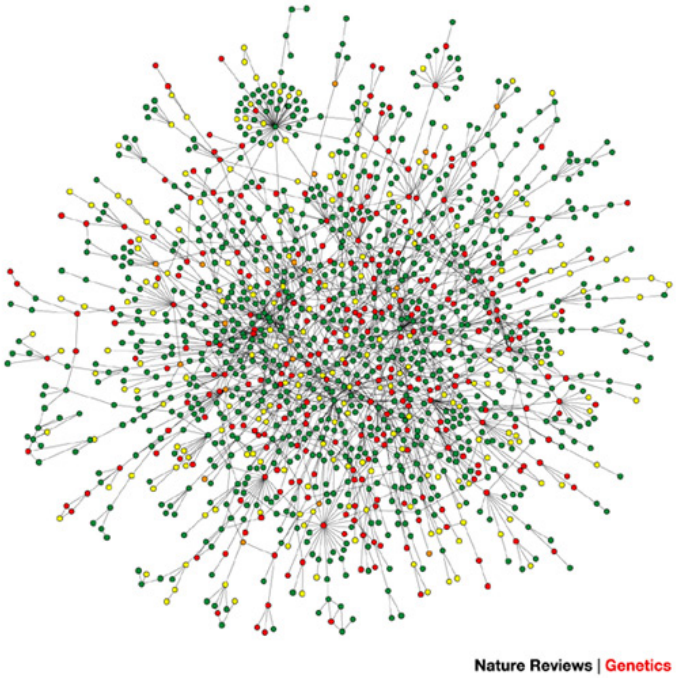
GRAPHES : REPRÉSENTATION ET ALGORITHMES ÉLÉMENTAIRES

Graphes non-orientés

Déf. Un **graphe non-orienté** est representé par un couple (V, E) où $E \subseteq \binom{V}{2}$ (paires non-ordonnées). V est l'ensemble des sommets et E est l'ensemble des arêtes.



Barabási & Oltvai *Nature Rev Genet* 5 :101, 2004

Graphes orientés

Déf. Un **graphe orienté** est representé par un couple (V, E) où $E \subseteq V \times V$ (paires ordonnées). V est l'ensemble des nœuds ou sommets, et E est l'ensemble des arcs.

Terminologie

Graphe non-orienté

L'arête $\{u, v\}$ est dénotée par uv.
Si $uv \in E$, alors v est adjacent à u.
L'arête $uv \in E$ est incidente aux sommets u et v.
Le degré de $u \in V$ est le nombre d'arêtes qui y sont incidentes.

Graphe orienté

L'arc (u, v) est dénotée par uv : l'arc part de u et arrive à v.
Si $uv \in E$, alors v est adjacent à u.
Le degré sortant de $u \in V$ est le nombre d'arcs qui y partent ; le degré rentrant est le nombre d'arcs qui y arrivent.

Chemins

Un **chemin** de longueur ℓ est une séquence $v_0, v_1, \ldots, v_\ell$ où $v_{i-1}v_i \in E$ pour tout $i = 1, \ldots, \ell$.

($\ell = 0$ est OK : chemin sans arêtes/arcs.)

Si $v_0 = v_\ell$, alors le chemin forme un **cycle**.

(Plus précisement, les sommets initial et final ne sont pas distingués dans le cycle.)

Le chemin $v_0 \cdots v_\ell$ est **élémentaire** ssi $v_1, \ldots, v_\ell$ sont distincts. Si $v_0 = v_\ell$, alors le chemin forme un **cycle élémentaire**.

Un graphe sans cycle est dit **acyclique**.

Un graphe non-orienté est **connexe** si chaque paire de sommets est relié par un chemin.

Un graphe non-orienté connexe acyclique est un **arbre**.

Sous-graphes

Le graphe $G' = (V', E')$ est un **sous-graphe** de $G = (V, E)$ ssi $V' \subseteq V$ et $E' \subseteq E$.

Étant donné un sous-ensemble de sommets $V' \subseteq V$, le sous-graphe de G **engendré** par V' est le graphe $G' = (V', E')$ avec $E' = \left\{uv \in E \colon u, v \in V'\right\}$.

Pondération

Graphe ponderé : chaque arc (ou arête) possède un **poids** ou coût associé, défini par la fonction de pondération $c\colon E \mapsto \mathbb{R}$.

