. **Deux conditions suffisantes.**

Proposition 5 (G. A. . Dirac, 1952)**.** *Un graphe d'ordre n dont les sommets sont de degré supérieur ou égal $\frac{n}{2}$ est hamiltonien.*

Proof. □

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.

Proof. □

Proof. 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. □

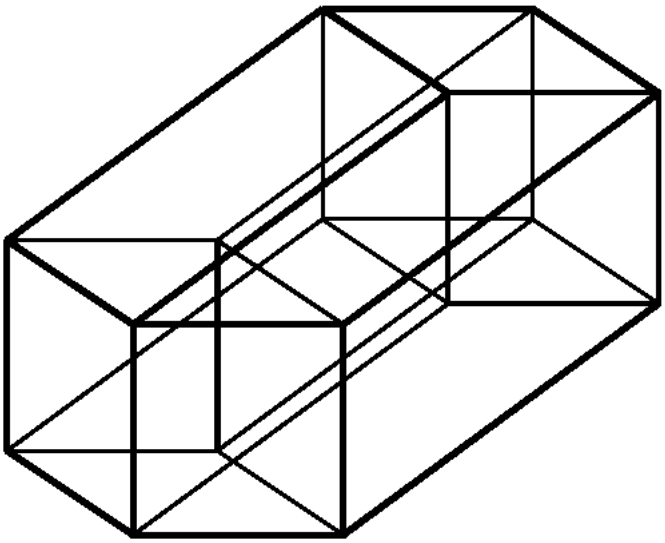


FIGURE 11
L'hypercube de dimension 4.

.

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.

4.3. **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 17 (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.*

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

Exercice 18 (code de Gray)**.** *Le code* LIST. *(9) 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. *(8) de la fonction* `enumGray` *?*

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;

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  void pbits( int x, int m )
5  {
6      while ( m--) {
7          printf (" %d ", x & 1);
8          x >>=1;
9      }
10     putchar('\n');
11 }
12
13 void enumGray( int  dimen )
14 { int limite  = 1 << dimen;
15     unsigned int x = 0, v;
16     unsigned int i = 1;
17     while ( i  < limite  ){
18         v = __builtin_ctz  ( i  );
19             x ^= 1 << v;
20         pbits(x , dimen);
21         i  = i + 1;
22     }
23 }
24
25 int main( int argc, char* argv[] )
26 {
27     enumGray( 3  );
28 }
```

LISTING 9. Énumération de Gray des mots binaires

- `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.

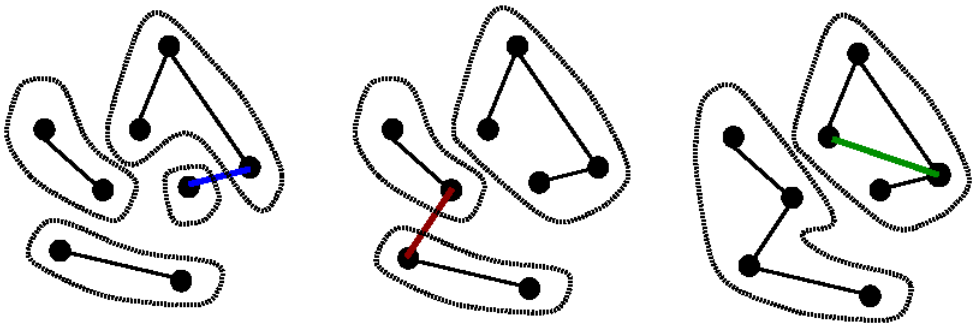


FIGURE 12. Évolution d'une structure d'ensembles disjoints.

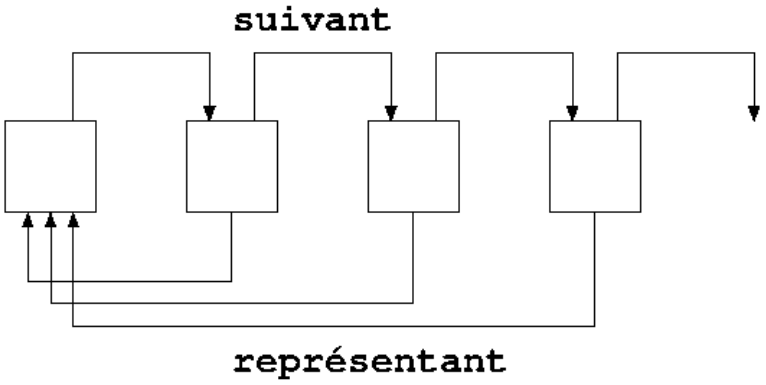


FIGURE 13. Représentation par liste chaînée.

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 19. *Implanter l'heuristique de l'union pondérée.*

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

Proof. □

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.

La détermination d'un représentant est l'opération la plus coûteuse. Pour éviter des arbres trop profonds, lors d'une union, le représentant

```
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 10. implantation par liste chaînée.

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 20. *Implanter l'heuristique de l'union par rang.*

Proposition 7 (union par rang)**.** *Le temps de calcul d'une succession de m opérations dont n opérations* `singleton` *est* $\mathrm{O}(\, m \log n \,)$.

Proof. 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 !

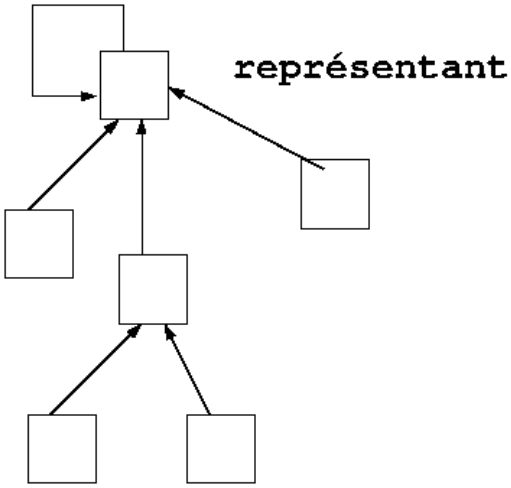


FIGURE 14. Représentation arbre.

