2019-2020, semestre automne
L3, Licence Sciences et Technologies

LIFAP6: Algorithmique, Programmation et Complexité

Chaine Raphaëlle (responsable semestre automne)
E-mail : raphaelle.chaine@liris.cnrs.fr
http://liris.cnrs.fr/membres?idn=rchaine

Parcours en largeur (BFS)

- Pour un sommet de départ s on commence par visiter tous les successeurs de s **avant** de visiter les autres descendants de s
- Le parcours en largeur consiste à visiter d'abord
  - tous les sommets à distance 1 de s,
  - puis ceux à distance 2 qui n'ont pas été visités,
  - et ainsi de suite …
- Le parcours en largeur permet donc de résoudre les problèmes de plus court chemin dans un graphe non valué

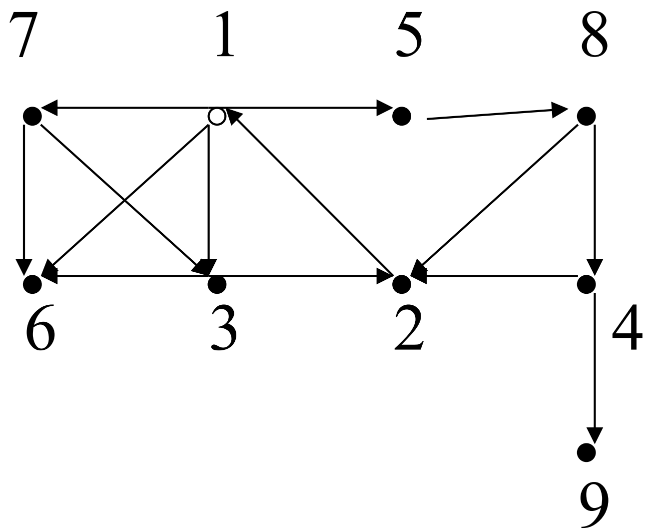
Parcours en largeur (BFS)

- Pour programmer l'algorithme, on utilise une structure de **file**:
  - lorsque à partir de s, on s'apprête à visiter ses successeurs non marqués, il est nécessaire de les ranger successivement dans une file
  - la recherche repartira ainsi de chacun des successeurs de s, à partir du premier.

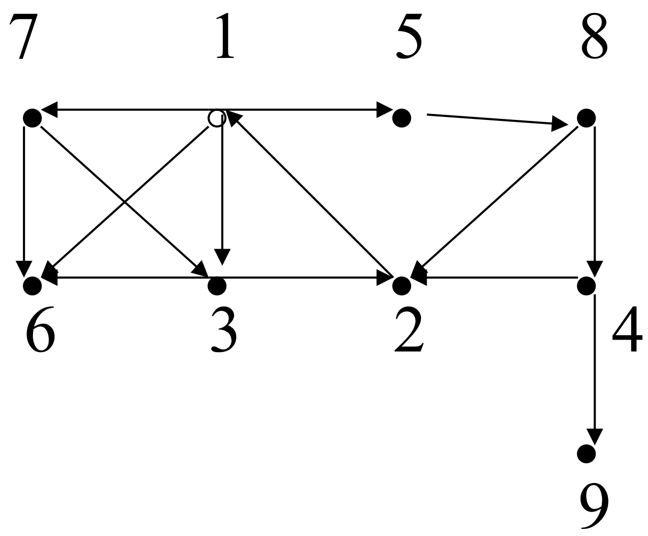
Parcours en largeur (BFS)

```
procédure BFS(donnée-résultat G: graphe, s : noeud)
variables
    u,v : nœud ; F : file
début
  pour tout noeud u ≠ s faire
      colorie(u,blanc); set_père(u, nil); set_dist(u,∞)
  finpour
  colorie(s,gris); set_père(s,nil); set_dist(s,0)
  initialise_file_vide(F); enfile(F,s)
  tant que non est-vide(F) faire
      u ← tête(F);
      pour tout noeud v successeur de u faire
          si couleur(v)=blanc alors
              colorie(v,gris); set_père(v,u); set_dist(v, u.dist +1)
              enfile(F,v)
          finsi
      finpour
      défile(F); colorie(u,noir) (facultatif : 2 couleurs suffisent))
  fintantque
fin
```

- Effectuer le parcours en largeur sur le graphe suivant



- Les sommets sont visités dans l’ordre
  1, 3, 5, 6, 7, 2, 8, 4, 9



- *Complexité en temps* : O(N+M)
  où N est le nombre de nœuds et M le nombre d'arcs
  - en effet, un sommet n'est mis dans la file qu'une seule fois (passage de blanc à gris) et les arêtes sont toutes parcourues 1 fois (découverte des voisins)

- *Complexité en espace* : O(N)
  la file a une longueur au plus en O(N) (si s est connecté à tous les autres sommets)

- En fait, parcours en largeur et en profondeur s’inscrivent dans une même stratégie générale d’exploration des nœuds du graphe
- Diffèrent suivant que les successeurs d’un nœud seront rangés dans une pile ou une file

```
procédure ExplorerGraphe (donnée-résultat G: Graphe, x : noeud)
variable
  E : Salle d'attente de Noeud
début
  pour tout nœud n de G faire
     colorie(n,blanc) …
  finpour
  initialiser(E); ajouter(E,x)
  répéter
     y← sommet(E); retirer_sommet(E),
     si couleur(y)=blanc alors
          colorie(y,gris);
          pour tout nœud z successeur de y faire
             si couleur(z) = blanc alors
                ajouter(z,E)
             finsi
          finpour
          colorie(y, noir) (facultatif))
      finsi
  jusque estvide(E)
fin
```