Poids d'un sous-graphe : somme de poids des arcs dans le sous-graphe

Poids d'un chemin : somme de poids des arcs dans le chemin

Questions intéressantes

Stocker les graphes dans le mémoire d'un ordinateur

Parcours d'un graphe

Vérifier si le graphe est connexe

Plus court chemins

Arbres couvrants

Comment stocker le graphe ?

Matrice d'adjacence : matrice $V \times V$, $A[u, v]$ donne le poids de uv ($\pm\infty$ ou NaN pour arcs non-existants), ou valeurs booléennes pour noter juste présence

Listes d'adjacence : liste $\mathsf{Adj}[u]$ pour chaque sommet u qui stocke l'ensemble $\{v : uv \in E\}$ ou l'ensemble des couples $\{\langle v, c(u, v)\rangle : uv \in E\}$.

Usage de mémoire : dépend de la densité du graphe $\frac{|E|}{|V|^2}$.

Déterminer si $uv \in E$ ou $c(u, v)$: rapide avec la matrice mais plus lente avec les listes d'adjacence

Parcours

Technique essentielle : marquage/coloriage «jamais vu», «déjà vu»

```
     Algo PARCOURS-PROFONDEUR(u)
P1   // prévisite de u
P2   marquer u // «déjà vu»
P3   pour tout v adjacent à u
P4       si v n'est pas marqué alors PARCOURS-PROFONDEUR(v)
P5   // post-visite de u
```

(Généralisation du parcours préfixe ou postfixe sur les arbres.)

Parcours en largeur

Utiliser une file FIFO (*queue*) Q

```
     Algo PARCOURS-LARGEUR(s)
L1   Q.enqueue(s)
L2   tandis que Q n'est pas vide
L3       u ← Q.dequeue()
L4       si u n'est pas marqué
L5           marquer u // «déjà vu»
L6           pour tout v adjacent à u faire
L7               si v n'est pas marqué alors Q.enqueue(v)
```

Temps de calcul : $O(|E|)$ avec listes d'adjacence ou $O(|V|^2)$ avec matrice d'adjacence

Remarque : on peut faire le parcours aussi en enfilant les arêtes

Parcours

Idée générale : suivre toujours une arête qui mène d'un sommet visité à un sommet non-visité

Parcours par profondeur : choix d'arête/arc uv où u est visité le plus récemment (pile)

Parcours par largeur : choix d'arête/arc uv où u est visité le moins récemment (queue)

Arbre couvrant

Déf Un arbre couvrant du graphe non-orienté $G = (V, E)$ est un sous-graphe $T = (V, E')$ connexe acyclique.

Problème de l'arbre couvrant minimal (ACM) : arbre couvrant avec poids minimum quand les arêtes sont ponderées.

Méthode gloutonne : on construit l'ACM arête par arête, en incluant une petite arête ou en excluant une grande arête à la fois.

ACM (cont)

Déf. Une **coupure** $(X, \bar{X})$ d'un graphe non-orienté $G = (V, E)$ est une partition de ses sommets : $X \subset V, \bar{X} = V \setminus X$. L'arête uv traverse la coupure (ou y appartient) si $u \in X$ et $v \in \bar{X}$.

Idée de base : grandes arêtes dans les cycles sont rejetées, petites arêtes dans coupures sont acceptées.

Coloriage des arêtes : **rouge** (rejetée) ou **bleue** (acceptée).

Règle bleue : Choisir une coupure sans arête bleue. Choisir une arête non-coloriée avec poids minimal qui traverse la coupure et la colorier par bleue.

Règle rouge : Choisir un cycle élémentaire sans arête rouge. Choisir une arête non-coloriée dans le cycle avec poids maximal et la colorier par rouge.

ACM (cont)

Thm Soit G un graphe connexe.

1. Il existe toujours un ACM qui contient toutes les arêtes bleues et aucune arête rouge, quel que soit l'ordre d'application des règles.
2. On peut appliquer soit la règle rouge soir la règle bleue tandis qu'il existe des arêtes non-coloriées.

