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 6

1.1.1 Fonction chromatique $P_G(k)$ 6

1.1.2 Nombre chromatique $\chi(G)$ 6

1.1.3 Décomposition de P_G 6

1.1.4 Polynôme chromatique ? 7

1.1.5 Application de la décomposition 7

1.1.6 Particularités des polynômes chromatiques 8

1.1.7 Polynôme chromatique de $K_{1,5}$ 10

1.1.8 Coloration de graphes non-connexes 10

1.1.9 Coloration d'arbres 10

1.1.10 Trois graphes 10

1.1.11 Polynôme chromatique de $K_{2,5}$ 11

1.1.12 Polynômes chromatiques de C_4 et C_5 11

1.1.13 Coloration de cycles 11

1.1.14 Coloration de graphes bipartis complets 12

1.2 Complexité 14

1.2.1 SAT $\propto$ 3–SAT 14

1.2.2 3–SAT $\propto$ 2–SAT ? 17

1.2.3 2–SAT, un problème polynomial 18

1.3 Calculabilité 21

1.3.1 Énumération des couples d'entiers 21

1.3.2 Codons et décodons. 21

1.3.3 Énumération des triplets d'entiers 22

1.3.4 Énumération de l'ensemble $[0; 1]$ 22

2 Partie pratique **25**

2.1 Spécifications fonctionnelles 25

2.1.1 Résolution du problème de flot maximum 25

2.1.2 Génération aléatoire d'un réseau de transport 26

2.2 Spécifications techniques 26

2.2.1 Langage de programmation et organisation 26

2.2.2 Représentation du problème de flot maximum 26

2.2.3 Modélisation 27

2.2.4 Types et structures 29

2.3 Génération aléatoire de réseaux de transport 32

2.3.1 Stratégies de génération des arcs 32

2.3.2 Méthode implémentée 34

2.4 Procédures principales 35

2.4.1 Algorithme d'Edmonds-Karp 35

2.4.2 Algorithme de Dinic 36

2.5 Tests & résultats 37

2.5.1 Méthode de test 37

2.5.2 Analyse des résultats 37

Chapitre 1

Partie théorique

Sommaire

1.1 Algorithmique 6
1.1.1 Fonction chromatique $P_G(k)$ 6
1.1.2 Nombre chromatique $\chi(G)$ 6
1.1.3 Décomposition de P_G 6
1.1.4 Polynôme chromatique ? 7
1.1.5 Application de la décomposition 7
1.1.6 Particularités des polynômes chromatiques 8
1.1.7 Polynôme chromatique de $K_{1,5}$ 10
1.1.8 Coloration de graphes non-connexes 10
1.1.9 Coloration d'arbres 10
1.1.10 Trois graphes 10
1.1.11 Polynôme chromatique de $K_{2,5}$ 11
1.1.12 Polynômes chromatiques de C_4 et C_5 11
1.1.13 Coloration de cycles 11
1.1.14 Coloration de graphes bipartis complets 12
1.2 Complexité 14
1.2.1 SAT $\propto$ 3–SAT 14
1.2.2 3–SAT $\propto$ 2–SAT ? 17
1.2.3 2–SAT, un problème polynomial 18
1.3 Calculabilité 21
1.3.1 Énumération des couples d'entiers 21
1.3.2 Codons et décodons. 21
1.3.3 Énumération des triplets d'entiers 22
1.3.4 Énumération de l'ensemble $[0; 1]$ 22

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 nî 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 :

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

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

- 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. On admettra que pour tout graphe G et tout arrête a de celui-ci (a et e sont supposés distinctes) :

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

Par induction structurelle on a donc $P_G(k) = P_{G-e}(k) - P_{G\setminus e}(k)$ pour tout graphe G.

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 :

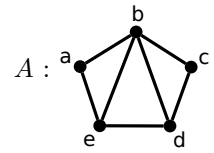
$$P_{P_n}(k) = k(k-1)^{n-1}$$

Prenons A le graphe initial :

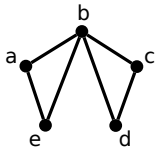
$$\begin{aligned}
P_A(k) &= P_B(k) - P_C(k) \\
&= \big(P_D(k) - P_E(k)\big) - \big(P_F(k) - P_{P_3}(k)\big) \\
&= \Big[\big(P_{P_5}(k) - P_{P_4}(k)\big) - \big(P_{P_4}(k) - P_{P_3}(k)\big)\Big] - \Big[\big(P_{P_4}(k) - P_{K_3}(k)\big) - P_{P_3}(k)\Big] \\
&= P_{P_5}(k) + 2P_{P_3}(k) - P_{K_3}(k) + 3P_{P_4}(k)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

Où :

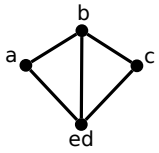
A :



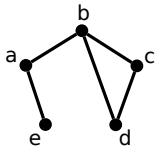
$B = A - (e,d)$:



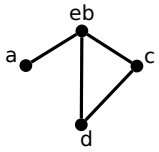
et $C = A\backslash(e,d)$:



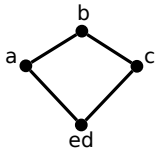
$D = C - (a,b)$:



