Recherche de cycles

cours de Graphe 2009 d'Eric Thierry, rapport de Brunie Nicolas

17 avril 2009

Table des matières

1 Introduction **2**

2 Définition **2**

3 Cas de base **2**
3.1 existence d'un cycle simple 3

4 Circuit de taille minimum **4**
4.1 Premier algorithme 4
4.1.1 Complexité et résultat 5
4.1.2 Pourquoi une telle imprécision 5
4.2 Algorithme de circuit minimum 7
4.3 analyse de complexité 8
4.4 cas des graphes orientés 9
4.5 Recherche de cycle et recherche de triangles 9
4.5.1 Algorithme naïf 9
4.5.2 Recherche de triangle par arbre couvrant 10
4.5.3 Méthode matricielle 10

5 Recherche de cycle de longueur fixée **11**
5.1 Recherche d'un chemin élémentaire de longueur fixée 11
5.2 Une esquisse d'algorithme 12

6 Cycle sans corde **12**
6.1 Lemme de base 12
6.2 algorithme 13
6.3 Le cas des trous impairs 15

7 Problème du cycle hamiltonien **15**
7.1 Définition 15

8 Conclusion **16**
8.1 Tableau Récapitulatif 16

9 Bibliographie 16

1 Introduction

Dans ce rapport nous allons voir quelques uns des problèmes de recherche de cycles. La recherche de cycle dans un graphe est d'une incroyable variété : cycle simple, élémentaire, avec ou sans corde, dans un graphe orienté ou non. Nous verrons ici qu'une partie de ce large problème. Nous étudierons des algoritmes pour décider de l'existence d'un cycle, pour trouver un cycle minimum, puis pour trouver un cycle sans corde de longueur supérieure à 5 (trou ou *hole*). Je me concentrerais plus particulièrement sur le cas non-orienté, plus simple, tout en présentant quelques résultats sur le cas orienté.

2 Définition

Tout au long de ce rapport, on considère un graphe G=(V,E), V est l'ensembles de sommets et E l'ensemble des arêtes. n est le cardinal de V et m le cardinal de E.
On considèrera souvent que les sommets sont labélisés par des entiers $i \in \{1...n\}$ uniques.
On distingue plusieurs forme de cycle :
- **Cycle (cas général)** : Un ensemble d'arêtes $\{e_1, ..., e_n\}$ distinctes deux à deux, telle que $\forall i$, $\exists$ u v w, tel que $e_i = (u, v)$ et $e_{i+1} = (v, w)$, autrement dit les arêtes forment un chemin. De plus on a une fermeture du chemin, $e_1 = (u, v)$ $e_n = (w, x) \rightarrow u = x$
- **cycle élementaire** : c'est un cycle qui ne repasse jamais deux fois par le même sommet.
- **cycle sans corde (*hole*)** : pour tout couple de sommet du cycle non adjacent dans ce cycle aucune arête ne les relie.

3 Cas de base

Nous allons d'abord nous intéresser au cas de base. On cherche un cycle sans plus de précision.

Pour cela on peut se servir d'une des définitions des arbres :

Définition 1 *Un arbre est un graphe acyclique maximal pour le nombre d'arête.*

Ce qui nous permet d'en déduire le lemme trivial suivant :

Lemme 1 *Une composante connexe de cardinal n contient un cycle* $\iff$ *elle a n arêtes ou plus.*

Preuve Soit T un arbre couvrant pour cette composante, T a (n-1) arêtes. Il existe donc une arête e = (u,v), de la composante ($u, v \in$ *composante*) , qui n'est pas dans T. On a donc une chemin de u à v dans T (par définition d'un arbre et d'un arbre couvrant) qui ne contient pas

e, puisque e n'est pas dans T. et donc ce chemin plus e forme un cycle.
Pour déterminer si un graphe possède ou non un cycle on peut donc appliquer ce lemme à chacun de ses composantes connexes. Si elles contiennent plus que leur cardinal d'arrêtes alors le graphe contient un cycle.
Ces résultats triviaux sont néanmoins utiles, on peut par exemple chercher un cycle en cherchant des arêtes de retour, des arêtes transverses ou des arêtes de progrès dans un arbre de parcours en profondeur (DFS) du graphe.

3.1 existence d'un cycle simple

La recherche d'un cycle simple se fait simplement en temps O(n). On effectue un parcours en profondeur. Dès que l'on rencontre une arête de retour, une arête de progrès ou une arête transverse, on a affaire à un cycle. On peut aussi faire un parcours en largeur (BFS).
Algorithme : existence de cycle : On part d'un sommet auquel on donne le niveau 0 et on visite ses voisins, auxquels on associe le niveau 1 et ainsi de suite :

- Initialement on a une file vide
- on alloue à chaque sommet le niveau NULL
- on place au hasard un sommet v dans la file, niveau(v) = 0.

Tant qu'il y a des sommets dans la file :

- on retire la tête de file : sommet u
- on parcours tous ses voisins
- pour chacun d'eux on applique ce qui suit :
  - s'il est de niveau strictement inférieur, on ne fait rien
  - s'il est de niveau égal ou supérieur, on vient de trouver un cycle. On retourne TRUE
  - s'il est de niveau NULL, on lui assigne le niveau niveau(u)+1 et on le place en fin de file.

Lorsque la file est vide, on retourne FALSE, le graphe est acyclique.

