

THÉORIE ET ALGORITHMIQUE DES GRAPHS LICENCE INFORMATIQUE

PHILIPPE LANGEVIN, IMATH, UNIVERSITÉ DE TOULON.

backtrack.tex	2025-03-24	07:14:10.832512338
connexe.tex	2025-03-23	19:58:16.798786980
disjoint.tex	2025-03-19	06:54:19.342497435
zoologie.tex	2025-03-09	15:28:31.230676466
reduction.tex	2025-03-09	15:28:31.249676583
implantation.tex	2025-02-25	07:36:46.516490285
topo.tex	2025-02-25	06:52:15.709785890
tag.tex	2025-02-25	06:52:15.708785885
rush.tex	2025-02-25	06:52:15.708785885
promenade.tex	2025-02-25	06:52:15.706785876
planaire.tex	2025-02-25	06:52:15.705785870
macros.tex	2025-02-25	06:52:15.704785865
np.tex	2025-02-25	06:52:15.705785870
hamilton.tex	2025-02-25	06:52:15.701785850
euler.tex	2025-02-25	06:52:15.700785845
devoir.tex	2025-02-25	06:52:15.698785835
coloriage.tex	2025-02-25	06:52:15.697785830
acm.tex	2025-02-25	06:52:15.695785820

TABLE DES MATIÈRES

1. Chemin eulérien	5
1.1. Terminologie	5
1.2. Caractérisation	8
1.3. Algorithme	9
2. Mise en oeuvre	9
2.1. organisation	9
2.2. Format de fichier source	13
2.3. matrice d'adjacence	13
2.4. Entrée-sortie	14
3. pile et promenade	18
3.1. pile de sommets	19
3.2. Expérience eulérien	20
3.3. Liste d'adjacence	20
4. Composante Connexe	20
4.1. Connexité	20
4.2. Parcours récursif	23
4.3. Matrice d'ajacence	24
4.4. Liste d'adjacence	25
4.5. Profilage	26
4.6. Classe polynomiale	28
4.7. Graphe aléatoire	28
5. Ensembles Disjoints	30
5.1. Structure d'ensembles disjoints	30
5.2. Implantation par liste	31
5.3. Forêt d'ensemble disjoint	33
5.4. Expérience Numérique	34
6. Graphe Hamiltonien	37
6.1. Le voyage autour du monde	37
6.2. Deux conditions suffisantes	38
6.3. Code de Gray	39
6.4. Suite de de Bruijn	41
7. Backtracking sur l'échiquier	41
7.1. Les reines de l'échiquier	41
7.2. Bit programming	44
7.3. Cycle Hamiltonien	45
7.4. Stable-Clique-Domination	46
8. Zoologie	48
8.1. Adjoint et complémentaire	48
8.2. Permutation	49
8.3. Isomorphisme de graphe	51

8.4.	invariant	52
8.5.	Classification	53
9.	Réduction	53
9.1.	réduction polynomiale	54
9.2.	Euler vs Hamilton	54
9.3.	Classe Non déterministe Polynomiale	56
9.4.	Problème NP-complet	56
9.5.	Gadget	56
9.6.	Primalité	56
10.	Arbre couvrant minimal	57
10.1.	Arborescence	57
10.2.	Graphe pondéré	60
10.3.	Stratégie glouton	60
10.4.	Algorithme de Kruskal	61
10.5.	Algorithme de Jarnik-Prim	62
10.6.	algorithme de Dijkstra	63
10.7.	Approximation	64
10.8.	$\frac{3}{2}$ -approximation de Christophides	66
11.	Coloriage des sommets	67
11.1.	Problème de coloration	67
11.2.	Coloriage glouton	67
11.3.	Polynôme chromatique	68
11.4.	Coloriage	68
11.5.	lien dansant	69
12.	Tri topologique	71
12.1.	Parcours en profondeur	71
12.2.	Tri topologique	72
12.3.	Fonction de Grundy	72
12.4.	Jeu sur les graphes	73
12.5.	Somme de graphe	74
12.6.	Nimber	76
	Références	77



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

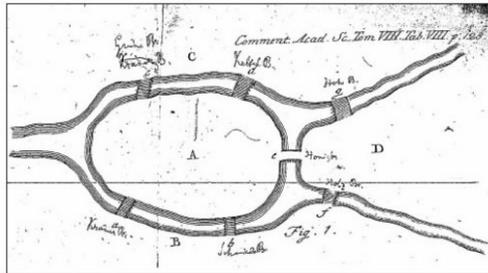


FIGURE 1 – Peut-on faire une promenade passant une et une seule fois par chacun des sept ponts de la ville de Königsberg ?

1. CHEMIN EULÉRIEN

La théorie des graphes s'est développée au cours du XX^e siècle, dans la note [19], 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 genèse 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 Königsberg ? 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.

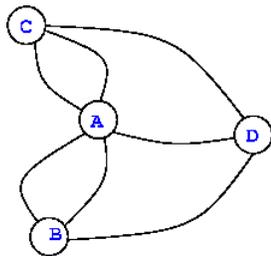


FIGURE 2 – 2-graphe.

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 inclus dans $\mathcal{P}_2(S)$. Une arête est une paire de sommets, ce sont les extrémités de l'arête. Une arête $\{x, y\}$ est notée xy ou yx , les sommets

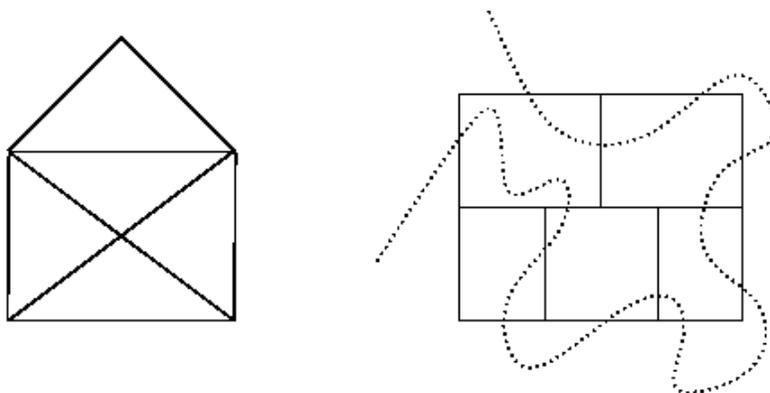


FIGURE 3 – Peut-on tracer la petite maison d'un trait de crayon sans passer deux fois par le même segment ? Peut-on tracer une courbe qui coupe une et une seule fois tous les segments de la figure ?

sont dits *adjacents* et l'arête $u = xy$ est dite *incidente* au sommet x . Nous utiliserons parfois les symboles $-$ et $—$ pour l'adjacence et l'incidence. Plus généralement, on dit que x est adjacent à $X \subseteq S$:

$$s - X \iff \exists x \in X, \quad s - x.$$

Deux arêtes u et v qui partagent un sommet sont dites *incidentes*. Plus généralement, on dit que $u \in A$ est incidente à $X \subseteq S$:

$$\exists x \in X, \quad u - x.$$

Un *chemin* de longueur n est une suite de sommets x_0, x_1, \dots, x_n tels que :

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

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 :

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

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 [13] s'applique sans difficulté et donne :

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

□

Exercice 1 (dénombrement des graphes). *Dénombrer les 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 .

Remarque 2. *Nous utiliserons les lettres m , n et p pour le nombre d'arêtes, de sommets et de composantes connexes.*

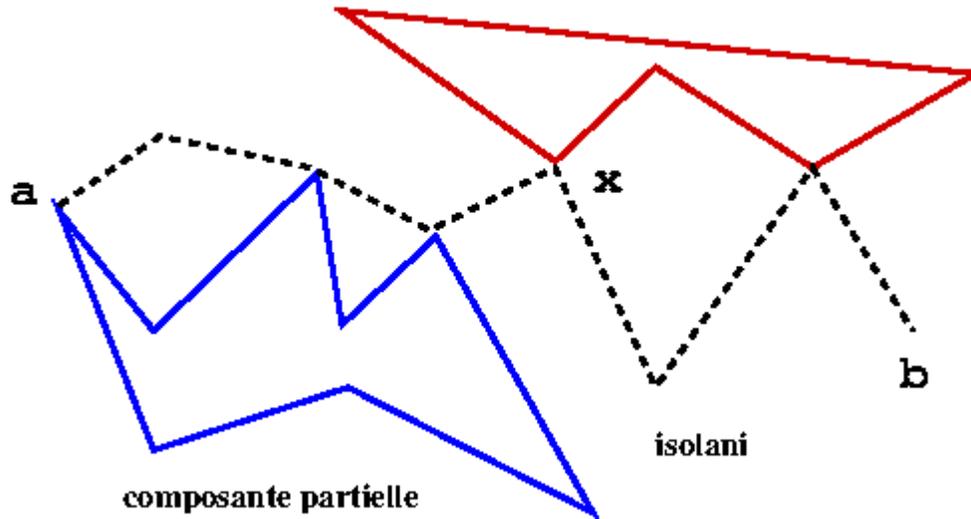


FIGURE 4 – Les composantes connexes du graphe partiel qui ne sont pas des singletons forment des cycles eulériens.

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.*

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

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.

```

1 Eulerien( s:sommet, G:graphe )
2
3   P ← promenade( s, G )
4   pour chaque sommet x de P
5       si degre( x ) > 0 alors
6           Eulerien(x, G)
7       sinon
8           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. `eulerien(s, G)` est quadratique en l'ordre du graphe.

Démonstration. \square

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 variables et `alias` dans votre fichier `.bashrc` :

```

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

```

```

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

```

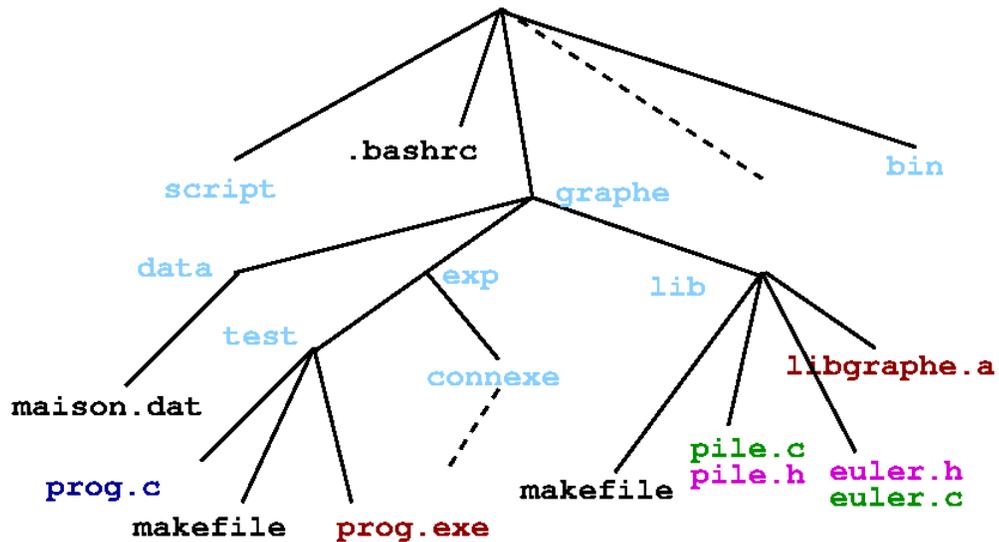
LISTING 1 – La bibliothèque statique `libgraphe.a`

FIGURE 5 – Hiérarchie des fichiers.

commande	option	commentaire
<code>man</code>	-k -f	à propos whatis
<code>bash</code>		interprète
<code>make</code>	-B -C -n	compilation automatique compiler tout compiler ailleurs juste pour voir
<code>gcc</code>	-Wall -g	compilateur avertissements courants produire un code débogable
<code>gdb</code>	-args	débogueur ligne argumentée
<code>valgrind</code>		débogueur

TABLE 1 – 6 commandes ...

```

1 OPTIONS= -Wall -g
2 OBJET=graphe.o inout.o
3 libgraphe.a : $(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 pour la bibliothèque

Les structures de données et algorithmes de bases seront implantés dans des fichiers séparés pour constituer une bibliothèque. Les fonctions de la bibliothèque seront exploitées par des expériences qui seront codées dans des répertoires dédiés.

Les sources **C** de la bibliothèque sont déposées dans un répertoire unique piloté par un fichier makefile pour une mise à jour d'une archive statique `libXYZ.a`. Vous trouverez ci-dessous un exemple de fichier makefile pour la bibliothèque et un peu plus loin un makefile pour un répertoire de test.

