

Optimisation et Complexité

Rapport de projet

CHEUCLE Pierre – ODIER Valentin – WAKIM Marie

09/05/2012

Contenu

Structure.....	3
Algorithme de Bloch	4
Mise en place.....	4
Algorithme de Ford-Fulkerson.....	6
Mise en place.....	6
Affichage des résultats.....	8
Fonctions pour l'algorithme de Bloch	11
Fonctions pour l'algorithme de Ford-Fulkerson	14
Exemple de résultats.....	18
Graphe TD :.....	18
Graphe du TAI :	20
Graphe 3 :	21

Introduction

Ce projet nous a amenés à réaliser l'algorithme de Bloch ainsi que l'algorithme de Ford-Fulkerson. Pour parvenir à cela nous avons d'abord réalisé l'algorithme de Bloch, puis du résultat de cet algorithme nous avons implémenté celui de Ford- Fulkerson.

Dans une première partie nous expliquerons la structure de données que nous avons choisie, puis comment nous avons mis en œuvre l'algorithme de Bloch, avant d'expliquer comment nous avons fait pour celui de Ford-Fulkerson.

En outre en annexes, nous avons expliqué chacune des fonctions utilisées pour les deux algorithmes et ajouté des captures d'écran de nos résultats.

Structure

Afin de réaliser le travail demandé nous avons créé deux structures de données :

Une nommée « arc », et une nommée « sommet » :

arc	sommet
• string villeDep • string villeDest • int capa • int flot • int capacite • bool sature • bool bloque • bool null	• string name • string tag • bool signe • unsigned rang

Pour la structure arc, nous avons besoin :

- Du point de départ (**villeDep**) et point d'arrivée (**villeDest**) de l'arc. Ces deux variables sont matérialisées par une chaîne de caractères (**String**) ;
- De la capacité (**capa**), du flot (**flot**), et de la capacité résiduelle (**capacite**) de l'arc. Ces variables sont des entiers positifs, nous avons donc utilisé un type **int** (nous aurions pu utiliser un **unsigned**);
- De l'« état » de l'arc, est-il praticable ou non ? Pour cela nous différencions trois cas : l'arc est saturé (la variable **sature** est mis à **true**), l'arc est bloqué (**bloque** est mis à **true**), ou l'arc est praticable (**null** à **true**).

Nous avons estimé que pour définir un sommet nous avons besoin de connaître :

- Son nom (**name**), le nom du sommet qui le marque (**tag**). Comme **villeDep** et **villeDest**, ces deux variables prendront en compte des chaînes de caractères : nous avons donc choisi un type String;
- Le signe du marquage (**signe**) : **true** pour positif, et **false** pour négatif ;
- Le rang de ce sommet (**rang**) : de type **unsigned** car jamais négatif.

Algorithme de Bloch

Mise en place

L'algorithme de Bloch permet d'obtenir un flot complet.

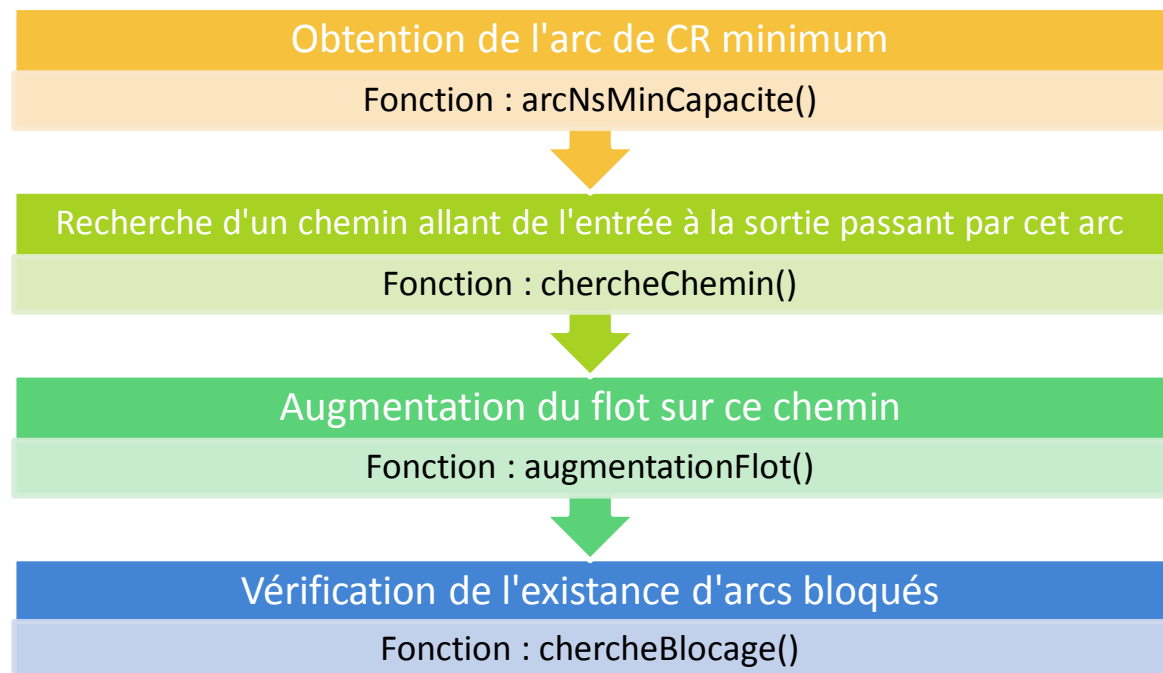
Afin d'expliquer comment nous avons implémenté l'algorithme de Bloch, nous allons expliquer et différencier les différentes parties de l'algorithme. Il s'effectue en quatre étapes :