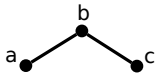
et $E = C\backslash(e,b)$:



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



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



1.1.6 Particularités des polynômes chromatiques

Nous cherchions à prouver que pour tout graphe G à n sommets et m arrêtes, avec $C = \{c_i\}$ un ensemble de coéfficients naturelles (donc positives) :

$$P_G(k) = k^n - mk^{n-1} + \sum_{i=2}^{n} \Big((-1)^i (c_i k^{n-i})\Big)$$

On nomera $F_k(n,m,C)$ cette forme polynomiale particulière. Soit H un prédicat tel que :

$$H(n,m) \Leftrightarrow \forall G(n,m),\ \exists C \mid P_G(k) = F(n,m,C)$$

On montrera par récurance que $H(n,m)$ est vrai pour tout n et tout m :

– **Sommets – cas de base**
Pour $n = 1$ on a $P_G(k) = k$, donc $H(1, 0)$ est vérifié.
– **Sommets – pas récursif**
Supposons qu'il existe un nombre de sommets n_0 et un nombre d'arrêtes m_0 tel que $H(n_0, m_0)$ soit vrai. Soit G un graphe à n_0 sommets et m_0 arrêtes, et G^+ le graphe généré en reliant un nouveau sommet s' à une selection de $n' \leq n_0$ sommets de G. On aura donc $(k - n')$ manières de colorier s'. De ce fait :

$$P_{G^+}(k) = P_G(k)(k - n')$$

Où $P_G(k)$ est de la forme F. $P_G^+(k)$ est donc de la forme :

$$\begin{aligned} & F(n, m, C)(k - n') \\ = \; & F(n, m, C)k - n'F(n, m, C) \end{aligned}$$

∗ **Maintient des signes alternatifs**
Le fait de multiplier par k "décale" les termes du polynomiale à gauche. Soustraire $n'F_k(n, m, C)$ retranche alors au coéfficient de chaque terme n' fois celui de leur voisin de gauche, avec $0 \leq n' \leq n$. Or si le term est positive, ce fameux voisin de gauche est négative par hypothèse, donc son coéfficient augmentera. De même si le term est négative son coéfficient va diminuer. Les signes resteront donc alternatifs dans $P_G^+(k)$.
∗ **Maintient de k^n**
k^n deviendra k^{n+1} dans $F_k(n, m, C)k$ et, n'aillant pas de voisin gauche dans $F_k(n, m, C)$, on ne lui retranchera rien. La propriété sur le terme de plus haut degré est donc maintenu.
∗ **Maintient de mk^{n-1}**
Le terme $-mk^{n-1}$, aillant pour voisin gauche k^n, deviendra

$$-mk^n - n'k^n = -(m + n')k^n$$

$(m + n')$ étant le nombre d'arrêtes de G^+. La propriétée sur le second terme de plus haut degré est alors maintenu aussi.

On a donc $H(n_0 + 1, m_0)$, qui nous servira pas la suite ...
– **Arrêtes – cas de base**
Pour $m = 0$ on a $G = \overline{K}_n$ et donc $P_G(k) = P_{\overline{K}_n}(k) = k^n$. Du coup $\forall n \in \mathbb{N}^+$, $H(n, 0)$ est vrai.
– **Arrêtes – pas récursive**
Supposons qu'il existe un nombre d'arrêtes m_0 tel que pour tout $m \leq m_0$ et tout n on a $H(n, m_0)$. Soit G un graphe à n sommets et $m_0 + 1$ arrêtes. D'après la formule du question 3 on a donc :

$$\begin{aligned} P_G(k) &= P_{G-e}(k) - P_{G\setminus e}(k) \\ &= F_k(n, m_0, C') - F_k(n - 1, m_0 + 1 - n', C'') \\ &= \Big(k^n - m_0k^{n-1} + \dots\Big) - \Big(k^{n-1} - (m_0 + 1 - n')k^{n-2} + \dots\Big) \\ &= k^n - (m_0 + 1)k^{n-1} + \dots \\ &= F_k(n, m_0 + 1, C''') \end{aligned}$$

On a admet ci-dessus la conservation de alternance des signes. De ce fait on a $H(n, m_0 + 1)$.

Nous avions démontrés :

$$H(1,0) \wedge H(n,0) \wedge (H(n,m) \Rightarrow H(n+1,m)) \wedge (H(n,m \Rightarrow H(n,m+1))$$

Par récurrence on a donc $H(n,m)$ pour tout pair (n,m) de $\mathbb{N}^* \times \mathbb{N}$

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, car même avec un seul cycle on aura au plus $k(k-1)^{n-2}(k-2) < k(k-1)^{n-1}$ colorations différentes.
- De même si G est une forêt, même s'il n'est composé que de deux composantes connexes, il devraient y avoir au moins $k^2(k-1)^{n-2} > k(k-1)^{n-1}$ colorations possibles.

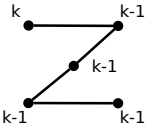
On a donc G connexe et sans cycle : c'est la définition même d'un arbre.

