



Ministère de l'Education Nationale
Université de Montpellier II
Place Eugène Bataillon
34095 Montpellier Cedex 5



TP FMIN105

Algorithmique / Complexité / Calculabilité

RAPPORT (DÉCEMBRE 2011)

Travail préparé par :

Thibaut MARMIN
Clément SIPIETER
William DYCE

<https://github.com/marminthibaut/acc-tp>

Table des matières

1	Partie théorique	5
1.1	Algorithmique	5
1.1.1	Fonction chromatique	5
1.1.2	Des cas un peu spéciaux...	5
1.1.3	Décomposition de P_G	6
1.1.4	Polynôme chromatique?	6
1.1.5	Application	7
1.1.6	Coefficients alternatifs	8
1.1.7	Application à $K_{1,5}$	8
1.1.8	Application aux graphes non connexes	8
1.1.9	Application aux arbres	8
1.1.10	$k^5 - 4k^4 + 6k^3 - 4k^2 + k$	9
1.1.11	Application à $K_{2,5}$	9
1.1.12	Application aux cycles	9
1.1.13	Démonstration sur les cycles	9
1.1.14	Démonstration sur les graphes bipartis complets	9
1.2	Complexité	10
1.2.1	SAT \propto 3-SAT	10
1.2.2	3-SAT \propto 2-SAT?	13
1.2.3	2-SAT, un problème polynomial	14
1.3	Calculabilité	17
1.3.1	Énumération des couples d'entiers	17
1.3.2	Codons et décodons...	17
1.3.3	Énumération des triplets d'entiers	18
1.3.4	Énumération de l'ensemble $[0; 1]$	18

2	Partie pratique	21
2.1	Objectifs	21
2.2	Spécification fonctionnelles	21
2.3	Spécification technique	21
2.3.1	Choix du langage de programmation	21
2.3.2	Structures	21
2.3.3	Representation d'un graphe	22
2.3.4	AdjacencyListGraph	23
2.3.5	MatrixGraph	25
2.3.6	Génération aléatoire d'un graphe	26
2.3.7	Dinic	28

Chapitre 1

Partie théorique

1.1 Algorithmique

1.1.1 Fonction chromatique

Le nombre de manières de colorier un graphe est le produit des nombres de façons de colorier chaque arc.

- Si le graphe G est complet, on aura k couleurs possibles pour le premier sommet, $(k - 1)$ pour le deuxième, etc... (Le graphe G étant complet, la couleur du premier sommet est nécessairement exclu des autres sommets) Le n^{e} sommet pourra être colorié de $k - (n - 1)$ manières. D'où :

$$P_{K_n}(k) = \prod_{i=0}^{n-1} (k - i)$$

- Si G est vide, la coloration d'un sommet ne contraint pas la coloration des autres sommets. On obtient alors :

$$P_{\overline{K_n}}(k) = k^n$$

1.1.2 Des cas un peu spéciaux...

$\chi(G)$ étant, par définition, le nombre minimum de couleurs nécessaires pour colorier G , si $k < \chi(G)$ alors le graphe G ne peut pas être colorié par k couleurs. Si $k \geq \chi(G)$ alors il doit y avoir au moins une manière de colorier G , celui utilisant $\chi(G)$ couleurs.

On a donc :

$$P_G(k) \begin{cases} = 0 & \text{si } k < \chi(G) \\ \geq 1 & \text{sinon} \end{cases}$$

1.1.3 Décomposition de P_G

Montrons d'abord que la propriété est vraie pour tout graphe complet K_n . Pour commencer on remarque que, pour tout arrête e :

- $K_{n \setminus e}$ est exactement K_{n-1} , et donc :

$$P_{K_{n \setminus e}}(k) = P_{K_{n-1}} = \prod_{i=0}^{n-2} (k - i)$$

- Soit $e = (a, b)$. On peut supposer (sans perte de généralité) que b est considéré en dernier lors de la coloration de K_n , donc qu'il lui reste $k - (n - 1)$ couleurs. Pour colorier K_{n-e} on aura un choix de plus pour lui, à savoir la couleur de a , donc $k - (n - 2)$ en totale. De ce fait :

$$P_{K_{n-e}}(k) = P_{K_{n-1}}(k)(k - (n - 2)) = \left(\prod_{i=0}^{n-2} (k - i) \right) (k - (n - 2))$$

On a donc très clairement :

$$\begin{aligned} P_{K_{n-e}}(k) - P_{K_{n \setminus e}}(k) &= \left(\prod_{i=0}^{n-2} (k - i) \right) (k - (n - 2)) - \prod_{i=0}^{n-2} (k - i) \\ &= \prod_{i=0}^{n-2} (k - i) (k - (n - 1)) \\ &= \prod_{i=0}^{n-1} (k - i) \\ &= P_{K_n}(k) \end{aligned}$$

Tout graphe de rang n pouvant se générer à partir de K_n (en enlevant des arrêtes) on cherchera à prouver que la suppression d'arrête conserve notre propriété. Autrement dit on aimerait montrer que pour tout graphe G et tout arrête a de celui-ci :

$$\begin{aligned} P_G(k) &= P_{G-e}(k) - P_{G \setminus e}(k) \\ \Rightarrow P_{G-a}(k) &= P_{G-e-a}(k) - P_{G \setminus e-a}(k) \end{aligned}$$

On supposera évidemment que a et e sont distinctes.