Algorithme :

- 1) dresser la liste des arcs dans l'ordre de la numérotation avec leur capacité.
- 2) choisir dans la liste des arcs praticables (non saturés, non bloqués) le premier arc de plus petite capacité résiduelle ρ_0 .
- 3) déterminer un chemin praticable de E à S , passant par l'arc précédent, en prenant les sommets dans l'ordre de la numérotation.
 - si le chemin est élémentaire (sans répétition de sommets), on augmente le flot des arcs de ce chemin de ρ_0 et on passe à 4).
 - si le chemin n'est pas élémentaire (il existe un circuit), on bloque les arcs de capacité la plus faible du circuit mis en évidence et retour à 2).
- 4) pour chaque sommet du chemin choisi, vérifier s'il existe des arcs à bloquer, les bloquer et retour à 2).

1. La condition d'arrêt de l'algorithme de Bloch est la suivante : il n'existe plus de chemin praticable, c'est-à-dire qu'il n'y a pas de chemin allant de l'entrée à la sortie du réseau de transport. Cette condition d'arrêt a été implémentée par une boucle **while**, où nous nous assurons que la fonction **estPraticable()** retourne **vrai**.
2. Puis dans cette boucle **while**, une fonction nous retourne l'arc non saturé possédant la plus petite capacité résiduelle : **arcNsMinCapacite()**. Pour réaliser cette opération nous avons besoin de connaître la capacité maximale. En effet, nous savons que les capacités sont au minimum égale à 0, mais le minimum d'un réseau peut très bien être : 1, 2, 3... Nous ne pouvons donc pas comparer nos valeurs avec 0, nous avons choisi de les comparer à la capacité résiduelle maximale : de partir du maximum pour descendre au minimum.
Nous récupérons l'arc non saturé de capacité résiduelle maximale (par la fonction **arcNsMaxCapacite()**), puis la fonction **arcNsMinCapacite()** nous retourne l'arc non saturé de capacité résiduelle minimale.
3. L'étape 3) concerne l'augmentation des flots. La fonction **chercheChemin()** nous permet de trouver un chemin élémentaire allant de l'entrée à la sortie passant par l'arc de capacité résiduelle minimale que nous avons obtenu précédemment. Une fois le chemin praticable trouvé, nous augmentons le flot sur chacun de ses arcs grâce à la fonction **augmentationFlot()**.
4. Puis nous vérifions si l'augmentation des flots a entraîné des blocages. La fonction **chercheblocage()** permet de modifier l'état des arcs du graphe susceptible d'être bloqués par la dernière augmentation de flot. Pour chaque arc du réseau non bloqué ou non saturé, nous contrôlons s'il est exploitable par au moins un chemin partant de l'entrée jusqu'à la sortie. Si ce n'est pas le cas, alors nous le bloquons.

Finalement nous avons le processus suivant qui se répète tant qu'il existe un chemin praticable :



De plus tout au long de l'algorithme, nous avons placé des fonctions d'affichage qui nous permettent de vérifier nos résultats.

Algorithme de Ford-Fulkerson

Mise en place

L'algorithme de Ford-Fulkerson permet, à partir d'un flot complet, d'obtenir un flot maximum. Nous sommes donc partis du résultat de l'algorithme de Bloch pour réaliser cet algorithme.

Sa structure est la suivante :

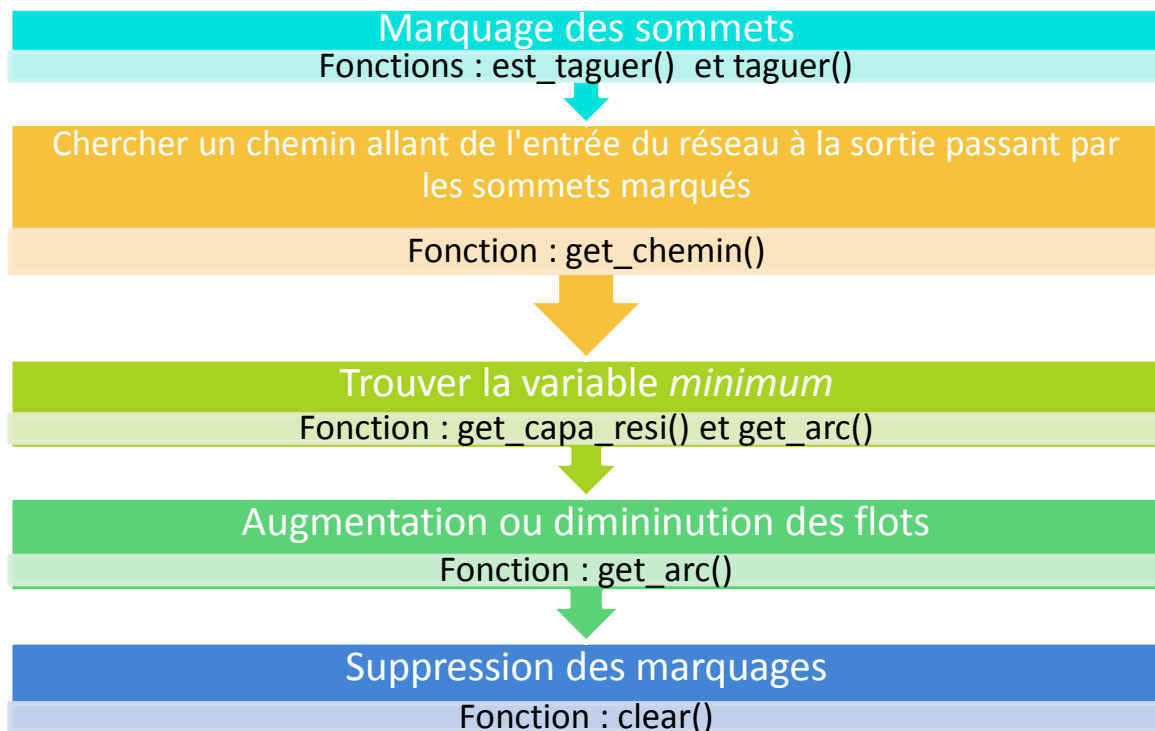
Algorithme :