Exercice 21. *Implanter l'heuristique de la compression de chemin.*

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 22. *Compléter la table :*

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

Proposition 8 (compression de chemin)**.** *Le temps de calcul d'une suite de m opérations dont n opérations* `singleton` *est* $\mathrm{O}(\, m \log^* n \,)$.

Proof. Hors programme. □

5.4. **Expérience Numérique.** Il s'agit de réaliser l'expérience numérique pour mettre en évidence une fonction de seuil σ 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

```
typedef struct _edt_ {
    struct _edt_ *rep;
    int rang;
    int num;
} enrdisjoint , *disjoint ;

disjoint representant(disjoint x){
while ( x->rep != x ) x = x -> rep;
return x;
}

disjoint singleton (int v);

void reunion(disjoint x, disjoint y)
{
    x = representant(x);
    y = representant(y);
    x->rep = y->rep

}
```

LISTING 11. implantation par forêt d'arbre.

```
#!/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 12. script à compléter

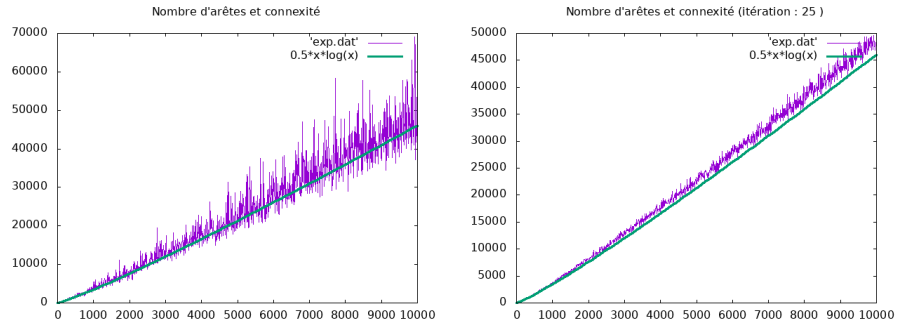


FIGURE 15. Nombre d'arêtes et connexité.

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

Exercice 23 (exemple de script)**.** *Que fait le script plotcov.sh ?*

Pratique 4 (scriptage empirique)**.** *Il s'agit de mettre en place une expérience numérique pour déterminer la fonction empirique σ 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. *(12) pour pour dessiner et identifier avec* `gnuplot` *le graphe de la fonction empirique.*

6. Arbre couvrant minimal

6.1. **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:*

(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.*

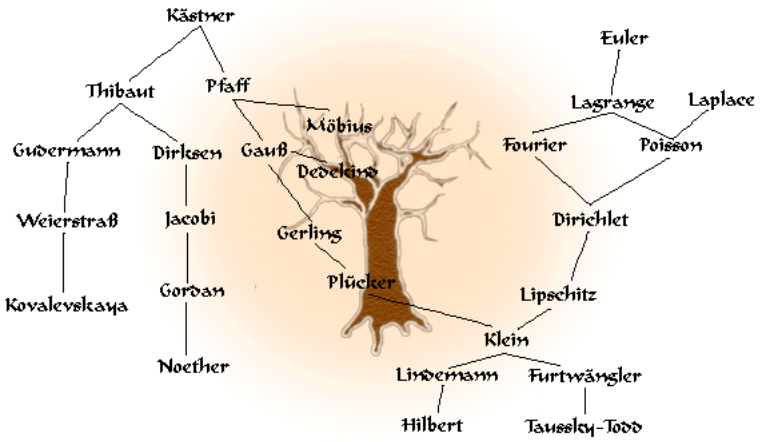


FIGURE 16. La décoration de l'arbre *mathematics genealogy project* inspire le respect mais ce n'est pas un arbre au sens de la théorie des graphes.

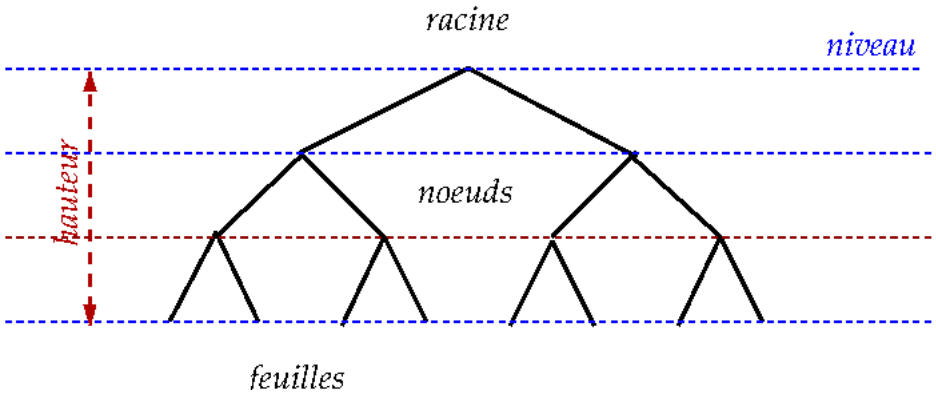


FIGURE 17. Un arbre binaire parfait de hauteur 4 à 15 noeuds organisés sur 4 niveaux et 8 feuilles au niveau 3. Conformément à l'usage algorithmique, la racine est en l'air sur le niveau 0 !

Proof. Les plus jolies preuves sont souvent les plus courtes ! □

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.*

```
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 13. Diviser pour régner

Exercice 24 (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.