Si E correspond à une Pile : parcours en profondeur

Si E correspond à une File : parcours en largeur

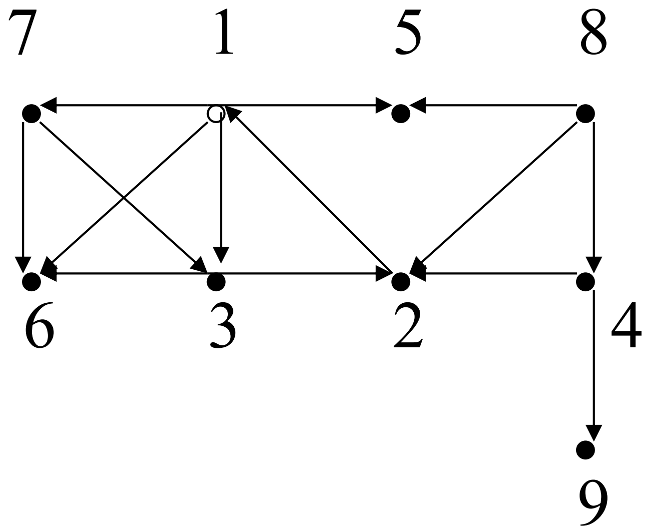
```
procédure ExplorerGraphe (donnée-résultat G: Graphe, x : noeud)
variable
   E : Salle d'attente de Noeud
début
   pour tout nœud n de G faire
      colorie(n,blanc) …
   finpour
   initialiser(E); colorie(x,gris); ajouter(E,x)
   répéter
      y← sommet(E); retirer_sommet(E)
      pour tout nœud z successeur de y faire
         si couleur(z) = blanc alors
            colorie(z, gris)
            ajouter(z,E)
         finsi
      finpour
      colorie(y, noir) (facultatif))
   jusque estvide(E)
fin
```

Si E correspond à une Pile : parcours en profondeur (légèrement différent de celui de la version récursive)

Si E correspond à une File : parcours en largeur

Tri topologique

- Problème :
  - Etant donné un graphe orienté **acyclique** modélisant une relation d'ordre partiel, trouver une relation d'ordre total entre les nœuds qui respecte l'ordre partiel
  - ie. trouver un ordre des nœud tel qu'un nœud soit toujours visité avant ses successeurs
- Utile pour les problèmes de séquencement

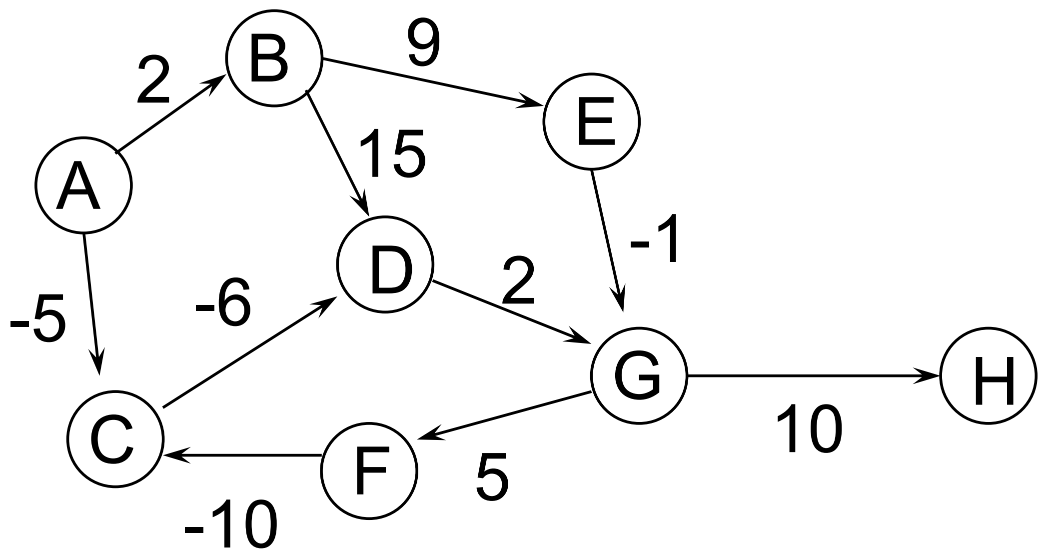


- Si chaque Nœud représente une tâche, la tâche 1 devra être réalisée avant les tâches 5 et 7, etc.

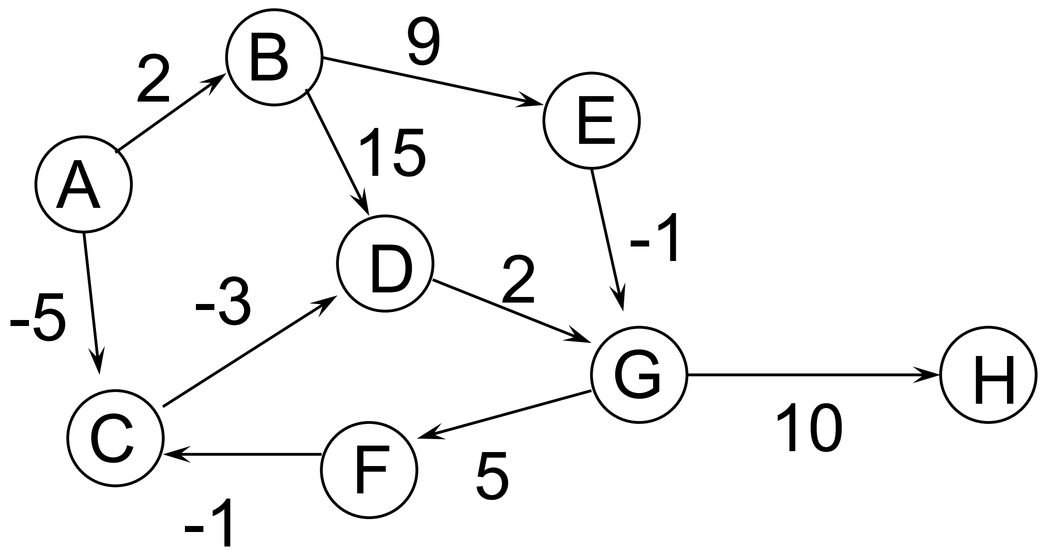
- Le parcours en profondeur permet de résoudre le problème du tri topologique
- Il suffit pour cela de classer les nœuds dans l'ordre inverse où ils sont coloriés en noir (**ordre postfixe inverse**)
- En effet un nœud est colorié en noir APRES les nœuds qu'il a permis de découvrir!

