Chapitre 7
Le problème du flot maximum

Claude-Guy Quimper

Introduction

- Le problème du flot maximum est un problème combinatoire sur des graphes.
- Plusieurs problèmes notamment dans les transports, la distribution de la marchandise et la conception de réseaux robustes se résolvent à partir d'un flot maximum dans un graphe.
- Plusieurs algorithmes de filtrage dont celui de la contrainte AllDifferent utilisent un flot pour filtrer les domaines des variables.

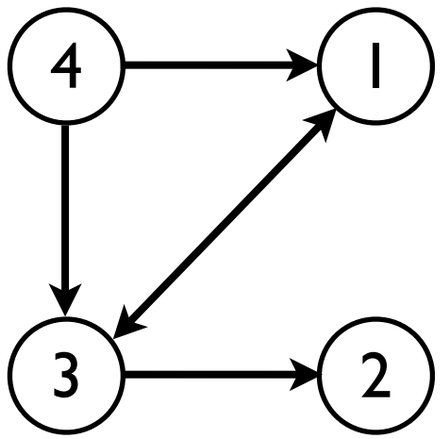
Introduction

- Ce chapitre est divisé en quatre sections.
  - Tout d'abord, nous allons nous donner les outils pour concevoir des algorithmes de graphes. Nous présenterons donc une structure de données pour encoder un graphe et présenterons un algorithme de fouille en profondeur sur un graphe.
  - Nous présenterons ensuite le problème du flot maximum et montrerons comment le résoudre.
  - Nous étudierons des problèmes pouvant être modélisés avec un flot maximum dans un graphe.
  - Finalement, nous présenterons comment le flot maximum peut filtrer la contrainte All-Different.

Les graphes orientés

- Un graphe orienté est un tuple (V, E) où V est l'ensemble des sommets (aussi appelés noeuds) et E ⊆ V x V est l'ensemble des arêtes.
- Une paire (a, b) ∈ E représente une arête orientée de a vers b.

Exemple de graphe orienté

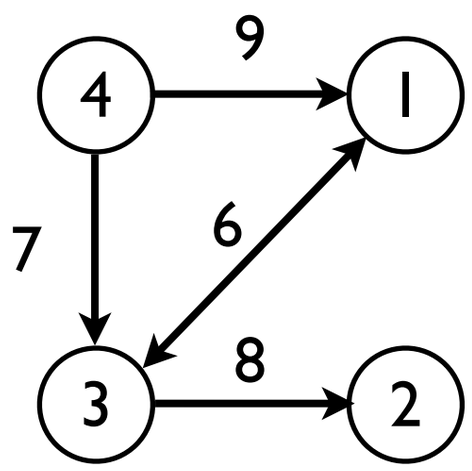


V = {1, 2, 3, 4}
E = {(1, 3), (3, 1), (3, 2), (4, 1), (4, 3)}

Les graphes orientés

- Un noeud *a* est **adjacent** au noeud *b* s'il existe une arête (a, b) ∈ E.
- Certains graphes ont des poids sur leurs arêtes. Ces poids sont donnés par une fonction *f*(*a*, *b*) qui prend deux noeuds en entrée et qui retourne un nombre (entier ou réel selon le contexte).
- Les poids peuvent entre autres servir à indiquer des distances, des coûts ou des capacités.

Exemple de fonction de poids



a	b	f(a,b)
1	3	6
3	1	6
3	2	8
4	1	9
4	3	7

Les graphes orientés

- Un **chemin** est une suite de noeuds distincts $n_0, n_1, \ldots n_{k-1}$ dont (n_i, n_{i+1}) est une arête pour tout $0 \leqslant i < k-1$. Le noeud n_0 est **l'origine** et le noeud n_{k-1} est la **destination**.

- Un **cycle** est une suite de noeuds distincts $n_0, n_1, \ldots, n_{k-1}$ et dont $(n_i, n_{i+1 \bmod k})$ est une arête pour tout $0 \leqslant i < k$.

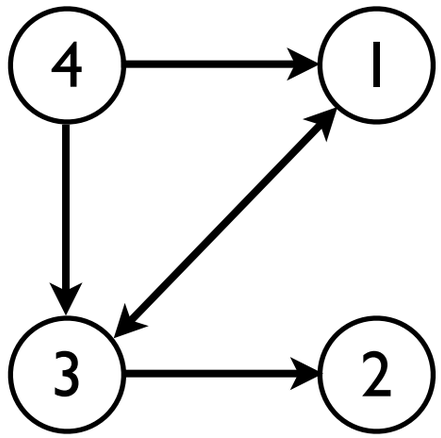
- Selon le contexte, les chemins ou les cycles sont parfois identifiés par l'ensemble des arêtes qui les composent plutôt que l'ensemble des sommets.

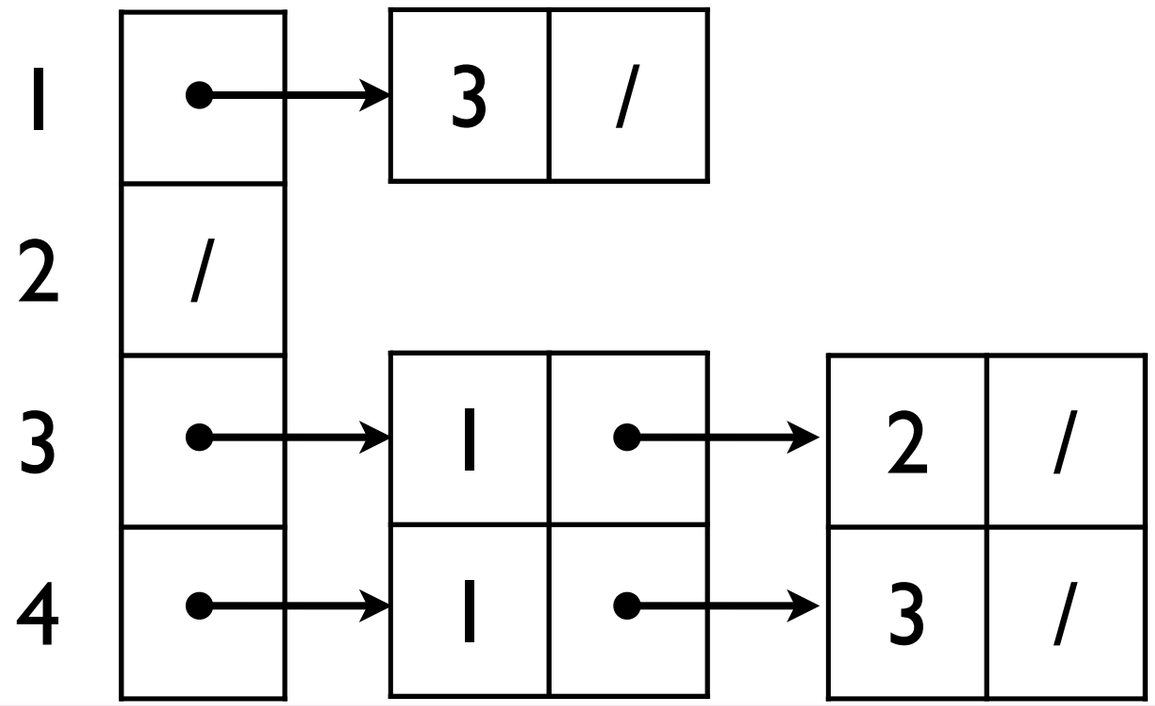
Structure de données

- Il existe différentes structures de données pour encoder un graphe. La plus simple étant probablement une matrice de proximité M où M[a,b] = 1 si (a,b) est une arête et M[a,b] = 0 si (a, b) n'est pas une arête.
- Nous utiliserons cependant une autre structure de données mieux adaptée à nos besoins.

Les listes de proximités

- On peut représenter un graphe en utilisant des listes de proximités.
- Cette structure de données possède un tableau Adj[1..n] avec une entrée par noeud. Une entrée A[i] de ce tableau pointe sur le début d'une liste chaînée contenant les noeuds adjacents au noeud i.
- Cette liste peut être doublement chaînée pour rendre plus efficaces les modifications apportées au graphe.





Fouille en profondeur

- Nous avons déjà vu au chapitre 2 comment se fait une fouille en profondeur dans un arbre. Nous voyons à présent comment faire une telle fouille dans un graphe orienté.

- La fouille en profondeur est au graphe ce que le couteau suisse est au bricolage. Elle possède plusieurs propriétés pouvant être exploitées dans la conception d'algorithmes.

Fouille en profondeur

Algorithme 1 : Fouille-en-profondeur(V, E)

```
// Initialisation
Crée la variable globale Couleur[1..|V|]
Crée la variable globale Parent[1..|V|]
pour tous les noeuds u ∈ V faire
    Couleur[u] ← blanc
    Parent[u] ← Nul
// Démarrage de la fouille
pour tous les noeuds u ∈ V faire
    si Couleur[u] = blanc alors
        Visite(u, V, E)
```

Algorithme 2 : Visite(u, V, E)

```
Couleur[u] ← Gris
pour v ∈ Adj[u] faire
    // Exploration de l'arête (u, v)
    si Couleur[v] = blanc alors
        Parent[v] ← u
        Visite(v, V, E)
Couleur[u] ← noir
```

- L'algorithme *Fouille-en-profondeur* initialise les vecteurs *Couleur* et *Parent* et démarre ensuite la visite des noeuds.
- L'algorithme *Visite* visite un noeud du graphe et toutes les arêtes adjacentes à ce noeud. Si le noeud *u* est adjacent à un *v* qui n'a pas été visité, alors il le visite.

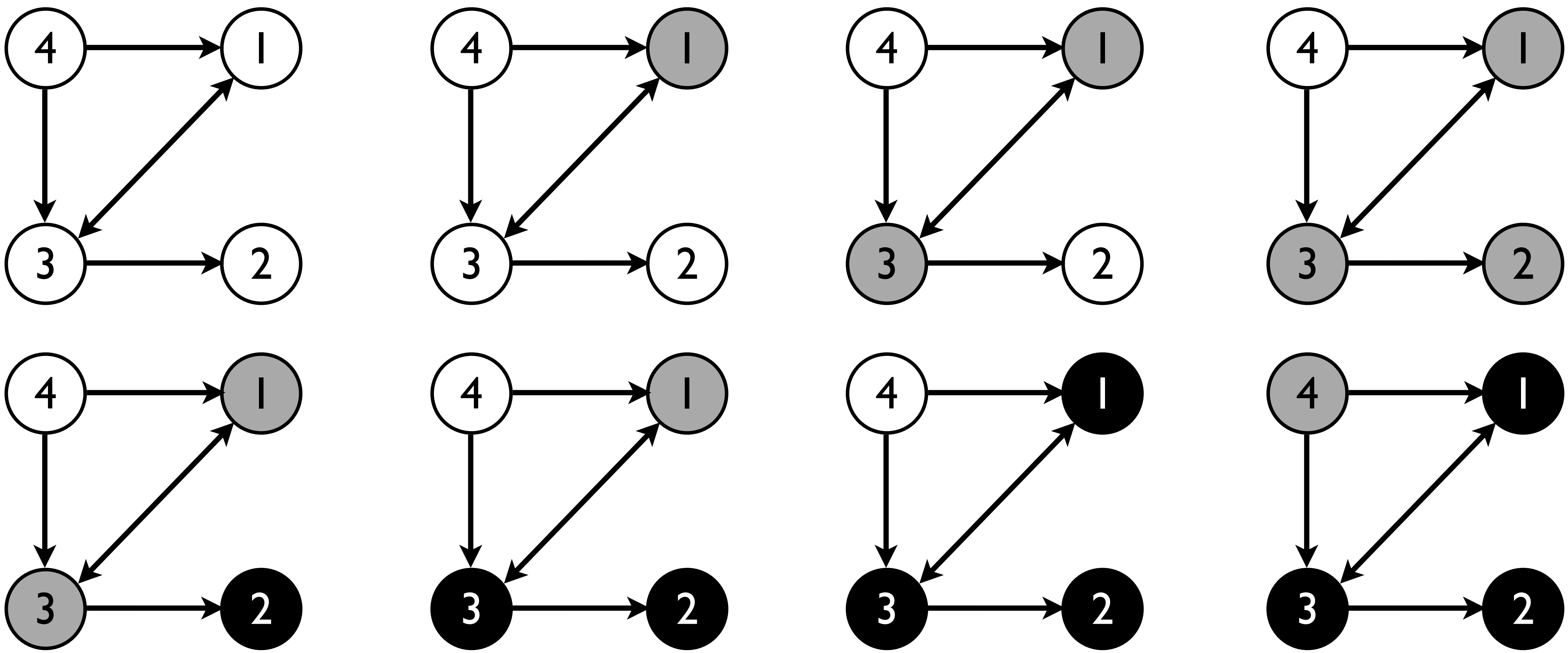
Code de couleurs

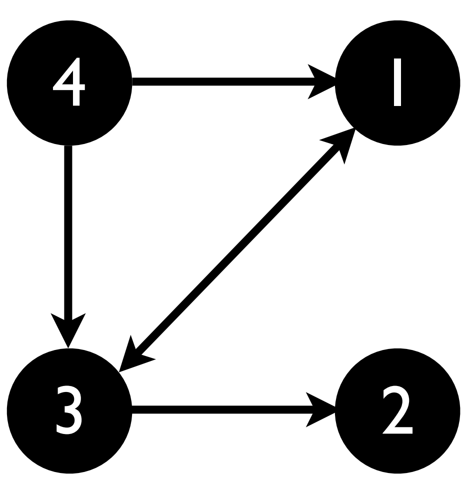
Couleur	Signification
Blanc	Le noeud n'a pas été visité
Gris	Le noeud a été visité, mais ses arêtes adjacentes n'ont pas toutes été traitées.
Noir	Le noeud a été visité et toutes ses arêtes adjacentes ont été traitées.

Le vecteur Parent

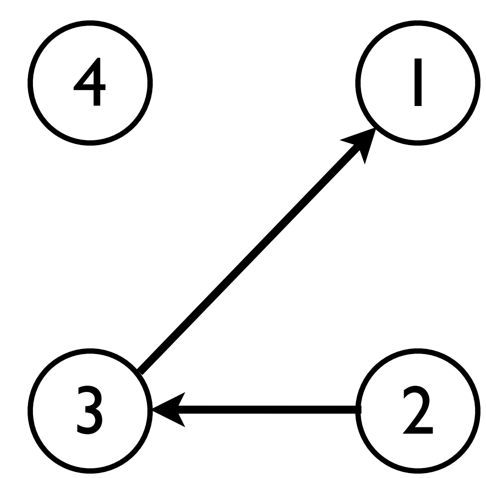
- Le vecteur Parent encode une forêt, c'est-à-dire une collection d'arbres.
- Tous les noeuds du graphe font partie de la forêt.
- Une arête (v, u) fait partie de la forêt si le noeud v a été visité lors du traitement de l'arête (u, v).
- On dit alors que u est le parent de v.
- La structure de données encode cette information sous la forme du tableau Parent où Parent[v] = u.

Exemple d’exécution





État du vecteur Parent
à la fin de la fouille:



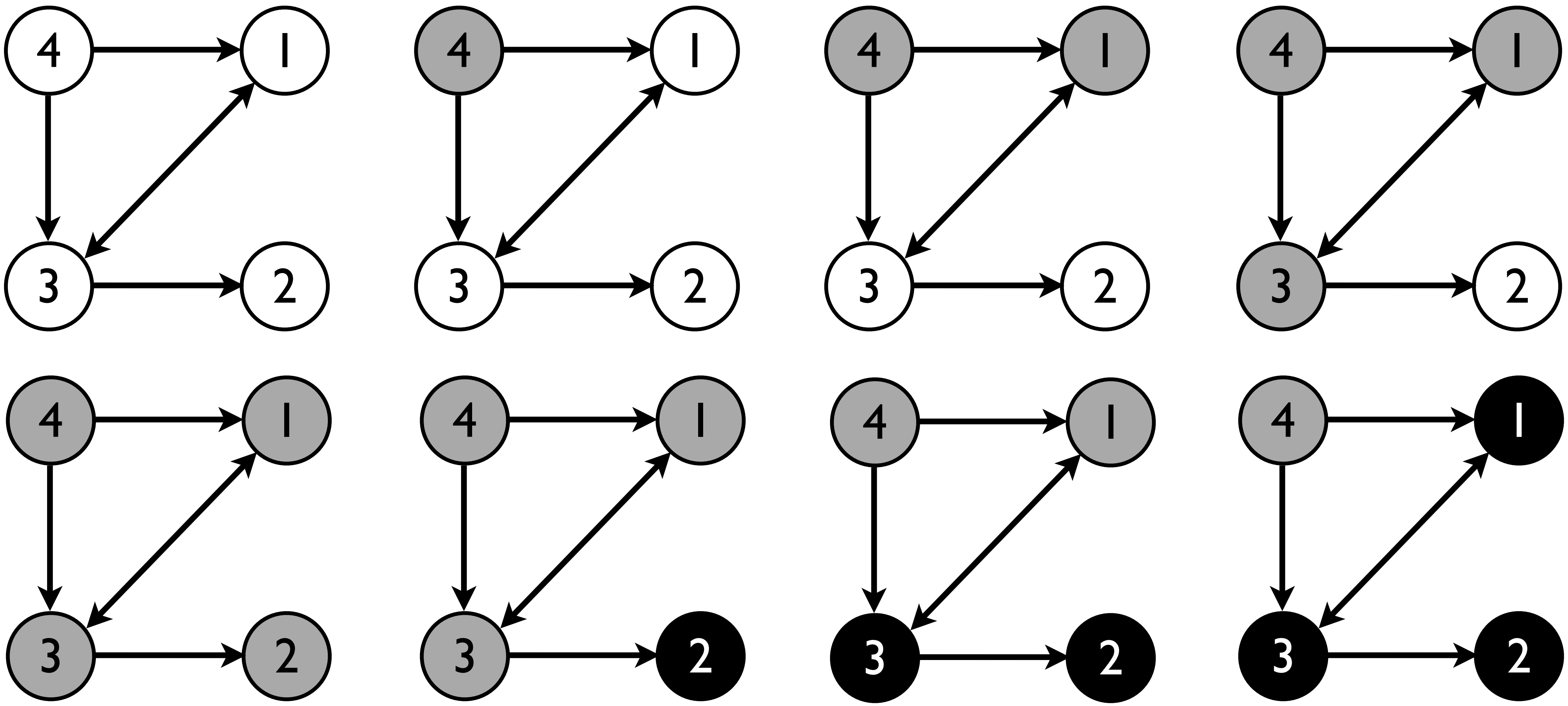
Complexité

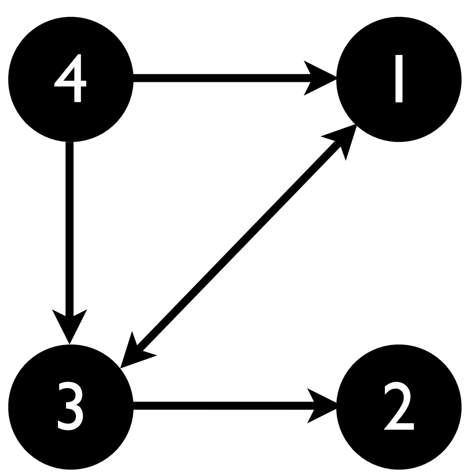
- L'initialisation se fait en O(|V|) étapes.
- Il y a exactement |V| appels à la procédure Visite.
- Un appel à la procédure Visite traite chaque arête adjacente au noeud *u* une seule fois. Puisque la procédure Visite n'est appelée qu'une seule fois par noeud, chaque arête est donc traitée qu'une seule fois.
- Au total, l'algorithme s'exécute donc en temps O(|V| + |E|).

Trouver un chemin

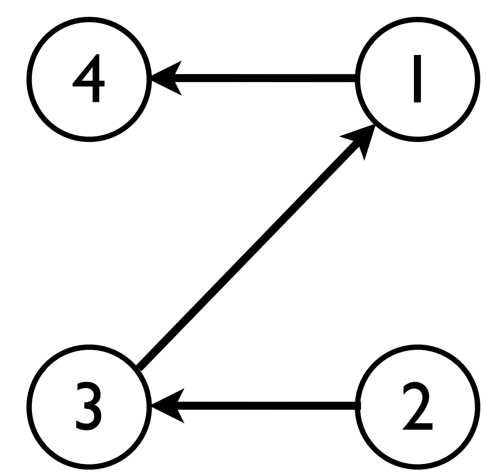
- **Problème:** Trouver un chemin reliant l'origine *s* à la destination *t*.
- **Solution:** L'algorithme de fouille en profondeur peut être adapté à cette fin.
  - Dans l'algorithme Fouille-en-profondeur, au lieu d'appeler la procédure Visite sur tous les noeuds, on ne fait que l'appeler sur le noeud d'origine *s*.
  - Les noeuds t, Parent[t], Parent[Parent[t]], ..., s forme le chemin de l'origine *s* à la destination *t* en sens inverse.
  - Si Parent[t] est nul, alors il n'y a pas de chemin allant de *s* vers *t*.

