

# ALGORITHMIQUE DISTRIBUÉE

## MIF12

### ALGORITHMES DISTRIBUÉS SUR LES GRAPHS

---

Isabelle GUERIN LASSOUS

[perso.ens-lyon.fr/isabelle.guerin-lassous/index-M1if12.htm](http://perso.ens-lyon.fr/isabelle.guerin-lassous/index-M1if12.htm)

[isabelle.guerin-lassous@univ-lyon1.fr](mailto:isabelle.guerin-lassous@univ-lyon1.fr)

# Hypothèses de ce cours

- Graphe représentant le système distribué
  - **Graphe connexe**
  - Liens de communication bidirectionnels
  - $n$  nœuds et  $m$  liens
  - $D$  diamètre du graphe
  - $d(u)$  : degré du nœud  $u$
- Nœuds / entités / processus **statiques**
- Chaque nœud a un **identifiant unique**
- **Messages toujours correctement reçus**
  - Mais il peut y avoir un délai dans la transmission du message
- **Canal FIFO**

# Algorithme synchrone

- Gouverné par une horloge globale externe
- Chaque nœud exécute une suite de rondes
- Le début de chaque ronde est déterminé par l'horloge globale
- Dans une ronde
  - Chaque nœud envoie un message à chacun de ses voisins
  - Tous les messages envoyés dans une ronde sont reçus et traités dans la même ronde

# Algorithme partiellement synchrone

- Il n'y a pas d'horloge globale, mais
- dans une ronde, chaque nœud envoie un message à chacun de ses voisins
  - Un nœud peut passer à la ronde  $r+1$  s'il a reçu un message de chacun de ses voisins de la ronde  $r$

# Algorithme asynchrone

- Pas d'horloge globale externe
- L'avancement d'un nœud est lié à ses propres calculs et aux messages qu'il reçoit
- **Dans ce cours**
  - on va principalement étudier des algorithmes asynchrones
  - parfois des algorithmes partiellement synchrones

# Apprendre le graphe sous-jacent au système distribué

## 1<sup>er</sup> algorithme

- Objectif
  - Que chaque nœud connaisse le graphe du système
- Initialisation
  - Chaque nœud connaît son  $ID$  et celles de ses voisins
- Principe
  - Chaque nœud envoie, à tous ses voisins, son  $ID$  et celles de ses voisins
  - Quand un nœud reçoit, pour la 1<sup>ère</sup> fois, la paire  $(ID(k) ; ID\text{-}voisins(k))$ 
    - Il met à jour le graphe appris
      - Liste de nœuds mise à jour avec  $ID(k)$
      - Liste de liens mise à jour avec les liens de  $ID(k)$
    - Il retransmet le message à tous ses voisins (sauf à celui qui lui a envoyé ce message)
- Discussion
  - Principe de retransmission / abandon des messages
  - Algorithme qui termine
  - Quand un nœud sait qu'il a terminé ?
    - Sans connaître au départ le nombre de nœuds et leur  $ID$
  - Complexité
    - Nombre de messages transmis  $\leq 2nm$

# Apprendre le graphe sous-jacent : exemple

## À partir de là...

- Une fois le graphe connu par tous les nœuds du système, on peut imaginer
  - Choisir un nœud particulier (**leader**) dans le système
  - Résoudre les problèmes avec des algorithmes séquentiels sur ce leader
  - Le leader envoie les résultats aux nœuds du réseau
- Faiblesses de l'approche
  - Peu robuste envers la dynamique (topologie, données d'entrées, ...)
  - Peut être coûteux en temps
  - Peu tolérant aux pannes



## Ou encore

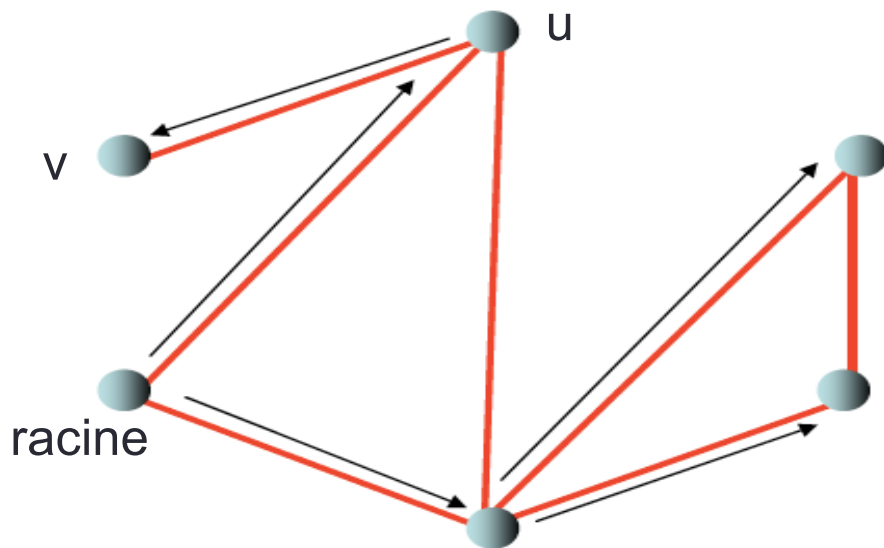
- Une fois le graphe connu par tous les nœuds du système, on peut imaginer que
  - chaque nœud exécute l'algorithme séquentiel sur le graphe appris
- Il faut s'assurer que les exécutions séquentielles sur les différents nœuds vont donner un résultat cohérent
  - Par ex. : algorithme de Dijkstra utilisé pour le routage ne donne pas de boucle de routage
- Peut être coûteux en temps
- Peu robuste envers la dynamique

# Opération de diffusion

- Diffusion / Broadcast
  - Un nœud (**source**) veut envoyer un message à tous les autres nœuds du système
- Hypothèses
  - Chaque nœud connaît son ID
  - Le nombre de nœuds dans le système n'est pas connu
  - Le graphe sous-jacent au système n'est pas connu
- Algorithme d'inondation
  - Utilisation du principe de retransmission / abandon des messages
    - avec une seule source
- Terminaison
  - mais la source ne sait pas que et quand l'algorithme a terminé
- Complexité
  - Au moins  $m$  envois du message & au plus  $2m-d(\text{source})$  envois
  - **Comment réduire ce nombre d'envois du message ?**

# Notion d'arbre : rappel

- **Arbre**
  - Graphe non orienté, acyclique et connexe
- **Arbre couvrant** d'un graphe non orienté et connexe
  - Arbre qui connecte tous les sommets du graphe



## Arbre couvrant enraciné

a une racine unique

$u \longrightarrow v$  :  $u$  est **parent** de  $v$  et  $v$  est **fil** de  $u$

$\longrightarrow$  : représente la descendance mais lien bidirectionnel

# Opération de diffusion sur un arbre

- **Algorithme d'inondation sur un arbre enraciné** (connu)
  - Racine envoie son message à ses fils
  - Nœuds qui reçoivent le message le retransmettent à leurs fils
  - Etc.
- **Terminaison de l'algorithme quand les feuilles reçoivent le message**
  - Mais la racine ne sait pas que l'algorithme est terminé
- **Complexité**
  - $n-1$  envois du message
- **Si de nombreux messages à envoyer dans le fonctionnement du système distribué**
  - Rentable de construire un arbre couvrant
  - Et de diffuser les messages sur cet arbre

# Opération de convergecast sur un arbre

- **Convergecast**
  - Inverse de la diffusion
  - Chaque nœud veut envoyer un message à un nœud spécifique (par ex. la racine de l'arbre)
- **Hypothèse**
  - Arbre enraciné connu
- **Algorithme Echo**
  - **Initié par les feuilles de l'arbre**
    - Feuilles envoient leur message à leur parent
    - Une fois les messages reçus de tous leurs fils, chaque nœud transmet un message à son parent ; message = données du nœud + données des fils
    - Une fois que la racine a reçu un message de chacun de ses fils, l'algorithme Echo est terminé
- **Terminaison**
  - Et la racine sait que l'algo est terminé, mais les autres nœuds de l'arbre ne le savent pas
- **Complexité**
  - $n-1$  messages

# Utilisation de l'algorithme Echo

- Pour
  - La **détection de terminaison, par la racine, d'algorithmes distribués**
  - Calculer le nombre de nœuds dans le système
  - Calculer la valeur maximale/minimale d'un paramètre dans le système
  - Calculer la somme des valeurs d'un paramètre
  - ...