Pour ce cours, les sources C seront considérées comme acceptables à condition de passer une compilation avec l'option `-Wall`, sans erreur ni avertissement. L'option de débogage `-g` est indispensable pour pouvoir utiliser les débogueurs `gdb` et `valgrind`.

Le programme `test.c` utilise la fonction `getopt` de la glibc pour gérer quelques options sur la ligne de commande. Il code une commande pour construire et dessiner un graphe aléatoire.

```
1 #include "graphe.h"
2 #include "inout.h"
3 #include <unistd.h>
4 extern int verb;
5 int main(int argc, char *argv[])
6 {
7     char *option = "n:p:v";
8     int opt, n=64;
9     float p = 0.5;
10    while ((opt = getopt(argc, argv, option)) != -1) {
11        switch (opt) {
12            case 'n':
13                n = atoi( optarg );
14                break;
15            case 'p':
16                p = atof( optarg );
17                break;
18            case 'v':
19                verb++;
20                break;
21            case '?':
22                exit (EXIT_FAILURE);
23        }
24    }
25    graphe g = randGraphe( p, n );
26    dessiner( g, NULL );
27    return 0;
28 }
```

Pratique 1 (Preliminaire). *Préparer l'environnement de travail pour les séances à venir.*

```

1  OPTIONS= -Wall -g -I$(GRAPHE)/lib -L$(GRAPHE)/lib
2
3  prog.exe : prog.c
4      gcc $(OPTIONS) -o prog.exe -lgraphe

```

LISTING 3 – makefile pour un répertoire test

- 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

```

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

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

Pratique 2 (commun). Dans les fichiers `graphe.[ch]`, coder les fonctions :

```

1 typedef struct {
2     int     nbs; // ordre du graphe
3     int     **mat; // matrice d'adjacence
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 4 – `graphe.h`

- `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!

```

#           0
#           / \
#           1 — 2
#           | X |
#           3  _ 4
nbs=5
0 1
0 2
1 2
1 3
1 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 ) > 1 )
9             printf ( "x=%d y=%d\n", x, y );
10        else fgetc ( src );
11    }

```

LISTING 5 – Le service minimum en matière d'entrée/sortie!

```

2 3
2 4
3 4

```

Le service minimum de la fonction `minilire` n'aura pas coûté bien cher en temps de développement ! Elle impose un format de fichier pas très agréable pour l'utilisateur.

```

nbs=5
0 : 1 2
1 : 3 4
2 : 3 4
3 : 4

```

Exercice 6. *Ecrire une fonction de lecture pour le format xyz : `fgets`, `sscanf`, `strtok` etc.*

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 7. *Utiliser l'outil de compilation `flex` pour la lecture des graphes au format xyz.*

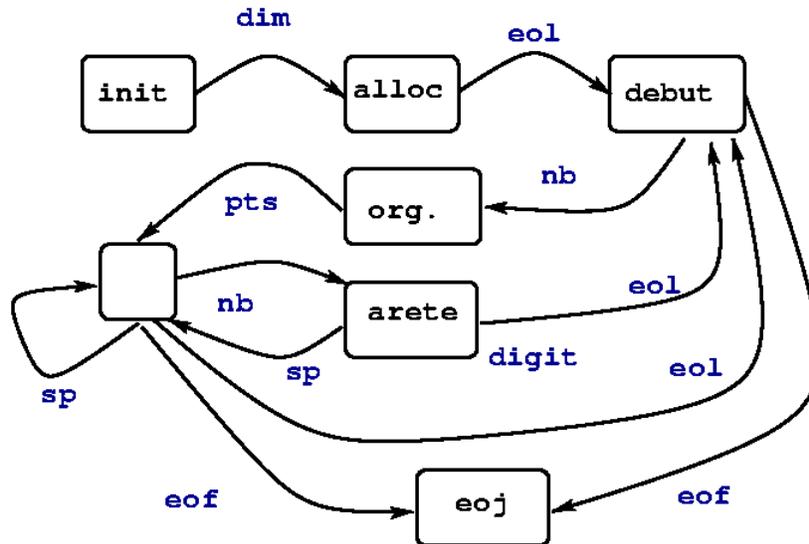


FIGURE 6 – Automate pour la lecture d'un graphe au format xyz.

Pratique 3 (entrée-sortie). *Il s'agit d'implanter des fonctions d'entrées-sorties pour manipuler les petits graphes au format décrit dans (2.2) en vous appuyant sur l'automate de la figure FIG. (6) et l'analyseur lexical LIST. (9). 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)`.

DOT(1)	General Commands Manual	DOT(1)
NAME		
<code>dot</code> – filter for drawing directed graphs		
<code>neato</code> – filter for drawing undirected graphs		
...		
SYNOPSIS		
<code>dot [options] [files]</code>		
<code>neato [options] [files]</code>		

La commande externe `dot` (et ses cousines) du projet `graphviz` 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`.

```

1 %{
2     #include "graphe.h"
3     int i, j, n, num = 0;
4     graphe g;
5 }%
6 NB  [0-9]+
7 %s  NBS  ADJ
8 %%
9 <INITIAL>nbs[ ]+= BEGIN(NBS);
10 <NBS>{NB} g = initGraphe(atoi(yytext));BEGIN(INITIAL);
11 <INITIAL>{NB} i = atoi(yytext);
12 <INITIAL>: BEGIN(ADJ);
13 <ADJ>{NB} g.adj[ i ][ j ] = g.adj[ j ][ i ] = 1;
14 #.*      ;
15 .        ;
16 \n      num++;
17 %%
18 int main(int argc, char **argv ) {
19     yyin = fopen(argv[ 1 ], "r");
20     yylex();
21     dessiner( g );
22     return 0;
23 }

```

LISTING 6 – clin d’œil au cours de compilation !

```

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

```

LISTING 7 – `inout.h`

SYSTEM(3) Manuel du programmeur Linux SYSTEM(3)

NOM

system – Executer une commande shell

SYNOPSIS

#include <stdlib.h>

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

LISTING 8 – maison au format dot

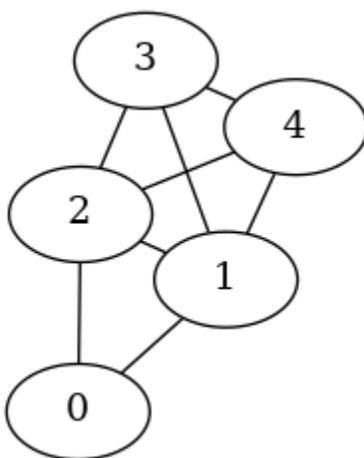


FIGURE 7 – neato -Tpng maison.dot -o maison.png.

```
int system(const char *command);
```

DESCRIPTION

La fonction de bibliothèque `system()` utilise `fork` (2) pour créer un processus fils qui exécute la commande d'interpréteur indiquée dans `command...`

Exercice 8. Comment construire l'image FIG. (8) à partir de la source `pi.c` ?

3. PILE ET PROMENADE

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

Exercice 9. Comment adapter l'algorithme `eulerien(G)` pour une promenade à pile.

```

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

```

LISTING 9 – Un exemple d’analyseur lexical.

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

Exercice 10 (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 11 (hypercube). *Quels sont les paramètres de l’hypercube de dimension r ? L’hypercube est-il eulérien ? Ecrire une fonction `graphe`*

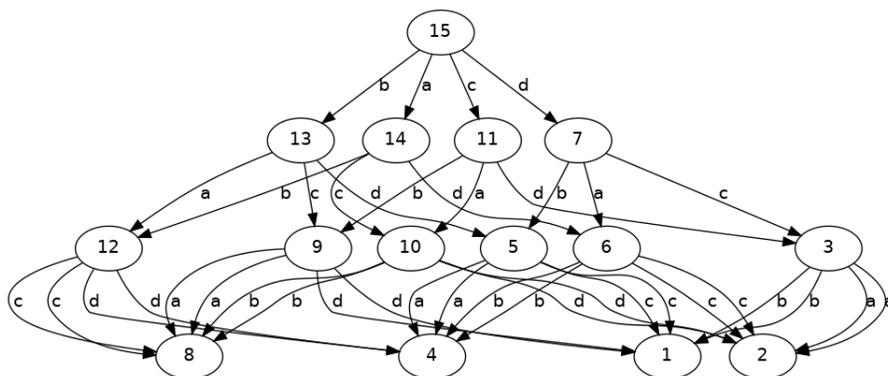


FIGURE 8 – Comment obtenir cette image à partir de la source `pi.c`?

`hypercube(int r)` qui construit le graphe de l'hypercube de dimension r .

3.2. Expérience eulérien.

Pratique 4 (commande `/usr/bin/time`).

Ecrire une commande `eulerien.exe` qui détermine un cycle eulérien 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.

Pratique 5. Dans le fichier `graphe.[ch]`

- 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 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.

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 typedef unsigned int set;
4 #define single(x) ( ( (set) 1 ) << x )
5 void pi(set X)
6 {
7     if (X) {
8         set Y = X;
9         while (Y) {
10            int y = __builtin_ctz (Y);
11            set Z = X ^ single(y);
12            if (Z)
13                printf ( "%d -> %d [ label=\"%c\" ^n", X, Z, y + 'a' );
14            pi(Z);
15            Y ^= single(y);
16        }
17    }
18 }
19
20 int main(int argc, char *argv[])
21 {
22     int n = atoi(argv[1]);
23     puts("digraph pi {");
24     set X = single(n) - 1;
25     pi(X);
26     puts("}");
27     return 0;
28 }

```

LISTING 10 – What that ?

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

Exercice 12 (abondance). *Montrer que les graphes connexes sont plus nombreux que les graphes déconnectés.*

Lemme 2. *Le détachement un sommet d'une petite composante vers une grande composante et la fusion de deux composantes sont deux transformations qui augmentent le nombre d'arêtes.*

Exercice 13 (Berge). (1) *Quel est le nombre d'arêtes de K_n ?*

```

1 pile promenade( sommet x , graphe G )
2 {
3   variable
4     y : sommet
5     P : pile
6     initialiser (P)
7     tantque ( degre(x) > 0 )
8       y ← un voisin de x
9       empiler( y, P );
10      deconnecter x et y
11      x ← y;
12 }
13 retourner P
14 ftq

```

LISTING 11 – 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 } enliste , * liste ;
7
8 void empiler( int s, liste *p );
9 int depiler ( liste *p );
10 #endif

```

LISTING 12 – Structure de pile basée sur des listes chaînées

(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.

```

1 typedef struct {
2     int     nbs; // ordre du graphe
3     int     **mat; // matrice d'adjacence
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 13 – graphe.h

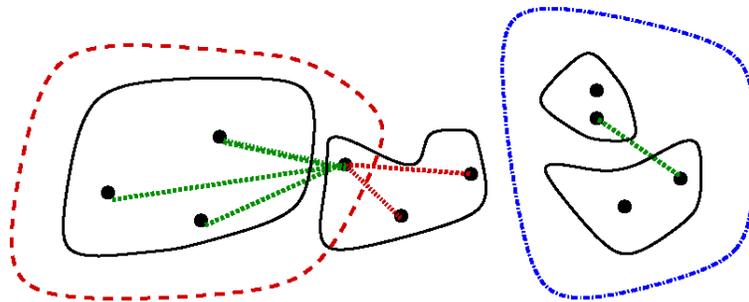


FIGURE 9 – augmentation du nombre d'arêtes d'un graphe non connexe...

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.

Exercice 14 (élimination de la récursivité). *Comment utiliser explicitement une structure de de pile pour éliminer la récursivité de LIST. (14).*

```

1
2 parcour( x, G )
3   marquer[x] ← 1
4   compteur++
5   pour chaque voisin y de x dans G
6     si marquer[y] < 1 alors
7       parcour( y , G )
8
9
10 connexe( graphe G )
11   initialiser les marques
12   compteur ← 0
13   choisir un sommet s
14   parcour( s , G )
15   retourner compteur == ordre(G)

```

LISTING 14 – Marquage des sommets par un parcours récursif en profondeur

```

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

```

Exercice 15. *Ecrire un algorithme pour dénombrer les composantes connexes d'un graphe.*

4.3. Matrice d'adjacence. 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)*

```

1 typedef struct _ls_ {
2     sommet num;
3     struct _ls_ * next;
4 } enliste , *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 15 – liste d'adjacence

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 . \square

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'adjacence). *Dans le cas d'une implantation par liste d'adjacence, le temps de calcul de $\text{connexe}(G)$ vérifie :*

$$T(m, n) = O(m + n)$$

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. \square

Exercice 16. Coder `int adjacent(sommet x, y, graphe g)` qui retourne 1 si xy est une arête du graphe.

Exercice 17. Coder `void relier(sommet x, y, graphe g)` qui ajoute l'arête xy au graphe.