Recherche de plus court chemin

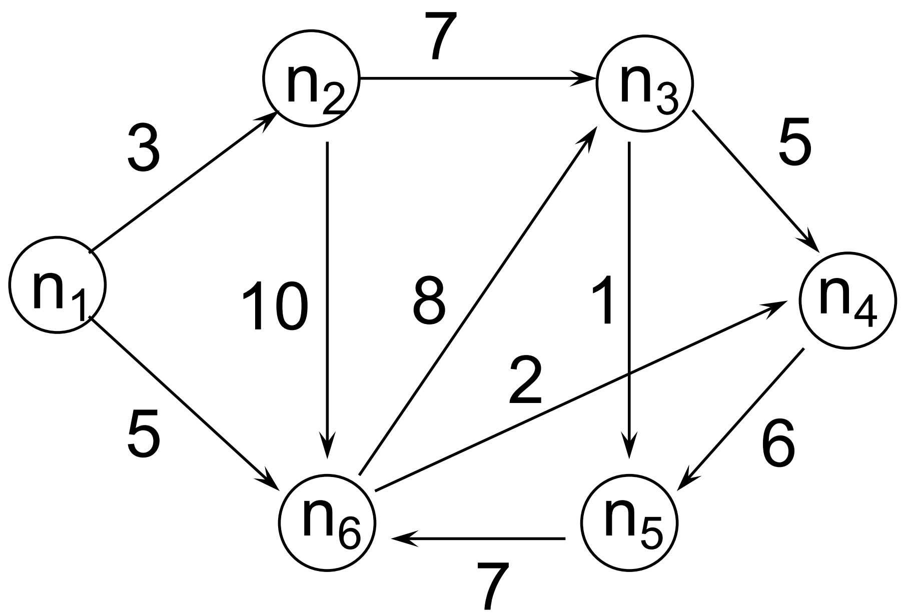
- Dans un graphe valué, il existe un plus court chemin entre 2 nœuds si il n’existe pas de circuit de valeur négative dans un chemin menant de l’un à l’autre
- Un tel circuit est dit absorbant



- Soit G un graphe orienté valué avec des valeurs ≥ 0
  - en fait on peut également avoir des valeurs négatives si cela ne provoque pas l'apparition de circuits absorbants



- Cas d’une représentation par matrice d’adjacence
  - M[i,j]=v_{ij} si les nœuds i et j sont reliés par un arc de valeur v_{ij}
  - M[i,i]=0 (entre un nœud et lui-même)
  - Distance infinie entre 2 nœuds non connectés



	n_1	n_2	n_3	n_4	n_5	n_6
n_1	0	3				5
n_2		0	7			10
n_3			0	5	1	
n_4				0	6	
n_5					0	7
n_6			8	2		0

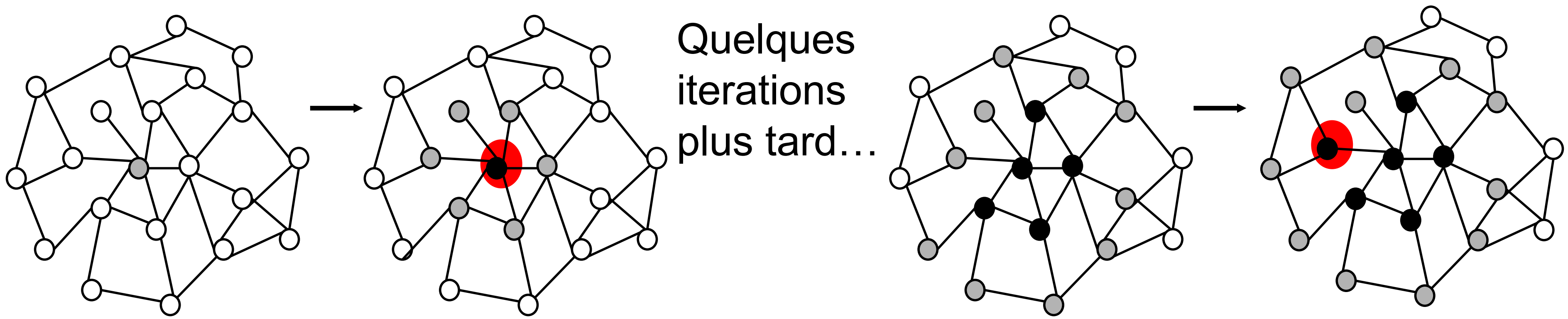
- Cas d’une représentation par liste d’adjacence
  - Plus de problème de matérialisation de la distance infinie

Algorithme de Dijkstra

- But : Connaître les plus courts chemins entre un nœud source donné S et TOUS les nœuds du graphe accessibles depuis S
- Valable uniquement pour les graphes valués positivement
- Construction incrémentale et gloutonne d'un ensemble E_{noir} de nœuds accessibles depuis S
- Initialisation :
  - $E_{noir\ 0}$ vide $E_{gris\ 0}=\{S\}$ Ensemble des nœuds gris
- Passage à l'étape suivante en coloriant en noir un nœud gris
  - $E_{noir\ i+1}=E_{noir\ i} \cup \{$**nœud de E_{gris}** le plus proche de S en empruntant un chemin qui ne traverse que des nœuds de $E_{noir\ i}\}$

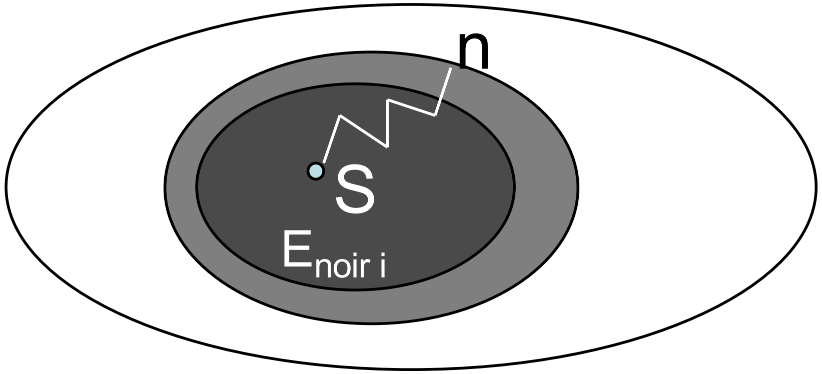
Algorithme de Dijkstra

– Passage à l'étape suivante
  - $E_{noir\ i+1}$=$E_{noir\ i}$ U {nœud **gris** le plus proche de S en empruntant un chemin qui ne passe que par des nœuds noirs}
– Affirmation : Les sommets entrent dans E par ordre croissant de distance à S ☺



- Algorithme glouton :
  - à chaque étape, les choix sont faits sur la base d'une stratégie locale, qui ne sera pas remise en cause plus tard lorsqu'on aura une meilleure vision globale
  - les algorithmes gloutons ont souvent un bon comportement asymptotique en termes de complexité mais ne sont pas toujours exacts …

- L’algorithme de Dijkstra est un algorithme glouton mais exact ☺
  - En effet, à chaque étape, **le nœud n Gris le plus proche de S « en empruntant un chemin qui ne traverse que des nœuds de $E_{noir\ i}$ »** se révèle être **le nœud de $E_{gris\ i}$ le plus proche de S,** même si on a le droit de passer aussi par des sommets gris ou blanc!