Remarque 1 *Une partie des preuves concernant ce théorême seront faites plus loin. On peut cependant déjà noter que :*

- *la file est rangée par niveau croissant : c'est une propriété du parcours en largeur (utilisée dans dijkstra sur graphe avec arête de poids unitaire).*
- *si on rencontre un voisin de niveau inférieur strictement, ce voisin nous aura traité lors de son étape. C'est donc un des pères du sommet courant dans ce pacours.*
- *On fait un nombre O(n) étapes. En effet, on considère au plus chacun des n sommets et au plus n arêtes. Si on considère plus de n arêtes, un cycle existe et il aura été découvert*
- *L'algorithme précédent s'applique à une composante connexe. On peut le généraliser. Les sommets sont labélisés de 1 à n. On forme au début une liste globale de tous les sommets et on construit un tableau de taille n où l'entrée i contient un pointeur vers la cellule du sommet i dans la liste globale . A chaque fois qu'on considère un sommet avec un niveau NULL (y compris le sommet de départ v), on le supprime de la liste, ceci se fait en temps constant en accédant à sa cellule de la liste par le tableau. Il faut donc une liste doublement chaîné[1]. Une fois que notre file est vide, on regarde si la liste globale l'est*

[1] suppresion en temps constant

```
FRONT(v,k,level);
Pour chaque u dans V faire level(u) := nil;
level(v) := 0;
push(v);
tant que file non vide faire
    u = pop();
    pour tout w dans voisin(u)
        si level(w) := nil alors
            level(w) := level(u) +1;
            push(w);
        sinon
          si level(u) <= level(w) alors
              k(v) := level(u);
              return
    fin pour tout
fin tant que.
k(v) := infinity
```

Fig. 1 – 4.1. circuit presque minimum

(test en tps constant au plus n fois). Si oui on arête, sinon on place le premier sommet de cette liste dans la file (en le supprimant puisqu'il a le niveau NULL).

Cette algorithme partage beaucoup de propriétés avec l'algorithme de [Ita77] pour la recherche de cycle minimum présenté dans la partie suivantes. Une partie des preuves seront faites à cette occasion.

4 Circuit de taille minimum

On cherche un cycle de longueur minimum. Tout d'abord nous nous intéressons à la recherche d'un cycle minimum dans un graphe non orienté. Par définition un cycle minimal est un cycle élèmentaire qui ne repasse pas deux fois par un même sommet (sauf fermeture du cycle).

4.1 Premier algorithme

Le premier algorithme présenté provient de [Ita77]. Il se subdivise en plusieurs procédures. La première procédure permet simplement de trouver un cycle *presque minimum*, c'est à dire dont la taille n'excède pas celle du minimum de plus de 1 (min ou min + 1).
Cette première procédure (4.1) sert à déterminer la longueur minimum d'un cycle à partir d'un sommet v. Le cycle ne passe pas forcèment par le sommet v. On obtient seulement une borne sur la longueur d'un circuit dans la composante connexe de v.

L'algorithme 4.1 qui s'appuie sur une file[2] est basé sur un parcours en largeur, puisque à chaque fois que l'algorithme est appelé sur un sommet on considère chacun de ses voisins (empilement / *push* dans la file).

4.1.1 Complexité et résultat

Lors d'un parcours en largeur on est amené à considérer successivement chaque sommet, puis chaque arrête partant de ce sommet. Mais dans notre cas nous considérerons au plus n arêtes. En effet (cf introduction), un graphe acylique ne peut pas avoir plus de —V—-1 arêtes. Comme nous partons dans une composante connexe (celle de v), l'exécution considèrera chaque sommet une fois et chaque arête s'il n'y a pas de cycle, donc O(n) étapes. S'il y a un cycle, l'exécution s'arêtera à la $|V|^{ème}$ arête au plus tard.

Supposons qu'on ait déjà considéré p sommet Ainsi que (p-1) arêtes (entre les sommets déjà considérés). Si l'on rencontre une nouvelle arête, deux cas se présentent :
- soit elle pointe vers un sommet non encore rencontré (de level nil)
- soit elle pointe vers un sommet déjà rencontré (c'est le cas si tout les sommets on été visité)

Si on a visité (n-1) arêtes sans rencontrer de cycle. Alors on a visité les n sommets du graphes ou on a trouvé un cycle. La prochaine arête si elle existe forme un cycle (détecté par l'algorithme).

Donc on aura une complexité en O(n) puisque dans le pire des cas il faudra considérer un nombre constant de fois chaque sommet pour trouver un cycle (dans le cas ou le graphe est un cycle par exemple, on devra considérer tous les sommets).
Cet algorithme termine dès qu'il considère une arête vers une sommet qu'il a déjà traité.
On a donc une complexité de $O(V^2)$. Puisqu'on va exécuter l'algorithme à partir de chaque sommet du graphe.
Cette algorithme ne peut pas vraiment être considéré comme un algorithme de recherche de cycle minimum, comme nous le verrons dans la suite. Néanmoins il nous donne des informations sur chacun des sommets qui vont nous être utile : le kmin définit pour chaque sommet.

