

Projet tutoré de SD-Graphes : Problème du flot maximum

Contexte du projet : Le but du projet est de faire l'algorithme de DINIC en C qui permettra de résoudre un problème de flot maximum.

Bien

I. Description de l'algorithme de DINIC

A. Algorithme pour chercher le plus court chemin

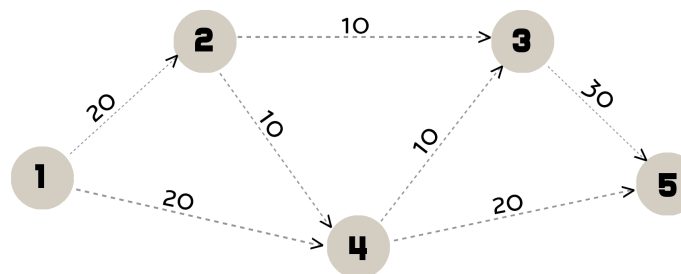
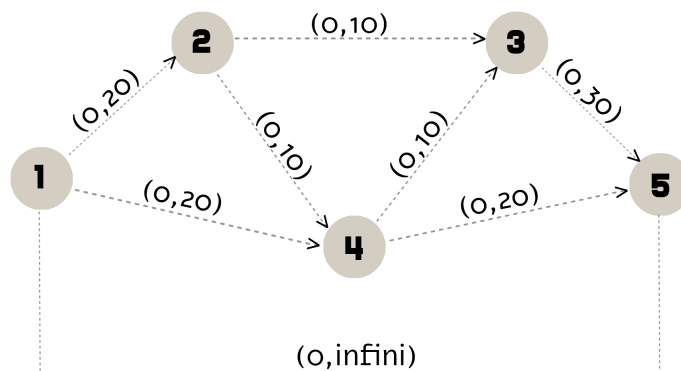
L'algorithme permettant de trouver le plus court chemin en nombre d'arcs est l'algorithme de parcours en largeur.

Bien, mais ici on est dans la justification du choix de l'algo largeur et son explication, plus que dans l'algo de DINIC vraiment...

B. Description de l'algorithme DINIC

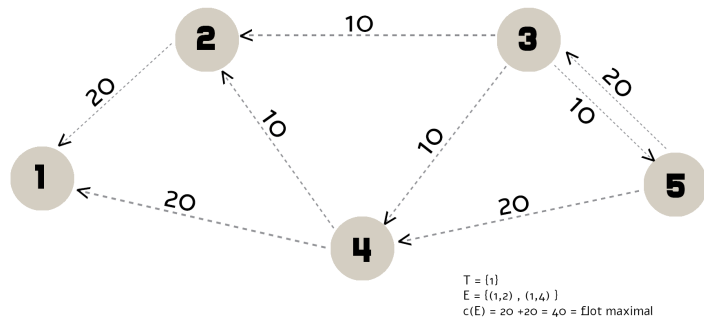
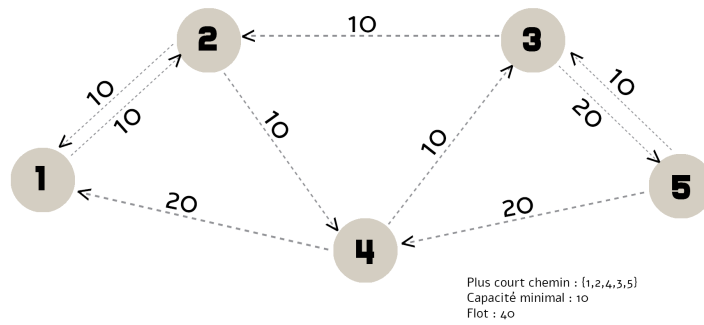
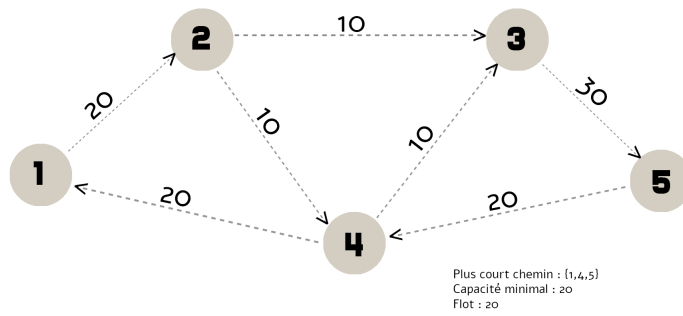
L'algorithme de parcours en largeur est un algorithme parcourant le graphe par couches. Les sommets de la couche n sont à une distance n du sommet source. L'exploration commence donc par la source, puis ses successeurs, puis les successeurs de ses successeurs, ... jusqu'à explorer le sommet puits et on sait alors que l'on peut s'arrêter. Grâce à un tableau répertoriant les prédécesseurs des sommets marqués (par convention, le sommet prédécesseur du sommet source est lui-même), on peut retrouver le chemin le plus court menant d'un sommet à un autre (ici, menant du sommet source au sommet puits).