- 1) Marquer l'entrée E du réseau (+).
- 2) Marquer l'extrémité terminale d'un arc (i, j) non saturé dont l'extrémité initiale est marquée (mettre $+i$ à côté de j) : on pourra augmenter le flot sur cet arc.
- 3) Marquer l'extrémité initiale d'un arc (k, l) transportant un flot non nul et dont l'extrémité finale est marquée (mettre $-l$ à côté de k) : on pourra diminuer le flot sur cet arc.
- 4) Deux possibilités :
 - si on n'arrive pas à marquer la sortie S , alors le flot est maximal.
 - si on arrive à marquer la sortie :
 - * prendre une chaîne de sommets marqués de E à S ;
 - * calculer $d_1 = \min [C(i, j) - \varphi(i, j)]$
 arcs (i, j) parcourus dans le sens des flèches
 (on peut augmenter le flot sur ces arcs d'une quantité d_1) ;
 - * calculer $d_2 = \min [\varphi(i, j)]$
 arcs (i, j) parcourus en sens opposé des flèches
 (on peut diminuer le flot sur ces arcs d'une quantité d_2) ;
 - * augmenter de $d = \min (d_1, d_2)$ le flot des arcs parcourus dans le sens des flèches ;
 - * diminuer de $d = \min (d_1, d_2)$ le flot des arcs parcourus dans le sens opposé des flèches ;
 - * recommencer à 1) jusqu'à ce que la sortie ne puisse plus être marquée.

1. Nous commençons par taguer l'entrée du réseau par un « + ». Une fonction nommée **taguer()** prend en paramètre : le sommet à taguer, le signe de ce dernier et d'où vient le tag. Pour marquer l'entrée nous avons choisi comme paramètre :
 - a. 1^{er} paramètre : l'entrée du réseau, que nous récupérerons parmi nos sommets.
 - b. 2nd paramètre : 1, nous taguons l'entrée par un « + », la variable **signe** prendra donc la valeur **1**.
 - c. 3^{eme} paramètre : entrée+1234, l'entrée étant taguée toute seule, nous avons choisi au hasard : « entrée+1234 », ceci n'ayant pas d'importance pour la suite.
2. Puis notre seconde étape consiste à taguer tous les autres sommets. Pour l'effectuer, une boucle vérifie pour chaque sommet s'il peut être tagué (c'est-à-dire si les conditions pour marquer un arc par un « + » ou un « - » sont vérifiées). Si c'est le cas nous marquons l'arc à l'aide de notre fonction **taguer()**.
 Nous sortons de notre boucle quand tous les arcs ont été vérifiés ou quand la sortie a été marquée.

3. Ensuite nous recherchons un chemin allant de l'entrée à la sortie passant par des sommets marqués. Nous vérifions que le chemin existe, si ce n'est pas le cas c'est que la sortie n'a pas pu être marquée : le flot est maximal, l'algorithme s'arrête. S'il existe, nous allons chercher le minimum (variable notée *minimum*) entre les flots et la capacité résiduelle. Pour chaque sommet nous regardons si :
 - a. Il est marqué par un « + », alors la variable *minimum* est comparée à la capacité résiduelle de chaque sommet. Pour chaque sommet nous regardons si sa capacité résiduelle est inférieure ou non, si oui alors notre *minimum* prend cette valeur.
 - b. Il est marqué par un « - », nous comparons *minimum* au flot de chaque sommet.
4. Puis, de nouveau nous distinguons les sommets marqués par un « + », de ceux marqués par un « - ». Nous augmentons le flot de *minimum* pour les « + », et nous diminuons le flot de *minimum* pour les « - ».
5. Enfin nous supprimons tous les marquages réalisés, et nous revenons à l'étape 2.

Après avoir marqué l'entrée, nous avons le processus suivant qui se répète tant que la sortie a pu être marquée:



Affichage des résultats

Pour afficher les résultats, nous utilisons la boîte de commande :

Bloch :

```

AFFICHAGE DES ARCS DU RESEAU
E-----[5]-0---->a
E-----[10]-0---->b
E-----[8]-0---->c
a-----[7]-0---->d
a-----[10]-0---->e
b-----[1]-0---->c
b-----[8]-0---->d
b-----[2]-0---->e
c-----[2]-0---->e
c-----[4]-0---->f
d-----[7]-0---->g
e-----[2]-0---->f
e-----[4]-0---->g
e-----[6]-0---->S
f-----[6]-0---->S
g-----[10]-0---->S

APRES AUGMENTATION DES FLOTS VOICI L'ETAT DU RESEAU
E-----[0]-5---->a sature
E-----[4]-6---->b bloque
E-----[3]-5---->c bloque
a-----[4]-3---->d bloque
a-----[8]-2---->e bloque
b-----[0]-1---->c sature
b-----[5]-3---->d bloque
b-----[0]-2---->e sature
c-----[0]-2---->e sature
c-----[0]-4---->f sature
d-----[1]-6---->g bloque
e-----[0]-2---->f sature
e-----[0]-4---->g sature
e-----[6]-0---->S bloque
f-----[0]-6---->S sature
g-----[0]-10---->S sature

