# Algorithme 2

## Graphes orientés

Louis-Claude Canon
louis-claude.canon@univ-fcomte.fr

Licence 2 Informatique – Semestre 4

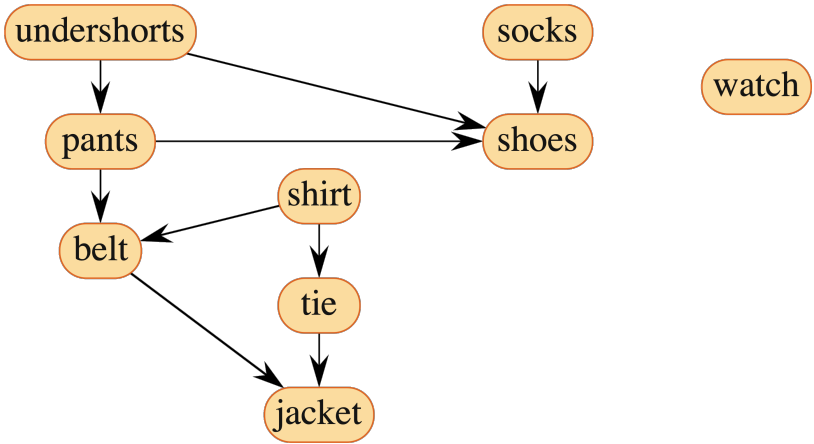
# Plan

# Plan

# Définition

- Certains graphes orientés peuvent être sans circuit (*Directed Acyclic Graph*).
- C'est le cas des graphes de dépendances où l'on cherche à trouver dans quel ordre réaliser des tâches qui dépendent les unes des autres (par exemple, les cibles d'un Makefile).
- Un *tri topologique* d'un graphe orienté sans circuit est un ordre sur les sommets qui respecte la direction des arcs : pour chaque arc $(u, v) \in E$, $u$ apparaît avant $v$ dans le tri topologique.

# Exemple



# Tri-topologique

- L'algorithme Tri-topologique s'appuie sur le parcours en profondeur.
- Ce parcours identifie rapidement quel sommet devrait être positionné en dernière position.
- On construit ainsi le tri topologique en partant de la fin.
- Dès que le parcours finit le traitement d'un sommet, celui-ci est positionné juste avant dans le tri topologique en construction.

---

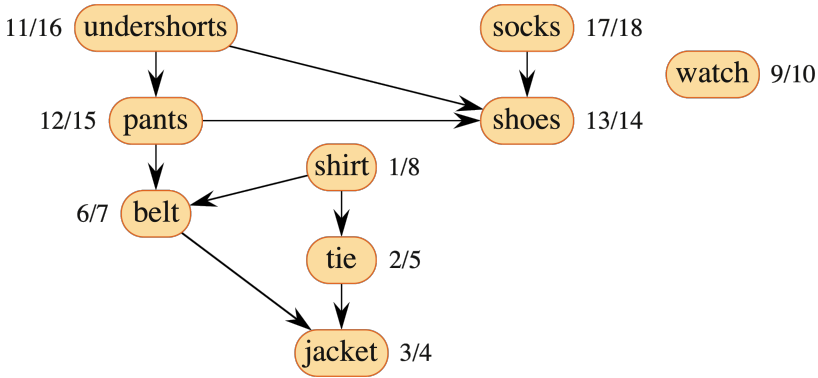
Tri-Topologique($G$)

---

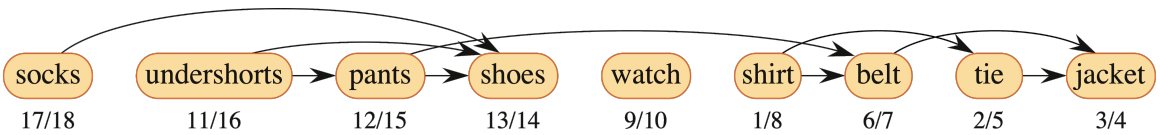
appeler PP($G$) pour calculer les dates de fin de traitement $v.f$
à la fin du traitement d'un sommet, on l'insère en début d'une liste chaînée
**retourner** la liste chaînée des sommets

---

# Exemple


11/16
undershorts
socks
17/18
watch
9/10
12/15
pants
shoes
13/14
shirt
1/8
6/7
belt
tie
2/5
jacket
3/4

# Exemple


socks
undershorts
pants
shoes
watch
shirt
belt
tie
jacket
17/18
11/16
12/15
13/14
9/10
1/8
6/7
2/5
3/4

# Preuve de validité (invariant)

**Invariant de boucle** Au début de chaque itération, la liste chaînée des sommets est un tri topologique partiel et chaque sommet inclus n'a pas d'arc vers des sommets non inclus.