C. Déroulement de l'algorithme de DINIC sur un réseau

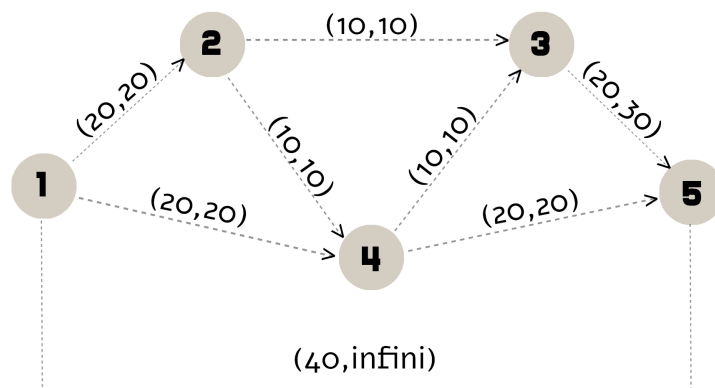


Plus court chemin : {1,4,5}
Capacité minimal : 20
Flot: 20

Bien



OK pour le déroulement,
 mais on ne sait pas trop quand nous
 sommes dans le graphe / graphe d'écart...



II. Analyse des structures de données pour implémenter le réseau et le graph d'écart associé

A. Comparaison

	Matrice d'incidence	Tableaux Sommets - Successeurs	Représentation par liste - successeur
Coût Mémoire	On a une matrice de dimension n par m . Donc, l'initialisation c'est un tableau de 2 dimensions $n \times m$ cases donc la complexité en coût mémoire vaut $(n \times m) \times \text{sizeof(int)}$.	La complexité en coût mémoire d'un tableaux sommets-successeurs est en $O(n + m)$, n étant le nombre de sommets et m étant le nombre d'arcs.	On a un tableau de taille n qui correspond au nombres de sommets donc n allocation dynamique de type liste. On a une liste chaînée de m maillons donc allocation dynamique de m maillon. Donc la complexité en coût de mémoire vaut $n + m$.
Coût de Traitement	Il faut parcourir la matrice afin de trouver les successeurs d'un sommet vu que c'est une matrice d'incidence donc on ne peut pas se concentrer sur une seule ligne ou colonne. $O(n \times m)$	On a directement l'indice dans le tableau de successeurs en fonction du tableau de sommet. Donc on n'a pas vraiment de traitement à effectuer. $O(n)$	On a une liste chaînée donc il y a le coût de parcourir des maillons. $O(n)$
Coût d'ajout / suppression	Il faut redimensionner la table 2D avec un realloc. coût ?	Redimensionner la table de successeurs de 1 et il faut insérer au bon indice et décaler le reste du tableau.	Ajout d'un maillon simple au sommet correspondant.

TBien

Les meilleures solutions pour chaque problème sont surlignées, après avoir comparé toutes les possibilités et leurs coûts respectifs, on a choisi d'utiliser une représentation liste - successeurs. Bien

Il aurait pu être intéressant de brièvement décrire les SD comparées

B. Choix des structures de données

/* Hypothèse de départ : tous les noms de sommets sont des numéros allant de 1 jusqu'à n (n soit le nombre de sommets) */

Bien

```
struct maillon_graph_reseau {
    int id;
    int flot;
    int capacite;
    struct maillon_graph_reseau* next;
};
```

/* Comme on travaille avec des numéros de 1 à n (n étant le nombre de sommets d'un graphe), on peut ignorer l'utilisation d'une liste et donner directement un pointeur vers un maillon de départ mais pour être plus claire et précis, on a adapté d'utiliser des listes */