6.2. **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 μ de longueur : x_0, x_1, ..., x_l par

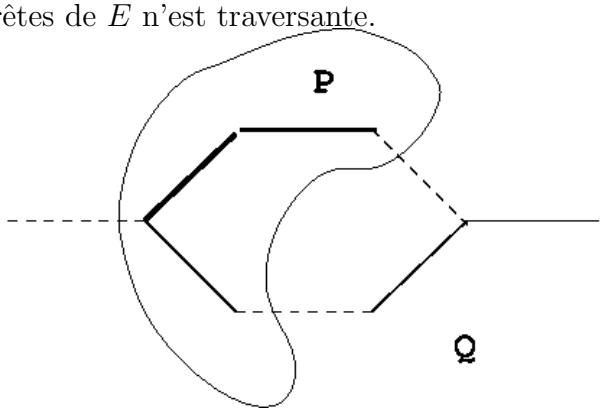
$$\mathrm{wt}(\mu) = \sum_{i=1}^{l} w(x_{i-1}x_i)$$

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

Problème 5 (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.

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

Proof. 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.

```
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 14. algorithme de Kruskal

6.4. **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* $\mathrm{O}(m \log m + m \log n)$.

Proof. 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. □

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

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

Proof. 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. □

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...

```
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 15. algorithme de Jarnik-Prim

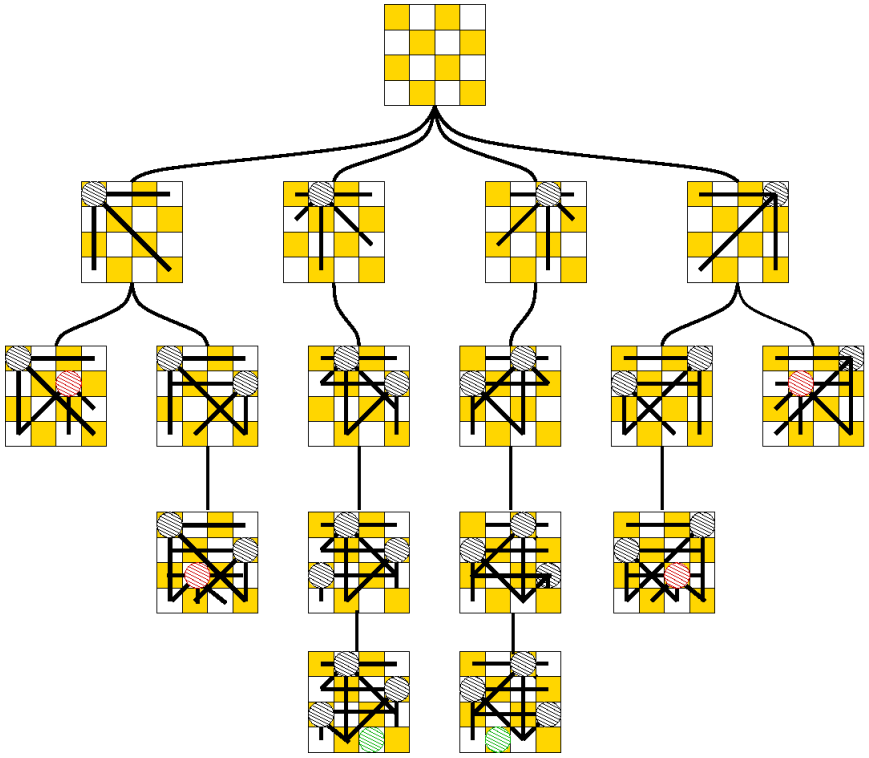


FIGURE 18. L'arbre de backtracking pour $Q(4) = 2$.

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

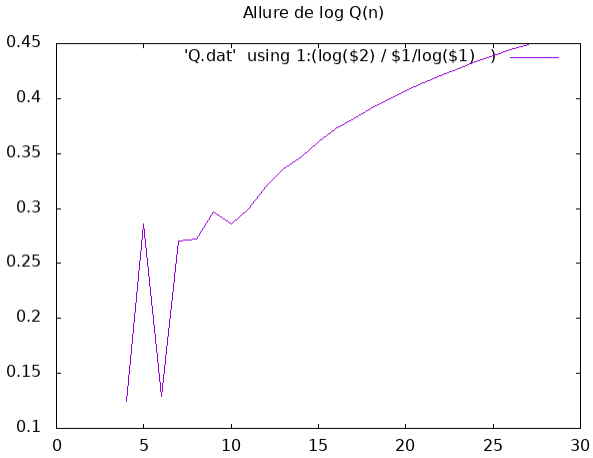


FIGURE 19. Comparaison avec la fonction $n\log(n)$.

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 6 (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. (19), suggère l'équivalence $log(Q(n)) \sim 0.5n\log(n)$.

Un algorithme suffisant pour résoudre les instances de petites tailles est décrit par LIST. (16).

Exercice 25. *Compléter la fonction* `accept`

Pratique 5 (profilage de l'arbre de récursion)**.** .

(1) *Implanter l'algorithme* `Reine`*.*
(2) *Faire des mesures de temps de calcul.*
(3) *Par extrapolation, quelles instances pourraient être traitées en 1 heure, 1 jour, 1 mois et 1 an ?*
(4) *Modifier les codes pour dessiner l'histogramme du nombre de noeuds visités par l'algorithme en fonction des niveaux.*

```
Reine( r : entier, pos : table )
si r > n alors
 traitement( pos )
 retourner

pour chaque colonne j
   si ( accept( r, j, pos ) ) alors
  pos[r] ← j
  Reine(r + 1, pos )
```

LISTING 16. Une première solution algorithmique

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

```
1 int mystibit( ensemble X )
```

```
#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 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. *(??).*

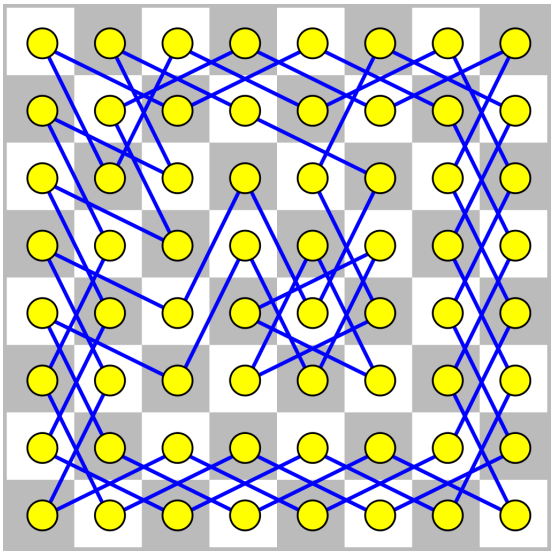


FIGURE 20. Un remarquable tour de cavalier dû à Euler.