# Première conclusion

- Utilisation des arbres permet
  - De communiquer (efficacement) de un vers tous
  - De communiquer (efficacement) de tous vers un
  - De récupérer des valeurs/paramètres de tous les nœuds
    - Et d'en faire des calculs
  - De savoir si tous les nœuds ont terminé
  - ...
- Comment construire des arbres couvrants de manière distribuée ?

# Construction d'un arbre couvrant

## Algorithme basé sur la diffusion et le convergecast

- **Initialisation**

- Racine connue dans le graphe (seulement par la racine)
- Chaque nœud connaît tous ses voisins

- **Algorithme**

1. Racine envoie un message *Join* à ses voisins pour rejoindre l'arbre
2. Pour chaque nœud  $u$  recevant un message *Join* du nœud  $v$ 
  - 1<sup>ère</sup> fois qu'un message *Join* est reçu  
 $Père(u) := v$   
 Enregistre qu'il a reçu un message de  $v$   
 S'il n'a pas reçu un message de tous ses voisins  
 Envoi d'un message *Join* à tous ses voisins sauf  $v$
  - Sinon (message *Join* déjà reçu)  
 Indique à  $v$  qu'il a déjà un père via un message *Back*(no)
  - S'il a reçu un message de tous ses voisins  
 Envoi d'un message *Back*( $u$ ) à son père
3. Pour chaque nœud  $u$  recevant un message *Back* de  $v$ 
  - Si  $v$  indique qu'il n'avait pas de père (*Back*( $v$ )) :  $Fils(u) := Fils(u) + \{v\}$
  - Enregistre qu'il a reçu un message de  $v$
  - Si  $u$  (non racine) a reçu un message de tous ses voisins  
 Envoi d'un message *Back*( $u$ ) à son père



## Construction d'un arbre couvrant : exemple

# Construction d'un arbre couvrant

## Algorithme basé sur la diffusion et le convergecast

- Terminaison
  - Quand la racine reçoit un message Back de chacun de ses voisins
- Plusieurs arbres peuvent être construits
  - Dépend de la vitesse des messages
- On peut vouloir construire des arbres avec des propriétés spécifiques
  - comme ?

# Arbre couvrant en largeur

## Algorithme sans contrôle centralisé

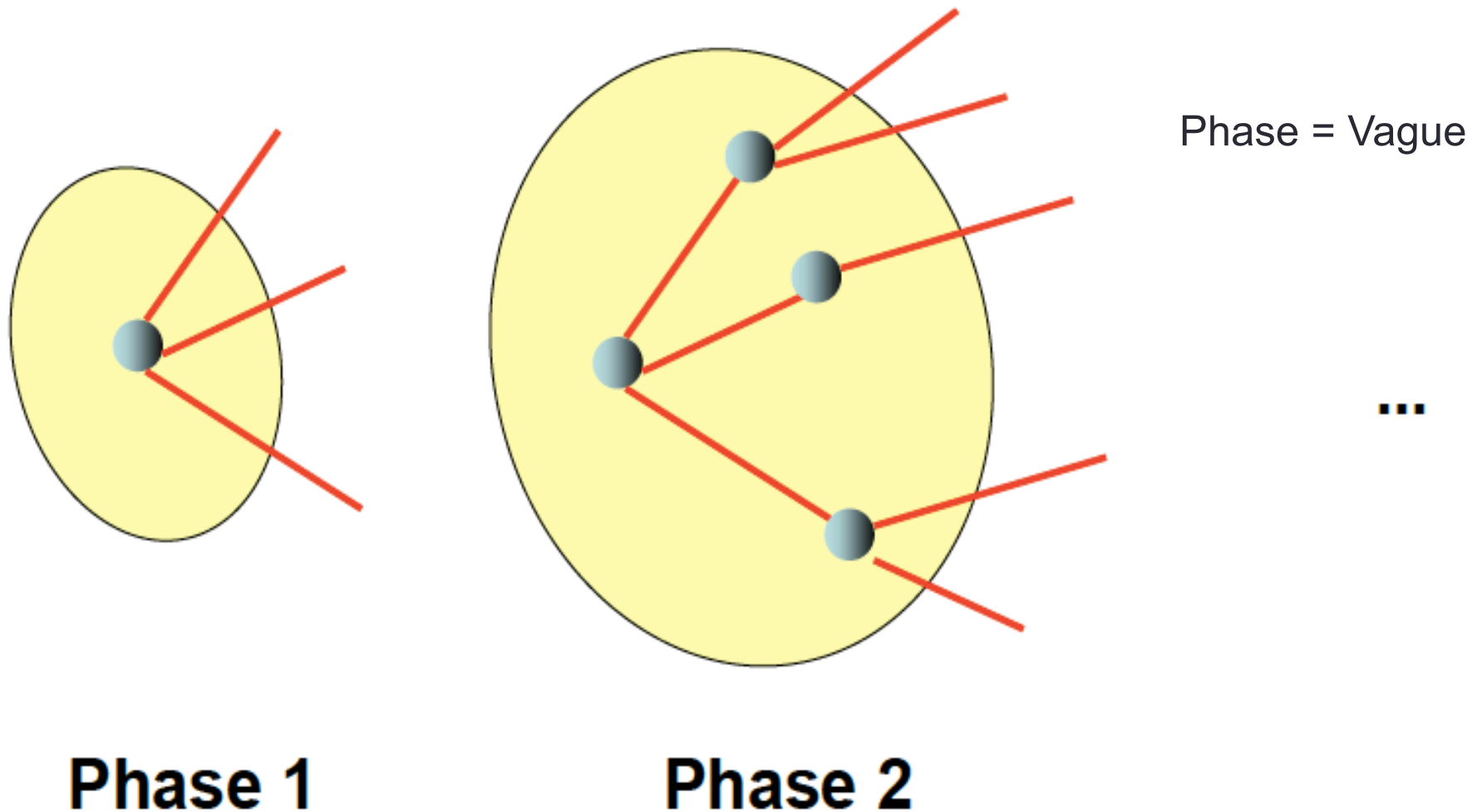
- Algorithme de Cheung – « à la Bellman-Ford »
  - Utilisation de la distance à la racine
  - $d_u$  = distance de  $u$  à la racine déterminée par  $u$
- Initialisation
  - Initialisation identique à l'algorithme précédent
  - $d_{racine} = 0$  et  $d_v = \infty$  pour  $v$  différent de la racine
- Algorithme
  - Assez proche de l'algorithme précédent
    - Utilisation de  $d_u$  dans les messages échangés
  - Racine envoie *Join*(1) à tous ses voisins
  - Si un nœud  $u$  reçoit, d'un voisin  $v$ , un message *Join*( $y$ ) avec  $y < d_u$  alors
    - $d_u := y$
    - Nœud  $u$  envoie le message *Join*( $y+1$ ) à tous ses voisins sauf à  $v$
  - Il faut aussi gérer les mises à jour sur les variables Père et Fils
    - Et ne pas oublier les messages Back pour la détection de terminaison !
    - Pas si simple

## Arbre couvrant en largeur

### Algorithme avec contrôle centralisé

- Algorithme de Zhu & Cheung – « à la Dijkstra »
  - Ajouter les nœuds les plus proches de la partie de l'arbre déjà construite
- Exécution par **vagues**
  - Le démarrage d'une vague est contrôlé par la racine
  - Vague 1 va inclure les voisins de la racine dans la racine
  - Vague 2 va inclure les voisins des voisins de la racine dans l'arbre
  - Etc.
- **Attention**
  - Une vague n'est pas une ronde

