Les graphes

D.E ZEGOUR
École Supérieure d'Informatique
ESI

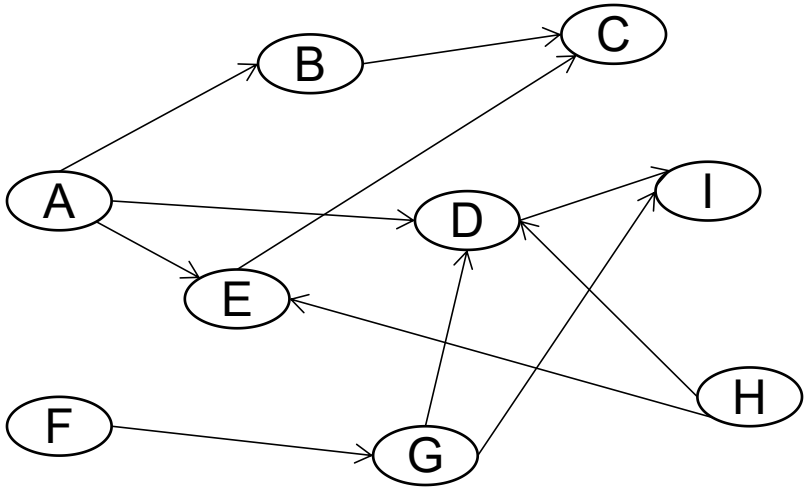
Les graphes

Introduction

Les graphes sont des modèles pour représenter des relations entre objets(Graphe orienté)

Si les relations sont symétriques, on parlera de graphe non orienté

Les sommets d'un graphe peuvent par exemple représenter des objets et les arcs des relations entre objets.



Les graphes

Graphes orientés

Ensemble de couples (u, v), u et v appartiennent à un ensemble de sommets(ou de nœuds). u est appelé la tête et v la queue.

A chaque couple(u, v) est associé un arc u-->v. Nous dirons que u est adjacent à v.

Un chemin est la séquence v1, v2, ...vn de sommets tels que v1→v2, v2 → v3, ...vn-1 → vn sont des arcs.

Longueur d'un chemin : nombre d'arcs qui le composent.

Un chemin simple : tous les nœuds, sauf éventuellement le premier et le dernier, sont distincts.

Circuit: chemin simple tel que v1=vn

Graphe étiqueté : Attribuer des valeurs aux arcs

Les graphes

Graphes orientés

Composante fortement connexe d'un graphe est composé de l'ensemble maximal de nœuds dans lequel il existe un chemin de tout nœud de l'ensemble vers chaque autre nœud de l'ensemble.

Formellement, soit G=(V, E) un graphe. On peut partitionner V en classes d'équivalence C1, C2, ...Cr

$v \equiv w \Leftrightarrow$ Il existe un chemin de v à w et un chemin de w à v.

Classe d'équivalence = Composante fortement connexe

Un graphe est dit fortement connexe s'il a une seule composante fortement connexe.

Les graphes

Graphes non orientés

Ensemble de paires ou d'arêtes(u, v). Chaque arête (u, v) représente en fait les deux arcs u-->v et v-->u.

On parlera de Chaine au lieu de chemin, Cycle au lieu de circuit,. Graphe connexe, etc.

Arbre : Graphe non orienté connexe sans cycle.

Les graphes

Machine abstraite

CréerNoeud(G, u)	Créer un nœud u du graphe G.
Liberernoeud(G, u)	Libérer le nœud d'adresse u du graphe G.
CreerArc(u, v, info)	Créer un arc de u vers v avec l'information info
LibererArc(u, v)	Libérer l'arc de u vers v
Info(u)	Info rattachée au nœud u
Aff_info(u, info)	Affecter la valeur info au nœud u
Arc(u, v)	Info rattachée à l'arc u – v
Aff_arc(u, v, info)	Affecter la valeur info à l'arc u - v
Adjacent(u, i)	Accès au I-ème nœud adjacent à u
Degre(u)	Nombre de nœuds adjacents à u
NoeudGraphe (G, i)	Accès au I-ème nœud du graphe G
NbreGraphe (G)	Nombre de nœuds du graphe G

Les graphes

Parcours des graphes

DFS : Depht First Search (Recherche en profondeur d'abord)

C’est le parcours le plus utilisé sur les graphes

C'est la généralisation du parcours Préordre sur les arbres

Les graphes

DFS : principe

Initialement tous les nœuds sont marqués " non visités".

Choisir un nœud v de départ et le marquer " visité".