Exercice 18. 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.

Pratique 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 comparer 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 16 – 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	calls	self	total
---	------------	------	-------	------	-------

time	seconds	seconds	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					
[1]	100.0	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 19. *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 classer 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. □

4.7. Graphe aléatoire. Un graphe $G(p, n)$ est un graphe léatoire d'ordre n et de paramètre $p < n$, s'obtient en reliant deux sommets

distincts d'un graphe d'ordre n avec la probabilité $p/(n-1)$. A l'opposé du graphe $G(0, n)$ est constitué de n points isolés, le graphe $G(n-1, n)$ est complet. Il est clair qu'une grande valeur de p a tendance à produire des graphes connexes alors qu'une petite valeur de p donne des sommets isolés et des composantes de petite tailles.

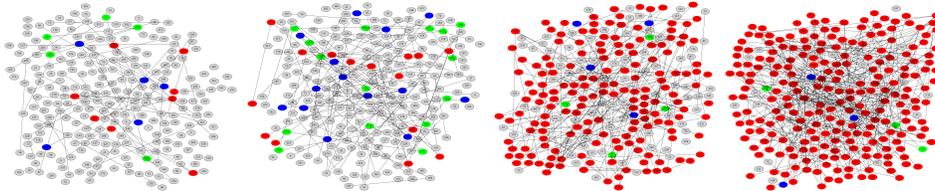
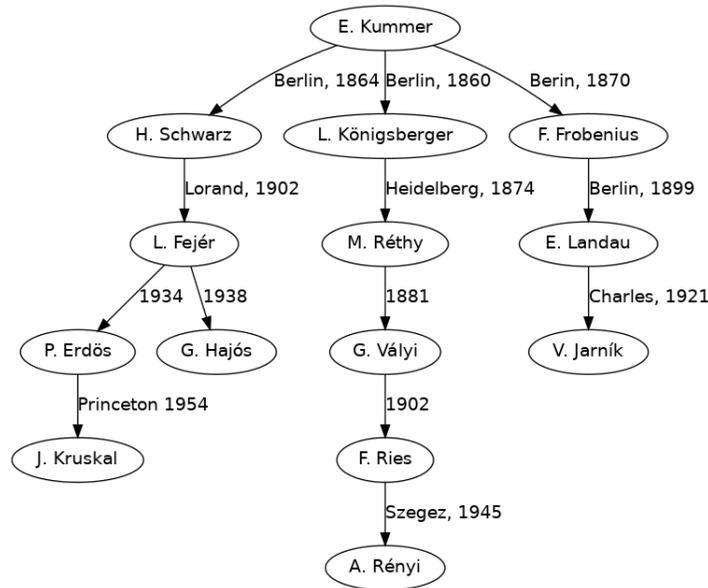


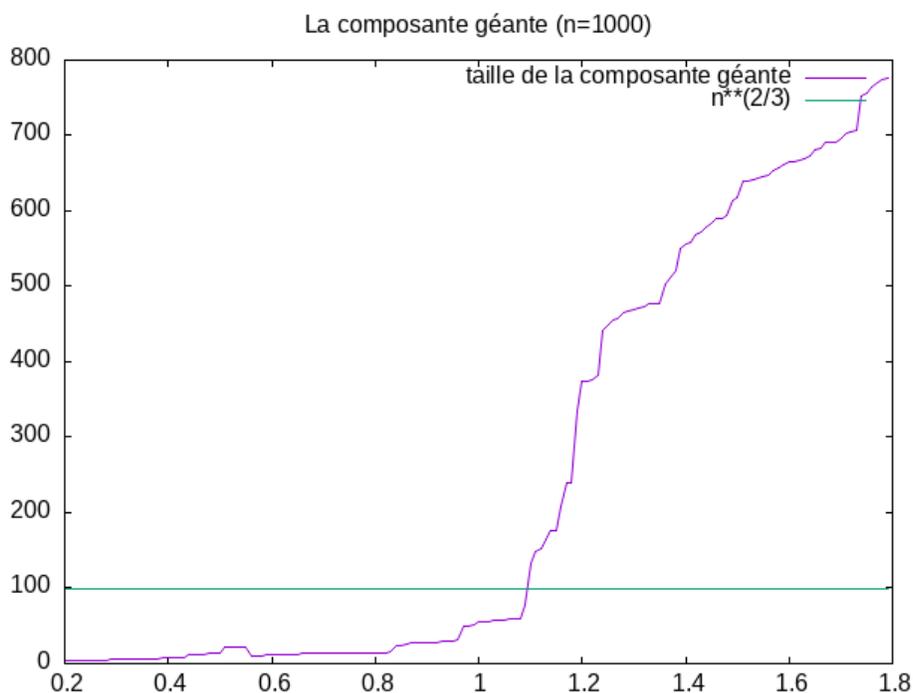
FIGURE 10 – Graphes $G(p, 256)$ avec $p = 0.5, 1, 1.5, 2.5$. Les sommets de la plus grande composante sont rouges, puis verts et bleus (Ange Colletto).



Dans un article de 1960, les mathématiciens hongrois Paul Erdős et Alfréd Rényi ont prouvé la formation d'une composante 'géante' de taille $n^{2/3}$, à partir de la valeur seuil $p = 1$.

La taille moyenne de la plus grande composante connexe de ces graphes a pour ordre de grandeur $n^{2/3}$ lorsque le paramètre p est positif et proche de 1.

Pratique 7 (Composante géante). *Réaliser une expérience numérique pour mettre en évidence le comportement de la composante géante d'un graphe aléatoire.*

FIGURE 11 – valeur de seuil $p = 1$ (Nikita Rodyhin).

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 jusqu'à 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 $\{0, 1, \dots, n - 1\}$. 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 .

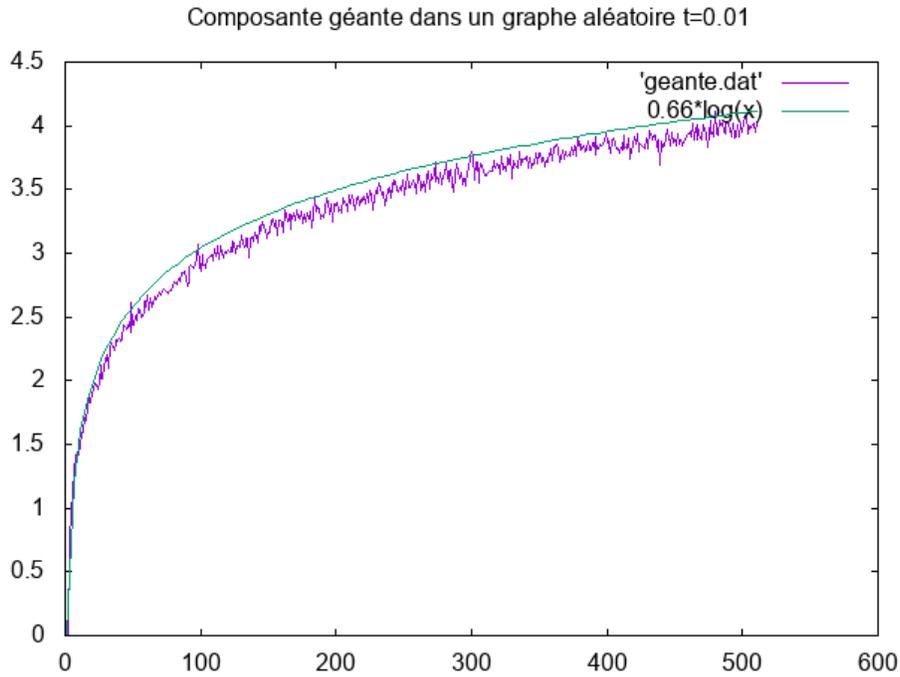


FIGURE 12 – Mise en évidence de l'exposant $\frac{2}{3}$ (Richard Sébastien).

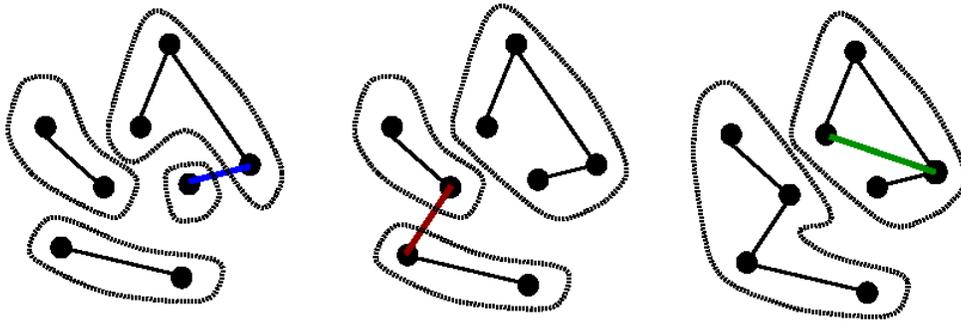


FIGURE 13 – Évolution d'une structure d'ensembles dis-joints.

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 $O(n + m^2)$. On peut y remédier par une heuristique simple et puissante. L'heuristique de

```

1 graph {
2 0 [style = filled , color = blue, pos = "0.002, 0.157!"]
3 1 [style = filled , pos = "-0.405, 0.462!"]
4 2 [style = filled , pos = "-0.175, -0.416!"]
5 3 [style = filled , color = red, pos = "-0.158, 0.075!"]
6 4 [style = filled , pos = "0.150, 0.269!"]
7 5 [style = filled , color = red, pos = "0.013, 0.182!"]
8 6 [style = filled , pos = "0.374, 0.000!"]
9 7 [style = filled , color = green, pos = "-0.490, -0.007!"]
10 8 [style = filled , pos = "-0.147443, 0.439!"]
11 9 [style = filled , color = green, pos = "0.493, -0.457!"]
12
13 3--5 [color = black]
14 7--9 [color = black]
15
16 }

```

LISTING 17 – Exemple de fichier pour fdp, dot

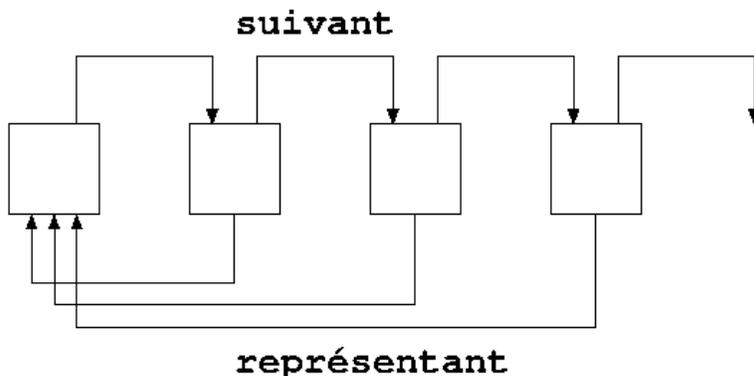


FIGURE 14 – Représentation par liste chaînée.

