INFO0501

STRUCTURES DE DONNÉES ET ALGORITHMES 2

COURS 5

GRAPHES
ARBRES COUVRANTS DE POIDS MINIMAL

UNIVERSITÉ DE REIMS CHAMPAGNE-ARDENNE

Pierre Delisle
Département de Mathématiques, Mécanique et Informatique
Octobre 2018

Plan de la séance

- Arbres couvrants de poids minimal
- Algorithme de Kruskal
  - Structure de données pour ensembles disjoints
- Algorithme de Prim

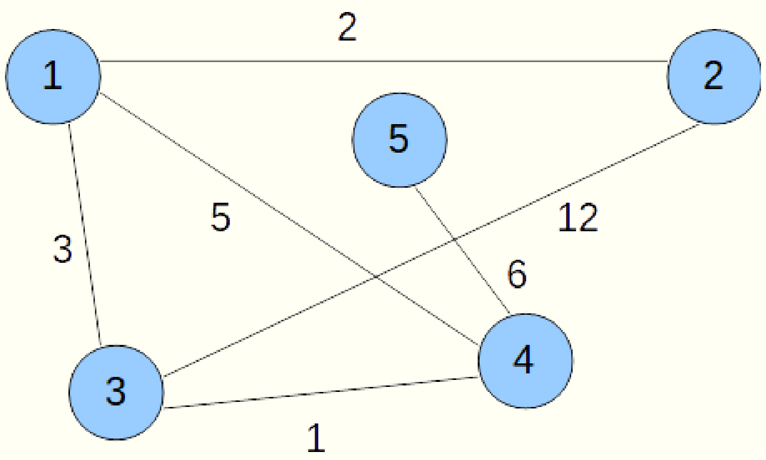
- Bibliographie
  - T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, "Algorithmique", 3e édition, Dunod, 2010

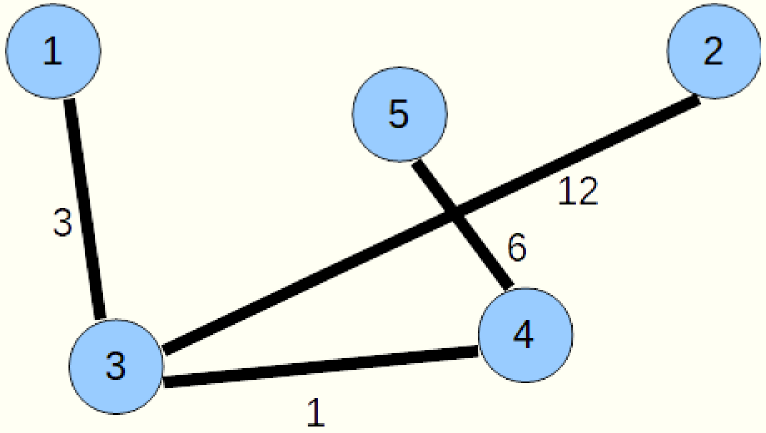
PROBLÈME DE L'ARBRE COUVRANT DE POIDS MINIMAL

Définition du problème

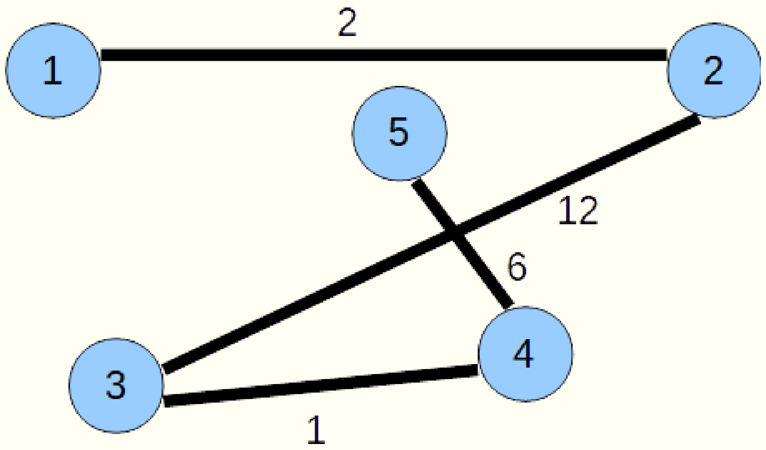
- Étant donné un graphe non orienté pondéré
- On cherche un graphe partiel de ce graphe
- … qui soit un arbre …
- … et qui soit de coût minimum

- Graphe partiel : l'arbre a pour ensemble de sommets tous les sommets du graphe initial
  - On dit que c'est un arbre couvrant
- En anglais : minimum spanning tree
- On suppose le graphe initial connexe