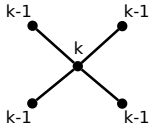
1.1.10 Trois graphes

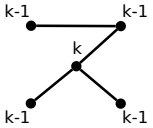
Grace aux dévelopments précédentes (questions 7 et 12) on reconnait :

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

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 (on l'appellera A_5) :

P_5 : 

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

Soit H un prédicat tel que :

$$H(n) \Leftrightarrow \Big(P_{C_n}(k) = \big((k-1)^n + (-1)^n(k-1)\big)\Big)$$

– Prenons comme cas de base $n = 3$:

$$\begin{aligned} P_{C_3}(k) &= P_{K_3}(k) \\ &= k(k-1)(k-2) \\ &= k^3 - 3k^2 + 2k \\ &= k^3 - 3k^2 + 3k - 1 - k + 1 \\ &= (k^2 - 2k + 1)(k-1) - (k-1) \\ &= (k-1)^3 - (k-1) \end{aligned}$$

$H(3)$ est donc vrai.

– Supposons $H(n)$ vrai pour un n donnée. Suivant cette hypothèse on a :

$$\begin{aligned} P_{C_{n+1}}(k) &= P_{C_{n+1}-e} - P_{C_{n+1}\setminus e}(k) \\ &= P_{P_{n+1}}(k) - P_{C_n}(k) \\ &= k(k-1)^n - \Big((k-1)^n + (-1)^n(k-1)\Big) \\ &= (k-1)(k-1)^n - (-1)^n(k-1) \\ &= (k-1)^{n+1}(-1)^{n+1}(k-1) \end{aligned}$$

On a donc $H(n+1)$.

Résultat des courses : $H(3) \wedge (H(n) \Rightarrow H(n+1))$. Par récurrence on a donc $H(n)$ vrai pour tout $n \geq 3$. À noter qu'un cycle de taille moins de 3, ça se voit pas souvent.

1.1.14 Coloration de graphes bipartis complets

Comme d'habitude, posons H un prédicat tel que :

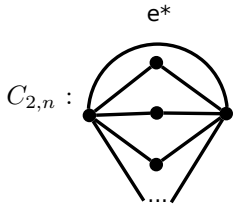
$$H(n) \Leftrightarrow \Big(P_{K_{2,n}}(k) = \big(k(k-1)^n + k(k-1)(k-2)^n\big)\Big)$$

– Prenons comme cas de base $n = 1$. Or $K_{2,1} = P_3$ donc :

$$\begin{aligned} P_{K_{2,1}}(k) &= P_{P_3}(k) \\ &= k(k-1)^2 \\ &= k(k-1)(1 + (k-2)) \\ &= k(k-1)^1 + k(k-1)(k-2)^1 \end{aligned}$$

$H(1)$ est donc vrai.

– On appelle $C_{2,n}$ le graphe pseudo-biparti complet à n sommets, qui se génère à partie de $K_{2,n}$ en reliant les deux sommets de la première partition :



On appelle cette arrête spéciale e^*. On a donc :

$$C_{2,n} - e^* = K_{2,n}$$

et

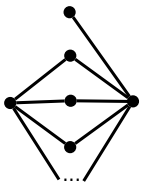
$$C_{2,n} \backslash e^* = S_n$$

On rappelle que S_n est un arbre. Finalement on admettera :

$$\forall e \quad K_{2,n} - e = K_{2,n}(k-1)$$

En effet il ne reste que $K_{2,n-1}$ plus un sommet pendant, donc coloriable de $k-1$ manières :

$K_{2,n-1}^{+1}$:



Supposons alors $H(n)$ vrai pour un n donnée.

$$\begin{aligned}
P_{K_{2,n+1}}(k) &= P_{K_{2,n+1}-e} - P_{K_{2,n+1}\backslash e} \\
&= P_{K_{2,n}}(k-1) - P_{C_{2,n}} \\
&= P_{K_{2,n}}(k-1) - P_{K_{2,n}} + P_{S_n} \\
&= (k-2)\big(k(k-1)^n + k(k-1)(k-2)^n\big) + k(k-1)^{n-1} \\
&= \dots \\
&= k(k-1)^{n+1} + k(k-1)(k-2)^{n+1}
\end{aligned}$$

On a donc $H(n+1)$.
Comme avant $H(1) \wedge (H(n) \Rightarrow H(n+1))$ implique par récurrence que $H(n)$ sera vrai pour tout $n \geq 1$.

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 :

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

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 :

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

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 :

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

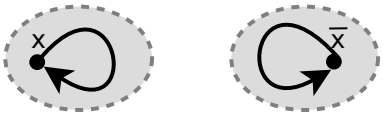
On a donc également des clauses de taille 3. La réduction définie ci-avant ne permet donc pas la réduction de 3–SAT vers 2–SAT.