Preuve Induction pour 1. Au début c'est vrai : il existe un ACM T avec toutes les arêtes bleues et aucune arête rouge. Supposons qu'on applique la règle bleue en coloriant l'arête uv, et soit T l'ACM avant l'application. Si $uv \in T$, alors OK. Si $uv \notin T$, alors il y a un chemin $u \rightsquigarrow v$ dans T, avec une arête e' qui traverse la même coupure. On a $c(e) \leq c(e')$ et donc l'arbre $T' = T \cup \{e\} \setminus \{e'\}$ est un ACM. Argument similaire pour la règle rouge...

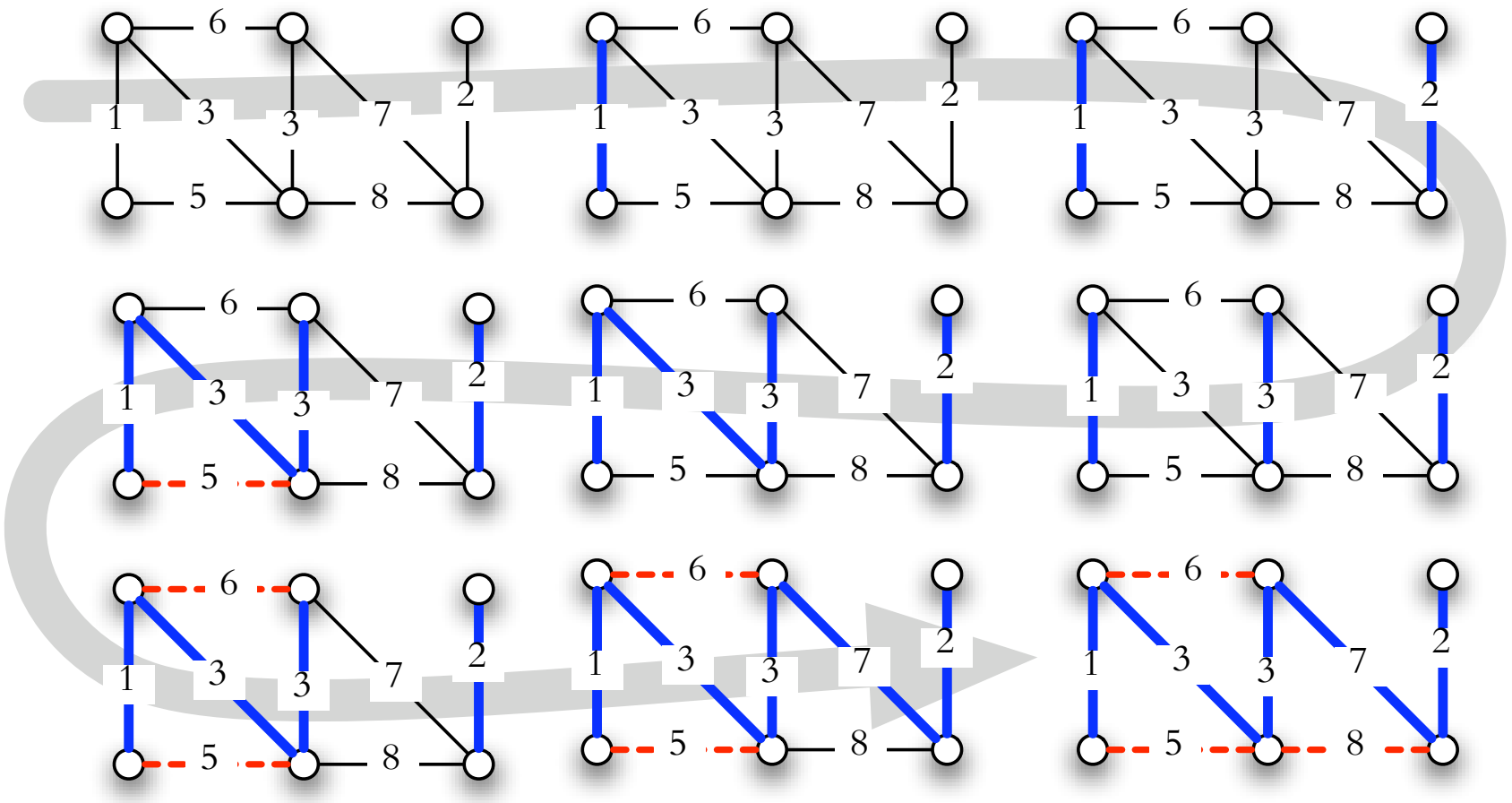
Pour démontrer 2, supposons par contradiction que la méthode finit avec des arêtes non-coloriées. Les arêtes bleues forment une forêt (ensemble d'arbres). S'il y a une arête e non-coloriée qui joint deux arbres bleus $\Rightarrow$ appliquer la règle bleue. S'il y a une arête e non-coloriée dont les extremités sont dans le même arbre $\Rightarrow$ appliquer la règle rouge. $\square$