# Arbre couvrant en largeur Algorithme avec contrôle centralisé



## Arbre couvrant en largeur

### Algorithme avec contrôle centralisé

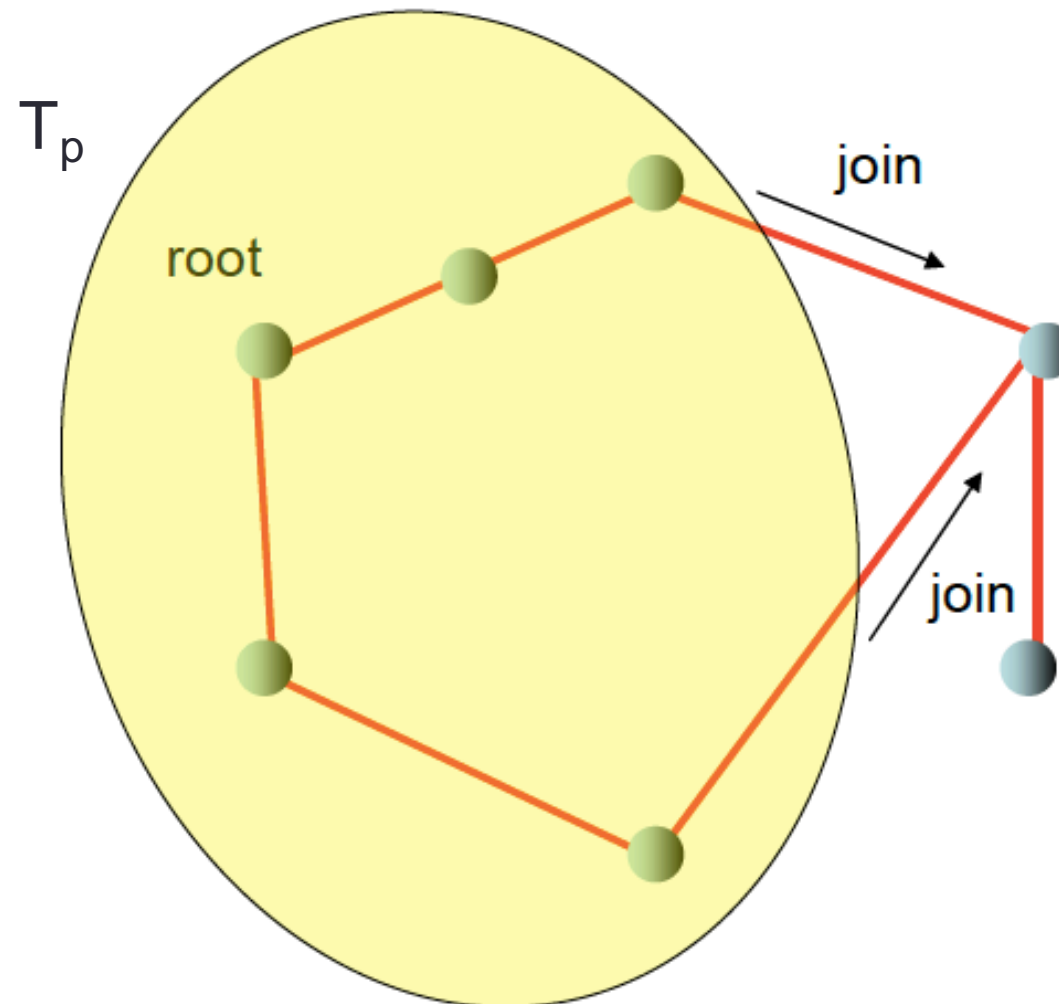
1.  $T_0$  = arbre contenant la racine
2.  $p=1$
3. Répéter (jusqu'à ne plus découvrir de nouveaux nœuds)
  - Racine démarre la vague  $p$  en diffusant le message  $Vague(p)$  dans l'arbre  $T_{p-1}$
  - Pour chaque feuille  $u$  de  $T_{p-1}$  recevant le message  $Vague(p)$ 
    - $u$  envoie un message  $Join(p)$  à tous ses voisins  $v$  sauf son père
  - Pour chaque nœud  $v$  recevant un message  $Join(p)$  de  $u$ 
    - Si c'est le 1<sup>er</sup> message  $Join$  reçu,  $v$  envoie un message  $ACK$  à  $u$  et devient feuille de l'arbre  $T_p$  et  $Père(v) := u$
    - Sinon  $v$  envoie un message  $NACK$  à  $u$
  - Pour chaque feuille  $u$  de  $T_{p-1}$ 
    - À la réception d'un message  $ACK$  envoyé par  $v$ 
      - $Fils(u) := Fils(u) + \{v\}$
    - Quand tous les voisins de  $u$  (sauf son père) ont envoyé à  $u$  les messages  $ACK$  ou  $NACK$ , alors  $u$  démarre l'algorithme Echo
  - Algorithme Echo dans l'arbre  $T_{p-1}$ 
    - Quand la racine reçoit un message  $Echo$  de chacun de ses fils
    - $p := p+1$  // Passage à la vague suivante

## Arbre couvrant en largeur

Algorithme avec contrôle centralisé : début de l'algorithme

## Arbre couvrant en largeur

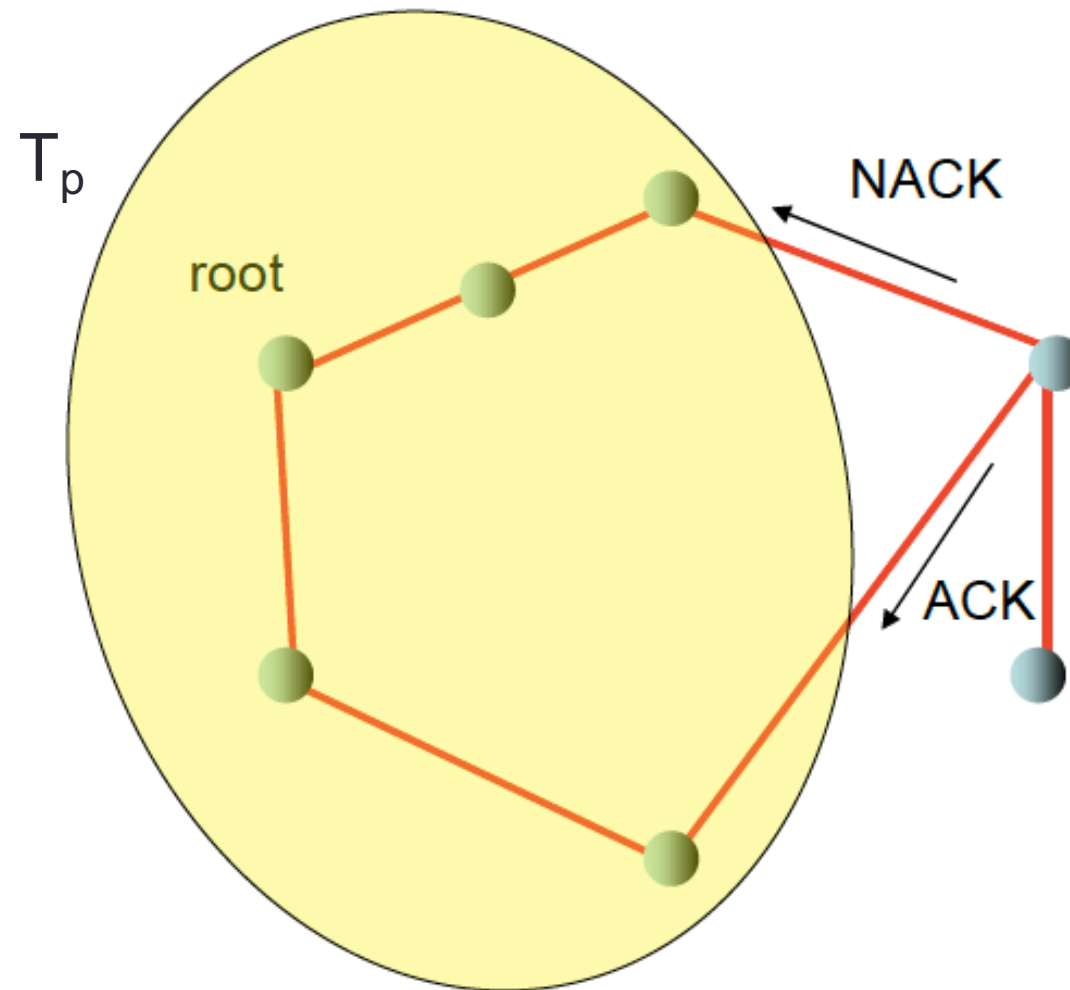
Algorithme avec contrôle centralisé : suite de l'algorithme