TODO FINISH

1.1.4 Polynôme chromatique ?

Soit H un prédicat tel que :

$$H(m) = \begin{cases} \top & \text{si } \forall G, \text{ graphe de } m \text{ arrêtes ou moins, } P_G(k) \text{ est polynomiale.} \\ \perp & \text{sinon.} \end{cases}$$

- Nous rappelons que $P_{\overline{K_n}}(k) = k^n$, donc $H(0)$ est vraie.
- Supposons $\exists m \in \mathbb{N} \mid H(m)$ l'est également. Ajoutons l'arc a à G . G_{m+e} est un graphe à $(m+1)$ arrêtes :

$$P_{G_{m+z}} = P_{G_{m+e}-e} - P_{G_{m+e} \setminus e}$$

Clairement $P_{G_{m+1}-e}$ et $P_{G_{m+1} \setminus e}$ ont $(m+1) - 1 = m$ arrêtes. Or par hypothèse de récurrence $H(m)$ est vraie, $P_{G_{m+1}}$ est la différence entre deux polynômes, donc est polynomiale lui-même. On a donc $H(m+1)$.

- On vient de montrer $(H(0) \wedge (H(m) \Rightarrow H(m+1)))$. Par récurrence on a donc $H(m)$ vrai $\forall m \in \mathbb{N}$.

1.1.5 Application

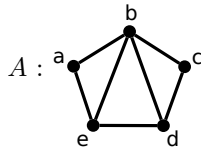
Utilisons la formule trouvée au point précédent, et admettons que pour P_n une chaîne de taille n on a :

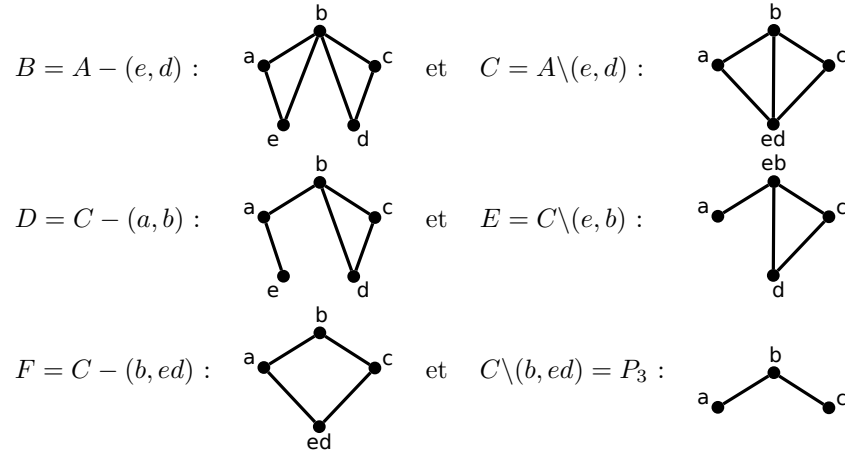
$$P_{P_n}(k) = k(k-1)^{n-1}$$

Prenons A le graphe initial :

$$\begin{aligned}
 P_A(k) &= P_B(k) - P_C(k) \\
 &= (P_D(k) - P_E(k)) - (P_F(k) - P_{P_3}(k)) \\
 &= \left[(P_{P_5}(k) - P_{P_4}(k)) - (P_{P_4}(k) - P_{P_3}(k)) \right] - \left[(P_{P_4}(k) - P_{K_3}(k)) - P_{P_3}(k) \right] \\
 &= P_{P_5}(k) + 2P_{P_3}(k) - P_{K_3}(k) + 3P_{P_4}(k) \\
 &= k(k-1)^4 + 2k(k-1)^2 + k(k-1)(k-2) - 3(k-1)^3 \\
 &= k(k-1) \left[(k-1)^3 + (k-1) + (k-2) - 3(k-1)^2 \right] \\
 &= (k^2 - k) \left[(k-1)^2((k-1) - 3) + 3k - 4 \right] \\
 &= (k^2 - k) [k^3 - 6k^2 + 12k - 8] \\
 &= k^5 - 7k^4 + 18k^3 - 20k^2 + 8k
 \end{aligned}$$

Où :





1.1.6 Coefficients alternatifs

TODO coefficient de k^n est 1, alternating - and + etc

1.1.7 Application à $K_{1,5}$

$K_{1,5}$ étant un arbre, on aura k choix de coloration pour la racine, peu importe le choix de celle-ci, et $k-1$ pour les autres, car chacun qu'on considère sera relié à exactement une autre déjà colorié. En totale ça nous fait donc :

$$\begin{aligned}
 P_{K_{1,5}}(k) &= k(k-1)^5 \\
 &= k((k-1)^2)^2(k-1) \\
 &= k(k^2 - (2k-1))^2(k-1) \\
 &= (k^5 - 4k^4 + 6k^3 - 4k^2 + k)(k-1) \\
 &= k^6 - 5k^5 + 10k^4 - 10k^3 + 5k^2 - k
 \end{aligned}$$

1.1.8 Application aux graphes non connexes

La coloration de chaque composante connexe C_i n'influx pas sur celui des autres. Du coup le nombre de manières de colorier un graphe entier est le produit des polynômes chromatiques de ses composantes connexes :

$$G = \bigcup_{i=0}^n C_i \Rightarrow P_G(k) = \prod_{i=0}^n P_{C_i}(k)$$

1.1.9 Application aux arbres

arbre

$$1.1.10 \quad k^5 - 4k^4 + 6k^3 - 4k^2 + k$$

D'après la propriété de la question 6, nos trois graphes G_1 , G_2 et G_3 auront 5 sommets et 4 arrêtes chacun.