ACM (cont)

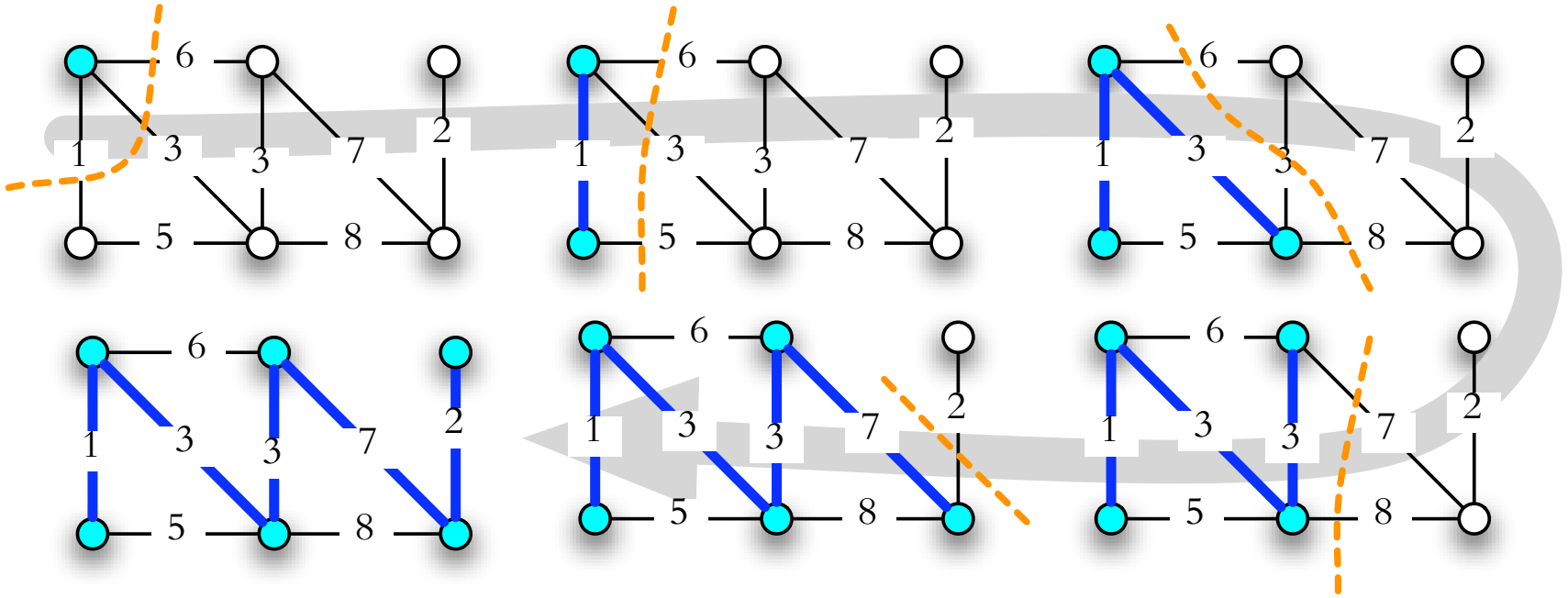
Algorithme de **Kruskal** : pour toute arête uv dans l'ordre non-décroissant, si u et v appartiennent au même arbre bleu, alors la colorier par rouge ; sinon par bleue

Algorithme de **Prim** (+Jarník et Dijkstra) : on maintient l'arbre T défini par les arêtes bleueus — appliquer la règle bleue à la coupure $(T, \bar{T})$.

