

2 Graphes de flots (en C)

Introduction aux graphes de flot

Définition 1 Un graphe de flot est un quintuplet $G = (S, A, c, s, t)$ où :

- (S, A) est un graphe orienté sans boucle (sans arc de la forme (v, v)) ;
- $s \in S$ est une source de (S, A) (degré entrant nul) et $t \in S$ est un puit de (S, A) (degré sortant nul) ;
- $c : A \rightarrow \mathbb{R}_+$ est une fonction dite de capacité.

On étend c à tous les couples (u, v) de sommets en posant $c(u, v) = 0$ si $(u, v) \notin A$.

La figure 1 représente un graphe de flot G_0 . Les capacités sont représentées sur les arcs.

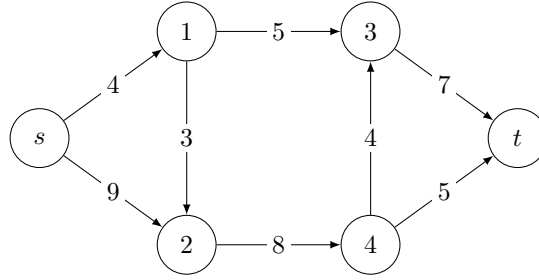


FIGURE 1 – Le graphe de flot G_0 .

Définition 2 Soit $G = (S, A, c, s, t)$ un graphe de flot. On appelle flot sur G une fonction $f : S^2 \rightarrow \mathbb{R}$ vérifiant les trois propriétés suivantes :

- pour $u, v \in S$, $f(u, v) = -f(v, u)$
- pour $u, v \in S$, $f(u, v) \leq c(u, v)$
- pour $u \in S \setminus \{s, t\}$, $\sum_{v \in S} f(u, v) = 0$

antisymétrie

respect de la capacité

conservation

On appelle débit du flot f la valeur $|f| = \sum_{u \in S} f(s, u)$.

La figure 2 représente le graphe de flot G_0 et un flot f compatible avec G . Pour chaque arc (u, v) , on représente sur l'arc $f(u, v)/c(u, v)$. Les autres valeurs nulles et négatives de f peuvent se déduire des valeurs positives.

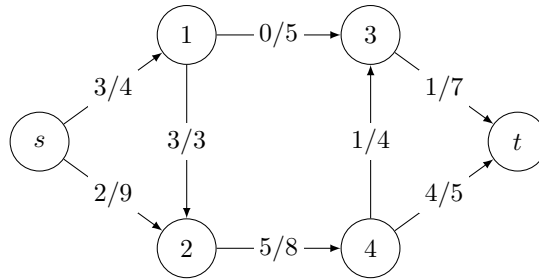


FIGURE 2 – Le graphe de flot G_0 et un flot f .

Dans la suite du sujet, G désignera toujours un graphe de flot et f un flot sur G .

1. Justifier que $f(u, v) > 0$ implique $(u, v) \in A$ et $f(u, v) < 0$ implique $(v, u) \in A$.

Définition 3 Pour $u \in S$, on appelle :

- flux sortant de u la quantité $\phi_+(u) = \sum_{v \in S, f(u,v) > 0} f(u, v)$;
- flux entrant de u la quantité $\phi_-(u) = \sum_{v \in S, f(u,v) < 0} f(v, u)$;
- flux net de u la quantité $\phi(u) = \phi_+(u) - \phi_-(u)$.

2. Montrer que la condition **conservation** est équivalente à la propriété suivante :

$$\forall u \in S \setminus \{s, t\}, \phi(u) = 0$$

3. Montrer que $|f| = \phi(s) = -\phi(t)$.

Le problème MAXFLOW, auquel on s'intéresse dans le reste du sujet, est défini comme suit :

Entrée : un graphe de flot G

Sortie : un flot f maximal sur G , c'est-à-dire un flot tel que $|f|$ soit maximal.

4. Représenter graphiquement une solution à MAXFLOW pour l'instance représentée sur la figure 1. On justifiera que le flot proposé est maximal.

