Faculté des Sciences de Montpellier 2

Master 1 IFPRU - 2010

Mémoire de Travaux Pratiques

Algorithmique|Complexité|Calculabilité

Sébastien PASTOR
Fabien PRIOTTO
Alain IBRAHIM
Noureddine Yacine NAIR BENREKIA

1 Descriptif pour la partie pratique

Dans ce projet, nous mettons en pratique les algorithmes de Dinic et Edmonds-Karp pour le calcul du flot maximum dans des graphes orientés et valués. Une des difficultés de cet exercice est liée à la complexité des algorithmes qui se déroulent en temps polynomial dans la taille de la donnée (nombre de sommets et nombre d'arcs).

Langage choisi: C++

C++ est un langage de programmation orienté objet qui offre notamment les avantages suivants:

- Le passage des paramétres par adresse: l'augmentation d'un flot par itérations génère autant d'instances de tableaux que de matrices associées aux graphes. Selon le nombre de sommets, ceux-ci peuvent devenir rapidement volumineux. C++ permet de passer en paramètre l'adresse des tableaux constituant les matrices. Par opposition au passage par valeur, le passage par adresse évite la copie supplémentaire des données en zone mémoire. On obtient donc un programme plus économique en terme de consommation de mémoire vive.
- C++ est un langage compilé qui ne ncessite pas la présence d'une machine virtuelle pour chaque architecture. La compilation, comme elle optimise les récursions, elle apporte un gain de temps à l'exécution appréciable pour les gros volumes de données que représentent les graphes avec un grand nombre de sommets et d'arcs.
- C++ supporte le polymorphisme. Cela nous permet d'envisager des types d'implmentation diffrents pour le stockage des chaînes, chemins et voisinage de sommets notamment. Les méthodes s'adaptent aux types des paramètres.
- C++ accepte la surcharge des fonctions, ce qui est un atout pour la réutilisation des classes. Notre programme pourra constituer une base pour l'écriture de programme plus évolué de résolution des problèmes de flots dans un réseau.

Précisions sur les structures de données:

Matrice d'adjacence :

Dans un graphe orienté G=(X,E), les arcs sont définis par leur origine et leur extrémité.

On admet que chaque sommet est désigné par un nombre entier. On a X, l'ensemble des sommets de G composé des n entiers énumérés à partir de 0. Tout arc de G peut être défini par un couple d'entiers.

Soient i et j deux sommets de X. Nous appellerons MatriceArcs=(a_{ij}) la matrice carrée associée (d'adjacence) à G de dimension n^2 telle que :

$$a_{ij} = \begin{cases} 1 & \text{si} \quad (\text{i},\text{j}) \quad \text{existe} \\ 0 & \text{sinon} \end{cases}$$

Pour cette matrice d'adjacence, la structure de données retenue dans notre programme est de type matrice de booléens. MatriceArcs[i][j] est une matrice de n^2 booléens pour tout i et j sommets de X.

Matrice des capacités :

Soit $\tilde{G}$=(X,E,C,s,t) le réseau de transport associé à G.
Nous appellerons MatriceCapacites=(c_{ij}) la matrice carrée de dimension n^2 telle que c_{ij} est une valeur entière qui représente la capacité de l'arc (i,j) dans $\tilde{G}$.
Pour cette matrice des capacités, la structure de donnée retenue dans notre programme est de type matrice d'entiers.
MatriceCapacites[i][j] est une matrice de n^2 entiers pour tout i et j sommets appertenant à X.
Résultats obtenus:

Nous avons mesuré le temps de calcul pour 3 volumes de donnés:

- 5 graphes aléatoires en augmentant progressivement le nombre de sommets jusqu'à 5000.
- 50 graphes aléatoires en augmentant progressivement le nombre de sommets jusqu'à 10000.

Ces relevés ont permis de tracer les courbes jointes en annexe qui permettent de visualiser des différences en termes de temps de calcul entre l'algorithme de Dinic et celui de Edmonds-Karp.

Le principe de nos **tests** est le suivant:

Selon les 2 algorithmes, le programme calcule le flot max pour un nombre donné de graphes aléatoires. La courbe des résultats est tracée à partir des temps moyens de calcul sur une échelle logarithmique décimale allant jusqu'au nombre maximum de sommets souhaité. Le fait de tester plusieurs graphes aléatoires à chaque augmentation du nombre de sommets permet de limiter le risque de mesures sur des graphes sans intérêt (s-p chemin trop court, graphe déconnectés ou faiblement connexes etc.)

2 Partie théorique

2.1 Partie Algorithmique

Exercice 1 : Modélisation et résolution d'un probléme d'ordonnancement
1. Construire le graphe G*.

La figure 1.1 page suivante donne le graphe des contraintes de précédences.