1.1.11 Application à $K_{2,5}$

polychrome of $K_{2,5}$

1.1.12 Application aux cycles

– Pour calculer P_{C_4} , commençons par constater que $C_3 = K_3$:

$$\begin{aligned} P_{C_4} &= P_{P_4} - P_{K_3} \\ &= k(k-1)^3 - k(k-1)(k-2) \\ &= (k^2 - k)((k-1)^2 - (k-2)) \\ &= (k^2 - k)(k^2 - 3k + 3) \\ &= k^4 - 4k^3 + 6k^2 - 3k \end{aligned}$$

– On peut alors utiliser P_{C_4} pour calculer P_{C_5} :

$$\begin{aligned} P_{C_5} &= P_{P_5} - P_{C_4} \\ &= k(k-1)^4 - k^4 - 4k^3 + 6k^2 - 3k \\ &= k^5 - 5k^4 + 10k^3 - 10k^2 + 4k \end{aligned}$$

1.1.13 Démonstration sur les cycles

cycles générales

1.1.14 Démonstration sur les graphes bipartis complets

graphe bipartie générale

1.2 Complexité

1.2.1 SAT \propto 3-SAT

(a) **Énoncé de SAT :**

Données : $\mathcal{V} = \{v_1, v_2 \dots v_n\}$ *Ensemble de n variables*

$\mathcal{C} = \{c_1, c_2, c_3 \dots c_m\}$ *Ensemble de m clauses*

où $c_i = (l_{i1} \vee l_{i2} \vee \dots \vee l_{ik})$ *Clauses de k littéraux*

avec $l_{ij} = v$ ou $\neg v$ *avec $v \in U$*

Problème : existe-il au moins une affectation des variables telle que chaque clause de \mathcal{C} soit vrai.

Énoncé de 3-SAT :

3-SAT est identique au problème SAT avec $k = 3$.

Données : $\mathcal{V} = \{v_1, v_2, v_3 \dots v_n\}$

$\mathcal{C} = \{c_1, c_2, c_3 \dots c_m\}$

où $c_i = (l_{i1} \vee l_{i2} \vee l_{i3})$

avec $l_{ij} = v$ ou $\neg v$

- (b) La réduction du problème SAT peut être défini en montrant que chaque clause c de \mathcal{C} peut-être transformée en un ensemble de clauses \mathcal{C}' tel que pour toute affectation rendant vrai l'ensemble des clauses de \mathcal{C} , on peut trouver une affectation rendant vrai chaque clause de \mathcal{C}' . Chaque clause de \mathcal{C}' devant être de taille exactement 3. La réciproque doit également être montrée.

Définissons les réductions :

$k = 1$

Soit ci_1 une clause de taille 1, on a $ci_1 = (l)$. Ajoutons deux variables $v_1, v_2 \notin \mathcal{V}$ et transformons la clause c en quatre clauses. On obtient l'ensemble $\mathcal{C}_1 = \{c_1, c_2, c_3, c_4\}$ avec :

$$c_1 = (l \vee v_1 \vee v_2)$$

$$c_2 = (l \vee v_1 \vee \neg v_2)$$

$$c_3 = (l \vee \neg v_1 \vee v_2)$$

$$c_4 = (l \vee \neg v_1 \vee \neg v_2)$$

$k = 2$

Soit ci_2 une clause de taille 2, on a $ci_2 = (l_1 \vee l_2)$. Ajoutons une variable $v \notin \mathcal{V}$ et transformons la clause c en deux clauses. On obtient l'ensemble $\mathcal{C}_2 = \{c_1, c_2\}$ avec :

$$c_1 = (l_1 \vee l_2 \vee v)$$

$$c_2 = (l_1 \vee l_2 \vee \neg v)$$

$k = 3$

La clause ci_3 ne subit pas de transformation.

$$\mathcal{C}_3 = \{ci_3\}$$

$k > 3$

Soit la clause $ci_k = (l_1 \vee l_2 \vee \dots \vee l_k)$. On ajoute $(k - 3)$ nouvelles variables $(v_1, v_2 \dots v_{k-3})$.

$$C_k = \underbrace{(l_1 \vee l_2 \vee v_1)}_{c_1} \bigwedge_{i=1}^{k-4} \left[\underbrace{(\neg v_i \vee l_{i+2} \vee v_{i+1})}_{c_{i+1}} \right] \wedge \underbrace{(\neg v_{k-3} \vee l_{k-1} \vee l_k)}_{c_{k-2}}$$

Montrons que SAT est vrai si et seulement si 3-SAT est vrai :

SAT \rightarrow 3-SAT

- Soit une interprétation I_1 qui satisfasse la clause ci_1 :

$$val(I_1, ci_1) = val(I_1, l) = vrai$$

Prenons une interprétation I'_1 avec $val(I_1, l) = val(I'_1, l)$, peu importe les affectations de v_1 et v_2 , l étant présent dans toutes les clauses de \mathcal{C}_1 :

$$val(I'_1, \mathcal{C}) = \top$$

- Soit une interprétation I_2 qui satisfasse la clause ci_2 :

$$\exists i, val(I_2, l_i) = \top$$

Prenons une interprétation I'_2 avec :

$$val(I_2, l_1) = val(I'_2, l_1)$$

$$val(I_2, l_2) = val(I'_2, l_2)$$

Peu importe l'affectation de v dans I'_2 , on a $val(I'_2, \mathcal{C}_2) = \top$.