Algorithme de Ford-Fulkerson

Définition 4 — La capacité disponible d'un arc $(u, v) \in A$ est la quantité $c(u, v) - f(u, v)$.

- Un arc est dit saturé si sa capacité disponible vaut zéro.
- La capacité disponible d'un chemin γ de s à t est le minimum des capacités disponibles de ses arcs.
- Un chemin de s à t est dit saturé si sa capacité disponible vaut zéro.

Si un chemin γ de s à t n'est pas saturé, il dispose d'une capacité disponible $m > 0$. On définit alors l'action de *saturation* du chemin γ comme la modification suivante de f :

- pour chaque arc (u, v) de γ :
 - $f(u, v) \leftarrow f(u, v) + m$;
 - $f(v, u) \leftarrow f(v, u) - m$

On remarquera que, après saturation du chemin γ , f est toujours un flot sur G et que γ est désormais saturé.

La figure 3 montre le résultat de la saturation du chemin $(s, 2, 4, 3, t)$ dans le graphe G_0 à partir du flot f de la figure 2. On a augmenté le débit de 3 le long du chemin. Les nouveaux arcs saturés sont $(2, 4)$ et $(4, 3)$.

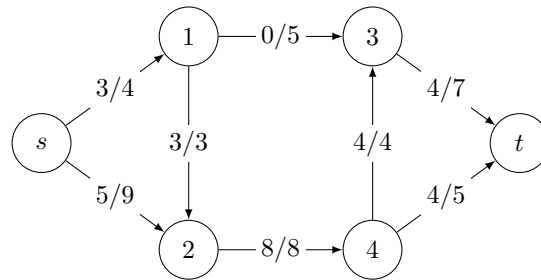


FIGURE 3 – Le graphe de flot G_0 et le flot f après saturation du chemin $(s, 2, 4, 3, t)$.

On considère l'algorithme suivant :

Algorithme 1 – Algorithme glouton pour MaxFlow

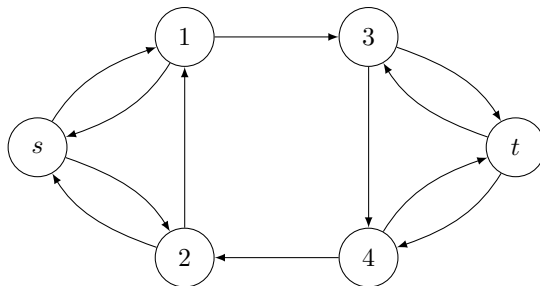
Entrées : Un graphe de flot $G = (S, A, c, s, t)$
 $f(u, v) \leftarrow 0$ pour tout $(u, v) \in S^2$
tant que il existe un chemin non saturé de s à t **faire**
 saturer ce chemin
renvoyer f

5. Expliquer comment trouver un chemin non saturé de s à t dans un graphe de flot, étant donné un flot f .
6. Terminer l'exécution de l'algorithme 1 sur le graphe G_0 à partir du flot f de la figure 3 et représenter graphiquement le résultat.

La question précédente et la question 4 montrent que l'algorithme 1 ne renvoie pas toujours un flot maximal. Pour corriger ce problème, il faut s'autoriser à faire « refluer » le flot en arrière le long d'un arc.

Définition 5 On définit le graphe résiduel $G_f = (S, A_f)$ où $A_f = \{(u, v) \in S^2 \mid c(u, v) - f(u, v) > 0\}$. Un chemin élémentaire de s à t dans G_f est appelé chemin améliorant pour f .

On a représenté ci-dessous le graphe résiduel associé au flot de la figure 3 :



7. Justifier que si $(u, v) \in A_f$, alors $(u, v) \in A$ ou $(v, u) \in A$.

Remarque 1 On fera bien attention à ne pas oublier dans la suite que le graphe résiduel peut contenir des arcs qui n'étaient pas dans le graphe G initial.