Arbre couvrant
(de poids 22)



Arbre couvrant
(de poids 21)

Applications

- Réseaux
  - On estime le coût des liaisons directes entres toutes les machines à relier
  - Puis on cherche à réaliser un réseau connexe à coût minimum
- Traitement d'images
  - On représente une image sous forme de pixels
  - On veut déterminer des régions dans l'image
  - Graphe : chaque pixel est un sommet à 8 voisins
  - On value les arêtes par la différence de gris
  - On construit l'arbre couvrant minimum, puis on sépare

Construction d'un arbre couvrant minimal

- Soit un graphe $G = (S, A)$
  - Non orienté
  - Connexe
  - Fonction de pondération p → attribue un poids $p(u, v)$ à chaque arête (u, v)
- Stratégies gloutonnes
  - On construit l'arbre arête par arête
  - … en ajoutant à chaque étape une arête appropriée de A

```
ACM-GÉNÉRIQUE (G, p)
    E = ∅
    Tant que E ne forme pas un arbre couvrant
        Trouver une arête (u, v) appropriée
        E = E ∪ (u, v)
    Retourner E
```

- 2 algorithmes principaux → Kruskal et Prim
  - Utilisent cette stratégie gloutonne
  - Utilisent une règle différente pour choisir l'arête (u, v) appropriée

ALGORITHME DE KRUSKAL

Principe de l'algorithme

- Pour déterminer un arbre couvrant de poids minimal d'un graphe connexe à $|S|$ sommets, ...
- ... on sélectionne les arêtes d'un graphe partiel initialement sans arête ...
- ... en itérant $|S|$ -1 fois l'opération suivante
  - Choisir une arête de poids minimal ne formant pas un cycle avec les arêtes précédemment choisies

- Exemple 1 : fonctionnement de l'algorithme de Kruskal

Mise en oeuvre

- Trier les arêtes par ordre de poids croissants
- Tant qu'on n'a pas retenu $|S| - 1$ arêtes
  - Considérer, dans l'ordre du tri, la 1ère arête non examinée
  - Si elle forme un cycle avec celles précédentes, la rejeter, sinon la garder
- Il reste à résoudre un problème
  - Comment déterminer si une arête choisie à l'étape i forme un cycle avec les arêtes précédemment choisies ?

Principe : gérer l'évolution des composantes connexes

- Pour qu'une arête (u,v) ferme un cycle, il faut que ses extrémités aient été précédemment reliées par une chaîne
  - Donc aient été dans une même composante connexe
- Nous allons donc gérer l'évolution des composantes connexes au fur et à mesure du choix des arêtes
  - Par tableau $\rightarrow$ plus simple mais moins efficace
  - Par ensembles disjoints $\rightarrow$ efficace

Gérer l'évolution des composantes connexes par tableau

- Initialement, le graphe ne contient aucune arête
  - Chaque sommet est une composante connexe
  - On initialise chaque sommet à son indice i
- Chaque fois qu'une arête $\{u, v\}$ est candidate
  - On compare les valeurs des indices de u et v
    - Si valeurs égales, u et v sont dans la même composante connexe $\rightarrow$ ajouter l'arête créerait alors un cycle donc on ne la retient pas
    - Si valeurs différentes, on garde l'arête $\{u, v\}$, on donne à l'indice de v la valeur de l'indice de u, ainsi que tout sommet d'indice v
    - Autrement dit, après sélection de l'arête $\{u,v\}$, tous les sommets de la composante connexe de u et de la composante connexe de v ne forment qu'une seule composante connexe et ont le même indice
- Exemple 2 : Gestion des composantes connexes par tableau

Algorithme de Kruskal

```
Trier les arêtes par poids croissants et les ranger dans tabAretes
POUR i de 1 à |S| FAIRE
    tabCC[i] ← i
cptArbre ← 0
cptAretes ← 1
TANT QUE cptArbre < |S| – 1
    {x,y} ← tabAretes[cptAretes]
    cptAretes ← cptAretes + 1
    SI tabCC[x] != tabCC[y]
        cptArbre ← cptArbre + 1
        tabArbre[cptArbre] ← {x,y}
        indCC ← tabCC[y]
        POUR i de 1 à |S|
            SI tabCC[i] = indCC ALORS
                tabCC[i] ← tabCC[x]
```

- *tabCC*
  - Tableau d'entiers associés aux sommets 1 à |*S*| qui tiendra à jour les indices de composantes connexes
- *tabAretes*
  - Tableau des arêtes, trié en ordre croissant de poids
- *tabArbre*
  - Tableau d'arêtes correspondant à l'arbre couvrant de poids minimum, de taille |*S*| - 1