OK

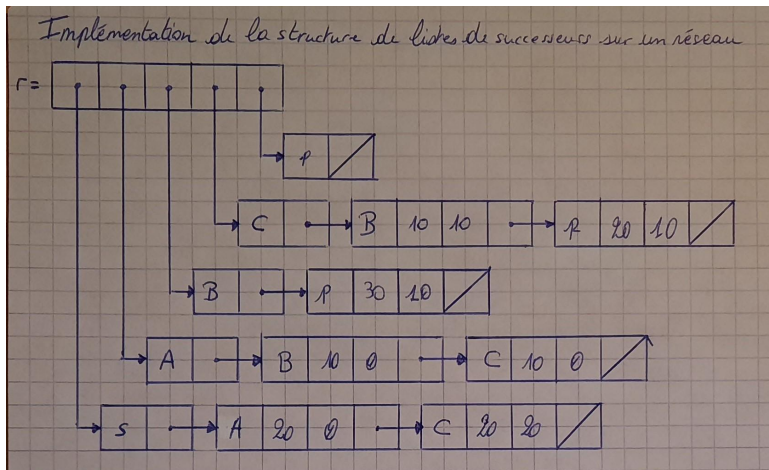
```
struct liste_graph_reseau {
    int id;
    struct maillon_graph_reseau* head;
};
```

```
typedef tabSommets liste_graph_reseau[N];
```

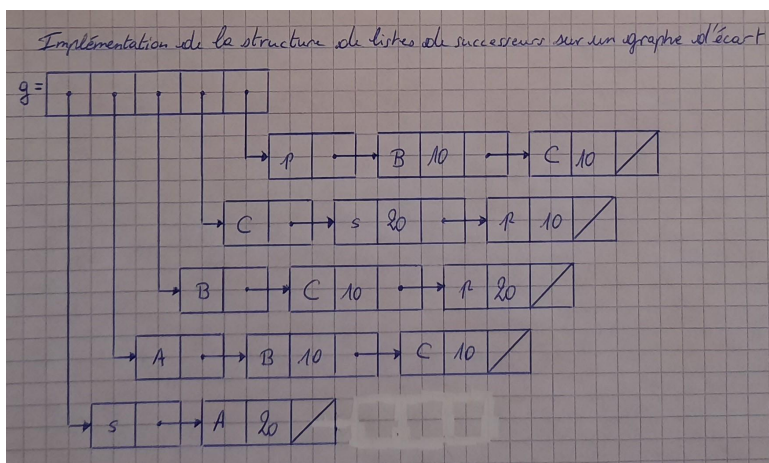
```
struct maillon_graph_ecart {
    int id;
    int flot_entrant;
    struct maillon_graph_ecart* next;
};
```

```
struct liste_graph_ecart {
    int id;
    struct maillon_graph_ecart* head;
};
```

C. Implémentation des structures de données sur un réseau et un graphe d'écart



OK



III. Décomposition de l'algorithme de DINIC

A. Structures de données du chemin

```
struct maillon_chemin {
    int id;
    struct maillon_chemin* next;
};

struct liste_chemin{
    int id;
    struct maillon_chemin* head;
};
```

voir si pertinent de garder une autre info?

B. Description des procédures de l'algorithme de DINIC (pseudo-langage)

```
#define NIL_lge (struct liste_graphe_ecart) 0
#define NIL_mge (struct maillon_graphe_ecart) 0

#define NIL_lr (struct liste_reseau) 0
#define NIL_mr (struct maillon_reseau) 0

#define NIL_lc (struct liste_chemin) 0
#define NIL_mc (struct maillon_chemin) 0
```

```
Action buildGraph(DIMACS, T, source, sink, n) :
    D : DIMACS : le fichier qui permet de construire le graph.
    R : T : tabSommets,
        source : source du réseau,
        sink : puits du graphe,
        n : nombre de sommets dans le réseau.
    L : src : seconde valeur sur la ligne (source de l'arc),
        dst : troisième valeur sur la ligne (destination de l'arc),
        capt : quatrième valeur sur la ligne (capacité de l'arc),
        indice : cinquième valeur sur la ligne (indication c:
commentaire, p:problème, n : description du sommet [source ou puits] a :
arc).
    TantQue la lecture du fichier n'est pas terminée
        Si indice = n
            Si WHICH == s
                source = value in the line
            Sinon
                sink = value in the line
            FinSi
        Sinon Si indice = p
            Initialiser T de taille n
        Sinon Si indice = a
            ajout_en_tete(dest, cap, T[src - 1]) /* on donne src-1
comme indice car on s'adapte à l'hypothèse de départ :sommets allant de
1 à n */ /* dest et cap sont lu dans le fichier = value in line */
            FinSi
        FinTantQue

FinAction
```

```

Action ajout_en_tete(dest, cap, L) :
    D : dst : destination de l'arc,
    cap : capacité de l'arc.
    D/R : liste_graph L.
    L : nouveau : struct maillon_reseau.

    struct maillon_reseau* nouveau
    /* en C, il faut faire une allocation dynamique */
    /* affectation des valeurs */
    nouveau.id = dst;
    nouveau.flot = 0;
    nouveau.capacite = cap;
    nouveau.next = L.tete;
    L.tete = nouveau;
FinAction