- Répartition des nœuds en 3 ensembles :
  - Ensemble blanc : nœuds non atteints par un chemin
  - Ensemble gris : nœuds atteints mais à partir desquels on n'a encore mené aucune exploration et pour lesquels il existe peut-être un meilleur chemin depuis S
  - Ensemble noir : nœuds dont on a fini d'explorer le voisinage et pour lesquels on connaît un plus court chemin depuis S
- Initialisation :
  - Nœuds tous blancs, sauf le sommet S de départ en gris

- Ensemble des nœuds noirs : $\mathbf{E_{noir\ i}}$
- Ensemble des nœuds gris $\mathbf{E_{gris\ i}}$: frontière extérieure de $\mathbf{E_{noir\ i}}$
- Ensemble des nœuds blancs : nœuds non voisins d'un nœud de $\mathbf{E_{noir\ i}}$

Algorithme de Dijkstra

- Mise en œuvre :
  A chaque étape, pour connaître rapidement le nœud Gris qui est le plus proche de S
  - Utilisation d'un tableau PCD (Plus Courte Distance) indicé par les numéros des sommets contenant 2 infos dist et pred.
    - PCD[Y].dist correspond à la plus courte distance entre S et Y si Y appartient à $E_{noir\ i}$
    - Sinon PCD[Y].dist correspond à la distance du plus court chemin entre S et Y ne traversant que des nœuds de $E_{noir\ i}$
  - Pour connaître le plus court chemin entre le sommet de départ et chacun des nœuds, on stockera également le prédécesseur, dans le chemin, de chaque nœud (dans PCD[Y].pred)

- Après chaque sélection d'un nouveau nœud Gris n pour entrer dans l'ensemble E_{noir}, on effectue un relâchement des arcs issus de n, c'est-à-dire une mise à jour des distances aux sommets directement accessibles depuis n
- Relâchement d'un arc (n,x)
  - Si cet arc permet d'améliorer la longueur du chemin menant de la source S à x :
    - Si x était blanc alors coloriage en gris
    - Mise à jour de PCD[x].dist et de PCD[x].pred

Algorithme de Dijkstra

procédure Dijkstra(**données** G : Graphe, S : indice de Nœud,
résultat PCD : tableau [indice de Noeud]
de paire(distance, indice de Nœud))

variables

n_i,n_{min} : indice de Nœud
min : distance

début

//Initialisation

pour chaque nœud n_i de G **faire**

Initialisation avec couleur blanche

finpour

couleur(s)←gris

…

Le type paire (distance, indice de Nœud) est un type composé d'un champ dist et d'un champ pred

....

tant que il reste des nœuds gris **faire**

Recherche du prochain nœud gris à colorier en noir :

min$\leftarrow\infty$

pour tout nœud n_i gris **faire**

Mise à jour éventuelle de min et n_{min} (avec dist(n_i) et n_i) si n_i est plus proche de S que les nœuds gris précédemment observés

finpour

pour tout arc $n_{min}n_i$ avec n_i non noir **faire**

Relâchement des arêtes issues de n_{min} (pour colorier de nouveaux sommets en gris et mettre à jour les chemins aux voisins déjà gris)

finpour

couleur(n_{min})$\leftarrow$noir

fintantque

fin

Algorithme de Dijkstra

procédure Dijkstra(**données** G : Graphe, S : indice de Nœud,
résultat PCD : tableau [indice de Noeud]
de paire(distance, indice de Nœud))

variables

n_i,n_{min} : indice de Nœud
min : distance

début

//Initialisation

pour chaque nœud n_i de G **faire**

PCD[n_i].dist←G[S,n_i]

si G[S,n_i] = ∞ **alors** PCD[n_i].pred ← 0, couleur(n_i) ←blanc
sinon PCD[n_i].pred ← S, couleur (n_i) ←gris,
finsi

finpour

S.couleur←noir

…

Coût de l'arête entre S et n_i

Pas d'arête entre S et n_i

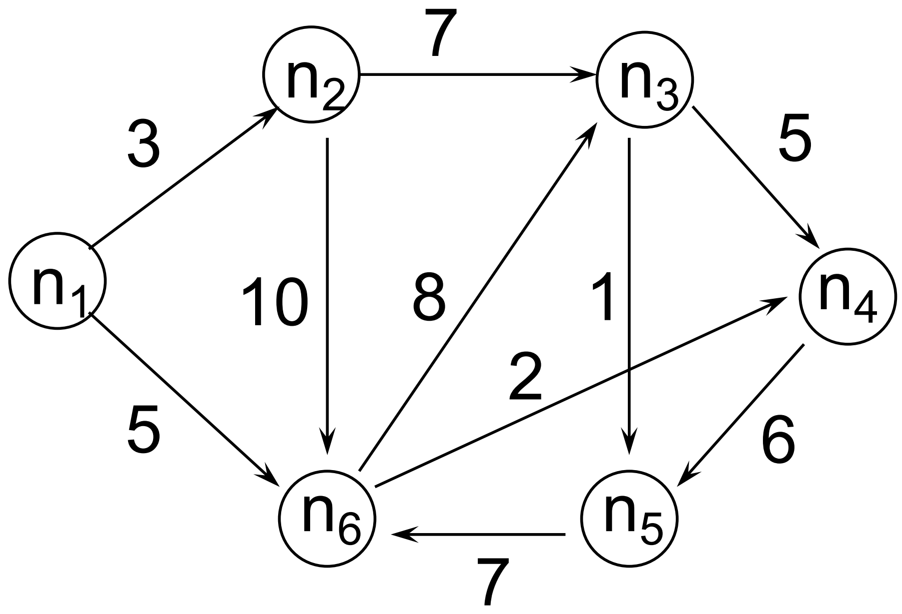
Le type paire (distance, Nœud) est un type composé d'un champ dist et d'un champ pred

```
….
tant que il reste des nœuds gris faire
    min←∞
    pour tout nœud n_i gris faire
        si PCD[n_i].dist < min alors
            min←PCD[n_i].dist, n_min←n_i
        finsi
    finpour
    pour tout arc n_min n_i avec n_i non noir faire
        si n_i.couleur=blanc alors
            n_i.couleur←gris
        finsi
        si PCD[n_min].dist + G[n_min,n_i] < PCD[n_i].dist alors
            PCD[n_i].dist ←PCD[n_min].dist + G[n_min,n_i]
            PCD[n_i].pred ← n_min
        finsi
    finpour
    couleur(n_min)←noir
fintantque
fin
```