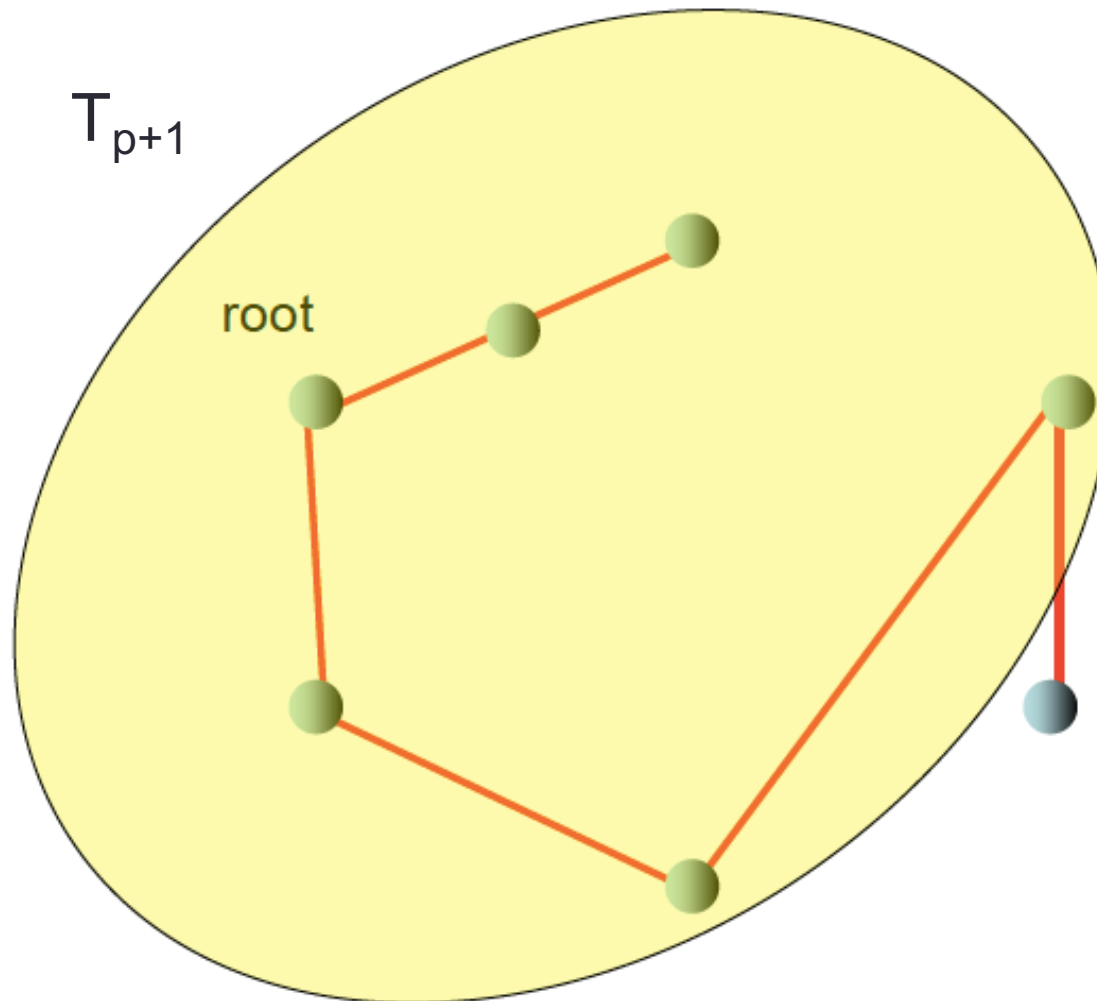
## Arbre couvrant en largeur

## Algorithme avec contrôle centralisé : suite de l'algorithme




## Arbre couvrant en largeur

Algorithme avec contrôle centralisé : suite de l'algorithme



## Complexité des deux algorithmes de construction d'un arbre couvrant en largeur

	Complexité temps	Nombre messages
À la Dijkstra	$O(D^2)$	$O(m+nD)$
À la Bellman-Ford	$O(D)$	$O(nm)$

- **Analyse au TD2**
- Peut-on faire mieux ?
  - Oui, mais pas le temps pour ce cours ... 

# Arbre couvrant en profondeur

## Algorithme simple

- **Initialisation**

- Racine connue (seulement par elle-même) ; chaque nœud connaît ses voisins
- $Visités(u)$  initialisée à vide pour chaque nœud // liste des voisins visités

- **Algorithme**

1. Racine envoie un message *Join* à un de ses voisins
2. Chaque nœud  $u$  recevant un message *Join* de  $v$ 
  - S'il n'a pas encore de parent
    - $Parent(u) := v$
    - $Visités(u) := \{v\}$
    - Si tous ses voisins ont été visités
      - $u$  envoie un message *Back(yes)* à  $v$
    - Sinon  $u$  envoie un message *Join* à un de ses voisins non encore visités
  - Sinon  $u$  envoie un message *Back(no)* à  $v$
3. Chaque nœud  $u$  recevant un message *Back* de  $v$ 
  - Si *yes*,  $Fils(u) := Fils(u) + \{v\}$
  - $Visités(u) := Visités(u) + \{v\}$
  - Si tous ses voisins ont été visités
    - $u$  envoie un message *Back(yes)* à son père si  $u$  n'est pas racine
  - Sinon  $u$  envoie un message *Join* à un de ses voisins non encore visités

- **Terminaison**

- Quand la racine sait que tous ses voisins ont été visités

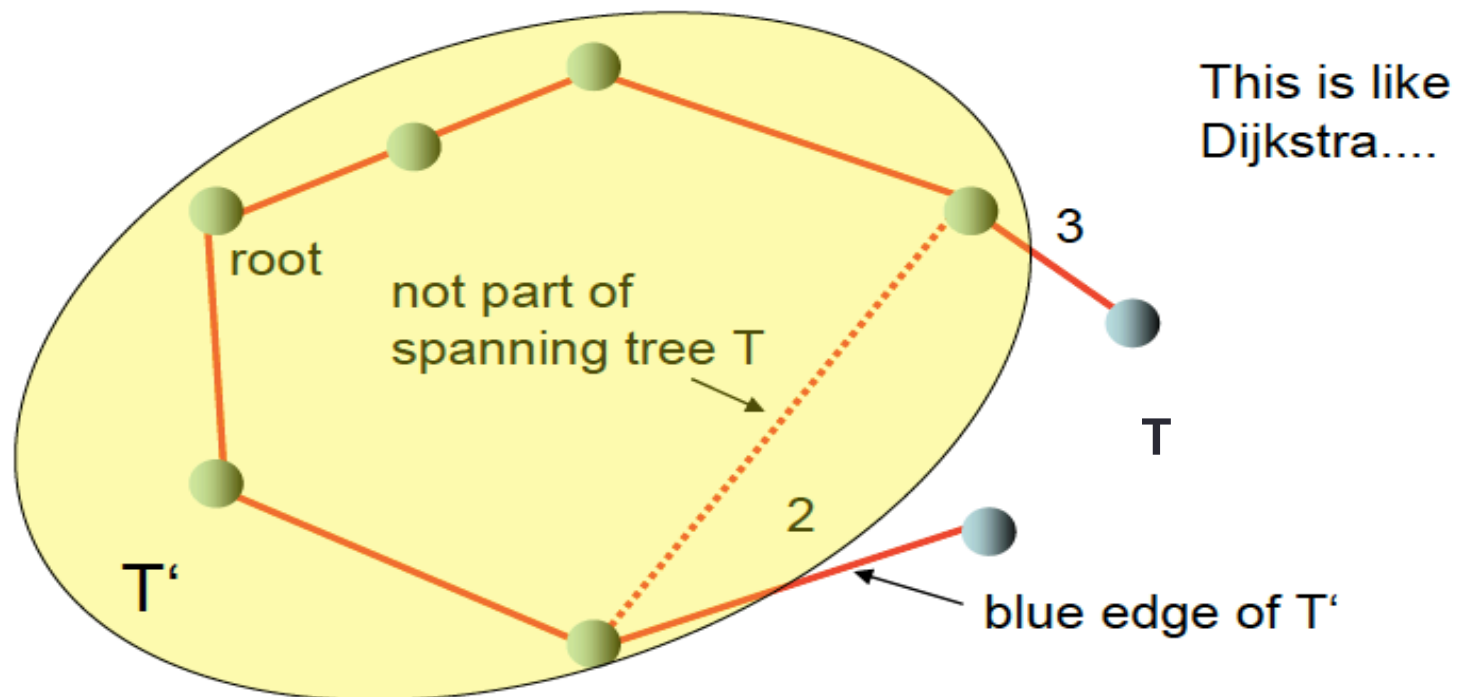
# Arbre couvrant minimum

- Métriques autres que la distance peuvent être intéressantes
- Liens peuvent avoir des métriques différentes
- Graphe pondéré
  - Graphe avec des poids sur les arêtes
- Arbre couvrant minimum
  - Minimise  $\text{Somme}_{\{e \text{ arête de l'arbre}\}} \text{poid}(e)$
- **Hypothèse**
  - Deux arêtes n'ont pas le même poids
    - Arbre couvrant minimum est unique