```

```

Action Initialiser_tabSommets (T, n) :
    D : n : int (nombre de noeuds).
    D/R : T : tabsommets

    Redimensionner le tableau
    Pour i allant de 0 à n :
        T[i].id = i + 1 /* on donne src-1 comme indice car on s'adapte
à l'hypothèse de départ */
    FinPour

FinAction

```

```

Fonction minCapa (chemin, graph) : (retourne int)
    D : chemin : liste_chemin,
        graph : tabSommets (Graph de réseau).
    L : min : int (capacité minimum d'un chemin).
    struct maillon_chemin* parcours

    parcours = chemin.head
    prec_id = chemin.id
    min = findCapaciteById(prec_id,parcours.id,graph)
    prec_id = parcours.id
    parcours = parcours.next


```

```

    TantQue parcours != NIL_mc :
        val = findCapaciteById(prec_id,parcours.id,graph) /* la
fonction retourne -1 s'il ne trouve pas de capacité, c'est un cas
limite, ce qui n'arrivera pas dans notre contexte */
        Si min > val et val != -1 :
            min = val
        FinSi
        prec_id = parcours.id
        parcours = parcours.next
    FinTantQue
    retourner (min)
FinFonction

```

Voir ma remarque ci-dessus
sur la définition de chemin



```

Fonction findCapaciteById(src, dst, graph): (retourne int)
    D : src : int (source de l'arc),
        dst : int (destination de l'arc),
        graph : tabSommets (Graph de réseau).
    L : parcours : struct maillon_reseau.

    struct maillon_reseau* parcours
    parcours = graph[src - 1].head /* on donne src - 1 comme indice
car on s'adapte à l'hypothèse de départ */
    TantQue parcours != NIL_mr et parcours.id != dst :
        parcours = parcours.next
    FinTantQue
    Si parcours == NIL_mr :
        retourner -1
    Sinon
        retourner parcours.capacite
    FinSi
FinFonction

```



```
Action updateFlowInNet(graph, graphEc):
```

```
  D : graphEc : tabSommetsEc (graph d'écart).
```

```
  D/R : graph : tabSommets (graph de réseau).
```

```
  struct maillon_reseau* parcours
```

```
  Pour i allant de 0 à len(tabSommets) :
```

```
    parcours = graph[i].head
```

```
    TantQue parcours != NIL_mr :
```

```
      parcours.flot = parcours.flot +
```

```
      getFlotFromGraphEc(id,parcours.id,graphEc)
```

```
      parcours = parcours.next
```

```
    FinTantQue
```

```
  FinPour
```

```
FinAction
```