A chaque job J_i et à chaque date de début possible (plus une unité) est associé un sommet v_{it}.
A chaque job J_i est associée une chaine d'arcs d'affectation. La capacité d'un arc d'affectation correspond au coût du démarrage w_{it} du job i à l'instant t.

i, t	0	1	2	3	4
1	0	2	5	0	1
2	1	1	2	4	-
3	1	10	2	3	3

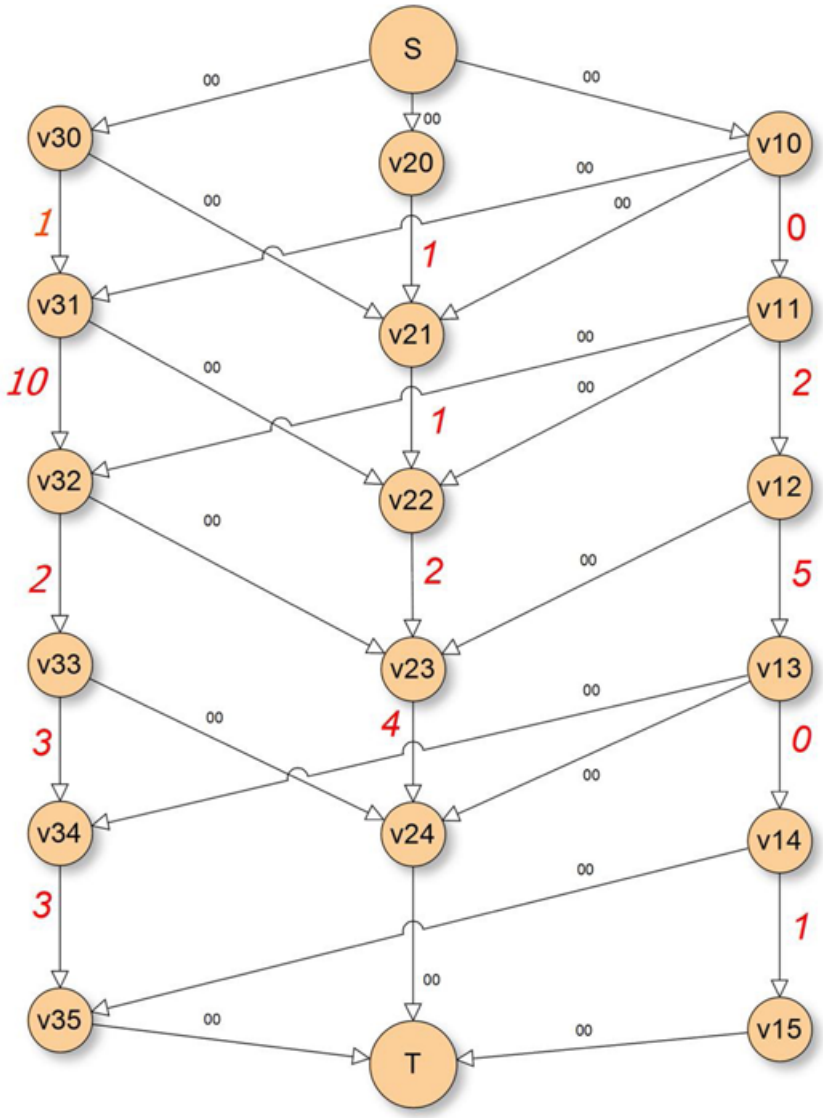


Fig 1.1 Graphe des précédences

2. Montrer qu'il existe une coupe dans G* de capacité minimale de laquelle sort un et un seul arc d'affectation par job.

Pour un graphe G =(X,E,s,t), une coupe est donnée par une partition de X en deux sous-ensembles disjoints X = Y U Z tels que **s** ∈ Y et **t** ∈ Z .

La capacité de la coupe est égale à la somme des capacités des arcs avants.

Pour montrer qu'il existe une coupe de capacité minimale dans un graphe on peut utiliser la propriété suivante :

Lorsque le flot est maximum entre la source et le puit, la valeur du flot est égale la capacité de la coupe minimale.

1) On recherche le flot maximum dans G*, qui initialement est nul. Pour cela on recherche les chaines améliorantes et on augmente le flot tant que cela est possible.

- au depart flot f=0.
- avec la chaine amliorante : (s,v_{20},v_{21},v_{22},v_{23},v_{24},t), on augmente le flot de 1, les arcs (v_{20},v_{21}) et (v_{21},v_{22}) sont saturés, f=1.
- avec la chaine améliorante : (s,v_{30},v_{31},v_{32},v_{33},v_{34},v_{35},t), on augmente le flot de 1, l'arc (v_{30},v_{31}) est saturé, f=2.
- avec la chaine améliorante : (s,v_{10},v_{31},v_{32},v_{33},v_{34},v_{35},t), on augmente le flot de 1, l'arc (v_{32},v_{33}) est saturé, f=3.
- avec la chaine améliorante : (s,v_{10},v_{31},v_{22},v_{23},v_{24},t), on augmente le flot de 1, l'arc (v_{22},v_{23}) est saturé, f=4.
- avec la chaine améliorante : (s,v_{10},v_{31},v_{32},v_{23},v_{24},t), on augmente le flot de 2, l'arc (v_{23},v_{24}) est saturé, f=6.

Il n'y a plus de chaine améliorante, la valeur du flot max dans G* est de 6.
Donc la capacité de la coupe minimale est aussi de 6.

2) On cherche une coupe de capacité égale à 6:
Avec S={s,v_{10},v_{20},v_{30} ,v_{21},v_{31}, v_{22},v_{32},v_{23}}, $\bar{S}$={v_{11},v_{12},v_{13},v_{33},v_{14},v_{24},v_{34},v_{15},v_{35},t}, la coupe (S,$\bar{S}$) est de capacité 6 et tous les arcs associés à cette coupe ont une capacité finie et il ne sort qu'un et un seul arc d'affectation par job.