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

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

```

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 18 – implantation par liste chaînée.

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.

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 21. *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 $O(m \log n)$.*

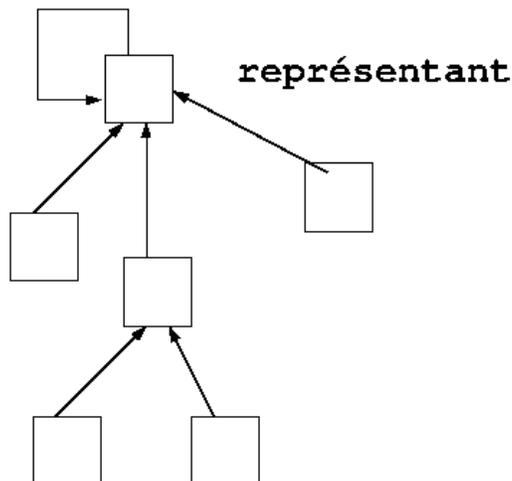


FIGURE 15 – Représentation arbre.

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

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 raccourcissent au fur et à mesure des recherches de représentants. C'est l'heuristique de la compression de chemin, d'une redoutable efficacité!

Exercice 22. *Implanter l'heuristique de la compression de chemin.*

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

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

Exercice 23. *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 $O(m \log^* n)$.*

Démonstration. Hors programme. \square

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

```

1 typedef struct _edt_ {
2     struct _edt_ *rep;
3     int rang;
4     int num;
5 } enrdisjoint , *disjoint ;
6
7 disjoint representant(disjoint x){
8     while ( x->rep != x ) x = x -> rep;
9     return x;
10 }
11
12 disjoint singleton(int v);
13
14 void reunion(disjoint x, disjoint y)
15 {
16     // version de base...
17     x = representant(x);
18     y = representant(y);
19     x->rep = y->rep
20 }
21 }

```

LISTING 19 – implantation par forêt d'arbre.

uniquement le nombre de ces arêtes. On procède 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 le résultat d'une expérience numérique en évitant d'inutiles répétitions de calculs.

Exercice 24 (exemple de script). *Que fait le script `plotcov.sh` ?*

Pratique 8 (scriptage empirique). *Il s'agit de mettre en place une expérience numérique pour déterminer la fonction empirique σ décrite plus haut.*

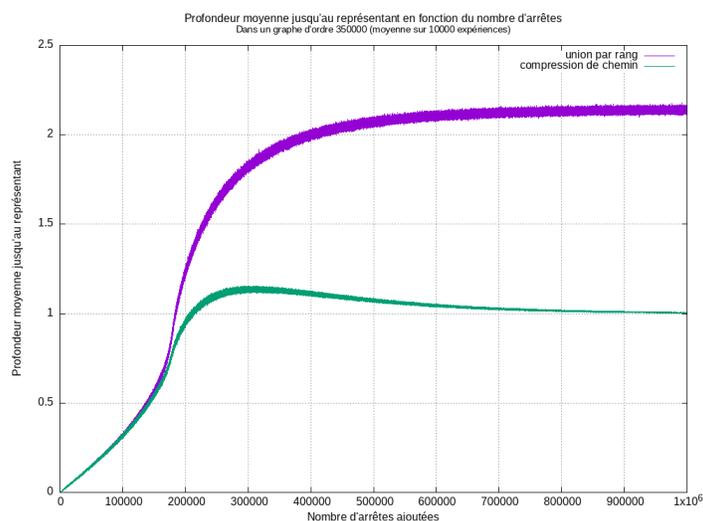


FIGURE 16 – Temps de calcul d'un représentant : compression de chemin vs union par rang, Sedkaoui Mathis 2023.

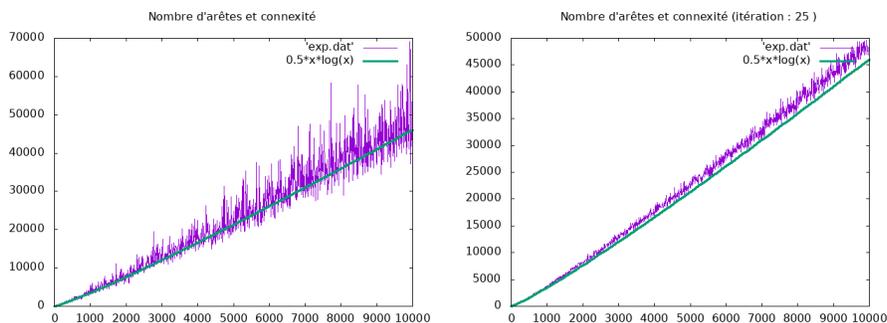


FIGURE 17 – Nombre d'arêtes et connexité.

- (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. (20) pour dessiner et identifier avec `gnuplot` le graphe de la fonction empirique.*

```

1 #!/bin/bash
2 if [ ! -f sigma.dat ] ; then
3     for(( n=1; n < 1024; n++ )); do
4         ./sigma.exe $n
5     done > sigma.dat
6 fi
7 gnuplot << EOJ
8     set term png
9     set output 'sigma.png'
10    plot 'sigma.dat', x*log(x)
11 EOJ
12 # $

```

LISTING 20 – script à compléter



FIGURE 18 – Les cinq solides de Platon : tétraèdre, hexaèdre, octaèdre, dodécaèdre et icosaèdre sont-ils Hamiltonien ?

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

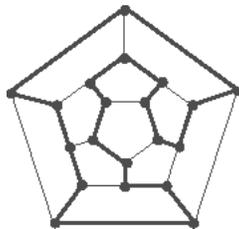


FIGURE 19 – Une route autour du monde.

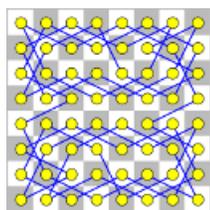


FIGURE 20

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 exhibe 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 25. 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 3. 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 (O. 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. □

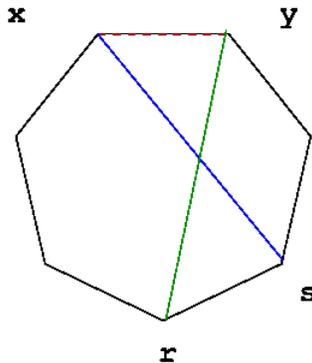


FIGURE 21

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. \square

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 $x = (x_m \cdots x_2x_1)$ et $y = (y_m \cdots y_2y_1)$ compte le nombre de différence entre les composantes :

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

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 $d_H(x, y) = 1$.

Exercice 26 (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.*

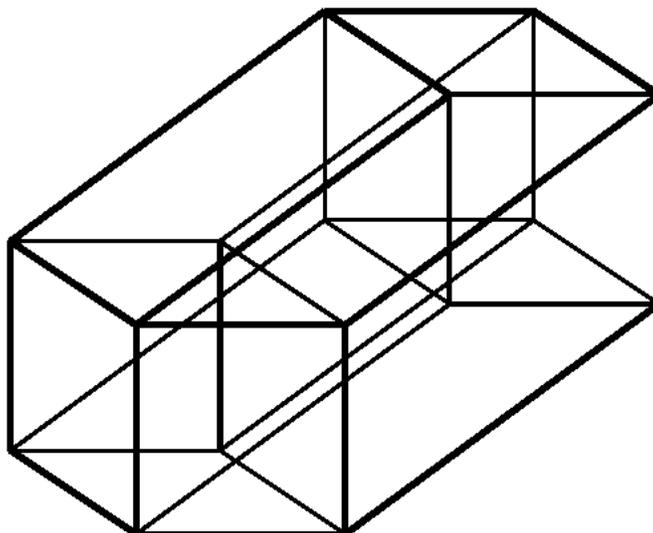


FIGURE 22
L'hypercube de dimension 4.

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

```

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

```

LISTING 21 – La
procédure
`enumGray` assemblée
par `gcc` option `-S` du
compilateur.

Exercice 27 (code de Gray). *Le code LIST. (22) affiche un cycle Hamiltonien de l'hypercube de dimension m . Il est utilisé 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 mnémotechnique assembleur LIST. (21) de la fonction `enumGray` ?*

6.4. Suite de de Bruijn. Le graphe de de Bruijn de rang n , est un graphe orienté d'ordre n dont les sommets sont les mots binaires de n -bit. Les arcs sont de la forme :

$$(x_1, x_2, \dots, x_n) \longrightarrow (x_2, x_3, \dots, x_n, b), \quad b \in \{0, 1\}$$

Les circuits de ce graphe sont les suites de de Bruijn. Le graphe est Hamiltonien, et le nombre de circuits a été déterminé en 1864 par Camille Flye Sainte-Marie, voir la déclaration de de Bruijn.

Exercice 28. *Montrer que le nombre de circuit du graphe de de Bruijn sur n -bit est inférieur à 2^{n-1} .*

Pratique 9. *Ecrire un programme pour lister les suites de De Bruijn de rang 2, 3, 4, 5 et 6.*

7. BACKTRACKING SUR L'ÉCHIQUIER

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

7.1. Les reines de l'échiquier. De combien de façons peut-on placer n dames (reines) 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

```

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 22 – Énumération de Gray des mots binaires

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 :

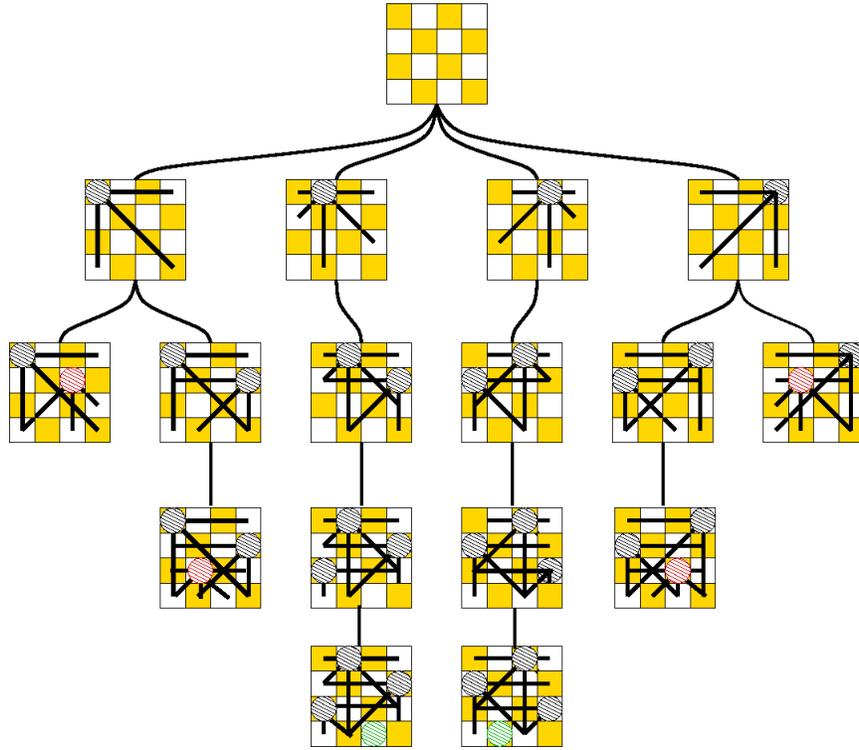


FIGURE 23 – L'arbre de backtracking pour $Q(4) = 2$.

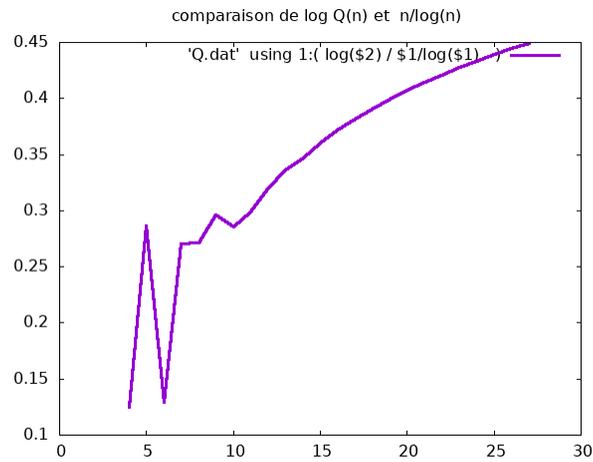


FIGURE 24 – Comparaison avec la fonction $n \log(n)$.

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

```

1 placeDame( r : entier , pos : table )
2 si r > n alors
3   traitement( pos )
4   retourner
5 pour chaque colonne j
6   si ( accept( r , j , pos ) ) alors
7     pos[r] ← j
8     place( r + 1 , pos )
9
10
```

LISTING 23 – Une première solution algorithmique

Le nombre de solutions est comparable au nombre de permutations. On rappelle que $\log(n!)$ est équivalent à $n \log(n) - n$. Le graphique FIG. (24), 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. (23).

Exercice 29. Compléter la fonction *accept*

Pratique 10 (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.
- (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 $\{0, 1, \dots, 63\}$. 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 :

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

```

1 int mystibit( ensemble X )
2 { int r;
3   for ( r = 0; X & (X-1) ; r++);
4   return r;
5 }

```

Exercice 30 (job). Une fois compris le résultat de la fonction `mystibit`, vous pourrez postuler sur des emplois de bit-programming !

Exercice 31 (décryptage). Décrypter la source du LIST. (??).

Exercice 32 (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 33. Implanter `hamilton(int s, int k, ullong libre)` pour compter le nombre de chemin hamiltonien dans un graphe.

Pratique 11. 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 !

Exercice 34 (black & white). Montrer que si $n > 1$ est impair alors le graphe du cavalier sur un échiquier $n \times n$ n'est pas hamiltonien

Exercice 35 (borderline). Montrer qu'il n'existe pas de cycle Hamiltonien dans le graphe du cavalier d'un échiquier $4 \times n$.