```
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.*

Pratique 6. *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 !*

```
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 )
```

7.4. **Ensemble stable maximal.** Dans un graphe, un ensemble stable est une partie de l'ensemble des sommets composées de sommets deux à deux non adjacents. La recherche d'un sensemble stable maximal 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 30. *Montrer que le temps de calcul de l'algorithme est exponentiel dans le pire des cas.*

Exercice 31. *Implanter* `int maxstab( graphe g)` *pour déterminer la taille d'un stable maximal.*

7.5. **Clique maximale.**

7.6. **Couverture minimale.**

8. Coloriage des sommets

8.1. **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 χ_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.*

Proof. Un bon exercice ! □

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

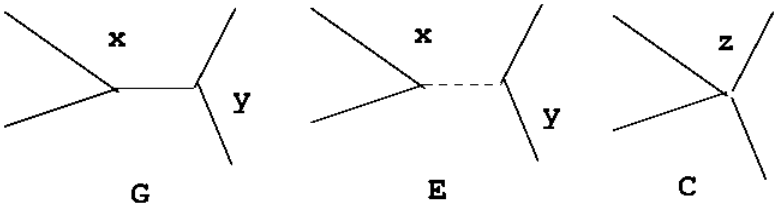


FIGURE 21. La transformation de Birkhoff. De gauche à droite, les graphe G, E et C.

8.2. **Coloriage glouton.**

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

Proof. 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. □

8.3. **Polynôme chromatique.**

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

Proof. 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 32. *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 33. *Déterminer le polynôme chromatique d'un arbre d'ordre n, d'un cycle d'ordre n.*

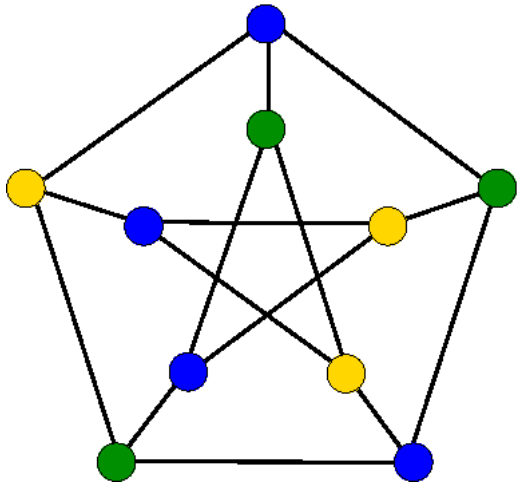


FIGURE 22. Le polynôme chromatique du graphe de Petersen est : $t^{10}-15t^9+105t^8-455t^7+1353t^6-2861t^5+4275t^4-4305t^3+2606t^2-704t$

```
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
```

8.4. **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 34 (acceptable)**.** *Compléter l'algorithme de coloriage.*

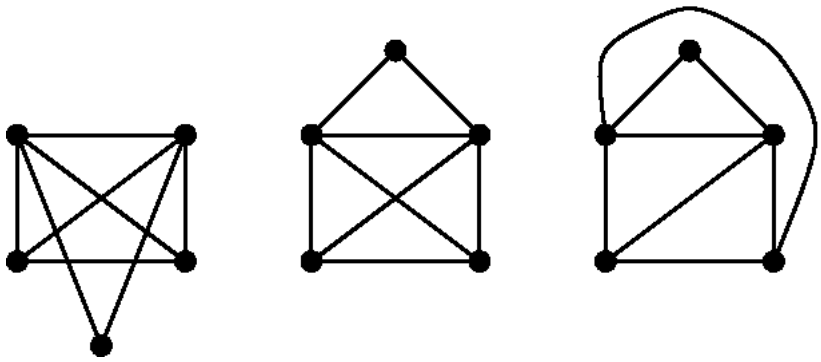


FIGURE 23. Trois représentations du graphe maison.

Pratique 7. (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. Graphe planaire

9.1. **Graphe sur le plan.** Un graphe sur le plan est un graphe dont les sommets sont des points, et les arêtes des tracés continus entre ces points. Tous les graphes peuvent être dessinés sur le plan. Sur l'ensemble des points qui ne sont pas sur des arêtes, on considère une relation d'équivalence de nature topologique : deux points sont equivalents si et seulement si on peut tracer une courbe continue qui passe par ces points sans couper d'arête du graphe. Il s'agit bien d'une relation d'équivalence. Les classes d'équivalence sont des faces de la représentation du graphe. Ainsi, un graphe planaire partitionne les points du plan en des régions bordées par des cycles qui sont des faces du graphe incluant une face extérieure (face à l'infini) et plusieurs faces intérieures (faces finies).

Remarque 1. *Le nombre de faces d'un graphe dessiné dans le plan dépend de la représentation.*

9.2. **Graphe planaire.** Un graphe est planaire quand il est possible de le dessiner sur le plan de sorte que les arêtes soient des courbes continues qui ne se croisent pas. On dit que le graphe est plongé dans le plan. La projection stéréographique montre qu'un graphe est planaire si et seulement s'il est sphérique.