Complexité améliorable en uilisant une file de priorité!

Sélection du Nœud à colorier en noir

Relâchement des arcs issus du nœud colorié en noir



Recherche des plus courts chemins entre n1 et les autres nœuds

Initialisation :

	n_1	n_2	n_3	n_4	n_5	n_6
dist	0	3	∞	∞	∞	5
pred	0	n_1	0	0	0	n_1

$E_{noir}=\{n1\}$

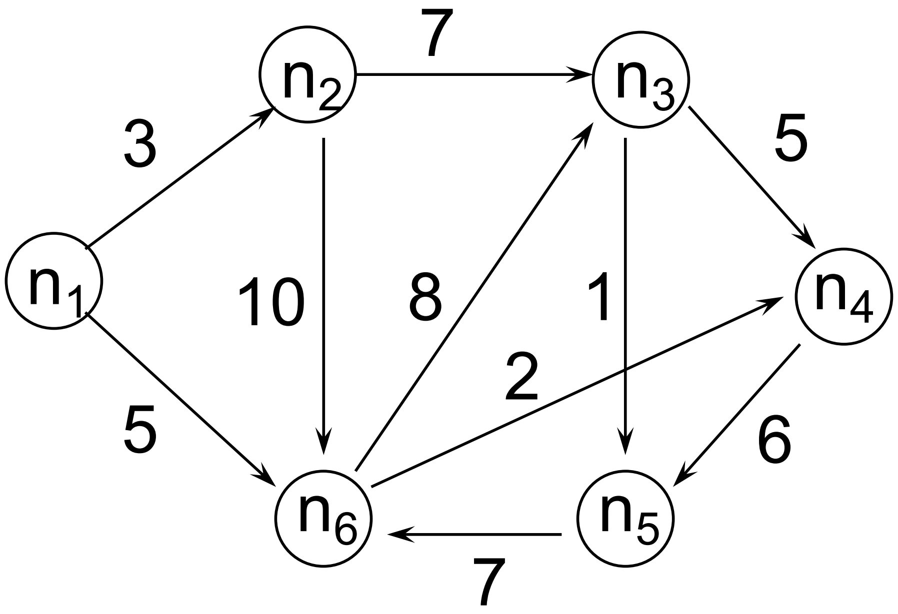
Etape 1

Sélection de n2

	n_1	n_2	n_3	n_4	n_5	n_6
dist	0	3	10	∞	∞	5
pred	0	n_1	n_2	0	0	n_1

$E_{noir}=\{n1,n2\}$

Relâchement de (n2,n3) et de (n2,n6)



Recherche des plus courts chemins entre n1 et les autres nœuds

Etape 2

Sélection de n6

	n_1	n_2	n_3	n_4	n_5	n_6
dist	0	3	10	7	∞	5
pred	0	n_1	n_2	n_6	0	n_1

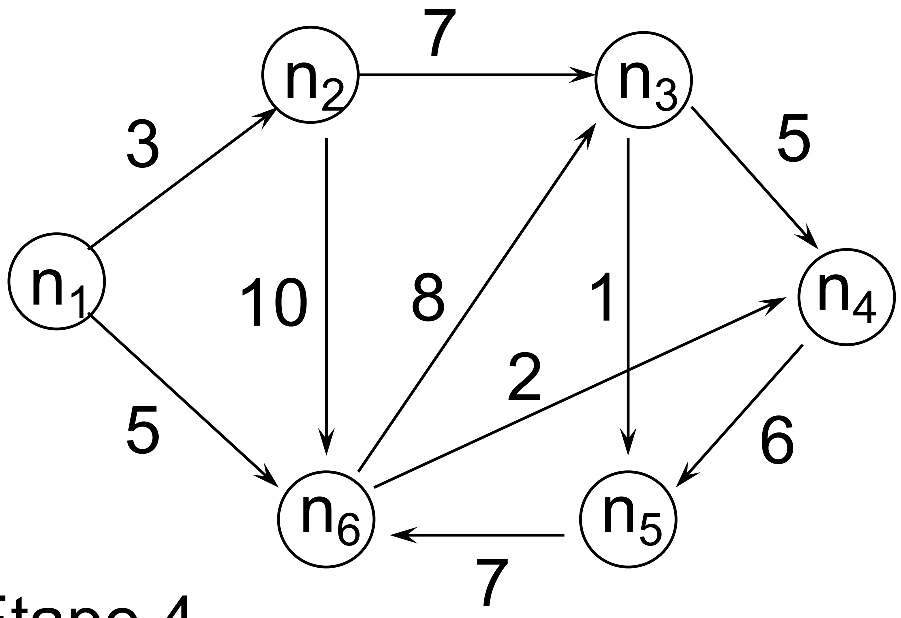
E={n1,n2,n6}
Relâchement de (n6,n3) et de (n6,n4)

Etape 3

Sélection de n4

	n_1	n_2	n_3	n_4	n_5	n_6
dist	0	3	10	7	13	5
pred	0	n_1	n_2	n_6	n_4	n_1

E={n1,n2,n6, n4}
Relâchement de (n4,n5)



Recherche des plus courts chemins entre n1 et les autres nœuds

Etape 4

Sélection de n3

	n_1	n_2	n_3	n_4	n_5	n_6
dist	0	3	10	7	11	5
pred	0	n_1	n_2	n_6	n_3	n_1

E={n1,n2,n6, n4, n3}

Relâchement de (n3,n5)

Sélection de n5

	n_1	n_2	n_3	n_4	n_5	n_6
dist	0	3	10	7	11	5
pred	0	n_1	n_2	n_6	n_3	n_1

E={n1,n2,n6, n4, n3, n5}

Plus court chemin entre n1 et n5?