Exemple



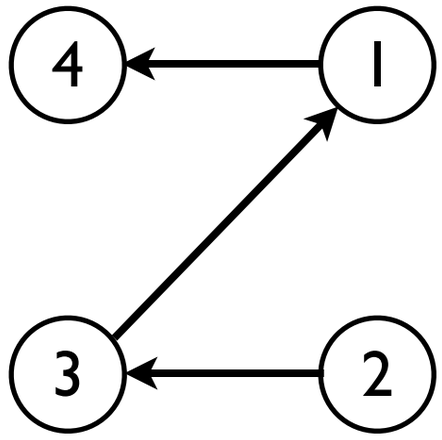


État du vecteur Parent
à la fin de la fouille:



Exemple

État du vecteur Parent
à la fin de la fouille:



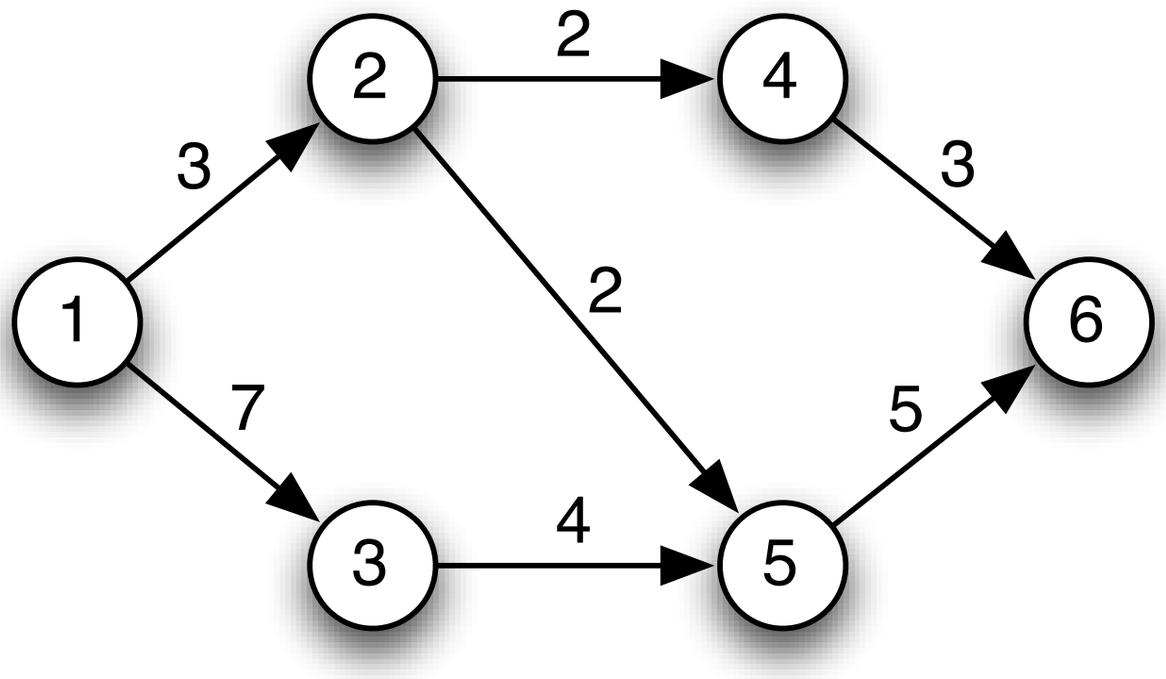
- Pour trouver un chemin du noeud 4 au noeud 3, on trouve la séquence 3, Parent[3] = 1, Parent[1] = 4. La séquence 3, 1, 4 est le chemin de 4 à 3 en ordre inverse.

Trouver un chemin

- Remarquez que cet algorithme trouve le chemin du noeud d'origine *s* vers tous les autres noeuds du graphe et pas seulement pour la destination *t*.
- Il est donc possible, dans un premier temps, de créer l'arbre généré par la fouille en profondeur et ensuite, de trouver les chemins du noeud *s* allant vers plusieurs autres noeuds du graphe.

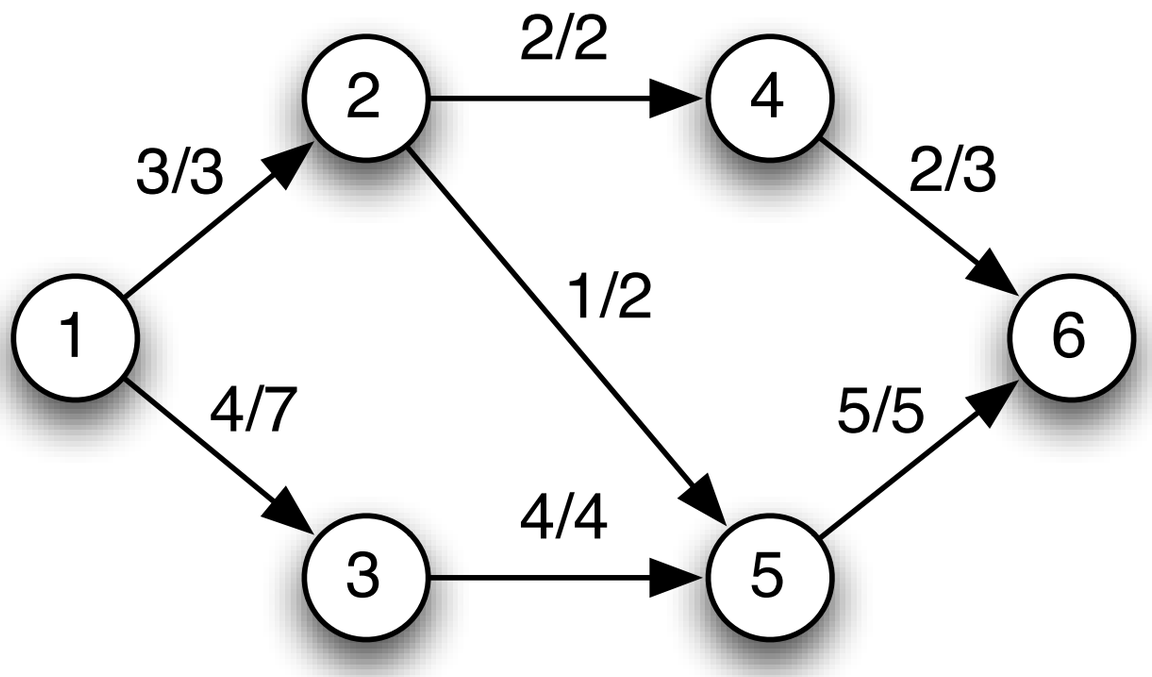
Problème du flot maximum

- Considérez le graphe suivant et imaginez que les arêtes sont des tuyaux et que les noeuds sont des joints. Chaque tuyau à une capacité indiquant le nombre de litres d'eau pouvant circuler à la minute.
- **Question:** Combien de litres d'eau à la minute peuvent circuler du noeud 1 au noeud 6?



Problème du flot maximum

- **Réponse:** 7 litres d'eau par minute.
- Le graphe suivant montre sur chaque arête la quantité d'eau qui s'écoule par rapport à la capacité de l'arête.
- Notez que rien ne se perd, rien ne se crée. La quantité d'eau entrant dans un noeud doit être la même qui en sort.



Problème du flot maximum

- Le problème du flot maximum est un problème d'optimisation qui peut être résolu en temps polynomial.
- Nous allons donc présenter un algorithme qui peut le résoudre efficacement.
- Ce problème peut modéliser plusieurs problèmes complexes (autre que l'écoulement de l'eau dans des tuyaux).
- Il peut également être utilisé comme sous-routine dans la résolution de problème NP-Difficile.

Définition du problème

- Le noeud *s* d'où provient l'eau s'appelle **la source**.
- Le noeud *t* où se rend l'eau s'appelle **le puits**.
- La **capacité** entre deux noeuds a et b est dénoté par c(a,b). Cette valeur est positive si (a, b) est une arête et nulle si (a, b) n'est pas une arête.
- Un **flot** est un vecteur f dont chaque composante est associée à une paire de noeuds. On note par f(a,b) la quantité de flot circulant entre le noeud *a* et le noeud *b*. Nous avons la relation suivante: $f(a,b) = -f(b,a)$
- Une instance du problème de flot maximum est caractérisée par le graphe G = (V, E), la source *s*, le puits *t* et le vecteur de capacité *c*.

Définition du problème

- La **contrainte de conservation du flot** requiert que la quantité d'eau entrant dans un noeud autre que la source et le puits soit la même quantité qui sort de ce noeud.

$$\sum_{b|(b,a)\in E} f(b,a) = \sum_{b|(a,b)\in E} f(a,b) \qquad \forall a \in V \setminus \{s,t\}$$

- De façon équivalente, nous avons cette relation.

$$\sum_{b\in V} f(a,b) = 0 \qquad \forall a \in V \setminus \{s,t\}$$

- La **contrainte de capacité** requiert que la quantité circulant sur une arête ne dépasse pas la capacité de cette arête.

$$f(a,b) \leq c(a,b) \qquad \forall (a,b) \in E$$

$$f(a,b) \leq 0 \qquad \forall (a,b) \notin E$$

Définition du problème

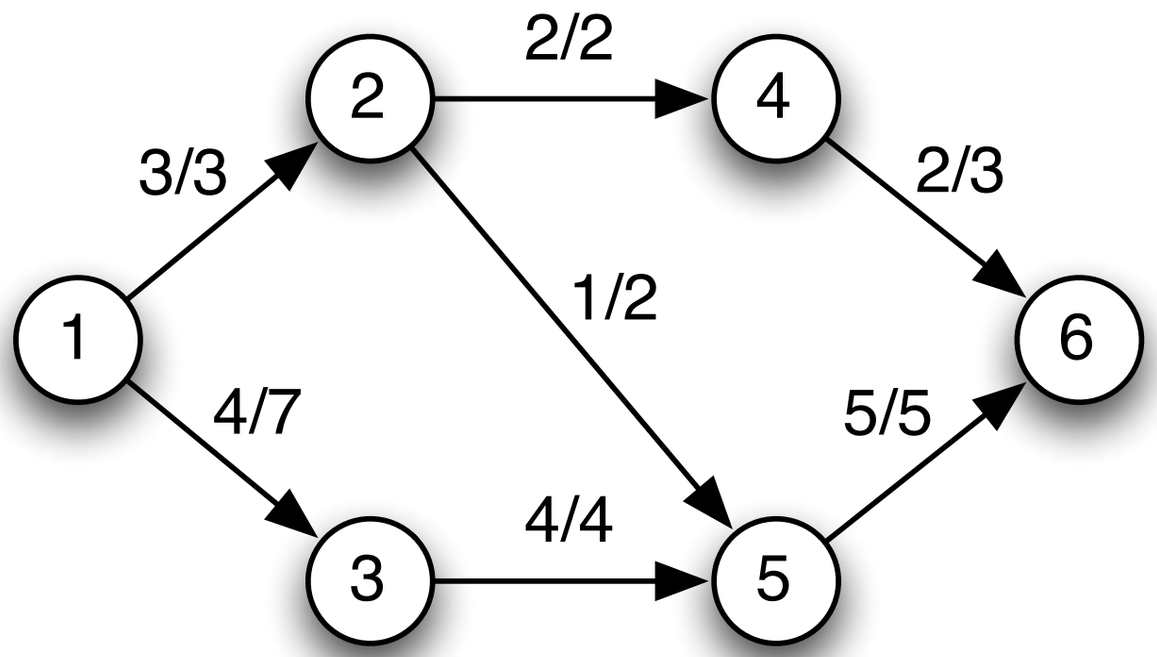
- Un **flot valide** est un flot qui satisfait la contrainte de conservation du flot et la contrainte de capacité.
- Le **flot nul** est un flot valide où f(a,b) = 0 pour toutes les arêtes.
- La **valeur d'un flot valide** est la quantité d'eau sortant de la source. Par la contrainte de conservation du flot, cette valeur est aussi égale à la quantité d'eau entrant dans le puits.

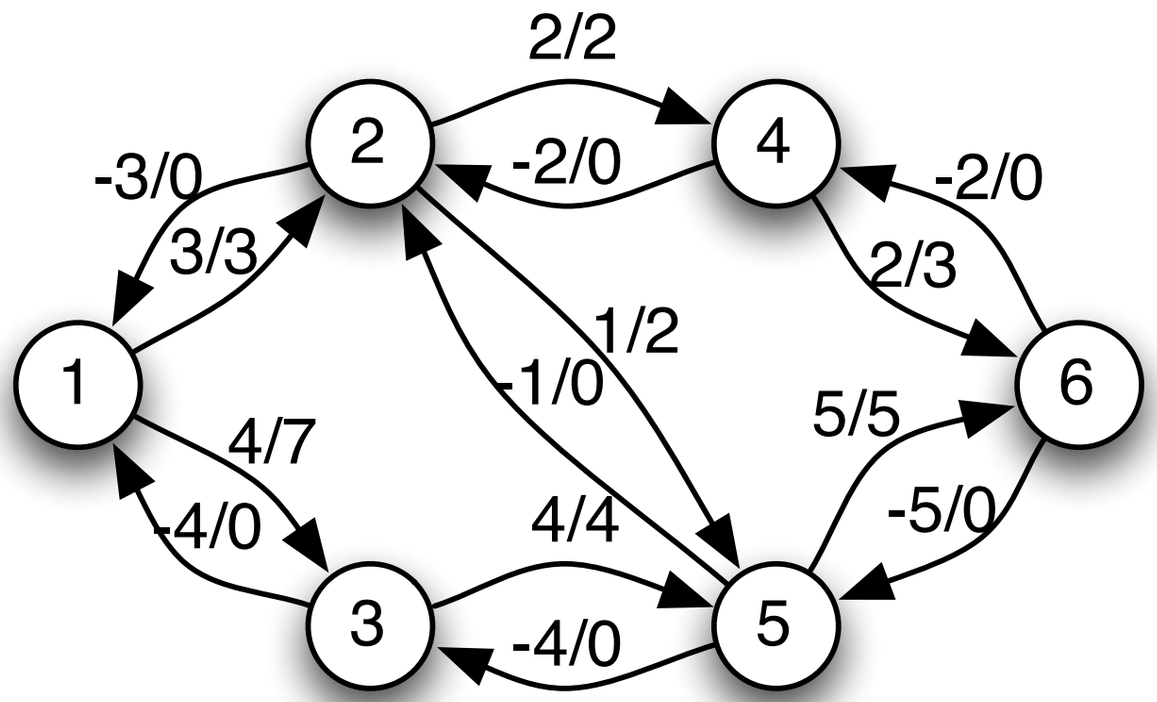
$$v(f) = \sum_{a \in V} f(s,a) = \sum_{a \in V} f(a,t)$$

- Le **problème du flot maximum** consiste à trouver un flot valide dont la valeur est maximale.

Représentation graphique

- Généralement, nous représentons seulement les arêtes dont la capacité est positive et les flots qui ne sont pas négatifs. Ainsi, ces deux représentations graphiques sont équivalentes.





L'algorithme de Ford-Fulkerson

- L'algorithme de Ford-Fulkerson est un algorithme itératif.
- À chaque itération, l'algorithme prend un flot valide et le transforme en un nouveau flot valide. Ce nouveau flot a une valeur de flot strictement plus grande ou alors, l'algorithme prouve qu'il n'existe pas de flot dont la valeur est plus grande.
- Cette augmentation se fait en poussant une quantité d'eau le long d'un chemin qu'on appelle chemin augmentant.
- Ces chemins apparaissent dans un autre graphe appelé graphe résiduel.

Le graphe résiduel

- Le graphe résiduel $G_f = (V, E_f)$ est utilisé pour trouver les chemins reliant la source au puits sur lesquels il est possible de pousser une quantité additionnelle d'eau et ainsi augmenter la valeur du flot f.
- Ce graphe est constitué des mêmes noeuds que le graphe original $G = (V, E)$ cependant, les arêtes peuvent avoir une orientation différente et un poids différent.

Le graphe résiduel

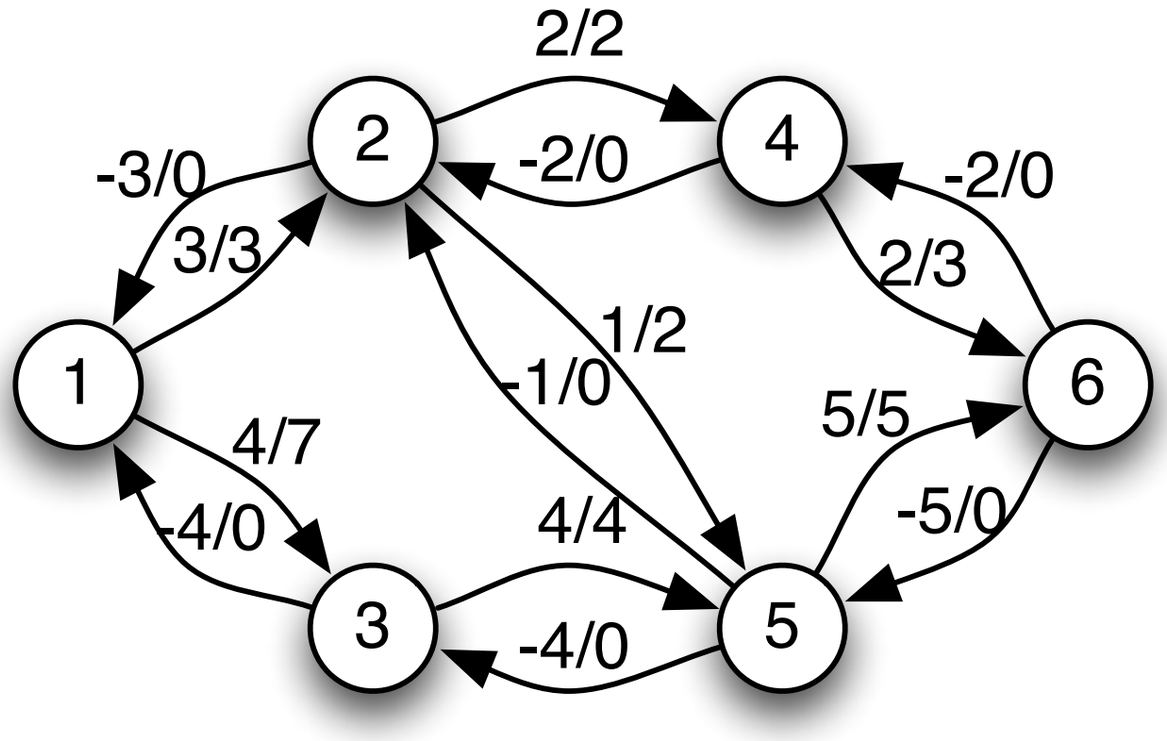
- Nous avons une arête (a, b) dans le graphe résiduel si et seulement si l'arête (a, b) n'est pas saturée.

$$(a,b) \in E_f \iff f(a,b) < c(a,b)$$

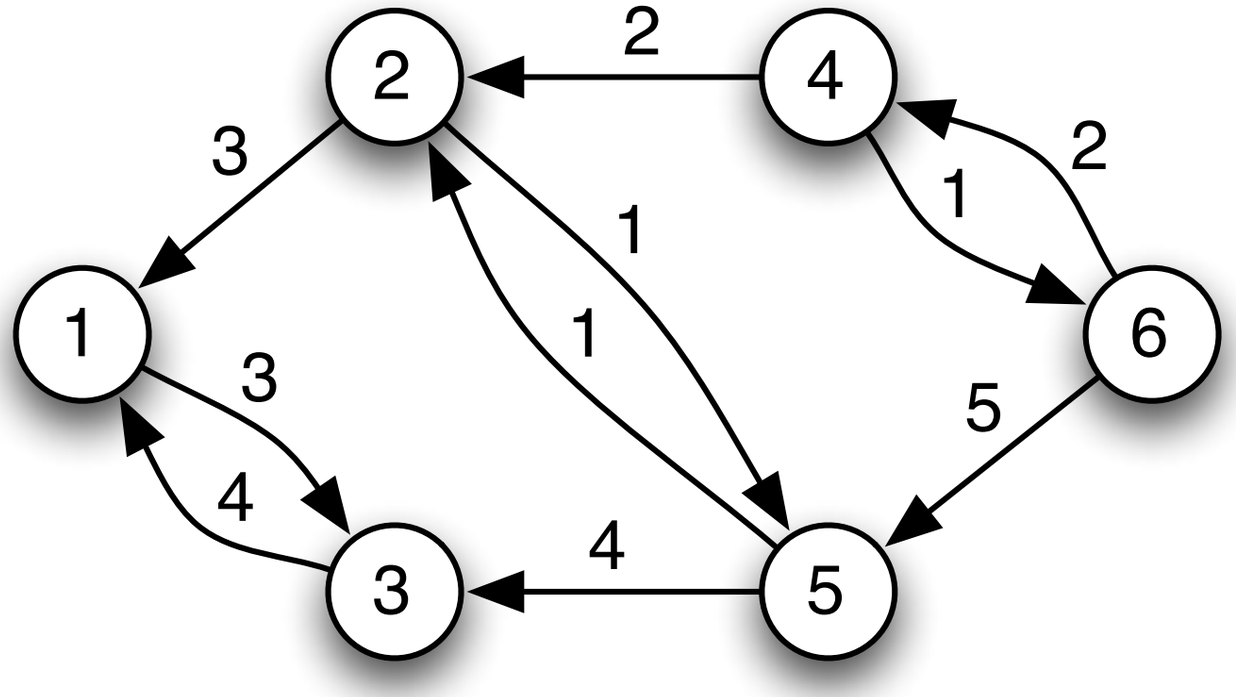
- La capacité résiduelle est donnée par la quantité de flot pouvant être ajoutée sur une arête sans violer la capacité de cette arête.

$$c_f(a,b) = c(a,b) - f(a,b)$$

Exemple de graphe résiduel



Graphe G avec flot f



Graphe résiduel G_f

Chemin augmentant

- Un chemin reliant la source *s* au puits *t* dans le graphe résiduel G_f est appelé **chemin augmentant**.
- C'est un chemin sur lequel il est possible de « pousser » au moins une unité de flot sur les arêtes, c'est-à-dire augmenter d'une unité la quantité de flot sur chaque arête du chemin sans dépasser les capacités des arêtes.
- **Théorème**: Considérons un chemin augmentant et soit q la plus petite capacité résiduelle $c_f(a,b)$ associée à une arête de ce chemin. Le flot f' est un flot valide de valeur v(f) + q.