## Arbre couvrant minimum

### Arête bleue

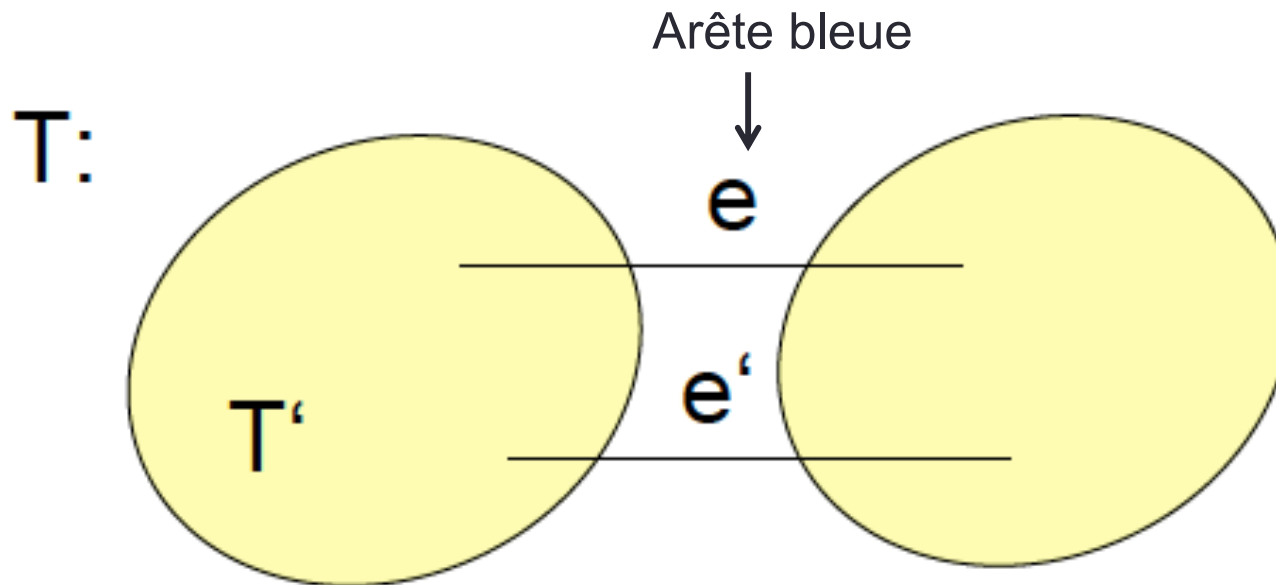
- $T$  arbre couvrant
- $T'$  sous-graphe de  $T$
- L'arête  $e=(u,v)$  est une **arête sortante** si  $u$  dans  $T'$  et  $v$  dans  $T \setminus T'$
- L'arête sortante de poids minimal est appelée **l'arête bleue**



# Arbre couvrant minimum

## Lemme

- Si  $T$  est un arbre couvrant minimum et  $T'$  un sous-graphe
  - Alors l'arête bleue de  $T'$  fait partie de  $T$
- **Esquisse de preuve par contradiction ?**





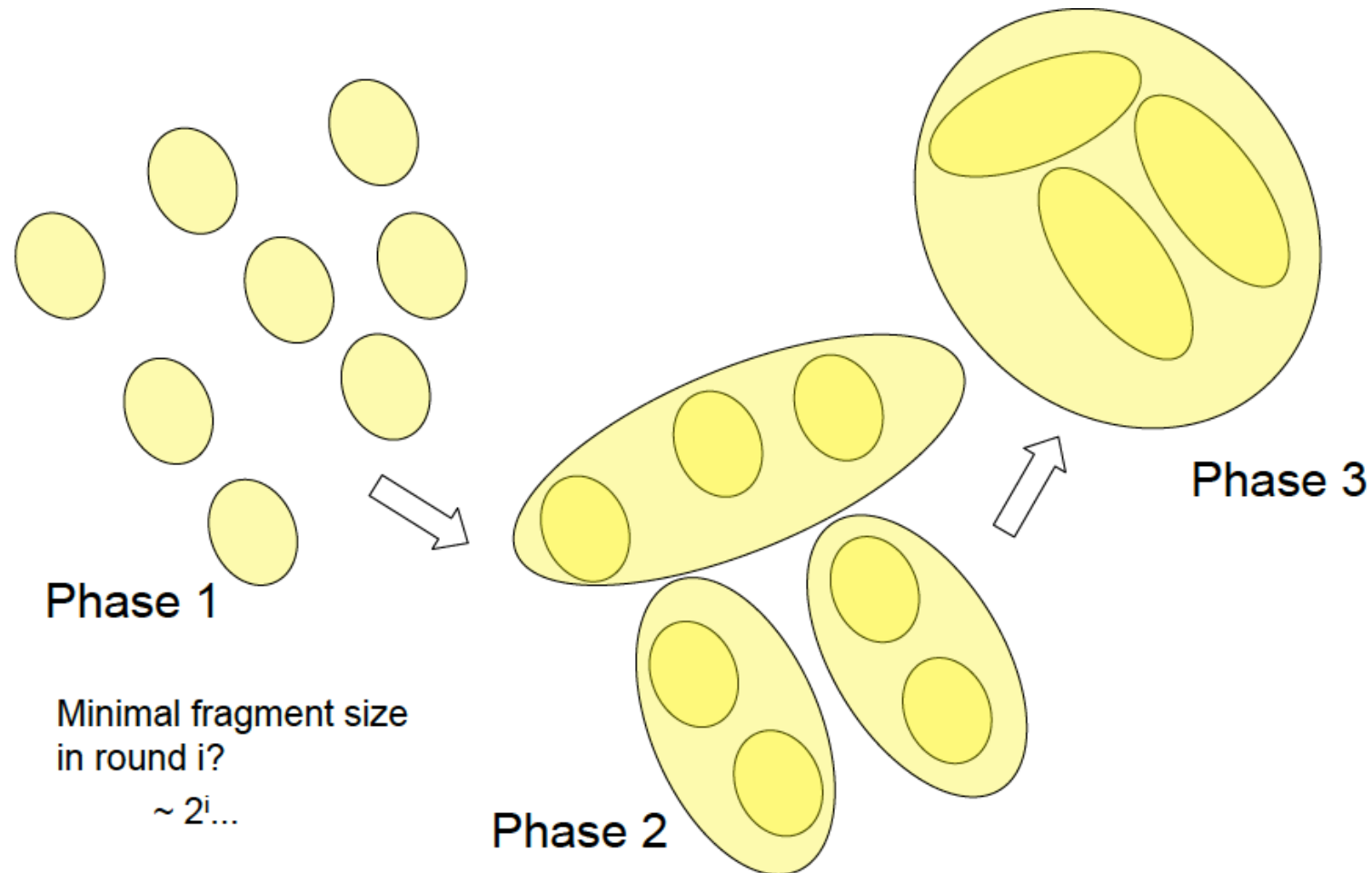


# Arbre couvrant minimum

## Idées

- Arêtes bleues sont la clé
- Construction par **fragments** correspondant au sous-graphe de l'arbre de poids minimum
  - À l'initialisation chaque fragment est un nœud
  - Fusion des fragments reliés par les arêtes bleues
    - On est sûr qu'aucun cycle n'est créé
  - Arrêt quand il n'y a plus de fragment isolé
- Version distribuée de l'algorithme de Kruskal