Algorithme de Dijkstra

Si n est le nombre de nœuds dans le graphe et m le nombre d'arcs

- Initialisation : Parcours de n nœuds en θ(n)
- (au plus) n étapes successives
  - A chaque étape on cherche le nœud gris à colorier en noir parmi les n_g nœuds gris (n_g<=n-n_n)
    - En θ(n_g) si cet ensemble de nœuds est représenté par une liste chainée
    - En θ(n) si cet ensemble est représenté par un tableau de booléen
    - En θ(lg(n_g)) si cet ensemble est représenté par un tas binaire
  - A chaque étape, des opérations de relâchement (mise à jour de distance), suite au recrutement d'un sommet x dans E_{noir}
    - En θ(degré$^+$(x)) si représentation de G par des listes d'adjacence
    - En θ(n) si représentation de G par matrice d'adjacence

  auxquelles il faut ajouter des opérations de réajustement de tas binaire si jamais on décide d'en utiliser un!
    - En θ(lg(n_g))*(degré$^+$(x))

Algorithme de Dijkstra

- Algorithme
  - en $\theta(n^2)$ pour une représentation par matrice d'adjacence,
  - ou en $\theta(n(lg(n)+m+lg(n)*m))$ si on représente G, E_{noir} et l'ensemble E_{gris} des nœuds gris par des structures judicieuses!

Réseau de transport

- Définition
  - Graphe orienté et valué G=(N,A) ayant un unique nœud **s** sans prédécesseur (**entrée** ou **source** du réseau) et un unique sommet **t** sans successeur (**sortie** ou **puit** du réseau)
  - Notion de capacité d'un arc u de A
    - C_u : saturation, valeur maximale pour l'arc
    - Φ_u : capacité réelle, flot

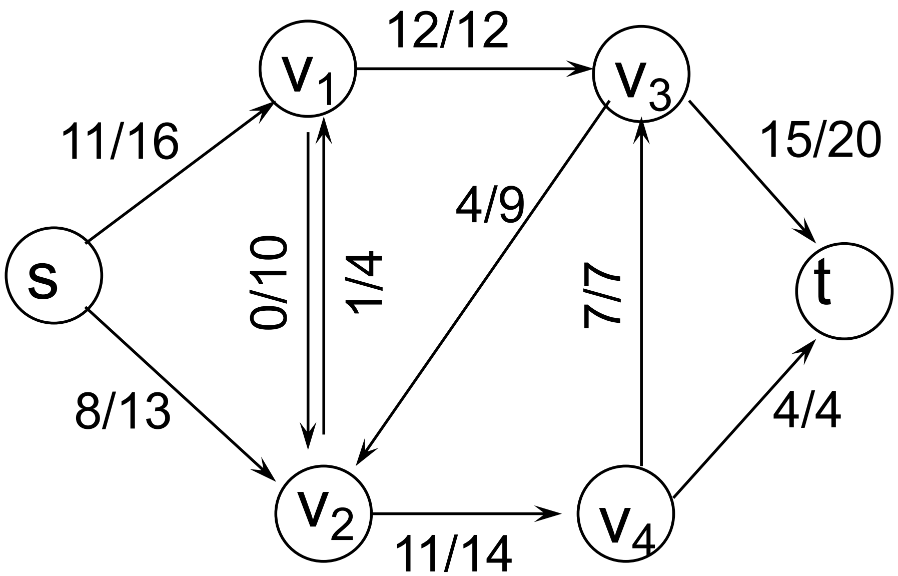
    On a $0 \leq \Phi_u \leq C_u$

- Lois de conservation aux nœuds
  - Pour tout nœud **x** différent de **s** ou **t**, le flux entrant doit correspondre au flux sortant

$$\sum_{y \in succ(x)} \phi_{(x,y)} = \sum_{z \in pred(x)} \phi_{(z,x)}$$

  - En ce qui concerne **s** et **t**, Φ_0 est la valeur du flot circulant entre **s** et **t** dans G

$$\phi_0 = \sum_{y \in succ(s)} \phi_{(s,y)} = \sum_{z \in pred(t)} \phi_{(z,t)}$$



- La loi de conservation aux nœuds est-elle vérifiée dans ce réseau de transport?

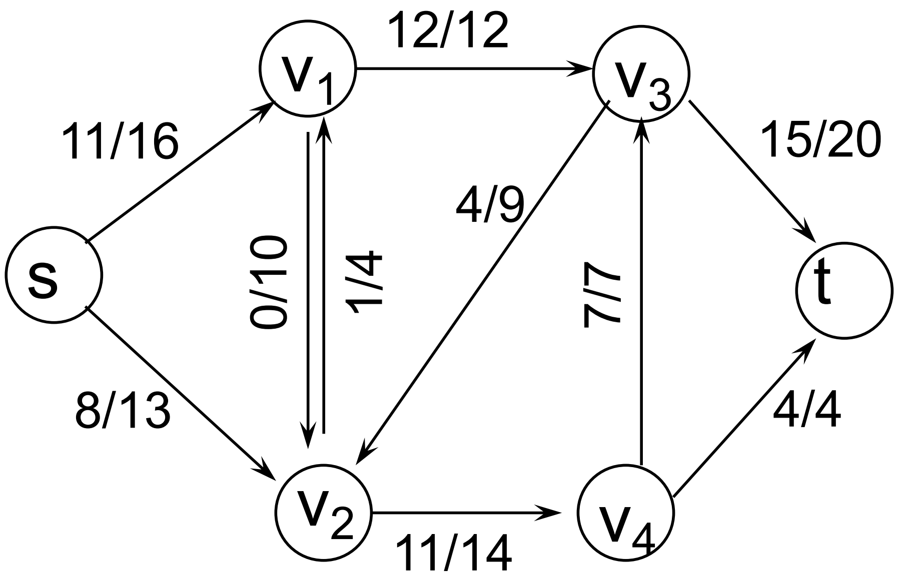
- Quelle est la valeur totale du flot circulant entre **s** et **t**?

