CHPS0742
Recherche Opérationnelle

Cours 3

Graphes

UNIVERSITÉ DE REIMS CHAMPAGNE-ARDENNE

Pierre Delisle
Université de Reims Champagne-Ardenne
Département de Mathématiques et Informatique
Septembre 2018

Plan de la séance

- Graphes
  - Généralités : théorie et représentation
  - Arbre couvrant de poids minimum
  - Problèmes de plus courts chemins
  - Parcours

Généralités sur les graphes

Graphes ?

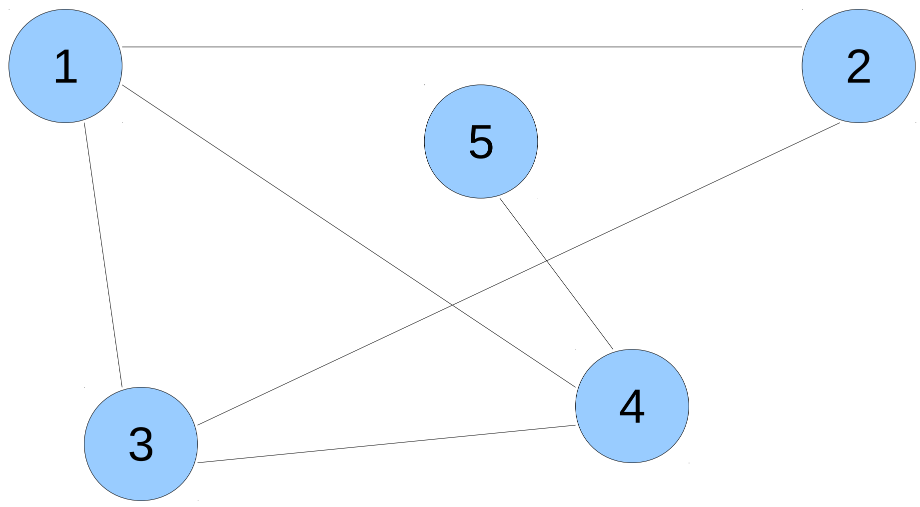
- Interviennent dans de nombreux problèmes
  - Modélisation de réseaux
  - Représentation d'un grand nombre de problèmes en recherche opérationnelle
- On verra plus tard un lien avec la programmation linéaire

Graphe non orienté

- Un graphe non orienté $G = (X, E)$ …
- … est défini par deux ensembles
  - Ensemble X des *sommets*
  - Ensemble E des *arêtes*
- Une arête, un élément e de E
  - Est défini par une paire de sommets distincts x et y de X
  - N'apparaît pas plusieurs fois dans E
- On dit que
  - x et y sont *incidents* à e
  - x et y sont les *extrémités* de e
  - x et y sont *adjacents*

$e = \{x, y\}$

Graphe non orienté



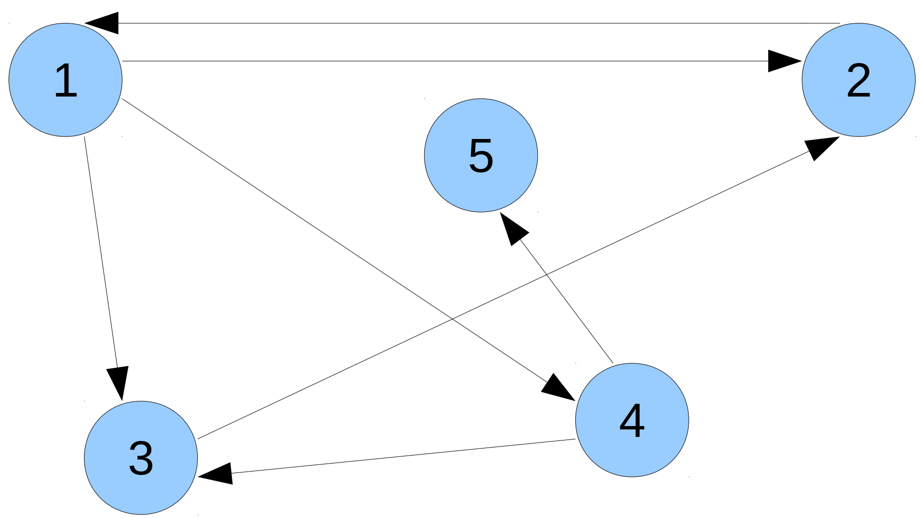
- $X = \{1, 2, 3, 4, 5\}$
- $E = \{(1, 2), (1, 3), (1, 4), (2, 3), (3, 4), (4, 5)\}$
- 1 et 2 sont adjacents
- 2 et 5 ne sont pas adjacents

Graphe orienté

- Un graphe orienté $G = (X, U)$ …
- … est défini par deux ensembles
  - Ensemble X des *sommets*
  - Ensemble U des *arcs*
- Un arc, un élément u de U
  - Est défini par un couple de sommets distincts x et y de X
  - N'apparaît pas plusieurs fois dans U
  - Mais on peut avoir (x,y) et (y, x), qui sont deux arcs distincts
- On dit que
  - u admet x comme *origine*, ou *extrémité initiale*
  - u admet y comme *extrémité finale* ou *terminale*

$u = \{x, y\}$

Graphe orienté

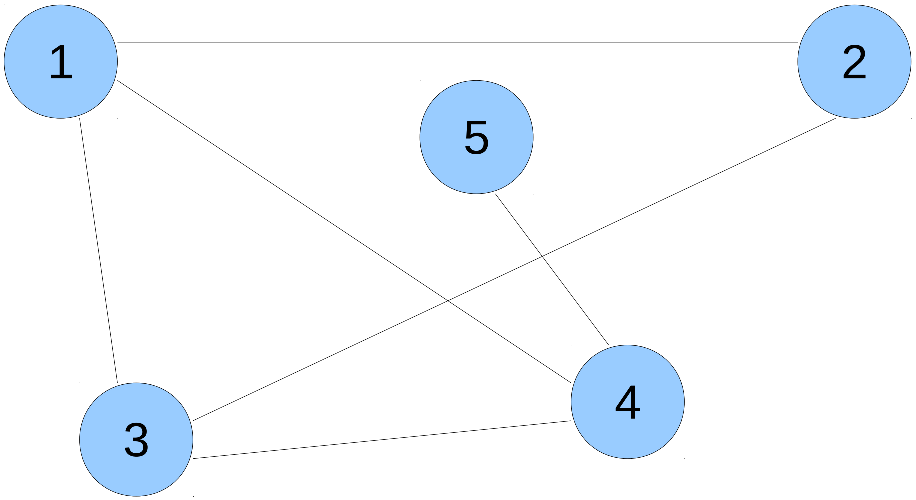


- $X = \{1, 2, 3, 4, 5\}$
- $U = \{(1, 2),(1, 3),(1, 4),(2, 1),(3, 2),(4, 3),(4, 5)\}$

Graphe fini

- Le cardinal de X
  - Est appelé *ordre* du graphe (nombre de sommets)
  - $n = |X|$
- Le cardinal de E ou de U
  - Est appelé *taille* du graphe (nombre d'arêtes ou arcs)
  - $m = |E|$ ou $m = |U|$
- Les cardinaux de X, E et U sont finis