**Initialisation** Comme la liste est vide, elle respecte la propriété de tri topologique.

**Conservation** Lorsqu'on ajoute un sommet, aucun des sommets de la liste chaînée n'avait d'arc vers lui, donc la nouvelle liste chaînée avec ce sommet en première position est un tri topologique. D'autre part, comme il finit d'être traité, il n'a pas d'arc vers aucun des sommets non inclus.

**Terminaison** À la fin, tous les sommets sont inclus dans la liste chaînée qui est un tri topologique.

# Preuve de validité (invariant)

**Invariant de boucle** Au début de chaque itération, la liste chaînée des sommets est un tri topologique partiel et chaque sommet inclus n'a pas d'arc vers des sommets non inclus.

**Initialisation** Comme la liste est vide, elle respecte la propriété de tri topologique.

**Conservation** Lorsqu'on ajoute un sommet, aucun des sommets de la liste chaînée n'avait d'arc vers lui, donc la nouvelle liste chaînée avec ce sommet en première position est un tri topologique. D'autre part, comme il finit d'être traité, il n'a pas d'arc vers aucun des sommets non inclus.

**Terminaison** À la fin, tous les sommets sont inclus dans la liste chaînée qui est un tri topologique.

# Preuve de validité (invariant)

**Invariant de boucle** Au début de chaque itération, la liste chaînée des sommets est un tri topologique partiel et chaque sommet inclus n'a pas d'arc vers des sommets non inclus.

**Initialisation** Comme la liste est vide, elle respecte la propriété de tri topologique.

**Conservation** Lorsqu'on ajoute un sommet, aucun des sommets de la liste chaînée n'avait d'arc vers lui, donc la nouvelle liste chaînée avec ce sommet en première position est un tri topologique. D'autre part, comme il finit d'être traité, il n'a pas d'arc vers aucun des sommets non inclus.

**Terminaison** À la fin, tous les sommets sont inclus dans la liste chaînée qui est un tri topologique.

# Preuve de validité (invariant)

**Invariant de boucle** Au début de chaque itération, la liste chaînée des sommets est un tri topologique partiel et chaque sommet inclus n'a pas d'arc vers des sommets non inclus.

**Initialisation** Comme la liste est vide, elle respecte la propriété de tri topologique.

**Conservation** Lorsqu'on ajoute un sommet, aucun des sommets de la liste chaînée n'avait d'arc vers lui, donc la nouvelle liste chaînée avec ce sommet en première position est un tri topologique. D'autre part, comme il finit d'être traité, il n'a pas d'arc vers aucun des sommets non inclus.

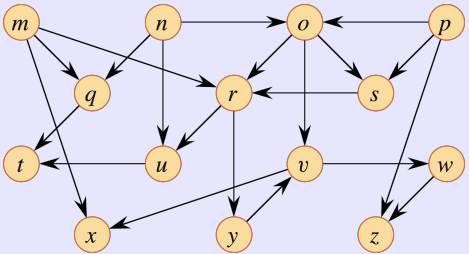
**Terminaison** À la fin, tous les sommets sont inclus dans la liste chaînée qui est un tri topologique.

# Analyse de complexité

- Le parcours en profondeur prend un temps $\Theta(V + E)$.
- La construction de la liste chaînée en partant de la fin représente $|V|$ itérations avec une complexité en temps en $\Theta(1)$ pour chaque itération.
- La complexité en temps de Tri-Topologique est $\Theta(V + E)$.

## Question

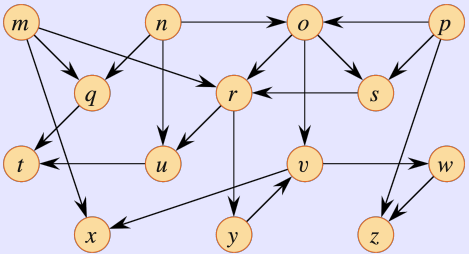
Quel est l'ordre dans lequel TRI-TOPOLOGIQUE trie les sommets quand on l'exécute sur le graphe suivant ? On supposera que les sommets sont toujours considérés dans l'ordre alphabétique.