- Soit une interprétation I_k qui satisfasse la clause ci_k :

$$\exists i, val(I_k, l_i) = \top$$

Prenons une interprétation I'_k telle que :

$$val(I_k, l_i) = val(I'_k, l_i)$$

$$\forall j \in \mathbb{N}^* \mid j \leq (i - 2), val(I'_k, v_j) = \top$$

$$\forall j \in \mathbb{N}^* \mid (i - 1) \leq j \leq (k - 3), val(I'_k, v_j) = \perp$$

On obtient :

$$val(I'_k, \mathcal{C}_k) = \top$$

3-SAT \rightarrow SAT

- Prenons une interprétation I_1 telle que $val(I_1, \mathcal{C}_1) = \top$.
Sans perte de généralité, on suppose que :

$$val(I_1, v_1) = val(I_1, v_2) = \top$$

La clause c_4 de \mathcal{C}_1 ne peut être satisfaite que si $val(I_1, l) = \top$.
On a donc :

$$val(I_1, ci_1) = \top$$

- Prenons une interprétation I_2 telle que $val(I_2, \mathcal{C}_2) = \top$.
Sans perte de généralité on suppose que :

$$val(I_2, v) = \top$$

La clause c_2 de \mathcal{C}_2 ne peut être satisfaite que si $val(I_2, (l_1 \vee l_2)) = \top$.

On a donc :

$$val(I_2, ci_2) = \top$$

- Prenons une interprétation I_k telle que $val(I_k, \mathcal{C}_k) = \top$ et montrons qu'il existe forcément un i tel que $val(I_k, l_i) = \top$.
Supposons que l'interprétation I_k est modèle de \mathcal{C}_k avec

$$\forall i \in \mathbb{N}^* \mid i \leq k, val(I_k, l_i) = \perp$$

$$\Rightarrow val(I_k, v_1) = \top \text{ (dans } c_1)$$

Donc :

$$\begin{aligned} \forall i \in \mathbb{N}^* \mid i \leq (k-4), val(I_k, v_{i+1}) &= \top \\ \Rightarrow val(I_k, v_{k-3}) &= \top \\ \Rightarrow val(I_k, c_{k-2}) &= \perp \\ \Rightarrow val(I_k, \mathcal{C}_k) &= \perp \end{aligned}$$

Pour que l'interprétation I_k satisfasse \mathcal{C}_k , il doit exister un $i \in \mathbb{N}^*$ tel que $i \leq k$ et que $val(I_k, l_i) = \top$.

On a donc :

$$val(I_k, ci_k) = \top$$

- (c) Le point (b) définit la réduction de SAT vers 3-SAT. Afin de montrer la NP-Complétude de 3-SAT, montrons que la réduction s'effectue en un temps polynomial.

Soit :

k la taille de la clause initiale,

v_k le nombre de variables à ajouter pour obtenir des clauses de taille 3,

w_k le nombre de clauses de taille 3 obtenues à partir de la clause initiale.

$$\begin{array}{ll} v_3 = 0 & w_3 = 1 \\ v_4 = 1 & w_4 = 2 \\ v_5 = 2 & w_5 = 3 \\ \vdots & \vdots \end{array}$$

Pour tout $k > 3$:

$$v_k = v_{\lceil \frac{k}{2} \rceil + 1} + v_{\lfloor \frac{k}{2} \rfloor + 1} + 1$$

$$w_k = w_{\lceil \frac{k}{2} \rceil + 1} + w_{\lfloor \frac{k}{2} \rfloor + 1}$$

$v_k = \theta(k)$, donc borné par la taille de F . La réduction s'effectue donc en un temps polynomial.

Il est possible de réduire le problème SAT à 3-SAT en un temps polynomial, SAT étant NP-complet, 3-SAT l'est aussi.

- (d) Soit \mathcal{C} un ensemble de clause à n_v variables avec n_1 clauses de taille 1, n_2 clauses de taille 2, n_3 clauses de taille 3, n_4 clauses de taille 4 et n_5 clauses de taille 5. Calculons le nombre de variables et le nombre de clauses obtenues après réduction (respectivement n'_v et n'_c).

Les points (b) et (c) permettent de déterminer pour une clause de taille k , le nombre de clause obtenues et le nombre de variables ajoutées après réduction. On peut donc en déduire la tableau suivant :

Taille de la clause dans \mathcal{C}	1	2	3	4	5
Nombre de clauses	n_1	n_2	n_3	n_4	n_5
Nombre de variables ajoutées par clause	2	1	0	1	2
Nombre de variables ajoutées au total	$2n_1$	n_2	0	n_4	$2n_5$
Nombre de clauses obtenues par clause	4	2	1	2	3
Nombre de clauses obtenues au total	$4n_1$	$2n_2$	n_3	$2n_4$	$3n_5$

On a donc :

$$\begin{aligned} n'_v &= n_v + 2n_1 + n_2 + n_4 + 2n_5 \\ n'_c &= 4n_1 + 2n_2 + n_3 + 2n_4 + 3n_5 \end{aligned}$$

1.2.2 3-SAT \propto 2-SAT ?

Cette réduction repose sur un principe qui consiste à décomposer une clause de taille k en plusieurs clauses de tailles inférieures.

Soit une clause $c = (l_1 \vee l_2 \vee l_3)$ une clause de taille 3 et I une interprétation qui satisfait c .

Cas 1 : décomposons cette clause en deux clauses c_1 et c_2 de tailles 1 et 2 :

$$\begin{aligned} c_1 &= (l_1) \\ c_2 &= (l_2 \vee l_3) \end{aligned}$$

Pour montrer l'équivalence 3-SAT \leftrightarrow 2-SAT, il faut ajouter une variable v aux deux clauses créées :

$$\begin{aligned} c_1 &= (l_1 \vee v) \\ c_2 &= (l_2 \vee l_3 \vee \neg v) \end{aligned}$$

On a donc la clause c_2 de taille 3.