1.2.3 2–SAT, un problème polynomial

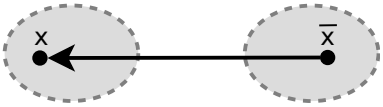
(a) Systèmes de deux clauses à deux littéraux :

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 de Philipe Gambette. 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 :

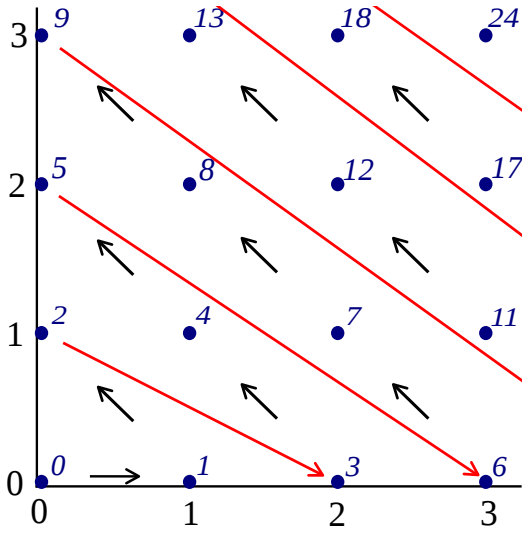


FIGURE 1.1 – La fonction de couplage de Cantor établit une bijection de $\mathbb{N} * \mathbb{N}$ dans $\mathbb{N}$.

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)
début
    s ← 0;
    t ← 0;
    tant que s ⩽ z faire
        s ← t*(t+1)/2;
        t ← t + 1;
    t ← t − 2;
    s ← t*(t+1)/2;
    y ← z − s;
    x ← t − y;
    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 :

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

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

Sommaire

2.1 Spécifications fonctionnelles 25
2.1.1 Résolution du problème de flot maximum 25
2.1.2 Génération aléatoire d'un réseau de transport 26
2.2 Spécifications techniques 26
2.2.1 Langage de programmation et organisation 26
2.2.2 Représentation du problème de flot maximum 26
2.2.3 Modélisation 27
2.2.4 Types et structures 29
2.3 Génération aléatoire de réseaux de transport . . 32
2.3.1 Stratégies de génération des arcs 32
2.3.2 Méthode implémentée 34
2.4 Procédures principales 35
2.4.1 Algorithme d'Edmonds-Karp 35
2.4.2 Algorithme de Dinic 36
2.5 Tests & résultats 37
2.5.1 Méthode de test 37
2.5.2 Analyse des résultats 37

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).

La densité du graphe correspond à son taux d'arêtes par rapport au nombre d'arêtes du graphe complet non orienté ayant le même nombre de sommets. Soit la formule suivante :

$$nb\_aretes = \frac{n * (n-1)}{2} * taux$$

2.2 Spécifications techniques

2.2.1 Langage de programmation et organisation

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.

Utilisation d'un gestionnaire de version

Nous avons d'utiliser un gestionnaire de version afin de pouvoir sauvegarder et partager notre code entre les membres de l'équipe.

2.2.2 Représentation du problème de flot maximum

Le problème du flot maximum n'est défini que pour un certain type de graphes appelé *réseau de transport*.

Réseau de transport

Les réseaux de transport peuvent êtres représenté par un graphe orienté pondéré pour lequel on défini un sommet source et un sommet puits. Le sommet source à la particularité de ne pas avoir de prédecesseur et le puits celui de ne pas avoir de successeur. La valuation d'un arcs représente sa capacité maximal.

Graphe d'écart

Les graphes d'écart peuvent êtres simplement représenté par un graphe orienté pondéré.

Graphe de couches

Les graphes de couches peuvent êtres représenté par un graphe orienté pondéré auquel on ajoute un tableau de listes de sommets indexé par le numéro de couche. Chaque liste représente donc une couche du graphe, elle contient donc l'ensemble des sommets se trouvant à une distance d du graphe où d est le numéro de la couche.

2.2.3 Modélisation

Diagramme de classes

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.

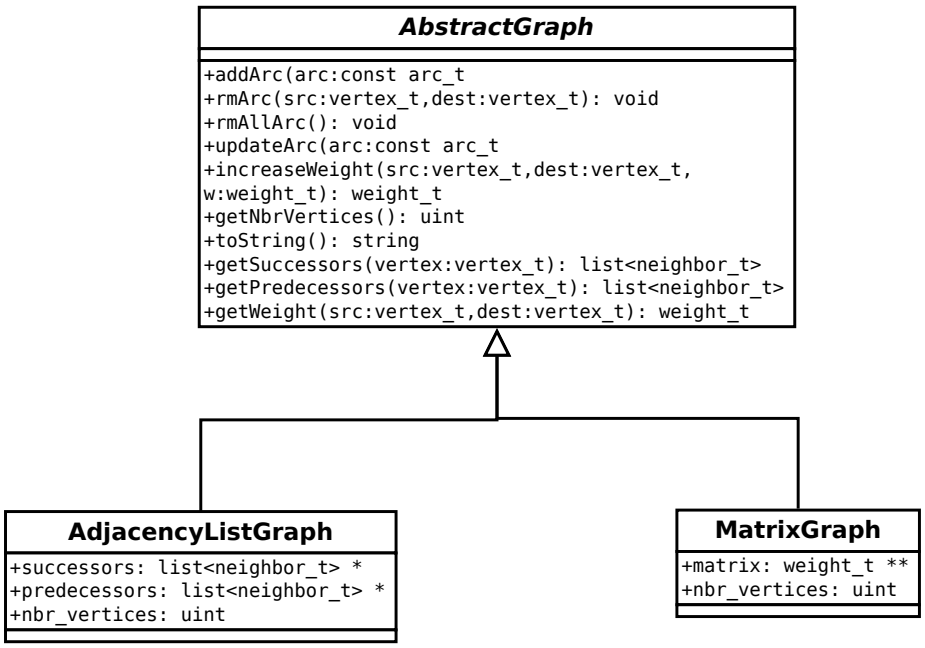


FIGURE 2.1 – Diagramme de classes.