1. $m, q, t, r, u, y, v, w, z, x, n, o, s, p$
2. $p, n, o, s, m, r, y, v, x, w, z, u, q, t$
3. $m, n, p, o, q, s, r, u, t, y, v, w, x, z$
4. $p, n, m, o, s, r, q, u, t, y, v, w, x, z$

## Question

Quel est l'ordre dans lequel TRI-TOPOLOGIQUE trie les sommets quand on l'exécute sur le graphe suivant ? On supposera que les sommets sont toujours considérés dans l'ordre alphabétique.




1. $m, q, t, r, u, y, v, w, z, x, n, o, s, p$
2. $p, n, o, s, m, r, y, v, x, w, z, u, q, t$ ✓
3. $m, n, p, o, q, s, r, u, t, y, v, w, x, z$
4. $p, n, m, o, s, r, q, u, t, y, v, w, x, z$

# Plan

# Contexte

- On utilise des graphe orienté sans circuit (DAG) pour représenter des planifications de projets.
- Chaque arc représente une activité à réaliser et son poids sa durée.
- Le *chemin critique* correspond aux activités qui déterminent le temps minimal pour réaliser l'ensemble du projet (celles qu'il ne faut pas retarder).
- L'algorithme du calcul du plus court chemin est une bonne base pour déterminer ce chemin critique.

# Rappel du plus court chemin

- On calcule la plus courte distance $v.d$ de l'origine à chaque sommet $v$.
- SOURCE-UNIQUE-INITIALISATION initialise la distance à 0 pour l'origine et à $\infty$ pour chaque autre sommet.
- RELÂCHER réduit cette distance en considérant un voisin donné.
- En considérant les sommets dans un ordre qui suit un tri topologique, on peut parcourir chaque sommet une seule fois.

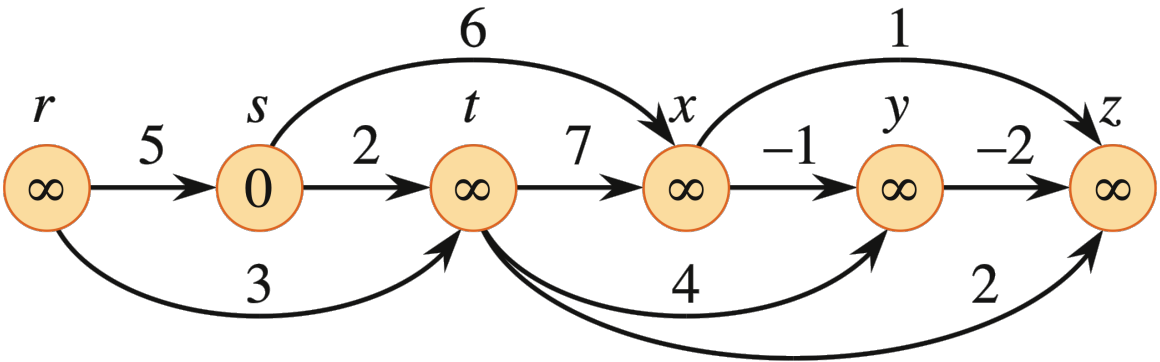
# Pseudo-code de PCC-GSS

PCC-GSS$(G, w, s)$

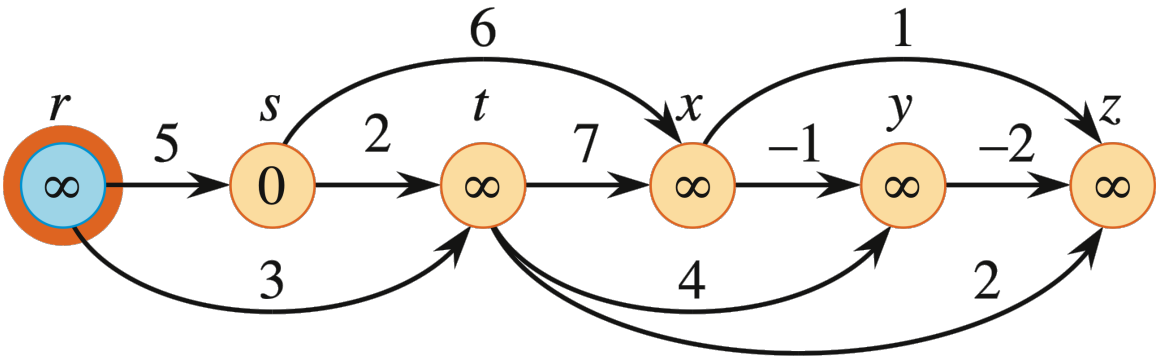
TRI-TOPOLOGIQUE$(G)$
SOURCE-UNIQUE-INITIALISATION$(G, s)$
**pour** chaque sommet $u \in G.V$ **faire**
  **pour** chaque sommet $v \in G.adj[u]$ **faire**
    RELÂCHER$(u, v, w)$