Graphe fini

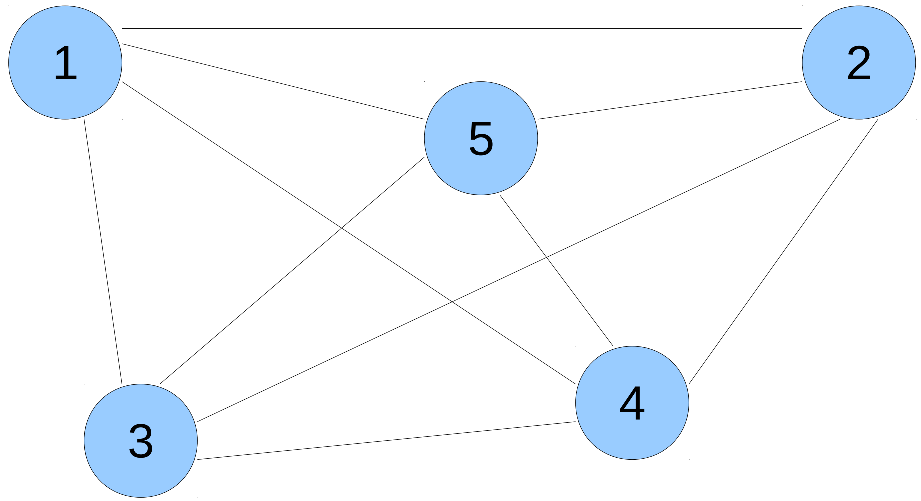


- L'ordre du graphe *n* est 5
- La taille du graphe *m* est 6

Graphe complet (ou clique)

- Un graphe complet à n sommets …
  - Noté K_n
- … est un graphe non orienté d'ordre n dont deux sommets quelconques sont adjacents
  - Il est donc de taille $n(n-1) / 2$

Graphe complet



- Graphe complet d'ordre
  - 5
- La taille du graphe est de
  - 5 * 4 / 2 = 10

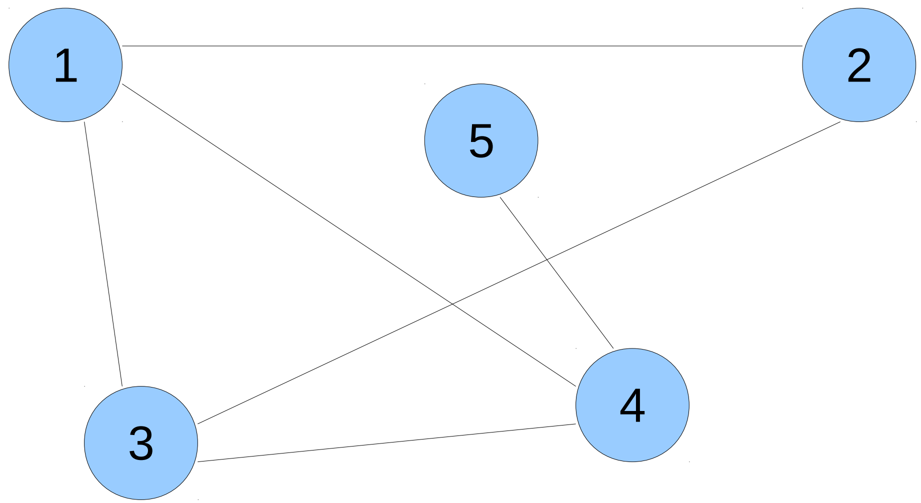
Graphe partiel

- Un graphe partiel de $G = (X, E)$ est un graphe
  - Ayant le même ensemble de sommets X que G
  - Ayant pour ensemble d'arête une partie de E

Sous-graphe

- Étant donnée une partie *Y* de *X*
- Un sous-graphe *F* de *G* engendré par *Y*
  - Est un graphe ayant pour ensemble de sommets *Y*
  - Une arête (arc) de *G* donnant naissance à une arête (arc) de *F* si et seulement si les deux extrémités de cette arête (arc) sont dans *Y*
- Autrement dit, un sous graphe *F* d'un graphe *G* est un graphe composé de certains sommets de *G* et de toutes les arêtes qui relient ces sommets dans *G*

Sous-graphe

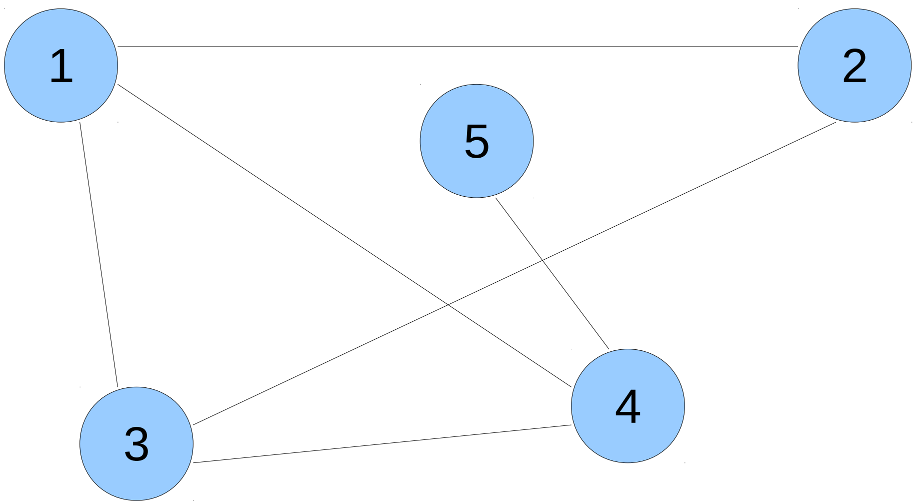


- *G' = ({1, 2, 3, 4, 5}, {(1, 2),(2, 3),(3, 4),(4, 5)}) est un graphe partiel*
- *G*' = ({1, 2, 3}, {(1, 2), (1, 3), (2, 3)}) est un sous-graphe complet d'ordre 3

Degré d'un sommet

- Étant donné un sommet x d'un graphe G non orienté
  - Le degré de x est le nombre d'arêtes incidentes à x
  - Les autres extrémités de ces arêtes constituent l'ensemble des voisins de x
- Autrement dit, le degré d'un sommet est le nombre d'arêtes dont ce sommet est une extrémité
- Dans un graphe orienté, on parle de degré entrant et sortant selon l'orientation des arcs

