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

# Chapitre 1

# Partie théorique

## 1.1 Algorithmique

### 1.1.1 Fonction chromatique $P_G(k)$

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 $n$î 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 Nombre chromatique $\chi(G)$

On l'appelle "nombre chromatique" de $G$ : $\chi(G)$ 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 :

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

On a donc très clairement :

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

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 :

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

On supposera évidemment que $a$ et $e$ sont distinctes.

TODO FINISH

### 1.1.4 Polynôme chromatique ?

Soit $H$ un prédicat tel que :

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

- Nous rappellons 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 recurrence $H(m)$ est vraie, $P_{G_{m+1}}$ est la différence entre deux polynomiales, 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 de la décomposition

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

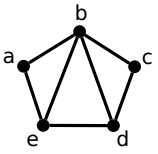
$$P_{P_n}(k) = k(k-1)^{n-1}$$

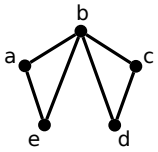
Prenons $A$ le graphe initial :

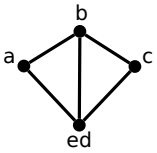
$$\begin{aligned}
P_A(k) &= P_B(k) - P_C(k) \\
&= \big(P_D(k) - P_E(k)\big) - \big(P_F(k) - P_{P_3}(k)\big) \\
&= \Big[\big(P_{P_5}(k) - P_{P_4}(k)\big) - \big(P_{P_4}(k) - P_{P_3}(k)\big)\Big] - \Big[\big(P_{P_4}(k) - P_{K_3}(k)\big) - P_{P_3}(k)\Big] \\
&= P_{P_5}(k) + 2P_{P_3}(k) - P_{K_3}(k) + 3P_{P_4}(k)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

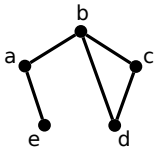
Où :

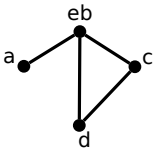
$A$ :

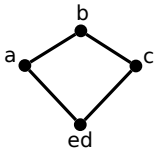


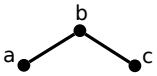
$B = A - (e, d)$ : 

et $C = A\backslash(e, d)$ : 


$D = C - (a, b)$ : 

et $E = C\backslash(e, b)$ : 


$F = C - (b, ed)$ : 

et $C\backslash(b, ed) = P_3$ : 


### 1.1.6 Coefficients alternatifs

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

### 1.1.7 Polynôme chromatique de $K_{1,5}$

$K_{1,5}$ 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 Coloration de 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 :

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

### 1.1.9 Coloration d'arbres

Supposons qu'on ait un graphe $G$ tel que $P_G(k) = k(k-1)^{n-1}$ :

– Intuitivement ceci veut dire qu'on a $k$ choix de couleurs pour le premier sommet colorié, puis $(k-1)$ pour chacun des $(n-1)$ autres. Du coup chaque sommet, lors de sa coloration, ne doit être en contact qu'avec un seul sommet déjà colorié. Ceci n'est possible que dans un graphe sans cycle.
–

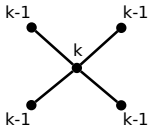
### 1.1.10 $k^5 - 4k^4 + 6k^3 - 4k^2 + k$

Grace aux développments précédentes (questions 7 et 12) on reconnait :

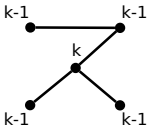
$$\begin{aligned} & k^5 - 4k^4 + 6k^3 - 4k^2 + k \\ = \; & k(k-1)^4 \\ = \; & P_{P_5}(k) \end{aligned}$$

Notre premier exemple sera donc $P_5$ le chemin de taille 5. Ensuite d'après la propriété de la question 6 nous ne cherchions que des graphes aillant 5 sommets et 4 arrêtes, et d'après celle de la question 9 ils doivent en plus être arbres. Nous proposons donc le graphe étoile $S_5 = K_{1,4}$ ainsi que l'arbre à 5 sommets dont la particularité est d'être sans particularité :

$S_5$ :




$A_5$ :




### 1.1.11 Polynôme chromatique de $K_{2,5}$

Nous avions déjà calculé le polynôme chromatique de $K_{1,5}$ :