- Pour chaque appel à RELÂCHER$(u, v)$, $u$ est avant $v$ dans le tri topologique.
- La structure de DAG simplifie l'algorithme en permettant un parcours unique.

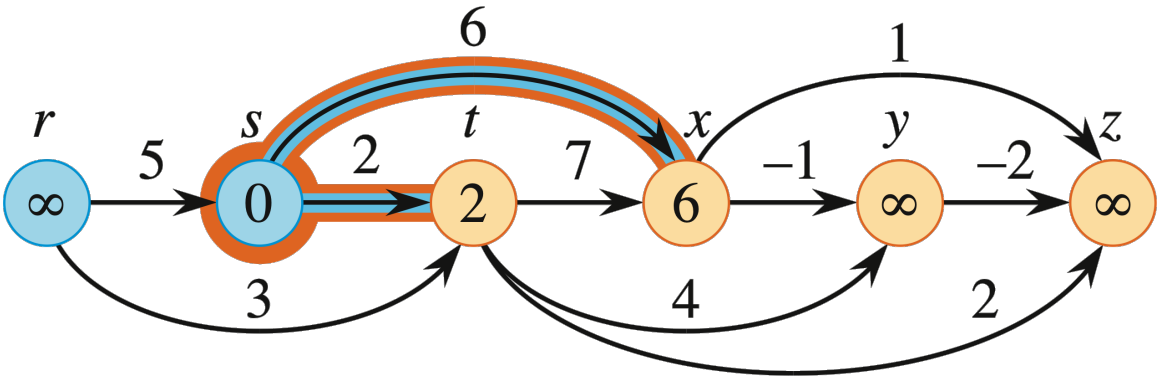
# Exemple



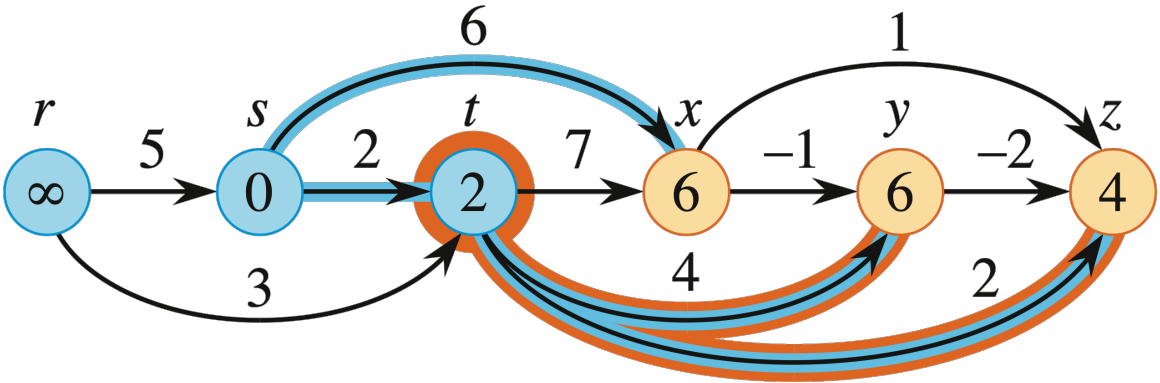
# Exemple



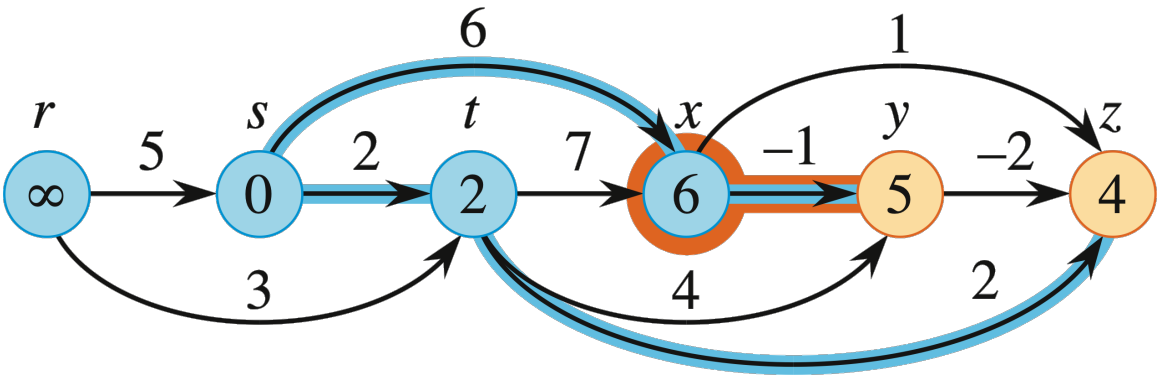