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 typedef unsigned long long int ullong;
4 int n, count = 0;
5 void qbit( int pos, int left , int right, int row )
6 {
7     ullong x, y;
8     if ( ! row ) { count++; return; }
9     row--;
10    left <<=1; right >>=1;
11    x = pos & (~left) & (~right);
12    while ( x ) {
13        y = x;
14        x &= (x - 1);
15        y = x ^ y;
16        qbit( pos & (~y), left | y, right | y , row );
17    }
18 }
19 int main( int argc, char* argv[] )
20 {
21     int n = atoi( argv[1] );
22     ullong t = 1;
23     t = (t << n) - 1;
24     qbit ( t, 0, 0, n);
25     printf ( "\nQ(%d)=%d\n", n, count );
26     return 0;
27 }

```

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 sensible 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 36. *Montrer que le temps de calcul de l'algorithme est exponentiel dans le pire des cas.*

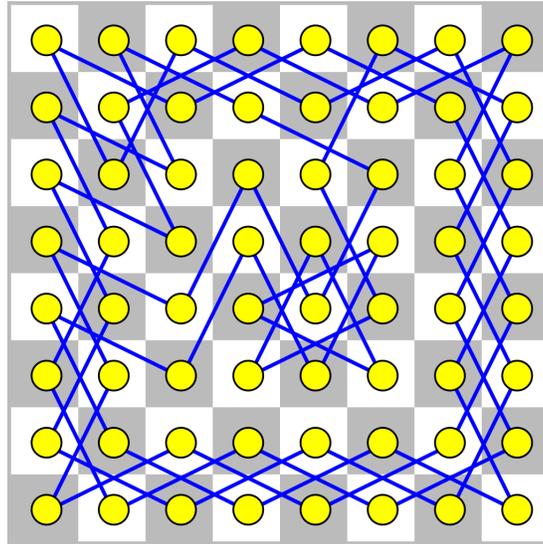


FIGURE 25 – Un remarquable tour de cavalier dû à Euler.

```

1 hamiltonien( s : sommet, k : entier )
2 si k = n alors
3   traitement()
4   retourner
5 pour chaque voisin t de s
6   si libre[ t ] alors
7     libre[ t ] ← faux
8     soluce[ k ] ← t
9     hamiltonien( t , k+1 )
10  libre[ t ] ← vrai

```

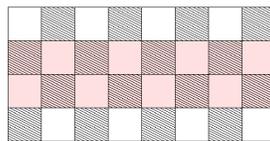


FIGURE 26 – Le graphe du cavalier dans l'échiquier $4 \times n$ n'est pas hamiltonien.

Exercice 37. *Implanter `int maxstab(graphe g)` pour déterminer la taille maximale d'un stable.*

```

1 maxStable( graphe G )
2   si m(G) = 0 retourner n(G)
3   s ← un sommet non isole de G
4   I ← retirerSommet( {s}, G )
5   i ← maxStable( I );
6   V ← les voisins de s dans G
7   J ← retirerSommet( V, I )
8   j ← maxStable( J )
9   retourner ...

```

TABLE 2 – Quelques nombres de Ramsey

	1	2	3	4	5
1	1				
2	1	2			
3	1	3	6		
4	1	4	9	18	
5	1	5	14	25	43–48

Exercice 38. 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 stable ?

Proposition 9 (Ramsey). Pour tout couple d'entiers (r, s) il existe un rang $R(r, s)$ à partir duquel tout graphe d'ordre $n \geq R(r, s)$ contient une clique d'ordre r ou un stable d'ordre s .

Démonstration. Montrer que $R(r, s) \leq R(r, s - 1) + R(r - 1, s)$. \square

Exercice 39. Comment remplir la table 2 ?

8. ZOOLOGIE

8.1. Adjoint et complémentaire.

Définition 1 (graphe complémentaire). Le graphe complémentaire d'un graphe $G(X, U)$ est le graphe dont l'ensemble des sommets est X et dont l'ensemble des arêtes est précisément le complémentaire de U .

Exercice 40. Que peut-on dire du diamètre du complémentaire d'un graphe non connexe ?

Pratique 12. Ecrire une fonction `graphe compl(graphe G)` qui retourne le complémentaire du graphe G .

Définition 2 (graphe adjoint). Le graphe adjoint (line graphe) d'un graphe $G(X, U)$ est le graphe d'incidence des arêtes de G . L'ensemble de ses sommets est U , une arête du graphe correspond à deux arêtes incidentes dans G .

Exercice 41. Exprimer le degré d'une arête rs dans $L(G)$ en fonction des degrés de ses extrémités r et s dans G .

Exercice 42. Montrer que si G est connexe alors $L(G)$ est connexe.

Exercice 43. Montrer que si G est eulérien alors $L(G)$ est eulérien.

Pratique 13. Ecrire une fonction `graphe adjoint(graphe G)` qui retourne l'adjoint du graphe G .

Exercice 44. Montrer que l'adjoint d'un graphe eulérien est hamiltonien. La réciproque est fautive. Donner un contre-exemple.

8.2. Permutation. On note $S(n)$ le groupe des permutations de l'ensemble $\{1, 2, \dots, n\}$. Les $n!$ éléments de $S(n)$ sont représentables par des tableaux de taille n . On dispose de la jolie formule de James Stirling pour approximer le nombre de permutation :

$$n! \sim \sqrt{2\pi n} n^n e^{-n} \quad (\text{Stirling})$$

Etant donnés r éléments a_1, a_2, \dots, a_r la permutation qui transforme a_1 en a_2, a_2 en a_3, \dots, a_r en a_1 est un cycle de longueur r noté $(a_1 a_2 \dots a_r)$. L'ensemble des a_i forme le support du cycle. Un cycle de longueur 2 est une transposition.

Exercice 45. Ecrire le cycle $(abcde)$ en un produit de transpositions.

Exercice 46. Le groupe $S(n)$ n'est commutatif. Que peut on dire de deux cycles à supports disjoints ?

On rappelle qu'une permutation possède une décomposition en produit de cycles à support disjoints et que l'ordre de la permutation est égal au plus commun multiple des longueurs des cycle de la décomposition.

Exercice 47. Observer le code et le script qui suivent. Commenter le résultat de ce script.

<pre> 1 #include <stdio.h> 2 #include <stdlib.h> 3 </pre>	<pre> 4 void doit(int t[], int n) 5 { int i; 6 for(i = 0 ; i < n; i++) </pre>
---	--

```

7   printf ( "%3d", t[i] );
8   putchar( '\n' );
9
10  }
11
12  int pi( int p, int t[], int
      n )
13  {
14  if ( p == n - 1 ) {
15      doit( t, n); return;
16  }
17  int i , tmp;
18  for( i = p ; i < n; i++ ) {
19      tmp = t[p];
20      t[p] = t[i] ;
21      t[i] = tmp ;
22      pi( p+1, t, n );
23      tmp = t[p] ;
24      t[p] = t[i] ;
25      t[i] = tmp ;
26  }
27  }
28
29  int main( int argc , char *
      argv[] )
30  {
31  int n = atoi( argv[1] );
32  int i, t[ 16 ];
33  for( i = 0; i < n; i++ ) t[
      i] = i;
34  pi( 0 , t, n );
35  return 0;
36  }

```