Vrai? Si on repousse du flot?

```
Fonction getFlotFromGraphEc(src, dest, graphEc):
```

```
  D : graphEc : tabSommetsEc (graph d'écart),
```

```
  src : int (source de l'arc),
```

```
  dest : int (destination de l'arc).
```

```
  struct maillon_graph_ecart* parcours
```

```
  parcours = graphEc[dest - 1].head /* on donne src-1 comme indice  
  car on s'adapte à l'hypothèse de départ */
```

```
  TantQue parcours != NIL_mge et parcours.id != src :
```

```
    parcours = parcours.next
```

```
  FinTantQue
```

```
  Si parcours != NIL_mge :
```

```
    retourner parcours.flot_entrant
```

```
  Sinon
```

```
    retourner 0
```

```
  FinSi
```

```
FinFonction
```

```

Action buildRG(reseau, n, graphe_ecart) :
    D : reseau (tableau contenant des listes décrivant un réseau à
partir duquel on réalise le graphe d'écart),
        n (nombre de sommets dans le réseau et le graphe d'écart).
    R : graphe_ecart (tableau contenant des listes décrivant le graphe
d'écart construit à partir du réseau reseau).
    L : i (entier parcourant le tableau du graphe d'écart),
        M_r (Maillon_reseau maillon permettant de parcourir une liste de
successeurs dans un réseau).
    Liste_graphe_ecart graphe_ecart[n]

    Pour i allant de 0 à n-1 Faire :
        init_liste_graphe_ecart(graphe_ecart[i], i+1) // la liste
représente les successeurs de i+1, on fixe donc l'id de la liste à i+1
        M_r = reseau[i].tete
        TantQue M_r != NIL_mr Faire :
            Si M_r.capacite - M_r.flot != 0 Faire :
                ajout_en_tete_graphe_ecart(graphe_ecart[i], M_r.id,
M_r.capacite)
            FinSi
            M_r = M_r.suivant
        FinTantQue
    FinPour
FinAction

```

[A voir en fonction des modifications éventuelles sur la structure du graphe écart](#)

```

Action shortPath(graphe_ecart, source, puits, n, plus_court_chemin) :
    D : graphe_ecart (tableau contenant le graphe d'écart que l'on
parcourt pour trouver le plus court chemin),
        source, puits (entiers indiquant le sommet source et le sommet
puits du graphe d'écart),
        n (nombre de sommets dans le graphe d'écart).
    R : plus_court_chemin (Liste_chemin décrivant le plus court chemin
en nombre d'arcs dans le graphe d'écart).
    L : file (File permettant le parcours en largeur du graphe),
        fini (Booléen indiquant si le puits à été ajouté à la file et on
peut s'arrêter),
        sommet (int sommet défilé pour ajouter ses successeurs dans la
file puis sommet permettant de remonter les prédécesseurs du puits),
        prec (int sommet prédecesseur de 'sommet' permettant de remonter
les prédécesseurs du puits).
    init_liste_chemin(plus_court_chemin, source)
    int predecesseurs[n]
    Pour i allant de 0 à n-1 Faire :
        predecesseurs[i] = 0
    FinPour
    predecesseurs[source - 1] = source
    init_file(file, n)
    enfiler(file, source)
    fini = Faux
    TantQue pas fini Faire :
        sommet = defiler(file)
        fini = enfiler_successeurs(graphe_ecart, file, sommet,
predecesseurs, puits) // ajoute les successeurs de sommet dans la file
et indique si le puits fait partie des successeurs ajoutés ou pas
    FinTantQue

    sommet = puits
    ajout_en_tete_chemin(plus_court_chemin, sommet)
    prec = predecesseurs[sommet - 1]
    TantQue prec != predecesseurs[sommet - 1] Faire :
        ajout_en_tete_chemin(plus_court_chemin, prec)
        prec = predecesseurs[sommet - 1]
    FinTantQue
FinAction

```

OK

```

Action updateFlowInRG(chemin, k, graphe_ecart) :
    D : chemin (Liste_chemin décrivant le chemin augmentant le flot),
        k (flot augmentant le flot du graphe d'écart).
    D/R : graphe_ecart (tableau de Liste_graphe_ecart contenant le
    graphe d'écart dont on augmente le flot de k).
    L : id_sommet (int sommet initial),
        M_c (Maillon_chemin successeur d u sommet id_sommet),
        M_ge (Maillon_graphe_ecart parcourant le graphe d'écart pour
    modifier son flot),
        sommet_flot_a_repousser (int sommet successeur au sommet
    id_sommet dont on change le flot de l'arc entre ces deux sommets).

    id_sommet = chemin.tete.id
    M_c = chemin.tete.suivant

    TantQue M_c != NIL_mc Faire :
        M_ge = graphe_ecart[id_sommet - 1].tete
        TantQue M_ge != NIL_mge et M_ge.id != M_c.id Faire :
            M_c = M_c.suivant
        FinTantQue
        Si M_c.flot_entrant - k == 0 Faire :
            retirer_de_la_liste(graphe_ecart[id_sommet], M_c.id)
        Sinon Faire :
            M_c.flot_entrant -= k
        FinSi

        sommet_flot_a_repousser = M_c.id
        M_ge = graphe_ecart[sommet_flot_a_repousser - 1].tete
        TantQue M_ge != NIL_mge && M_ge.id != id_sommet Faire :
            M_ge = M_ge.suivant
        FinTantQue
        Si M_ge == NIL_mge Faire :

ajout_en_tete_graphe_ecart(graphe_ecart[sommet_flot_a_repousser - 1],
id_sommet, k)
        Sinon Faire :
            M_c.flot_entrant += k
        FinSi
        id_sommet = M_c.id
        M_c = M_c.suivant
    FinTantQue
FinAction