4.1.2 Pourquoi une telle imprécision

Penchons nous un peu sur le cycle trouvé par 4.1. Cette procédure nous donne avec le kmin un borne, on sait que le cycle minimum est soit de longueur 2 * kmin +1 ou 2 * kmin + 2 (preuve plus loin). La figure 2 présente un exemple d'exécution de l'algorithme :
- le niveau de chaque sommet est initialisé à **nil**
- Le sommet vert, (0) est le sommet de départ : niveau(0) = 0
- la première étape consiste à considérer un à un chacun des voisins de (0) [ici en bleu]
- dans l'ordre (arbitraire) noté sur le schéma chaque sommet va se voir attribuer un niveau de 1, et être placé dans la file
- (0) est ensuite enlevé de la file
- On considère ensuite les sommets par ordre d'arrivé dans la file.

[2] premier entrant premier sortant

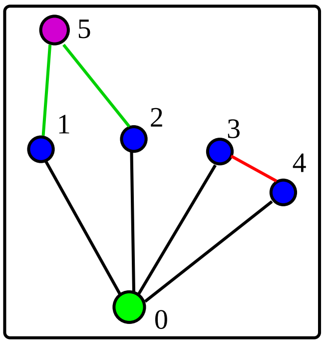


FIG. 2 – cas d'imprécision

– l'exécution sur (1) permet de donner à (5) le niveau 2,
– (0) n'est pas considéré comme voisin de (1) parce qu'il a un niveau strictement inférieur
– (1) est retiré de la file.
– (2) est considéré
– l'exécution sur 2 stoppe l'algorithme, en effet (5) est voisin de 2 avec un niveau supérieur . On vient donc de trouver un cycle =¿ k((0)) = 1.

On remarque tout de suite à l'aide du schéma que ce cycle n'est pas un cycle minimum. Le cycle (0,3,4) est en effet plus court. Mais il n'a pas été considéré par l'algorithme à cause de l'ordre d'exécution.

Lemme 2 *L'exécution de l'algorithme* **??** *sur un graphe un sommet G=(V,E). Si le cycle minimum trouvé est de longueur impaire (2 * kmin + 1), alors il est minimum dans graphe. Si le cycle maximum trouvé est de longueur paire* $= 2 \times kmin + 2$*. Alors la longueur du cycle minimum est égale à* $2 \times k + 1$ *ou* $2 \times k + 1$*.*

Si le cycle minimum trouvé est de longueur impaire, cela veut dire que les deux sommets reliés par la dernière étape de l'algorithme sont au même niveau (niveau = kmin).
Ceci implique qu'ils sont à la même distance du premier sommet ((v) dans l'algorithme). La dernière étape de l'algorithme consiste seulement à considérer l'arête entre les deux.
On ne peut pas avoir de cycle plus court.
Si c'était le cas alors :

– soit ce cycle se refermerait sur deux sommets de niveaux strictement inférieurs, donc l'algorithme l'aurait déjà trouvé puisqu'il traite les sommets par niveau croissant à cause de la structure de la file.

– soit ce cycle relierait un sommet de ce dernier niveau et un sommet du niveau précédent. Dans ce cas il aurait été détecté par l'algorithme lors du traitement du sommet de niveau précédent.

En effet

Lemme 3 *Lors de l'exécution de l'algorithme, les sommets sont traités par* ***niveau / level*** *croissant.*

La preuve est triviale est provient du fait qu'il s'agit d'un parcours en largeur.
Par récurrence.
Initialement on a qu'un sommet de niveau 0.
Supposons la file constituée de sommets de niveau n et n + 1 rangés par ordre croissant (au moins 1 sommets de niveau n).
La prochaine étape considèrera un sommet u de niveau n (ordre) donc on n'ajoutera à la file que ces voisins qui n'ont pas déjà été traités (de niveau **nil**) et auxquels on affectera le niveau niveau(u)+1 = n+1 en les plaçant en bout de file. On gardera donc une file ordonnée par niveau et on n'aura une amplitude sur les niveaux dans la file d'au plus 2.

Remarque 2 *Triviallement on ne peut pas avoir d'arête entre deux sommets dont la différence de niveau est supérieur strictement à 1.*
En effet soit u et v, tel que niveau(u) ¡ niveau(v) - 1 et $(u, v) \in E$.
- *l'algorithme étiquette d'abord u en traitant un voisin de u de niveau niveau(u) - 1*
- *u est alors placé dans la file (v ne s'y trouve pas puisque de niveau supérieur à u)*
- *l'algorithme arrive à l'étape de traitement de u*
- *il parcourt les voisins de u*
- *le niveau assigné à v n'a pu être au plus que level(u) + 1 puisqu'à cause de la structure de file on n'a traité que des sommets de niveau inférieur ou égal à u. Ou bien v a toujours le niveau* ***nil****, dans quel cas, le traitement des voisins de u donnera à nil le niveau level(u) +1*

Dans le cas ou l'algorithme trouve un cycle pair, c'est à dire reliant deux sommets de niveaux différents (au plus d'un, cf ci-dessus). On n'est pas sûr que ce soit un cycle minimum (cf exemple 2).
On peut tout de même affirmer que si un cycle de taille inférieure de plus de 2 à ce cycle existait, il aurait été détecté par l'algorithme. En effet il relierait des sommets de niveaux strictement inférieurs aux deux sommets considérés et donc il aurait été traité préalablement.
Donc finalement, bien que la preuve ait été fastidieuse par souci du détail (le principe étant on ne peut plus simple) nous disposons d'un algorithme capable de trouver un cycle minimum dans le cas où celui-ci est impair et un cycle presque minimum sinon.