Degré d'un sommet

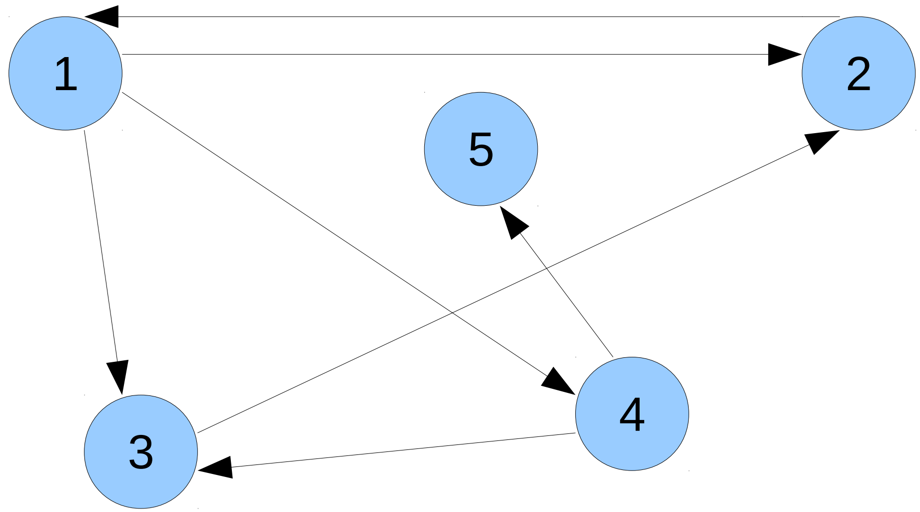


- Degré de 1 : 3, Degré de 5 : 1
- 3, 4 et 2 sont les voisins de 1

Prédécesseur et Successeur

- *y* est un *prédécesseur* de *x*
  - Si l'arc (*y*,*x*) existe
- *y* est un *successeur* de *x*
  - Si l'arc (*x*,*y*) existe
- Quand le prédécesseur de *x* est unique
  - Ce sommet est le *père* de *x*
  - *x* est un *fils* de ce sommet

Prédécesseur et Successeur

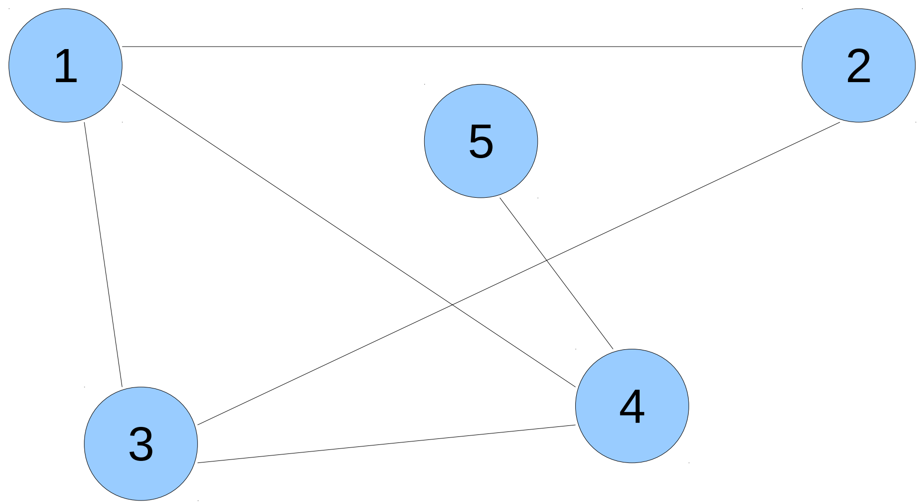


- 1 est un prédécesseur de 3
- 5 est un successeur de 4
- 1 est le père de 4
- 4 est le fils de 1

Chaîne et Cycle

- Soit $G = (X, E)$ un graphe non orienté
- Une chaîne est une suite
  - $x_1 e_1 x_2 e_2 \ldots x_{k\text{-}1} e_{k\text{-}1} x_k$
  - Avec $x_i \in X$, $e_j \in E$ et $e_j = \{x_j, x_{j+1}\}$
- Autrement dit, une chaîne est une liste ordonnée de sommets telle que chaque sommet de la liste soit adjacent au sommet suivant
- Un cycle est une chaîne dont les deux extrémités coïncident et composée d'arêtes distinctes

Chaîne et Cycle

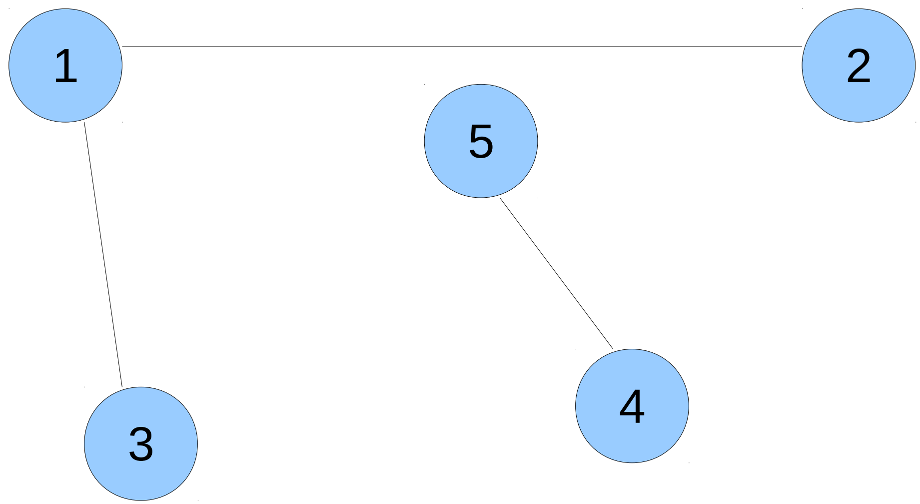


- (2, 1, 3, 4, 5) est une chaîne de longueur 4
- (1, 2, 3, 4, 1) est un cycle de longueur 4

Graphe connexe

- Un graphe est connexe si, pour toute paire de sommets, il existe une chaîne les joignant
- Une composante connexe d'un graphe est un sous-graphe connexe maximal
  - On ne peut y ajouter d'autres sommets en conservant la connexité du sous-graphe
- À noter d'un arbre est un graphe connexe et sans cycle

Graphe connexe

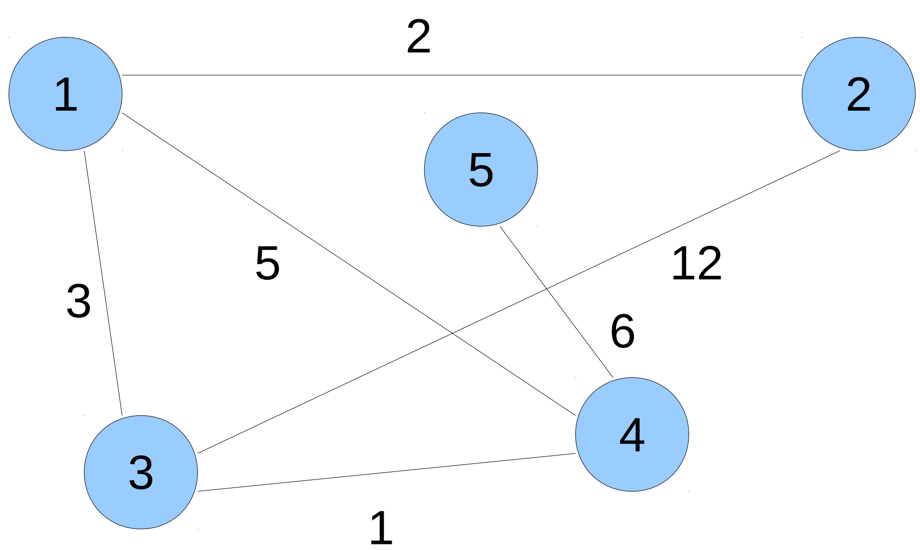


- Ce graphe n'est pas connexe
- Ce graphe possède 2 composantes connexes
- Les 2 sous-graphes sont des arbres