# Exemple



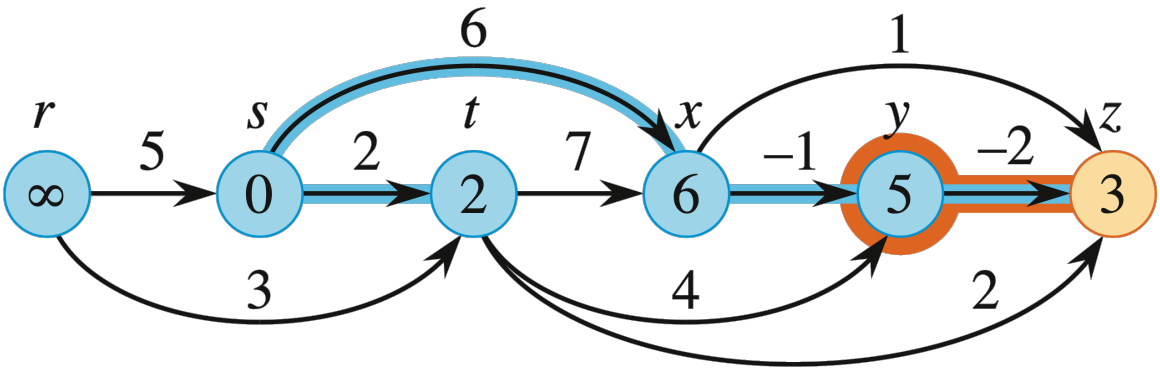
# Exemple



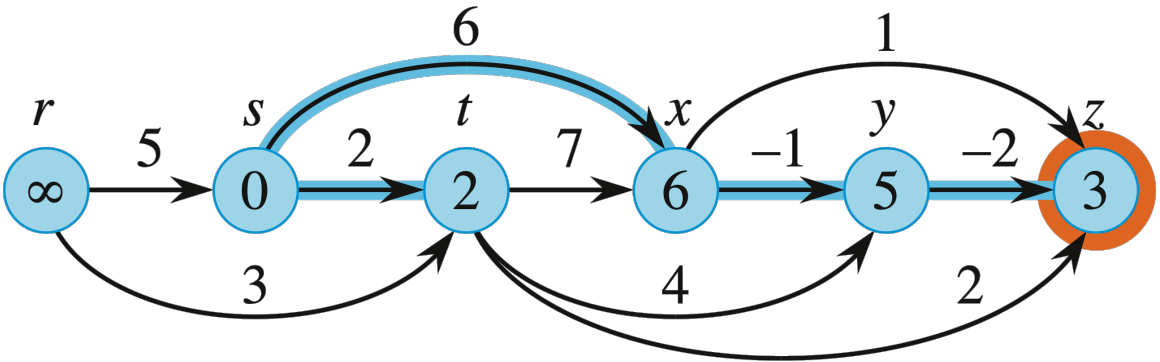
# Exemple



# Exemple



# Exemple



# Preuve de validité (invariant)

**Invariant de boucle** Au début de chaque itération, $v.d$ est la plus petite distance des sommets $v$ à partir de l'origine en ne passant que par ceux déjà parcourus.