Exercice 35. *Vérifier qu'un arbre est planaire avec une seule face !*

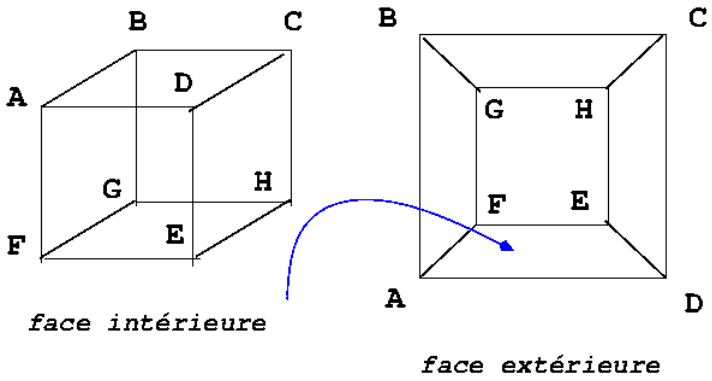
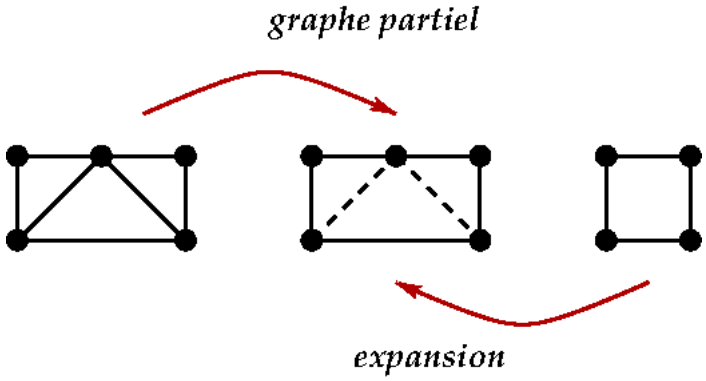


FIGURE 24. Projection stéréographique d'un cube.

Exercice 36. *Montrer qu'un graphe connexe dont un dessin planaire possède une seule face est un arbre.*

Théorème 8 (Kuratowski, Wagner)**.** *Un graphe est planaire si et seulement s'il ne possède pas de graphe partiel qui soit une expansion de K_5 ou $K_{3,3}$.*

Proof. La preuve n'est pas au programme. Nous verrons que la condition est suffisante. □



9.3. **Formule d'Euler.**

Lemme 3 (arête de cycle)**.** *Une arête de cycle d'un graphe planaire est à la frontière d'exactement deux faces.*

Proof. □

Théorème 9 (Euler)**.** *Un graphe planaire connexe d'ordre n possédant m arêtes et délimitant f faces satisfait à l'égalité d'Euler :*

$$n - m + f = 2. \tag{3}$$

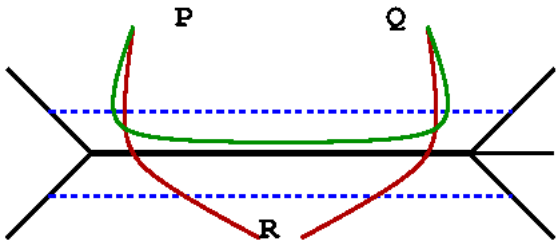


FIGURE 25. Une arête de cycle est à la frontière de deux faces.

Proof. On commence par remarquer que le résultat (3) est vrai pour les arbres. Considérons un graphe graphe planaire connexe qui ne soit pas un arbre. Il possède au moins un cycle et dinc une arête de cycle. Le lemme (3) montre que la suppression de cette arête conduit à un graphe connexe avec une face de moins et une arête de moins, ce qui laisse invariant la somme alternée de l'équation (3). □

Exercice 37. *Montrer le graphe K_5 n'est pas planaire.*

Exercice 38. *Montrer le graphe $K_{3,3}$ n'est pas planaire.*

9.4. **Le théorème des quatre couleurs.**

Théorème 10 (K. Apple, W. Haken)**.** *Tout graphe planaire est 4-colorable.*

La première démonstration complète de ce résultat consomma 1200 heure de calcul au milieu des années soixante, le résultat a été confirmé (2015) par une preuve formalisée par G. Gonthier et B. Winter au moyen de l'assistant de preuve COQ. D'après Paul Erdös, une solution simple n'est pas à exclure. Notons qu'à l'instar de Minkowski, quelques mathématiciens de premier plan ont échoué sur cette question.

9.5. **Genre.** Plus généralement un graphe est dit de genre g quand il est plongeable dans une surface de genre g. Le plan et la sphère sont des surfaces de genre nul. Le tore est une surface de genre 1. Le genre d'une surface compte les trous !

Exercice 39. *Montrer le graphe K_5 est torique.*

Exercice 40. *Montrer le graphe $K_{3,3}$ est torique.*

10. TRI TOPOLOGIQUE

10.1. **Parcours en profondeur.** Le parcours en profondeur, horodaté et tricolore est trés utile pour réaliser des preuves algorithmiques.

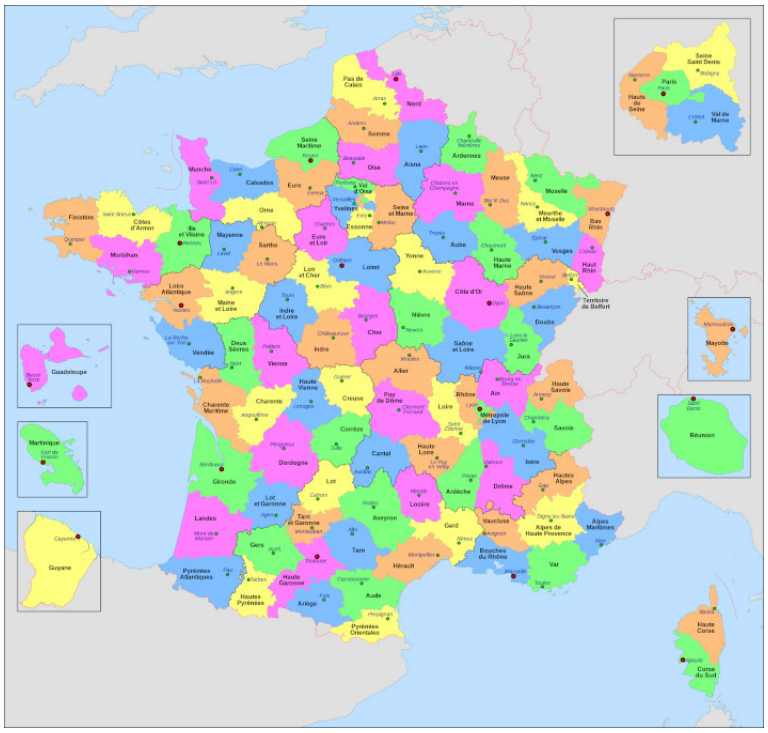


FIGURE 26. La France des départementales en 4 couleurs.