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

Or $K_{2,5}$ se construit à partir de $K_{1,5}$ par l'ajout d'un sommet relié au 5 de la partition majoritaire. Ce nouveau sommet pourra être colorié de $(k-5)$ manières, car seront interdis les couleurs de ses 5 voisins. On a donc :

$$\begin{aligned} P_{K_{2,5}} &= (P_{K_{1,5}}(k))(k-5) \\ &= (k^6 - 5k^5 + 10k^4 - 10k^3 + 5k^2 - k)(k-5) \\ &= k^7 - 10k^6 + 35k^5 - 60k^4 + 55k^3 - 26k^2 + 5k \end{aligned}$$

### 1.1.12 Polynômes chromatiques de $C_4$ et $C_5$

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

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

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

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

### 1.1.13 Coloration de cycles

cycles générales

### 1.1.14 Coloration de 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 définit 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 $ci_k = (l_1 \vee l_2 \vee \cdots \vee l_k)$. On ajoute $(k-3)$ nouvelles variables $(v_1, v_2 \dots v_{k-3})$.

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

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 :

  $$\begin{aligned} val(I_k, l_i) &= val(I'_k, l_i) \\ \forall j \in \mathbb{N}^* \mid j \leq (i-2), val(I'_k, v_j) &= \top \\ \forall j \in \mathbb{N}^* \mid (i-1) \leq j \leq (k-3), val(I'_k, v_j) &= \bot \end{aligned}$$

  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 satisfaire 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

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

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

Donc :

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

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.

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

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 :

$$n'_v = n_v + 2n_1 + n_2 + n_4 + 2n_5$$
$$n'_c = 4n_1 + 2n_2 + n_3 + 2n_4 + 3n_5$$

### 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 :

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

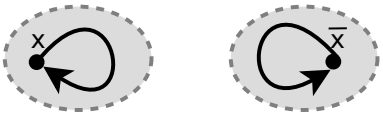
On a donc également des clauses de taille 3. La réduction définie ci-avant ne permet donc pas la réduction de 3–SAT vers 2–SAT.

### 1.2.3 2–SAT, un problème polynomial

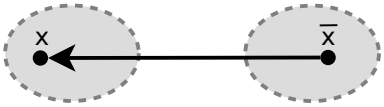
(a) Systèmes de deux clauses à deux littéraux :

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


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




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




Insaisissabilité 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.S = ∅ ; // Ensemble des sommets du graphe
3      Graphe.A = ∅ ; // Ensemble des arcs du graphe
4      // Initialisation des sommets
5      pour tous les v ∈ V faire
6          ajouter(Graphe.S,v);
7          ajouter(Graphe.S,¬v);
8      // Parcours des clauses
9      pour tous les c ∈ C faire
10         ajouter(Graphe.A,(¬c.x, c.y));
11         ajouter(Graphe.A,(¬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 ← 0;
3      pour tous les s ∈ G.S faire
4          DEBUT[s] ← 0;
5          CFC[s] ← 0;
6      Pile ← ∅;
7      numCFC ← 0;
8      pour tous les s ∈ 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 ← date+1;
3      DEBUT[s] ← date;
4      min ← DEBUT[s];
5      Empiler(Pile,s);
6      pour tous les v ∈Adj[s] faire
7          si DEBUT[v]= 0 alors
8              min ←
               MIN(min,Tarjan_Rec(v,date,DEBUT,Pile,numCFC,CFC)));
9          sinon si CFC[v]= 0 alors
10             min ← MIN(min,DEBUT[v]);
11     si min=DEBUT[s] alors
12         Ncfc ← numCFC +1;
13     répéter
14         k ← Depiler(Pile);
15         CFC[k] ← numCFC;
16     jusqu'à k ≠ s;
17     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 $O(|\mathcal{V}|)$.

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

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 $O(|\mathcal{C}|^2)$.