# Arbre couvrant minimum Fusion des fragments



Au maximum  $\log_2(n)$  phases

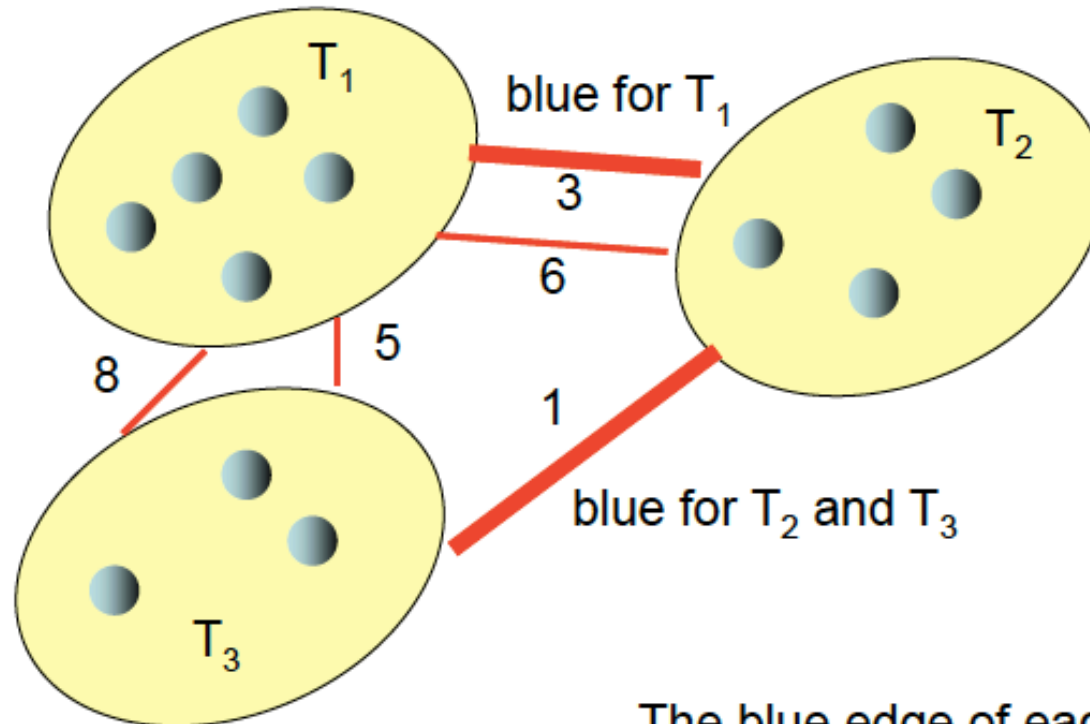
## Arbre couvrant minimum

### Algorithme de Gallager-Humblet-Spira

1. Chaque nœud est la racine de son propre fragment
2. ID du fragment = ID de la racine
3. Répéter jusqu'à ce que tous les nœuds soient dans le même fragment
  - Chaque nœud apprend les ID des fragments de ses voisins
  - La racine de chaque fragment trouve l'arête bleue  $b=(u,v)$  du fragment avec une diffusion et un convergecast au sein du fragment
  - Inversion de la relation parent – fils sur le chemin racine –  $u$  au sein de chaque fragment
    - Et  $u$  devient la racine du fragment
  - $u$  envoie une requête de fusion à  $v$
  - Si  $v$  a aussi envoyé une requête de fusion sur la même arête  $b$ 
    - Alors fusion des fragments et celui qui a le plus petit ID devient la racine du fragment fusionné
    - Sinon  $v$  devient le parent de  $u$
  - La nouvelle racine informe tous les nœuds de son fragment sur son ID

# Arbre couvrant minimum

## Algorithme de Gallager-Humblet-Spira : exemple

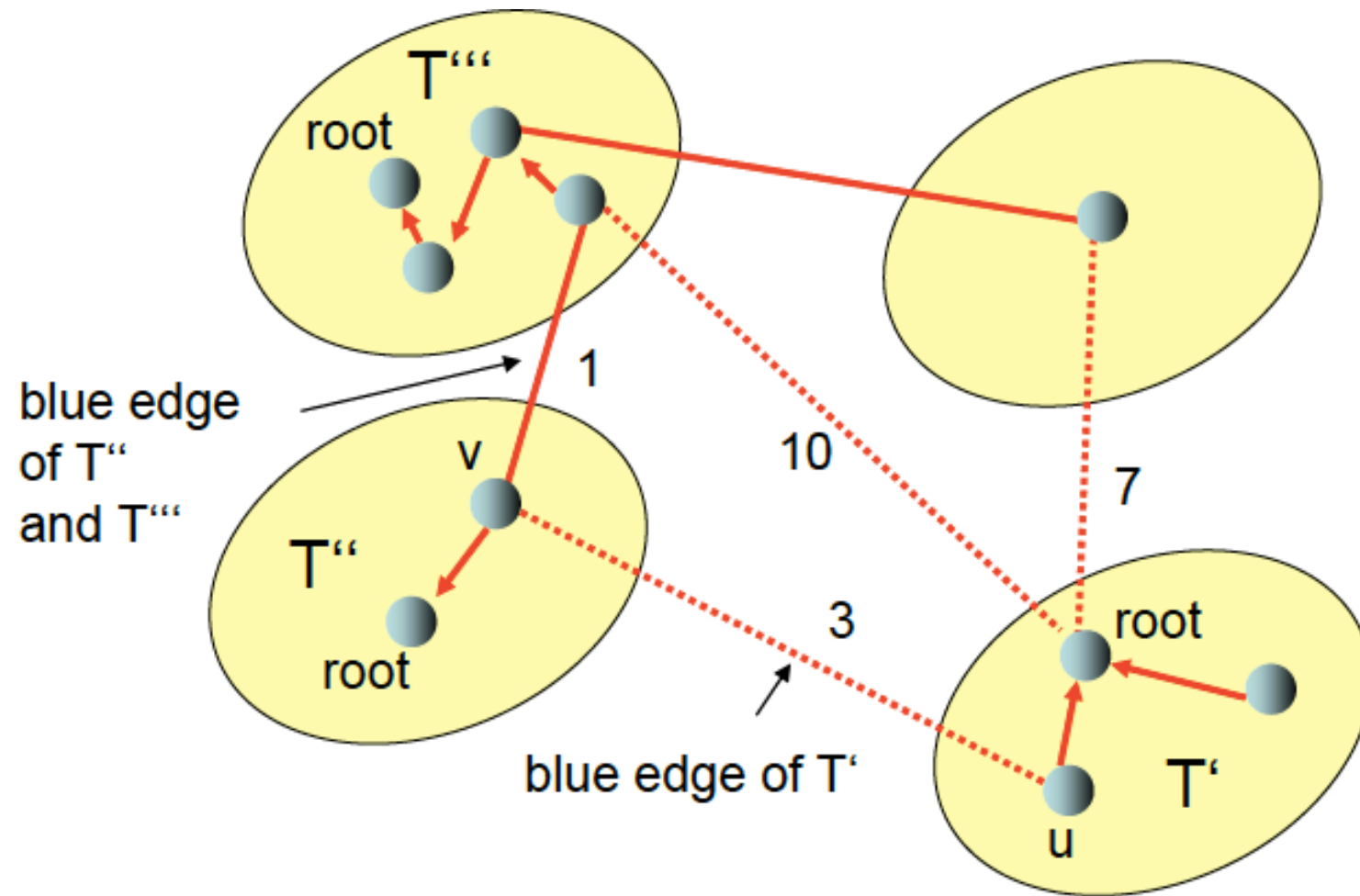


The blue edge of each fragment can  
be taken for sure: cycles not possible!  
(Blue edge lemma!)