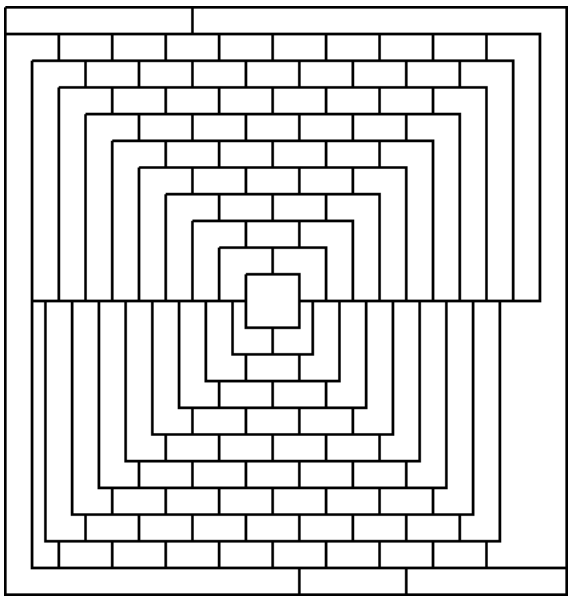


FIGURE 27. Comment colorier la carte du jardin de Martin Gardner avec 4 couleurs?

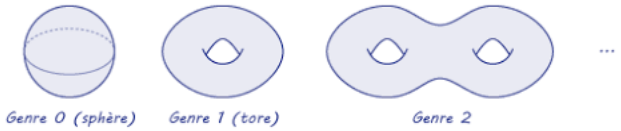


FIGURE 28. Trois surfaces de petit genre.

```
tps ← 0

PPROF( s : sommet, g : graphe )
d[ s ] ← tps ← tps + 1
clr[ s ] ← GRIS
pour chaque arc st
 si clr[t] = BLANC
  PPROF( t )
clr[ s ] ← NOIR
f[ s ] ← tps ← tps + 1


PARCOURS( G : graphe )
pour chaque sommet s
 clr[ s ] ← BLANC
pour chaque sommet s
 si clr[ s ] = BLANC
  PPROF( s, g )
```

LISTING 17. Parcours en profondeur avec horodatage

A chaque sommet s, sont associées deux dates $d(s)$ et $f(s)$. Toutes les dates sont distinctes, $d(s)$ est la date de découverte du sommet s, $f(s)$ est la date de fin de traitement. Les sommets sont coloriés en trois couleurs : blanc, gris, noir. Les sommets sont initialisés à blanc, un sommet s devient gris à l'instant $d(s)$, noir à l'instant $f(s)$.

Exercice 41 (PP en action)**.** *Préciser les prochaines étapes du parcours horodaté en cours d'exécution sur le graphe* FIG. *(29).*

Lemme 4 (des intervalles)**.** *Les intervalles* $[d(s), f(s)]$ *d'un parcours horodaté sont emboîtés ou disjoints.*

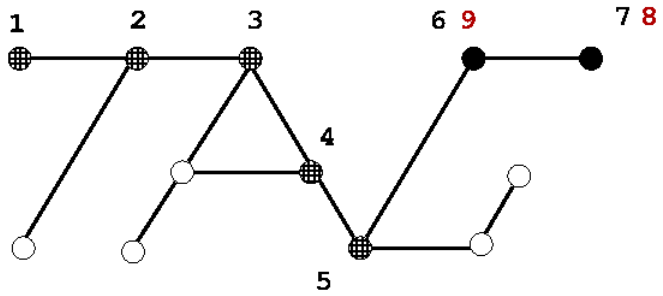
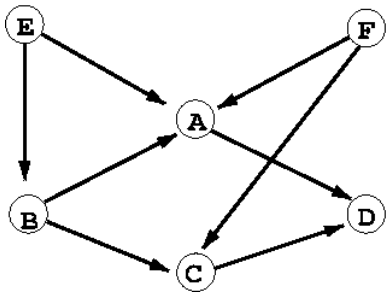


FIGURE 29. Le parcours en profondeur horodaté en action sur le graphe TAG.

Proof. Considérons deux sommets s et t tels que $d[s] < d[t]$. A l'instant $d[t]$, le sommet s ne peut pas être BLANC. Si s est NOIR alors les intervalles sont disjoints. Si s est GRIS alors t est un descendant de s et tous les voisins de t seront visités avant les voisins de s, autrement dit $f[t] < f[s]$, les intervalles sont emboîtés. □

10.2. **Tri topologique.**



Effectuer un tri topologique d'un graphe acyclique orienté (DAG) c'est ordonner les sommets du graphe de sorte que si st est un arc alors s précède t. L'ordonancement des tâches est une application classique du tri topologique.