Chaque nœud adjacent à v, non visité, est à son tour visité en utilisant DFS récursivement.

Une fois tous les nœuds accessibles à partir de v ont été visités, la recherche de v (DFS(v)) est complète.

Si certains nœuds du graphe restent "non visités", sélectionner un comme nouveau nœud de départ et répéter le processus jusqu'à ce que tous les nœuds soient visités.

Les graphes

DFS : Algorithme

```
Dfs(v):
Mark(v) := "visité"
Pour i:=1, Degre(v)
Si Mark(Adjacent(v, i)) = " non visité"
        Dfs(Adjacent(v, i))
Fsi
Finpour
```

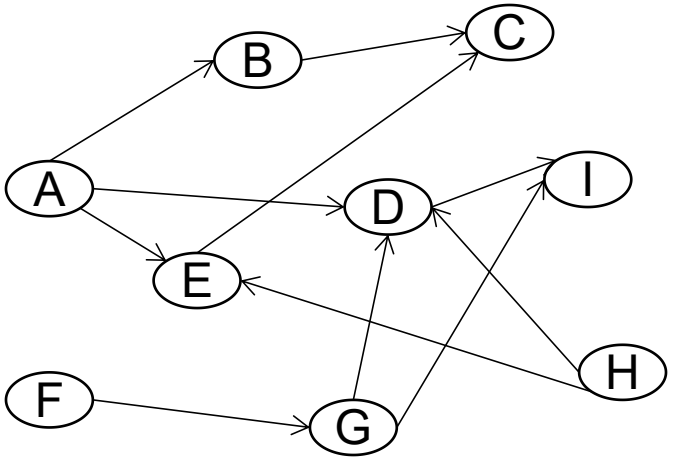
```
Initialisation :
Pour i:=1, Nbregraphe(G)
        Mark(Noeudgraphe(G, i)):="
non visité " Finpour

Appelant :
        Pour v =1, Nbregraphe(G)
            Si Mark(Noeudgraphe(G, i))
="non visité"
                DFS(Noeudgraphe(G, i))
            Fsi
        Finpour
```

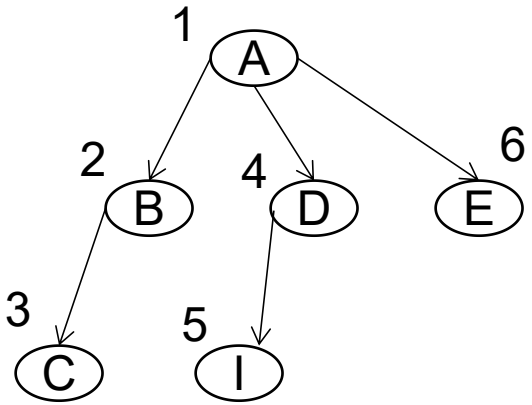
L'algorithme construit une forêt (implicite) de recouvrement des recherches.

Les graphes

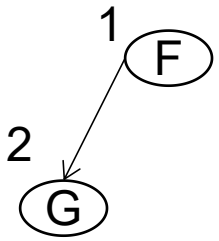
DFS : Exemple



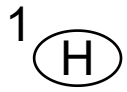
Choisir A



Choisir F



Choisir H



Les graphes

Parcours des graphes

BFS : Breadh First Search (Recherche en largeur)

C’est le parcours le moins utilisé sur les graphes

C'est la généralisation du parcours niveau par niveau défini sur les arbres

Les graphes

BFS : principe

Initialement tous les nœuds sont marqués " non visités".

Choisir un nœud v de départ et le marquer " visité".

Chaque nœud adjacent à v, non visité, est à son tour visité en utilisant BFS

Une fois tous les nœuds accessibles à partir de v ont été visités, la recherche de v (BFS(v)) est complète

Si certains nœuds du graphe restent "non visités", sélectionner un comme nouveau nœud de départ et répéter le processus jusqu'à ce que tous les nœuds soient visités.