So we can do it **in parallel!**

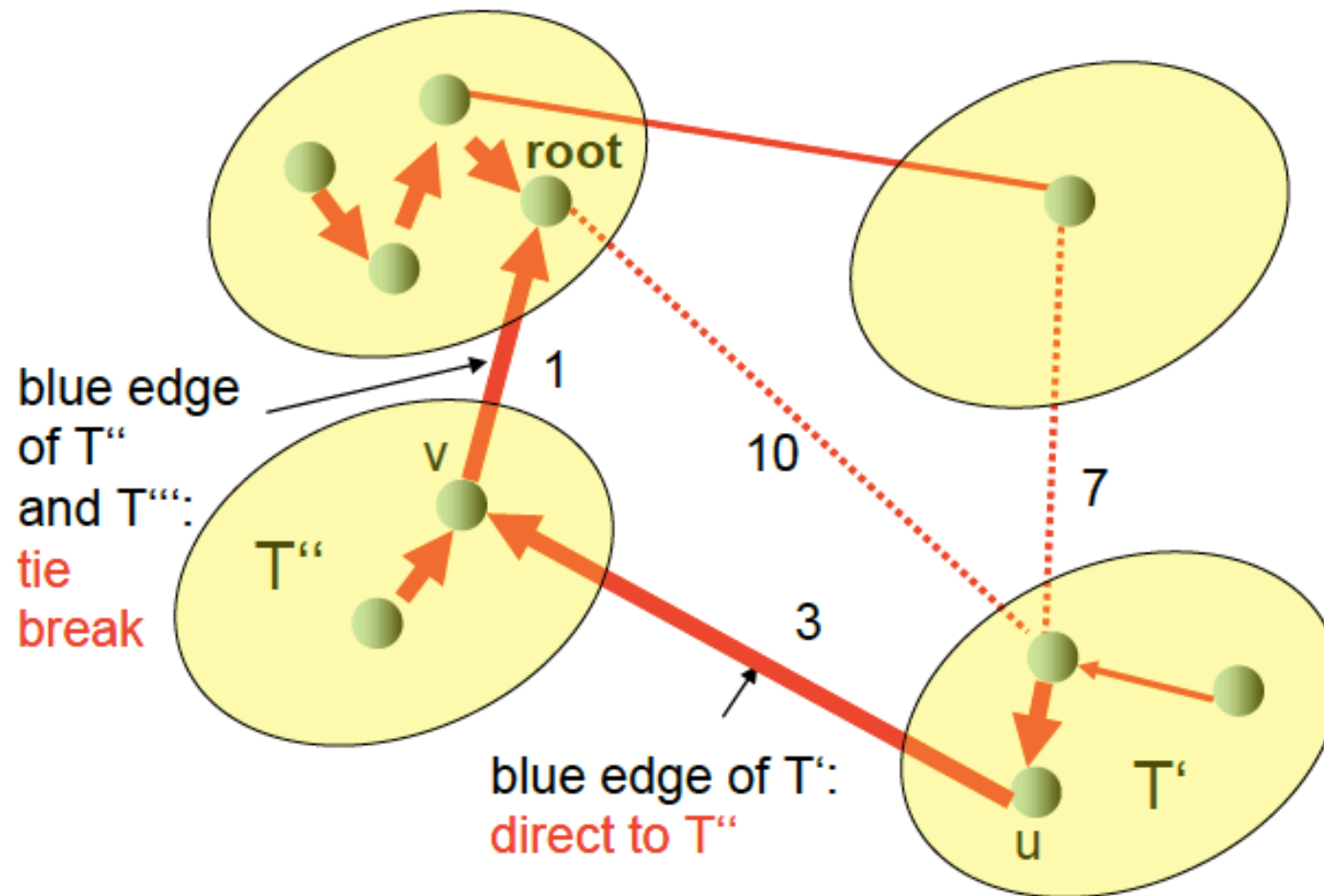
# Arbre couvrant minimum

## Algorithme de Gallager-Humblet-Spira : exemple



# Arbre couvrant minimum

## Algorithme de Gallager-Humblet-Spira : exemple



# Coloriage de graphe

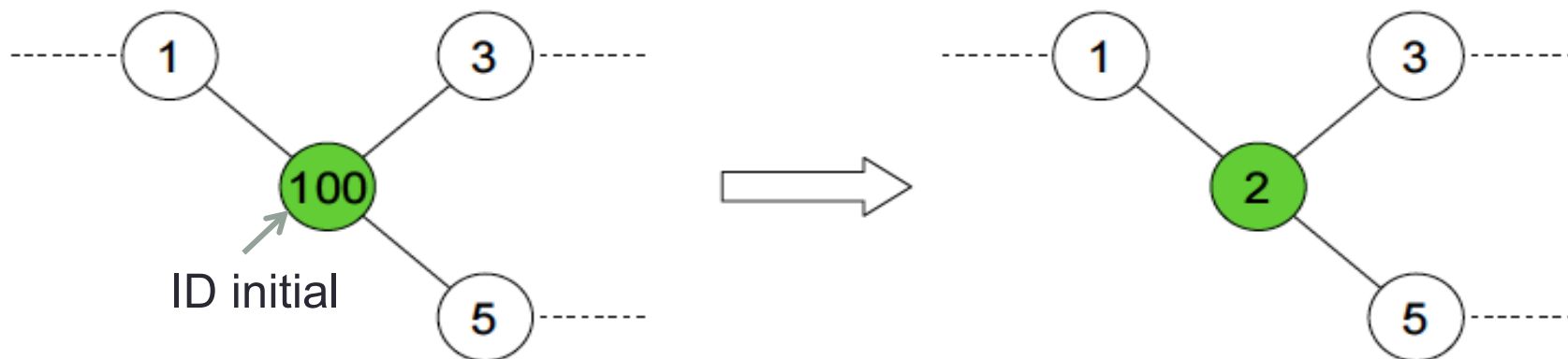
- Problème du coloriage d'un graphe
  - Colorier tous les nœuds du graphe tel que deux sommets voisins n'ont pas la même couleur
  - On cherche souvent à avoir un petit nombre de couleurs
- Applications variées
  - Allocation de ressources
  - Ordonnancement
  - Compilation

# Coloriage glouton distribué

- **Hypothèse**
  - Chaque nœud a un **unique ID**
- Algorithme **partiellement synchrone**
  - Fonctionnement par rondes (non synchronisées sur un temps global)
- **Initialisation**
  - Chaque nœud connaît tous ses voisins
- Chaque nœud  $v$ 
  - $Couleur[v] := indécis$
  - Envoie  $Couleur[v]$  et son  $ID$  à tous ses voisins
  - Attend un message de chacun de ses voisins
  - Tant que le nœud  $v$  a des voisins indécis et d' $ID$  plus élevée
    - Envoie  $Couleur[v]$  et son  $ID$  à tous ses voisins indécis
    - Attend un message de chacun de ses voisins indécis
  - Choisit la plus petite couleur libre possible
  - $Couleur[v] := couleur-choisie$
  - Envoie  $Couleur[v]$  à chacun de ses voisins



# Coloriage glouton distribué



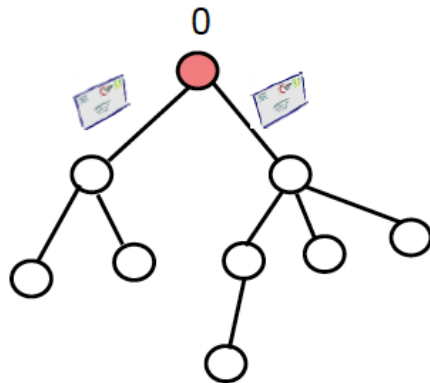
- Terminaison
  - Quand il n'y a plus de nœud indécis
  - Un nœud ne sait pas si l'algorithme a terminé
    - Il sait seulement s'il a terminé et si ses voisins ont terminé
- Complexité
  - Au plus  $n$  rondes
- Nombre de couleurs
  - au plus  $\Delta + 1$
  - $\Delta$  = degré du graphe (max des degrés)

# Coloriage distribué d'un arbre

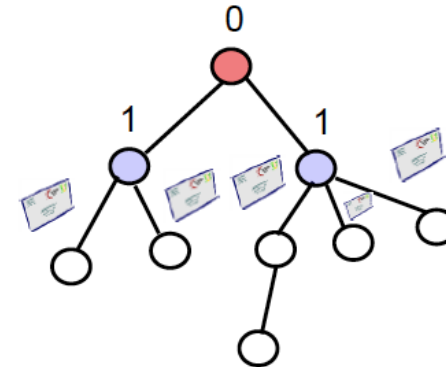
- Arbre enraciné
- Algorithme très simple mais lent
  - Racine prend la couleur  $C_p=0$  et envoie cette couleur à ses fils
  - Chaque nœud  $v$  exécute l'algorithme suivant
    - Si le nœud  $v$  reçoit la couleur  $C_p$  (de son parent), alors
      - Le nœud  $v$  prend la couleur  $C_v = 1 - C_p$
      - Il envoie cette couleur à ses fils