(e)
- On appellera Absurd–Graph le problème de décision consistant de savoir si un variable partage avec la négation une composante fortement connexe (ou $CFC$) du graphe de satisfaction.
- Toute arc ajouté par $GrapheSatisfaction$ correspond à une contrainte de la forme $((x \vee y) \wedge \neg x) \Rightarrow y)$, donc une implication. Étant donnée les $CFC$, calculés par l'algorithme de $Tarjan$, on peut vérifier linéairement en le nombre de sommets si le graphe de satisfaction est "absurde", ce qui corresponderait effectivement à $(\neg(x) \Leftrightarrow x)$.
- Dans le cas contraire il suffit de prendre l'inverse de l'ordre topologique calculée par $Tarjan$ et d'affecter les variables de chaque composante comme précise l'article. Ceci nous garantie de ne pas avoir d'interprétations $\bot \Rightarrow \top$, seuls à pouvoir casser l'enchaînement des implications.
- Nous finissions alors soit avec une affection modèle, soit l'affirmation de l'insatisfiabilité de l'instance 2–SAT. Un algorithme pour résoudre Absurd–Graph (à savoir $Tarjan$ et des poussières) permet donc de résoudre le problème 2–SAT.

## 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'image [1] suivante :

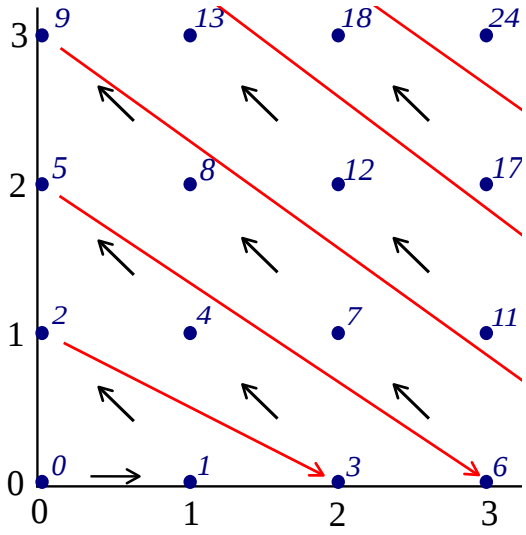



FIGURE 1.1 – La fonction de couplage de Cantor établit une bijection de $\mathbb{N} * \mathbb{N}$ dans $\mathbb{N}$.

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

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

suivant :