$$f'(a,b) = \begin{cases} f(a,b) + q & \text{si } (a,b) \text{ est sur le chemin augmentant} \\ f(a,b) - q & \text{si } (b,a) \text{ est sur le chemin augmentant} \\ f(a,b) & \text{sinon} \end{cases}$$

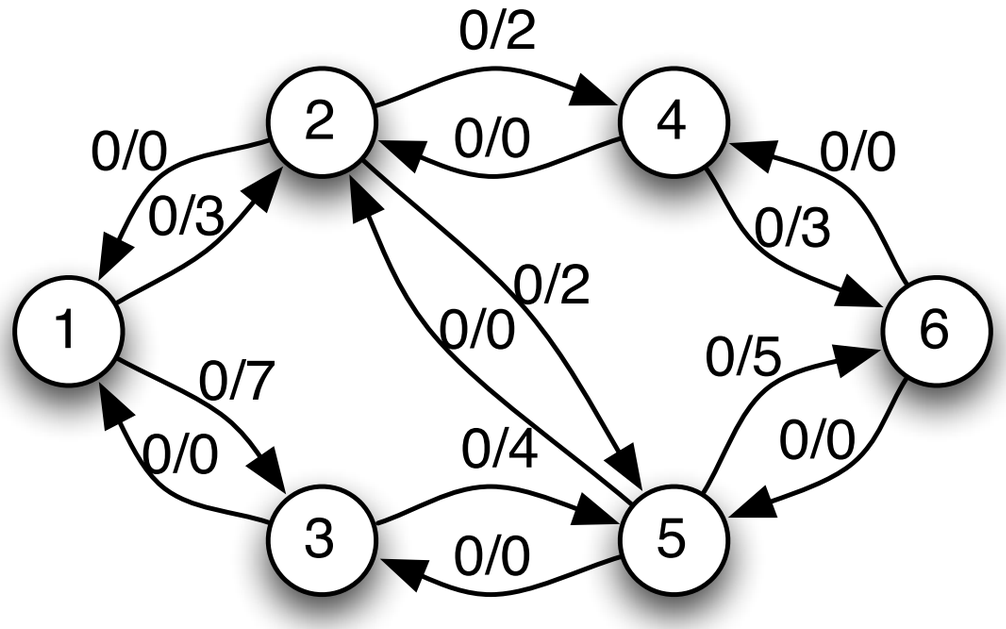
Algorithme de Ford-Fulkerson

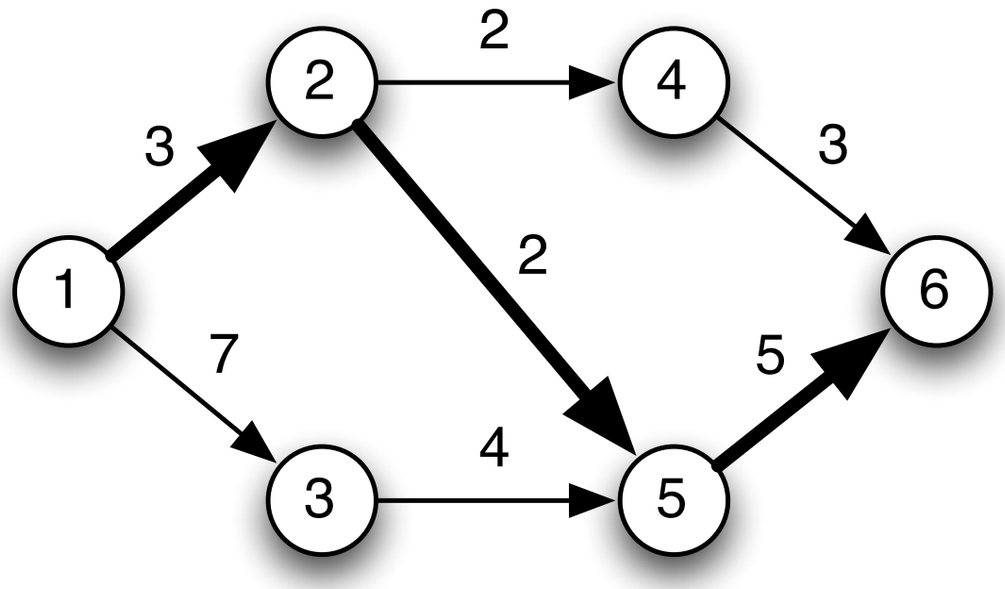
- L'algorithme de Ford-Fulkerson procède de la façon suivante.

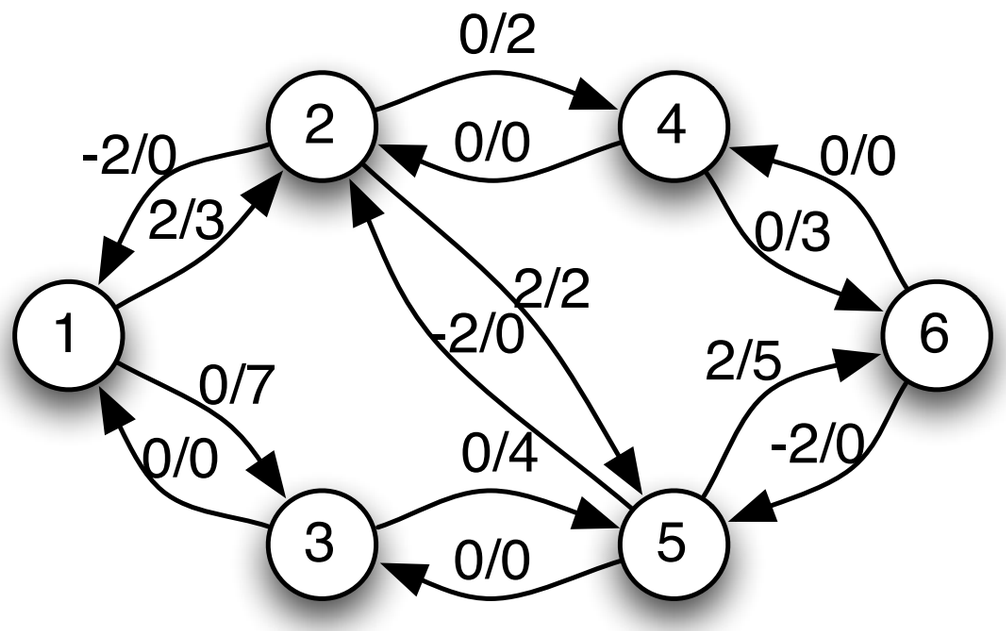
Algorithme 3 : Ford-Fulkerson(V, E, c, s, t)

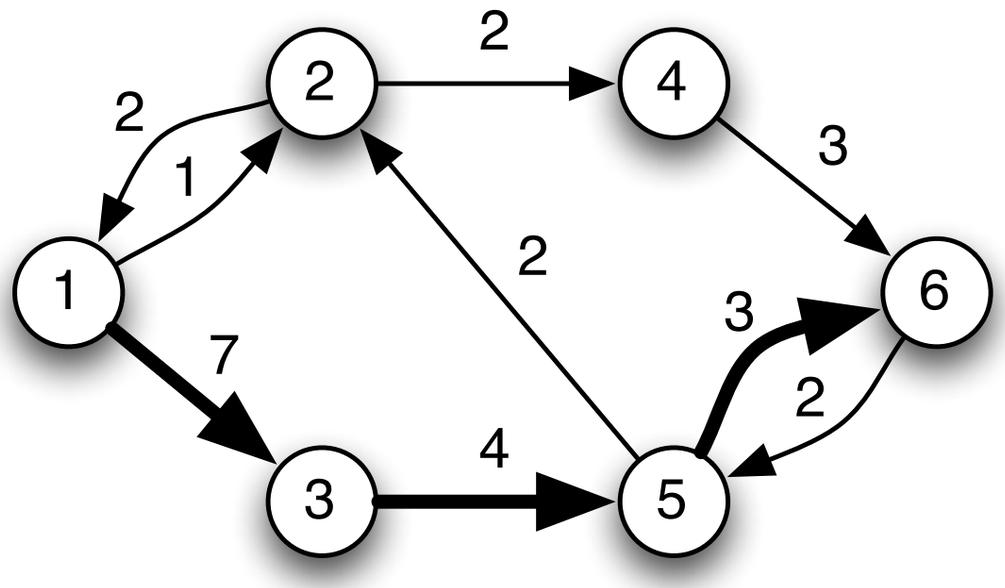
Constuire un vecteur f avec $\binom{|V|}{2}$ entrées initialisées à 0.
répéter
  Construire le graphe résiduel G_f
  Trouver un chemin C allant de s à t dans G_f
  si *un tel chemin existe* **alors**
    Soit q la plus petite capacité résiduelle d'une arête sur le chemin C
    pour *toutes les arêtes* (a, b) *du chemin* C **faire**
      $f(a, b) \leftarrow f(a, b) + q$
      $f(b, a) \leftarrow f(b, a) - q$
jusqu'à *ce qu'on ne trouve pas de chemin entre* s *et* t *dans* G_f
retourner f