8. Représenter graphiquement le graphe résiduel de G_0 pour le flot obtenu à la question 6.

L'algorithme de Ford-Fulkerson est alors le suivant :

Algorithme 2 – Ford-Fulkerson

Entrées : Un graphe de flot $G = (S, A, c, s, t)$
Sorties : Un flot maximal f sur G
 $f(u, v) \leftarrow 0$ pour tout $(u, v) \in A$
tant que il existe un chemin améliorant pour f **faire**
 saturer ce chemin
renvoyer f

9. Terminer l'exécution de l'algorithme de Ford-Fulkerson sur le graphe G_0 avec le flot obtenu à la question 6.

Programmation

On travaille dans le langage C, et l'on suppose avoir inclus tous les entêtes standard :

```
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include <stdbool.h>
#include <assert.h>
#include <limits.h> // fournit INT_MAX, INT_MIN (entre autres)
```

On suppose de plus disposer des deux fonctions suivantes (dont la spécification devrait être évidente) :

```
int min(int x, int y);
int max(int x, int y);
```

On représente en C un graphe de flot par la structure suivante :

```
typedef int vertex;

struct flow_graph {
    int n;
    vertex s;
    vertex t;
    vertex **adj;
    int *degrees;
    int **capacity;
};
```

```
typedef struct flow_graph flow_graph;
```

- L'ensemble S des sommets est $[0 \dots n - 1]$.
- s et t sont respectivement le sommet s source et le sommet t puits.
- $degrees$ est un tableau de taille n tel que le degré sortant du sommet u soit égal à $degrees[u]$.
- adj est un tableau de taille n , dont les éléments sont des tableaux d'entiers.
- Pour $u \in [0 \dots n - 1]$, le tableau $adj[u]$ est de taille $degrees[u]$ et contient les successeurs de u .
- $capacity$ est une matrice de taille $n \times n$ (un tableau de n tableaux de n entiers chacun).
- $capacity[u][v]$ indique la valeur de $c(u, v)$.

Remarque 2 Important : on supposera que l'arc (v, u) existe dans le graphe dès que l'arc (u, v) existe (quitte à ce que $c(v, u)$ soit égal à zéro). Autrement dit, si v apparaît dans le tableau $adj[u]$, alors u apparaît dans le tableau $adj[v]$.

10. Écrire une fonction `zeroes` prenant en entrée un entier n (supposé strictement positif) et renvoyant une « matrice » de taille $n \times n$ initialisée avec des zéros. Par *matrice*, on entend un pointeur vers un bloc alloué `mat` de taille n , telle que `mat[i]` soit un bloc alloué de taille n rempli de zéros, et ce pour $0 \leq i < n$.

```
int **zeroes(int n);
```

11. Écrire une fonction `free_matrix` permettant de libérer la mémoire associée à une matrice du type de celle renvoyée par `zeroes`.

```
void free_matrix(int **mat, int n)
```

Dans toute la suite, un flot sera représenté par une matrice d'entiers. On pourra toujours supposer que, si une fonction prend en entrée un graphe de flot G et un flot f représenté par un f de type `int**`, alors f a la bonne dimension (est une matrice $n \times n$, autrement dit).

On souhaite implémenter une structure de file dotée de l'interface suivante :

```
// création d'une file vide, complexité O(1)
queue *queue_create(void);

// libération de la mémoire, complexité O(longueur)
void queue_free(queue *q);

// nombre d'éléments présents dans la file, complexité O(1)
int queue_length(queue *q);

// extraction du plus ancien élément (erreur si vide), complexité O(1)
vertex queue_pop(queue *q);

// ajout d'un élément, complexité O(1)
void queue_push(queue *q, vertex v);
```

On choisit de réaliser cette structure à l'aide d'une liste simplement chaînée, en conservant un pointeur vers le premier élément de la liste et un pointeur vers le dernier élément.

```
struct cell {
    vertex data;
    struct cell *next;
};
typedef struct cell cell;

struct queue {
    cell *left;
    cell *right;
    int len;
};
typedef struct queue queue;
```

Dans la structure `queue`, le champ `left` pointe vers la cellule la plus ancienne (qui sera la tête de la liste), le champ `right` vers la plus récente (qui sera le dernier élément de la liste). Le champ `len` indique le nombre d'éléments actuellement présents dans la liste. On maintiendra l'invariant suivant :

- si la file est vide, `len` vaut 0 et `left` et `right` valent tous les deux NULL;
- sinon, ni `left` ni `right` ne sont égaux à NULL.