```

```
Action init_liste_graphe_ecart(liste_graphe_ecart,
valeur_id_sommet_init) :
    D : valeur_id_sommet_init (int id sommet dont on crée la liste de
successeur).
    D/R : liste_graphe_ecart (Liste_graphe_ecart que l'on crée pour
retrouver les successeurs du sommet initial).

    liste_graphe_ecart = allocation de mémoire pour un élément de taille
(Liste_graphe_ecart)
    liste_graphe_ecart.id = valeur_id_sommet_init
    liste_graphe_ecart.tete = NIL_mge
FinAction
```

```
Action ajout_en_tete_graphe_ecart(liste_graphe_ecart,
valeur_id_sommet_courant, valeur_flot_sommet_courant) :
    D : valeur_id_sommet_courant (int id sommet que l'on ajoute à la
liste de successeurs),
    valeur_flot_sommet_courant (int flot entre le sommet dont
liste_graphe_ecart est la liste de successeurs et le sommet courant).
    D/R : liste_graphe_ecart (Liste_graphe_ecart contenant les
successeurs du sommet initial).

    maillon_graphe_ecart = allocation de mémoire pour un élément de
taille (Maillon_graphe_ecart)
    maillon_graphe_ecart.id = valeur_id_sommet_courant
    maillon_graphe_ecart.flot_entrant = valeur_flot_sommet_courant
    maillon_graphe_ecart.suivant = liste_graphe_ecart.tete
    liste_graphe_ecart.tete = maillon_graphe_ecart
FinAction
```

```

Action retirer_de_la_liste(liste_graphe_ecart, int_id_sommet_a_retirer)
:
    D : int_id_sommet_a_retirer (int id du sommet à retirer de la liste
de successeurs).
    D/R : liste_graphe_ecart (Liste_graphe_ecart de laquelle on va
retirer le sommet souhaité).
    L : maillon_graphe_ecart (maillon permettant de parcourir la liste
et récupère le maillon d'après comme suivant lorsque le suivant est le
maillon à retirer),
        maillon_graphe_ecart_succ (maillon permettant de parcourir la
liste et prend la place de son prédécesseur lorsque celui-ci est le
maillon à retirer).

    maillon_graphe_ecart = liste_graphe_ecart.tete
    maillon_graphe_ecart_succ = maillon_graphe_ecart.suivant
    TantQue maillon_graphe_ecart_succ != NIL_mge et
maillon_graphe_ecart_succ.id != int_id_sommet_a_retirer Faire :
        maillon_graphe_ecart = maillon_graphe_ecart_succ
        maillon_graphe_ecart_succ = maillon_graphe_ecart_succ.suivant
    FinTantQue

    Si maillon_graphe_ecart_succ != NIL_mge Faire :
        maillon_graphe_ecart.suivant = maillon_graphe_ecart_succ.suivant
    FinSi
FinAction

```

```

Action init_liste_chemin(liste_chemin, valeur_id_sommet_init) :
    D : valeur_id_sommet_init (source du chemin).
    D/R : liste_chemin (liste contenant le chemin partant de la source
et allant jusqu'au puits).

    liste_chemin = allocation de mémoire pour un élément de taille
(Liste_chemin)
    liste_chemin.id = valeur_id_sommet_init
    liste_chemin.tete = NIL_lc
FinAction