**Algorithme 4:** CalculXY($z$)

```
   Données : z // Rang du couple (x,y)
1  début
2      s ← 0;
3      t ← 0;
4      tant que s ⩽ z faire
5          s ← t∗(t+1)/2;
6          t ← t + 1;
7      t ← t − 2;
8      s ← t∗(t+1)/2;
9      y ← z − s;
10     x ← 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** :

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

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

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

Prenons une suite $r = (r_1, r_2, r_3, \ldots)$ 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 :

$$\begin{array}{ccccccccccccc}
r_1 & = & 0 & , & \mathbf{4} & 2 & 9 & 6 & 4 & 6 & 1 & \ldots \\
r_2 & = & 0 & , & 2 & \mathbf{7} & 3 & 2 & 9 & 4 & 0 & \ldots \\
r_3 & = & 0 & , & 6 & 4 & \mathbf{1} & 1 & 5 & 1 & 2 & \ldots \\
r_4 & = & 0 & , & 3 & 0 & 5 & \mathbf{9} & 0 & 4 & 3 & \ldots \\
r_5 & = & 0 & , & 9 & 1 & 3 & 3 & \mathbf{1} & 8 & 2 & \ldots \\
r_6 & = & 0 & , & 0 & 2 & 0 & 8 & 3 & \mathbf{2} & 7 & \ldots \\
r_7 & = & 0 & , & 2 & 5 & 7 & 3 & 6 & 4 & \mathbf{0} & \ldots \\
\vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \ddots \\
 & & & & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \\
x & = & 0 & , & 1 & 1 & 2 & 1 & 2 & 1 & 1 & \ldots
\end{array}$$

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 peu 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

*Le but de ce TP est d'implémenter deux algorithmes de résolution du problème de flot maximum : l'algorithme d'Edmonds-Karp et l'algorithme de Dinic. Nous commencerons par spécifier les fonctionnalités que devra implémenter notre programme, puis nous détaillerons la manière dont ces fonctionnalités ont été développées. Une troisième partie sera consacrée aux tests effectués sur les deux algorithmes ainsi qu'à l'analyse des résultats.*

## 2.1 Spécifications fonctionnelles

### 2.1.1 Résolution du problème de flot maximum

Le programme doit être capable de :
- générer et d'actualiser les graphes d'écarts successifs
- calculer la valeur du flot obtenu à partir du graphe d'écart final
- résoudre le problème de flot maximum en suivant l'algorithme d'**Edmonds-Karp**
- résoudre le problème de flot maximum en suivant l'algorithme de **Dinic**
- retourner la solution de manière exploitable pour l'analyse

Il faudra veiller à conserver la complexité des deux algorithmes, notamment en prenant garde aux structures de données et librairies utilisées.

### 2.1.2 Génération aléatoire d'un réseau de transport

La génération aléatoire de graphes de type réseau de transport permettra de tester les deux algorithmes. Il faudra veiller à ce que le graphe respecte les conditions d'un réseau de transport notamment la possession d'une source et d'un puits, la pondération des arcs (capacités), et assurer la connexité du graphe. La génération de ce réseau de transport devra être paramétrable selon la taille (nombre de sommets) et la densité (nombre d'arcs).

## 2.2 Spécifications techniques

### 2.2.1 Programmation C++

En plus de sa notoriété, nous avons choisi de développer l'application en C++ car il s'agit d'un bon compromis entre langage orienté objet et langage de bas niveau. Nous pourrons ainsi abstraire la gestion des graphes (notamment des structures de données) dans nos algorithmes, tout en gardant la possibilité d'optimiser le code grâce à la flexibilité du langage C.

### 2.2.2 Structures de données

#### Graphes

Les graphes peuvent êtres stockés à l'aide différents types de structures de données. Nous avons choisi d'en implémenter deux : par listes d'adjacences et par matrice d'adjacences.

**Listes d'adjacences** : chaque sommet possède la liste de ses voisins. Ces listes ont l'avantage d'allouer de la mémoire uniquement lorsqu'une information doit être stockée.

**Matrice d'adjacences** : la mémoire allouée pour cette structure de données ne dépend que du nombre de sommets ($n^2$). Cette structure a l'avantage d'offrir un accès direct à un arc pour deux sommets donnés.

Dans un but d'optimisation mémoire, on utilise en général des listes d'adjacences lorsque l'on travail sur des graphes peu denses. En effet, la taille allouée par cette structure de données étant directement dépendante du nombre d'arcs, elle est donc réduite par rapport aux matrices d'adjacences. Sur des graphes très dense, on utilisera en revanche des matrices d'adjacences, permettant un accès aux données plus rapide.

Les ressources matériels disponibles peuvent également orienter ce choix.

### 2.2.3 Modélisation

Afin de s'abstraire de la structure de donnée, nous avons choisi de créer une classe abstraite `AbstractGraph` dont deux classes fille héritent. Un graphe peut donc être de type `AdjacencyListGraph` ou `MatrixGraph`. Cela permet une grande généricité des algorithme développés, ce qui permet d'utiliser une structure de données de manière totalement détachée des algorithmes.

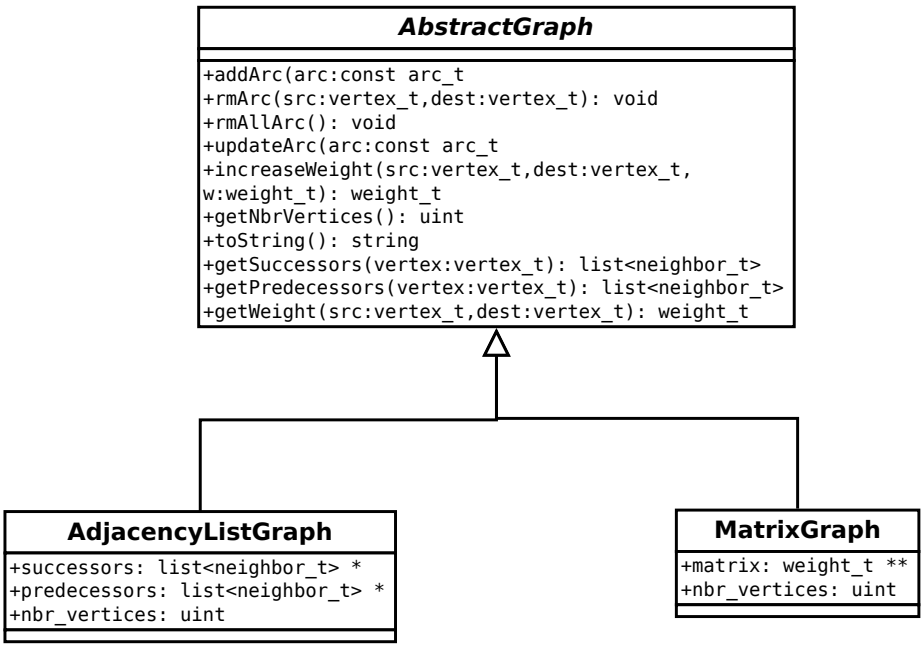



FIGURE 2.1 – Diagramme de classes.

### 2.2.4 Types et structures

Déclaration des types `weight_t`, `vertex_t` et `path_t`, et des structures `edge`, `neighbor_t`