Les graphes

BFS : Algorithme

```
BFS(v):
Mark(v) := "visité"
CreerFile(F) ; Enfiler(F, v)
Tantque Non Filevide(F) :
  Défiler(F, v)
  Pour i:=1, Degre(v)
   Si Mark(Adjacent(v, i))="non visité"
     Mark(Adjacent(v, i)):="visité"
     Enfiler(F, w)
     [Inserer( v-->w , Arbre )]
   Fsi
  Finpour
Fintantque
```

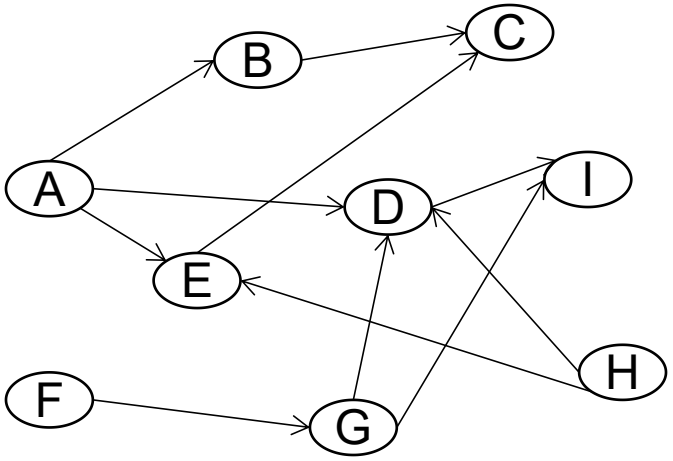
```
Initialisation :
Pour i:=1, Nbregraphe(G)
            Mark(Noeudgraphe(G, i)):=" non visité "
Finpour


Appelant :
Pour v =1, Nbregraphe(G)
   Si Mark(Noeudgraphe(G, i)) ="non visité"
      BFS(Noeudgraphe(G, i))
   Fsi
            Finpour
```

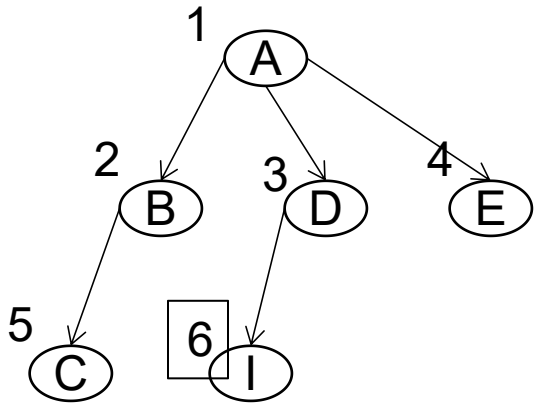
L'algorithme construit une forêt (implicite) de recouvrement des recherches.

Les graphes

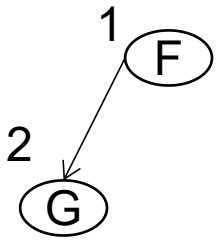
BFS : Exemple



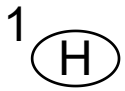
Choisir A



Choisir F



Choisir H



Les graphes

<u>Graphes : Application 1</u> → Détermination du plus court chemin (Algorithme de Dijkstra)

Soit G = (V, E) un graphe où V est l'ensemble des nœuds et E l'ensemble des arcs. Chaque arc a une valeur positive. Soit S un nœud quelconque.

Le problème consiste à déterminer le coût du plus court chemin de S vers tous les autres nœuds de G (longueur d'un chemin = somme des coûts des arcs sur ce chemin)

G peut être par exemple, une carte de vols aériens, dans laquelle les nœuds représentent les villes et chaque arc u-->v représente la ligne aérienne de u vers v.

La valeur de l'arc est donc le temps de vol de u vers v.

Les graphes

<u>Graphes : Application 1</u> → Détermination du plus court chemin (Algorithme de Dijkstra)

Considérations:

Soient V = { 1, 2, ..., n} et S = {1}.

C[1..n, 1..n] tableau de coûts tel que C[i, j] est le coût de l'arc i→j sinon c'est l'infini.

D[2..n] : distances du nœud 1 à tous les autres nœuds.

Le tableau D contiendra après exécution, les plus courtes distances de S vers tous les autres nœuds.

Les graphes

<u>Graphes : Application 1</u> → Détermination du plus court chemin (Algorithme de Dijkstra)

{Initialisation des distances de S vers tous les autres nœuds}
Pour i=2, n :
    D[i] := C[1, i]
Finpour

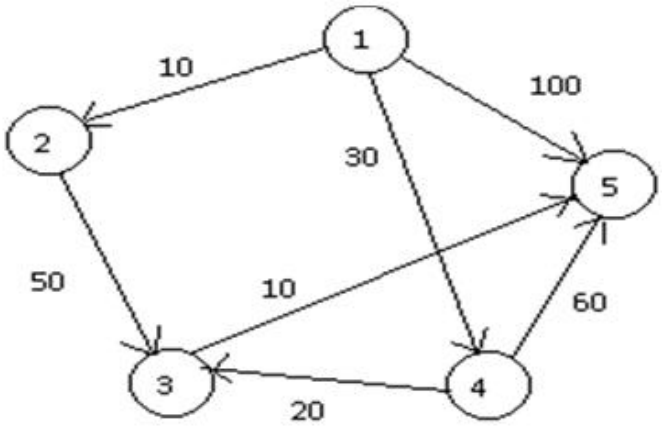
Pseudo-algorithme:
Pour i=1, n-1
    . Choisir un nœud w dans V-S tel que D[w] est minimale.
    . Ajouter w à S
    . Pour chaque nœud v dans V-S
        D[v]=Min(D[v],D[w]+C[w,v])
    Finpour
Finpour