Graphe pondéré

- Graphe dont les arêtes ou les arcs sont munis d'une valuation
  - Coût ou poids ou longueur
- Le coût (ou poids, ou longueur) d'un graphe partiel ou d'une chaîne est alors la somme des coûts des arêtes ou des arcs le constituant
- Une plus courte chaîne entre deux sommets est, parmi les chaînes qui la relient, une chaîne de poids minimum

Graphe pondéré



- Le poids de la chaîne (2, 1, 3, 4, 5) est 12
- C'est la plus courte chaîne reliant 2 et 5

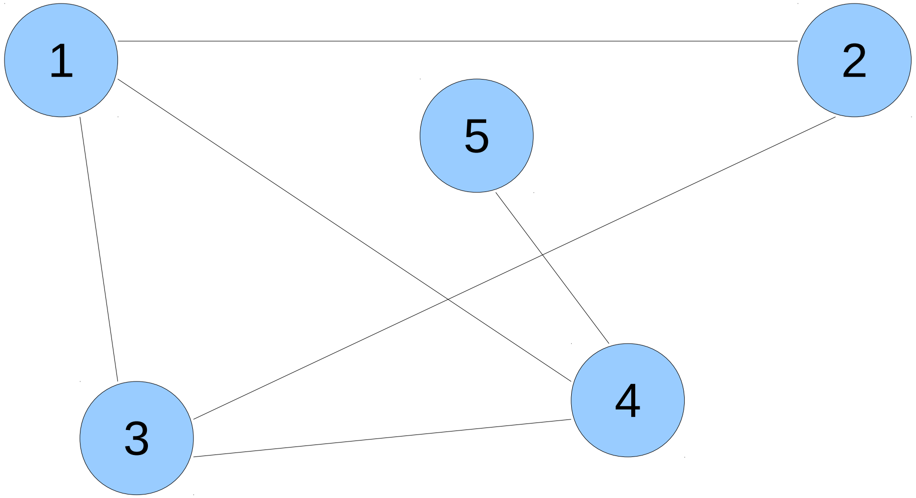
Représentation informatique

- On suppose les sommets de *G* numérotés par les entiers de 1 à *n*, chaque sommet étant repéré par son numéro
- Deux structures de données sont souvent utilisées pour représenter un graphe
  - Matrice d'adjacence
  - Tableau de listes d'adjacence

Matrice d'adjacence

- On suppose donnés (ex. lus dans un fichier)
  - L'ordre n du graphe
  - L'ensemble des arêtes $\{i, j\}$ du graphe
- On représente le graphe par une matrice d'adjacence, c'est-à-dire une matrice carrée *adj* à n lignes et n colonnes, telle que
  - $adj[i, j] = 1 = adj[j, i]$ si i et j sont adjacents
  - $adj[i, j] = 0$ sinon
  - $adj[k, k]$ sera pris égal à 0 ou 1 selon le problème

Matrice d'adjacence

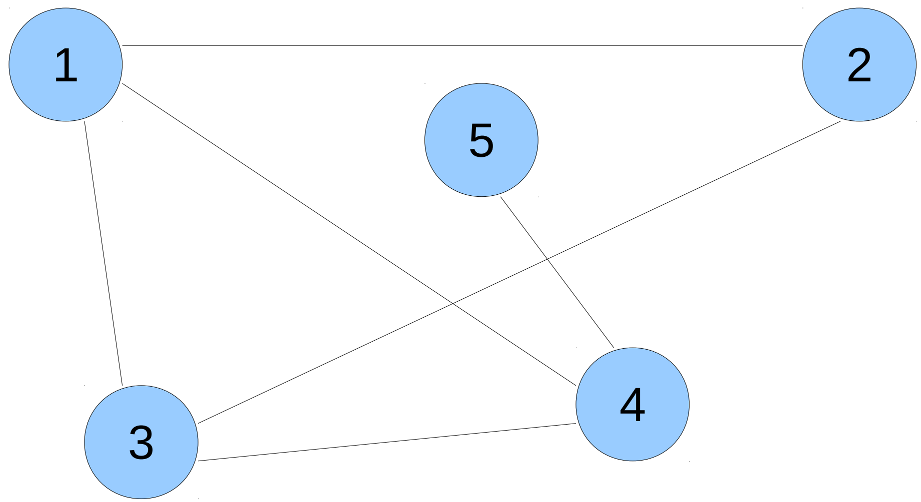


adj	1	2	3	4	5
1	0	1	1	1	0
2	1	0	1	0	0
3	1	1	0	1	0
4	1	0	1	0	1
5	0	0	0	1	0

Tableau de listes d'adjacence

- On suppose donnés (ex. lus dans un fichier)
  - L'ordre *n* du graphe
  - L'ensemble des arêtes {*i*, *j*} du graphe
- On définit un tableau de pointeurs de taille *n*
  - Les *n* cases du tableau sont en bijection avec les sommets 1, 2, … *n* du graphe
  - Le pointeur en position *i* est la tête d'une liste chaînée qui contient les voisins du sommet *i*
    - Un voisin *j* de *i* doit appartenir à la liste correspondant au sommet *i*, et *i* à la liste de *j*

Tableau de listes d'adjacence



1	→ 2 → 3 → 4
2	→ 1 → 3
3	→ 1 → 2 → 4
4	→ 1 → 3 → 5
5	→ 4

Remarques

- Le choix entre ces deux structures dépend
  - De l'algorithme que l'on désire implémenter
  - De la taille du graphe : on peut sauver de l'espace mémoire par le tableau de listes d'adjacence si le nombre d'arêtes est petit
- On peut facilement généraliser leur utilisation
  - Pour des graphes orientés
  - Pour des graphes pondérés
- D'autres peuvent être imaginées selon l'algo.

Complexité d'un algorithme

Complexité en quelques mots

- Paramètre important pour mesurer l'efficacité d'un algorithme
- Évalue un majorant du nombre d'opérations élémentaires qu'on doit effectuer, dans le pire des cas, pour obtenir le résultat recherché
- Opérations élémentaires
  - Comparaison, affectation, op. arithmétique, etc., …
  - … appliquées à des types simples (entier, réel, …)
- Exprimé en fonction de la taille des données

Un peu plus formellement...

- Soit un algorithme *A* ...
- … permettant de résoudre un problème *P*
- Soit *I* une instance de *P*
  - Spécification des données du problème à traiter
- Soit *f*(*A*,*I*) le nombre d'opérations élémentaires effectuées pour passer de *I* au résultat voulu à l'aide de *A*
- La complexité C_A de *A* est définie comme la fonction qui, sur l'ensemble des instances *I* de taille fixée, considère le maximum de *f*(*A*,*I*)
  - $C_A(n) = \max \{f(A,I)$ pour toute instance I telle que $|I| = n\}$
  - Où $|I|$ représente la taille de *I*

Un peu plus formellement...