```

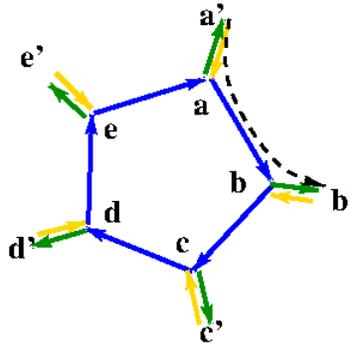
1 :: gcc pi.c
2 :: for x in {1..10};
3   do /usr/bin/time --format=" $x cpu=%U" ./a.out $x \
4     | grep 0 | sort | uniq -c | wc -l; done

```

1 1 cpu=0.00	8 24	15 8 cpu=0.02
2 1	9 5 cpu=0.00	16 40320
3 2 cpu=0.00	10 120	17 9 cpu=0.20
4 2	11 6 cpu=0.00	18 362880
5 3 cpu=0.00	12 720	19 10 cpu=2.20
6 6	13 7 cpu=0.00	20 3628800
7 4 cpu=0.00	14 5040	

Exercice 48 (décomposition-ordre). *Coder une fonction pour calculer l'ordre d'une permutation. Préciser le mode d'emploi en fonction de n*
...

La conjugaison est une opération qui facilite l'analyse des groupes non commutatifs. Rappelons que $f \circ s \circ f^{-1}$ est la permutation conjuguée de s par la permutation f .

FIGURE 27 – conjugaison d'un cycle $f\sigma f^{-1}$.

Lemme 3 (conjugaison). *Le conjugué d'un cycle de longueur $r > 0$ est un cycle de longueur r . Plus précisément, le conjugué du cycle $(a_1 a_2 \dots a_r)$ par une permutation π est égal au cycle $(\pi(a_1) \pi(a_2) \dots \pi(a_r))$.*

Exercice 49. *Montrer que l'ensemble des transpositions est un système générateur de $S(n)$ permutations.*

Exercice 50. *Montrer que les transpositions de la forme $(i, i + 1)$ engendrent toutes les permutations.*

Exercice 51. *Montrer que le groupe des permutations peut être engendré par les deux éléments $\tau : (12)$ et $\sigma : (12 \dots n)$.*

8.3. Isomorphisme de graphe. Un isomorphisme du graphe $G(X, U)$ sur le graphe $G'(X', U')$ est une bijection $f: X \rightarrow X'$ qui conserve l'adjacence : toute arête $xy \in U$ est transformée en une arête $f(x)f(y) \in U'$. Le problème d'isomorphisme de graphe qui consiste à décider de l'existence d'un isomorphisme entre deux graphes est un des rares problèmes NP dont on ne sait pas (encore) dire s'il est polynomial ou bien NP-complet.

Les isomorphismes du graphe $G(X, U)$ sur lui même forment un sous groupe du groupe des permutations de l'ensemble des sommets X , on le note $\text{AUT}(G)$.

Lemme 4 (classe). *Le nombre de graphes de sommets X qui sont isomorphes à G est égal au quotient de $n!$ par $\text{AUT}(G)$.*

Démonstration. □

Notons $\text{CLS}(n)$ le nombre de classes d'isomorphie de graphe d'ordre n . Pour une permutation π , on note $\text{FIX}(\pi)$ le nombre de graphes fixés par la permutation π . La célèbre formule de Burnside donne :

Exercice 56 (invariant distribution). *Il existe deux classes de graphes d'ordre 5 admettant la distribution : 1 sommet de degré 1, 3 sommets de degré 2 et 1 sommet de degré 3. Dessinez !*

La distribution des degrés d'un graphe d'ordre n ayant m arêtes forme une "partition" de l'entier $2m$ en n parts. Si n_k désigne le nombre de sommets de degré k alors

$$\sum_{k=0}^n n_k k = 2m, \quad n = \sum_{k=0}^n n_k.$$

Dans la littérature, une partition de n est une suite croissante de n entiers strictement positifs $(n_k)_{k=1}^n$ telle que :

$$\sum_{k=1}^n n_k k = n.$$

Pratique 15 (énumération de partition). *Coder une fonction récursive `int enumpart(n)` pour énumérer toutes les partitions d'un entier n .*

On note $P(n, k)$ le nombre de partitions de n en k parties. Établir que $P(n, n) = P(n, 1) = 1$, $P(n, k) = 0$ si $k > 0$ et dans les autres cas :

$$P(n, k) = P(n - 1, k - 1) + P(n - k, k)$$

Pratique 16 (dénombrement de partition). *Coder une fonction récursive `int partcpt(n, k)` pour calculer $P(n, k)$.*

8.5. Classification.

Proposition 10. *Le groupe \mathfrak{S}_n peut être engendré une transposition et un cycle.*

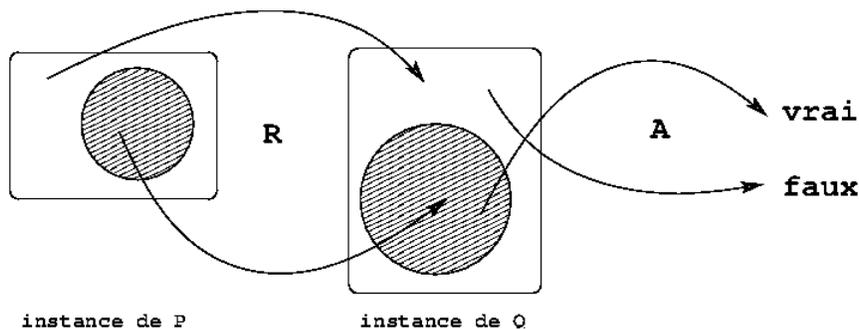
Démonstration. □

Pour une petite valeur de n , on peut représenter les graphes d'ordre n par des mots binaires. On peut utiliser la proposition précédente pour parcourir l'orbite d'un graphe G par un parcours en profondeur.

Pratique 17 (orbite). *Écrire une fonction `void repres(int n)` qui énumère toutes les classes d'isomorphies en précisant la taille des orbites et les ordres des stabilisateurs.*

9. RÉDUCTION

Nous disposons d'un algorithme rapide pour décider de l'existence d'un cycle eulérien dans un graphe, et d'un algorithme exponentiel pour décider de son caractère hamiltonien ou pas. Comment prouver que le second problème est réellement plus difficile que le premier ?

FIGURE 29 – schéma d’une réduction R de \mathcal{P} vers \mathcal{Q} .

9.1. réduction polynomiale. Un problème de décision \mathcal{P} se réduit polynomialement à un problème de décision \mathcal{Q} quand il existe un algorithme polynomial R qui transforme les instances positives de \mathcal{P} en des instances positives de \mathcal{Q} , et les instances négatives de \mathcal{P} en des instances négatives de \mathcal{Q} . Dans cette situation, par composition, d’un algorithme A de résolution du problème \mathcal{Q} on déduit un algorithme de résolution du problème \mathcal{P} .

Remarque 4. *L’existence d’une réduction polynomiale de \mathcal{P} vers \mathcal{Q} permet de résoudre les instances de \mathcal{P} au moyen d’un algorithme de résolution du problème \mathcal{Q} . Autrement dit, le problème \mathcal{P} est théoriquement plus facile que le problème \mathcal{Q} .*

9.2. Euler vs Hamilton. Considérons le graphe de la figure FIG. (30) composé de 6 sommets de différentes couleurs reliés par 7 arêtes le graphe est semi-eulérien.

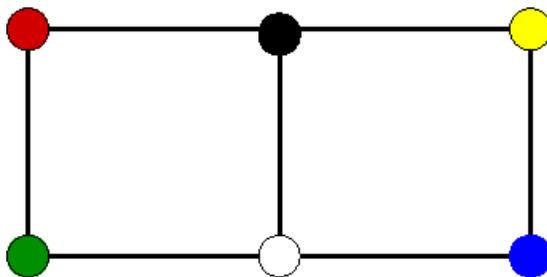


FIGURE 30

Par définition, un chemin dans un graphe est une suite de sommets, mais c’est aussi une suite d’arêtes. Ainsi, pour un chemin :

$$a - b - c - d - e - f - g - h$$

dans un graphe G peut être vu comme un chemin

$$a - ab - b - bc - c - cd - d - de - e - ef - f - fg - g - gh - h$$

dans un graphe G' dont les sommets sont à la fois des arêtes et des sommets de G .

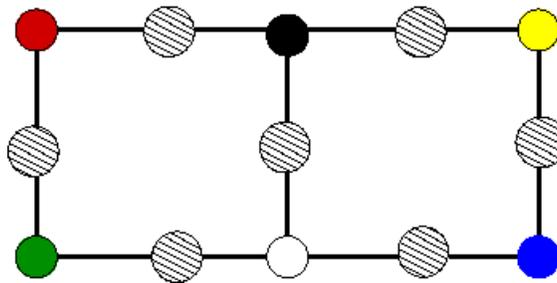


FIGURE 31

Considérons la transformation d'un graphe $G(S, A)$ en un graphe H dont les sommets sont obtenus à partir des sommets et des arêtes de G . Chaque arête de G est une arête de H . À chaque sommet s de degré d dans G est associée une clique C_s d'ordre d . On ajoute des arêtes dans H de sorte que chacune des arêtes incidentes à un sommet s dans G soit adjacente à un et un seul des sommets de la de c_s .

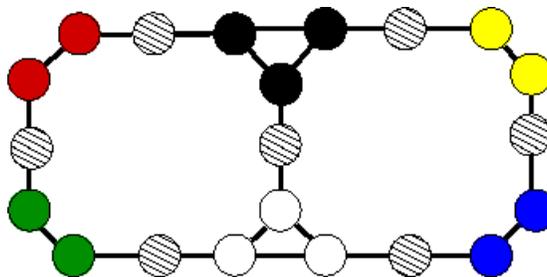


FIGURE 32

Remarque 5. *La théorie de la complexité montre que tout problème d'une classe plus large que la classe polynomiale se réduit au problème du cycle hamiltonien.*

Exercice 57. *Comment réduire le problème de la connexité au problème du cycle hamiltonien ?*

9.3. Classe Non déterministe Polynomiale. La classe **NP** est constituée des problèmes de décision dont on peut vérifier une solution en temps polynomial.

Le problème du cycle eulérien est dans la classe **P**. Comme tous les problèmes de la classe **P**, le problème est aussi dans la classe **NP**. Le problème du cycle hamiltonien est dans la classe **NP** car il est facile de vérifier quand un chemin est un cycle hamiltonien. Une des questions les plus importante posée par l'informatique théorique se résume à :

Existe-t-il un algorithme polynomial pour résoudre le problème du cycle hamiltonien ?

9.4. Problème NP-complet. Un problème \mathcal{Q} de la classe **NP** est dit **NP-complet** si pour tout problème de la classe **NP**, il existe une réduction vers \mathcal{Q} . Le théorème de Cook est le point de départ de la théorie de la complexité en informatique théorique. La preuve est hors au programme, quelques conséquences concerne notre sujet.

Théorème 3 (Cook, 1972). *Le problème 3-SAT est NP-complet.*

Démonstration. Voir le cours de Jean-Pierre Zanotti. □

Il faut expliquer 3-SAT. Soient x_1, x_2, \dots, x_n un ensemble de variables logiques. Les formules x_i et \bar{x}_i sont des littéraux. Une clause est une disjonction de littéraux distincts. Le nombre de littéraux qui compose une clause est le poids. Une conjonction de clauses, est une forme normale conjonctive.

Problème 6 (k -SAT). *Etant donné une forme normale conjonctive Φ dont toutes les clauses sont de poids inférieur ou égal à k . Existe-t-il une solution au problème $\Phi(x_1, x_2, \dots, x_n)$?*

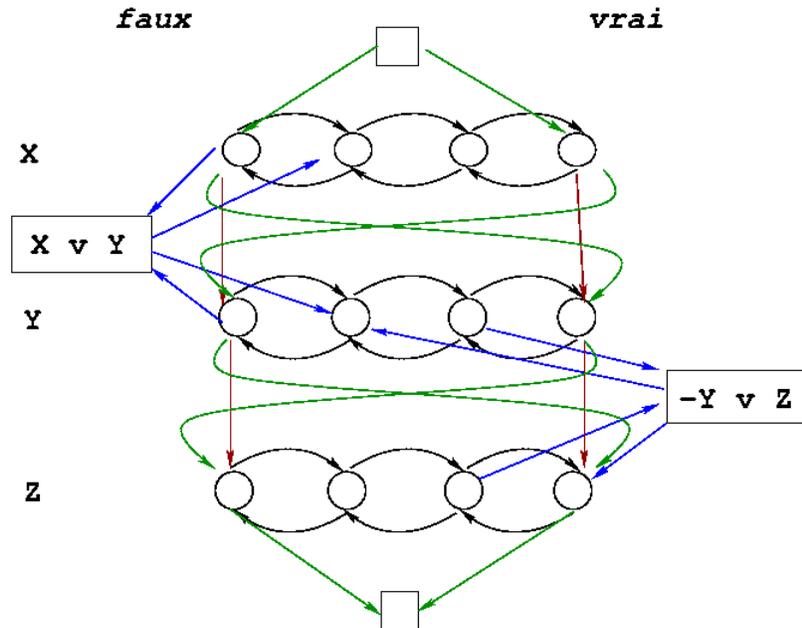
Remarque 6. *Le problème 2-SAT est dans la classe **P**. Un des objectifs de ce cours est de vérifier ce point !*

9.5. Gadget.

Théorème 4. *Le problème du cycle Hamiltonien est NP-complet.*

9.6. Primalité. L'histoire du problème de décision PREMIER mérite le détour. Tout d'abord le problème opposé i.e. le problème COMPOSÉ est trivialement dans la classe **NP**. En effet, pour vérifier qu'un nombre est composé, il suffit de fournir deux facteurs appropriés.

Proposition 11 (V. Pratt, 1975). *Le problème PREMIER est dans la classe **NP**.*

FIGURE 33 – gadget de réduction 3-SAT $\langle \text{HAM}^\dagger \rangle$.

Démonstration. Pour certifier qu'un entier n est premier, il s'agit principalement de fournir un résidu d'ordre $n - 1$. Je renvoie le lecteur vers un de mes livres favoris [18]. \square

On déduit du résultat de Pratt que le problème PREMIER est dans à la fois dans la classe **NP** et dans la classe **co-NP**. Notons que c'est bien le cas de tous les problèmes polynomiaux. Une trentaine d'années plus tard, A. Manindra, K. Neeraj et S. Nitin recevront le prix Gödel pour avoir établi que ce problème de décision est bien dans la classe P.

10. ARBRE COUVRANT MINIMAL

10.1. 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...

Exercice 58 (détection de cycle). *Modifier le parcours récursif en profondeur pour détecter la présence d'un cycle dans un graphe.*

Exercice 59 (coloration-cycle de longueur impair). *Comment détecter la présence d'un cycle de longueur impair dans un graphe.*

Théorème 5 (Big Equivalence on Tree). *Soit $G(S, A)$ un graphe d'ordre n à m arêtes. Les 6 assertions qui suivent sont équivalentes :*



MR2123939 (2006a:11170)

Reviewed

[Back to search](#) | [Previous](#) | [N](#)

Agrawal, Manindra (6-IITK-CSE); Kayal, Neeraj (6-IITK-CSE); Saxena, Nitin (6-IITK-CSE)

PRIMES is in P. (English summary)

Ann. of Math. (2) **160** (2004), no. 2, 781–793.

See erratum in: [MR3898710](#)

Classifications

[11Y11](#) - Primality

[11A51](#) - Factorization; primality

[68Q15](#) - Complexity classes (hierarchies, relations among complexity classes, etc.)

[68Q25](#) - Analysis of algorithms and problem complexity

Citations

From References: 221

FIGURE 34 – PREMIER est dans la classe P !

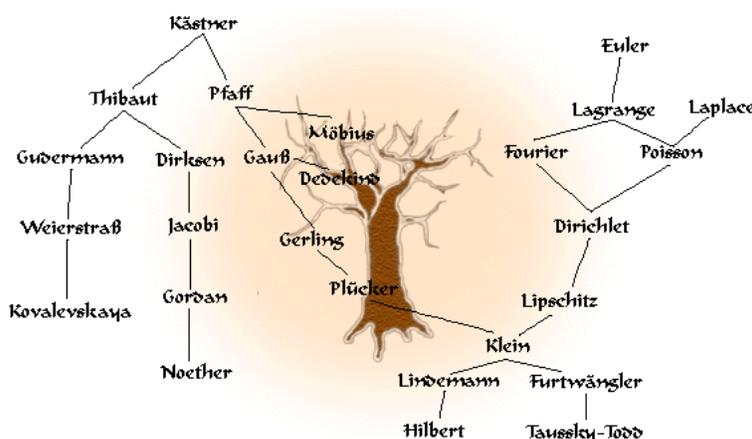


FIGURE 35 – La décoration de l'arbre *mathematics genealogy project* inspire le respect mais ce n'est pas un arbre au sens de la théorie des graphes.

- (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.

Démonstration. Les plus jolies preuves sont souvent les plus courtes ! Montrons par exemple, que (1) implique (3). On procède par induction

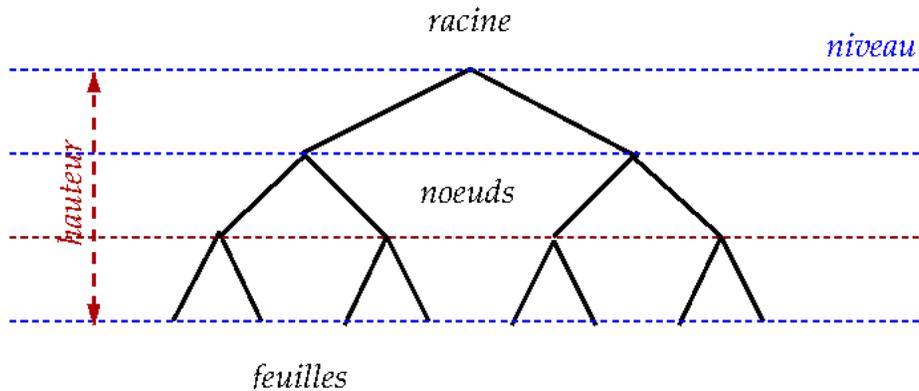


FIGURE 36 – Un arbre binaire parfait de hauteur 4 à 15 noeuds organisés sur 4 niveaux et 8 feuilles au niveau 3. Conformément à l’usage algorithmique, la racine est en l’air sur le niveau 0!

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) $n - 2$. \square

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

Proposition 12. *Un graphe possède un arbre couvrant si et seulement s’il est connexe.*

Exercice 60 (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 $c(n) := n - 1 + \lceil \log_2 n \rceil - 1$ suffisent pour résoudre le problème des deux plus grandes valeurs. Comment ?*