```

```
Action ajout_en_tete_chemin(liste_chemin, valeur_id_sommet_courant) :  
    D : valeur_id_sommet_courant (sommet à ajouter dans le chemin).  
    D/R : liste_chemin (liste représentant le chemin).
```

```
    maillon_chemin = allocation de mémoire pour un élément de taille  
(Maillon_chemin)  
    maillon_chemin.id = valeur_id_sommet_courant  
    maillon_chemin.suivant = liste_chemin.tete  
    liste_chemin.tete = maillon_chemin  
FinAction
```

```
Action init_file(file, taille) :
```

```
    D : taille (taille de la file, pour ne pas avoir à la  
redimensionner, on la mettra souvent au nombre de noeuds dans le  
graphe).
```

```
    D/R : file (file accueillant les sommets déjà parcourus par le  
parcours en largeur).
```

```
    file.taille = taille  
    file.tab = allocation de mémoire pour un tableau de taille  
(file.taille * int)  
    file.write_end = 0  
    file.read_end = 0  
    file.n = 0  
FinAction
```

```
Action enfiler(file, sommet_a_enfiler) :
```

```
    D : sommet_a_enfiler (sommet à ajouter en fin de file).
```

```
    D/R : file (file dans laquelle on ajoute le nouveau sommet  
parcouru).
```

```
    Si file.n < file.taille Faire :
```

```
        file.tab[file.write_end] = sommet_a_enfiler
```

```
        file.n += 1
```

```
        file.write_end = (file.write_end + 1) % file.taille
```

```
    FinSi
```

```
FinAction
```

```

Fonction enfiler_successeurs(graphe_ecart, file,
sommets_dont_on_doit_enfiler_les_successeurs,
table_predecesseurs_a_mettre_a_jour, sommets_a_trouver):
    D : graphe_ecart (graphe à parcourir pour retrouver les successeurs
du sommet défilé).
        sommets_dont_on_doit_enfiler_les_successeurs (sommets défilés dont
on doit enfiler les successeurs)
        sommets_a_trouver (on arrête le parcours lorsqu'on enfile ce
sommet)
    D/R : file (file dans laquelle on enfile les successeurs du sommet
que l'on vient de défiler)
        table_predecesseurs_a_mettre_a_jour (tableau permettant de
retrouver les prédécesseurs du sommet puits, aussi appelé
sommets_a_trouver)
    L : M_ge (maillon permettant de parcourir la liste des successeurs
du sommet défilé),
        flag (permet de s'arrêter lorsque le puits est ajouté et donc
marqué par le parcours).
    M_ge = graphe_ecart[sommets_dont_on_doit_enfiler_les_successeurs -
1].tete
    flag = Faux
    TantQue M_ge != NIL_mge Faire :
        enfiler(file, M_ge.id)
        table_predecesseurs_a_mettre_a_jour[M_ge.id - 1] =
sommets_dont_on_doit_enfiler_les_successeurs
        Si M_ge.id == sommets_a_trouver Faire :
            flag = Vrai
        FinSi
    FinTantQue
FinFonction

```

```

Fonction defiler(file) :
    D/R : file (file de laquelle on va retirer, défiler le premier
élément).
    L : sommet (sommet à renvoyer si la file n'est pas vide).

    Si file.n > 0 Faire :
        sommet = file.read_end
        file.n -= 1
        file.read_end = (file.read_end + 1) % file.taille
        retourner(sommet)
    Sinon Faire :
        retourner(-1)
    FinSi

```


FinFonction

C. L'algorithme principal (pseudo-langage)

```
Action main() :  
    DIRACS = fichier contenant la description du réseau  
    buildGraph(DIMACS,R, source, sink, n) /* source et sink sont passés  
    en paramètre et on va utiliser ces deux variables pour les fonctions qui  
    suivent. De même, n est le nombre de sommets, une variable qui est  
    passée en paramètre qui sera utilisé après */  
    f = 0  
    DINIC(R, f, source, sink, n)  
    afficher("Le flot est passé de 0 à " + f)  
    retourner(0)  
FinAction
```

```
Action DINIC(R, f, source, sink, n) :  
    D : R (réseau dont on cherche à maximiser le flot),  
        n (nombre de noeuds dans le réseau).  
    D/R : f (flot que l'on maximise).  
    /* Initialisation */  
    buildRG(R, n, graphe_ecart)  
    /* Programme */  
    fini = shortPath(graphe_ecart, source, sink, n, plus_court_chemin)  
    TantQue (pas fini) Faire :  
        min = minCapa (plus_court_chemin, graphe_ecart)  
        updateFlowInRG(plus_court_chemin, min, graphe_ecart)  
        f += min  
        fini = shortPath(graphe_ecart, source, sink, n,  
plus_court_chemin)  
    FinTantQue  
    updateFlowInNet(R,graphe_ecart)  
FinAction
```

Etait demandé / chaque itération / chemin
Mais OK