- Il est souvent difficile de déterminer de façon précise l'expression de C_A
  - On ne s'intéresse donc qu'à un majorant asymptotique de $C_A(n)$
  - que l'on note $O(g(n))$
  - Signifie qu'il existe une constante K et une valeur N_K telles qu'on ait, pour tout n vérifiant $n >= N_K$
  - $|C_A(n)| <= K * |g(n)|$

Un peu plus formellement...

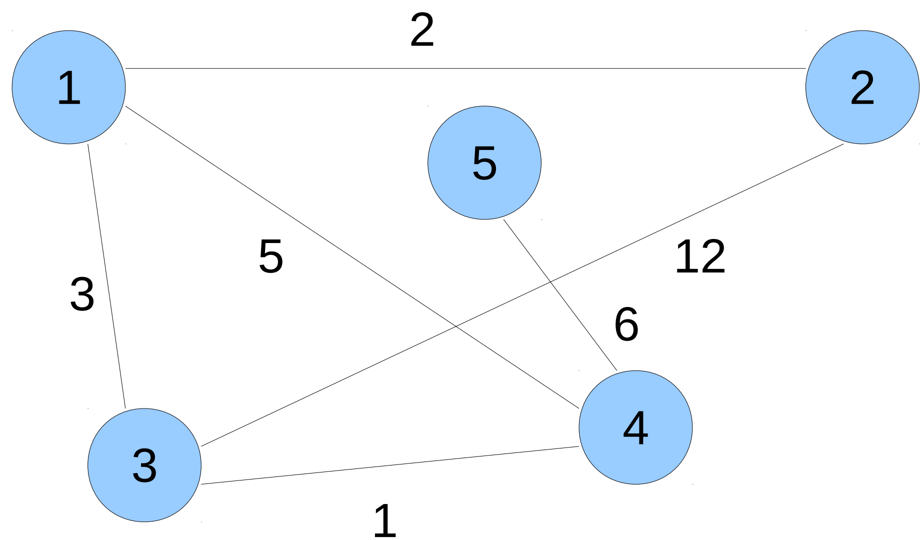
- Les algorithmes dont la complexité est majorée par un polynôme n^k de la taille n des données
  - Sont dits polynomiaux : en $O(n^k)$
  - Les autres sont dits exponentiels : ex. n^n
- La complexité donne des indications sur le temps de calcul
- Pour un algorithme linéaire, donc en $O(n)$
  - On peut prévoir qu'on problème de taille 2 fois plus grande prendra (à peu près) 2 fois plus de temps
  - Pour un algorithme en $O(n^3)$: 8 fois plus de temps
- On préférera généralement des algos de faible complexité

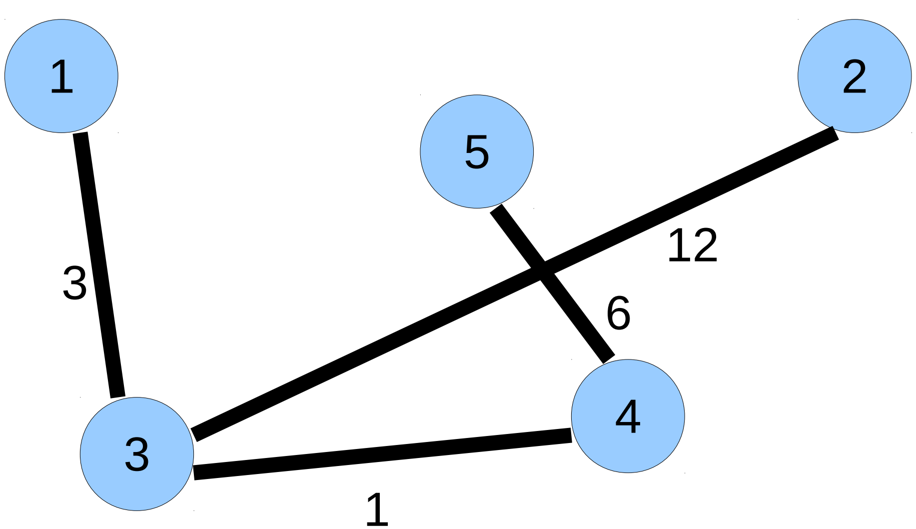
Problème de l'arbre couvrant de poids minimum

Définition du problème

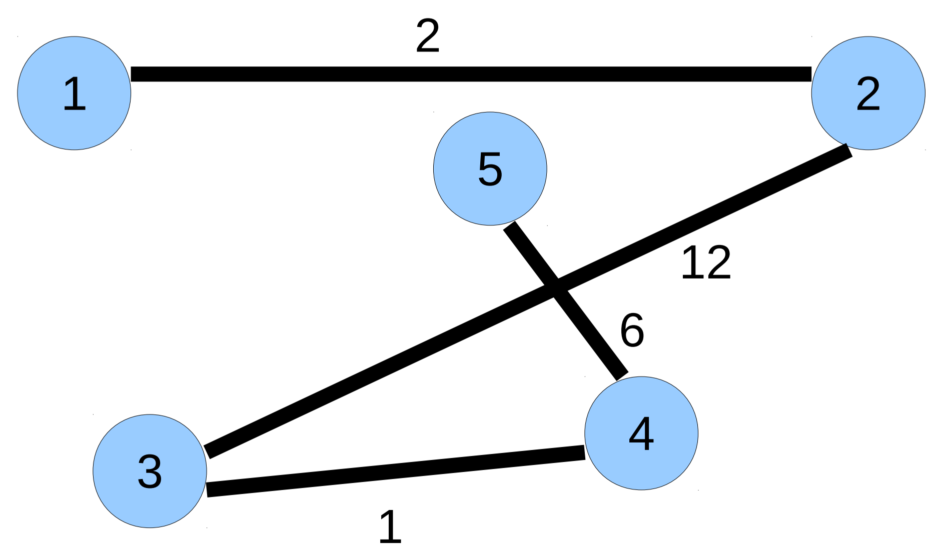
- Étant donné un graphe non orienté pondéré
- On cherche un graphe partiel de ce graphe
- … qui soit un arbre …
- … et qui soit de coût minimum
- Graphe partiel : l'arbre a pour ensemble de sommets tous les sommets du graphe initial
  - On dit que c'est un arbre couvrant
- En anglais : minimum spanning tree
- On suppose le graphe initial connexe

Arbre couvrant





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 de l'arbre couvrant minimum, puis on sépare

Algorithme 1

Algorithme de Kruskal

Principe de l'algorithme

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

Mise en oeuvre

- Trier les arêtes par ordre de poids croissants
- Tant qu'on n'a pas retenu $n - 1$ 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 (x,y) 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

Principe : gérer l'évolution des composantes connexes

- 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 {*x*, *y*} est candidate
  - On compare les valeurs des indices de *x* et *y*
    - Si valeurs égales, *x* et *y* sont dans la même composante connexe → ajouter l'arête créerait alors un cycle donc on ne la retient pas
    - Si valeurs différentes, on garde l'arête {*x*, *y*}, on donne à l'indice de *y* la valeur de l'indice de *x*, ainsi que tout sommet d'indice *y*
    - Autrement dit, après sélection de l'arête {*x*,*y*}, tous les sommets de la composante connexe de *x* et de la composante connexe de *y* ne forment qu'une seule composante connexe et ont le même indice