```

Ford-Fulkerson :

TRACE D'EXECUTION							
Ea	5	5	5	3	0	0	0
Eb	9	9	7	7	7	4	4
Ec	8	7	7	7	7	7	3
ad	7	7	7	7	4	4	4
ae	10	10	10	8	8	8	8
bc	0	0	0	0	0	0	0
bd	8	8	8	8	8	5	5
be	2	2	0	0	0	0	0
ce	1	0	0	0	0	0	0
cf	4	4	4	4	4	4	0
dg	7	7	7	7	4	1	1
ef	1	0	0	0	0	0	0
eg	4	4	2	0	0	0	0
eS	6	6	6	6	6	6	6
fS	5	4	4	4	4	4	0
gS	10	10	8	6	3	0	0

```

E-----[0]-5---->a sature
E-----[4]-10---->b bloque
E-----[3]-5---->c bloque
a-----[4]-0---->d bloque
a-----[8]-5---->e bloque
b-----[0]-1---->c sature
b-----[5]-7---->d bloque
b-----[0]-2---->e sature
c-----[0]-2---->e sature
c-----[0]-4---->f sature
d-----[1]-7---->g bloque
e-----[0]-2---->f sature
e-----[0]-3---->g sature
e-----[6]-4---->S bloque
f-----[0]-6---->S sature
g-----[0]-10---->S sature

```

Néanmoins cet affichage reste peu lisible, c'est pourquoi nous avons eu l'idée de concevoir une fonction permettant de faire un affichage en html. Ainsi l'affichage de nos résultats est le suivant (cf. page suivante).

Cet affichage est plus clair et plus facile à utiliser. Implémenter de l'html dans notre code nous a permis d'avoir un résultat visuellement agréable (cf. annexes pour d'autres exemples).

Algorithme de Bloch

Noir : normal. Rouge : saturé. Bleu : bloqué.

Arcs

Ea	[5]0	[5]0	[5]0	[5]2	[5]5	[5]5	[5]5
Eb	[10]1	[10]1	[10]3	[10]3	[10]3	[10]6	[10]6
Ec	[8]0	[8]1	[8]1	[8]1	[8]1	[8]1	[8]5
ad	[7]0	[7]0	[7]0	[7]0	[7]3	[7]3	[7]3
ae	[10]0	[10]0	[10]0	[10]2	[10]2	[10]2	[10]2
bc	[1]1	[1]1	[1]1	[1]1	[1]1	[1]1	[1]1
bd	[8]0	[8]0	[8]0	[8]0	[8]0	[8]3	[8]3
be	[2]0	[2]0	[2]2	[2]2	[2]2	[2]2	[2]2
ce	[2]1	[2]2	[2]2	[2]2	[2]2	[2]2	[2]2
cf	[4]0	[4]0	[4]0	[4]0	[4]0	[4]0	[4]4
dg	[7]0	[7]0	[7]0	[7]0	[7]3	[7]6	[7]6
ef	[2]1	[2]2	[2]2	[2]2	[2]2	[2]2	[2]2
eg	[4]0	[4]0	[4]2	[4]4	[4]4	[4]4	[4]4
eS	[6]0	[6]0	[6]0	[6]0	[6]0	[6]0	[6]0
fS	[6]1	[6]2	[6]2	[6]2	[6]2	[6]2	[6]6
gS	[10]0	[10]0	[10]2	[10]4	[10]7	[10]10	[10]10

Chemins analysés

E - b - c - e - f - S
E - c - e - f - S
E - b - e - g - S
E - a - e - g - S
E - a - d - g - S
E - b - d - g - S
E - c - f - S

Algorithme de Ford Fulkerson

Diminution de la CR en bleu. Augmentation en rouge.

Arcs

Ea	[5]5	[5]5	[5]5
Eb	[10]6	[10]7	[10]10
Ec	[8]5	[8]5	[8]5
ad	[7]3	[7]3	[7]0
ae	[10]2	[10]2	[10]5
bc	[1]1	[1]1	[1]1
bd	[8]3	[8]4	[8]7
be	[2]2	[2]2	[2]2
ce	[2]2	[2]2	[2]2
cf	[4]4	[4]4	[4]4
dg	[7]6	[7]7	[7]7
ef	[2]2	[2]2	[2]2
eg	[4]4	[4]3	[4]3
eS	[6]0	[6]1	[6]4
fS	[6]6	[6]6	[6]6
gS	[10]10	[10]10	[10]10

Conclusion

Ce projet nous a permis de mettre en place les algorithmes de Bloch et de Ford-Fulkerson.

Nous avons réussi à réaliser ces deux algorithmes et pris en compte des cas particuliers :

- *la présence de circuits dans un réseau*
- *si un sommet a un nom avec plus d'un caractère, tel que : « Eab »*
- *si le graphe n'est pas connexe.*

Ces deux algorithmes sont de bonnes méthodes pour obtenir un flot complet pour l'une et maximum pour l'autre.

Annexes

Fonctions pour l'algorithme de Bloch

Voici un descriptif des fonctions employées pour cet algorithme dans l'ordre d'utilisation:

Nom :	<code>lirefichierArc(vector<arc> &lesArcs, string &entree, string &sortie)</code>
But :	Lire notre fichier .txt
Entrée :	L'arc qui va être rempli, deux string représentant le sommet initial et final
Sortie :	Int, en l'occurrence 0

Fonction de lecture qui charge le graphe en mémoire

Nom :	<code>afficheArc(arc arc)</code>
But :	Afficher un arc
Entrée :	L'arc que nous voulons afficher
Sortie :	void

Nous affichons dans cet ordre : le sommet initial, la capacité de l'arc, son flot, le sommet final. Puis nous vérifions s'il est saturé et/ou bloqué, nous le signalons si c'est le cas.