Remarques

- Après la sélection d'une arête valide
  - On doit parcourir tout le tableau tabCC pour mettre à jour les indices de composantes connexes
  - $O(S)$
- On peut réduire ce temps par l'utilisation d'une **structure de données pour ensembles disjoints**

STRUCTURES DE DONNÉES POUR ENSEMBLES DISJOINTS

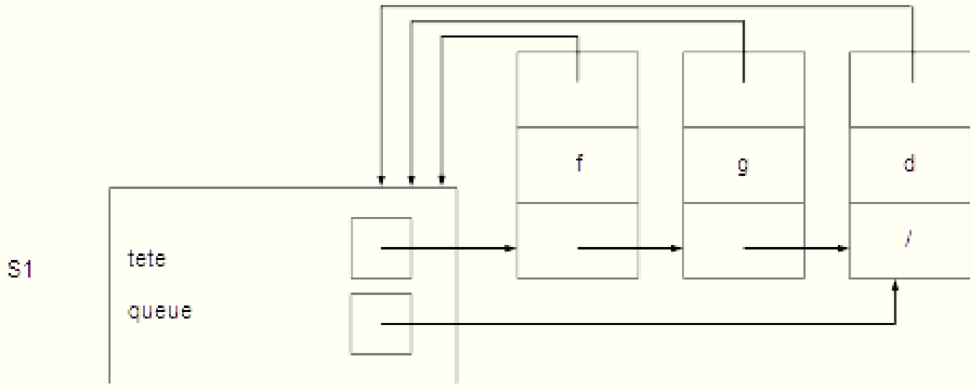
Ensembles disjoints

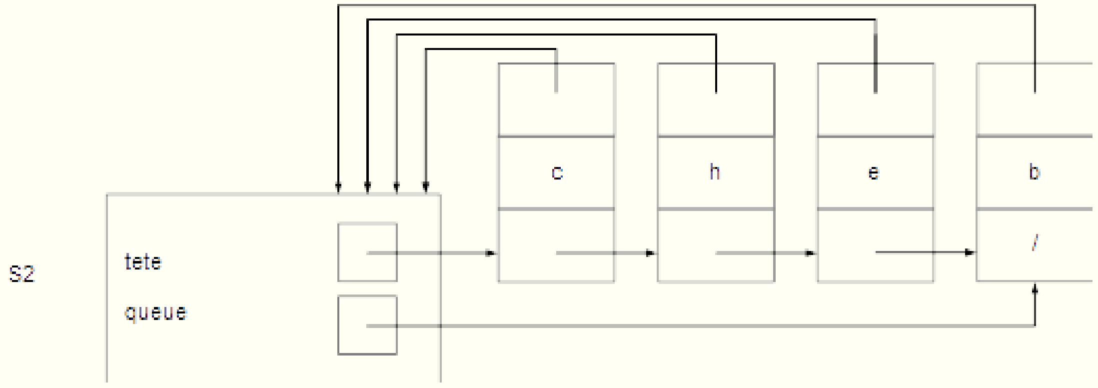
- On veut regrouper n éléments distincts dans une collection $S = \{S_1, S_2, \ldots\ S_k\}$ d'ensembles dynamiques disjoints
  - Chaque élément fait partie d'un seul ensemble
  - Chaque ensemble est identifié par un représentant
- Opérations sur les ensembles disjoints
  - CRÉER-ENSEMBLE (x)
  - TROUVER-ENSEMBLE (x)
  - UNION (x, y)