Structures de données

- *tabCC*
  - Tableau d'entiers associés aux sommets 1 à *n* 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 *n* - 1

Algorithme de Kruskal

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

Complexité de l'algorithme

- Trier les arêtes
  - $O(m \log_2 m)$ opérations (m est la taille du graphe)
- Autres initialisations
  - $O(n)$
- Examiner les arêtes candidates
  - $O(1)$ en cas de refus, $O(n)$ en cas de retenue
- On doit retenir $n - 1$ arêtes
  - $O(n^2)$ pour les arêtes retenues + $O(m)$ pour les refus
  - Donc $O(n^2)$ pour cette partie
- Donc, au total
  - $O(n^2 + m \log_2 m)$
  - Ou $O(n^2 \log_2 n)$ si on considère m de l'ordre de n^2

Algorithme 2

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

Description de l'algorithme

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

Description de l'algorithme

- On choisit alors de faire entrer dans *tabArbre* le sommet *x* dont la distance *d*(*x*) 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*)

Algorithme de Prim

$tabArbre \leftarrow \{x_0\}$ (x_0 étant un sommet quelconque)

$pivot \leftarrow \{x_0\}$

POUR tout sommet x autre que x_0 FAIRE

    $d(x) \leftarrow +\infty$

POUR i de 1 à $n - 1$ FAIRE

    POUR tout sommet z voisin de *pivot* non dans *tabArbre* FAIRE

        SI $p(pivot, z) < d(z)$ ALORS

            proche(z) $\leftarrow$ *pivot*

            $d(z) \leftarrow p(pivot,z)$

    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*

Complexité de l'algorithme

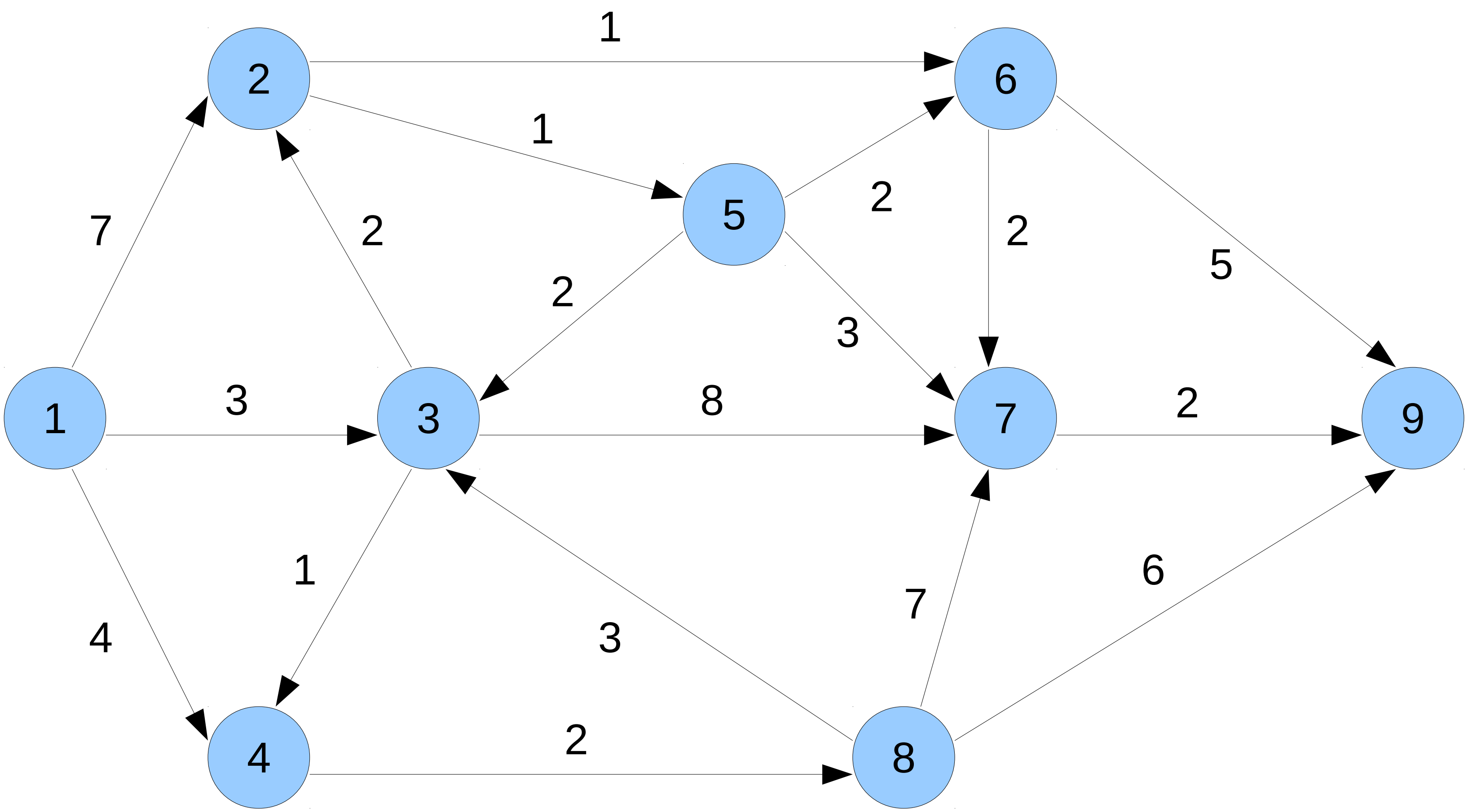
- Initialisations : O(*n*)
- Pour chacun des *n*-1 passages dans la 1ère boucle POUR
  - Il y a au plus *n* – 1 sommets voisins de pivot dont il faudra mettre à jour les attributs
    - Les mises à jour se font en O(1)
    - Donc O(*n*) pour la deuxième boucle POUR
  - La détermination du pivot suivant revient à trouver le minimum d'au plus *n* valeurs
    - Donc O(*n*)
  - Mise à jour de *tabArbre* : O(1)
- Donc, au final
  - $O(n^2)$

Problèmes de plus courts chemins

Chemin, chemin élémentaire et circuit

- Soit $G = (X, U)$ un graphe orienté et valué
  - On attribue des poids (longueurs, coûts) aux arcs
- Un chemin est une suite
  - $x_1 e_1 x_2 e_2 \ldots e_n x_{n+1}$
  - de sommets x_i et d'arcs e_j
  - telle que l'arc e_j ait pour origine x_j et pour extrémité x_{j+1}
- Poids du chemin : somme des poids des arcs
- Chemin élémentaire : n'utilise pas 2 fois le même sommet
- Circuit : chemin dont les extrémités coïncident