Nom :	<code>afficheLesArcs(vector<arc> lesArcs)</code>
But :	Afficher un vector d'arc
Entrée :	Le vector que nous voulons afficher
Sortie :	void

Le vector « *lesArcs* » contient tous les arcs de notre graphe, afin de les afficher nous parcourons les cases de notre vector en utilisant à chaque itération la fonction *afficheArc()*.

Nom :	<code>bloch(vector<arc> &lesArcs, string entree, string sortie)</code>
But :	Exécute l'algorithme de bloch : trouve un flot complet
Entrée :	Le graphe, l'entrée, la sortie
Sortie :	Int, renvoie 0 s'il n'y a pas d'erreur

Réalise l'algorithme de Bloch comme expliqué dans la partie « Mise en place de l'algorithme ».

Nom :	estPraticable(vector<arc> lesArcs)
But :	Permet de tester si le réseau est praticable
Entrée :	Notre tableau d'arc à tester
Sortie :	Bool

Pour chaque arc du réseau, une boucle while vérifie si les variables *sature* et *bloque* sont à *false* si c'est le cas l'arc est praticable. S'il reste encore des arcs non bloqués ou non saturés, alors il existe encore des chemins à explorer.

Nom :	arcNsMaxCapacite(vector<arc> lesArcs)	arcNsMinCapacite(idem)
But :	Retourne l'arc non saturé ayant la plus grande capacité résiduelle	ayant la plus petite capacité résiduelle
Entrée :	Notre vector à étudier	Notre vector à étudier
Sortie :	arc, retourne l'arc recherché	arc, retourne l'arc recherché

Après avoir vérifié que le réseau était praticable, nous devons chercher l'arc de plus petite capacité. Pour cela :

- Nous récupérons l'arc de capacité la plus élevée par la fonction *arcNsMaxCapacite()*, que nous définissons comme arc de capacité maximum
- Puis pour chaque arc nous comparons sa capacité avec celui de l'arc trouvé. Si la capacité est inférieure nous mettons à jour l'arc minimum, sinon nous testons l'arc suivant. Ainsi nous sommes sûrs de tester tous les arcs et d'obtenir le minimum.

Nom :	augmentationFlot(vector<arc> chemin, vector<arc> &lesArcs, int augmentation)
But :	Augmente le flot des arcs du chemin choisi lors de l'exécution de l'algorithme de Bloch
Entrée :	Le chemin praticable, le graphe, la capacité du chemin minimale que nous allons ajouter ou soustraire
Sortie :	void

Cette fonction permet d'augmenter le flot sur le chemin choisi, le « chemin praticable » est trouvé grâce à la fonction *chercheChemin()* expliquée ci-dessous. Si après l'augmentation du flot, la capacité résiduelle d'un arc est égale à 0, alors la fonction le sature.

Nom :	chercheChemin(arc arcCapaciteMin, vector<arc> lesArcs, string entree, string sortie)
But :	Chercher le chemin allant de l'entrée à la sortie passant par l'arc de capacité minimale
Entrée :	L'arc de capacité minimale, notre graphe, l'entrée du graphe, la sortie du graphe
Sortie :	vector <arc> : c'est-à-dire le chemin voulu

La construction du chemin est réalisée en deux étapes : d'abord, nous cherchons le chemin praticable allant de l'arc à l'entrée, puis celui allant de l'arc jusqu'à la sortie. Si la fonction détecte l'existence d'un chemin non élémentaire, nous bloquons l'arc de capacité minimale du circuit, puis nous recommençons la recherche.

Nom :	arcNsPrecedent(arc unArc, vector<arc> lesArcs)	arcNsSuivant(arc unArc, vector<arc> lesArcs)
But :	Retourner l'arc non saturé précédent celui passé en paramètre.	Retourner l'arc non saturé suivant celui passé en paramètre.
Entrée :	L'arc à prendre en compte, et notre vector d'arc (c'est-à-dire le graphe).	
Sortie :	arc	

Ces deux fonctions sont utilisées dans la fonction *chercheChemin()* et permettent d'obtenir les arcs précédents (arc -> entrée) et suivants (arc -> sortie), afin de constituer le chemin allant de l'entrée à la sortie.

Nom :	void chercheBlocage(vector<arc> &lesArcs, string entree, string sortie)
But :	Chercher des arcs bloqués
Entrée :	Notre vector d'arc comportant l'ensemble des arcs, l'entrée, la sortie
Sortie :	Ne retourne rien

Pour chaque arc nous vérifions s'il est bloqué, par les arcs le précédant ou par les arcs le succédant, si c'est le cas nous plaçons la variable « bloque » à *true*.

Fonctions pour l'algorithme de Ford-Fulkerson

Nom :	<code>lecture_sommets(vector<arc> &lesArcs, vector<sommet> &lesSommets)</code>
But :	Créer notre tableau de sommets
Entrée :	L'ensemble des arcs de notre réseau, le futur ensemble de sommets
Sortie :	Void

Cette fonction prend en paramètre nos arcs (*lesArcs*), ainsi qu'un vector (*lesSommets*). Au départ, ce dernier est vide : cette fonction a pour but de le remplir.