Kruskal : exemple



Prim : exemple



Kruskal : code

Implantation utilise le tableau trié $E[1..m]$ des arêtes : $c(E[i]) \leq c(E[i+1])$ et une structure de données pour la connexité avec opérations $\mathsf{union}(x, y)$ et $\mathsf{find}(x)$ (vu au début du cours)

Algo ACM-KRUSKAL$(V, c, E[1..m])$

AK1 initialiser $T \leftarrow \emptyset$ // *(c'est l'ACM qu'on construira)*

AK2 **pour tout** $i \leftarrow 1, 2, \ldots, m$ **faire**

AK3 considérer l'arête $uv = E[i]$

AK4 **si** $\mathsf{find}(u) \neq \mathsf{find}(v)$ **alors** ajouter uv à T

AK5 $\mathsf{union}(u, v)$

AK6 retourner T

Kruskal

structures de données

«pour toute arête uv dans l'ordre non-décroissant»
trier les arêtes (ou utiliser un tas)

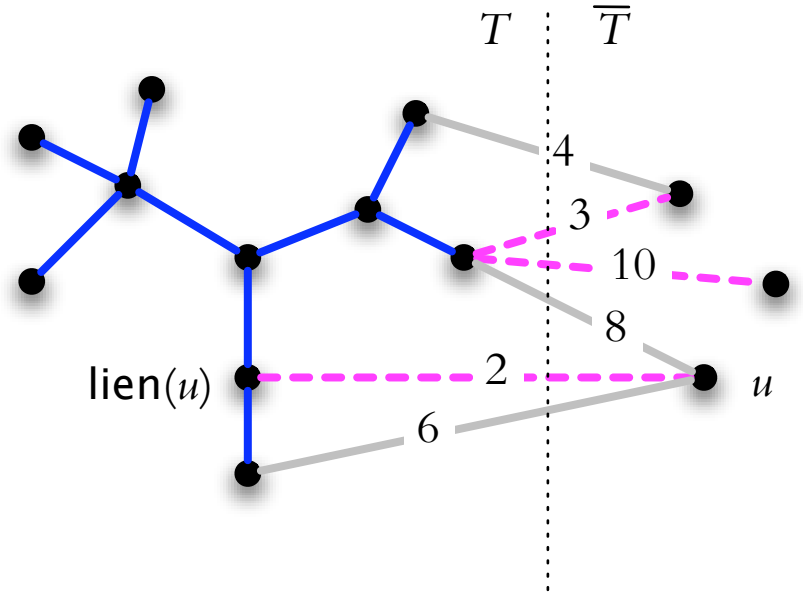
«si u et v appartiennent au même arbre bleu»
on doit tester connexité — utiliser une structure de données pour Union-Find

$|V| = n$ sommets, $|E| = m$ arêtes

Temps de calcul : $O(m \log m) \subseteq O(m \log n)$ pour trier, $O(m\alpha(m,n))$ pour appartenance

Total : $O(m \log n + m\alpha(m,n)) = O(m \log n + m \log^* n) = O(m \log n)$

Prim



Idée de l'implantation : pour chaque $u \notin T$ la meilleure arête $(u, \mathsf{lien}(u))$ dans la coupure $(T, \overline{T})$ est placée dans un tas

Prim : code

Implantation utilise un tas H, on a besoin d'un sommet de départ s

```
      Algo ACM-PRIM(V, c, s)
AP1   pour tout u ∈ V initialiser cle(u) ← ∞ ; lien(u) ← null
AP2   cle(s) ← 0 ; initialiser H ← ∅ ; H.insert(s)
AP3   tandis que H ≠ ∅
AP4       u ← H.deleteMin() ; cle(u) ← −∞
AP5       ajouter arête (u, lien(u)) si lien(u) ≠ null
AP6       pour tout v ∈ Adj[u] faire
AP7           si c(u, v) < cle(v) alors
AP8               cle(v) ← c(u, v) ; lien(v) ← u
AP9               si v ∉ H alors H.insert(v) sinon H.decreaseKey(v)
```

Prim (cont)

$|V| = n$, $|E| = m$

n fois deleteMin(),
n fois insert(),
$m - n + 1$ fois decreaseKey().

Temps de calcul avec tas d-aire :

$O(nd \log_d n + m \log_d n)$, choisir $d = \lceil 2 + m/n \rceil$.
$\Rightarrow$ temps de $O(m \log_{2+m/n} n)$. Quand $m = \Omega(n^{1+\epsilon})$ avec $\epsilon > 0$, on a $O(m/\epsilon)$.

Temps de calcul avec tas Fibonacci (permet decreaseKey en $O(1)$ temps amorti) :

$O(m + n \log n)$