Cas 2 : décomposons cette clause en trois clauses c_1 , c_2 et c_3 de taille 1 :

$$\begin{aligned} c_1 &= (l_1) \\ c_2 &= (l_2) \\ c_3 &= (l_3) \end{aligned}$$

Pour montrer l'équivalence 3-SAT \leftrightarrow 2-SAT, il faut ajouter deux variables v_1 et v_2 aux trois clauses créées :

$$\begin{aligned} c_1 &= (l_1 \vee v_1 \vee \neg v_2) \\ c_2 &= (l_2 \vee \neg v_1 \vee v_2) \\ c_3 &= (l_3 \vee v_1 \vee v_2) \end{aligned}$$

On a donc également des clauses de taille 3. La réduction définie ci-avant ne permet donc pas la réduction de 3-SAT vers 2-SAT.

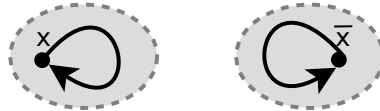
1.2.3 2-SAT, un problème polynomial

(a) Systèmes de deux clauses à deux littéraux :

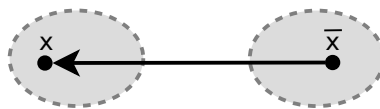
Insatisfiable : $(x \vee x) \wedge (\neg x \vee \neg x)$



Valide : $(x \vee \neg x) \wedge (\neg x \vee x)$



Contingent : $(x \vee x) \wedge (x \vee x)$



Insatisfaisabilité du premier ensemble de clauses est clairement visible sur le graphe car les sommets x et $\neg x$ sont dans la même composante fortement connexe.

Le deux autres ensembles sont satisfiables, les deux sommets ne sont pas dans la même composante fortement connexe.

(b) L'algorithme suivant permet la génération du graphe correspondant à l'ensemble de clauses passé en paramètres, que nous appellerons graphe de satisfaction :

Algorithme 1: GrapheSatisfaction(\mathcal{C}, \mathcal{V})

Données :

\mathcal{C} // Ensemble de clauses

\mathcal{V} // Ensemble des variables

1 début

2 Graphe. $\mathcal{S} = \emptyset$; // Ensemble des sommets du graphe

3 Graphe. $\mathcal{A} = \emptyset$; // Ensemble des arcs du graphe

4 // Initialisation des sommets

5 **pour tous les** $v \in \mathcal{V}$ **faire**

6 ajouter(Graphe. \mathcal{S} , v);

7 ajouter(Graphe. \mathcal{S} , $\neg v$);

8 // Parcours des clauses

9 **pour tous les** $c \in \mathcal{C}$ **faire**

10 ajouter(Graphe. \mathcal{A} , $(\neg c.x, c.y)$);

11 ajouter(Graphe. \mathcal{A} , $(\neg c.y, c.x)$);

12 **retourner** Graphe;

Cet algorithme effectue un parcours de \mathcal{V} et un parcours de \mathcal{C} , sa complexité est donc $O(|\mathcal{C}| + |\mathcal{V}|)$.

(d) Les composantes fortement connexes du graphe de satisfaction généré, ainsi

que leur ordre topologique, peuvent être calculées par l'algorithme de Tarjan.

Algorithme 2: Tarjan_Main(G)

Données : G // Le graphe

```

1 début
2   date  $\leftarrow 0$ ;
3   pour tous les  $s \in G.S$  faire
4     DEBUT[s]  $\leftarrow 0$ ;
5     CFC[s]  $\leftarrow 0$ ;
6   Pile  $\leftarrow \emptyset$ ;
7   numCFC  $\leftarrow 0$ ;
8   pour tous les  $s \in G.S$  faire
9     si DEBUT[s] = 0 alors
10      Tarjan_Rec(s,date,DEBUT,Pile,numCFC,CFC);
11 retourner Comp;
```

Algorithme 3: Tarjan_Rec(s ,date,DEBUT,Pile,numCFC,CFC)

Données :
 s // Le sommet
date // Date de visite du sommet courant
DEBUT // Tableau de dates de visites pour chaque sommet
Pile // Pile de sommets
numCFC // Numéro de la CFC
CFC // Liste des CFC

```

1 début
2   date  $\leftarrow$  date+1;
3   DEBUT[s]  $\leftarrow$  date;
4   min  $\leftarrow$  DEBUT[s];
5   Empiler(Pile,s);
6   pour tous les  $v \in Adj[s]$  faire
7     si DEBUT[v] = 0 alors
8       min  $\leftarrow$ 
9         MIN(min,Tarjan_Rec(v,date,DEBUT,Pile,numCFC,CFC));
10    sinon si CFC[v] = 0 alors
11      min  $\leftarrow$  MIN(min,DEBUT[v]);
12  si min=DEBUT[s] alors
13    Ncfc  $\leftarrow$  numCFC +1;
14  répéter
15    k  $\leftarrow$  Depiler(Pile);
16    CFC[k]  $\leftarrow$  numCFC;
17  jusqu'à k  $\neq$  s;
18 retourner Comp;
```

L'algorithme **Tarjan_Main** initialise la date de visite de chaque sommet à zéro. On constate que les deux algorithmes exécutent **Tarjan_Rec** uniquement sur des sommet dont la date de première visite est nulle. Or chaque

appel à `Tarjan_Rec` affecte une date de visite supérieure à zéro au sommet courant. `Tarjan_Rec` est donc appelé exactement une fois par sommet.

De même, un sommet n'est empilé qu'à l'exécution de `Tarjan_Rec`, donc chaque sommet ne sera empilé (et donc dépilé) qu'une seule fois. La boucle de l'algorithme `Tarjan_Rec` (ligne 13) a une complexité globale en $O(|V|)$.

En revanche, la boucle ligne 6 est effectuée une fois pour chaque voisin du sommet courant, donc $|V|$ fois au pire pour chaque appel. `Tarjan_Rec` n'étant appelée que $|V|$ fois en totale on arrive donc à une complexité de $O(|V|^2)$.