Exécution de Ford-Fulkerson

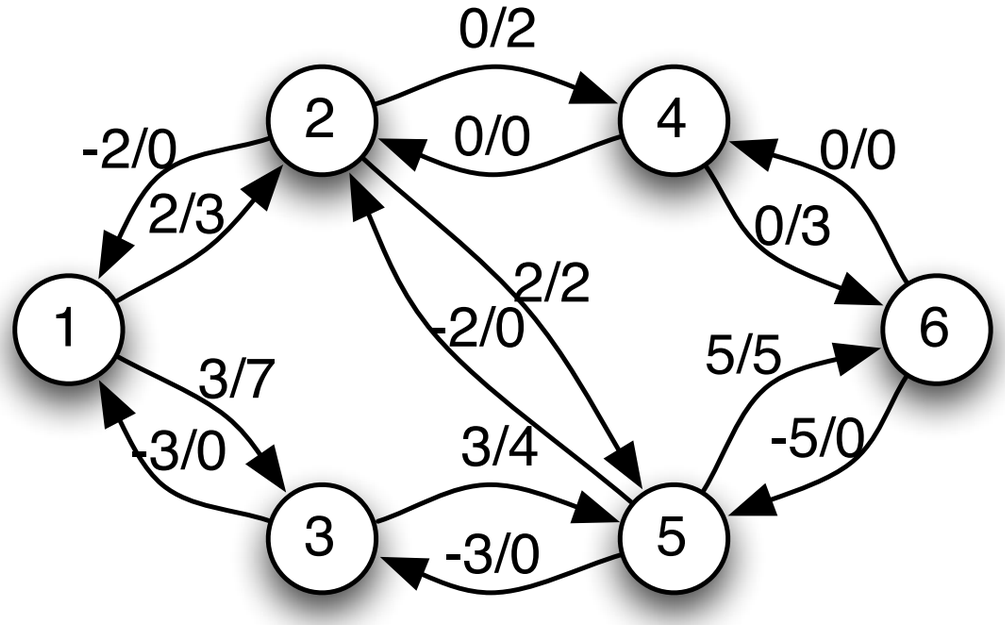


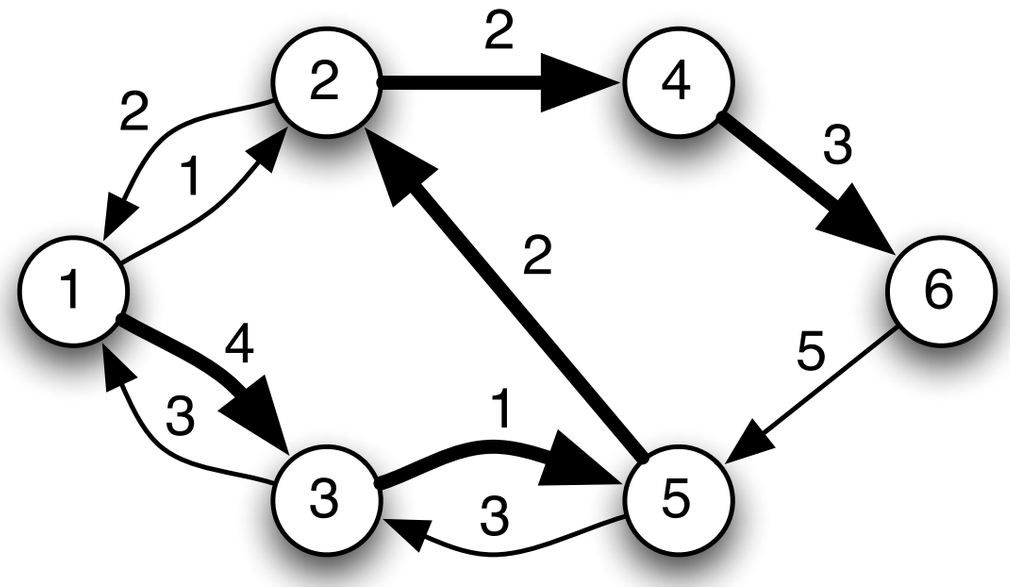


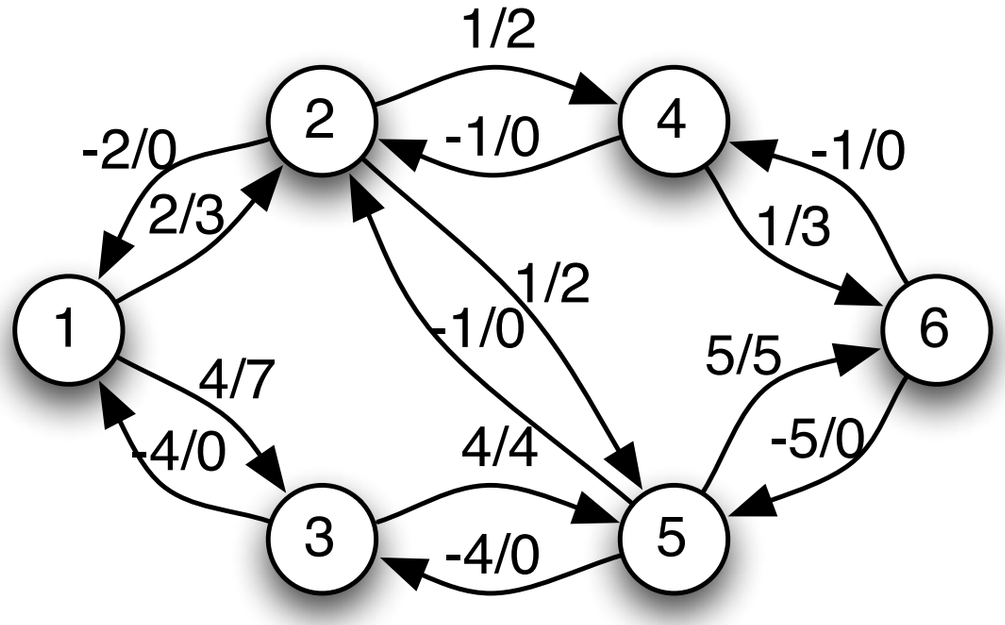


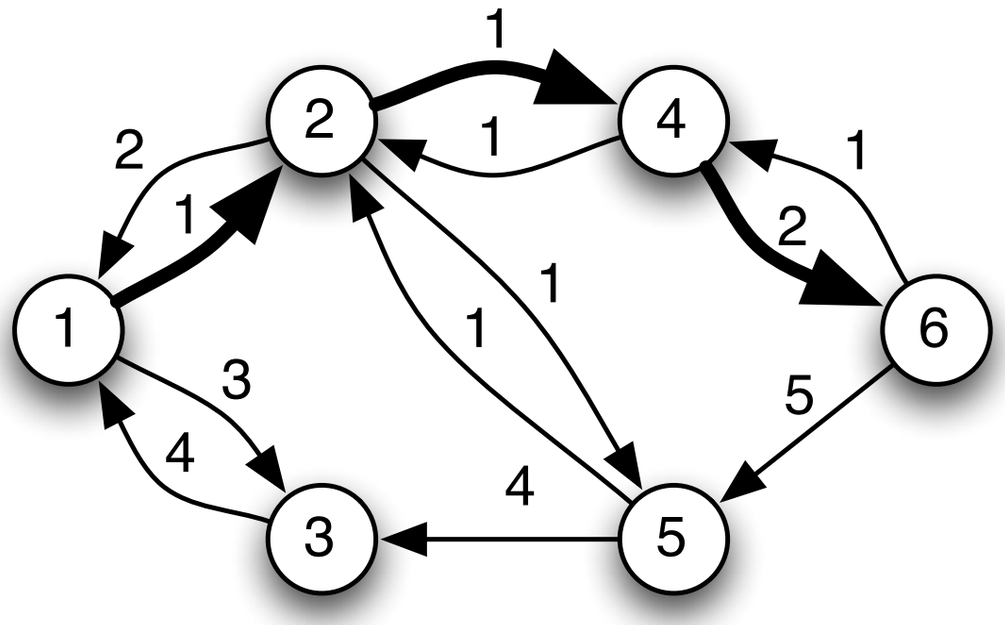


Exécution de Ford-Fulkerson

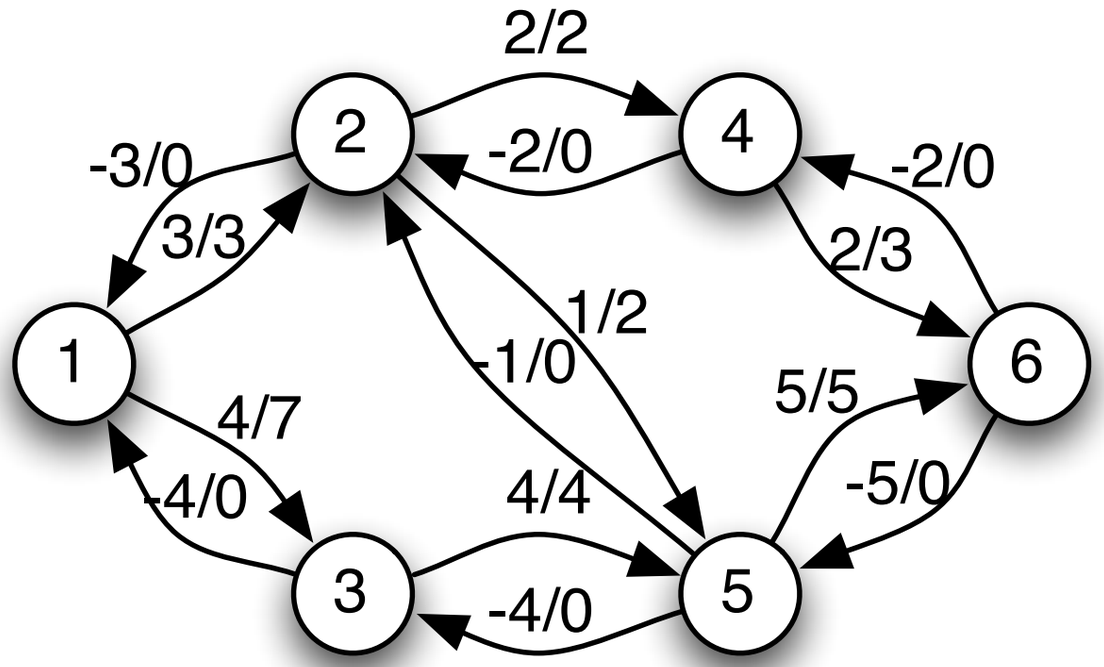


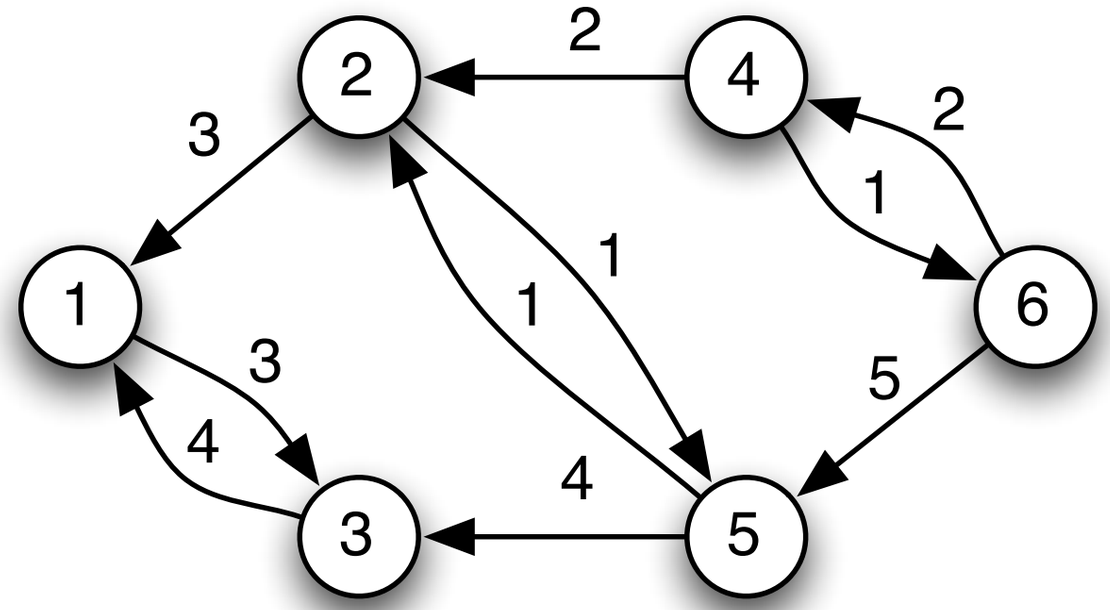






Exécution de Ford-Fulkerson



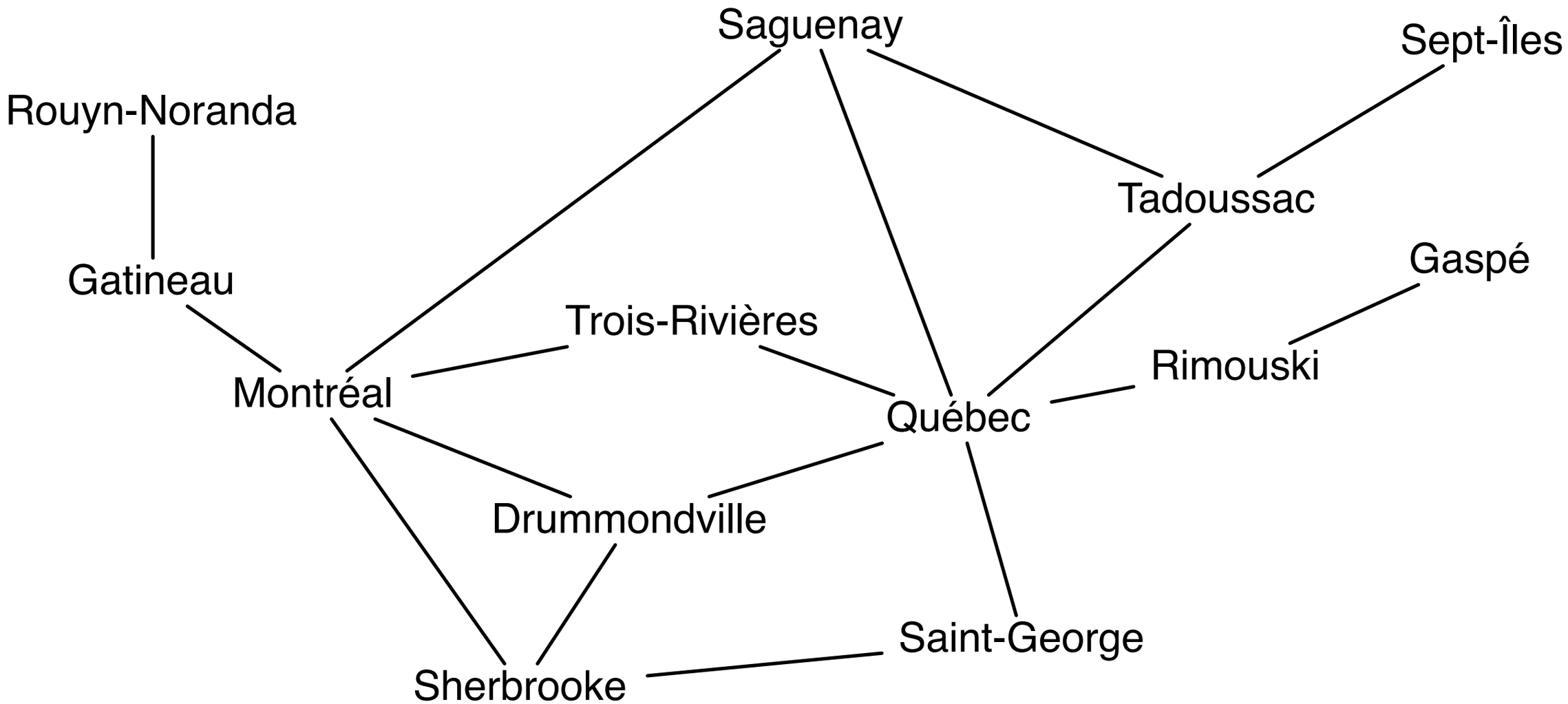


Complexité

- À chaque itération, l'algorithme de Ford-Fulkerson recherche un chemin dans le graphe résiduel, ce qui prend O(|E| + |V|) étapes. Puisque le graphe est connecté, nous avons |V| - 1 ≤ |E|. La recherche d'un chemin se fait donc en O(|E|) étapes.
- Le graphe résiduel peut ensuite être mis à jour en traitant chaque arête du chemin augmentant. Cette mise à jour prend donc O(|V|) étapes.
- Finalement, il peut y avoir autant d'itérations que la valeur du flot, c'est-à-dire v(f).
- La complexité totale de l'algorithme est donc de O(v(f)|E|).

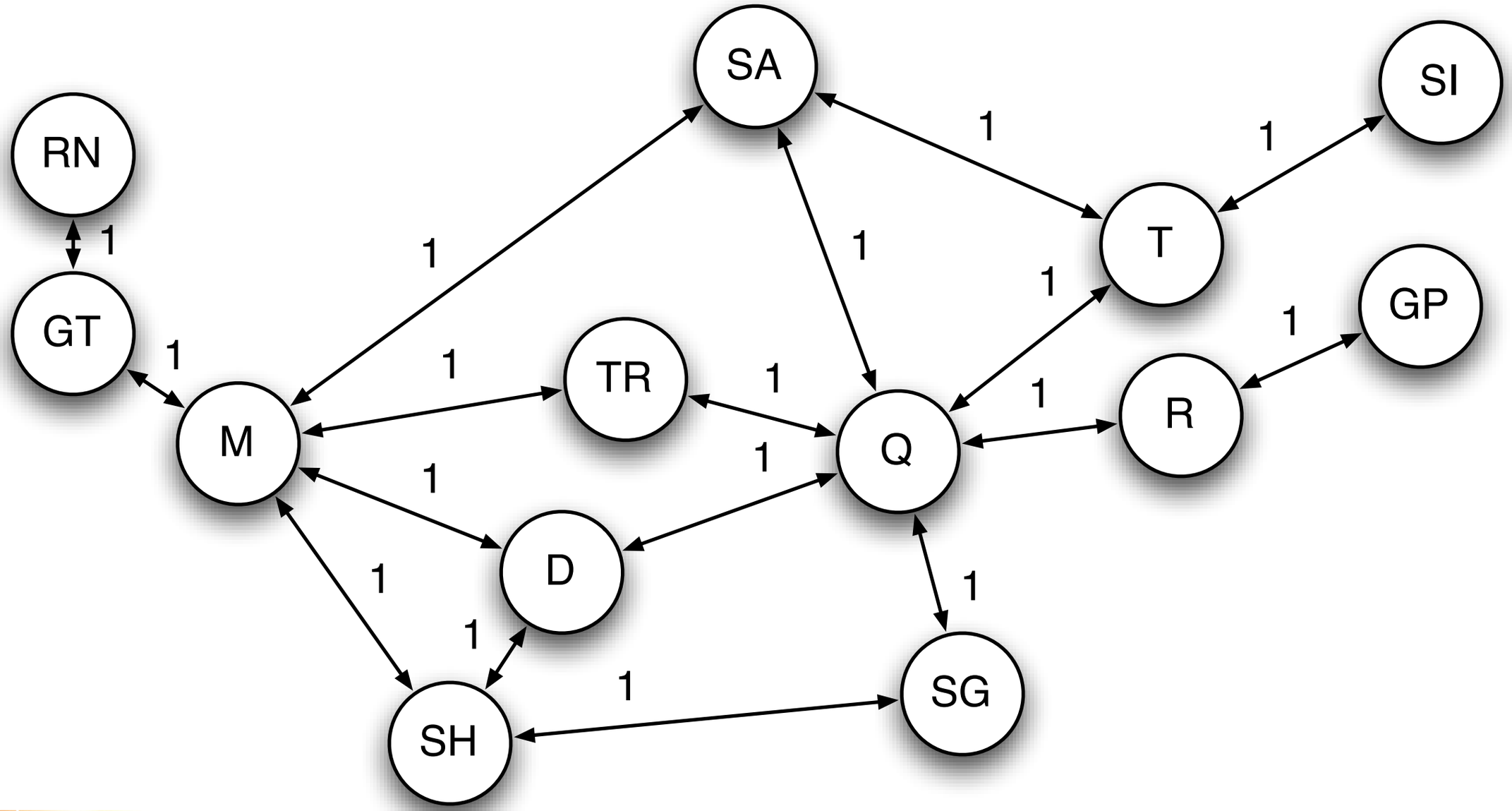
Robustesse d'un réseau

Une compagnie de câblodistribution a relié les villes du Québec avec de la fibre optique. Combien de fibres faut-il couper pour déconnecter Montréal de Québec?



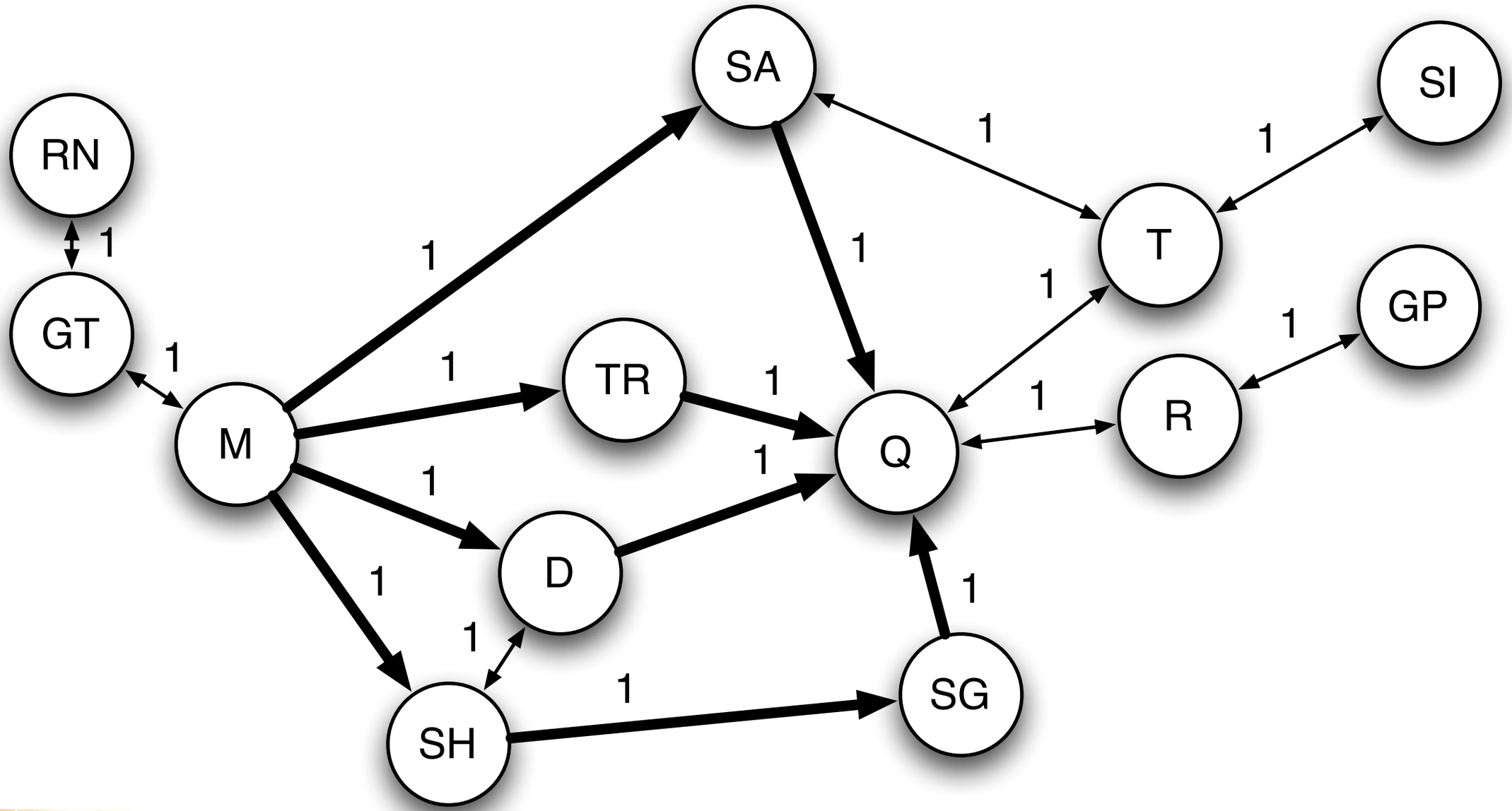
Robustesse d'un réseau

On construit un graphe où la capacité de chaque lien est d'une unité. Le flot maximum entre Montréal (M) et Québec (Q) est le nombre de fils devant être coupés pour déconnecter les deux villes.



Robustesse d'un réseau

On construit un graphe où la capacité de chaque lien est d'une unité. Le flot maximum entre Montréal (M) et Québec (Q) est le nombre de fils devant être coupés pour déconnecter les deux villes.



Coupe

- Une bipartition d'un ensemble V est formée de deux ensembles disjoints S et T de sorte que V est l'union de S et T.

$$S \cup T = V \qquad S \cap T = \emptyset$$

- Une **coupe** dans un réseau est une bipartition (S, T) des noeuds telle que la source est dans l'ensemble S et le puits est dans l'ensemble T.

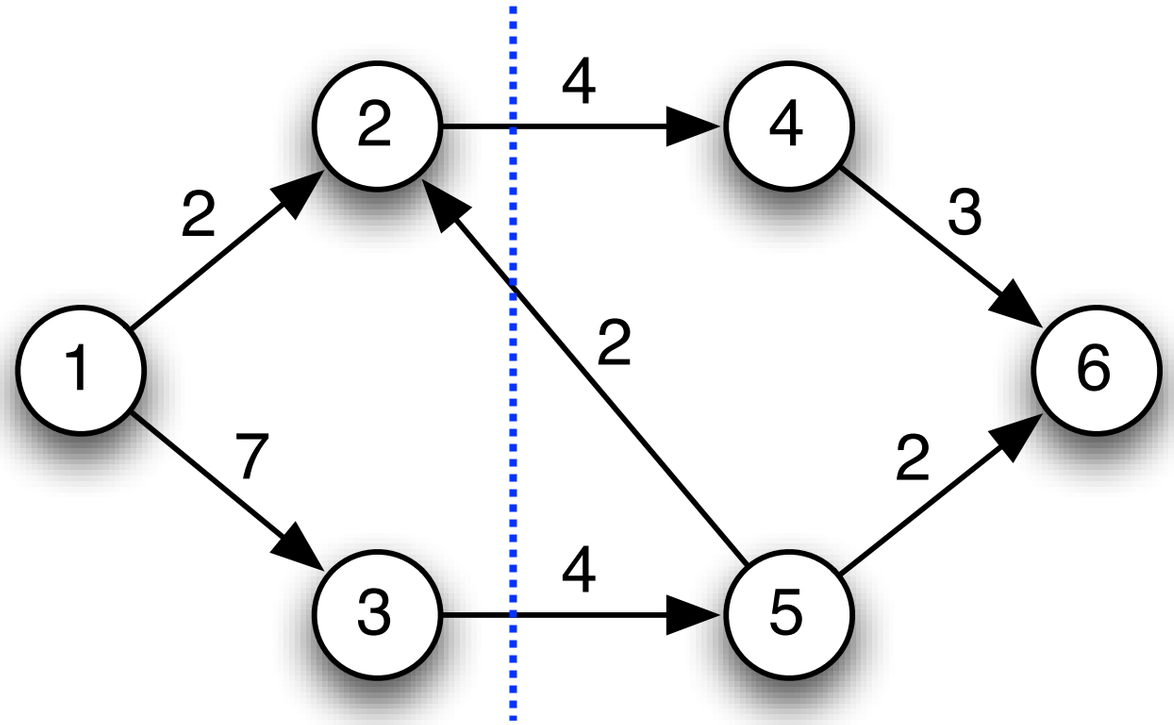
$$S \cup T = V \qquad S \cap T = \emptyset$$

$$s \in S \qquad t \in T$$

- La **capacité d'une coupe** (S, T) est la capacité des arêtes reliant un noeud dans S à un noeud dans T.

$$c(S,T) = \sum_{a \in S} \sum_{b \in T} c(a,b)$$

Exemple de coupe



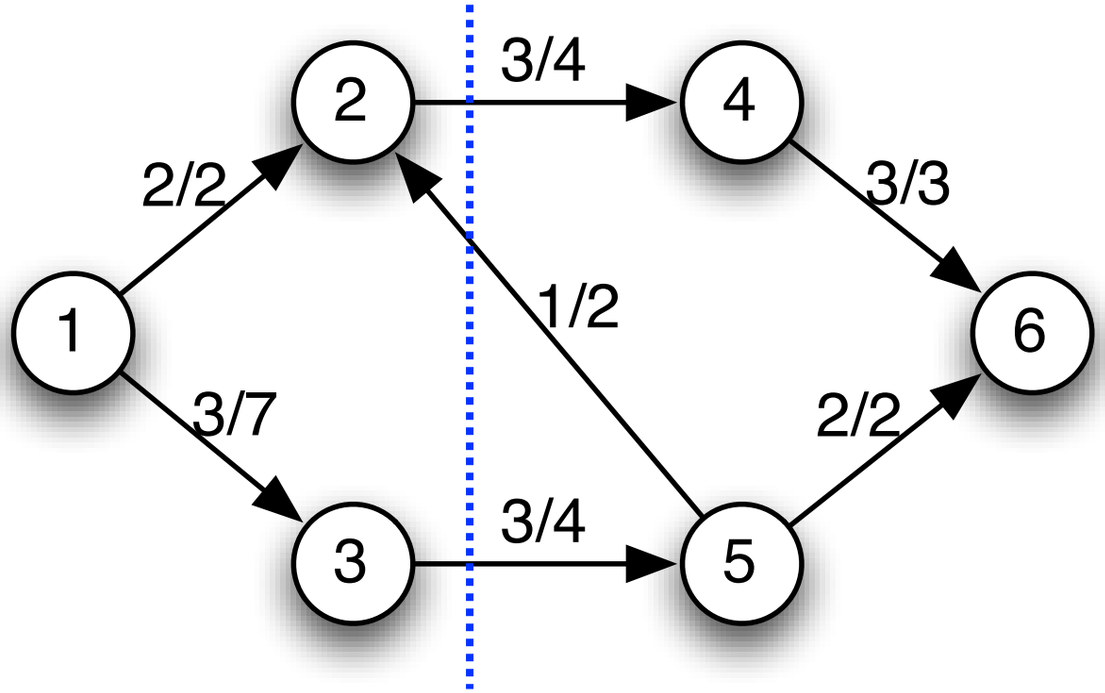
$S = \{1, 2, 3\}$ $T = \{4, 5, 6\}$

- La capacité de cette coupe est de 8.
- Nous écrivons $c(\{1, 2, 3\}, \{4, 5, 6\}) = 8$.

Le flot net d'une coupe

- Le **flot net d'une coupe** (S, T) est la quantité de flot (positive ou négative) passant d'un noeud dans S à un noeud dans T.

$$f(S,T) = \sum_{a \in S, b \in T} f(a,b)$$



$$f(\{1,2,3\},\{4,5,6\}) = f(2,4) + f(2,5) + f(3,5) = 3 - 1 + 3 = 5$$

Le flot net d'une coupe

- **Théorème:** Pour toute coupe (S, T) et tout flot valide f, la valeur de flot v(f) est égale au flot net de la coupe f(S, T).
- **Démonstration**:

$$
\begin{aligned}
v(f) &= \sum_{v \in V} f(s,v) \\
&= \sum_{v \in V} f(s,v) + \sum_{u \in S \setminus \{s\}} \sum_{v \in V} f(u,v) \quad \text{Par conservation du flot dans } u \\
&= \sum_{v \in V} \left(f(s,v) + \sum_{u \in S \setminus \{s\}} f(u,v) \right) \\
&= \sum_{v \in V} \sum_{u \in S} f(u,v)
\end{aligned}
$$

Propriétés d'une coupe

(suite de la démonstration)

$$
\begin{aligned}
v(f) &= \sum_{v\in T}\sum_{u\in S} f(u,v) + \sum_{v\in S}\sum_{u\in S} f(u,v) \\
&= \sum\sum f(u,v) + \sum_{v\in S,u\in S,v<u} (f(u,v)+f(v,u)) \\
&= \sum_{v\in T}\sum_{u\in S} f(u,v) \\
&= f(S,T)
\end{aligned}
$$

Théorème du flot maximum et de la coupe minimum

- Ces trois affirmations sont équivalentes:

  1. f est un flot maximum dans G;

  2. Le graphe résiduel G_f n'a pas de chemin augmentant;

  3. Il existe une coupe (S, T) dont la capacité c(S, T) est égale à v(f).

- **Démonstration:**

  - (1 => 2) S'il existait un chemin augmentant, on pourrait obtenir un flot de plus grande valeur ce qui est impossible avec un flot maximum.

Propriétés d'une coupe

- (2 => 3) Soit S l'ensemble des noeuds pouvant être rejoints par la source s dans le graphe résiduel G_f et soit $T = V \setminus S$. La partition (S, T) est une coupe puisque la source est dans S et le puits est dans T.
- Soit deux noeuds $u \in S$ et $v \in T$.
  - Si $(u, v) \in E$ est une arête alors $f(u, v) = c(u, v)$ sinon s pourrait rejoindre v dans G_f.
  - Si $(u, v) \notin E$ et $(v, u) \in E$ est une arête alors $f(u, v) = 0$ sinon s pourrait rejoindre v dans G_f.
  - Sinon, $f(u, v) = 0$ car aucun flot ne peut circuler entre deux noeuds qui ne sont pas reliés par une arête.

Propriétés d'une coupe

$$
\begin{aligned}
v(f) &= f(S,T) \\
&= \sum_{a \in S} \sum_{b \in T} f(a,b) \\
&= \sum_{(a,b) \in E, a \in S, b \in T} f(a,b) + \sum_{(b,a) \in E, (a,b) \notin E, a \in S, b \in T} f(a,b) \\
&= \sum_{(a,b) \in E, a \in S, b \in T} f(a,b) + 0 \\
&= c(S,T)
\end{aligned}
$$

Propriétés d'une coupe

- (3 => 1) La valeur d'un flot ne peut pas dépasser la capacité d'une de ses coupes.
- **Démonstration**:

$$
\begin{aligned}
v(f) &= f(S,T) \\
&= \sum_{a \in S, b \in T} f(a,b) \\
&\leq \sum_{a \in S, b \in T} c(a,b) \qquad \text{Par la contrainte de capacité} \\
&= c(S,T)
\end{aligned}
$$

- Un flot dont la valeur est égale à la capacité d'une coupe est donc maximum.