**Initialisation** $v.d$ est initialement à l'infini sauf l'origine et c'est effectivement la distance à partir de l'origine sans passer par un sommet.

**Conservation** Lorsqu'on parcourt un sommet, on relâche ses voisins et on réduit donc la distance de tous les sommets à partir de l'origine en ne passant que par ceux déjà parcouru, y compris ce nouveau sommet.

**Terminaison** À la fin, $v.d$ représente bien le plus court chemin.

# Preuve de validité (invariant)

**Invariant de boucle** Au début de chaque itération, $v.d$ est la plus petite distance des sommets $v$ à partir de l'origine en ne passant que par ceux déjà parcourus.

**Initialisation** $v.d$ est initialement à l'infini sauf l'origine et c'est effectivement la distance à partir de l'origine sans passer par un sommet.

**Conservation** Lorsqu'on parcourt un sommet, on relâche ses voisins et on réduit donc la distance de tous les sommets à partir de l'origine en ne passant que par ceux déjà parcouru, y compris ce nouveau sommet.

**Terminaison** À la fin, $v.d$ représente bien le plus court chemin.

# Preuve de validité (invariant)

**Invariant de boucle** Au début de chaque itération, $v.d$ est la plus petite distance des sommets $v$ à partir de l'origine en ne passant que par ceux déjà parcourus.

**Initialisation** $v.d$ est initialement à l'infini sauf l'origine et c'est effectivement la distance à partir de l'origine sans passer par un sommet.

**Conservation** Lorsqu'on parcourt un sommet, on relâche ses voisins et on réduit donc la distance de tous les sommets à partir de l'origine en ne passant que par ceux déjà parcouru, y compris ce nouveau sommet.

**Terminaison** À la fin, $v.d$ représente bien le plus court chemin.

# Preuve de validité (invariant)

**Invariant de boucle** Au début de chaque itération, $v.d$ est la plus petite distance des sommets $v$ à partir de l'origine en ne passant que par ceux déjà parcourus.

**Initialisation** $v.d$ est initialement à l'infini sauf l'origine et c'est effectivement la distance à partir de l'origine sans passer par un sommet.

**Conservation** Lorsqu'on parcourt un sommet, on relâche ses voisins et on réduit donc la distance de tous les sommets à partir de l'origine en ne passant que par ceux déjà parcouru, y compris ce nouveau sommet.

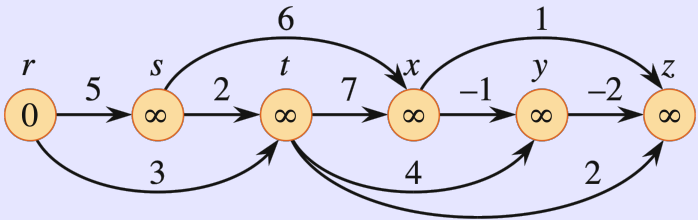
**Terminaison** À la fin, $v.d$ représente bien le plus court chemin.

# Analyse de complexité

- Le tri topologique est en $\Theta(V + E)$.
- L'initialisation est en temps $\Theta(V)$.
- Les boucles permettent de parcourir tous les arcs et d'appliquer le relâchement, qui est en $\Theta(1)$ pour chacune.
- La complexité en temps de $\mathrm{PCC\text{-}GSS}$ est donc $\Theta(V + E)$.

## Question

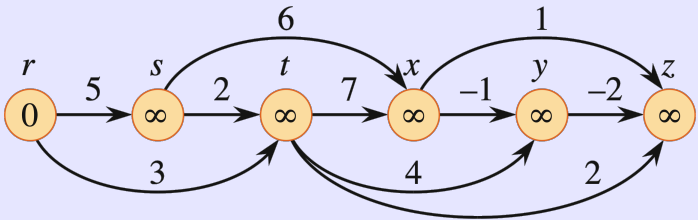
Quel est le résultat de PCC-GSS pour le graphe suivant lorsque $r$ est l'origine ?




1. $(0, 5, 3, 11, 7, 4)$
2. $(0, 5, 3, 10, 7, 5)$
3. $(0, 5, 3, 11, 7, 5)$
4. $(0, 5, 3, 10, 7, 4)$

## Question

Quel est le résultat de PCC-GSS pour le graphe suivant lorsque $r$ est l'origine ?