```

1 PPV( t : tableau )
2 objet a, b, c, d
3 si ( t = 2 ) alors
4     si t[0] > t[1] alors t[0] ↔ t[1]
5     retourner (t[0], t[1] )
6     fsi
7 fsi
8 (a, b) ← PPV ( gauche t )
9 (c, d) ← PPV ( droite t )
10 si a < c alors
11     retourner c < b ? (a, c) : (a, b)
12 sinon
13     retourner a < d ? (c, a) : (c, d)
14
15 fsi

```

LISTING 24 – Diviser pour régner

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.

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

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

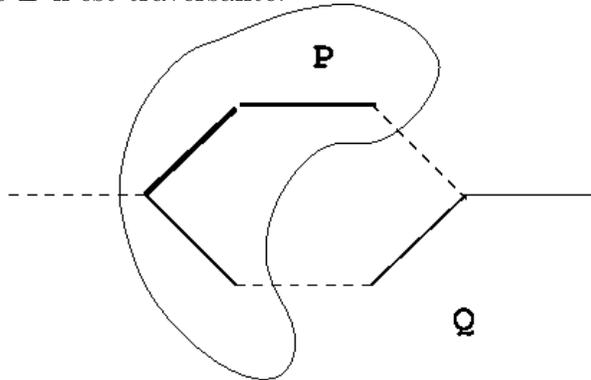
Plus généralement, le coût d'un graphe partiel est la somme des poids de ses arêtes.

Problème 7 (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.

10.3. 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 6. (*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 .

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. \square

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

10.4. 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 6. Le temps de calcul de l'algorithme de Kruskal sur un graphe connexe d'ordre n et m arêtes est $O(m \log m + m \log n)$.

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 structure d'ensembles disjoints. \square

```

1 KRUSKAL( G, w )
2 pour chaque sommet s
3   singleton(s)
4 r ← 0
5 pour chaque arete xy par ordre croissant
6   si representant(x) != representant(y)
7   alors union(x, y )
8     r ← r + w(xy)
9   fsi
10 retourner r

```

LISTING 25 – algorithme de Kruskal

```

1 typedef struct {
2   int x, y; float w;
3 } arc;
4
5 static int cmp(const void *p1, const void *p2)
6 {
7   if ( ((arc*) p1)->w < ((arc*)p2)->w ) return -1;
8   if ( ((arc*) p1)->w > ((arc*)p2)->w ) return +1;
9   return 0;
10 }
11 //...
12 qsort(t, m, sizeof(arc), & cmp );

```

LISTING 26 – usage de `qsort`

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

Théorème 7. *Le temps de calcul de l'algorithme de Prim sur un graphe connexe d'ordre n et m arêtes est $O(n \log n + m \log n)$.*

Démonstration. L'extraction d'un élément prioritaire, la mise à jour d'une file de priorité sont de complexité $O(\log n)$. Le second terme compte globalement la découverte et la mise à jour des priorités des voisins. \square

```

1 JARNIK-PRIM( G, w )
2 r ← 0
3 initialiser une file de priorite
4 p ← ordre( G )
5 tant que p > 1
6   s = prioritaire()
7   r ← r + cle [ s ]
8   pour chaque t voisin de s
9     misajour( s, t )
10  p ← p-1
11 retourner r

```

LISTING 27 – algorithme de Jarnik-Prim

```

1 DIJKSTRA( G, w, s )
2 E ← ensemble vide
3 F ← sommets de G
4 d ← [+inf, +inf, ..., +inf]
5 d [s] ← 0
6 tantque non vide ( F )
7   u ← extraire( E, F )
8   retirer u de F
9   ajouter u a E
10  pour chaque voisin v de u
11    relacher( u, v, w )

```

LISTING 28 – algorithme de Dijkstra

Exercice 61. On considère les instances de graphe d'ordre n avec $f(n)$ arêtes. Quel choix faire entre Prim ou Kruskal en fonction de f ? Attention aux pièges...

$\log n$	λn	$n \log n$	$n\sqrt{n}$
?	?	?	?

Remarque 1 (Lathuile). La structure de tas de Fibonacci (hors programme) de Michael Fredman et Robert Tarjan publiée en 1987 permet de réaliser des mises à jour efficaces sur la file de priorité, pour obtenir une complexité arêtes est $O(n \log n + m)$.

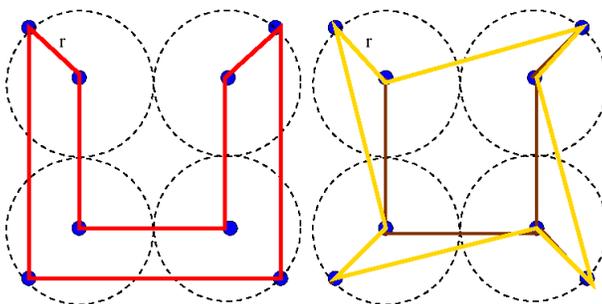
10.6. **algorithme de Dijkstra.** L'algorithme de Dijkstra un parfait analogue de l'algorithme de Prim permet de calculer les plus court chemins d'origine donnée pour une pondération positive.

Exercice 62. *Expliciter la fonction $relacher(u, v, w)$ de l'algorithme Dijkstra.*

L'algorithme de Strassen utilise le principe "diviser pour régner" pour calculer le produit de deux matrices carrées. Il est de complexité $\Theta(n^{2.81})$ sur les matrices $n \times n$.

Exercice 63 (triangle). *Comment compter le nombre de triangle d'un graphe ?*

10.7. **Approximation.** Le problème du voyageur de commerce est un problème d'optimisation qui consiste à déterminer une tournée optimale dans un graphe pondéré. Il s'agit donc de déterminer cycle hamiltonien de coût minimal. Notons T^* une tournée optimale, de coût $wt(T^*)$. Le problème est NP-dur et pour $\mathbb{R} \ni \alpha \geq 1$, on peut être satisfait de déterminer une α -approximation i.e. une tournée de coût au plus $\alpha wt(T^*)$.



Théorème 8. *Le parcours raccourci d'un ACM d'un graphe métrique complet est une 2-approximation du problème du voyageur de commerce.*

Démonstration.

□

Pratique 18. — *Implanter la 2-approximation du Théorème 8.*
— *Appliquer aux nuages de points.*

Exercice 64. *Comment améliorer une tournée dans le plan Euclidien quand deux de ses arêtes sont sécantes ?*

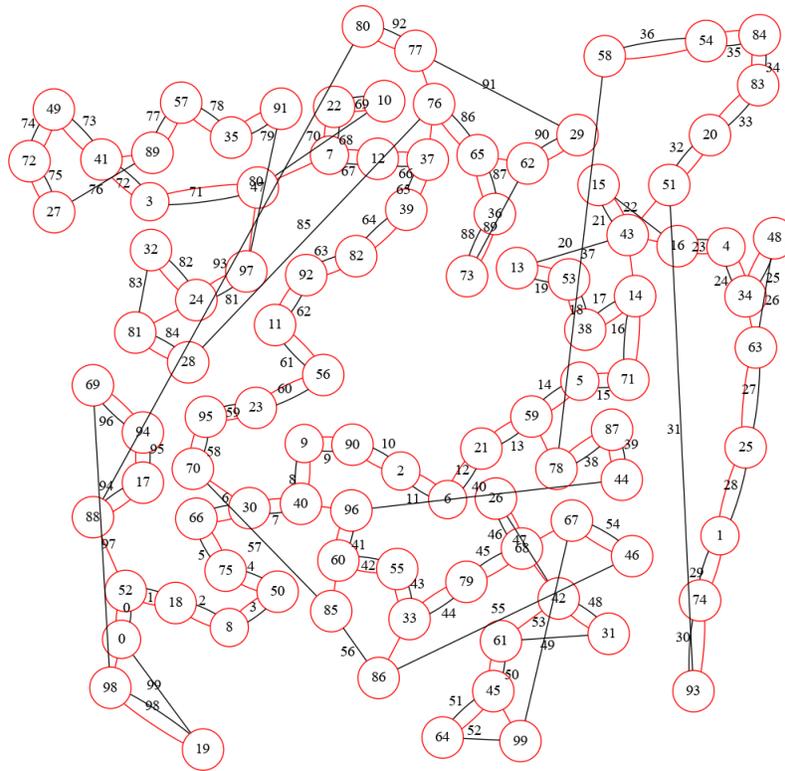


FIGURE 37 – Un exemple de 2-approximation sur un nuage de 100 points, les arêtes de l'ACM sont rouges, celles de la tournée sont en noire (Stéphane Rossi)

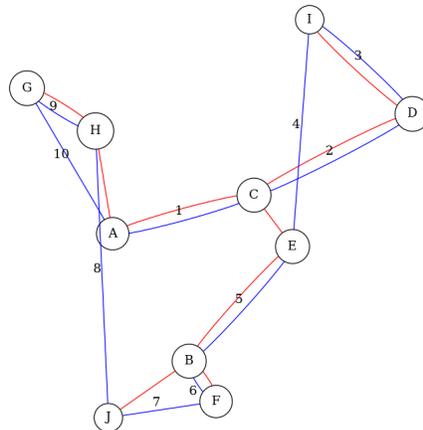


FIGURE 38 – Un exemple de 2-approximation sur un nuage de 10 points. En rouge, un arbre couvrant minimal, la tournée en bleue (Alexandre Jaillant).

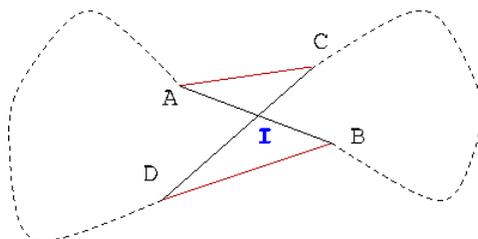


FIGURE 39 – Avec le croisement des segments $[AB]$ et $[CD]$, la tournée est améliorable... Exercice !

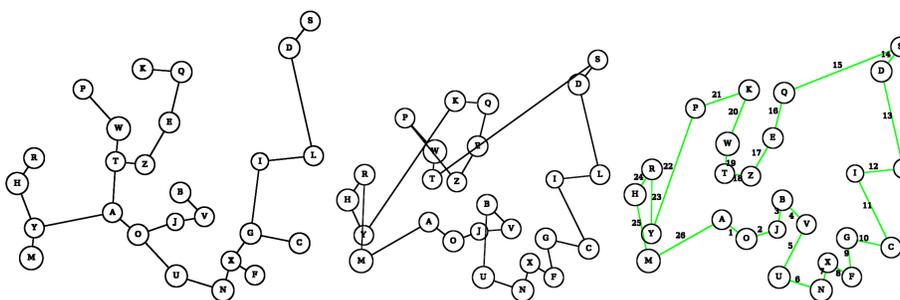


FIGURE 40 – Amélioration de la tournée par élimination des croisements (Alexandre Jaillant).

10.8. $\frac{3}{2}$ -**approximation de Christophides**. Au milieu des années 70, Nicos Christophides a proposé un algorithme de $\frac{3}{2}$ -approximation pour le graphe complet métrique. Un couplage d'un graphe est un ensemble d'arêtes deux à deux non incidentes. Un couplage de cardinal maximal est dit maximal. Il est parfait quand il recouvre tous les sommets. Un couplage optimal d'un graphe pondéré est un couplage maximal qui minimise le coût. L'algorithme des fleurs et des pétales d'Edmonds (1965) est un algorithme polynomial (hors programme) qui détermine un couplage maximal dans un graphe qui peut être adapté pour déterminer un couplage optimal dans un graphe pondéré.

Exercice 65. On considère la procédure suivante appliquée à un graphe complet métrique $G(S, U)$ dont une tournée optimale est notée T .

- (1) Déterminer les arêtes Y d'un arbre couvrant minimal G .
- (2) Comparer $wt(Y)$ et $wt(T)$.
- (3) L'ensemble I des sommets impairs de Y est pair. vrai ou faux ?
- (4) On note G' le graphe induit sur I .
- (5) G' possède un couplage parfait. Pourquoi ?

- (6) On note M un couplage parfait optimal de G' . Comparer $\text{wt}(M)$ et $\text{wt}(T)$.
- (7) On note G'' le graphe partiel $G''(S, Y \cup M)$.
- (8) Le graphe G'' est-il connexe ?
- (9) Les sommets de G'' sont tous pairs. Pourquoi ?
- (10) G'' possède un cycle eulérien E . Pourquoi ?
- (11) Comparer $\text{wt}(M) + \text{wt}(Y)$ et $\text{wt}(E)$.
- (12) On $\text{wt}(E) \leq \frac{3}{2}\text{wt}(Y)$. Pourquoi ?
- (13) Comment obtenir une tournée R à partir de E ?
- (14) Comparer $\text{wt}(R)$, $\text{wt}(E)$ et $\text{wt}(T)$.

11. COLORIAGE DES SOMMETS

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

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

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

Démonstration. Un bon exercice! □

Problème 8 (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.

11.2. Coloriage glouton.

Théorème 9 (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 avec la plus petite couleur disponible. □

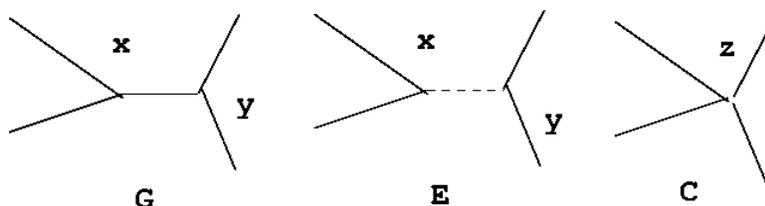


FIGURE 41 – La transformation de Birkhoff. De gauche à droite, les graphes G , E et C .

11.3. Polynôme chromatique.

Théorème 10 (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 :

$$(3) \quad \gamma_G(t) = \gamma_E(t) - \gamma_C(t).$$

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

Exercice 66. *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 67. *Déterminer le polynôme chromatique d'un arbre d'ordre n , d'un cycle d'ordre n .*

11.4. 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 .

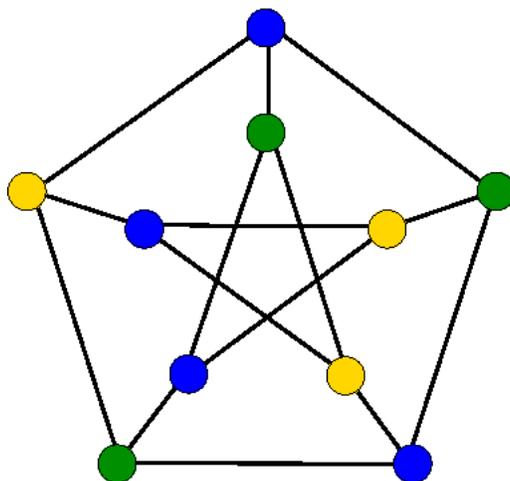


FIGURE 42 – Le polynôme chromatique du graphe de Petersen est : $t^{10} - 15t^9 + 105t^8 - 455t^7 + 1353t^6 - 2861t^5 + 4275t^4 - 4305t^3 + 2606t^2 - 704t$