```
1  typedef int weight_t;
2  typedef uint vertex_t;
3
4  typedef struct
5  {
6    vertex_t u;
7    vertex_t v;
8  } edge;
9
10 typedef struct
11 {
12   vertex_t vertex_src;
13   vertex_t vertex_dest;
14   weight_t weight;
15 } arc_t;
16
17 typedef struct
18 {
19   vertex_t vertex;
20   weight_t weight;
21 } neighbor_t;
22
23 typedef list<vertex_t> path_t;
```

### 2.2.5 Classe `AbstractGraph`

Cette classe permet une abstraction des différentes représentations d'un graphe (structures de données utilisées). Elle déclare les méthodes qui doivent êtres implémentées dans les classes filles.

Header de la classe `AbstractGraph`

```
1  class AbstractGraph
2  {
3  public:
4
5    AbstractGraph();
6    virtual
7    ~AbstractGraph() = 0;
8
9    virtual bool
10   addArc(const arc_t &arc) = 0;
11
12   virtual bool
13   addArc(vertex_t src, vertex_t dest, weight_t w) = 0;
14
15   virtual void
16   rmArc(const arc_t &arc) = 0;
17
18   virtual void
19   rmArc(vertex_t src, vertex_t dest) = 0;
20
21   virtual void
22   rmAllArc() = 0;
23
24   virtual void
25   updateArc(const arc_t &arc) = 0;
26
27   virtual weight_t
28   increaseWeight(vertex_t src, vertex_t dest, weight_t w) = 0;
29
30   virtual uint
31   getNbrVertices() const = 0;
32
33   virtual string
34   toString() const;
35
36   virtual list<neighbor_t>
37   getSuccessors(vertex_t vertex) const = 0;
38
39   virtual list<neighbor_t>
40   getPredecessors(vertex_t vertex) const = 0;
41
42   virtual weight_t
43   getWeight(vertex_t src, vertex_t dest) const = 0;
44
45 };
```

### 2.2.6 Classe `AdjacencyListGraph`

Cette classe représente un réseau de transport sous la forme de deux listes d'adjacences : une représente les successeurs d'un sommet, l'autre les prédécesseurs.

Ce doublon d'information permet d'accélérer l'accès aux voisins d'un sommet, notamment à ces prédécesseurs. En effet, cette méthode nous permet d'accéder à une liste des prédécesseurs directement (complexité de $O(1)$) alors que l'accès via les listes des successeurs implique le parcours de toutes ces listes (complexité de $O(nm)$).

Ces doubles listes d'adjacences nous assurent un gain de performances en terme de rapidité, qui se fait au détriment de la quantité de mémoire utilisé, qui se trouve doublée.

Header de la classe `AdjacencyListGraph`