4.2 Algorithme de circuit minimum

On a vu que l'algorithme précédent ne permettait pas de trouver un circuit minimum. Néanmoins il va nous être utile. On définit

$$F(v) = \{u\} \bigcup \{x | x \in V, xestdanslafileetlevel(x) = level(u)\}$$

Quelques explications : L'ensemble F(v) est définit à la fin de l'exécution du premier algorithme FRONT sur v. On vient de trouver un cycle. **u** est le dernier sommet considéré (le dernier sommet qu'on a retiré de la file). On sait qu'il existe un cycle entre u et un autre sommet (possiblement d'un niveau supérieur) et que ce cycle n'est pas forcèment minimal puisqu'on pourrait avoir un cycle entre deux sommets du même niveau que u (cf paragraphes précédents). L'idée est alors de regrouper ces sommets dans un ensemble et de tester si un couple de ces

sommets forme une arête. Si c'est le cas, on vient de trouver un cycle.
Voici donc l'algorithme complet de recherche d'un cycle minimum :

```
1.   CIRCUIT_MINIMUM(G=(V,E)):
2.   pour chaque v dans V, calculer Algo(v);
3.   on cherche kmin;
4.   if kmin = infini alors pas de cycle, return;
5.   pour chaque v dans V tel que k(v) = kmin faire
6.      construire F(v)
7.      on prépare une réprésentation de F(v) en vecteur ordonné
8.      a = EDGE(F(v))
9.      si a != nil alors le circuit minimum est de taille 2*kmin + 1, return
10.  longueur = 2 * kmin + 2;
```

Déroulement de l'algorithme :

- on exécute FRONT sur chacun des sommets
- on définit k(u) (kmin de u) pour chaque $u \in V$
- on détermine le k minimum : kmin parmi les sommets
- si kmin est infini, cela veut dire qu'il n'y a aucun cycle dans aucune des composantes connexes de G
- sinon on construit F(v)
- la procédure EDGE(S) détermine s'il y a une arête entre deux sommet de S
- s'il n'y en a un, elle rend l'arête. On a donc un cycle de longueur impaire (cycle que l'on referme entre deux sommets de même niveau lors de l'exécution de FRONT sur un sommet du cycle)
- sinon on continue
- Si aucune arête n'existe dans tout les F(v) des k(v) minimum :
  on a qu'un cycle de longueur paire (celui détecté dans FRONT)

Remarque 3 *Nous venons de décrire un algorithme pour trouver un circuit minimum dans un graphe. Cet algorithme donne la longueur de ce circuit minimum. Pour effectivement exhiber un circuit minimum il suffit d'exécuter la procédure **FRONT** sur un sommet de kmin appartenant au cycle. Si le cycle trouvé est de longueur impaire, on prend comme sommet celui dont l'itération ligne 5 à 9 à mener à trouver le cycle. Sinon on prend n'importe quel sommet dont le k est minimum.*
L'exécution de FRONT va nous permettre d'obtenir le cycle. On détermine par Algo0 l'ensemble F(v). Si le cycle est impair, l'arête a rendue par EDGE(v) lors de l'exécution de MIN_CIRCUIT ferme le cycle, on par de ces deux extrémités et on remonte vers v (niveau décroissant). Si le cycle est pair, l'exécution de Front nous donne directement les deux sommets finaux, on a plus qu'à remonter les niveaux vers v.

4.3 analyse de complexité

Nous avions déjà vu que l'exécution de FRONT sur les n sommets se faisait en $O(n^2)$. La procédure EDGE(S) (complètement décrite dans [Ita77]) raffine la recherche d'un couple adjacent dans pour qu'on obtienne une complexité moyenne de O(n) tout en ayant une complexité dans le pire des cas de $O(n^2)$. Le reste des étapes sont en temps constantes et exécutées au plus n fois (un seul niveau de boucle). On a donc finalement une complexité moyenne en O(n^2) et O(n^3) dans le pire des cas.

4.4 cas des graphes orientés

La "métathéorie" qui annonce que le cas orienté est plus difficile que le cas non-orienté trouve ici une illustration.
Les graphes orientés (traités deans [JBj01]) ont de nombreuses applications, comme la modélisation se circuit électronique / booléen. Pour la recherche d'un circuit orienté minimum, [Ita77] propose deux algorithmes un en $O(n^2 \times log(n))$ [temps moyen] et un second en temps $O(n^{log(7)} \times log(n))^3$ (basé sur la multiplication booléenne de matrice).

4.5 Recherche de cycle et recherche de triangles

Une des techniques présentées dans [Ita77] pour la recherche de cycle est l'utilisation d'un graphe intermédiaire. En fait on dispose d'un algorithme qui transforme un graphe G en un graphe G' **??**, ceci à partir de l'ensemble F(v) décrit précedemment. Nous ne nous étendrons pas sur la méthode, juste savoir que la recherche de triangle a un intérêt certains dans ce cas là. On trouve les cycles minimum de G en cherchant les triangles dans G'. La recherche de triangle apparait plus simple que la recherche de cycle générique. Nous allons étudier 2 techniques de recherche de triangles qui utilisent 2 approches assez différentes.

Il existe plusieurs méthodes pour chercher un triangle dans un graphe. Tout d'abord un algorithme naif sans intérêt. Ensuite, la méthode matricielle qui utilise la matrice d'adjacence pour déterminer si il y a un cycle entre 3 sommets. La seconde méthode utilise un arbre couvrant et un résultat liant cet arbre à l'existence d'un triangle. Dans le cas général cette méthode a une de complexité : $O(m^{\frac{3}{2}}))$.