Pour une démonstration mathématique, se reporter à l'annexe Exercice 1 : "Démonstration de l'existence d'un seul arc sortant de chaque job dans la coupe min" jointe en pages annexes du présent mémoire.

3. Montrer que l'on peut associer un ordonnancement réalisable à toute coupe de capacité finie minimale. Quel est le coût de cet ordonnancement ?

Comme la coupe est finie (il n'existe pas d'arcs avants de capacité infinie) et minimale (On a déja prouvé qu'il sort un et un seul arc d'affectation par job), on conclut que toutes les contraintes sont réspectées , donc peut associer un ordonnancement réalisable et valide à cette coupe et le coût de cet ordonnancement est bien égal à la valeur de la coupe minimale.

exemple : ci-dessous l'ordonnancement associé à notre coupe min .

J	t
1	0
2	3
3	2

Le coût de cet ordonnacement :
Cout = w_{10} + w_{23} + w_{32} = 0 + 4 + 2 = 6 = C_{min}

4. Montrer qu'à tout ordonnancement réalisable correspond un coupe dont la capcité est égale au coût de l'ordonnancement.

Soit un ordonnancement réalisable (respecte toutes les contraintes) , la coupe associée à cet ordonnancement est C(S,$\bar{S}$) tél que :
S = {v_{1i},v_{2i},v_{3i}} U prec(v_{1i}) U prec(v_{1i}) U prec(v_{1i}) , la capacité de cette coupe est égale à la somme des capacités de ses arcs avants et qu'est égale aussi à la somme des coûts de démarages (w_{1i} + w_{2i} + w_{3i}).

exemple :
L'ordonnancement réalisable est :

J	t
1	0
2	2
3	1

Auquel , on associe une coupe finie C(S,$\bar{S}$) , tél que :
S={v_{10},v_{22},v_{31},s,v_{30},v_{21},v_{20}} On conclut que la somme des capacités des arcs avants est bien égale à la somme des coûts de démarages.
Coût=(w_{10} + w_{22} + w_{31})=C(S,$\bar{S}$)=12.
prec(X) : les sommets qui pr'ecedent le sommet X.

Exercice 2 : Coupes et chemins arcs-disjoints

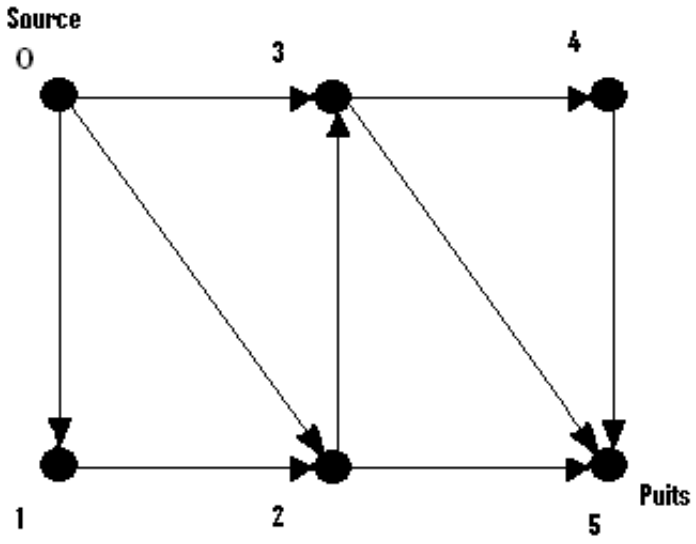


Fig 3.1 réseau adapté

On considère le réseau G(X,E,C,s,t) donné par la figure 1.
X: l'ensemble des sommets de G , X={s,2,3,4,5,t}.
E: l'ensemble des arcs du réseau,
E={(s,4),(s,3),(s,2),(2,3),(3,4),(3,t),(4,t),(4,t),(5,t)}.
C: E⇒N
$\forall i, j \in X$, C_{ij}=1
s: le sommet source.
t: le sommet puits.

1. Calculer le nombre maximum des chemins d'arcs-disjoints à partir de la source jusqu'au puits dans G.

Solution:

Théorème 1:
Soient les fonctions P et N á valeurs entières définies sur les couples de sommets distincts de G:

P(a,b) représente le nombre maximum de chemins d'origine a et d'extrémité b, deux à deux arcs-disjoints.

N(a,b) représente le nombre minimum d'arcs qu'il faut supprimer pour qu'il n'existe plus de chemin de a vers b.

Si f_{ab} est un flot de valeur maximum val(f_{ab}) dans le réseau R_{ab} alors val(f_{ab}) = P(a,b). De plus, pour toute paire de sommets distincts a et b de G, P(a,b) = N(a,b), c'est-à-dire que le nombre minimum d'arcs qu'il faut détruire pour supprimer tout chemin de a à b est égal au nombre maximum de chemins deux à deux arcs-disjoints de a à b (ce dernier résultat constitue un des théorèmes de **Menger**).

s-t chemins d'arcs-disjoints:
Deux s-t-chemins sont dits d'arcs-disjoints s'ils n'ont pas d'arcs en commun (mais ils ont au moins deux sommets communs: s et t).

Chercher un maximum de chemins arc-disjoints revient á chercher un flot à flux entiers à valeur maximum dans le réseau.