```
1  class AdjacencyListGraph : public AbstractGraph
2  {
3
4  public:
5    //**************************
6    // CONSTRUCTOR
7    //**************************
8
9    AdjacencyListGraph(uint nbr_vertices);
10
11   AdjacencyListGraph(const AbstractGraph& graph);
12   AdjacencyListGraph(const AdjacencyListGraph& graph);
13
14   AdjacencyListGraph &
15   operator=(const AbstractGraph& graph);
16   AdjacencyListGraph &
17   operator=(const AdjacencyListGraph& graph);
18
19   virtual
20   ~AdjacencyListGraph();
21
22   virtual bool
23   addArc(const arc_t &arc);
24
25   virtual bool
26   addArc(vertex_t src, vertex_t dest, weight_t w);
27
28   virtual void
29   rmArc(const arc_t &arc);
30
31   virtual void
32   rmArc(vertex_t src, vertex_t dest);
33
34   virtual void
35   rmAllArc();
36
37   virtual void
38   updateArc(const arc_t &arc);
39
40   virtual weight_t
41   increaseWeight(vertex_t src, vertex_t dest, weight_t w);
42
43   virtual uint
44   getNbrVertices() const;
45
46   virtual list<neighbor_t>
47   getSuccessors(vertex_t vertex) const;
48
49   virtual list<neighbor_t>
50   getPredecessors(vertex_t vertex) const;
51
52   virtual weight_t
53   getWeight(vertex_t src, vertex_t dest) const;
54
55 private:
56   list<neighbor_t> *successors, *predecessors;
57   uint nbr_vertices;
58
59 protected:
60   void
61   _clear();
62
63   void
64   _construct(const AbstractGraph& graph);
65
66 };
```

### 2.2.7 Classe `MatrixGraph`

Header de la classe `MatrixGraph`

```
1  class MatrixGraph : public AbstractGraph
2  {
3
4  public:
5    //**************************
6    // CONSTRUCTOR
7    //**************************
8
9    MatrixGraph(uint nbr_vertices);
10
11   MatrixGraph(const AbstractGraph& graph);
12
13   MatrixGraph(const MatrixGraph& graph);
14
15   MatrixGraph &
16   operator=(const AbstractGraph& graph);
17
18   MatrixGraph &
19   operator=(const MatrixGraph& graph);
20
21   virtual
22   ~MatrixGraph();
23
24   virtual bool
25   addArc(const arc_t &arc);
26
27   virtual bool
28   addArc(vertex_t src, vertex_t dest, weight_t w);
29
30   virtual void
31   rmArc(const arc_t &arc);
32
33   virtual void
34   rmArc(vertex_t src, vertex_t dest);
35
36   virtual void
37   rmAllArc();
38
39   virtual void
40   updateArc(const arc_t &arc);
41
42   virtual weight_t
43   increaseWeight(vertex_t src, vertex_t dest, weight_t w);
44
45   virtual uint
46   getNbrVertices() const;
47
48   virtual list<neighbor_t>
49   getSuccessors(vertex_t vertex) const;
50
51   virtual list<neighbor_t>
52   getPredecessors(vertex_t vertex) const;
53
54   virtual weight_t
55   getWeight(vertex_t src, vertex_t dest) const;
56
57 private:
58   weight_t **matrix;
59   uint nbr_vertices;
60
61 protected:
62   void
63   _clear();
64
65   void
66   _construct(int nbr_vertices);
67
68   void
69   _construct(const AbstractGraph& graph);
70
```

```
71  };
```

## 2.3 Génération de réseaux de transport aléatoires

```
1   /**
2    * A random flow network generator
3    * Attention si le graph passe en parametre contient des arcs ceux-ci seront
4    * supprime.
5    * @param graph une reference vers un graph initialiser avec un nombre de sommets
6    * @param rate la proportion d'arcs a ajouter au graphe en pourcentage par rapport au graphe complet.
7    * @param min_weight valuation minimal des arcs
8    * @param max_weight valuation maximal des arcs
9    */
10  void
11  flowNetworkGenerator(AbstractGraph& graph, float rate, uint min_weight = 1,
12      uint max_weight = 1);
13
14  \subsection{Fonctions generales}
15
16  /**
17   * Cette procedure genere une chaine de caracteres representant l'affichage
18   * de la valeur total du flot sur le reseau de transport ainsi que la valeur
19   * du flot sur chaque arc.
20   * @param flow_network le reseau de transport
21   * @param residual_network le graphe d'ecart associe
22   */
23  string
24  flowToString(const AbstractGraph& flow_network,
25      const AbstractGraph& residual_network);
26
27
28
29  \subsection{Edmonds-Karp}
30
31  /**
32   * Cette fonction retourne le plus court chemin en nombre d'arcs depuis
33   * le sommet start jusqu'au sommet end
34   * @param g un graphe
35   * @param start le sommet de depart
36   * @param end le sommet d'arriver
37   * @return le plus court chemin en nombre d'arcs de start a end
38   */
39  path_t
40  leastArcsPath(AbstractGraph &g, vertex_t start, vertex_t end);
41
42  /**
43   * Cette fonction retourne la plus petite valuation presente sur un chemin
44   * donne dans un graphe
45   * @param g un graphe
46   * @param path une chemin dans g
47   * @return la plus petite valuation presente sur le chemin path dans g
48   */
49  weight_t
50  lightestArc(AbstractGraph& g, path_t path);
51
52  /**
53   * Cette fonction converti un chemin en chaine de caractere dans un but d'affichage
54   * @param path le chemin
55   * @param g le graphe
56   */
57  string
58  pathToString(path_t path, const AbstractGraph& g);
59
60  /**
61   * Mise a jour du graphe d'ecart depuis un chemin et la valeur du flot a ajouter
62   * sur ce chemin
63   * @param le graphe de couche
64   * @param p le chemin
65   * @param k la valeur du flot a ajouter
66   */
```