1. $(0, 5, 3, 11, 7, 4)$
2. $(0, 5, 3, 10, 7, 5)$ ✓
3. $(0, 5, 3, 11, 7, 5)$
4. $(0, 5, 3, 10, 7, 4)$

# Plan

# Résumé de la première moitié

- Les tas et les files de priorités : extraction du maximum, augmentation, insertion.
- Les forêts d'ensembles disjoints.
- Les arbres de recherche : arbre rouge-noir, B-arbre, arbre de rangs et arbre d'intervalles.

# Récapitulatif sur les algorithmes

| Séance | Problème | Algorithme | Complexité |
|---|---|---|---|
| CM7 | Parcours | PL | $O(V+E)$ |
| CM7 | Parcours | PP | $\Theta(V+E)$ |
| CM8 | Arbres couvrants | ACM-Kruskal | $O(E \log E)$ |
| CM8 | Arbres couvrants | ACM-Prim | $O(E \log V)$ [1] |
| CM9 | Plus court chemin | Bellman-Ford | $O(V(V+E))$ |
| CM9 | Plus court chemin | Dijkstra | $O((V+E) \log V)$ [1] |
| CM10 | Flot maximum | Ford-Fulkerson | $O(VE^2)$ [2] |
| CM11 | Couplage | Hopcroft-Karp | $O(\sqrt{V}E)$ |
| CM11 | Couplage | Gale-Shapley | $O(V^2)$ |
| CM12 | Parcours | Tri-Topologique | $\Theta(V+E)$ |
| CM12 | Plus court chemin | PCC-GSS | $\Theta(V+E)$ |

1. Ou $O(E + V \log V)$ avec un tas de Fibonacci.
2. Avec Edmonds-Karp.

# Résumé des principes

- Extension par la largeur/frontière : PL, ACM-Prim et Dijkstra.
- Chemin améliorant : Ford-Fulkerson (Edmonds-Karp) et Hopcroft-Karp.

# Plan

# Épreuve de TP : approche incrémentale

- Le temps de développement est toujours limité.
- Le développement incrémental permet d'optimiser l'ordre des opérations pour maximiser le résultat exploitable (on cherche à éviter avant tout le "blob" de code qu'il sera plus rapide de recommencer que de rattraper).
- On ordonne par *incrément* : un petit regroupement de fonctionnalités le plus unitaire possible (quel est le minimum qui permet d'avoir un résultat exploitable ?).
- On réalise tous les développements associés (conception, implémentation, test, documentation, etc.) en laissant le reste de coté.
- À chaque moment, on a toujours un ou plusieurs incréments qui sont fonctionnels, ça fournit une bonne base pour le futur.

# Épreuve de TP : approche incrémentale

- Le temps de développement est toujours limité.
- Le développement incrémental permet d'optimiser l'ordre des opérations pour maximiser le résultat exploitable (on cherche à éviter avant tout le "blob" de code qu'il sera plus rapide de recommencer que de rattraper).
- On ordonne par *incrément* : un petit regroupement de fonctionnalités le plus unitaire possible (quel est le minimum qui permet d'avoir un résultat exploitable ?).
- On réalise tous les développements associés (conception, implémentation, test, documentation, etc.) en laissant le reste de coté.
- À chaque moment, on a toujours un ou plusieurs incréments qui sont fonctionnels, ça fournit une bonne base pour le futur.

# Épreuve de TP : approche incrémentale

- Le temps de développement est toujours limité.
- Le développement incrémental permet d'optimiser l'ordre des opérations pour maximiser le résultat exploitable (on cherche à éviter avant tout le "blob" de code qu'il sera plus rapide de recommencer que de rattraper).
- On ordonne par *incrément* : un petit regroupement de fonctionnalités le plus unitaire possible (quel est le minimum qui permet d'avoir un résultat exploitable ?).
- On réalise tous les développements associés (conception, implémentation, test, documentation, etc.) en laissant le reste de coté.
- À chaque moment, on a toujours un ou plusieurs incréments qui sont fonctionnels, ça fournit une bonne base pour le futur.

# Épreuve de TP : approche incrémentale

- Le temps de développement est toujours limité.
- Le développement incrémental permet d'optimiser l'ordre des opérations pour maximiser le résultat exploitable (on cherche à éviter avant tout le "blob" de code qu'il sera plus rapide de recommencer que de rattraper).
- On ordonne par *incrément* : un petit regroupement de fonctionnalités le plus unitaire possible (quel est le minimum qui permet d'avoir un résultat exploitable ?).
- On réalise tous les développements associés (conception, implémentation, test, documentation, etc.) en laissant le reste de coté.
- À chaque moment, on a toujours un ou plusieurs incréments qui sont fonctionnels, ça fournit une bonne base pour le futur.