Résumé des propriétés des flots et des coupes

- La valeur d'un flot et le flot net d'une coupe sont égaux.

$$v(f) = f(S, T)$$

- La valeur du flot maximum est égale à la plus petite capacité d'une de ses coupes.

$$v(f) = \min_{S|s \in S, t \notin s} c(S, V \setminus S)$$

- Trouver le flot maximum est un problème de maximisation.
- Trouver la coupe de capacité minimum est un problème de minimisation.
- Les deux problèmes convergent vers les mêmes valeurs.

Problème d'ordonnancement

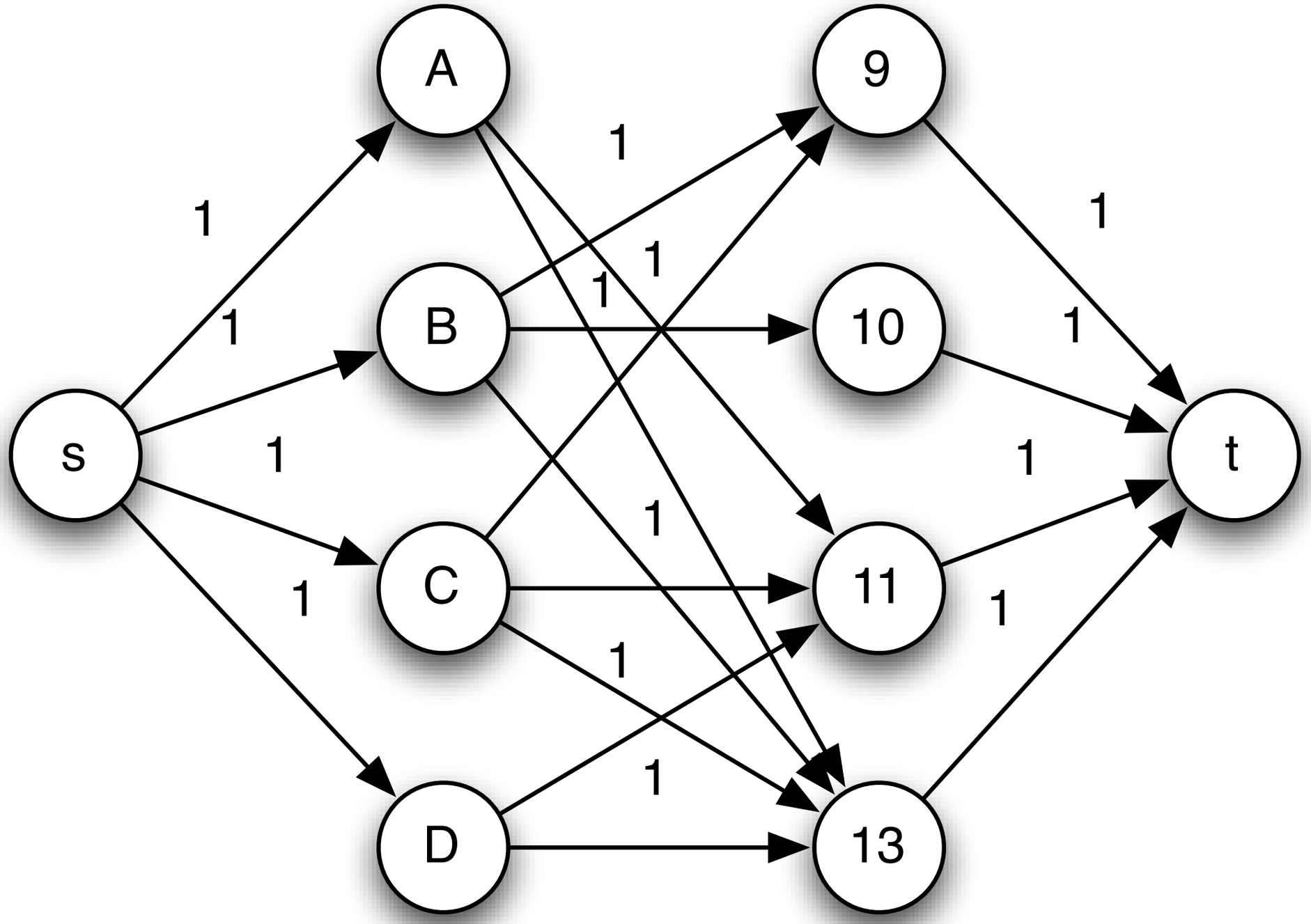
Personne	Disponibilités
Alice	11h00 13h00
Benoît	9h00 10h00 13h00
Clotilde	9h00 11h00 13h00
Dany	11h00 13h00

- Quatre personnes doivent donner un séminaire.
- Quatre plages horaires ont été prévues à 9h00, 10h00, 11h00 et 13h00.
- On a demandé à chaque orateur de fournir leurs disponibilités.
- **Problème:** Produire l'horaire des séminaires.

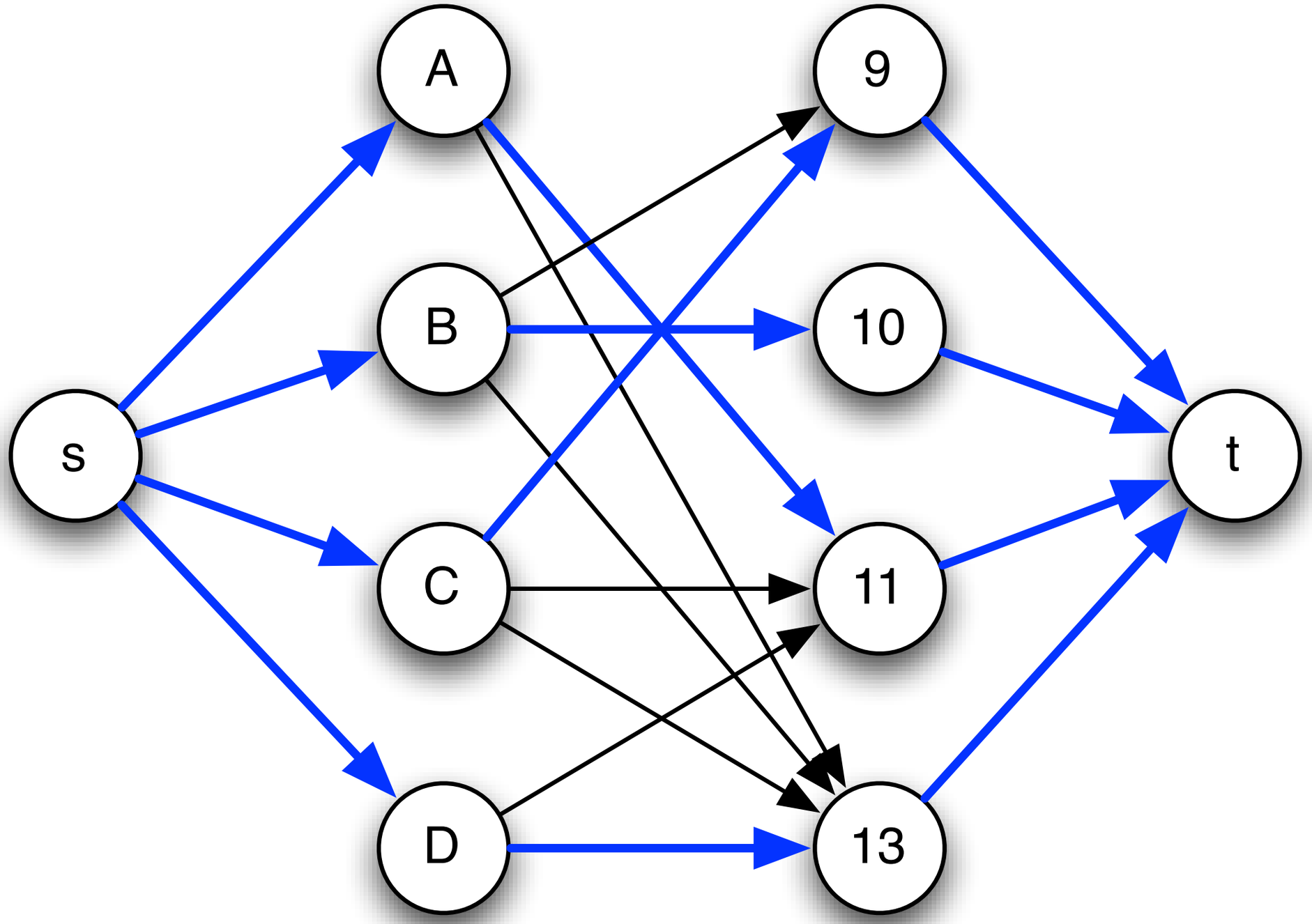
Problème d'ordonnancement

- On construit un graphe avec un noeud par personne et un noeud par plage horaire.
- On ajoute une arête entre une personne et ses plages de disponibilité d'une capacité d'une unité de flot.
- La source est reliée à toutes les personnes par des arêtes de capacité un.
- Toutes les plages horaires sont reliées au puits par une arête de capacité un.

Problème d'ordonnancement

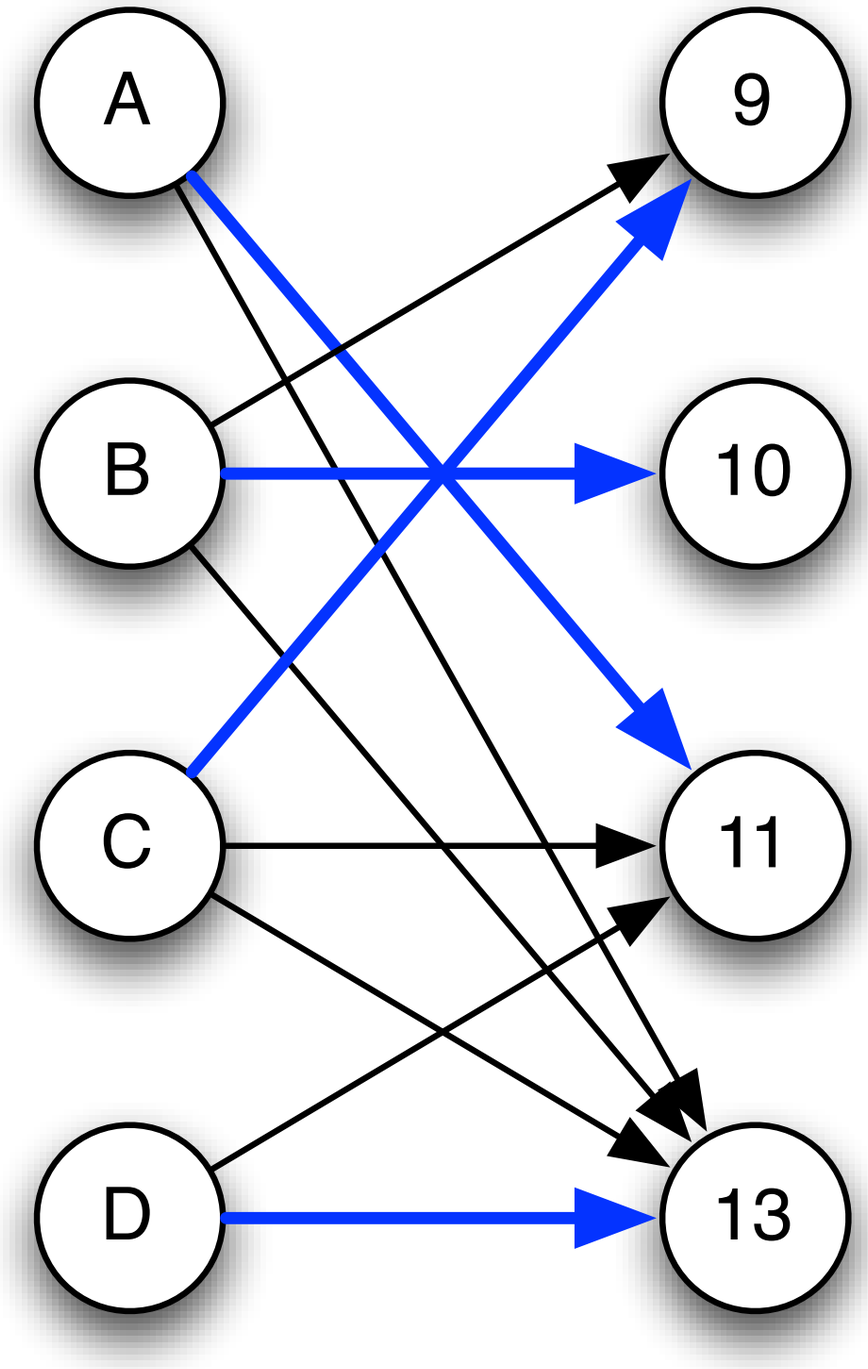


Problème d'ordonnancement



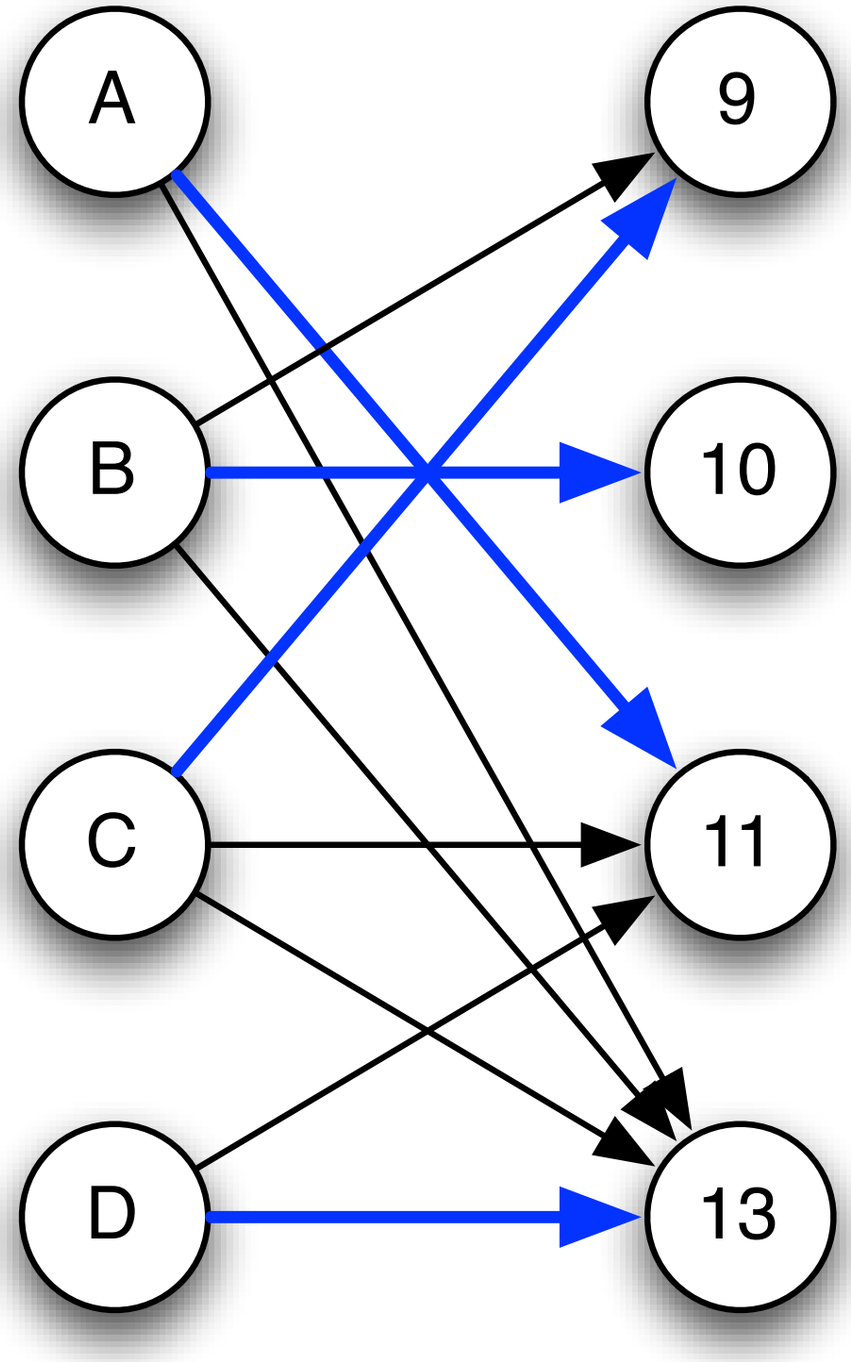
Couplage

- La partie centrale du graphe s'appelle un couplage.
- Un **couplage** est un sous-ensemble d'arêtes qui ne sont pas adjacentes.
- Un **couplage maximum** est un couplage dont la cardinalité est maximum.



All-Different

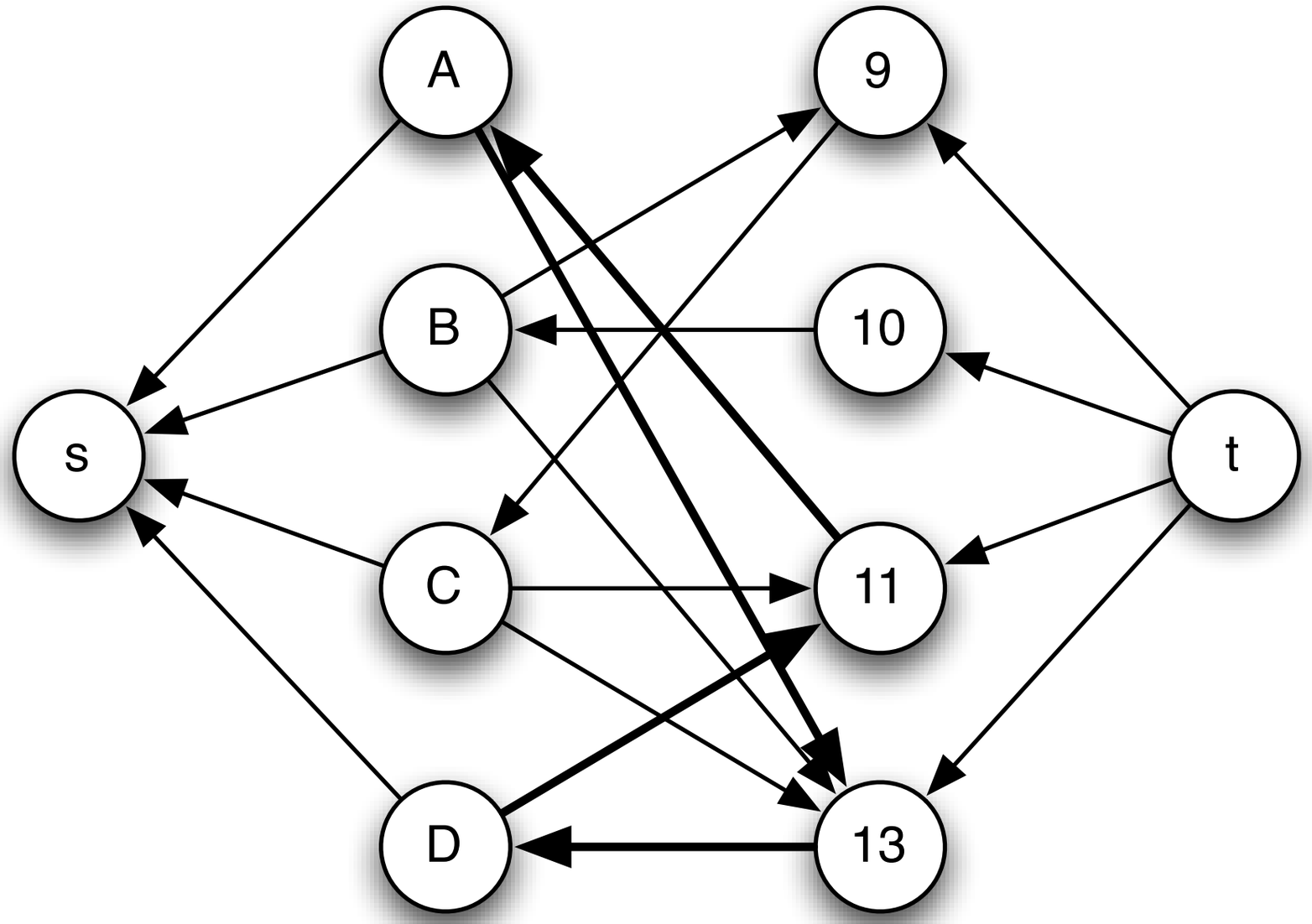
- Un couplage maximum est également un support de domaine pour la contrainte All-Different qui requiert les variables de sa portée à prendre des valeurs distinctes.
- Dans ce cas-ci, nous avons le support pour la contrainte
  All-Different(A, B, C, D)
  avec les domaines
  dom(A) = {11, 13}
  dom(B) = {9, 10, 13}
  dom(C) = {9, 11, 13}
  dom(D) = {11, 13}



All-Different

- Nous avons donc un algorithme pour détecter si la contrainte All-Different peut être satisfaite.
- Il nous faut à présent un algorithme de filtrage.
- Nous trouverons la réponse dans le graphe résiduel du flot constituant un support de domaine à la contrainte.