```
67  void
68  updateResidualNetwork(AbstractGraph& residualNetwork, path_t p, uint k);
69
70
71  /**
72   * algorithme d'Edmonds-Karp
73   * @param flow_network le reseau de transport
74   * @param src le sommet source
75   * @param dest le puit
76   * @return le graphe d'ecart final
77   */
78  AdjacencyListGraph
79  edmondsKarp(const AbstractGraph& flow_network, vertex_t src, vertex_t dest);
```

## 2.4 Procédures principales

### 2.4.1 Algorithme de Dinic

#### Mise à jour du graphe d'écart

```
1  /**
2   * Mise a jour du graphe d'ecart depuis un flot
3   * @param residual_network le graphe de couche
4   * @param p le flot
5   */
6  void
7  updateResidualNetwork(AbstractGraph& residual_network, AbstractGraph& flow);
```

#### Calcul du flot bloquant

```
1  /**
2   * Calcul du flot bloquant
3   * @param level_graph le graphe de couche
4   * @param src la source
5   * @param dest le puit
6   * @return un flot bloquant
7   */
8  AdjacencyListGraph
9  blockingFlow(LevelGraph& level_graph, vertex_t src, vertex_t dest);
```

#### Exécution de Dinic

```
1  /**
2   * algorithme de Dinic
3   * @param flow_network le reseau de transport
4   * @param src le sommet source
5   * @param dest le puits
6   * @return le graphe d'ecart final
7   */
8  AdjacencyListGrap
9  dinic(const AbstractGraph& graph, vertex_t src, vertex_t dest);
```

## 2.5 Tests & résultats

### 2.5.1 Méthode de test

#### Série de tests

Pour tester les performances des deux algorithmes implémentés, nous avons générer une série de problèmes à résoudre sur des réseaux de transports ayant les paramètres suivants :

– nombre de sommets variant de 100 à 1000 par palier de 100,
– densité du graphe variant de 10% à 100% par palier de 10.

Pour chaque test (un nombre de sommets et une densité donnés), nous avons généré dix graphes afin de travailler sur des moyennes lors de l'analyse.

#### GNU gprof

Nous avons évalué les algorithmes en fonction de leurs temps d'exécution. Le temps réel d'exécution (real time) ayant peu de sens pour effectuer des statistiques correctes, nous avons choisi de mesurer les temps CPU (CPU time). Le temps CPU est le temps alloué au processus par le système d'exploitation sur le processeur. Contrairement au temps réel, le temps CPU est indépendant des autres processus en cours d'activité et aux interruptions systèmes : il s'agit du temps effectivement passé par le CPU pour traiter le processus.

L'analyse des temps CPU a été faite à l'aide de l'outil GNU gprof. Son utilisation requière l'ajout de l'argument `-pg` lors la compilation. A l'exécution du programme, un fichier `gmon.out` est généré. La commande `gprof` permet ensuite de créer un fichier texte de statistiques. Pour chaque méthode, un grand nombre de données sont disponibles, nous nous sommes intéressés principalement aux suivantes :

– pourcentage du temps CPU total,
– temps CPU total,
– temps CPU par appel, de manière cumulative (en prenant en compte les appel à d'autres fonctions) ou non.

### 2.5.2 Analyse des résultats