Les graphes peuvent être 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.

Application aux réseaux de transport

Les deux classe précédentes n'étant pas spécialisées dans la modélisation de réseaux de transport, nous adopterons les conventions suivantes :

- le sommet zéro (premier sommet) représente la source,
- le sommet n (dernier sommet) représente le puits,
- la pondération affecté à chaque arc représente sa capacité.

2.2.4 Types et structures

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

```
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;
```

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`

```
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;
```

```
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  };
```

Classe `AdjacencyListGraph`

Cette classe représente un orienté valué 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  };
```

Classe `MatrixGraph`

Cette classe représente un graphe orienté valué sous la forme d'une matrice d'adjacence. L'absence d'un arc est représenté par la valeur -1. Toute valeur positive représente la pondération de l'arc.

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 aléatoire de réseaux de transport

2.3.1 Stratégies de génération des arcs

La première stratégie utilisée par notre application était naïve. Nous commencions par tracer un chemin de la source vers le puits pour s'assurer de la

connexité du graphe. Puis on ajoutait aléatoirement des arcs.

Algorithme 5: flowNetworkGenerator1(G,rate,min_weight,max_weight)

```
   Données :
   G // Graphe d'entrée
   rate // Densité
   min_weight // Capacité minimum
   max_weight // Capacité maximum
1  début
2      nb_vertices ← G.getNbrVertices();
3      nb_arcs ← nbr_arcs_max(nbr_vertices)*rate;
4      //génération d'un chemin
5      pour u allant de 1 à (nbr_vertices − 1 faire
6          weight ← rand(min_weight, max_weight);
7          g.add(u − 1, u);
8          −−nb_arcs;
9      //génération des arcs
10     tant que nb_arcs> 0 faire
11         u ← rand % (nb_vertices - 1);
12         v ← (rand % (nb_vertices - 1)) + 1;
13         si !G.contains(u, v) alors
14             weight ← rand(min_weight, max_weight);
15             G.add(u,v,weight);
16             −−nbr_arcs;
17     retourner G;
```

Cet algorithme, bien que « plutôt » fonctionnel pour des graphes peu denses ne fourni aucune garantie d'arrêt. Nous avons donc réfléchi à une deuxième approche, qui consistait à générer tout les arcs possibles dans un tableau de

type vector puis de piocher parmi ces arcs.

Algorithme 6: flowNetworkGenerator(G,rate,min_weight,max_weight)

```
   Données :
   G // Graphe d'entrée
   rate // Densité
   min_weight // Capacité minimum
   max_weight // Capacité maximum
 1 début
 2     nb_vertices ← G.getNbrVertices();
 3     nb_arcs ← nbr_arcs_max(nbr_vertices)*rate;
 4     vector<edge>vector;
 5     edge e;
 6     //génération de tous les arcs possibles
 7     pour  u allant de 1 à (nbr_vertices − 1 faire
 8         //Pour tout u, on crée l'arc u, u+1 afin d'assurer l'existence
 9         //d'un chemin entre la source et le puits
           G.addArc(u,u + 1,rand(min_weight, max_weight));
10         −−nb_arcs;
11         pour v allant de u + 2 à nb_vertices−1 faire
12             e.u ← u;
13             e.v ← v;
14             vector.push_back(e);
15     //ajout des arcs
16     tant que nb_arcs> 0 faire
17         index ← rand() % vector.size();
18         tant que vector[index]=nil faire
19             index = ++index % vector.size();
20         e ← vector.get(index);
21         si e.u = 0 OU e.v = nb_vertices −1 OU rand() % 2 = 1 alors
22             G.addArc(e.u,e.v,randMinMax(min_weight,max_weight);
23         sinon
24             G.addArc(e.v,e.u,randMinMax(min_weight,max_weight);
25         −−nb_arcs;
26         vector[index]=nil;
27     retourner G;
```

Dans tous les cas, nous sommes certain de l'arrêt de cet algorithme. Il faut veiller à utiliser un type tableau en marquant à « nil » les arcs ajoutés. En effet, nous avions commencé par utiliser une liste chainée mais la fonction `list.get(...)` a une complexité en $O(n^2)$.

2.3.2 Méthode implémentée

Cette fonctionnalité est assurée par la fonction `flowNetworkGenerator` qui construit aléatoirement un réseau de transport sur le graphe passé en paramètre.

Elle permet de spécifier le nombre de sommets, la densité du graphe et l'intervalle des capacités aléatoires.

Notons que cette méthode utilise la fonction `rand()` fournis par la librairie standard. Cette fonction « pseudo-aléatoire » nécessite l'initialisation d'un générateur via la fonction `srand(int)`. Il est donc possible de sauvegarder la « graine », paramètre passée à la fonction `srand(int)`, afin de pouvoir régénérer un graphe identique.

Entête de la méthode de génération aléatoire de réseaux de transport