1. Pour chaque arc, nous regardons si la valeur de la variable *villeDep* (représentant le nom d'un sommet) se trouve dans le vector *lesSommets*. Si ce n'est pas le cas nous ajoutons un sommet à notre vector en précisant son nom et son rang.
2. Puis nous regardons si la valeur de la variable *villeDest* (représentant le nom d'un sommet) se trouve dans le vector *lesSommets*. Si ce n'est pas le cas nous ajoutons un sommet à notre vector en précisant son nom et son rang.

Nom :	<code>estdans(vector<sommet> &lesSommets, string &name)</code>
But :	Vérifier si un sommet existe déjà dans notre liste de sommets
Entrée :	L'ensemble de nos sommets, le nom du sommet à vérifier
Sortie :	Bool true ou false

Il s'agit de la fonction utilisée dans la précédente (*lecture_sommets()*), afin d'être sûr de ne pas ajouter de doublons. Un sommet est identifié par son nom, nous comparons donc notre variable *name* au nom des différents sommets. Si le sommet existe alors la fonction retournera *true*.

Nom :	<code>afficher_sommet(vector<sommet> &lesSommets);</code>
But :	Affichage des sommets
Entrée :	L'ensemble de nos sommets
Sortie :	Void

Simple affichage des sommets, nous affichons pour chaque sommet : son nom, son rang, et le nom du sommet qui le marque.

Nom :	<code>Q2(vector<arc> &lesArcs, vector<sommet> lesSommets, string entree, string sortie);</code>
But :	Effectuer l'algorithme de Ford-Fulkerson
Entrée :	L'ensemble de nos arcs, l'ensemble de nos sommets, l'entrée du réseau, la sortie du réseau
Sortie :	Void

Réalise l'algorithme de Ford-Fulkerson comme expliqué dans la partie « Mise en place de l'algorithme ».

Nom :	<code>get_chemin(string entree, string sortie, vector<sommet> &lesSommets, vector<arc> &lesArcs)</code>
But :	Obtenir un chemin allant de l'entrée à la sortie passant par des sommets marqués
Entrée :	L'entrée du réseau, la sortie du réseau, l'ensemble de nos sommets marqués, l'ensemble de nos arcs
Sortie :	<code>vector<sommet></code> , correspondant aux différents sommets du chemin

Trouve un chemin pour aller de l'entrée à la sortie en ne passant que par des sommets que nous avons marqués.

Nom :	<code>est_dans(vector<sommet> &a, vector< vector<sommet> > &tab)</code>
But :	Chercher si un sommet appartient à un ensemble de sommets
Entrée :	Le sommet à tester, notre ensemble de sommets
Sortie :	Bool, retourne <i>true</i> si vrai

Vérifie si l'élément *a* se situe bien dans l'élément *tab*. Pour ce faire nous parcourons le vector *tab* à la recherche du nom du sommet *a*.

Nom :	<code>get_sommet(string name, vector<sommet> &lesSommets)</code>
But :	Retourner un sommet
Entrée :	Le nom recherché, notre ensemble de sommets
Sortie :	<code>sommet&</code> , le sommet

Nous parcourons le vector *lesSommets* à la recherche du sommet de nom *name*. Une fois ce sommet trouvé nous le retournons.

Nom :	<code>est_tague(sommet &s)</code>
But :	Vérifier si un sommet est tagué
Entrée :	Le sommet à vérifier
Sortie :	Bool, true si sommet marqué

La variable *tag*, qui correspond au nom du sommet qui marque, est comparée à une chaîne de caractères vide. Si la taille de *tag* est la même que la chaîne vide, alors le sommet n'est pas marqué.

Nom :	<code>taguer(sommet &s, bool signe, string from)</code>
But :	Permettre de taguer un sommet
Entrée :	Le sommet à marquer, le signe du marquage, le nom du tagueur
Sortie :	Void

Nous remplaçons la valeur de la variable *signe* du sommet, et celle de la variable *tag* du sommet, par celles entrées en paramètre.

Nom :	<code>est_sature(arc &a)</code>	<code>bool est_vide(arc &a)</code>
But :	Vérifier si l'arc est saturé	Vérifier si le flot de l'arc est nul
Entrée :	L'arc à vérifier	L'arc à vérifier
Sortie :	Bool, saturé ou non	Bool, vide ou non

Pour la fonction *est_sature()*, nous regardons si la capacité est égale au flot.

Pour la seconde, nous regardons si le flot est égal à zéro.

Nom :	<code>set_rang_sommet(unsigned rang, string name, vector<sommet> &lesSommets)</code>
But :	Assigner un rang à un sommet
Entrée :	La valeur du rang, le nom de notre sommet, notre ensemble de sommets
Sortie :	void

Nous parcourons la liste des sommets à la recherche du sommet du nom *name*. Une fois ce sommet trouvé : son attribut *rang* prend la valeur de la variable *rang* passée en paramètre de la fonction.

Nom :	<code>get_rang_sommet(string name, vector<sommet> &lesSommets)</code>
But :	Permettre de connaître le rang d'un sommet
Entrée :	Le nom du sommet dont nous cherchons le rang, l'ensemble de nos sommets
Sortie :	Unsigned, le rang du sommet

Retourne le rang du sommet de nom *name*.

Nom :	<code>set_rang_sommets(vector<arc> &lesArcs, vector<sommet> &lesSommets, string &entree)</code>
But :	Attribuer les rangs des sommets
Entrée :	L'ensemble de nos arcs, l'ensemble de nos sommets, l'entrée du réseau
Sortie :	Void

Cette fonction nous permet de calculer les rangs des sommets de notre réseau.