4.5.1 Algorithme naïf

L'algorithme le plus simple de recherche de triangles est le suivant :

```
G = (V,E) un graphe simple
1.  Pour chaque arête e=(x,y) dans E faire
2.    chercher z tel que x,y,z soit distinct et (x,z),(y,z) dans E
3.    si un tel z est trouvé : OK
4.    sinon on continue
```

La complexité est de $O(n \times m)$: O(m) itérations et chaque itérations se fait en O(n) si on dispose d'une matrice d'adjacence.

[3] l'article data de 1977, on connait aujourd'hui un algorithme en (O$n^{2,376}$) pour la multiplication de matrice [DC87]

4.5.2 Recherche de triangle par arbre couvrant

[Ita77] propose un autre algorithme basé sur la recherche dans un arbre couvrant (*spanning tree*).

Lemme 4 *Soit G un graphe et T un arbre couvrant de G (enraciné) .*
Il existe un triangle dans G qui contiennent une arête de $T \equiv$ *Il existe une arête hors de l'arbre (x,y) tel que (père(x),y)* $\in E$

Preuve : $\leftarrow$: (x,y) $\in E$ et (père(x),y) $\in E$, hors par définition de l'arbre couvrant (père(x),x) $\in E$ donc (x,père(x),y) forme un triangle avec l'arête (x,père(x)) $\in T$.
$\rightarrow$: On suppose disposer d'un triangle (x,y,z), avec (x,z) $\in T$. On suppose sans perte de généralité que x = père(z).
Deux cas se présente alors :
- $(y, z) \notin T$, on a alors par définition du triangle (père(z)=x,y) $\in E$ et (z,y) vérifie la condition de non-appartenance à T.
- $(y, z) \in T$, on a alors y=père(z).
  Comme $(y, z) \in T$, on a soit y=père(z), soit z=père(y).
  Par définition, z n'a qu'un père (x), y ne peut donc pas être son père.
  On en déduit z=père(y).
  T acyclique permet de déduire, $(x, y) \notin T$.
  On se retrouve donc avec $(y, x) \notin T$, et (père(y)=z,x) $\in E$, CQFD.

Pour la recherche de triangle, on construit un arbre couvrant enraciné, on cherche une arête qui vérifie le lemme. S'il y en a une, on a un triangle. Sinon aucune des arêtes de l'arbre ne font parties d'un triangles, on les supprime de G et l'on recommence.
Chacune des construction de l'arbre couvrant et de la recherche se fait en O(e).
[Ita77] donne la complexité de cette algorithme : O($e^{\frac{3}{2}}$) dans le pire des cas.

4.5.3 Méthode matricielle

Il existe une autre méthode recherche de triangle : la multiplication de matrice.
En effet, soit G=(V,E) un graphe de matrice d'adjacence M.
On réalise l'opération suivante :

$$B = (M \otimes M)ANDM$$

où $\otimes$ est une multiplication booléen, et **AND** le ET logique.

Lemme 5 $B_{i,j} = 1 \iff$ *un triangle passe par l'arête (i,j)*

Preuve La preuve est assez simple, mais on se permet de la détailler ici : $\leftarrow$ Si un triangle passe par l'arête (i,j) cela veut dire que $M_{i,j} = 1$ et qu'on a un chemin de longueur deux allant de i à j donc que $(M \otimes M)_{i,j} = 1$ donc lorsqu'on fait le ET logique de ces deux termes, on trouver bien 1.
$\rightarrow$ Si $B_{i,j} = \sum_{k=1}^{n} M_{i,k} \otimes M_{k,j} \ AND \ M_{i,j}$ vaut 1, alors $M_{i,j}$ vaut 1. On a donc une arête entre

i et j.
De plus on a au moins un terme de la somme qui est non nul, et comme on suppose travailler sur des graphes simples (sans boucle) $\exists k$, $k \neq i$ et $k \neq j$, tel que $M_{i,k} = 1$ et $M_{k,j} = 1$, ou encore on a un chemin de longueur 2 entre i et j (n'utilisant pas (i,j)). On a donc bien un triangle (i,j,k).
Compléxité On obtient donc un algo de recherche d'un triangle en O(M(n)). M(x) étant la complexité d'une multiplication booléenne de matrices carrés de taille x. Lors de l'écriture de [Ita77], M(x) valait $n^{log(7)}$ soit environ $n^{2.807}$ (algorithme de Strassen). On connait aujourd'hui un meilleur algorithme : l'algorithme de Coppersmith-Winograd en $O(n^{2.376})$ (pour plus de détails : [DC87]). On a donc accès à cette complexité pour notre recherche de triangle et donc indirectement notre recherche de circuit minimum.

5 Recherche de cycle de longueur fixée

Cette partie est purement informative et ne contient aucune demonstration. La recherche des cycles de longueurs fixées constitue une des parts non négligeables de la recherche de cycle, et nous allons donc au moins l'aborder succintement pour avoir une idée des algorithmes connus.

5.1 Recherche d'un chemin élémentaire de longueur fixée

De nombreux algorithmes de recherche d'un cycle élémentaire utilise la recherche d'un chemin élèmentaire. Intéressons-nous donc à un algorithme pour trouver des chemins de longueurs fixées.
Placons nous dans le cas général :
- On cherche un chemin de longueur k
- ce chemin est élementaire : il ne repasse pas deux fois par un même sommet ou n'utilise deux fois une même arêtes.