Opérations sur les ensembles disjoints

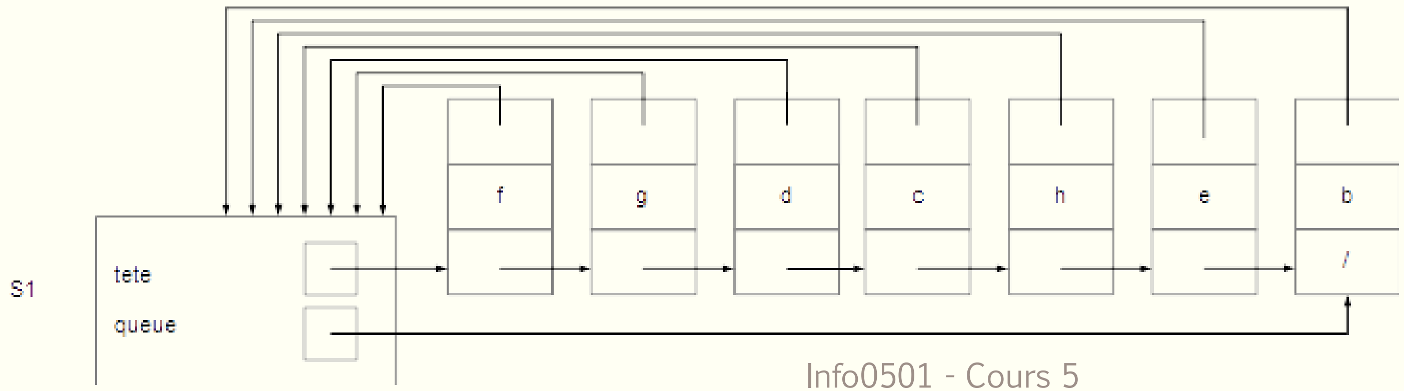
- CRÉER-ENSEMBLE (x)
  - À partir d'un pointeur sur un objet x
  - Crée un nouvel ensemble dont le seul membre (et le représentant) est x
  - Ensembles disjoints
    - x ne doit pas être déjà membre d'un autre ensemble
- TROUVER-ENSEMBLE (x)
  - Retourne un pointeur vers le représentant de l'ensemble contenant x
- UNION (x, y)
  - Réunit les ensembles dynamiques qui contiennent x et y…
  - … appelés S_x et S_y
  - … dans un nouvel ensemble qui est l'union de S_x et S_y
  - Le représentant du nouvel ensemble sera un élément quelconque d'un des 2 ensembles
  - Détruit les ensembles initiaux S_x et S_y
- Exemple 3
  - Application d'une structure de données d'ensembles disjoints
  - Détermination des composantes connexes d'un graphe non orienté

Représentation d'ensembles disjoints par listes chaînées



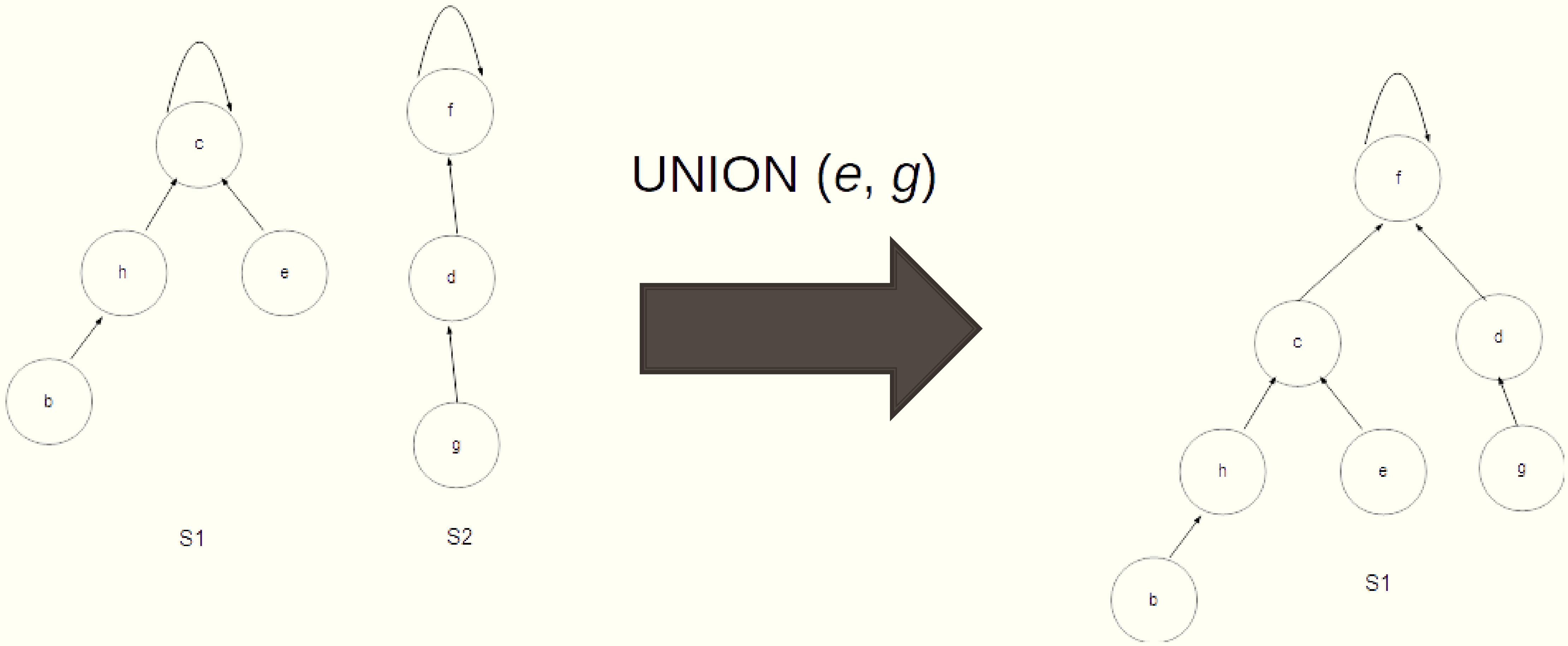


UNION (g, e)