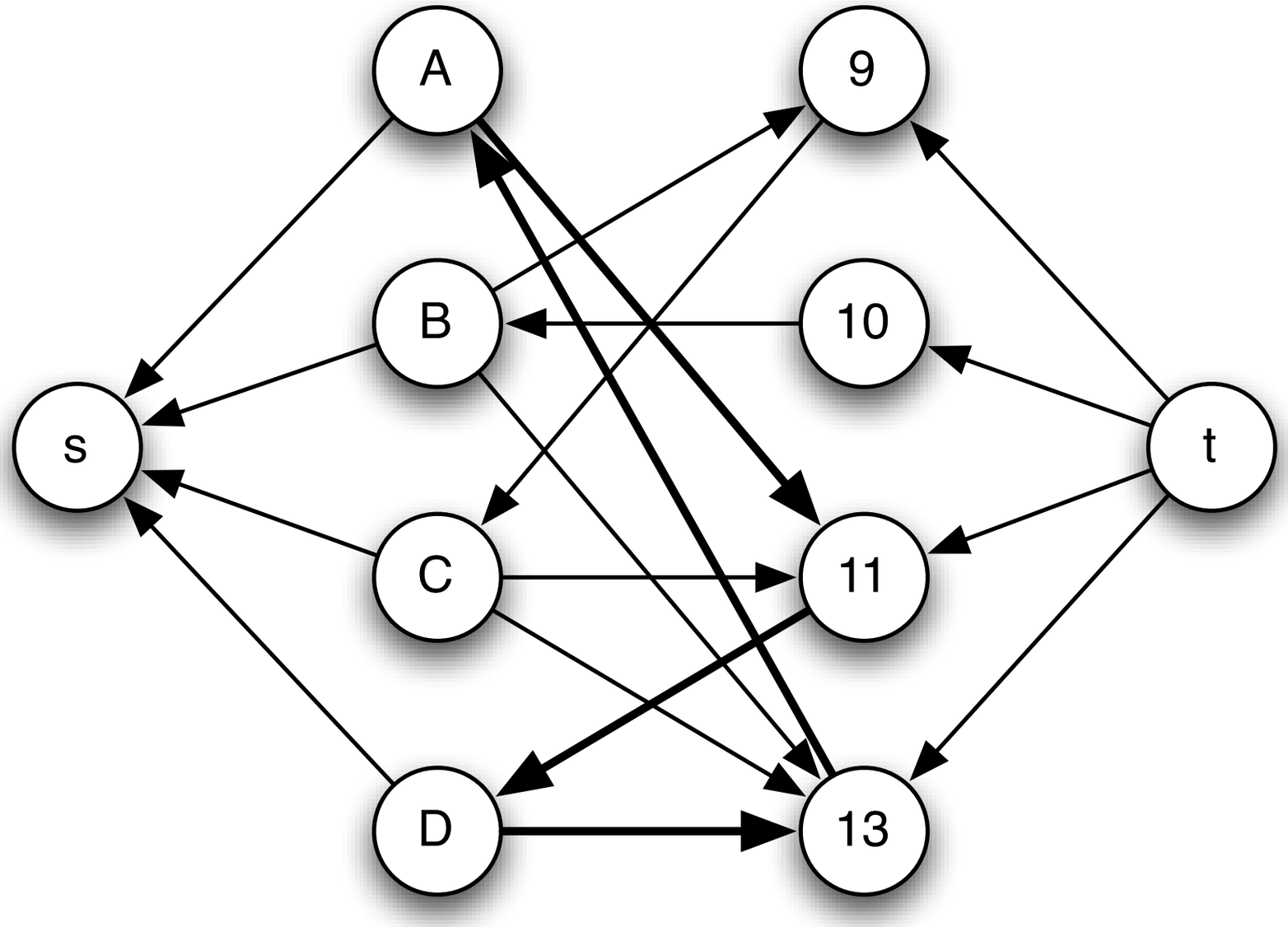
All-Different



Support:
A = 11
B = 10
C = 9
D = 13

- La capacité résiduelle de chaque arête est d'une unité.
- **Question:** Qu'arrive-t-il si on pousse une unité de flot sur le cycle A, 13, D, 11, A?

All-Different

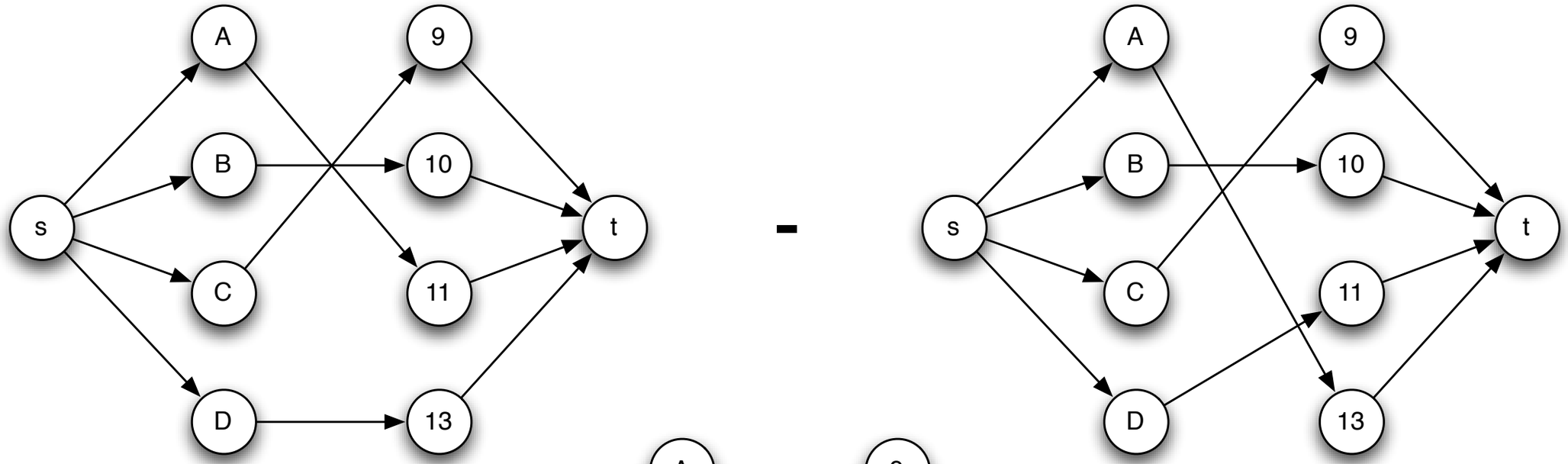


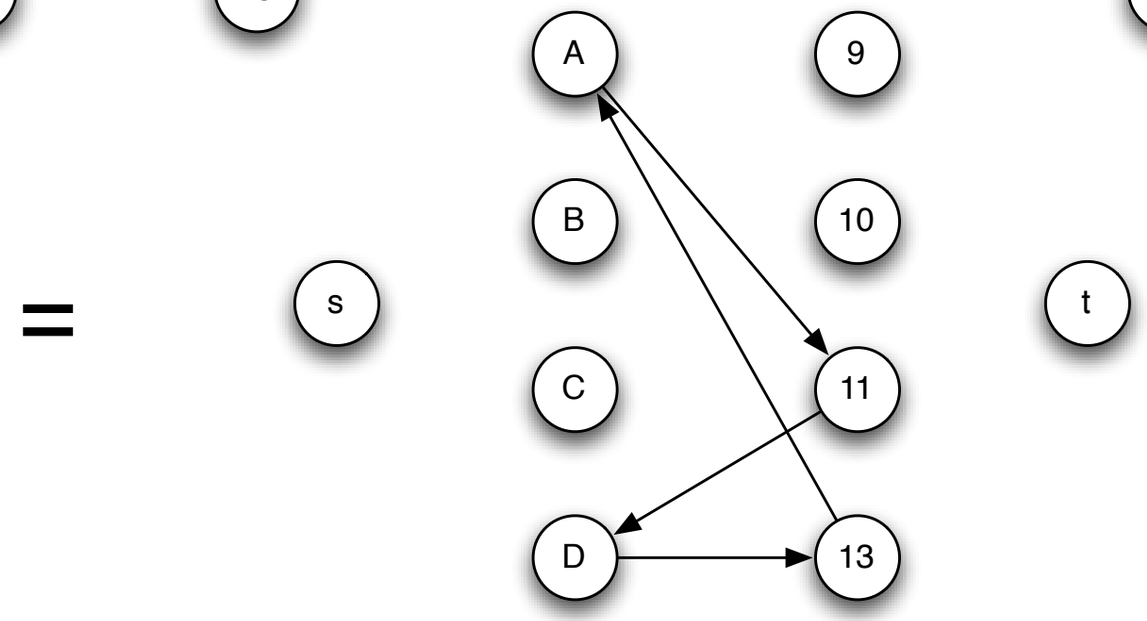
Support:
A = 13
B = 10
C = 9
D = 11

- **Réponse:** on obtient un nouveau support de domaine.

All-Different

- La différence entre deux flots passant par le graphe de la contrainte All-Different est un ensemble de cycles disjoints. Il est toujours possible de transformer un flot f en un flot g en poussant une unité de flot sur ces cycles.





All-Different

- **Conséquence:** La valeur v a un support de domaine dans le domaine de X_i si et seulement si, pour un flot f, l'arête (v, X_i) appartient au graphe résiduel ou l'arête (X_i, v) appartient à un cycle du graphe résiduel.

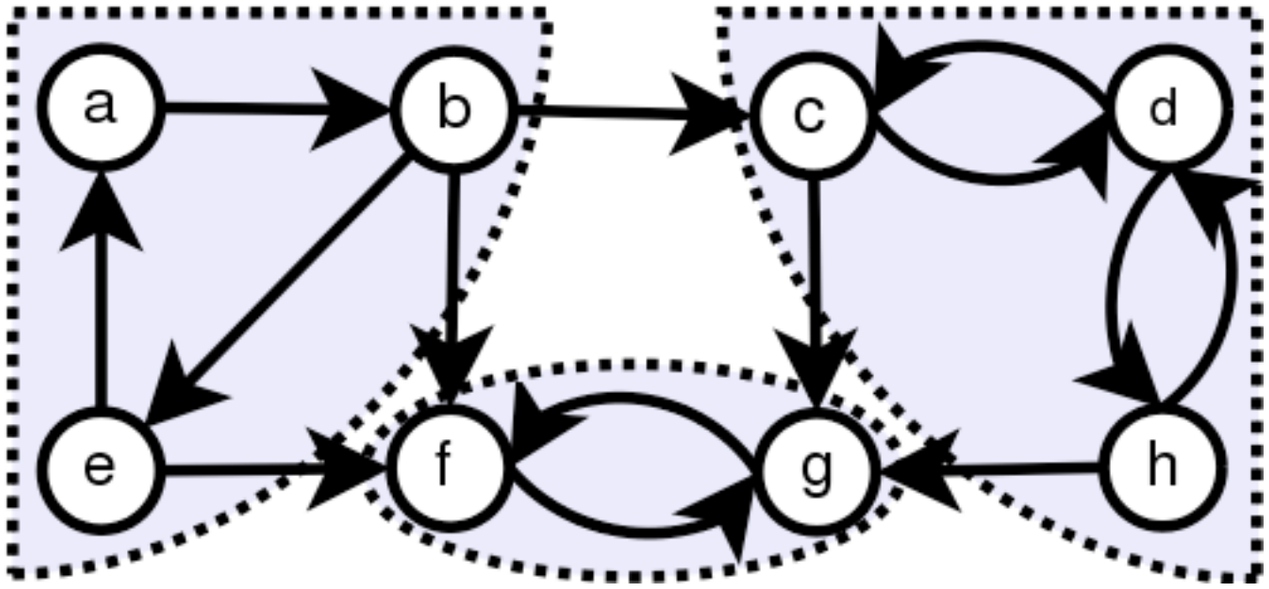
Algorithme de Régin

- L'algorithme de Régin est un algorithme de filtrage pour la contrainte All-Different.
- La première étape consiste à calculer un couplage en utilisant, par exemple, l'algorithme de Ford-Fulkerson. On obtient donc un flot f.
- L'algorithme marque ensuite toutes les arêtes faisant partie d'un cycle dans le graphe résiduel G_f.
- La phase de filtrage retire v du domaine de X_i si le flot circulant sur l'arête (X_i, v) est nul et si l'arête (X_i, v) n'est pas marquée.

Composantes fortement connexes

- Un graphe orienté est **fortement connexe** s'il existe, pour toute paire de noeuds (a, b), un chemin entre a et b.
- Une **composante fortement connexe** d'un graphe est un ensemble maximal de noeuds qui forment un sous-graphe fortement connexe.
- Dit autrement, une composante fortement connexe est un ensemble de noeuds où chaque noeud peut rejoindre tous les autres noeuds de cet ensemble. L'ensemble est maximal, donc on ne peut pas y ajouter un noeud supplémentaire.

Composantes fortement connexes



Ce graphe contient trois composantes fortement connexes: {a, b, e}, {f, g} et {c, d, h}

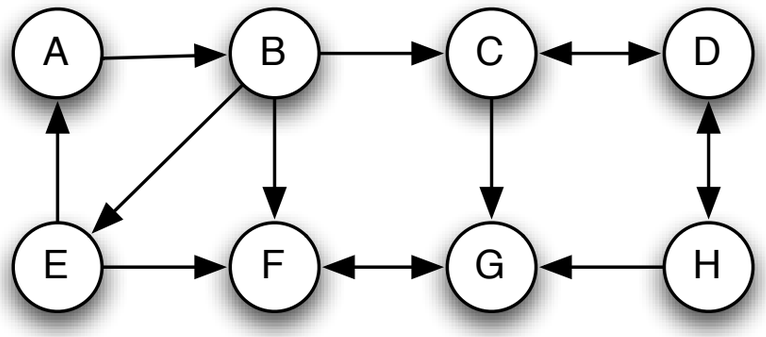
Source: Wikipédia, rubrique « Composante fortement connexe »

L'algorithme de Kosaraju

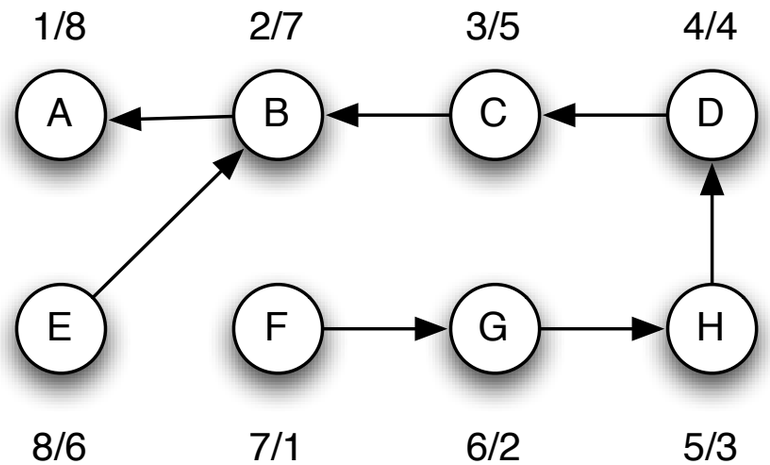
- L'algorithme de Kosaraju permet de calculer les composantes fortement connexes d'un graphe.
- Cet algorithme effectue une fouille en profondeur sur le graphe.
- Chaque fois qu'un noeud est colorié en noir, ce noeud est ajouté à une pile.
- L'algorithme construit ensuite le graphe transposé, c'est-à-dire le graphe où toutes les arêtes ont leur orientation inversée.
- Puis, l'algorithme procède à une fouille en profondeur sur le graphe transposé en traitant les noeuds dans l'ordre inverse qu'ils ont été empilés.
- À la fin de cette deuxième fouille, chaque arbre dans la forêt définie par le vecteur *Parent* est une composante fortement connexe.

L'algorithme de Kosaraju

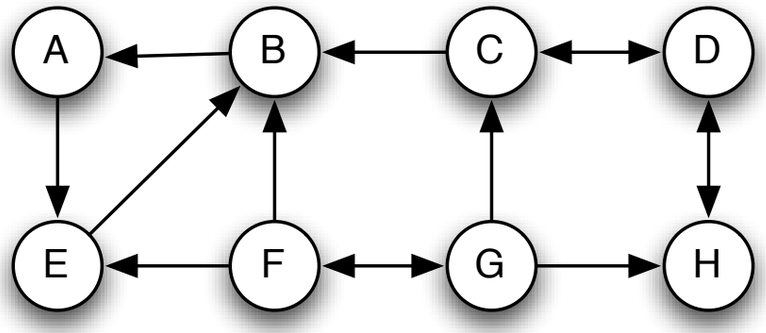
Graphe original



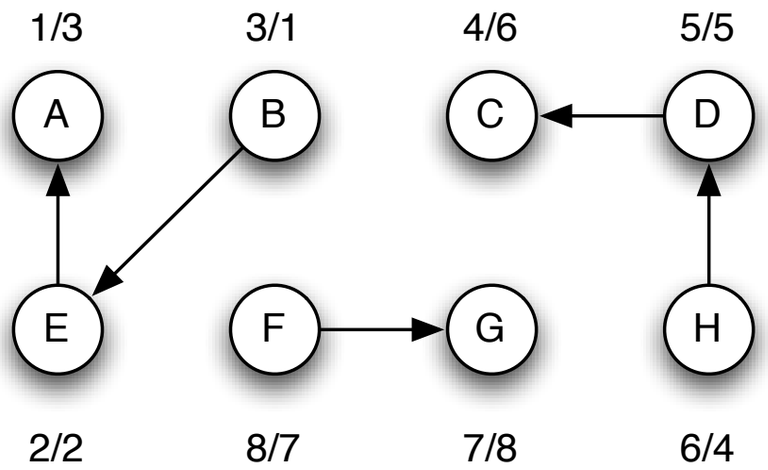
Vecteur Parent après une fouille en profondeur ordre de visite / post-ordre



Graphe transposé



Vecteur Parent après une fouille en profondeur ordre de visite / post-ordre



Algorithme de Régin

Algorithme 4 : Filtrage-All-Different($X_1, \ldots, X_n$)

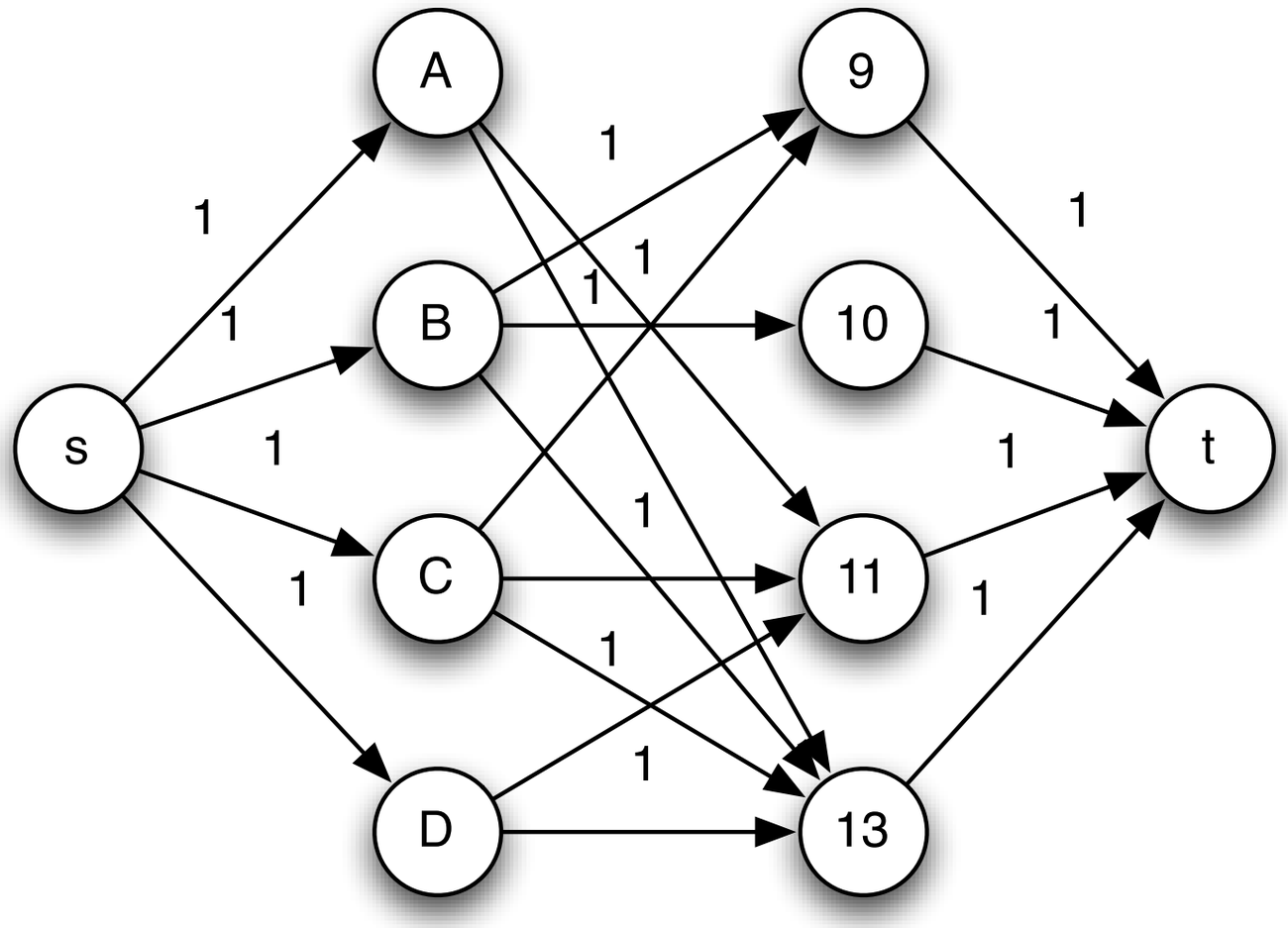
Construire un graphe G avec l'ensemble des noeuds $V = \{X_1, \ldots, X_n\} \cup \bigcup_{i=1}^{n} \mathrm{dom}(X_i) \cup \{s, t\}$
pour $i \in 1..n$ **faire**
  pour $v \in \mathrm{dom}(X_i)$ **faire**
    Ajouter l'arête (X_i, v) de capacité 1 au graphe G
  Ajouter l'arête (s, X_i) de capacité 1 au graphe G
pour $v \in \bigcup_{i=1}^{n} \mathrm{dom}(X_i)$ **faire**
  Ajouter l'arête (v, t) de capacité 1 au graphe G
Utiliser l'algorithme de Ford-Fulkerson pour calculer un flot maximum f entre s et t
si *la valeur du flot* $v(f)$ *n'est pas* n **alors**
  retourner La contrainte est insatisfiable
Utiliser l'algorithme Kosaraju pour calculer les composantes fortement connexes dans le graphe résiduel
pour $i \in 1..n$ **faire**
  pour $v \in \mathrm{dom}(X_i)$ **faire**
    si $f(X_i, v) = 0 \wedge composante(X_i) \neq composante(v)$ **alors**
      $\mathrm{dom}(X_i) \leftarrow \mathrm{dom}(X_i) \setminus \{v\}$
retourner La contrainte est satisfiable