Ford-Fulkerson

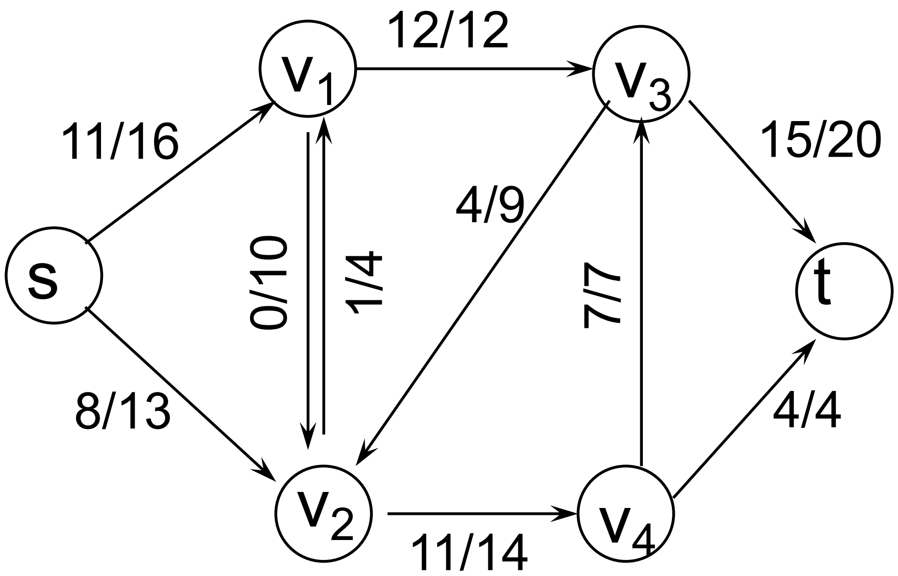
- L'algorithme de Ford-Fulkerson permet de connaître le flot maximal supportable par un réseau de transport
- En partant d'une valeur nulle, l'algorithme consiste à augmenter la valeur du flot de manière itérative tout en respectant la capacité de l'ensemble des arcs

Ford Fulkerson

- A chaque itération :
  - On cherche un chemin non saturé de G reliant **s** à **t** le long duquel on pourrait augmenter le flot (chemin dit « améliorant »)
  - Pour cela :
    - On considère le graphe résiduel G' à partir de l'état actuel des flots dans G
    - Un chemin non saturé dans G correspond à un chemin dans G'
  - Si un tel chemin existe, on augmente le flot de la quantité correspondante
- Attention :
  - Pour augmenter la valeur du flot circulant entre **s** et **t** on pourra être amené à diminuer le flot dans certains arcs ☺



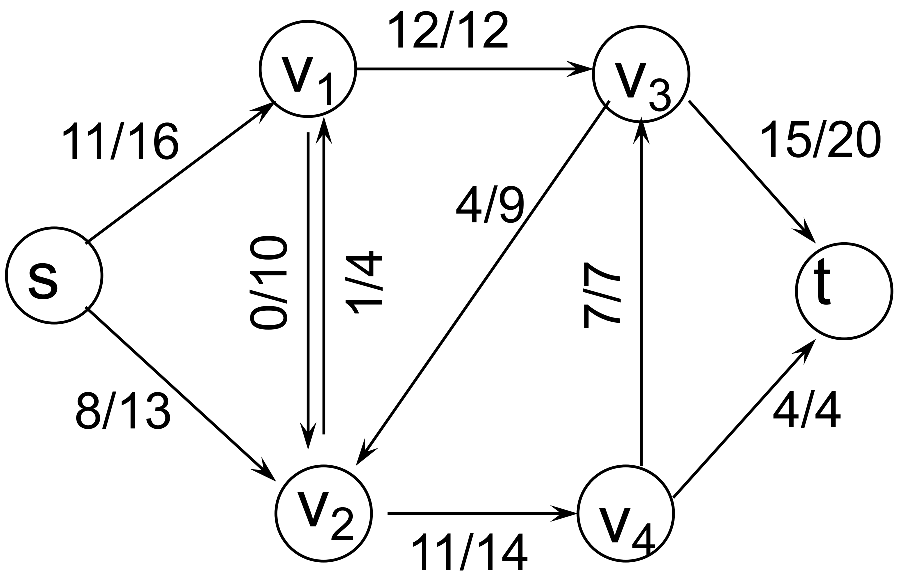
- Testez si le flot est optimal en recherchant un chemin améliorant (s’il existe)



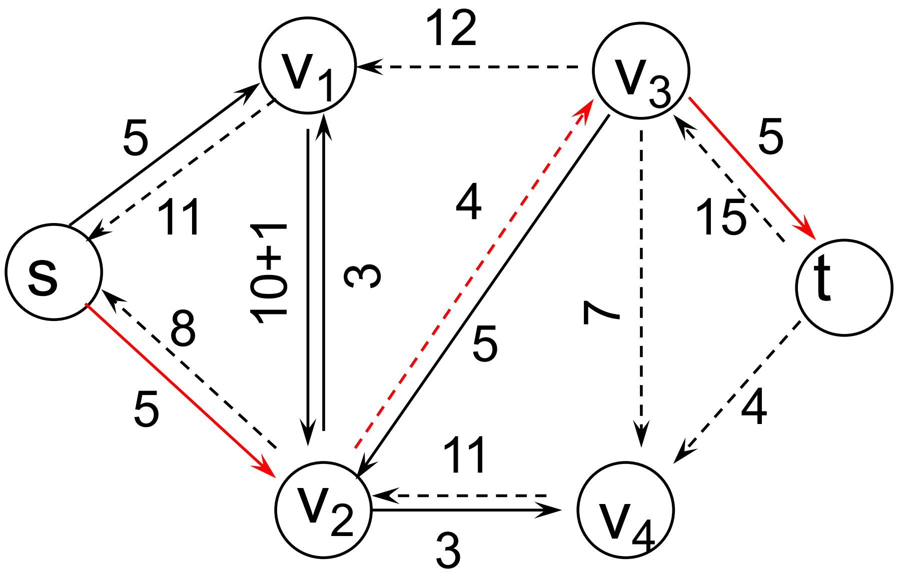
Etant donné un réseau de transport G, les arcs du graphe résiduel G' permettent d'identifier:

• les arcs de G dans lesquels il est encore possible d'augmenter le flux (arcs orientés dans le même sens dans G et G')

• les arcs de G dans lesquels il est possible de décrémenter le flux (arcs orientés en sens inverse dans G et G')



- Réseau résiduel G’



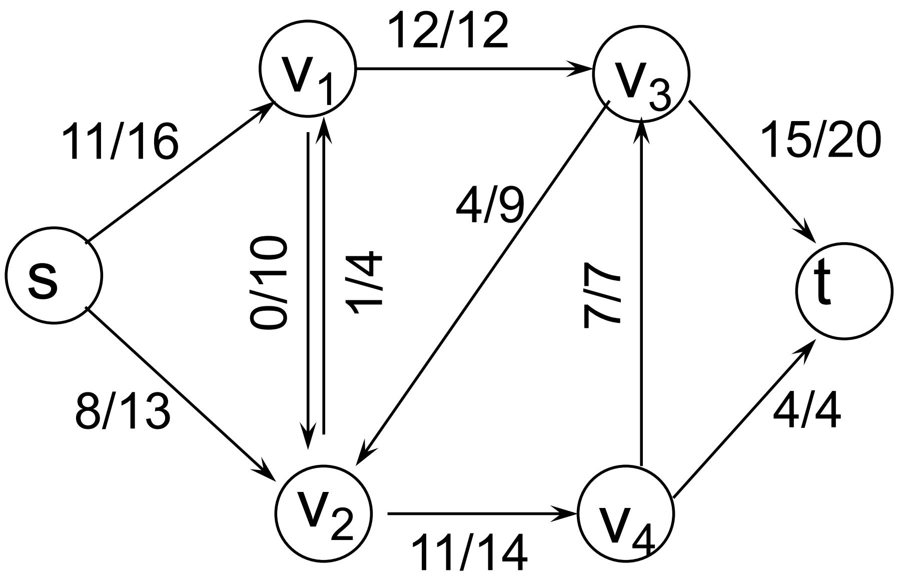
On a trouvé un chemin améliorant s, v2, v3, t.

- Recherche d’un chemin améliorant sans calculer l’ensemble du graphe résiduel
  - Initialisation :
    - au début, tous les sommets de **G** sont non marqués (blancs)
    - Les nœuds de **G** sont les nœuds de **G’**
  - On marque (en gris) l’entrée **s** du réseau de **G**
  - Tant que la sortie **t** est non marquée (blanche) et qu’il reste des sommets marqués non examinés (ie des sommets gris)
    - On examine (avant passage au noir) un sommet **x** marqué mais non examiné
      - On marque (en gris) **tous les successeurs** **y** de **x** correspondant à des arcs non saturés
        L’arc **(x,y)** est un arc de **G’** et on lui associe la valeur $C_{(x,y)} - \Phi_{(x,y)}$
      - On marque **tous les prédécesseurs** **z** de **x** correspondant à des arcs portant un flux strictement positif
        L’arc **(x,z)** est un arc de **G’** et on lui associe la valeur $\Phi_{(z,x)}$

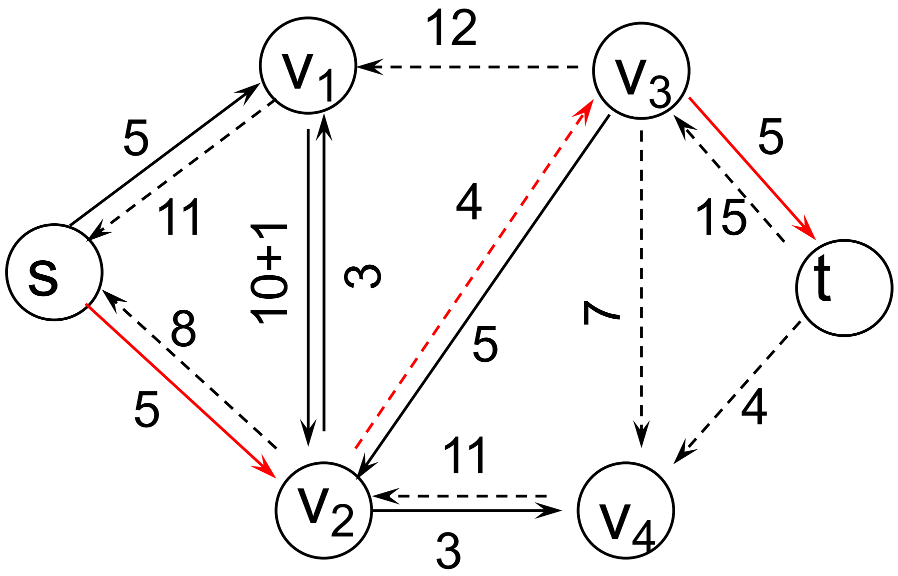
- Si on réussit à marquer **t**, alors on a trouvé un chemin améliorant :
  - Ce chemin améliorant, ainsi que sa capacité, apparaissent dans le graphe résiduel **G’** en cours de construction

    Capacité **ε**= min(valeurs des arcs du chemin améliorant)
- Attention :
  - Certains arcs du chemin améliorant correspondent à des arcs de **G** (appelons $\mathbf{L^+}$ leur ensemble)
  - Certains arcs du chemin améliorant correspondent à des arcs de **G** pris dans le sens inverse (appelons $\mathbf{L^-}$ leur ensemble)

- Il est alors possible d’augmenter de **ε** la valeur du flot dans le réseau en augmentant de **ε** la valeur des arcs de $\mathbf{L^+}$ et en diminuant de **ε** la valeur des arcs de $\mathbf{L^-}$



- Réseau résiduel G'



On a trouvé un chemin améliorant s, v2, v3, t.
La capacité de ce chemin est 4.
Dans le graphe G, on améliore donc le flot en affectant :

- 8+4=12 à l'arc (s,v2)
- 4-4 =0 à l'arc (v3,v2)
- 15+4=19 à l'arc (v3,t)

- Méthode de Ford-Fulkerson
  - Initialiser le flot Φ de tous les arcs à 0
  - Tant qu'il existe un chemin améliorant p de capacité ε, modifier les arcs correspondant dans G
  - Retourner Φ_0