Dans le pire des cas le nombre de variables d'une instance de 2-SAT est égale à deux fois le nombre de clauses (chaque clause comportant dans ce cas deux variables uniques). Or notre conversion génère deux sommets par variable. La complexité de l'algorithme en fonction du nombre de clauses est donc de $O(|C|^2)$.

- (e) Nous passerons par la double-implication pour montrer l'équivalence entre le problème 2-SAT et le tri topologique dans notre graphe de satisfaction. Avant de commencer, notons que les arcs dans le graphe de satisfaction correspondent à des implications. En effet on ajoute un arc de x vers $!x$...
blah blah

– **Tri topologique \Rightarrow 2-SAT**

Étant donnée les composantes fortement connexes (CFC) du graphe étiqueté on peut vérifier linéairement en le nombre de sommets du graphe qu'on a une variable ensemble avec sa négation.

Dans un tel cas

donc un instance 2-SAT insatisfiable. Si par contre n'a pas de variables ensembles avec leurs négative dans le même CFC il suffit de prendre l'ordre topologique calculée dans l'ordre inverse et d'affecter les variables de chaque composant comme précisée dans l'article. Nous finissons alors ou une affectation modèle, ou l'affirmation de l'insatisfiabilité de la forme normale conjonctive. Un tri topologique de notre graphe étiqueté permet donc de résoudre le problème 2-SAT.

– **2-SAT \Rightarrow Tri topologique**

1.3 Calculabilité

1.3.1 Énumération des couples d'entiers

La stratégie d'énumération des couples d'entiers peut être visualisée sur un graphique en suivant les diagonales successives comme sur l'image¹ suivante :

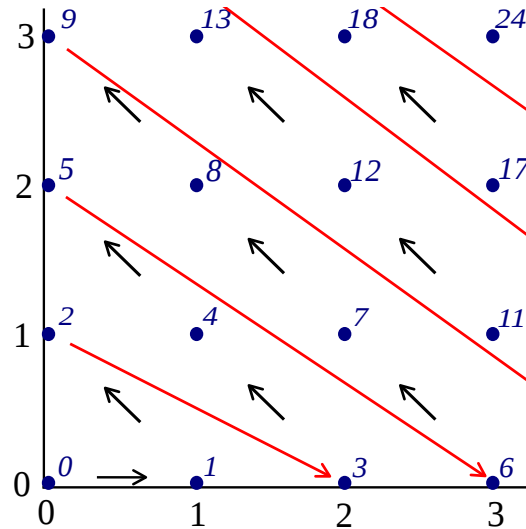


FIG. 1.1 – La fonction de couplage de Cantor établit une bijection de $\mathbb{N} * \mathbb{N}$ dans \mathbb{N} .

Soit $(x, y) \in \mathbb{N} * \mathbb{N}$ un couple. On trie par ordre lexicographique $(x + y)$. Ainsi on obtient le tableau suivant :

(x, y)	$(0, 0)$	$(1, 0)$	$(0, 1)$	$(2, 0)$	$(1, 1)$	$(0, 2)$	$(3, 0)$	$(2, 1)$	$(1, 2)$	$(0, 3)$...
$(x + y)$	0	1	1	2	2	2	3	3	3	3	...
$c_2(x + y)$	0	1	2	3	4	5	6	7	8	9	...

1.3.2 Codons et décodons...

Fonction de codage

$$c_2(x, y) = \frac{(x + y)(x + y + 1)}{2} + y$$

Fonctions de décodage Les fonctions de décodage ne peuvent pas être décrites sous la forme de formules arithmétiques. Elles nécessitent l'algorithme sui-

¹Image provenant de Wikipedia, ce fichier est disponible selon les termes de la licence Creative Commons.

vant :

Algorithme 4: CalculXY(z)

Données : z // Rang du couple (x,y)

```

1 début
2    $s \leftarrow 0$ ;
3    $t \leftarrow 0$ ;
4   tant que  $s \leq z$  faire
5      $s \leftarrow \frac{t*(t+1)}{2}$ ;
6      $t \leftarrow t + 1$ ;
7    $t \leftarrow t - 2$ ;
8    $s \leftarrow \frac{t*(t+1)}{2}$ ;
9    $y \leftarrow z - s$ ;
10   $x \leftarrow t - y$ ;
11  retourner Couple( $x,y$ );

```

1.3.3 Énumération des triplets d'entiers

Codage des triplets : il peut avoir lieu de manière récursive :

$$c_3(x, y, z) = c_2(x, c_2(y, z))$$

Généralisation au codage des k-uplets :

$$c_k(x_1, x_2, \dots, x_k) = c_2(x_1, c_{k-1}(x_2, \dots, x_k))$$

Avec :
$$c_2(x, y) = \frac{(x+y)(x+y+1)}{2} + y$$

1.3.4 Énumération de l'ensemble $[0; 1]$

Prenons une suite $r = (r_1, r_2, r_3, \dots)$ qui énumère les réels de l'intervalle $[0; 1]$, puis créons un réel x compris dans cet intervalle, tel que si la $n^{\text{ième}}$ décimale de r_n est égale à 1, la $n^{\text{ième}}$ décimale de x est égale à 2. Dans la cas contraire, la $n^{\text{ième}}$ décimale de x est égale à 1.

On obtient sur cet exemple :

r_1	=	0	,	4	2	9	6	4	6	1	...
r_2	=	0	,	2	7	3	2	9	4	0	...
r_3	=	0	,	6	4	1	1	5	1	2	...
r_4	=	0	,	3	0	5	9	0	4	3	...
r_5	=	0	,	9	1	3	3	1	8	2	...
r_6	=	0	,	0	2	0	8	3	2	7	...
r_7	=	0	,	2	5	7	3	6	4	0	...
\vdots		\vdots		\vdots	\vdots	\vdots	\vdots	\vdots	\vdots	\vdots	\ddots
				\downarrow	\downarrow	\downarrow	\downarrow	\downarrow	\downarrow	\downarrow	
x	=	0	,	1	1	2	1	2	1	1	...