```
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);
```

2.4 Procédures principales

2.4.1 Algorithme d'Edmonds-Karp

La méthode `edmondsKarp` permet l'exécution de cet algorithme. Il prend en paramètre par référence un réseau de transport.

Entête de la méthode d'exécution d'Edmonds-Karp

```
1  /**
2   * algorithme d'Edmonds-Karp
3   * @param flow_network le reseau de transport
4   * @param src le sommet source
5   * @param dest le puit
6   * @return le graphe d'ecart final
7   */
8  AdjacencyListGraph
9  edmondsKarp(const AbstractGraph& flow_network, vertex_t src, vertex_t dest);
```

La recherche du plus court chemin en nombre d'arc est implémenté par la fonction `leastArcsPath` qui effectue un parcours en largeur. Sa complexité est donc en $O(n + m)$ avec structure par listes d'adjacences, et en $O(n^2)$ par avec une structure par matrice d'adjacences.

La mise à jour du graphe d'écart est assurée par la méthode `updateResidualNetwork` avec une complexité en $O(n)$.

Entête des principales méthodes utilisées par la procédure `edmondsKarp`

```
1  /**
2   * Cette fonction retourne le plus court chemin en nombre d'arcs depuis
3   * le sommet start jusqu'au sommet end
4   * @param g un graphe
5   * @param start le sommet de depart
6   * @param end le sommet d'arriver
7   * @return le plus court chemin en nombre d'arcs de start a end
8   */
```

```
9  path_t
10 leastArcsPath(AbstractGraph &g, vertex_t start, vertex_t end);
11
12 /**
13  * Cette fonction retourne la plus petite valuation presente sur un chemin
14  * donne dans un graphe
15  * @param g un graphe
16  * @param path une chemin dans g
17  * @return la plus petite valuation presente sur le chemin path dans g
18  */
19 weight_t
20 lightestArc(AbstractGraph& g, path_t path);
21
22 /**
23  * Mise a jour du graphe d'ecart depuis un chemin et la valeur du flot a ajouter
24  * sur ce chemin
25  * @param le graphe de couche
26  * @param p le chemin
27  * @param k la valeur du flot a ajouter
28  */
29 void
30 updateResidualNetwork(AbstractGraph& residualNetwork, path_t p, uint k);
```

2.4.2 Algorithme de Dinic

Entête de la méthode d'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);
```