# Épreuve de TP : approche incrémentale

- Le temps de développement est toujours limité.
- Le développement incrémental permet d'optimiser l'ordre des opérations pour maximiser le résultat exploitable (on cherche à éviter avant tout le "blob" de code qu'il sera plus rapide de recommencer que de rattraper).
- On ordonne par *incrément* : un petit regroupement de fonctionnalités le plus unitaire possible (quel est le minimum qui permet d'avoir un résultat exploitable ?).
- On réalise tous les développements associés (conception, implémentation, test, documentation, etc.) en laissant le reste de coté.
- À chaque moment, on a toujours un ou plusieurs incréments qui sont fonctionnels, ça fournit une bonne base pour le futur.

# Exemple du mini-projet

- Ordre numéro 1 :
  1. implémentation de hamming
  2. implémentation de manhattan
  3. implémentation de neighbors
  4. implémentation de twin
  5. implémentation de solve
  6. test
- C'est au milieu du développement de solve que la contrainte de temps se fait sentir, les tests sont bâclés, il reste plein de bugs, la note est inférieure à 10.

- Ordre numéro 2 :
  1. implémentation de hamming
  2. implémentation de neighbors
  3. implémentation de solve
  4. test
- Le second incrément aurait consisté à rajouter la détection de la solvabilité et le troisième les optimisations et manhattan, mais par manque de temps, seul le premier incrément est réalisé et bien testé, la note peut aller jusqu'à 15.

# Bien commencer un développement logiciel

1. Lire tout le sujet une première fois.
2. Préparer les fichiers `.h` (copier-coller si nécessaire).
3. Préparer les fichiers `.c` avec des squelettes de fonction (test des pointeurs et retourner les valeurs par défaut).
4. Préparer le fichier `_tests.c` avec un test trivial.
5. Préparer le `Makefile`.
6. `make all`
7. Corriger toutes les erreurs, les avertissements, les accès et les fuites mémoires.
8. Choisir une fonction dans le sujet.
9. Implémenter la fonction.
10. Écrire le test de la fonction.
11. Recommencer à l'étape 6.

# Avant la soumission d'un code

Se garder du temps avant la fin pour vérifier les points suivants :

- Soumettre les fichiers **tels quels** (fichiers sources non compressés, non archivés, non renommés) sur le dépôt de devoir Moodle associé.
- S'assurer que ça compile sans avertissement, que le code est indenté, avec aucun accès mémoire invalide et le moins possible de fuites mémoires (`make all`).
- Relire le sujet une dernière fois pour vérifier le respect des consignes.

# Devoir surveillé

La méthode pour la conception d'algorithmes est la suivante :

1. comprendre l'énoncé ;
2. faire un exemple non trivial pour résoudre à la main ;
3. en l'absence d'idée d'algorithme, recommencer avec un exemple plus grand ;
4. identifier les étapes de l'algorithme trouvé ;
5. convertir en pseudo-code.

# Plan

# Résumé

## Contenu

- Certains algorithmes sont spécifiques aux graphes orientés.
- Le tri topologique permet d'ordonner les sommets d'un graphe orienté sans circuit (ou DAG).
- Le plus court chemin peut être calculé plus efficacement dans un DAG grâce au tri topologique.

## Prochaines échéances

- Projet-tournoi : à rendre la semaine du 26/5.
- DS 2 et épreuve TP : semaine du 26/5.
- Épreuve de seconde chance et restitution du projet-tournoi : semaine du 2/6.

# Résumé

## Contenu

- Certains algorithmes sont spécifiques aux graphes orientés.
- Le tri topologique permet d'ordonner les sommets d'un graphe orienté sans circuit (ou DAG).
- Le plus court chemin peut être calculé plus efficacement dans un DAG grâce au tri topologique.

## Prochaines échéances

- Projet-tournoi : à rendre la semaine du 26/5.
- DS 2 et épreuve TP : semaine du 26/5.
- Épreuve de seconde chance et restitution du projet-tournoi : semaine du 2/6.