Le réel x ne peut pas être énuméré par la suite r car il diffère de sa première décimale dans r_1 , de sa deuxième décimale dans r_2 , ... de sa $n^{\text{ième}}$ décimale dans r_n . Pourtant le réel x est clairement dans l'intervalle $[0; 1]$.

L'ensemble des éléments de l'intervalle $[0; 1]$ n'est donc pas dénombrable, donc pas énumérable. On ne peut donc pas trouver de fonction de codage pour cet ensemble.

On peut généraliser à l'ensemble $\mathbb{R} : [0; 1]$ étant inclus dans \mathbb{R} , et $[0; 1]$ n'étant pas dénombrable, l'ensemble \mathbb{R} n'est pas dénombrable.

Chapitre 2

Partie pratique

2.1 Objectifs

Le but de ce projet est de réaliser un solveur du problème de flot maximum. Pour cela, il nous est proposé d'implémenter l'algorithme d'Edmonds-Karp ainsi que l'algorithme de Dinic.

2.2 Spécification fonctionnelles

Le programme devra être capable de générer aléatoirement des graphes de type *réseau de transport* comportant un nombre de sommets et d'arêtes défini. Il devra aussi proposer deux fonctions de résolution du problème de flots sur un tel réseau, l'une implémentant l'algorithme d'Edmonds-Karp et l'autre celui de Dinic. Ces deux fonctions devront générer et retourner le graphe d'écart final. Enfin le programme devra disposer d'une fonction d'affichage de la valeur du flot maximum ainsi que de la valeur du flot sur chaque arc.

2.3 Spécification technique

2.3.1 Choix du langage de programmation

Le programme sera réalisé en C++ afin de bénéficier des performances en termes de rapidité d'exécution du langage ainsi que de l'aspect objet permettant d'abstraire la représentation d'une graphe.

2.3.2 Structures

```
typedef int weight_t;  
typedef uint vertex_t;
```

```
typedef struct
{
    vertex_t u;
    vertex_t v;
} edge;

typedef struct
{
    vertex_t vertex_src;
    vertex_t vertex_dest;
    weight_t weight;
} arc_t;

typedef struct
{
    vertex_t vertex;
    weight_t weight;
} neighbor_t;

typedef list<vertex_t> path_t;
```

2.3.3 Représentation d'un graphe

Nous avons choisi d'abstraire la représentation d'un graphe via une classe abstraite que nous avons nommé `AbstractGraph` dont nous proposerons deux implémentations une par liste d'adjacence *AdjacencyListGraph* et une par matrice *MatrixGraph*

```
class AbstractGraph
{
public:

    AbstractGraph();
    virtual
    ~AbstractGraph() = 0;

    virtual bool
    addArc(const arc_t &arc) = 0;

    virtual bool
    addArc(vertex_t src, vertex_t dest, weight_t w) = 0;

    virtual void
    rmArc(const arc_t &arc) = 0;

    virtual void
    rmArc(vertex_t src, vertex_t dest) = 0;
```

```

virtual void
rmAllArc() = 0;

virtual void
updateArc(const arc_t &arc) = 0;

virtual weight_t
increaseWeight(vertex_t src, vertex_t dest, weight_t w) = 0;

virtual uint
getNbrVertices() const = 0;

virtual string
toString() const;

virtual list<neighbor_t>
getSuccessors(vertex_t vertex) const = 0;

virtual list<neighbor_t>
getPredecessors(vertex_t vertex) const = 0;

virtual weight_t
getWeight(vertex_t src, vertex_t dest) const = 0;

};

```

2.3.4 AdjacencyListGraph

La classes AdjacencyListGraph devra représenter un graphe sous forme de deux liste d'adjacence une liste d'adjacence représentant les successeur d'un sommet et l'autre représentant les prédécesseur d'un sommet.

Ce doublon de l'information permet d'accéléré l'accès au voisin d'un sommet, notamment à ces prédécesseur. En effet en sauvegardant une liste des prédécesseurs de chaque arcs nous accédons a un prédécesseur d'un sommet en $O(m)$ alors que si nous avions qu'une liste des successeur cette accès ce ferait en $O(nm)$.

Ce gain de performance en terme de rapidité ce fait au détriment de la quantité de mémoire utilisé, mais celle-ci n'augmente que d'une valeur constante par rapport au nombre d'arcs.

```

class AdjacencyListGraph : public AbstractGraph
{

```

```

public:

```

```

//*****
//      CONSTRUCTOR
//*****

```

```
AdjacencyListGraph(uint nbr_vertices);

AdjacencyListGraph(const AbstractGraph& graph);
AdjacencyListGraph(const AdjacencyListGraph& graph);

AdjacencyListGraph &
operator=(const AbstractGraph& graph);
AdjacencyListGraph &
operator=(const AdjacencyListGraph& graph);

virtual
~AdjacencyListGraph();

virtual bool
addArc(const arc_t &arc);

virtual bool
addArc(vertex_t src, vertex_t dest, weight_t w);

virtual void
rmArc(const arc_t &arc);

virtual void
rmArc(vertex_t src, vertex_t dest);

virtual void
rmAllArc();

virtual void
updateArc(const arc_t &arc);

virtual weight_t
increaseWeight(vertex_t src, vertex_t dest, weight_t w);

virtual uint
getNbrVertices() const;

virtual list<neighbor_t>
getSuccessors(vertex_t vertex) const;

virtual list<neighbor_t>
getPredecessors(vertex_t vertex) const;

virtual weight_t
getWeight(vertex_t src, vertex_t dest) const;

private:
    list<neighbor_t> *successors, *predecessors;
    uint nbr_vertices;
```



```
protected:
    void
    _clear();

    void
    _construct(const AbstractGraph& graph);

};
```