Entête des principales méthodes utilisées par la procédure `dinic`

```
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);
8
9  /**
10  * Calcul du flot bloquant
11  * @param level_graph le graphe de couche
12  * @param src la source
13  * @param dest le puit
14  * @return un flot bloquant
15  */
16 AdjacencyListGraph
17 blockingFlow(LevelGraph& level_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 à 20%, 50% et 80%,
- capacité des arcs variant de 1 à 20.

Pour chaque test (un nombre de sommets et une densité donnés), nous avons généré 100 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

A partir des données obtenues, nous avons essayé de créer une série de graphiques ayant comme ordonnée le temps CPU, et en abscisse :

- (1) le nombre de sommets (en conservant une densité identique),
- (2) la densité (en conservant un nombre d'arcs identique),
- (3) le nombre d'arcs (en faisant donc varier le nombre d'arc et la densité).

Seule la représentation (1) permet de mettre en évidence un phénomène intéressent. Ci-après sont donc présentés trois graphiques, pour des densités de 20%, 50% et 80%.

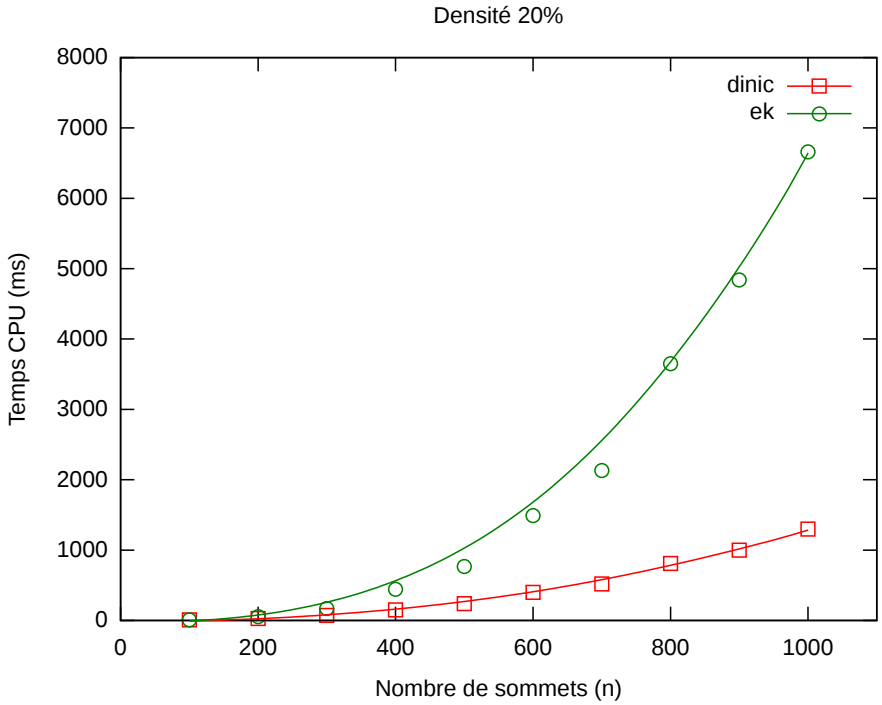


FIGURE 2.2 – Temps CPU en fonction du nombre de sommets, densité de 20%

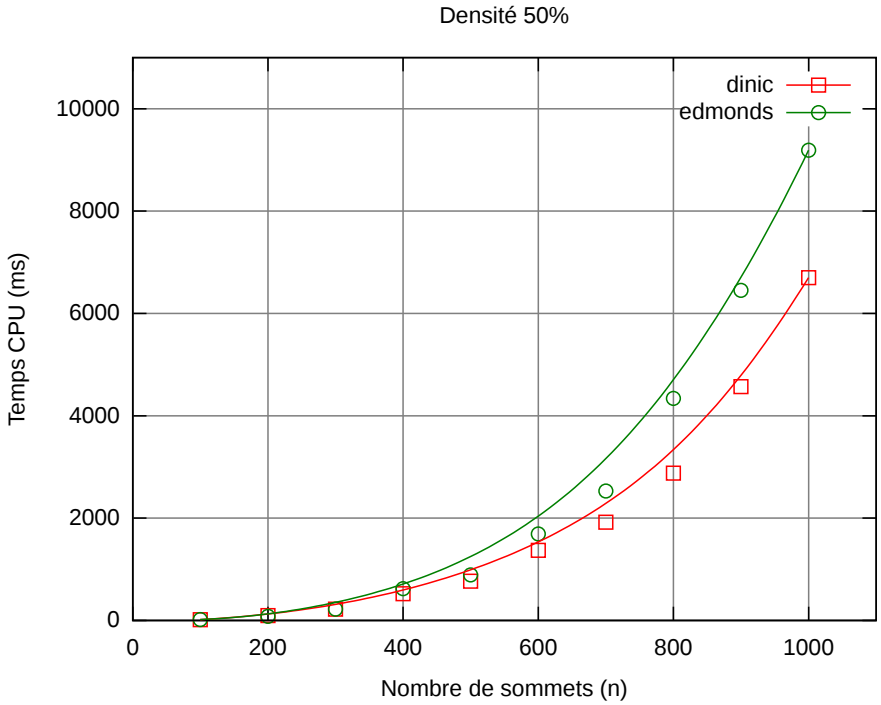


FIGURE 2.3 – Temps CPU en fonction du nombre de sommets, densité de 50%

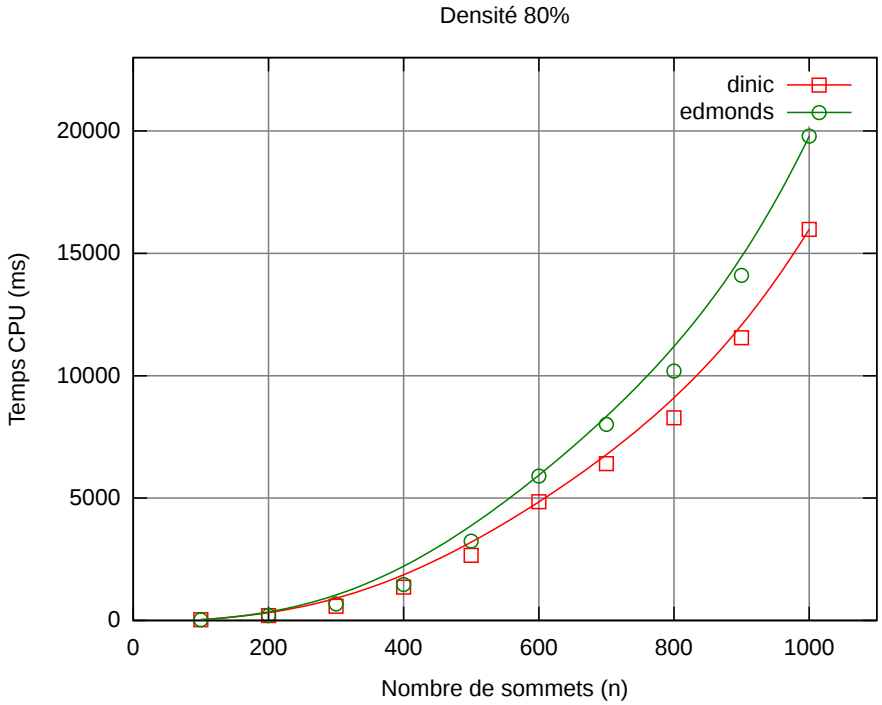