On effectue les insertions à droite de la liste suivant le principe suivant (les éléments modifiés ou ajoutés apparaissent en gras, ceux supprimés en pointillés) :

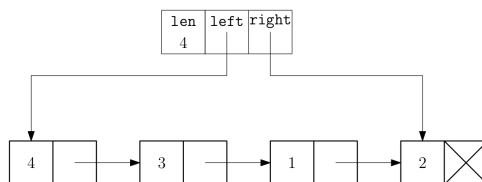


FIGURE 4 – File $\leftarrow (4, 3, 1, 2) \leftarrow$.

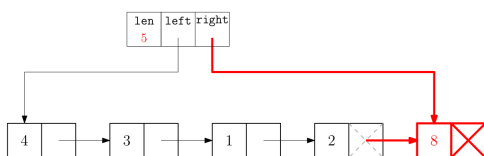


FIGURE 5 – Insertion de 8.

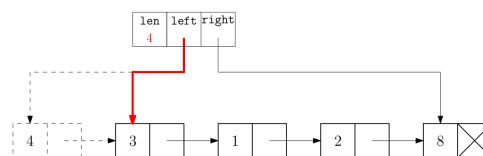


FIGURE 6 – Extraction (on renvoie 4).

12. Écrire une fonction `queue_create` renvoyant (un pointeur vers) une nouvelle file vide.
13. Écrire une fonction `queue_length` qui renvoie la longueur d'une file.
14. Écrire une fonction `queue_push` qui ajoute un élément à une file. L'insertion se fera en accord avec le schéma de la page précédente, c'est-à-dire « à droite » de la liste. On prendra garde à gérer correctement l'insertion dans une file vide.
15. Écrire une fonction `queue_pop` qui extrait l'élément le plus ancien (c'est-à-dire le plus « à gauche ») d'une file. On vérifiera que l'opération est licite (au moyen d'une assertion), et l'on prendra garde à gérer correctement l'extraction depuis une file ne contenant qu'un seul élément.
16. Écrire une fonction `cell_free` qui libère la mémoire associée à une cellule, ainsi qu'à toutes les cellules qui la suivent dans la liste.

```
void cell_free(cell *c);
```

17. Écrire une fonction `queue_free` qui libère la mémoire associée à une file.
18. Pourrait-on facilement (et efficacement) insérer un élément à gauche de la file ? supprimer un élément à droite de la file ?
19. Écrire une fonction `bfs_residual` prenant en entrée un graphe de flot G à n sommets et un flot f sur G , et effectuant une recherche en largeur d'un chemin de s à t dans le graphe résiduel G_f . La fonction renverra un pointeur vers un bloc alloué `parent` de taille n codant l'arbre associé à ce parcours de la manière suivante :

- `parents[g->s]` vaut `g->s`;
- pour chaque autre sommet u visité lors du parcours, `parents[u]` vaut v , où v est le sommet depuis lequel on a exploré u ;
- pour les sommets u n'ayant pas été explorés, `parents[u]` vaut -1 .