Trace

- Considérez la contrainte All-Different(A, B, C, D) avec les domaines suivants:

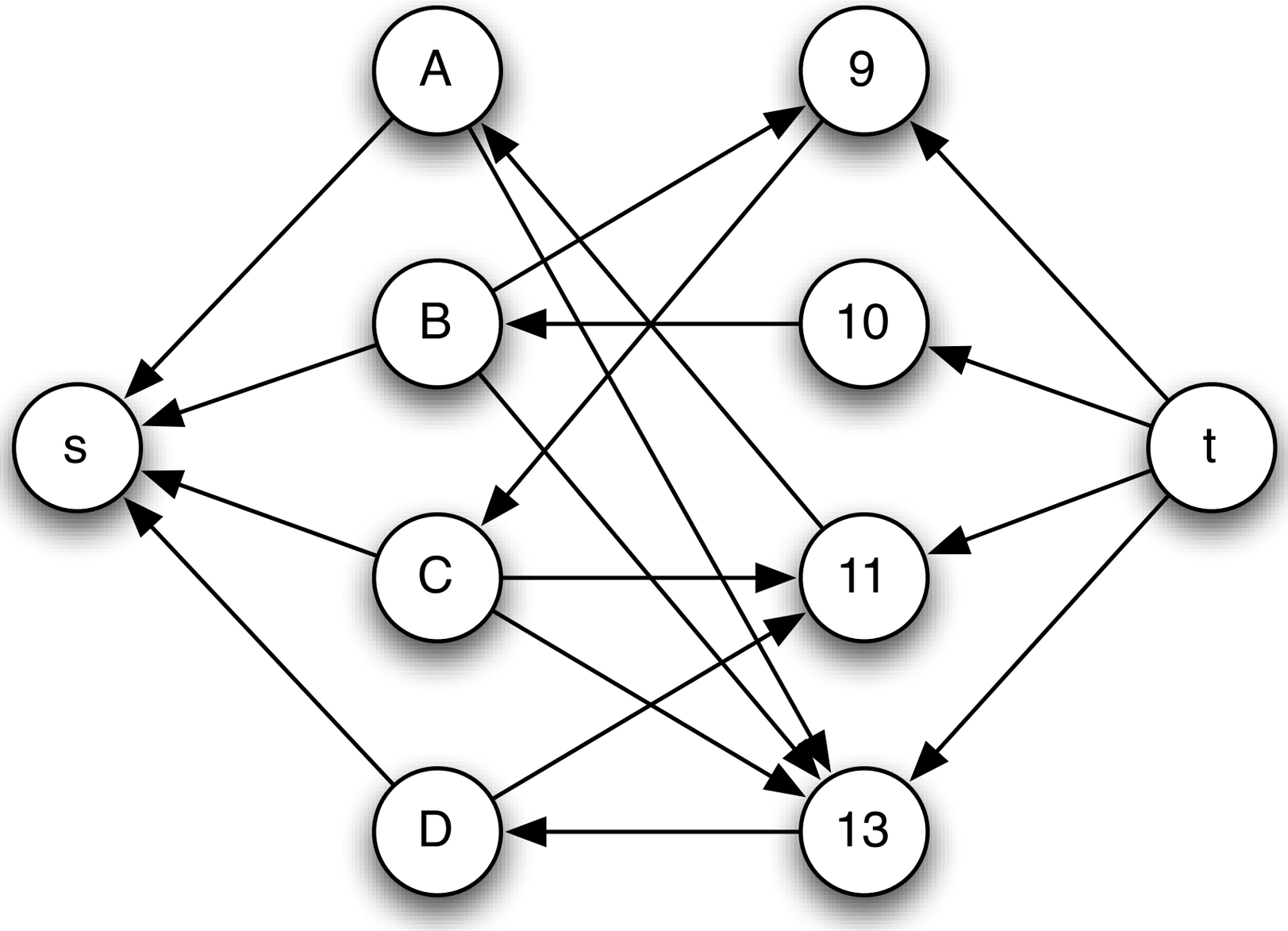
  dom(A) = {11, 13}
  dom(B) = {9, 10, 13}
  dom(C) = {9, 11, 13}
  dom(D) = {11, 13}

- Construction du graphe.



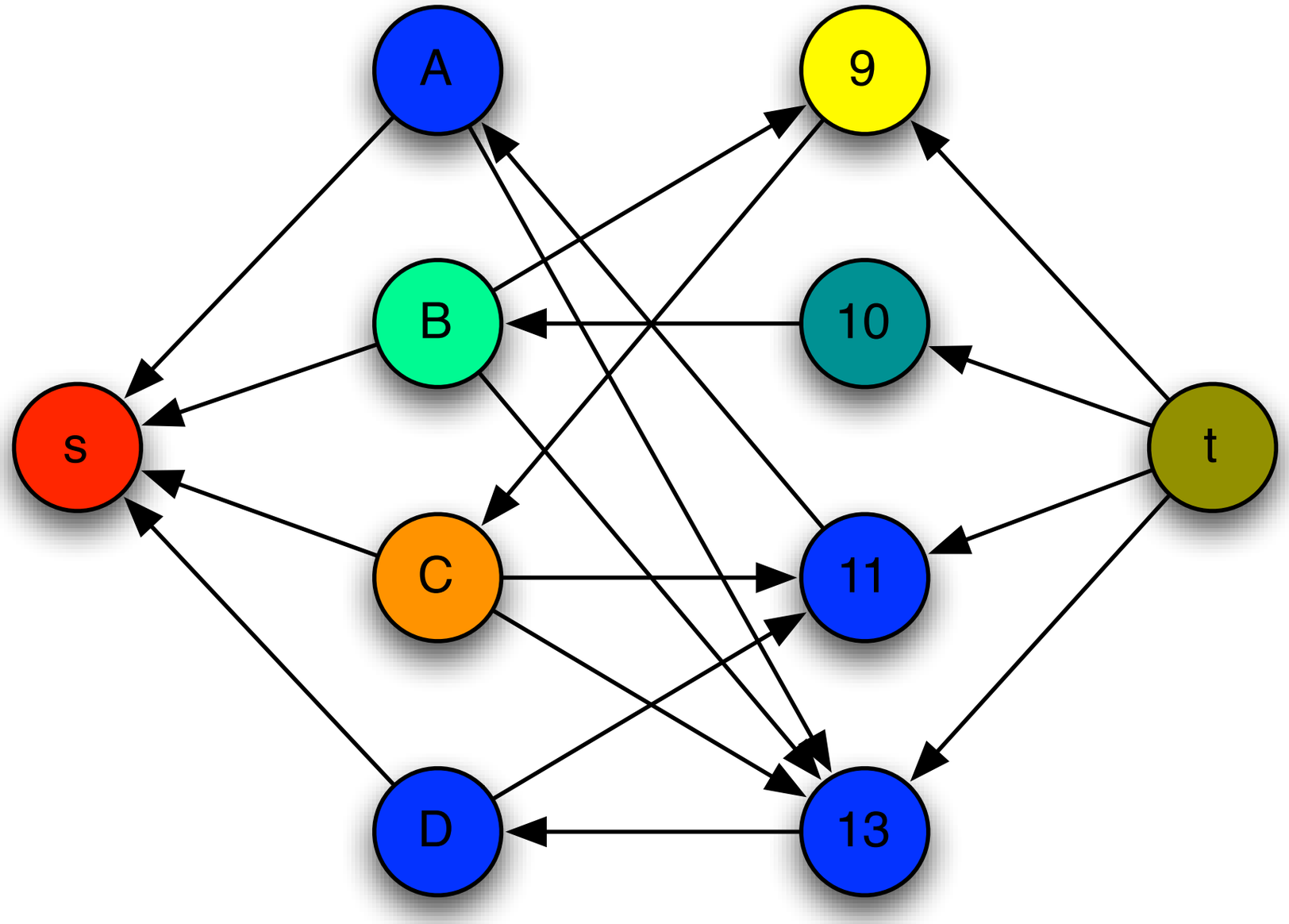
Trace

- Calcul d'un flot (couplage) dans le graphe et construction du graphe résiduel.



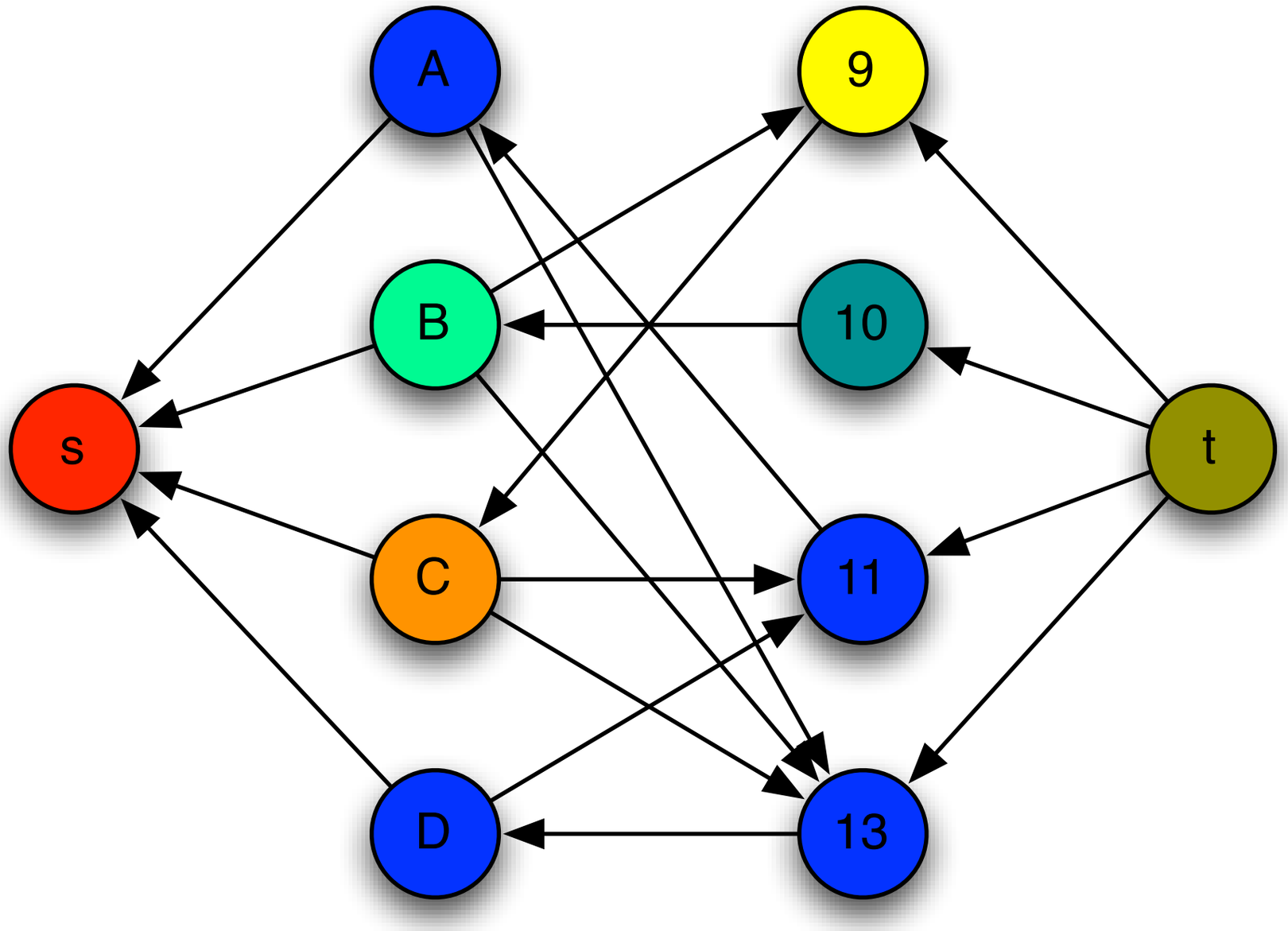
Trace

- Calcul des composantes fortement connexes.



Trace

- Filtrage des domaines.



dom(A) = {11, 13}
dom(B) = {9, 10, 13}
dom(C) = {9, 11, 13}
dom(D) = {11, 13}

dom(A) = {11, 13}
dom(B) = {10}
dom(C) = {9}
dom(D) = {11, 13}

Complexité

- Soit D la somme des cardinalités des domaines.
- Analyse de complexité:
  - Calcul d'un flot: O(n D)
  - Calcul des composantes fortement connexes: O(D)
  - Autres boucles de l'algorithme: O(D)
  - Total: O(nD)

Complexité

- En pratique, il est possible de ramener l'algorithme à une complexité de $O(\delta D)$ où δ est le nombre de valeurs supprimées des domaines depuis le dernier appel à l'algorithme de filtrage.
- À chaque fois qu'une valeur v est supprimée du domaine de X_i, on vérifie si le flot est encore valide. Si ce n'est pas le cas, c'est qu'il y avait une unité de flot circulant sur l'arête (X_i, v). On calcule donc un chemin augmentant dans le graphe résiduel dont l'origine est Xi et la destination est v. Ce chemin peut être calculé en temps O(D) et rétablira la validité du flot.

Production et consommation de flot

- Considérons une variante du problème du flot maximum où il n'y a ni source ni puits, mais où les noeuds peuvent produire ou consommer des unités de flots.
- Pour chaque noeud i du graphe, on associe une valeur b_i.
  - Si $b_i > 0$, alors le noeud i injecte b_i unités de flot dans le graphe.
  - Si $b_i < 0$, alors le noeud i consomme b_i unités de flot.
- La somme des valeurs de b_i pour tous les noeuds est nulle.
- On révise la contrainte de conservation du flot comme suit:

$$\sum_{b|(a,b)\in E} f(a,b) - \sum_{b|(b,a)\in E} f(b,a) = b_a \qquad \forall\, a \in V$$

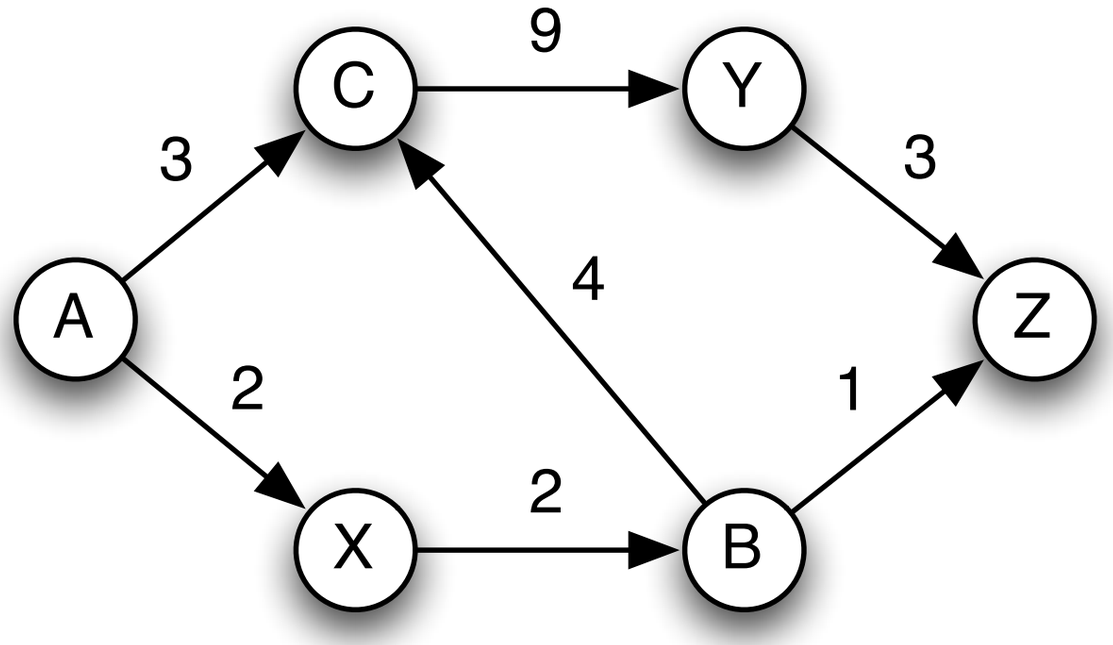
Application

- Considérons un réseau de production de marchandises ou chaque noeud d'un réseau est un producteur ou un consommateur. Une capacité de transport est donnée entre ces acteurs.

- Le problème consiste à déterminer les quantités de marchandises à transporter entre chaque producteur et consommateur.

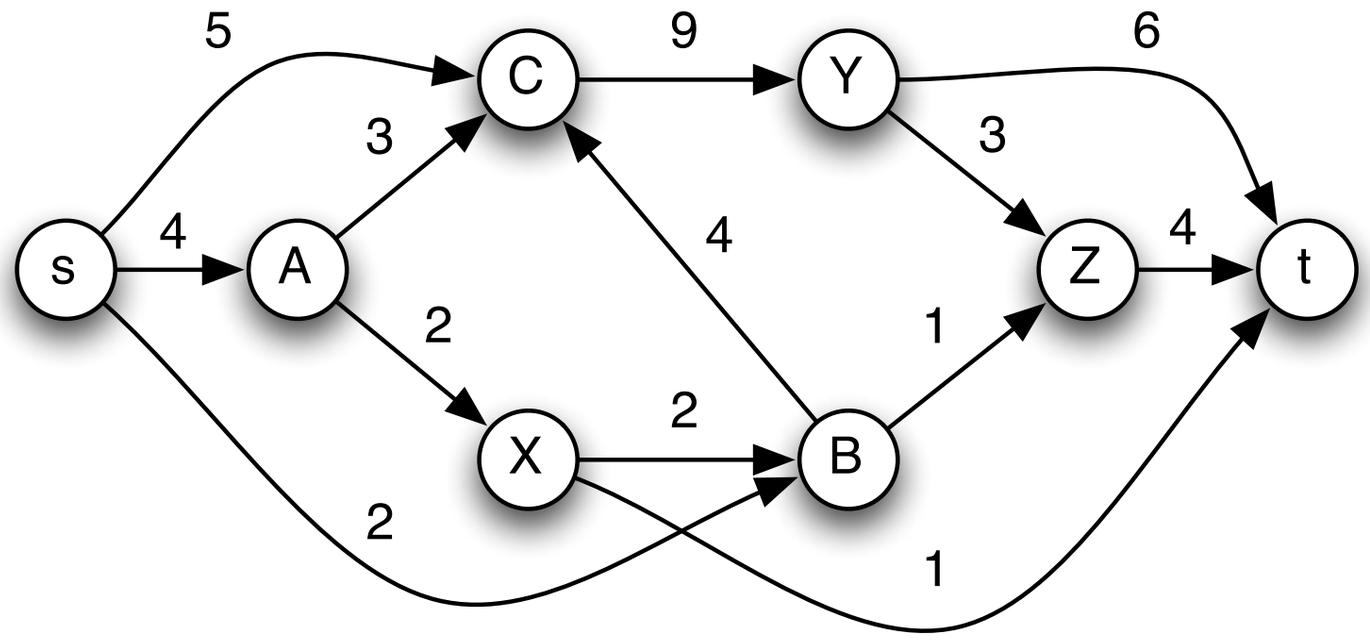
Producteur	Production
A	4
B	2
C	5

Consommateur	Consommation
X	1
Y	6
Z	4



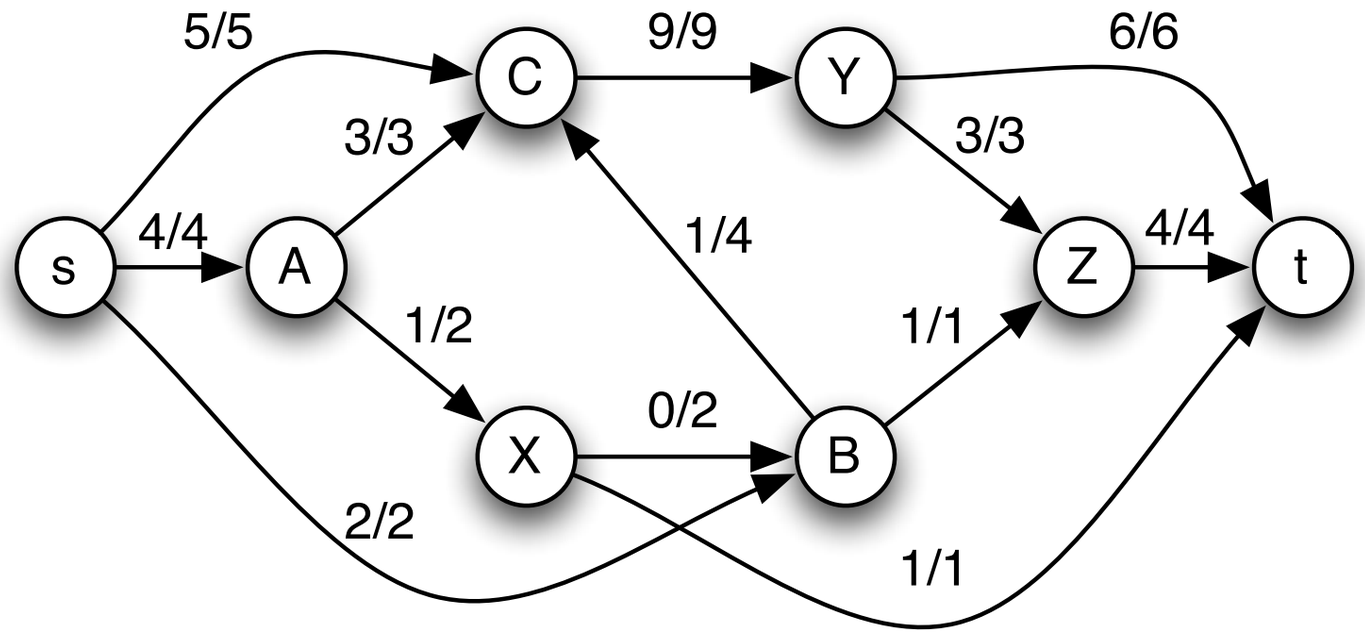
Solution

- On crée un noeud source que l'on relie à chaque producteur et un noeud puits relié à chaque consommateur.
- Les capacités de ces nouvelles arêtes représentent la production et la consommation de chacun.