Nom :	<code>get_arc(string villeDep, string villeDest, vector<arc> &lesArcs)</code>
But :	Obtenir un arc à partir du nom de départ, et du nom de destination
Entrée :	Le nom de départ, le nom de destination, l'ensemble de nos arcs
Sortie :	<code>arc&</code>

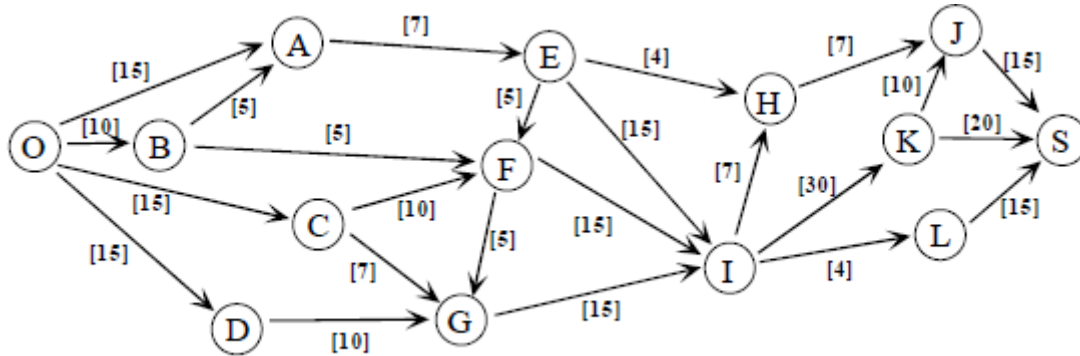
Pour chaque arc nous vérifions deux choses :

- si la variable *villeDep* (passée en paramètre) correspond à une des valeurs parmi notre ensemble d'arcs.
- si la variable *villeDest* (passée en paramètre) correspond à une des valeurs parmi notre ensemble d'arcs.

Si ces deux conditions sont vérifiées en même temps, c'est-à-dire pour le même arc, nous avons trouvé l'arc que nous cherchions : nous le retournons.

Exemple de résultats

Graphe TD :



Bloch :

Arcs

OA	[15]4	[15]7	[15]7	[15]7	[15]7	[15]7	[15]7	[15]7	[15]7
OB	[10]0	[10]0	[10]2	[10]5	[10]5	[10]5	[10]5	[10]5	[10]5
OC	[15]0	[15]0	[15]0	[15]0	[15]3	[15]7	[15]10	[15]15	[15]15
OD	[15]0	[15]0	[15]0	[15]0	[15]0	[15]0	[15]0	[15]0	[15]5
AE	[7]4	[7]7	[7]7	[7]7	[7]7	[7]7	[7]7	[7]7	[7]7
BA	[5]0	[5]0	[5]0	[5]0	[5]0	[5]0	[5]0	[5]0	[5]0
BF	[5]0	[5]0	[5]2	[5]5	[5]5	[5]5	[5]5	[5]5	[5]5
CF	[10]0	[10]0	[10]0	[10]0	[10]3	[10]7	[10]10	[10]10	[10]10
CG	[7]0	[7]0	[7]0	[7]0	[7]0	[7]0	[7]0	[7]5	[7]5
DG	[10]0	[10]0	[10]0	[10]0	[10]0	[10]0	[10]0	[10]0	[10]5
EF	[5]0	[5]3	[5]3	[5]3	[5]3	[5]3	[5]3	[5]3	[5]3
EH	[4]4	[4]4	[4]4	[4]4	[4]4	[4]4	[4]4	[4]4	[4]4
EI	[15]0	[15]0	[15]0	[15]0	[15]0	[15]0	[15]0	[15]0	[15]0
FG	[5]0	[5]3	[5]5	[5]5	[5]5	[5]5	[5]5	[5]5	[5]5
FI	[15]0	[15]0	[15]0	[15]3	[15]6	[15]10	[15]13	[15]13	[15]13
GI	[15]0	[15]3	[15]5	[15]5	[15]5	[15]5	[15]5	[15]10	[15]15
HJ	[7]4	[7]7	[7]7	[7]7	[7]7	[7]7	[7]7	[7]7	[7]7
IH	[7]0	[7]3	[7]3	[7]3	[7]3	[7]3	[7]3	[7]3	[7]3
IK	[30]0	[30]0	[30]2	[30]5	[30]8	[30]8	[30]11	[30]16	[30]21
IL	[4]0	[4]0	[4]0	[4]0	[4]0	[4]4	[4]4	[4]4	[4]4
JS	[15]4	[15]7	[15]9	[15]12	[15]15	[15]15	[15]15	[15]15	[15]15
KJ	[10]0	[10]0	[10]2	[10]5	[10]8	[10]8	[10]8	[10]8	[10]8
KS	[20]0	[20]0	[20]0	[20]0	[20]0	[20]0	[20]3	[20]8	[20]13
LS	[15]0	[15]0	[15]0	[15]0	[15]0	[15]4	[15]4	[15]4	[15]4

Chemins analysés

0 - A - E - H - J - S
0 - A - E - F - G - I - H - J - S
0 - B - F - G - I - K - J - S
0 - B - F - I - K - J - S
0 - C - F - I - K - J - S
0 - C - F - I - L - S
0 - C - F - I - K - S
0 - C - G - I - K - S
0 - D - G - I - K - S

Ford :