- Temps d'exécution
  - CRÉER-ENSEMBLE(x) ?
    - $O(1)$
  - TROUVER-ENSEMBLE(x) ?
    - $O(1)$
  - UNION(x, y) ?
    - $O(n)$

Représentation d'ensembles disjoints par arbres



Algorithme de Kruskal avec utilisation d'ensembles disjoints

- On crée un ensemble disjoint pour chaque sommet de G
- On trie les arêtes de G
- Pour chaque arête prise par ordre croissant
  - Si l'origine et l'extrémité ne font pas partie du même ensemble, conserver l'arête et réunir les deux ensembles
- Les arêtes retenues constituent alors un arbre couvrant de poids minimal de G

ALGORITHME DE PRIM

Principe de l'algorithme

- Permet de trouver un arbre couvrant de poids minimum sans devoir trier les arêtes
- On étend, de proche en proche, un arbre couvrant des parties des sommets du graphe
  - En atteignant un sommet supplémentaire à chaque étape …
  - … en prenant l'arête la plus légère parmi celles qui joignent l'ensemble des sommets déjà couverts …
  - … à l'ensemble des sommets non encore couverts
- Exemple 4 : Fonctionnement de l'algorithme

Algorithme de Prim

```
tabArbre ← {x0} (x0 étant un sommet quelconque)
pivot ← {x0}
POUR tout sommet u autre que x0
    d(u) ← +∞
POUR i de 1 à |S| – 1
    POUR tout sommet z voisin de pivot non dans tabArbre
        SI p(pivot, z) < d(z)
            proche(z) ← pivot
            d(z) ← p(pivot,z)
    Parmi les sommets qui ne sont pas dans tabArbre, déterminer
    un sommet pivot qui réalise le minimum des valeurs de d
    Ajouter pivot et l'arête {pivot, proche(pivot)} à tabArbre
```

- *tabArbre* : contenant l'arbre en construction
- On note $p(u, v)$ le poids de l'arête (u, v)
- À chaque étape, on ajoute à *tabArbre* un sommet et une arête
- À une étape donnée, pour chaque sommet u non dans *tabArbre*, on associe proche(u)
  - Le sommet dans *tabArbre* qui est tel que le poids de l'arête {u, proche(u)} soit minimum sur l'ensemble des sommets de *tabArbre*
  - Ce poids est dit distance $d(u)$ de u à *tabArbre*

Algorithme de Prim

```
tabArbre ← {x0} (x0 étant un sommet quelconque)
pivot ← {x0}
POUR tout sommet u autre que x0
    d(u) ← +∞
POUR i de 1 à |S| – 1
    POUR tout sommet z voisin de pivot non dans tabArbre
        SI p(pivot, z) < d(z)
            proche(z) ← pivot
            d(z) ← p(pivot,z)
    Parmi les sommets qui ne sont pas dans tabArbre, déterminer
    un sommet pivot qui réalise le minimum des valeurs de d
    Ajouter pivot et l'arête {pivot, proche(pivot)} à tabArbre
```

- On choisit alors de faire entrer dans *tabArbre* le sommet *u* dont la distance *d*(*u*) à *tabArbre* est minimum
    - Ce sommet sera le *pivot* de l'étape suivante
- On met ensuite à jour les attributs des sommets *z* qui ne sont pas encore dans *tabArbre* et qui sont voisins de *pivot*
    - En comparant l'ancienne distance *d*(*z*) de *z* à *tabArbre* à la nouvelle façon d'atteindre *z* à partir du *pivot*
        - Si *p*(*pivot*,*z*) est plus petit que l'actuelle valeur de *d*(*z*) ...
        - ... alors proche(*z*) prend *pivot* comme valeur
        - ... et *d*(*z*) reçoit *p*(*pivot*,*z*)

Remarques

- Après la mise à jour des distances entre le pivot actuel et ses voisins
  - On doit parcourir tout le tableau d pour déterminer le pivot de l'itération suivante
  - $O(S)$
- On peut réduire ce temps par l'utilisation d'une file de priorités min implémentée par tas

PROCHAIN COURS

PLUS COURTS CHEMINS