Solution

- Le flot maximum dans ce nouveau graphe fournit la solution au problème.



Capacités minimales

- Dans certains problèmes, on veut pouvoir forcer une quantité minimale de flot à circuler sur une arête du graphe.
- En plus d'avoir une capacité maximale c(a, b) sur l'arête (a, b), on peut également aussi avoir une capacité minimale l(a,b).

$$l(a,b) \leq f(a,b) \leq c(a,b)$$

- Nous avons la relation l(a,b) = -c(b,a).
- L'algorithme de Ford-Fulkerson ne peut pas directement être utilisé sur ce problème puisque le flot nul n'est pas nécessairement un flot valide.

Graphe résiduel

- Supposons que nous avons un flot valide f.
- La définition du graphe résiduelle demeure inchangée.

$$(a,b) \in E_f \iff f(a,b) < c(a,b)$$

$$c_f(a,b) = c(a,b) - f(a,b)$$

- Notez toutefois que puisque l(a,b) = -c(b,a), si nous avons f(a,b) = l(a,b), l'arête (b, a) ne fera pas partie du graphe résiduel.
- **Démonstration**:

$$f(a,b) = l(a,b) \iff f(b,a) = -l(a,b) = c(b,a)$$

- **Conséquence**: on ne peut pas reprendre une unité de flot d'une arête où le flot est déjà à son minimum.

Chemin augmentant

- Avec cette *nouvelle* définition de graphe résiduel qui empêche de reprendre du flot à une arête dont le flot est déjà minimal, on peut appliquer l'algorithme de Ford-Fulkerson afin de trouver un flot maximal.
- Cependant, l'algorithme de Ford-Fulkerson requiert un flot valide qui n'est pas nécessairement le flot nul.
- **Question** : Comment trouver un flot initial valide?

Flot initial valide

- Nous modifions le flot de la façon suivante. D'abord, nous ajoutons une arête entre le puits et la source avec les capacités l(t, s) = 0 et c(t, s) = ∞.
- La contrainte de conservation des flots s'applique donc sur tous les noeuds, y compris la source et le puits.

$$\sum_{(a,b)\in E} f(a,b) - \sum_{(b,a)\in E} f(b,a) = 0 \qquad \forall\, a \in V$$

$$l(a,b) \leq f(a,b) \leq c(a,b)$$

- Pour résoudre ces équations, nous procédons au changement de variable f'(a,b) = f(a,b) - l(a,b).

Flot initial valide

$$\sum_{(a,b)\in E} (f'(a,b) + l(a,b)) - \sum_{(b,a)\in E} (f'(b,a) + l(b,a)) = 0 \qquad \forall\, a \in V$$

$$\sum_{(a,b)\in E} f'(a,b) - \sum_{(b,a)\in E} f'(b,a) = \sum_{(b,a)\in E} l(b,a) - \sum_{(a,b)\in E} l(a,b) \quad \forall\, a \in V$$

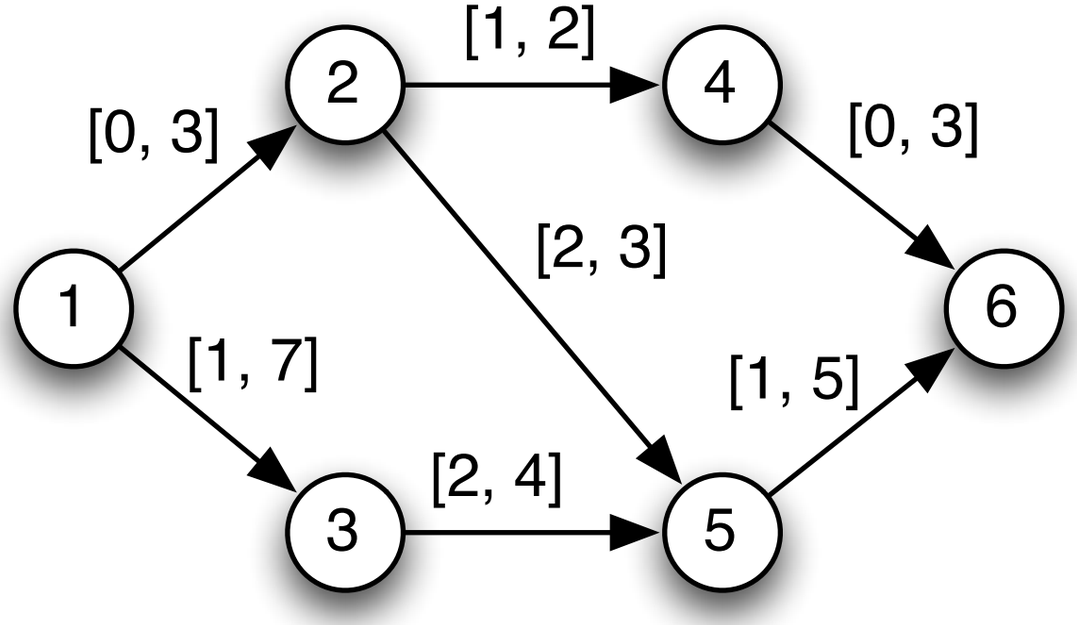
- Quant aux variables f'(a,b), elles sont bornées par ces inégalités.

$$0 \leq f'(a,b) \leq c(a,b) - l(a,b)$$

- Ces contraintes sont exactement celles du problème de production et de consommation du flot.
- On peut donc le résoudre en ajoutant une nouvelle source et un nouveau puits.

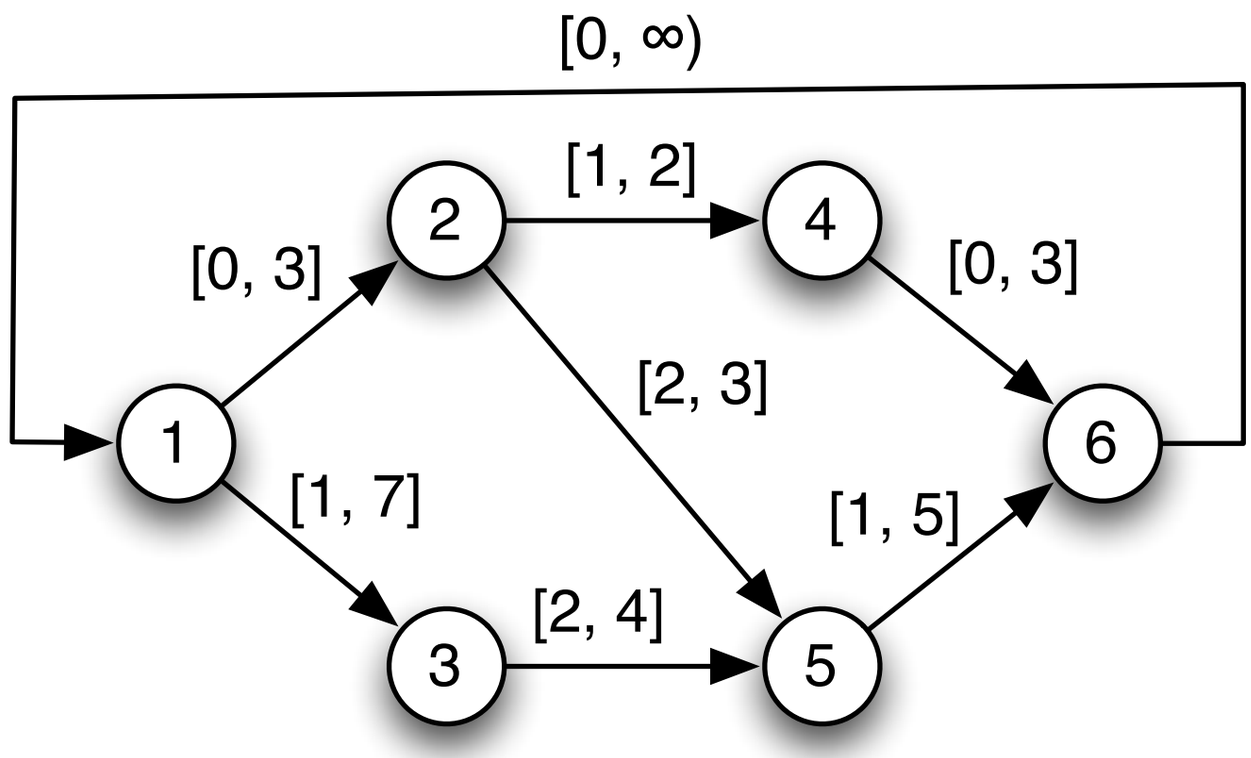
Exemple

Flot avec capacités minimales



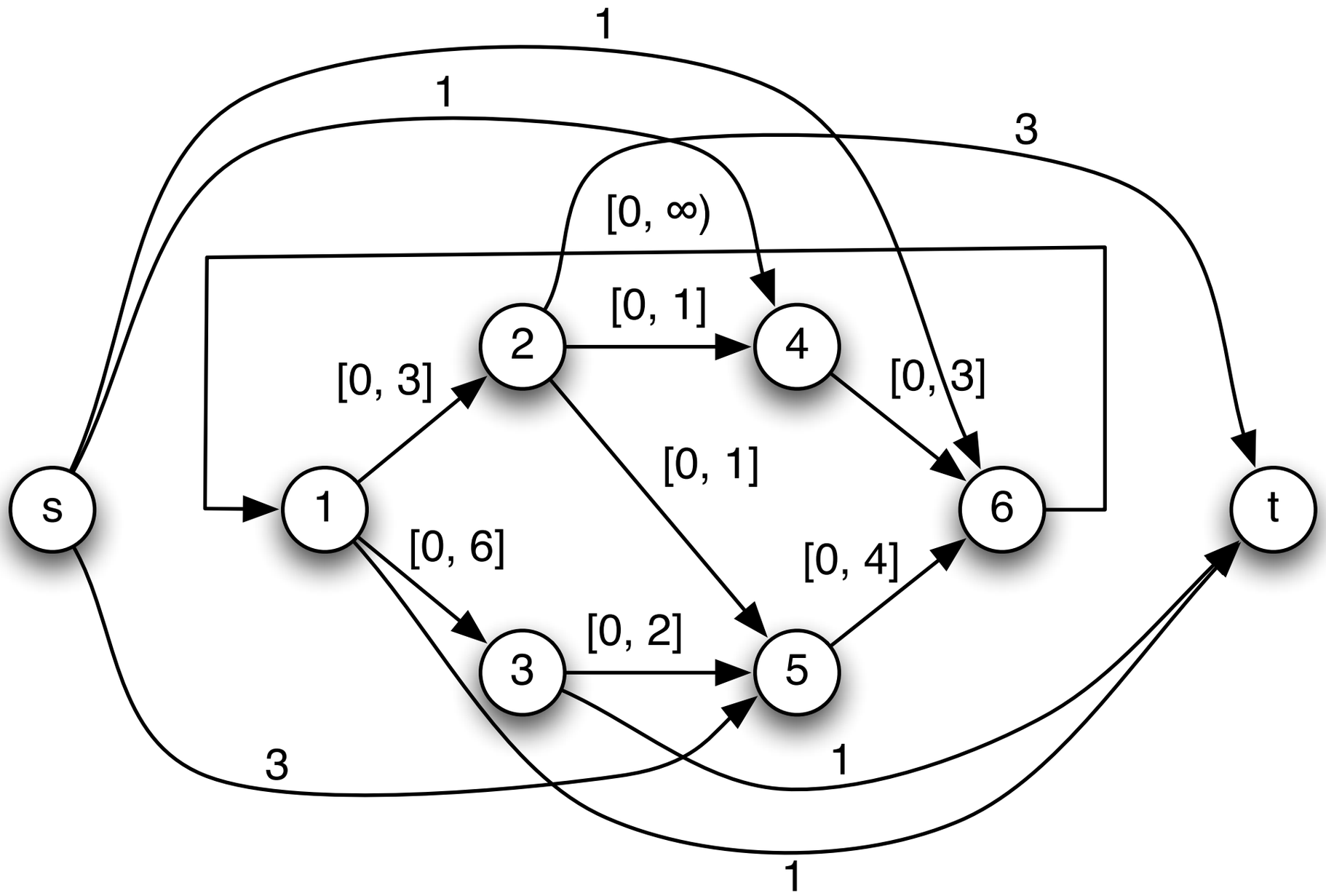
Exemple

Flot sans source ni puits



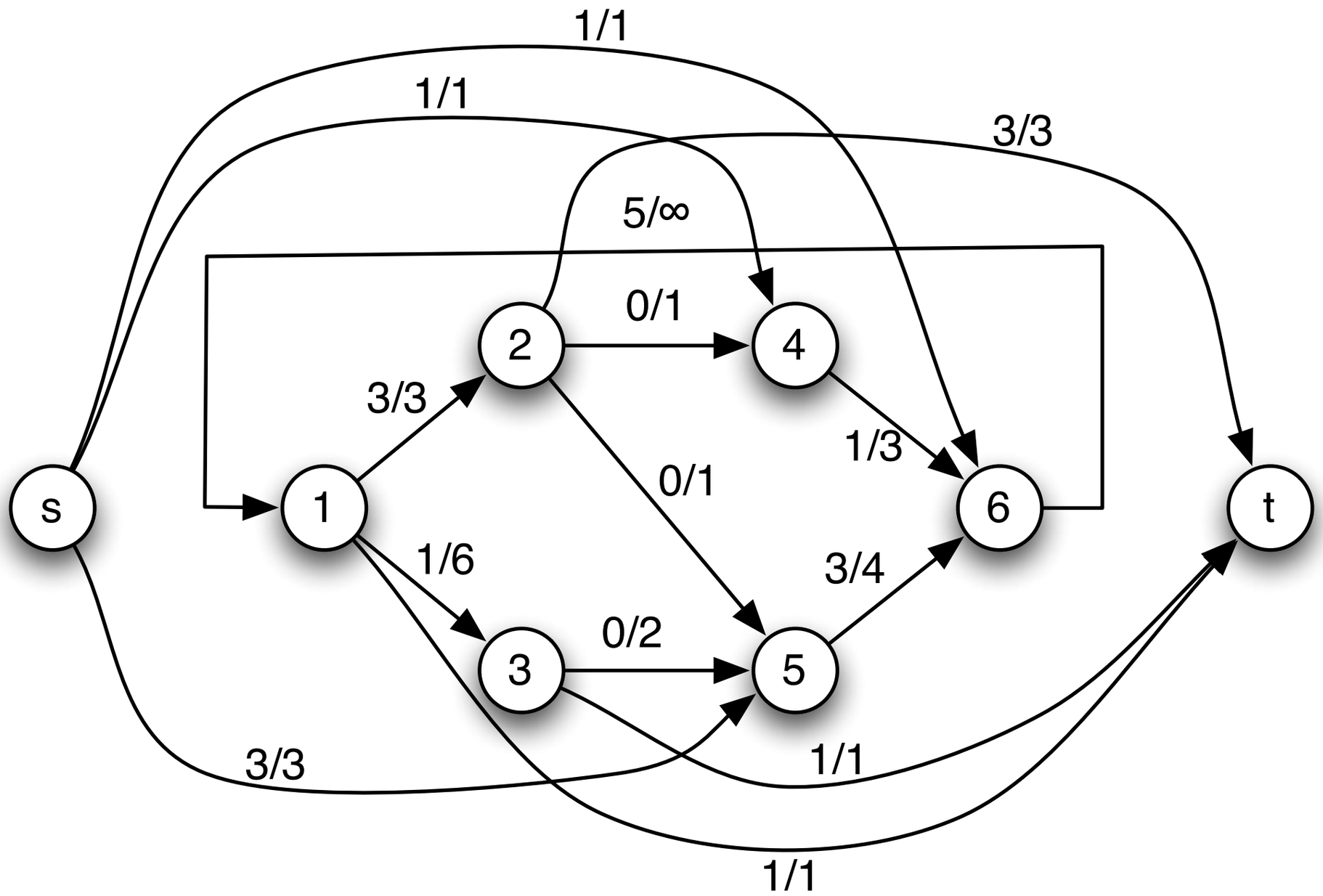
Exemple

Flot sans capacités minimales



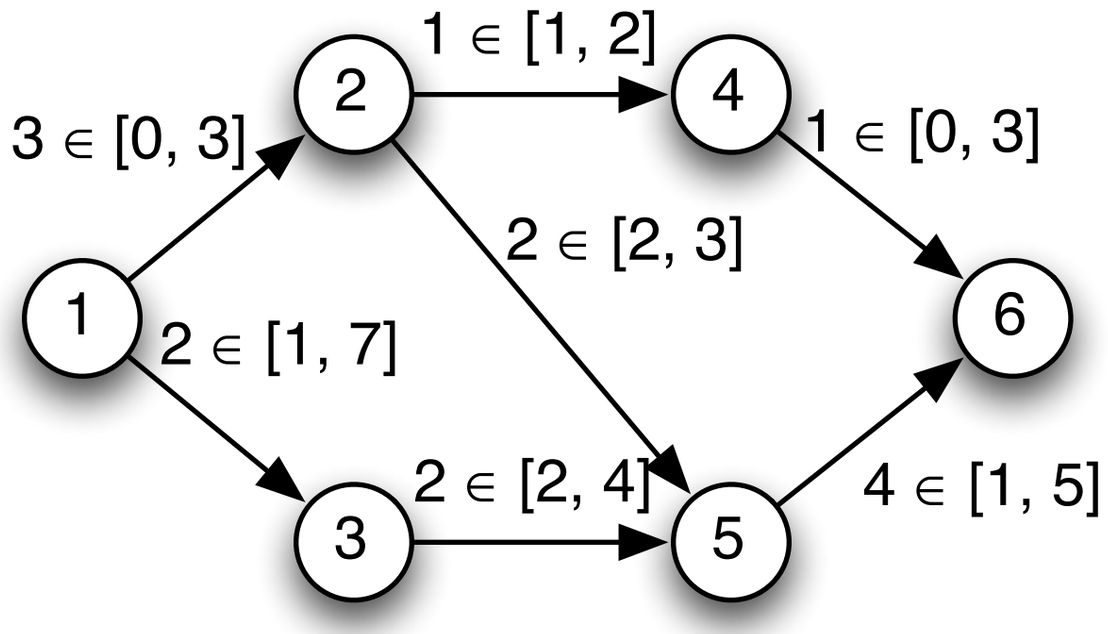
Exemple

Calcul du flot maximum



Exemple

Conversion en flot initial



Ce flot est valide, mais il n'est pas maximum. On peut augmenter sa valeur de flot en utilisant l'algorithme de Ford-Fulkerson.

Conclusion

- Nous avons étudié le problème du flot maximum dans un graphe. Ce problème est un problème de maximisation.
- L'algorithme de Ford-Fulkerson procède en cherchant des chemins augmentant dans le graphe résiduel.
- Nous avons vu le problème de la coupe minimum d'un graphe. Ce problème est un problème de minimisation.
- Le problème du flot maximum et de la coupe minimum sont équivalent dans le sens que la valeur de la coupe est égale à la valeur du flot.

Conclusion

- Nous avons vu comment un flot dans un graphe peut filtrer la contrainte All-Different.
- La variante où il y a des noeuds producteurs et consommateurs peut être ramenée au problème avec un seul puits et une seule source.
- On peut utiliser Ford-Fulkerson pour trouver un flot valide à la variante où les arêtes ont des capacités minimales.

Références

- Le livre principal est « Introduction to algorithms » par Thomas H. Cormen, Charles E. Leiserson, Ronald Rivest et Clifford Stein. Third edition.
- Fouille en profondeur: Section 22.3
- Problème du flot maximum et de la coupe minimum: Section 26
- Algorithme de Kosaraju: Section 22.5
- Algorithme de Régin: Handbook of constraint programming, chapitre « Global Constraints », section « Complete Filtering Algorithms », sous-section « The Alldifferent Constraint ».
- Flots avec production et consommation du flot et/ou capacités minimales: Network Flows, Theory, Algorithm, and Applications par Ravindra K. Ahuja, Thomas L. Magnanti et James B. Orlin. 1993. Application 6.1 et section 6.7.