# Coloriage distribué d'un arbre : exemple

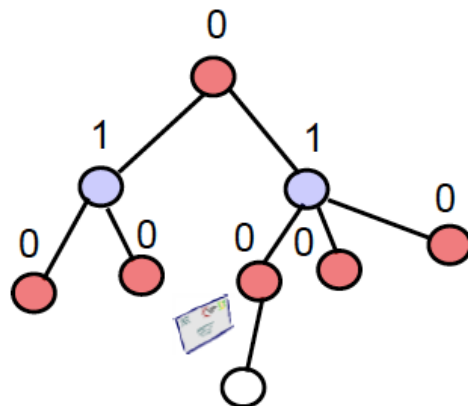
Ronde 1



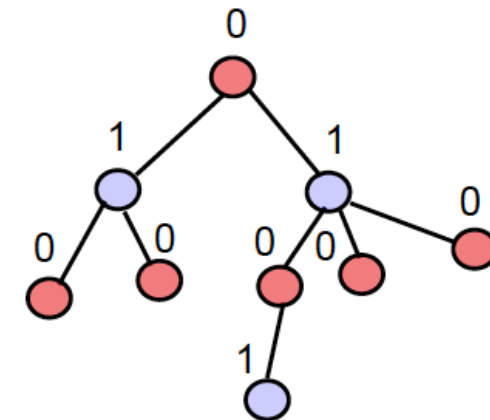
Ronde 2



Ronde 3



Ronde 4



# Coloriage distribué d'un arbre

- Terminaison
  - Quand les feuilles choisissent leur couleur
  - Mais les nœuds, autres que les feuilles, ne savent pas si l'algorithme a terminé
- Complexité
  - En temps
    - Proportionnel à la profondeur de l'arbre
      - Un nœud à l'étage  $i$  doit attendre que les nœuds précédents dans l'arbre aient choisi leur couleur
    - Profondeur de  $n$  pour des arbres déséquilibrés
  - En messages
    - $n-1$  messages envoyés
- Nombre de couleurs : 2
- Peut-on faire plus rapide ?
  - Idée : utiliser les IDs pour construire les couleurs

# Coloriage distribué rapide d'un arbre

- **Hypothèses**

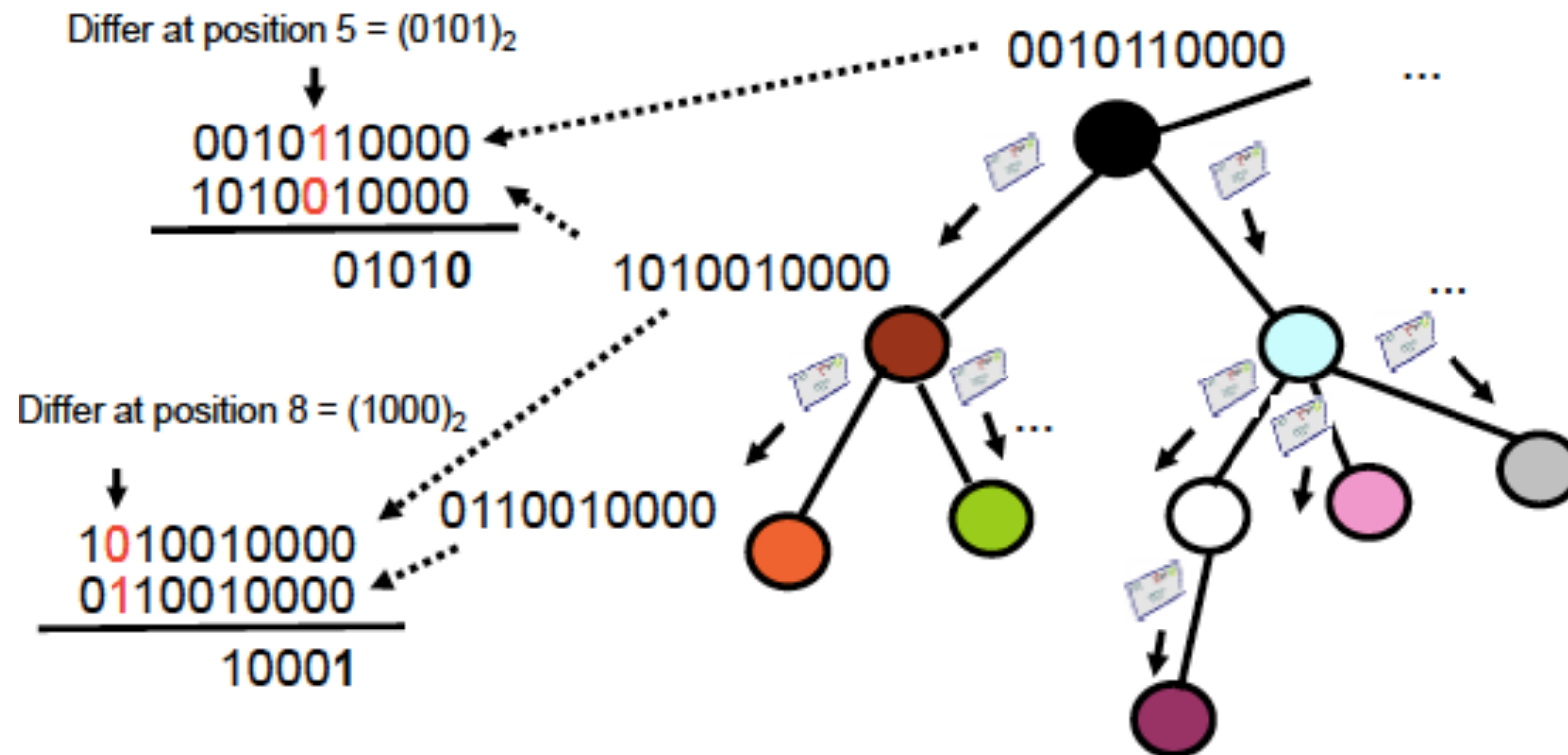
- Nœuds ont un ID encodé sur  $\log n$  bits ( $n$  = nombre de nœuds)
- Initialement, couleur d'un nœud = ID

- Racine prend la couleur 0

- Chaque nœud  $v$  exécute l'algorithme suivant

- Nœud  $v$  envoie sa couleur  $C_v$  à tous ses fils
- Nœud  $v$  répète les instructions suivantes
  - recevoir la couleur  $C_p$  de son parent
  - exprimer  $C_p$  et  $C_v$  en binaire
  - $i :=$  le plus petit indice où  $C_v$  et  $C_p$  diffèrent
  - $C_v := i$  (exprimé en binaire). $i^{\text{e}}$  bit de  $C_v$
- Jusqu'à ce que tous les nœuds aient une couleur dans l'ensemble  $\{0, \dots, 5\}$

# Coloriage distribué rapide d'un arbre : exemple



# Coloriage distribué rapide d'un arbre

- Algorithme **faiblement synchrone**
  - À chaque ronde
    - Chaque nœud attend un message de son parent (sauf la racine)
    - Chaque nœud envoie un message à ses fils (sauf les feuilles)
  - Tous les nœuds participent à l'algorithme à chaque ronde
- Complexité
  - Le nombre de bits / couleurs est diminué d'un facteur  $\log_2$  à chaque ronde
  - Le nombre de rondes est proportionnel à  $\log_2^* n$ 
    - $\log^* n$  = nombre de fois où il faut appliquer le log à  $n$  avant d'obtenir une valeur  $\leq 2$
  - $n-1$  messages envoyés à chaque ronde
- Nombre de couleurs : 6
- **Intuition sur pourquoi le coloriage est valide ?**
- Possible de réduire le nombre de couleurs

# Ce qu'il faut retenir

- Notions d'algorithmes synchrone, partiellement synchrone et asynchrone
- Propriétés des arbres
- Principe de retransmission / abandon des messages
- Algorithmes d'inondation sur un graphe / arbre
- Principe de l'algorithme Echo et son utilité
- Algorithmes de construction d'un arbre couvrant et leurs spécificités
- Principe des messages retour pour que la racine détermine la terminaison de l'algorithme
- Principe des algorithmes par vague
- Algorithme pour construire un arbre couvrant minimum & notion d'arête bleue
- Algorithmes de coloriage de graphe et d'arbre
- Technique pour diminuer la taille de l'encodage
  - donc pour réduire la valeur d'un paramètre