Exemple

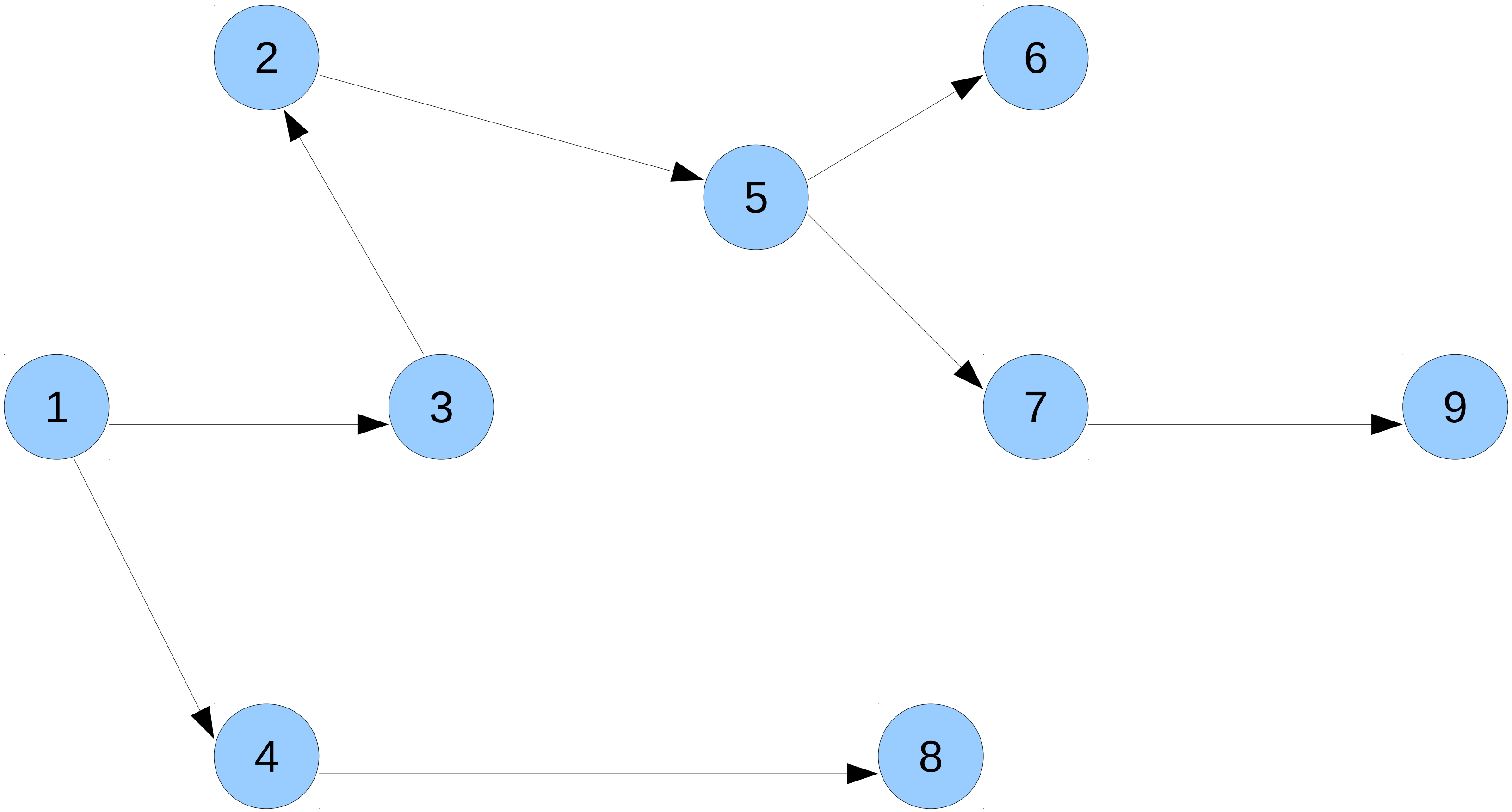


- 1, 3, 7, 9 est un chemin élémentaire de poids 13 entre 1 et 9
- 1, 4, 8, 9 est un chemin élémentaire de poids 12 entre 1 et 9
- 2, 5, 3, 2 est un circuit de poids 5

Arborescence

- La racine d'un graphe *G* orienté est un sommet *r* tel qu'il existe un chemin de *r* à tout sommet de *G*
- Une arborescence de racine *r* est un graphe orienté tel que
  - Le graphe non orienté sous-jacent (en oubliant les orientations) est un arbre
  - Pour tout sommet *x*, l'unique chaîne entre *r* et *x* du graphe non orienté sous-jacent correspond à un chemin de *r* vers *x* dans l'arborescence

Exemple



- Arborescence dont la racine est le sommet 1
- Le graphe non orienté sous-jacent est un arbre
- L'unique chaîne entre 1 et tout sommet est un chemin

Remarques

- Un graphe orienté A est une arborescence de racine r si et seulement si
  - Le graphe non orienté sous-jacent est un arbre
  - Le degré entrant de r est égal à 0
  - Le degré entrant de tout sommet x différent de r est égal à 1
- On connaît donc une arborescence lorsque l'on connaît, pour chaque sommet, son père dans l'arborescence

Remarques

- Pour la suite, on suppose qu'il existe toujours
  - un chemin d'un sommet vers un autre
  - un chemin d'un sommet vers tous les autres
- On peut toujours considérer qu'on travaille avec un graphe complet
  - S'il n'existe pas d'arc entre les sommets x et y, on peut en ajouter un de poids infini
  - S'il n'existe pas de chemin entre x et y, le plus court chemin entre x et y sera de poids infini

Définition du problème

- On considère les 3 problèmes suivants
- Problème 1
  - Étant donnés deux sommets *x* et *y*, trouver un chemin de poids minimum (ou plus court chemin) de *x* à *y*
- Problème 2
  - Étant donné un sommet *x*, trouver un plus court chemin de *x* à tous les sommets du graphe
  - Revient à déterminer une arborescence de racine *x* couvrant les sommets du graphe et constituée de plus courts chemins de *x* aux autres sommets

Définition du problème

- Problème 3
  - Trouver, pour toute paire de sommets *x* et *y*, un plus court chemin de *x* à *y*
- Il existe des algorithmes polynomiaux pour résoudre ces problèmes
- On ne considère pas le cas des graphes où il y a des circuits absorbants (de poids négatif)
  - Le problème ne peut alors être résolu en temps polynomial, il faut presque faire une énumération

Remarques

- On ne connaît pas, pour résoudre le problème 1, de bien meilleure solution que de résoudre le problème 2
- Pour résoudre le problème 3, on peut soit
  - Résoudre le problème 2 à partir de tout sommet (*n* fois pour un graphe d'ordre *n*)
  - Utiliser une méthode matricielle
- Le problème 2 est donc central
- Lorsque tous les poids sont positifs → Dijkstra

Principe de l'algorithme de Dijkstra

- On étend une arborescence …
- … initialement réduite à *r* …
- … en gagnant à chaque étape un sommet non encore couvert et un arc dont ce sommet est l'extrémité (l'origine étant déjà dans l'arborescence)
- L'ensemble des sommets couverts par l'arborescence à une étape donnée est noté *arbre*
- À la fin de l'algorithme cette arborescence donne, pour chaque sommet *x* du graphe, un plus court chemin de *r* à *x* : l'arborescence des plus courts chemins de *r* à *x*