Proposition 11. *On obtient un tri topologique d'un graphe acyclique orienté en ordonnant les sommets un ordre de fin de traitement en profondeur.*

Proof. □

Exercice 42. *Quelle est la valeur maximale de fin de traitement pour un parcours en profondeur horodaté ? Préciser le temps de calcul de la procédure de tri topologique.*

Exercice 43. *Ecrire un algorithme pour détecter la présence d'un circuit dans un graphe orienté.*

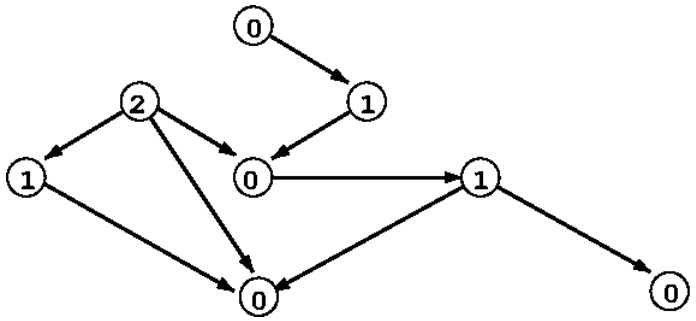


FIGURE 30. Function de Grundy sur un graphe.

Pratique 8. *Implanter la fonction* `int* triTopologique( graphe G)` *qui retourne un tableau de sommets du graphe orienté G dans un ordre topologique.*

10.3. **Fonction de Grundy.** Pour tout ensemble d'entiers naturels X, la plus petite valeur ne figurant pas dans X est notée $\operatorname{mex} X$.

Une fonction de Grundy sur un graphe orienté est une application g sur l'ensemble des sommets tel que:

$$g(s) = \operatorname{mex}\{g(t) \mid s \to t\}.$$

Proposition 12. *Un graphe orienté avec deux fonctions de Grundy possède un circuit.*

Proof. Supposons g et g' deux fonctions de Grundy différentes sur un même graphe. Il existe un sommet s tel que $g(s) \neq g'(s)$. Mézalor, la même chose pour un voisin de s et ainsi de suite. La finitude du nombre de sommets implique que ces sommets sont sur un cycle. □

Proposition 13. *Un graphe orienté acyclique possède une et une seule fonction de Grundy.*

Proof. La fonction de Grundy d'obtient de proche en proche, en parcourant les sommets dans l'ordre inverse d'un tri topologique. □

Pratique 9. *Implanter la fonction* `int* Grundy( graphe G)` *qui retourne le tableau des valeurs de Grundy des sommets du graphe G.*

10.4. **Jeu sur les graphes.** Sur un graphe orienté acyclique, on définit un jeu à deux joueurs, les sommets du graphe sont les positions du jeu. Chaque joueur joue à tour de rôle. Jouer un coup légal dans la position x c'est offrir à son adversaire une position y telle que xy est un arc du graphe. Le perdant est celui des joueurs ne pouvant plus jouer de de coup légal.

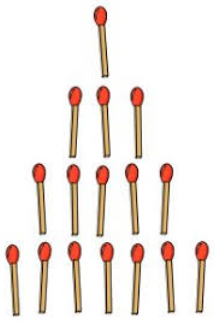


FIGURE 31. Position initiale du célèbre jeu de nim.

Proposition 14 (stratégie gagnante)**.** *Un joueur face à une position dont la valeur de Grundy n'est pas nulle joue un coup gagnant en plaçant son adversaire dans une position dont la valeur de Grundy est nulle.*

Proof. Élémentaire ! □

Dans le jeu de nim classique, popularisé par "L'année dernière à Marienbad", un étrange et mystérieux film d'Alain Resnais, on place 4 rangées d'allumettes sur une table. Une rangée avec 1 allumette, une seconde avec 3 allumettes, une troisième avec 5, et 7 allumettes sur la dernière rangée. Chaque joueur joue à tour de rôle en enlevant au moins une allumette dans une seule rangée. Le joueur face à une table vide a perdu.

Exercice 44. *La position initiale est-elle perdante ou gagnante ?*

Exercice 45. *Dans le jeu de nim inverse, le joueur face à une table vide est gagnant. Quelle est la nature de la position initiale pour le jeu de nim inverse ?*

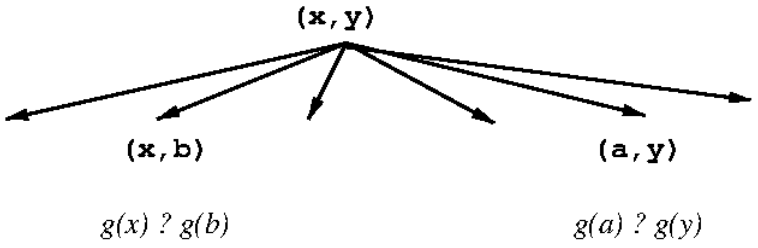


FIGURE 32. Un nouvel opérateur ?

10.5. **Somme de graphe.** On considère deux graphes $\Gamma(X, A)$ et $\Gamma'(Y, B)$. La somme de ces deux graphes est un graphe dont les sommets sont des couples de $X \times Y$ reliés par des arcs de la forme :

$$(x, y) \to (a, y), \quad (x, y) \to (x, b),$$

où $x \to a \in A$ et $y \to b \in B$ sont des arcs des graphes.

Lemme 5 (somme de graphe acyclique)**.** *La somme de deux graphes acyclique est un graphe acyclique.*

Proof. Exercice. □

Supposons l'acyclicité des graphes. Notons g_X et g_Y leur fonction de Grundy. Comment déterminer la fonction de Grundy g_S de la somme ? Anticipons, et supposons que la valeur de Grundy en (x, y) ne dépende que des valeurs $g_X(x)$ et $g_Y(y)$. Ainsi, $g_s(x, y) = g_X(x) \oplus g_Y(y)$ pour un mystérieux opérateur binaire $\oplus$ défini sur l'ensemble des entiers naturels vérifiant la formule de récurrence :

$$x \oplus y = \operatorname{mex}_{\substack{a<x \\ b<y}} \{a \oplus y, x \oplus b\}. \tag{4}$$

Exercice 46. *Montrer que pour tout entiers x et y, $x \oplus y \leq x + y$.*

Exercice 47. *Compléter la table* TAB. *(4) de la nim-addition.*

On remarque sur le champ que la loi de $\oplus$ est régulière

$$x \oplus y = x \oplus z \Longrightarrow y = z.$$

et que pour toute valeur $z < x \oplus y$ s'écrit sous la forme $z = x \oplus \beta$ avec $\beta < y$ ou $z = \alpha \oplus y$ avec $\alpha < x$.