Recherche du flot maximum f_{max} :
En appliquant l'algorithme de **Ford-Fulkerson**.

Initialisation: f = 0
Recherche de chemins améliorants :
1. On prend le chemin améliorant {(s,2),(2,3),(3,t)}.
Min(1 - 0 , 1 - 0 , 1 - 0) = 1,donc on augmente le flot de 1.
f = f + 1 = 1.
2. On prend le chemin améliorant {(s,3),(3,4),(4,t)}.
Min(1 - 0 , 1 - 0 , 1 - 0) = 1,donc on augmente le flot de 1.
f = f + 1 = 2.
3. On prend le chemin améliorant {(s,4),(4,5),(5,t)}.
Min(1 - 0 , 1 - 0 , 1 - 0) = 1, donc on augmente le flot de 1.
f = f + 1 = 3.
==> Il ne reste plus de chemins améliorants (tous les (s-t) chemins comptent au moins un arc saturé).
f = f_{max} = 3

Méthode de vérification du flot maximum :

Selon le théorème du flot max, si f est un flot dans G, alors on a f_{max}=C_{min}.
Prenons la coupe min S = {s,4,2,3} et $\bar{S}$ = {5,t}.
On a bien:
$C_{min} = C_{45} + C_{4t} + C_{3t} = 1 + 1 + 1 = 3 = f_{max}$.
On a bien , P(s,t) = f_{max} = 3. (**cf théoreme 1)**
Donc, le nombre maximum des chemins arcs-disjoints à partir de la source jusqu'au puits dans G est égal à 3.

2. Enumérer toutes les **s-t** coupes dans le réseau de transport G :

Solution: Le nombre de **s-t** coupes (**nbC**) dans le graphe G est égal au nombre de partitions de l'ensemble Y=X-{s,t}.
nbC $= 2^{|Y|} = 2^{|X|-2} = 2^4 =$ **16 partitions**.

S	$\bar{S}$	Arcs avants	Arcs arrières
{s}	{2,3,4,5,t}	(s,2),(s,3),(s,4)	{}
{s,3,5}	{2,4,t}	(s,4),(3,t),(s,2),(5,t),(3,4)	(4,5),(2,3)
{s,3,4}	{2,5,t}	(4,5),(4,t) ,(3,t) ,(s,2)	(2,3)
{s,4,5}	{2,3,t}	(s,2),(s,3),(4,t),(5,t)	(3,4)
{s,3}	{2,4,5,t}	(s,4),(3,t),(s,2),(3,4)	(2,3)
{s,4}	{2,3,5,t}	(s,2),(s,3),(4,5),(4,t)	(3,4)
{s,2}	{3,4,5,t}	(s,4),(s,3),(2,3)	{}
{s,2,3}	{4,5,t}	(s,4),(3,t),(3,4)	{}
{s,2,4}	{3,5,t}	(4,5) ,(4,t),(s,3),(2,3)	(3,4)
{s,2,3,4}	{5,t}	(4,5),(4,t),(3,t)	{}
{s,2,3,4,5}	{t}	(5,t),(4,t),(3,t)	{}
{s,3,4,5}	{2,t}	(s,2),(4,t) ,(3,t),(5,t)	(2,3)
{s,2,4,5}	{3,t}	(5,t),(4,t),(s,3),(2,3)	(3,4)
{s,5}	{2,3,4,t}	(s,4),(s,3),(s,2),(5,t)	(4,5)
{s,2,5}	{3,4,t}	(s,4),(s,3),(2,3),(5,t)	(4,5)
{s,2,3,5}	{4,t}	(s,4),(3,t),(3,4),(5,t)	(4,5)

Tableau 1 - Enumération des (s-t) coupes.

3. vérifier que le nombre maximum de chemins arcs-disjoints à partir du sommet source jusqu'au puits est égal au nombre minimum d'arcs avants dans une **s-t** coupe.

A. Depuis **2.1**, P(s,t) $= f_{max} = 3$.
B. Selon le Tableau 1, le nombre minimum d'arcs avants dans une **s-t** coupe du réseau G est égal à 3 et comme toutes les capacités sont unitaires , toute **s-t** coupe possédant 3 arcs avants est minimale $C_{min} = 3$.
De (A) et (B), on a bien $\mathbf{P(s,t) = C_{min} = 3}$.

2.2 Partie Complexité

Exercice 3 :

1) Rappeler la réduction de SAT à 3-SAT

a) énoncer **SAT** et **3-SAT**

Définitions :

Une **clause** est une disjonction de littéraux. (Liaison par OU logique)

Un **littéral** désigne soit une variable booléenne, soit sa forme complémentée (ie sa négation).

Une **clause d'ordre n** est une disjonction d'au plus n littéraux. C'est à dire une proposition de la forme $v_1 \vee v_2 \vee ... v_n$.

Un problème **SAT** est un problème de décision visant à savoir si une formule logique est satisfaisable. Le probléme **SAT** s'énonce de la manière suivante:
Soit V un ensemble de variables booléennes.
Soit C un ensemble de clauses sur V. (ie ensemble de clauses constituées de littéraux appartenant à V.)
Existe-t'il une instance de V qui satisfait l'ensemble des clauses de C ?
$\exists$ V: $\{v_1, v_2 ... \ v_n\} \Rightarrow \{0,1\} \mid \forall$ i, V(C_i)=1

Le problème **3-SAT** est un cas particulier du problème SAT ou toutes les clauses sont d'ordre 3.