FIGURE 2.4 – Temps CPU en fonction du nombre de sommets, densité de 80%

L'évolution du temps CPU de l'algorithme d'Edmonds-Karp change peu en fonction de la densité du graphe. Ces premiers graphiques montre également que l'algorithme de Dinic reste plus performant en terme de temps CPU même si l'on constate que l'écart diminue lorsque la densité augmente.

Ce résultat est cohérent, étant donné les complexité des algorithmes :
- $O(nm^2)$ pour Dinic,
- $O(n^2m)$ pour Edmonds-Karp,

il est normal que la densité influe plus sur le premier que le second.

Nous nous sommes cependant intéressés aux résultats obtenus sur des plus petits graphes. Voici deux graphiques qui montre nos résultats sur des graphes inférieurs à 200 sommets.

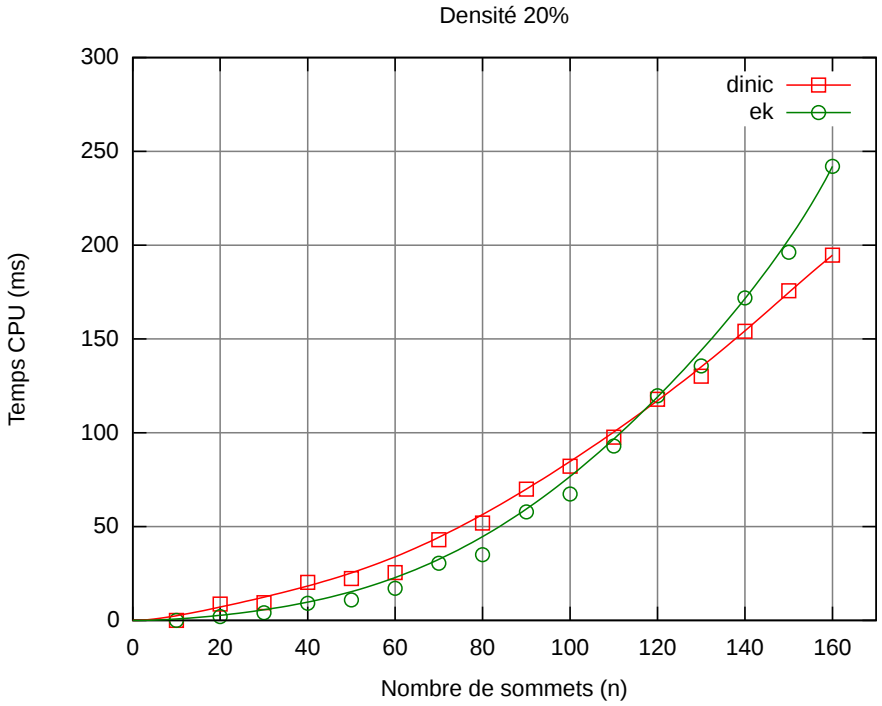


FIGURE 2.5 – Temps CPU en fonction du nombre de sommets (valeur faible), densité de 20%

On remarque ici qu'en faite, l'algorithme d'Edmonds-Karp est plus rapide que Dinic sur des graphes de 160 sommets pour une densité de 20%, et 120 sommets pour une densité de 80%.

De manière général, on constate que l'algorithme de Dinic est plus rapide qu'Edmonds-Karp, surtout sur des graphes peu denses. Edmonds-Karp demeure cependant plus rapide sur des graphes extrêmement petits (nombre de sommets inférieurs à 150), d'autant plus si le graphe est peu dense.

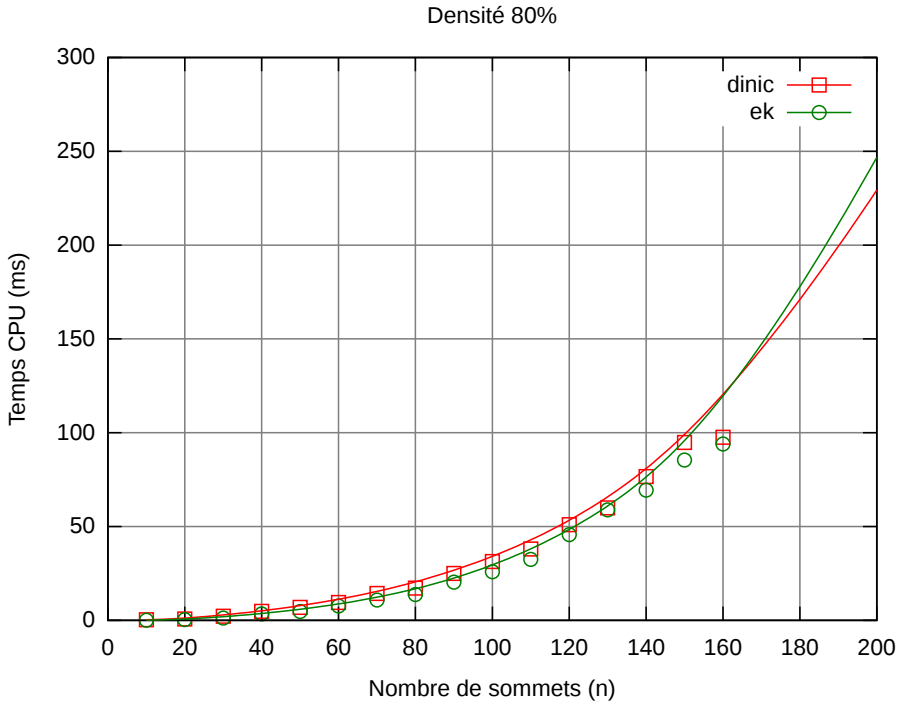


FIGURE 2.6 – Temps CPU en fonction du nombre de sommets (valeur faible), densité de 80%