Deux fonctions à définir

- Fonction *distance*(*x*)
  - À la fin de l'algorithme, donnera la longueur du plus court chemin entre le sommet *r* et le sommet *x*
  - À une itération quelconque, donne la longueur d'un plus court chemin de *r* à *x* n'utilisant comme sommets intermédiaires que ceux déjà dans *arbre*
- Fonction *père*(*x*)
  - À la fin de l'algorithme, donnera le père de *x* dans l'arborescence des plus courts chemins de *r* à *x*
  - À une itération quelconque, désigne le prédécesseur de *x* dans le plus court chemin de *r* à *x* n'utilisant que les sommets dans *arbre*

Algorithme de Dijkstra

arbre ← {*r*} (r étant la racine de l'arborescence)
pivot ← *r*
distance(r) ← *0*
POUR tout sommet *x* autre que *r* FAIRE
    distance(*x*) ← ∞
POUR *i* de 1 à *n* – 1 FAIRE
    POUR tout sommet *y* non dans *arbre* et successeur de *pivot* FAIRE
        SI *distance*(*pivot*) + *p*(*pivot*, *y*) < *distance*(*y*) ALORS
            distance(*y*) ← *distance*(*pivot*) + *p*(*pivot,y*)
            pere(*y*) ← *pivot*
    Chercher, parmi les sommets non dans *arbre*, un sommet *y* tel que distance(*y*) soit minimum
    pivot ← *y*
    arbre ← *arbre* ∪*pivot*

- Exemple

Complexité de l'algorithme

- Initialisation de *distance* : O(n)
- Insertion de n – 1 noeuds dans l'arbre : O(n)
- À chaque étape, l'actualisation de *distance*
  - O(d^+(*pivot*)) (nb d'arcs sortants de *pivot*) calculs
  - On a m arcs, donc O(m)
- À chaque étape, trouver *distance* minimum
  - n sommets, donc O(n)
- Au final : O(n^2)

D'autres algorithmes...

- Graphes sans circuit
  - Algorithme de Bellman : $O(n^2)$
- Cas général : valuations positives et négatives, circuits absorbants
  - Algorithme de Ford : $O(n^3)$
- Plus courts chemins de tout sommet à tout sommet
  - Algorithme de Dantzig : $O(n^3)$

Parcours de graphes

Définition

- Soit $G = (X, U)$ un graphe orienté …
- … et r un sommet de G
- Chaque sommet peut être ou non dans l'état « marqué »
  - Marquer un sommet : passer son état de « non marqué » à « marqué »
- Chaque arc peut être ou non dans l'état « traversé »
  - Traverser un arc (x, y) : regarder si son extrémité y est marquée
  - Ne peut être fait que lorsque x est marqué

Définition

- Un sommet *r* doit être choisi comme point de départ du parcours
  - On effectue alors un parcours à partir de *r*
- Pour tout sommet *x* dans *X* – {*r*}
  - On associe à *x* un sommet *père*(*x*)

Définition

- On peut alors définir tout algorithme de parcours à partir de *r* de la façon suivante :

Au départ, aucun sommet n'est marqué
Marquer *r*
TANT QUE Il existe un sommet marqué *x* et un arc (*x*, *y*) non traversé
    Choisir cet arc (*x*, *y*) et le traverser
    SI le sommet *y* n'est pas marqué ALORS
        Marquer le sommet *y*
        père(*y*) $\leftarrow$ *x*

- Les différents algorithmes de parcours se distinguent alors par la façon de choisir l'arc à traverser

Définition

- Si on note *M* l'ensemble des sommets marqués par un algorithme de parcours …
- … le graphe *A* ayant *M* pour ensemble de sommets …
- … et les arcs (*père*(*x*), *x*) pour *x* dans *M* – {*r*} …
- … est une arborescence de racine *r*
  - Elle est appelée *arborescence du parcours*
  - Les arcs (*père*(*x*), *x*) s'appellent *arcs arborescents*

Complexité

- Si le graphe est codé par des listes d'adjacence (voisins ou successeurs) de chaque sommet
  - Complexité : O(m) (m = nombre d'arcs ou arêtes)
- Si le graphe est codé par sa matrice d'adjacence
  - Complexité : O(n^2)
- Même complexité pour les graphes complets

Utilité

- Sert de base à plusieurs algorithmes
- Permet d'étudier les propriétés du graphe
  - Le graphe est-il connexe ?
  - Le graphe est-il biparti ?
- On pourra aussi faire des traitements sur les sommets et les arcs/arêtes durant le parcours
- 2 sortes de parcours
  - Parcours « Marquer-examiner »
  - Parcours en profondeur

Parcours marquer-examiner

- Utilisation d'une « liste d'attente » notée *L*
  - Au départ vide, contiendra des sommets

```
Mettre r dans la liste d'attente L
TANT QUE L n'est pas vide
    Retirer un sommet x
    POUR Tout arc (x, y)
        Traverser l'arc (x, y)
        SI y n'est par marqué ALORS
            Marquer y
            père(y) ← x
            Mettre y dans la liste d'attente
```

Parcours marquer-examiner

- On obtient un parcours différent selon l'ordre dans lequel les sommets sont choisis et retirés de la liste
  - File → Parcours en largeur
  - Pile → Proche du parcours en profondeur
- Exemples

Parcours en profondeur

- En anglais : *Depth First Search* (*DFS*)
  - À partir du sommet $r \rightarrow DFS(r)$
- Peut être défini de façon itérative
  - Définition d'un sommet courant
- Ou récursive
  - Pour chaque sommet y non marqué, on appelle $DFS(y)$

DFS itératif → *DFS*(*r*)

```
Marquer r
courant ← r
TANT QUE L'algorithme n'est pas terminé
    x ← courant
    SI Il existe un arc (x, y) non traversé ALORS
        Traverser l'arc (x, y)
        SI y n'est pas marqué ALORS
            Marquer y
            père(y) ← x
            courant ← y
    SINON
        SI x != r ALORS
            courant ← père(x)
        SINON
            L'algorithme est terminé
```

DFS récursif → *DFS*(*x*)

```
Marquer x
POUR Tout arc (x, y) FAIRE
    Traverser l'arc (x, y)
    SI y n'est pas marqué ALORS
        père(y) ← x
        DFS(y)
```

- Exemple

Numérotation préfixe et postfixe

- Lorsqu'on effectue un parcours en profondeur, on peut associer une numérotation aux sommets
- Numérotation préfixe
  - Attribué au moment où un marque un sommet
- Numérotation postfixe
  - Attribué au moment où on revient en arrière
- Exemple

Synthèse

- On a vu différents problèmes de graphes …
  - Arbre couvrant de poids minimum
  - Plus court chemins
  - Parcours
- … qui ont pour solution des algorithmes polynomiaux
  - $O(n^2)$ et $O(n^3)$
  - Ce sont des problèmes « faciles »
- Maintenant → les problèmes « difficiles »

La semaine prochaine

Résolution de problèmes NP-difficiles par des méthodes exactes