(b) définir la réduction :

Définitions :

Un problème est a **complexité polynomiale** s'il existe un algorithme polynomial pour le résoudre.

Un algorithme est a complexité polynomiale s'il existe un polynôme p tel que le nombre instructions exécutées par le processus soit au plus p(n), ou le maximum est pris sur toutes les données d'entrée de taille n.

Un problème de décision P se réduit polynomialement à un autre problème de décision P' s'il existe un algorithme polynomial qui transforme toute entrée u de P en une entrée u'=R(u) de P' telle que :

u ∈ Oui(P) ⇔ u' ∈ Oui(P').
u ∈ Non(P) ⇔ u' ∈ Oui(P').

(c) justifier alors que 3-SAT est NP-complet.

Définition

La **classe NP** est l'ensemble des problèmes décidables en temps polynomial par une machine de Turing non déterministe.

Un problème P de NP est **NP complet** si pour tout problème P' de NP, il existe une réduction polynomiale de P à P'.

Sachant que SAT est NP-complet (**Cf Théorème de Cook-Levin**), on justifie que 3-SAT est NP-complet en montrant que pour tout instance de SAT, on peut trouver par réduction une instance de 3-SAT équivalente.

Démonstration :

On remarque que toute clause d'ordre n $\geqslant 3$ de la forme $v_1 \vee v_2 \vee ... \vee v_n$ peut se mettre sous la forme d'une conjonction de clauses d'ordre 3:
$(v_1 \vee v_2 \vee u_1) \wedge (\neg u_1 \vee v_3 \vee u_2) \wedge ... \wedge (\neg u_{n-4} \vee v_{n-2} \vee u_{n-3}) \wedge (\neg u_{n-3} \vee v_{n-1} \vee v_n)$.

D'autre part, on remarque que toute clause d'ordre 2 de la forme $v_1 \vee v_2$ peut s'écrire sous la forme $(v_1 \vee v_2 \vee y) \wedge (v_1 \vee v_2 \vee \neg y)$.

Preuve :
Posons, C= C' ∧ C"
si y=0, on a C'= $v_1 \vee v_2$ et C"= 1 donc C ⇔ $(v_1 \vee v_2)$.
si y=1, on a C'= 1 et C"= $v_1 \vee v_2$ donc C ⇔ $(v_1 \vee v_2)$.

On peut donc remplacer chaque clause C d'un problème SAT par une conjonction de clauses à 3 littéraux connectées par de nouvelles variables:
R(C) = (x ∨ y ∨ a) ∧ (¬ a ∨ z ∨ b) ∧ (¬b ∨ t ∨ c) ∧ (¬c ∨ u ∨ d) ∧ . . .
La formule F = $C_1 \wedge C_2 \wedge ...$ est satisfaisable si et seulement si la formule R(F) = R(C_1)∧ R(C_2)∧... est satisfaisable.

On peut construire R(F) en temps linéaire et R(F) est équivalente à F (On a bien égalité entre les résultats pour un mème modèle de C).
Un problème SAT est donc polynomialement réductible a un problème 3-SAT. Donc 3-SAT est NP-complet. CQFD.

[3]Cf. *Satisfaction et Optimisation de Contraintes* de Lakhdar SAIS C.R.I.L.

(d) Application: Si un ensemble de clauses contient n_v variables, n_1 clauses à un littéral, n_2 clauses à 2 littéraux, n_3 clauses à 3 littéraux, n_4 clauses à 4 littéraux et n_5 clauses à 5 littéraux, combien les système obtenu par votre réducation contient-il de variables et de clauses ?

Soient X un ensemble de n_v variables.
Soit C un ensemble de clauses contenant:
- n_1 clauses à un littéral.
- n_2 clauses à 2 littéraux.
- n_3 clauses à 3 littéraux.
- n_4 clauses à 4 littéraux.
- n_5 clauses à 5 littéraux.

- n_1 clauses à un littéral comptabilisent n_1 clauses / 1 littéral.
- n_2 clauses à 2 littéraux comptabilisent n_2 clauses / 2 littéraux.
- n_3 clauses à 3 littéraux comptabilisent n_3 clauses / 3 littéraux.

Par réduction, on peut transformer toute clause d'ordre supérieur à 3 en n-2 clauses à 3 variables.

-> une clause à 4 littéraux ce réduit en 2 clauses à 3 littéraux
donc n_4 clauses à 4 littéraux comptabilisent 2 x n_4 clauses à 3 littéraux

-> une clause à 5 littéraux ce réduit en 3 clauses à 3 littéraux
donc n_5 clauses à 5 littéraux se réduisent en 3 x n_5 clauses à 3 littéraux

Le système obtenu par réduction contient:

$n_1 + n_2 + n_3 + (2 * n_4) + (3 * n_5)$ clauses;
et : $n_1 + (2*n_2) + (3*n_3) + (2*n_4 *3) + (3*n_5 *3)$ variables.

2) Pourquoi le principe de réduction ne permet-il pas de réduire 3-SAT à 2-SAT ?