```

1 coloriage( s, k )
2 si s == nbs alors
3   compteur += 1
4   retourner
5 fsi
6 pour chaque couleur c < k
7   si acceptable( s, c ) alors
8     clr[s] ← c
9     coloriage( s + 1 )
10    clr[s] ← 0
11 fsi

```

Exercice 68 (acceptable). Compléter l'algorithme de coloriage.

Pratique 19. On peut utiliser des piles d'ensembles pour suivre l'évolution des contraintes.

- (1) Planter l'algorithme du calcul des colorations d'un graphe.
- (2) Vérifier que le polynôme chromatique de la maison à 5 sommets est :

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

11.5. lien dansant.

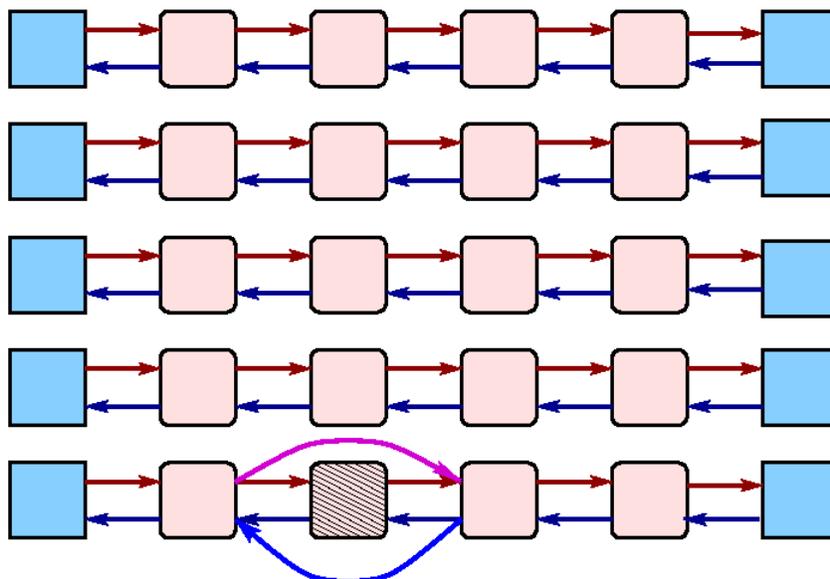


FIGURE 43 – Un réseau de liens dansant pour le coloriage du graphe maison.

```

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

```

LISTING 29 – structure pour les liens dansants

Exercice 69. *Comment coder simplement l'initialisation du réseau de liens ?*

Pratique 20. *Utiliser la technique des liens dansants sur une liste doublement chaînée pour décider k -colorabilité d'un graphe.*

```

1 tps ← 0
2
3 PPROF( s : sommet, g : graphe )
4 d[ s ] ← tps ← tps + 1
5 clr[ s ] ← GRIS
6 pour chaque arc st
7   si clr[ t ] = BLANC
8     PPROF( t )
9 clr[ s ] ← NOIR
10 f[ s ] ← tps ← tps + 1
11
12
13 PARCOURS( G : graphe )
14 pour chaque sommet s
15   clr[ s ] ← BLANC
16 pour chaque sommet s
17   si clr[ s ] = BLANC
18     PPROF( s, g )

```

LISTING 30 – Parcours en profondeur avec horodatage

12. TRI TOPOLOGIQUE

12.1. Parcours en profondeur. Le parcours en profondeur, horodaté et tricolore est très utile pour réaliser des preuves algorithmiques.

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 70 (PP en action). *Préciser les prochaines étapes du parcours horodaté en cours d'exécution sur le graphe FIG. (44).*

Lemme 7 (des intervalles). *Les intervalles $[d(s), f(s)]$ d'un parcours horodaté sont emboîtés ou disjoints.*

Démonstration. 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. \square

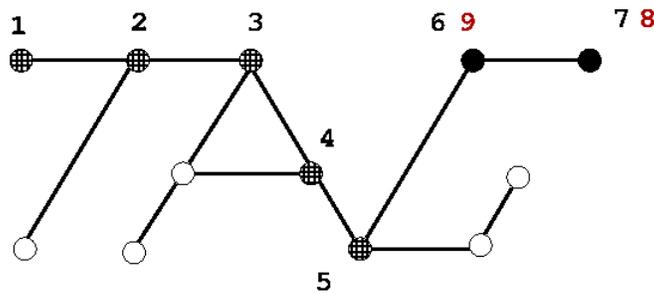
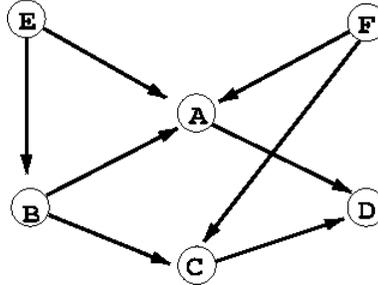


FIGURE 44 – Le parcours en profondeur horodaté en action sur le graphe TAG.

12.2. 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'ordonnement des tâches est une application classique du tri topologique.

Proposition 14. *On obtient un tri topologique d'un graphe acyclique orienté en ordonnant les sommets un ordre de fin de traitement en profondeur.*

Démonstration. □

Exercice 71. *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 72. *Ecrire un algorithme pour détecter la présence d'un circuit dans un graphe orienté.*

Pratique 21. *Implanter la fonction `int* triTopologique(graphe G)` qui retourne un tableau de sommets du graphe orienté G dans un ordre topologique.*

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

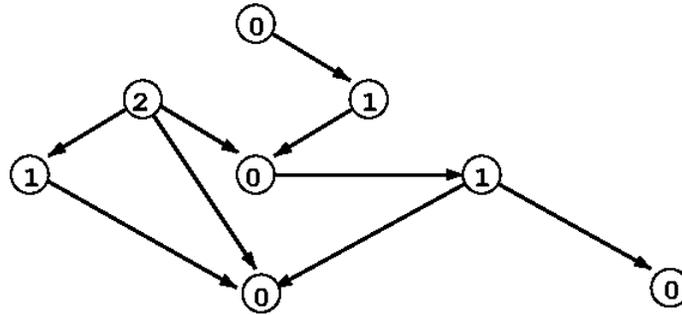


FIGURE 45 – Fonction de Grundy sur un graphe.

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

$$g(s) = \text{mex}\{g(t) \mid s \rightarrow t\}.$$

Proposition 15. *Un graphe orienté avec deux fonctions de Grundy possède un circuit.*

Démonstration. 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. \square

Proposition 16. *Un graphe orienté acyclique possède une et une seule fonction de Grundy.*

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

Pratique 22. *Implanter la fonction `int* Grundy(graphe G)` qui retourne le tableau des valeurs de Grundy des sommets du graphe G .*

12.4. 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 coup légal.

Proposition 17 (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.*

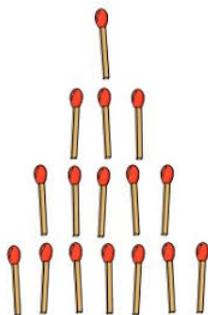


FIGURE 46 – Position initiale du célèbre jeu de nim.

Démonstration. 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 73. *La position initiale est-elle perdante ou gagnante ?*

Exercice 74. *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 ?*

12.5. **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 :

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

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

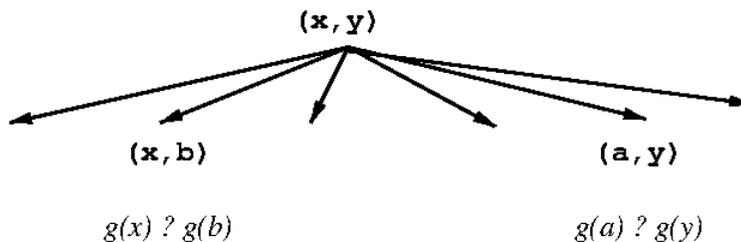


FIGURE 47 – Un nouvel opérateur ?

Lemme 8 (somme de graphe acyclique). *La somme de deux graphes acyclique est un graphe acyclique.*

Démonstration. 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 :

$$(4) \quad x \oplus y = \text{mex}_{\substack{a < x \\ b < y}} \{a \oplus y, x \oplus b\}.$$

Exercice 75. *Montrer que pour tout entiers x et y , $x \oplus y \leq x + y$.*

Exercice 76. *Compléter la table TAB. (4) de la nim-addition.*

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

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

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

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

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

Supposons par exemple que

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

TABLE 3 – 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	$x \oplus b$...	?
\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	...	b	...	y

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. \square

Proposition 19. *La nim somme de x et y n'est rien d'autre que leur XOR bit-à-bit.*

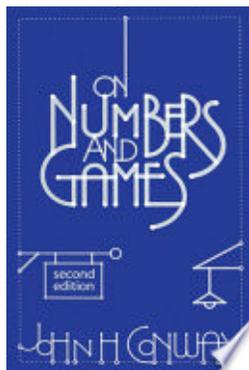
Démonstration. Il suffit d'établir que pour $x = \sum_{i=0}^{r-1} x_i 2^i$:

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

\square

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.*

Démonstration. Par la mex-définition. \square



12.6. **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 :

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

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

Exercice 77. Compléter la table de la nim-multiplication pour les entrées strictement inférieures à 4. Commenter !

Exercice 78. 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.

Démonstration. 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. \square

Exercice 79. Établir deux propriétés au choix de la nim-multiplication.

Pratique 23. Mettre oeuvre la nim-multiplication.

RÉFÉRENCES

- [1] Claude Berge. *Graphes. μ_B* . Dunod, Paris, third edition, 1983.
- [2] John H. Conway. *On Numbers and Games*. A. K. Peters, 2 edition, 2000.
- [3] T. H. Cormen, C. Leiserson, R. Rivest, and X. Cazin. *Introduction à l'algorithmique*. Science informatique. Dunod, 1994.
- [4] Lucas Édouard. *Récréations mathématiques—tome 1—les traversées. les ponts. les labyrinthes*, 1882. <http://gallica.bnf.fr>.
- [5] Noam Elkies and Richard Stanley. The mathematical knight, 200x. <http://www-math.mit.edu/~rstan/papers/knight.pdf>.
- [6] 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.
- [7] 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.

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