2.3.5 MatrixGraph

```
class MatrixGraph : public AbstractGraph
{
```

```
public:
```

```
//*****
//      CONSTRUCTOR
//*****
```

```
MatrixGraph(uint nbr_vertices);
```

```
MatrixGraph(const AbstractGraph& graph);
```

```
MatrixGraph(const MatrixGraph& graph);
```

```
MatrixGraph &
operator=(const AbstractGraph& graph);
```

```
MatrixGraph &
operator=(const MatrixGraph& graph);
```

```
virtual
~MatrixGraph();
```

```
virtual bool
addArc(const arc_t &arc);
```

```
virtual bool
addArc(vertex_t src, vertex_t dest, weight_t w);
```

```
virtual void
rmArc(const arc_t &arc);
```

```
virtual void
rmArc(vertex_t src, vertex_t dest);
```

```
virtual void
rmAllArc();
```

```

virtual void
updateArc(const arc_t &arc);

virtual weight_t
increaseWeight(vertex_t src, vertex_t dest, weight_t w);

virtual uint
getNbrVertices() const;

virtual list<neighbor_t>
getSuccessors(vertex_t vertex) const;

virtual list<neighbor_t>
getPredecessors(vertex_t vertex) const;

virtual weight_t
getWeight(vertex_t src, vertex_t dest) const;

private:
    weight_t **matrix;
    uint nbr_vertices;

protected:
    void
    _clear();

    void
    _construct(int nbr_vertices);

    void
    _construct(const AbstractGraph& graph);

};

```

2.3.6 Génération aléatoire d'un graphe

```

/**
 * A random flow network generator
 * Attention si le graph passé en paramètre contient des arcs ceux-ci seront
 * supprimé.
 * @param graph une référence vers un graph initialiser avec un nombre de sommets
 * @param rate la proportion d'arcs à ajouter au graphe en pourcentage par rapport au
 * @param min_weight valuation minimal des arcs
 * @param max_weight valuation maximal des arcs
 */
void
flowNetworkGenerator(AbstractGraph& graph, float rate, uint min_weight = 1,
    uint max_weight = 1);

```

```
\subsection{Fonctions générales}
```

```
/**
 * Cette procédure génère une chaîne de caractères représentant l’affichage
 * de la valeur total du flot sur le réseau de transport ainsi que la valeur
 * du flot sur chaque arc.
 * @param flow_network le réseau de transport
 * @param residual_network le graphe d’écart associé
 */
string
flowToString(const AbstractGraph& flow_network,
             const AbstractGraph& residual_network);
```

```
\subsection{Edmonds-Karp}
```

```
/**
 * Cette fonction retourne le plus court chemin en nombre d’arcs depuis
 * le sommet start jusqu’au sommet end
 * @param g un graphe
 * @param start le sommet de départ
 * @param end le sommet d’arriver
 * @return le plus court chemin en nombre d’arcs de start à end
 */
path_t
leastArcsPath(AbstractGraph &g, vertex_t start, vertex_t end);
```

```
/**
 * Cette fonction retourne la plus petite valuation présente sur un chemin
 * donné dans un graphe
 * @param g un graphe
 * @param path une chemin dans g
 * @return la plus petite valuation présente sur le chemin path dans g
 */
weight_t
lightestArc(AbstractGraph& g, path_t path);
```

```
/**
 * Cette fonction converti un chemin en chaîne de caractère dans un but d’affichage
 * @param path le chemin
 * @param g le graphe
 */
string
pathToString(path_t path, const AbstractGraph& g);
```

```
/**
 * Mise à jour du graphe d’écart depuis un chemin et la valeur du flot à ajouter
 * sur ce chemin
```

```

    * @param le graphe de couche
    * @param p le chemin
    * @param k la valeur du flot à ajouter
    */
void
updateResidualNetwork(AbstractGraph& residualNetwork, path_t p, uint k);

/**
 * algorithme d'Edmonds-Karp
 * @param flow_network le réseau de transport
 * @param src le sommet source
 * @param dest le puit
 * @return le graphe d'écart final
 */
AdjacencyListGraph
edmondsKarp(const AbstractGraph& flow_network, vertex_t src, vertex_t dest);

```

2.3.7 Dinic

```

/**
 * Mise à jour du graphe d'écart depuis un flot
 * @param residual_network le graphe de couche
 * @param p le flot
 */
void
updateResidualNetwork(AbstractGraph& residual_network, AbstractGraph& flow);

/**
 * Génération du graphe de couche associé au réseau de transport
 * @param residual_network le graphe d'écart
 * @param src la source
 * @param dest le puit
 * @return le graphe de couche
 */
LevelGraph
generateLevelGraph(const AbstractGraph& residual_network, vertex_t src,
    vertex_t dest);

/**
 * Calcul du flot bloquant
 * @param level_graph le graphe de couche
 * @param src la source
 * @param dest le puit
 * @return un flot bloquant
 */
AdjacencyListGraph
blockingFlow(LevelGraph& level_graph, vertex_t src, vertex_t dest);

```

```
/**
 * algorithme de Dinic
 * @param flow_network le réseau de transport
 * @param src le sommet source
 * @param dest le puit
 * @return le graphe d'écart final
 */
AdjacencyListGraph
dinic(const AbstractGraph& graph, vertex_t src, vertex_t dest);
```