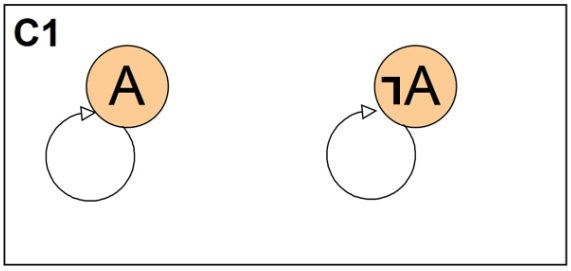
Le principe de réduction s'applique quand on peut trouver une formule équivalente à celle du problème initial par l'ajout de variables tout en diminuant l'ordre de la forme normale conjonctive.
Dans le cas d'un problème initial 3-SAT, l'ajout d'une variable ne permet pas de réduire l'ordre de toutes les clauses. On ne peut donc pas réduire 3-SAT à 2-SAT par cette méthode.

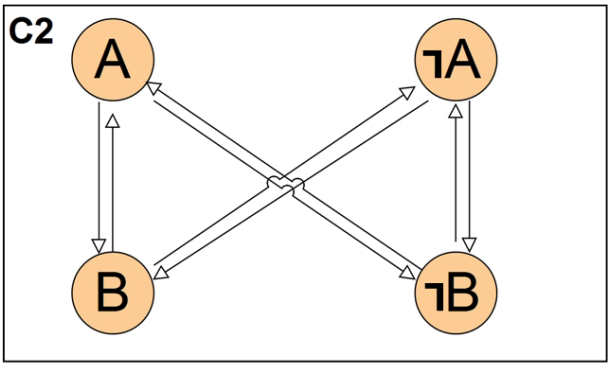
3) Prouver que 2-SAT est un probléme polynomial

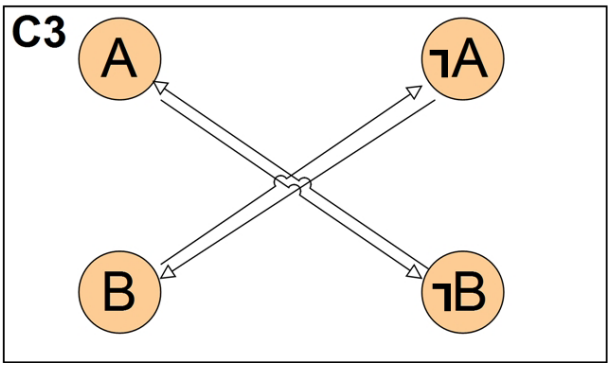
(a) vous commencerez par fabriquer trois ensembles de 2-clauses, le premier valide, le deuxiéme insatisfiable et le troisiéme contingnent, et pour chacun de ces ensembles de clauses vous construirez le graphe correspondant. Vous expliquerez comment apparait sur chacun des trois graphes la validité de l'ensemble de clauses correspondant.

Soit les ensembles de 2-clauses suivants correspondant à ces critéres:
valide : C_1 : (A $\bigvee \neg$ A)
insatisfiable C_2 : (A $\bigvee$ B) $\bigwedge$ ($\neg$A $\bigvee \neg$B) $\bigwedge$ ($\neg$A $\bigvee$ B) $\bigwedge$ (A $\bigvee \neg$B)
contingent C_3 : (A $\bigvee$ B) $\bigwedge$ ($\neg$A $\bigvee \neg$B)

Pour chaque ensembles de 2-clauses on crée un graphe où les sommets sont les variables de l'ensemble ainsi que leur négation.
Les arêtes sont représentè de la facon suivante : chaque clause de la forme $(y \vee z)$ peut être vue comme l'implication $\neg y \Rightarrow z$ et $\neg z \Rightarrow y$, donc on ajoute pour chaque clause les arêtes $(\neg y, z)$ et $(\neg z, y)$.
Une clause est satisfiable si et seulement si pour chaqu'une de ces variables propositionnelle v_i, les sommets v_i et $\neg v_i$ du graphe 2-SAT sont dans deux composantes fortement connexes distinctes.
- C1 est satisfiable car A et $(\neg A)$ ne sont pas dans la même composantes fortement connexes.
- C2 a comme composante connexe le sous graphe formé par les sommets A,B,$\neg A$ et $\neg B$. Donc il y a une boucle entre A et $\neg A$ ainsi qu'entre B et $\neg B$ ce qui revient aux 2 équivalences A $\Longleftrightarrow \neg A$ et B $\Longleftrightarrow \neg B$ ce qui est toujours faux donc C2 n'est pas satisfiable.
- C3 est satisfiable car il n'y a pas de composantes fortement connexes.


C1
A
¬A


C2
A
¬A
B
¬B


C3
A
¬A
B
¬B

(b) Vous expliciterez ensuite l'algorithme de transformation et vous évaluerez sa complexité.
algorithme 1 permettant de créer les sommets du graphe G = (S,A):

```
SOIT c de type clause
POUR c DANS ensemble de 2clause FAIRE
    x ← c.first()          // (a,b).first() return a
    y ← c.second()         // (a,b).first() return b
    IF NOT(x DANS Graphe)
        Graphe.ajoutSommet(x)
        IF NOT(¬x DANS Graphe)
            Graphe.ajoutSommet( ¬x)
        ENDIF
    ENDIF
    IF NOT(y DANS Graphe)
        Graphe.ajoutSommet(y)
        IF NOT(¬y DANS Graphe)
            Graphe.ajoutSommet( ¬y)
        ENDIF
    ENDIF
    Garphe.ajoutArc(¬x , y)
    Graphe.ajoutArc(¬y ,x)

FINPOUR
```

O(**n**) avec n nombre de clause dans l'ensemble de 2-clause

(c) Vous expliciterez ensuite l'algorithme d'exploration du graphe et vous évaluerez sa complexité en fonction de la taille de l'ensemble de clauses initial.

```
POUR x DANS Variable
    SI ¬x ϵ CFC(x)
        FAUX
    ENDIF
FINPOUR
VRAI
```

O(**mC**) avec C la complexité de la recherche composante fortement connexe et m le nombre de variables.