Une méthode connue pour obtenir de tel chemin est la multiplication de Matrice.
Soit G=(V,E) Notre graphe, On suppose disposer d'une matrice $((A)_{(i,j) \in \{1 \dots |V|\}})$ (matrice d'adjacence) $A_{i,j} = 1$ si et seulement si $(i, j) \in E$.
Si l'on considère la matrice $B = A^2$, on se rend compte que $B_{(i,j)}$ est égal au nombre de chemin de longueur 2 entre i et j.
En effet
$$B_{i,j} = \sum_{k=1}^{|V|} A_{i,k} \times A_{k,j}$$
Or $A_{i,k} \times A_{k,j}$ vaut 1 si et seulement si $A_{i,k}$ et $A_{k,j}$ valent 1 c'est à dire que (i,j,k) est un chemin de longueur 2 de i à j. Donc ce produit est égale au nombres de ces chemins.

On peut ainsi recommencer le processus k fois pour obtenir pour chaque couple (i,j) le nombre de chemin de longueur k qui relie i à j [4]. Cette méthode a cependant un inconvénient : on obtient pas des chemins élémentaires. En effet, les chemins obtenus peuvent repasser par les même sommets, si par exemple à une étape on obtient $M_{i,i} = 1$, c'est à dire que l'on a une

[4]Comme d'habitude, on assimile i au sommet dont le label est i (idem pour j)

boucle de retour sur i. Alors à l'étape suivante certains des chemins (comptés) pour aller de i à ces voisins seront des chemins qui bouclent sur i.
Pour éviter cela, on nettoie la diagonale à chaque itération (en y plaçant des 0). Cela évite de repasser par un même point.

5.2 Une esquisse d'algorithme

Un des algorithmes existant pour la recherche de cycles de taille fixée fonctionnent comme suit :
- on cherche tous les chemins de longueur k et k+1
- On se sert de la théorie des sous-collections représentatives :au lieu de tester directement l'intersection de deux ensembles on fabrique des sous-ensembles qui intersectent entre eux si et seulement si les ensembles de départ intersectent.
- On utilise cette théorie pour trouver des chemins qui n'intersectent qu'en leur extrémités, on obtient donc ainsi des cycles par recollement des chemins

On peut trouver une étude précise de cette recherche dans [NA97] qui donne aussi des résultats algorithmiques : un algorithme en $O(n^w \times log(n))$ [5] pour la recherche de cycle de longueur fixée, $O(m^{2-\frac{2}{k}})$ pour des cycles pairs et $O(m^{2-\frac{2}{k+1}})$ pour des cycles de longueur impaire.

6 Cycle sans corde

Les cycles sans cordes est l'une des classes de cycle très étudiées. Plus particulièrement les trous de longueur impaire (*hole*) qui sont des cycles sans cordes de longueur impaire ont été particulièrement étudiés.
Ceci à cause de la conjecture forte des graphes parfaits, (prouvées récemment) : un graphe est parfait si et seulement si il ne contient pas de trou de longueur impaire. Avec une seconde conjecture [6] sur les graphes parfaits qui annonce qu'il existe un algorithme polynomial pour déterminer sur un graphe est parfait ou non. A ce jour, l'algorithme le plus rapide trouvé est en $O(n^9)$.

Plutôt que de traiter le cas des trous de longueurs impaires, je vais ici traiter le cas général de la découverte d'un cycle sans corde.

6.1 Lemme de base

Définition 2 *Dans un graphe G, Un chemin sans corde de longueur k, aussi note P_k est un chemin de longueur k tel que chaque sommet non adjacent n'est pas relié dans G.*

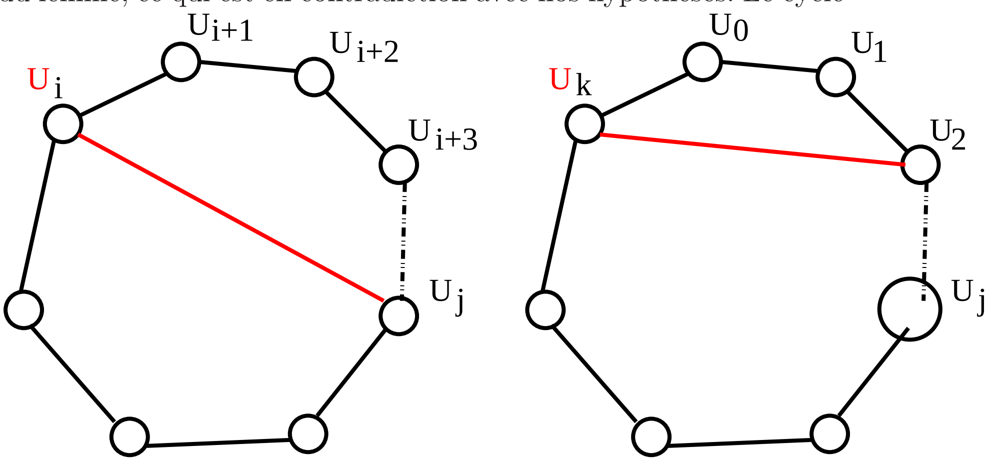
L'algorithme que je vais vous présenter (traité dans [SN]) se base sur le résultat suivant :

Lemme 6 *Un graphe G, non orienté continent un trou $\iff$ G contient un cycle $u_0u_1..u_k$, avec $k \geq 4$ tel que $\forall 0 \leq i \leq k-3$, $u_i, u_{i+1}, u_{i+2}, u_{i+3}$ et $u_{k-2}u_{k-1}u_ku_0$ sont des P_4 de G*