Arcs

0A	[15]7	[15]7	[15]7
0B	[10]5	[10]5	[10]5
0C	[15]15	[15]15	[15]15
0D	[15]5	[15]7	[15]10
AE	[7]7	[7]7	[7]7
BA	[5]0	[5]0	[5]0
BF	[5]5	[5]5	[5]5
CF	[10]10	[10]10	[10]10
CG	[7]5	[7]5	[7]5
DG	[10]5	[10]7	[10]10
EF	[5]3	[5]3	[5]0
EH	[4]4	[4]4	[4]4
EI	[15]0	[15]0	[15]3
FG	[5]5	[5]3	[5]0
FI	[15]13	[15]15	[15]15
GI	[15]15	[15]15	[15]15
HJ	[7]7	[7]7	[7]7
IH	[7]3	[7]3	[7]3
IK	[30]21	[30]23	[30]26
IL	[4]4	[4]4	[4]4
JS	[15]15	[15]15	[15]15
KJ	[10]8	[10]8	[10]8
KS	[20]13	[20]15	[20]18
LS	[15]4	[15]4	[15]4

Graphe du TAI :

Algorithme de Bloch

Noir : normal. Rouge : saturé. Bleu : bloqué.

Arcs

Ea	[5]0	[5]0	[5]0	[5]2	[5]5	[5]5	[5]5
Eb	[10]1	[10]1	[10]3	[10]3	[10]3	[10]6	[10]6
Ec	[8]0	[8]1	[8]1	[8]1	[8]1	[8]1	[8]5
ad	[7]0	[7]0	[7]0	[7]0	[7]3	[7]3	[7]3
ae	[10]0	[10]0	[10]0	[10]2	[10]2	[10]2	[10]2
bc	[1]1	[1]1	[1]1	[1]1	[1]1	[1]1	[1]1
bd	[8]0	[8]0	[8]0	[8]0	[8]0	[8]3	[8]3
be	[2]0	[2]0	[2]2	[2]2	[2]2	[2]2	[2]2
ce	[2]1	[2]2	[2]2	[2]2	[2]2	[2]2	[2]2
cf	[4]0	[4]0	[4]0	[4]0	[4]0	[4]0	[4]4
dg	[7]0	[7]0	[7]0	[7]0	[7]3	[7]6	[7]6
ef	[2]1	[2]2	[2]2	[2]2	[2]2	[2]2	[2]2
eg	[4]0	[4]0	[4]2	[4]4	[4]4	[4]4	[4]4
eS	[6]0	[6]0	[6]0	[6]0	[6]0	[6]0	[6]0
fS	[6]1	[6]2	[6]2	[6]2	[6]2	[6]2	[6]6
gS	[10]0	[10]0	[10]2	[10]4	[10]7	[10]10	[10]10

Chemins analysés

E - b - c - e - f - S
E - c - e - f - S
E - b - e - g - S
E - a - e - g - S
E - a - d - g - S
E - b - d - g - S
E - c - f - S

Algorithme de Ford Fulkerson

Diminution de la CR en bleu. Augmentation en rouge.

Arcs

Ea	[5]5	[5]5	[5]5
Eb	[10]6	[10]7	[10]10
Ec	[8]5	[8]5	[8]5
ad	[7]3	[7]3	[7]0
ae	[10]2	[10]2	[10]5
bc	[1]1	[1]1	[1]1
bd	[8]3	[8]4	[8]7
be	[2]2	[2]2	[2]2
ce	[2]2	[2]2	[2]2
cf	[4]4	[4]4	[4]4
dg	[7]6	[7]7	[7]7
ef	[2]2	[2]2	[2]2
eg	[4]4	[4]3	[4]3
eS	[6]0	[6]1	[6]4
fS	[6]6	[6]6	[6]6
gS	[10]10	[10]10	[10]10

Graphe 3 :

A	I	
A	B	5
A	C	8
A	D	9
B	C	7
B	E	4
C	F	10
C	H	5
D	C	1
D	G	5
D	H	4
E	I	5
F	H	5
F	I	5
G	I	6
H	G	4

Algorithme de Bloch

Noir : normal. Rouge : saturé. Bleu : bloqué.

Arcs

AB	[5]0	[5]0	[5]0	[5]4	[5]5	[5]5
AC	[8]0	[8]3	[8]3	[8]3	[8]3	[8]7
AD	[9]1	[9]1	[9]3	[9]3	[9]3	[9]3
BC	[7]0	[7]0	[7]0	[7]0	[7]1	[7]1
BE	[4]0	[4]0	[4]0	[4]4	[4]4	[4]4
CF	[10]1	[10]1	[10]1	[10]1	[10]2	[10]6
CH	[5]0	[5]3	[5]3	[5]3	[5]3	[5]3
DC	[1]1	[1]1	[1]1	[1]1	[1]1	[1]1
DG	[5]0	[5]0	[5]2	[5]2	[5]2	[5]2
DH	[4]0	[4]0	[4]0	[4]0	[4]0	[4]0
EI	[5]0	[5]0	[5]0	[5]4	[5]4	[5]4
FH	[5]1	[5]1	[5]1	[5]1	[5]1	[5]1
FI	[5]0	[5]0	[5]0	[5]0	[5]1	[5]5
GI	[6]1	[6]4	[6]6	[6]6	[6]6	[6]6
HG	[4]1	[4]4	[4]4	[4]4	[4]4	[4]4

Chemins analysés

A - D - C - F - H - G - I
A - C - H - G - I
A - D - G - I
A - B - E - I
A - B - C - F - I
A - C - F - I

Algorithme de Ford Fulkerson

Diminution de la CR en bleu. Augmentation en rouge.

Arcs

AB	[5]5
AC	[8]7
AD	[9]3
BC	[7]1
BE	[4]4
CF	[10]6
CH	[5]3
DC	[1]1
DG	[5]2
DH	[4]0
EI	[5]4
FH	[5]1
FI	[5]5
GI	[6]6
HG	[4]4