On demande une complexité en $(|S| + |A|)$.

```
vertex *bfs_residual(flow_graph *g, int **f)
```

20. Écrire une fonction `path_capacity` qui prend en entrée un graphe de flot G , un flot f et un arbre de parcours `parent` tel que renvoyé par la fonction `bfs_residual` et renvoie la capacité disponible du chemin reliant s à t dans l'arbre `parent`. Si un tel chemin n'existe pas, la fonction renverra zéro.

```
int path_capacity(flow_graph *g, int **f, vertex *parent)
```

21. Écrire une fonction `saturate_path` prenant en entrée un graphe de flot G , un flot f et un arbre de parcours `parent` et ayant pour effet de saturer le chemin éventuel reliant s à t dans l'arbre `parent` (en modifiant f). Cette fonction renverra `true` s'il y avait un chemin, `false` sinon.

```
bool saturate_path(flow_graph *g, int **f, vertex *parent)
```

22. Écrire une fonction `step` prenant en entrée un graphe de flot G et un flot f et ayant le comportement suivant :
 - s'il n'existe pas de chemin améliorant pour f , alors f n'est pas modifié et la fonction renvoie `false`;
 - sinon, la fonction détermine un chemin améliorant, modifie f en saturant ce chemin et renvoie `true`.

```
bool step(flow_graph *g, int **f)
```

23. Déterminer la complexité de la fonction `step`.
24. Écrire une fonction `ford_fulkerson` prenant en entrée un graphe de flot G et renvoyant le flot f calculé par l'algorithme de Ford-Fulkerson : `int **ford_fulkerson(flow_graph *g)`.

3 Complexité et terminaison

25. Dans cette question, on suppose que les capacités sont entières, mais on ne fait pas d'hypothèse sur l'algorithme de parcours utilisé pour trouver un chemin améliorant, à part que cette recherche se fait en $(|S| + |A|)$. Montrer que l'algorithme de Ford-Fulkerson termine, et majorer sa complexité temporelle en fonction de $|S|$, $|A|$ et M , où M est le débit d'un flot maximal sur G .

4 Correction de l'algorithme

Définition 6 Soit $G = (S, A, c, s, t)$ un graphe de flot. On appelle coupe de G un ensemble $X \subseteq S$ tel que $s \in X$ et $t \in \bar{X}$, où $\bar{X}$ désigne le complémentaire de X dans S .

La capacité d'une coupe X est la quantité $C(X) = \sum_{(u,v) \in X \times \bar{X}} c(u,v)$.

26. Dans le graphe G_0 donné figure 1, déterminer la capacité de la coupe $X = \{s, 1, 3\}$.

27. Soit f un flot de G . Montrer que $|f| = \sum_{(u,v) \in X \times \bar{X}} f(u,v) \leq C(X)$.

28. Montrer l'équivalence entre les trois propriétés suivantes :

- (a) f est un flot maximal ;
- (b) il n'existe pas de chemin améliorant pour f ;
- (c) il existe une coupe X telle que $|f| = C(X)$.

29. En déduire la correction de l'algorithme de Ford-Fulkerson.

5 Commandes pour les fonctions de hachage

Opérations sur les tables de hachage : Le module `Hashtbl` offre les fonctions suivantes :

- `('a, 'b) Hashtbl.t`
The type of hash tables from type `'a` to type `'b`.
- `create : int -> ('a, 'b) t`
`Hashtbl.create n` creates a new, empty hash table, with initial size n . For best results, n should be on the order of the expected number of elements that will be in the table. The table grows as needed, so n is just an initial guess.
- `add : ('a, 'b) t -> 'a -> 'b -> unit`
`Hashtbl.add tbl key data` adds a binding of key to data in table `tbl`.
- `remove : ('a, 'b) t -> 'a -> unit`
`Hashtbl.remove tbl x` removes the current binding of x in `tbl`, restoring the previous binding if it exists. It does nothing if x is not bound in `tbl`.
- `mem : ('a, 'b) t -> 'a -> bool`
`Hashtbl.mem tbl x` checks if x is bound in `tbl`.
- `iter : ('a -> 'b -> unit) -> ('a, 'b) t -> unit`
`Hashtbl.iter f tbl` applies f to all bindings in table `tbl`. f receives the key as first argument, and the associated value as second argument. Each binding is presented exactly once to f .
- `find_opt : ('a, 'b) t -> 'a -> 'b option`
`Hashtbl.find_opt tbl x` returns the current binding of x in `tbl`, or `None` if no such binding exists.