Rappels :

- Le rang d'un sommet s, noté r(s), est l'indice du sommet s dans la liste définissant le parcours en profondeur.
- Le point d'attache at(y) d'un sommet y , est le sommet de plus petit rang extrémité d'un chemin de G, d'origine y et contenant au plus un arc (u, v) tel que rang(u) > rang(v).
- Le rang du point d'attache est appelé rang d'attache.
- CFC : composante fortement connexe

PROCEDURE CFC(G) [1] [2]
    V $\leftarrow \phi$
    r $\leftarrow$ 0 //Compteur rang
    k $\leftarrow$ 0 //Compteur CFC
    PileVide(Π)
    TANTQUE |V| < |S| FAIRE
        choisir s $\in$ S\V
        Desc(s)
    FINTANTQUE
        La complexité de cette algorithme est de $\mathcal{O}(n+m)$
            avec n = |S| et m = |A|

PROCEDURE Desc(s)
Empiler(s,Π)
V $\leftarrow$ V $\cup$ {s}
r $\leftarrow$ r + 1
r(s) $\leftarrow$ r
Θ(s) $\leftarrow$ r //rang d'attache
POUR TOUT successeur x de s FAIRE
    SI x $\notin$ V ALORS // si x nést pas d'eja visité
        Desc(x)
        Θ(s) $\leftarrow$ min{Θ(s), Θ(x)}
    SINON
        SI x est DANS Π ALORS Θ(s) $\leftarrow$ min {Θ(s), r(x)}
    FINSI
FINPOUR
SI Θ(s) = r(s) ALORS
    k $\leftarrow$ k + 1
    REPETER
        z $\leftarrow$ Dépiler(Π)
        c(z) $\leftarrow$ k
    JUSQUA z = s
FIN SI

[1] http://www.dix.polytechnique.fr/INF431/X08-2009-2010/AmphisBW/amphi08x4.pdf
[2] http://www-master.ufr-info-p6.jussieu.fr/2005/IMG/pdf/tarjan-2.pdf

La complexité de l'exploration du graphe en fonction de la taille de l'ensemble de clauses initial est polynomiale.

(d) enfin vous justifirez l'équivalence de la réponse au problème 2-SAT et au problème qui est posé dans le paragraphe.

le probleme 2-SAT a une solution si et seulement si les clauses n'impliquent pas q'un litteral soit vrai et faux à la fois, par construction du graphe cette situation est équivalente à la présence d'une composante fortement connexe incluant un littéral et son complementaire, ce qui est calculé par l'algorithme de Tarjan.

2-SAT, d'après la définition, cherche à déterminer la satisfaisabilité d'une formule F du calcul propositionnel en forme normale conjonctive d'ordre 2.
F est satisfaisable si et seulement si pour chaque variable propositionnelle v_i de F, les sommets v_i et $\neg v_i$ du graphe 2-SAT sont dans deux composantes fortement connexes distinctes.
L'algorithme de Tarjan permet de calculer les composantes fortement connexes d'un graphe G = (S,A) orienté en $\mathcal{O}(|S| + |A|)$, donc 2-SAT est bien de classe $\mathcal{P}$.

2.3 Partie Calculabilité

Exercice 4 :

1. Comment énumérer les couples d'entiers.

Il est possible de représenter tout les couples d'entiers sur un plan en utilisant un repère cartésien. On placera le couple (a,b) au point d'abscisse a et d'ordonné b. Ce qui donne :
...
(0,5)(1,5)(2,5)(3,5)(4,5)(5,5)
(0,4)(1,4)(2,4)(3,4)(4,4)(5,4)
(0,3)(1,3)(2,3)(3,3)(4,3)(5,3)
(0,2)(1,2)(2,2)(3,2)(4,2)(5,2)
(0,1)(1,1)(2,1)(3,1)(4,1)(5,1)
(0,0)(1,0)(2,0)(3,0)(4,0)(5,0) ...

Une énumeration possible et d'énumérer les diagonales, de la plus proche à la plus éloignée de l'origine du repère, en commenant par les ordonnées les plus faibles :

20
14 19
9 13 18
5 8 12 17
2 4 7 11 16
0 1 3 6 10 15

Objectif :
obtenir la bijection qui à partir de la place d'un entier dans l'énumeration donne le couple associé.