[5] w : coefficient de complexité d'une multilication de matrice
[6] prouvée aussi

Preuve $\rightarrow$, Trivialement si on est en présence d'un trou dans G, alors c'est un cycle sans corde, donc il vérifie les conditions (tout chemin de taille 4 inclus dans le trou n'admet pas pas de corde).
$\leftarrow$ On suppose que G contient un cycle C, vérifiant les conditions du Lemme. On prend pour C le cycle minimal de G vérifiant ces conditions.
Le fait que G vérfie les conditions du lemme implique qu'il a au moins 5 sommets : 4 sommets formant un chemin sans corde et donc au moins un sommet de plus pour refermer le cycle.
Le cycle C est alors sans corde.
Raisonnons par l'absurbe, supposons que C est au moins une corde. On peut attribuer à chaque corde une longueur ($|(u_s, u_t)| = |s - t|$). Soit u_i et u_j les extrémités de la corde la plus courte (6.1), $j > i$.
On a $|i - j| \geq 4$. En effet $j \geq 4$ car toutes les suites de 4 sommets du cycle sont des P_4 (ceci inclut $u_0u_1u_2u_3$) et que $u_{j-3}..u_j$ est un P_4 par hypothèse.

De cela, on peut déduire que $u_{j-2}u_{j-1}u_ju_i$ est un P_4. Ceci vient du fait que les trois premiers sommets font partis d'un P_4. et que si le dernier sommet u_i formait une arête avec u_{j-2} ou u_{j-1} alors on aurait une corde plus courte que celle supposée minimale : c'est absurde. En effet $|(j-1) - i| < |j - i|$ car $i < j$, idem pour $j - 2$.
De plus on a $u_lu_{l+1}u_{l+2}u_{l+3}$ qui est un P_4, pour $l \in i, .., j-3$. On a donc un cycle plus court que C qui vérifie les hypothèses du lemme, ce qui est en contradiction avec nos hypothèses. Le cycle C est donc bien sans corde.



6.2 algorithme

L'algorithme proposé dans [SN] se base sur le lemmes précédent : on part de P_3 qu'on essaie d'étendre en P_4 dans une suite de P_4 qui suit les hypothèses du Lemme (tout sous ensemble de 4 sommets contigïs du chemin en cours de construction est un P_4).

```
HDAlgorithm(G=(V,E))
pour chaque u dans V:
     pour chaque(v,w) dans E:
          not_in_hole[(v,w),u] := 0;
     in_path[u] := 0;
```

```
6.  pour chaque u de V faire:
7.       in_path[u] := 1;
8.       pour chaque e=(v,w) dans E faire:
9.            si u est adjacent à v et non-adjacent à w et not_in_hole[(u,v),w ] == 0:
10.               in_path[v] := 1;
11.               process(u,v,w);
12.               in_path[v] := 0;
13.      in_path[u] := 1;
14.      retourner : G ne contient pas de cycle sans corde
```

Cet algorithme ce sert d'une procédure process, reportée ci-dessous :

```
1.    process(a,b,c)
2.    in_path[c] := 1;
3.    pour chaque sommet d adjacent à c dans G:
4.        si d n'est ni adjacent à a ni à b dans G:
5.            si in_path[d] == 1:
6.                retourner : G a un cycle sans corde
7.            sinon si not_in_hole[(b,c),d] == 0:
8.                process(b,c,d)
9.    in_path[c] := 0;
10.   not_in_hole[(a,b),c] := 1;
11.   not_in_hole[(c,b),a] := 1;
```

La preuve complète de l'algorithme ainsi qu'une analyse fine de la complexité sont traitées dans [SN]. Nous allons nous intéresser à une compréhension plus informelle de cette méthode.
Pour chaque sommet, on essaie de former un chemin "sans corde" à partir de lui (ligne 6). Plutôt que d'avoir un vrai chemin sans corde, on cherche juste à obtenir un chemin qui vérifie les propriétés du Lemme, ce qui au final revient au même. Par "chemin sans corde", on entend ici : dont les sommets contigüs forment tous des P_4. On peut donc se restreindre à chercher des P_4 pour allonger notre chemin. Bien evidemment, dès que le chemin boucle sur lui même, comme il vérifie alors le lemme, on a un cycle de P_4 donc un cycle sans corde longueur supérieure à 5
On part d'abord de chaque P_3 partant de ce sommet (ligne 8). Dès qu'on a un P_3, on appelle la procédure **process** sur lui. L'appel de cette procédure implique que chaque u,v,w du P_3 est placé à 1 dans in_path. On prolonge ce P_3 en un P_4 dans G (ligne 3 et 4 de process).
A ce moment là, on a un "chemin sans corde", dont tous les sommets ont leur variable in_path à 1. réciproquement si in_path de u est à 1 alors u fait partie du chemin sans corde temporaire. Si l'on prolonge notre P_3 par un P_4 dont la nouvelle extrémité est déjà dans le chemin sans corde temporaire alors on vient de trouver un trou. Ce trou est au moins de longueur 5 puisqu'on a fait attention d'avoir un P_4 a,b,c,d et d non-adjacent ni à a ni à b.
Sinon on a simplement réussi à agrandir notre "chemin sans corde", tout en vérifiant toujours les conditions "sans corde". on a un chemin qui se finit par le P_4 a,b,c,d et on cherche à ajouter un sommet e tel que b,c,d,e soit un P_4. On a donc plus qu'à rappeler **process** sur b,c,d.
Les variables not_in_hole servent à indiquer qu'un sommet ne fait pas partie du trou et in_path

que le sommet est dans le chemin courant (pour détecter un bouclage synonime de trou).

la complexité de l'algorithme précédent est $O(m^2)$.