Proposition 15. $(\mathbb{N}, \oplus)$ *est un groupe commutatif dont le neutre est 0 et dans lequel chaque élément est son propre opposé.*

TABLE 1. Le remplissage de la table de la nim-addition se fait en partant du coin en bas à gauche. La valeur d'une case est égale à la plus petite valeur exclue de la rangée et de la colonne correspondante.

x	$\dots$	$\dots$	$\dots$	$\dots$	$\dots$	$x \oplus b$	$\dots$	?
$\vdots$								$\vdots$
a								$a \oplus y$
$\vdots$								$\vdots$
3	3	2	1					$\vdots$
2	2	3	0					$\vdots$
1	1	0	3					$\vdots$
0	0	1	2	3				$\vdots$
$\oplus$	0	1	2	3	$\dots$	b	$\dots$	y

Proof. L'associativité demande un peu de travail, prouvons par induction sur l'entier N que :

$$x + y + z = N \Longrightarrow (x \oplus y) \oplus z = x \oplus (y \oplus z).$$

Supposons par exemple que

$$(x \oplus y) \oplus z < x \oplus (y \oplus z),$$

et donc $(x \oplus y) \oplus z = \alpha \oplus (y \oplus z)$ avec $\alpha < x$, ou $(x \oplus y) \oplus z = x \oplus \sigma$ avec $\sigma < (y \oplus z)$. Dans le premier cas, nous en déduisons que

$$(x \oplus y) \oplus z = \alpha \oplus (y \oplus z) = (\alpha \oplus y) \oplus z$$

et par régularité $x \oplus y = \alpha \oplus y$ puis l'absurde $x = \alpha$. Le second cas se traite de façon similaire. □

Proposition 16. *La nim somme de x et y n'est rien d'autre que leur* XOR *bit-à-bit.*

Proof. Il suffit d'établir que pour $x = \sum_{i=0}^{r-1} x_i 2^i$:

$$x = x_0 2^0 \oplus x_1 2^1 \oplus \cdots \oplus x_{r-1} 2^{r-1}$$

□

Théorème 11 (Sprague-Grundy)**.** *La fonction de Grundy de la somme de deux graphes orientés acycliques est égale à la nim-somme de leur fonction de Grundy.*

Proof. Par la mex-définition. □

FIGURE 33. ONAG.

10.6. **Nimber.** L'inventif mathématicien John Conway a mis en évidence un phénomène extraordinaire en osant une définition récursive pour une multiplication. Envisageons un produit $\otimes$ sur l'ensemble des entiers naturels qui soit distributif par rapport à la nim-addition. Plus encore, imaginons, l'intégrité de cette loi : un produit est nul si et seulement si l'un des facteurs est nul. Pour tout entier $a < x$, pour tout entier b, le produit $(x \oplus a) \otimes (y \oplus b)$ n'est jamais nul, ce qui suggère la définition récursive pour une nim-multiplication :

$$x \otimes y = \operatorname{mex}_{\substack{a<x \\ b<y}} \{a \otimes y \oplus a \otimes b \oplus x \otimes b\}. \tag{5}$$

Curieusement cette loi vérifie toutes les propriétés algébriques usuelles : neutralité de 1, associativité, commutativité, distributivité sur la nim-addition.

Exercice 48. *Compléter la table de la nim-multiplication pour les entrées strictement inférieures à 4. Commenter !*

Exercice 49. *Calculer le nim produit de* $4 \otimes 4$ *!*

On peut alors démontrer que

Théorème 12 (corps de Conway)**.** $(\mathbb{N}, \oplus, \otimes)$ *est un corps commutatif.*

Proof. C'est l'objet du chapitre 6 du livre de Conway *On Numbers and Games* : the Curious Field On_2. La preuve est un petit peu délicate mais vous pouvez vous amuser à esquisser les étapes d'une démonstration. □

Exercice 50. *Établir deux propriétés au choix de la nim-multiplication.*

Pratique 10. *Mettre oeuvre la nim-multiplication.*

REFERENCES

[1] Claude Berge. *Graphes.* μ_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] 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.
[4] 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.

[5] 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.
[6] Donaldi E. Knuth. Dancing links, 2000. http://arxiv.org/abs/cs/0011047.
[7] Philippe Langevin. Le tournoi de nimbad à euphoria, 19xy. http://langevin.univ-tln.fr/problemes/NIMBAD/marienbad.html.
[8] Philippe Langevin. Une brève histoire des nombres, 2003. http://langevin.univ-tln.fr/notes/rsa/rsa.pdf.
[9] Harold. F. Mattson. *Discrete Mathematics with Applications*. Wiley international editions. Wiley, 1993.
[10] Herbert J. Ryser and Paul Camion. *Méthodes Combinatoires*. Monographie Dunod. Dunod, 1969.
[11] Robert Sedgewick. Permutation generation methods, x. https://www.cs.princeton.edu/~rs/talks/perms.pdf.
[12] Robert Sedgewick and Jean-Michel Moreau. *Algorithmes en langage C*. I.I.A. Informatique intelligence artificielle. Dunod, 1991.
[13] Neil. J. A. Sloane. On-line encyclopedia of integer sequences, 1964. https://oeis.org/?language=french.
[14] Peter Wegner. A technique for counting ones in a binary computer. *Communications of the ACM*, 3(5), 1960.
[15] Herbert S. Wilf. *Algorithms and complexity*. Prentice-Hall international editions. Prentice-Hall, 1986.
[16] Jacques Wolfmann and Michel Las Vergnas. Matériaux de mathématiques discrètes pour le musée de la villette, 19xy.