Remarque :
on peut voir que sur une mème diagonale, la somme abscisse + ordonnée est constante, et égale au numero de la colonne.
On remarque aussi que pour tout entier n, le pied de la n-ième colonne est égal a la somme des entiers de 0 a n.
Si l'on retranche a l'énumeration ces valeurs, on obtient:

5
4 4
3 3 3
2 2 2 2
1 1 1 1 1
0 0 0 0 0 0

En conclusion, on a la bijection :

num : $\mathbb{N}^2 \Rightarrow \mathbb{N}$
(a,b) $\Rightarrow$ b + $(a+b)(a+b+1)/2$

la réciproque peut-être obtenue avec la fonction récursive définie par:

Enum : N $\Rightarrow \mathbb{N}^2$
n $\Rightarrow Enum^{-1}$(0,n)
$Enum^{-1} : \mathbb{N}^2 \Rightarrow \mathbb{N}^2$
(i,j) $\Rightarrow$ si i $\leq$ j alors $Enum^{-1}$(i+1,j-i)
sinon (i-(j+1),j)

2. Donner les fonctions de codage et de décodage f_1(z) $\Rightarrow$ x et f_2(z) $\Rightarrow$ y On notera les fonction f_1 et f_2 les fonctions de $\mathbb{N} \Rightarrow \mathbb{N}$ tel que pour tout n de $\mathbb{N}$: Enum(n)= (f_1(n),f_2(n))

2) bijection de $\mathbb{N} \Rightarrow \mathbb{N}^2$

On définit pour tout n de $\mathbb{N}$ la fonction Φn:
Φn: $\mathbb{N}^n \Rightarrow \mathbb{N}^{n+1}$ (a1,...,an) $\Rightarrow$ (a1,...,a(n-1),f_1(an),f_2(an))

Pour les indices 1 a n-1, la fonction de passage et l'identit qui est évidemment bijective.

Le passage du dernier indice de l'antćédent, aux deux derniers indices de l'image est la bijection qui énumère les couples.
Φn est donc bien bijective.

On peux alors définir pour tout entier n, une bijection de $\mathbb{N}$ sur $\mathbb{N}^n$, par composition des bijections Φn.
On obtient pour les premiers entiers:

$\mathbb{N}$ n
$\mathbb{N}^2$ (f_1(n),f_2(n))
$\mathbb{N}^3$ (f_1(n),f_1(f_2(n)),f_2(f_2(n)))
.
.
.
$\mathbb{N}^n$ ...

3. Coder les triplets. Généralisation au k-uplets. Puis aux listes de longueurs quelconques.

Cas des listes de taille quelconque
On étudie L le sous ensemble des listes de taille quelconque, contenant toute les listes ordonnées, sans doublons.

Prenons le segment réel [0;1[representé en nombre binaire.
On definit la fonction ψ definie par:
ψ: L $\Rightarrow$ [0,1[
l $\Rightarrow$ 0,r0|r1|r2|r3|r4... avec r_i= 1 si i $\in$ l
0 sinon.
ex : $\psi((0;2;4;5)) = 0{,}101011 \Rightarrow \psi((1;7)) = 0{,}01000001$

Comme L est ordonnée et n'a pas de doublon, ψ est une bijection.
On sait que [0;1[n'est pas énumerable. La bijection ψ prouve donc que L n'est pas énumerable.

Comme L est inclue dans l'ensemble des listes de taille quelconque, **l'ensemble des listes de taille quelconque ne peux pas ètre énumerable**.

4) Enumeration des fonctions C :

Toute fonction écrite en C, et ayant une syntaxe correcte peux ètre contenu dans un fichier de longueur finie. Pour une taille finie, on peut lister tous les fichiers dont l'analyse syntaxique C est correcte. (Il suffit de les compiler un par un).

Les fichiers de longueur n étant en nombre finis quelque soit la valeur de n (2^n pour ètre précis), on peux énumerer de manière évidente l'ensemble F de ces fichiers:

Le fichier vide a le numero 1.
On énumère ensuite les fichiers de longueur 1.
Puis les fichiers de longueur 2.
Puis les fichiers de longueur 3. ...
Comme l'ensemble des fonctions C peut ètre mis en bijection avec un sous ensemble de F énumerable, **les fonctions C sont énumérables**.

Cas des fonctions C qui bouclent:
Du point de vue Informatique, le problème consistant à savoir si une fonction va boucler ou non est indécidable (le programme qui détecte les procédure qui bouclent n'existe pas. Le test de l'arret est indécidable). **On ne peut pas créer une procedure pour énumérer les fonctions C qui bouclent.**

3 Partie pratique sur les algorithmes de flots

Exercice 5 : La méthode de Edmonts-Karp et celle de Dinic

Quelques procédures :

1. Procédure qui qui retourne un plus court chemin en nombre d'arcs de s à p : **getPCC(int o, int a)** : Recherche d'un plus court chemin en nombre d'arcs entre 2 sommets origine et arrivée.
Adaptation de l'algorithme de Dijkstra dans le cas ou tout les arcs ont pour valeur 1.Ne prend pas en compte les arcs de capacité nulle.

2. Programmer l'algorithme d'Edmonds-Karp :
applyEdmondsKarp() : A chaque itération, l'algorithme d'Edmonds-Karp trouve un plus court chemin et augmente le flot autant que possible sur ce chemin jusqu'à ce que tous les (s-t) chemins soient saturés.

3. Programmer l'algorithme Dinic : **applyDinic()** : A chaque itération d, l'algorithme de Dinic calcule tous les plus courts (s-t) chemins en nombre d'arcs de longueur d et augmente le flot sur tous ces chemins. Il prend du temps pour calculer les plus courts chemin mais l'augmentation du flot est optimale à chaque itération.