Remarque 4 (Cas des antitrous) *Les anti-trous qui sont les complémentaires des trous, sont aussi beaucoup étudiés. On pourrait avoir envie pour leur détection, d'appliquer l'algorithme précédent sur le graphe complémentaire. Cette solution a le désavantage d'avoir une complexité dans le pire des cas en $O(n^4)$, Puisque un $O(m^2)$ sur un graphe avec possiblement $O(n^2)$ arêtes. On trouve dans [SN] un algorithme en $O(n + m^2)$ pour ce problème.*

6.3 Le cas des trous impairs

Les trous impairs sont très étudiés à cause de la conjecture sur les graphes parfaits. Etant donnée l'importance des graphes parfaits, il serait intéressant d'avoir un algorithme polynomial performant pour leur reconnaissance. A ce jour[7], l'algorithme le plus rapide connu pour les trous impairs est en $O(n^2)$.

7 Problème du cycle hamiltonien

La recherche d'un cycle hamiltonien fait partie des célèbres problèmes NP-complet. Cette section qui aurait pu être placé en annexe résume simplement ce problème.

7.1 Définition

Définition 3 *Un **cycle hamiltonien** dans un graphe G=(V,E) est un cycle qui passe une et une seule fois par chacun des sommets du graphes. C'est donc un donc un cycle simple maximum .*
*Un graphe est dit **hamiltonien** s'il contient un cycle hamiltonien.*

Le problème version "arête" de la recherche d'un circuit hamiltonien est la recherche d'un cycle eulérien : cycle passant une et une seule fois par chaque arête du graphe.

Théorème 1 (Euler) *Un graphe connexe admet un cycle eulérien si et seulement si tout ses sommets sont de degrés pairs.*

Ce problème peut être résolu en temps **polynomial**, il suffit en effet de vérifier que tous les sommets du graphes sont de degré pair ce qui se fait trivialement en temps O(m) avec des listes d'adjacences ou $O(n^2)$ si l'on dispose d'une matrice d'adjacence.
La preuve basée sur l'algorithme d'euleur (qui construit un cycle eulérien s'il existe) peut être trouvée dans deux nombreux ouvrages [8].
Le problème de cycle Hamiltonien est quand à lui NP-Complet, il existe des réductions à partir de 3-SAT à l'aide des habituels widgets pour forcer la valeur des instances de 3-SAT.

[7] à ma connaissance (non exhaustive)
[8] par exemple http ://www.g-scop.inpg.fr/ rapinec/Graphe/Chemin/algorithme_euler.html

	Graphe non orienté		
	cycle simple	cycle élementaire	cycle sans corde
existence	$O(n)$	$O(n)$	$O(n)$ [*hole*, $O(m^2)$]
minimum	$O(n^2)$,impair $O(n*m)$ ou O($M(n) \times log(n)$)		
pair plus court	$O(n^2)$		
de longueur donnée	C_{2k} : O(k!*n)		
hamiltonien	NP-complet	NP-complet	O(n)
eulérien	polynomial	-	

FIG. 3 – Tableau récapitulatif des complexités

8 Conclusion

Nous n'avons étudié en profondeur que quelques problèmes sur les graphes non-oriéntés. Il en existe beaucoup d'autres et encore plus si l'on ajoute les graphes orientés.
Bien que de nombreux résultats existent déjà, certaines parties restent active (trou impair ...). La recherche de cycle dans les graphes, loin de tourner en rond montre une grande diversité autant dans les méthodes de résolution que dans les résultats. Nous avons ici abordés des problèmes linéaire, polynomiax et NP-complet. Certains sont traités par des parcours à base de liste d'adjacences et d'autres par multiplication matriciele. D'autres encore font appel à des résultats poussés sur les ensembles et leurs intersections (sous-collection représentative).

8.1 Tableau Récapitulatif

Quelques remarques :

- on ne peut avoir de cycle hamiltonien sans corde si le graphe n'est pas un cycle (de type C_n). On a juste a vérifier que chaque sommet et de dégré deux et qu'en partant d'un sommet, le seul chemin possible revient à ce sommet en ayant parcouru tous les autres sommets
- décider l'existence d'un cycle sans corde de $longueur \geq 5$ (*it* hole) se fait en $O(m^2)$, il existe aussi un algorithme pour exhiber le cycle impair en $O(n+m^2)$

9 Bibliographie

Références

[DC87] S. Winograd D. Coppersmith. Matrix multiplication via arithmetic progressions. *Proceedings of the Nineteenth Annual ACM Symposium on Theory of Computing*, pages 1 – 6, 1987.

[Ita77] A. Itai. Finding a minimum circuit in a graph. 1977.

[JBj01] G. Gutin J. Bang-jensen. *Digraphs : Theory, Algorithms and Applications*. 2001.

[NA97] U. Zwick N. Alon, R. Yuster. Finding and counting given length cycles. *Algorithmica 17*, pages 209 – 223, 1997.

[SN] L. Palios S.D. Nikolopoulos. Hole and antihole detection in graphs.