Les graphes

Graphes : Application 1 → Détermination du plus court chemin (Algorithme de Dijkstra)

Exemple

S	{1}	{1,2}	{1,2,4}	{1,2,4,3}	{1,2,4,3,5}
w	-	2	4	3	5
D[2]	10	10	10	10	10
D[3]	∞	60	50	50	50
D[4]	30	30	30	30	30
D[5]	100	100	90	60	60



Les graphes

<u>Graphes : Application 2</u> → Problème du Voyageur du Commerce (PVC)

Un cycle Hamiltolnien dans un graphe non orienté est un chemin incluant tous les nœuds du graphe.

Le PVC consiste à trouver dans un graphe étiqueté non orienté un cycle Hamiltolnien de poids minimal.

En d'autres termes :

Étant donné n cités avec des distances entre chacune d'entre elles. Trouver un cycle Hamiltonien de poids minimal.

Les graphes

<u>Graphes : Application 2</u> → PVC (Algorithme de Krustal)

Pseudo_algorithme:

1. Trier toutes les arêtes(il existe C_n^2)

2. Prendre les arêtes une à une dans l'ordre en considérant les deux conditions suivantes :
   - Aucun sommet ne doit avoir un degré supérieur à 2.
   - Le seul cycle formé est le cycle final, quand le nombre d'arêtes acceptées est égal au nombre de sommets du graphe.

Les graphes

Graphes : Application 2 → PVC (Algorithme de Krustal)

Exemple

6 villes avec les coordonnées suivantes :

a=(0, 0) b=(4, 3) c=(1, 7)
d=(15, 7) e=(15, 4) f=(18, 0)

Il existe 6! = 720 permutations.

Par contre seulement 15 arêtes (ab, ac, ad, ae, af, bc, be, bd, be, bf, ...).

Les graphes

Graphes : Application 2 → PVC (Algorithme de Krustal)

Scénario

Choix de l'arête (d, e) car elle a une longueur égale à 3, la plus courte.

On examine ensuite les arêtes (b, c), (a, b) et (e, f) de poids 5.(ordre quelconque). Elles sont toutes les 3 acceptables conformément aux critères 1. et 2.

La prochaine arête la plus petite est (a, c) de poids 7,08. Comme elle forme un cycle avec(a, b) et (c, d) elle est rejetée.

L'arête (d,e) est écartée dans les mêmes conditions.

Les graphes

Graphes : Application 2 → PVC (Algorithme de Krustal)

Scénario (suite)

L'arête (b, e) est à son tour écartée car elle porte le degré de b et e à 3.

Idem pour (b, d)

L'arête suivante(c, d) est acceptée.

On a maintenant un chemin a->b->c->d->e->f

L'arête (a, f) est acceptée et ferme l'itinéraire.

Les graphes

Graphes : Représentation mémoire

1. Matrice

Si V = {v1, v2, ..., vn} est l'ensemble des sommets, le graphe est représenté par une matrice A[n, n] de booléens tels que A[i, j] est vrai s'il existe un arc de i vers j.
Si le graphe est étiqueté, A[i, j] représentera la valeur.

2. Tableaux de listes linéaires chaînées

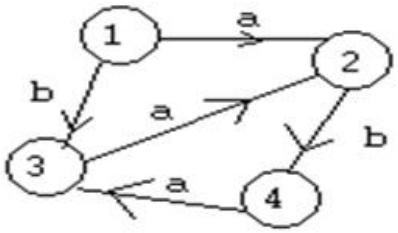
Le graphe est représenté par un tableau Tete[1..n] où Tete[i] est un pointeur vers la liste des sommets adjacents à i.

3. Liste de listes linéaires chaînées

Le graphe est représenté par une liste de nœuds. Chaque nœud pointe vers la liste des sommets qui lui sont adjacents.

Les graphes

Graphes : Représentation mémoire



Graphe étiqueté

	1	2	3	4
1		a	b	
2				b
3		a		
4			a	

Matrice

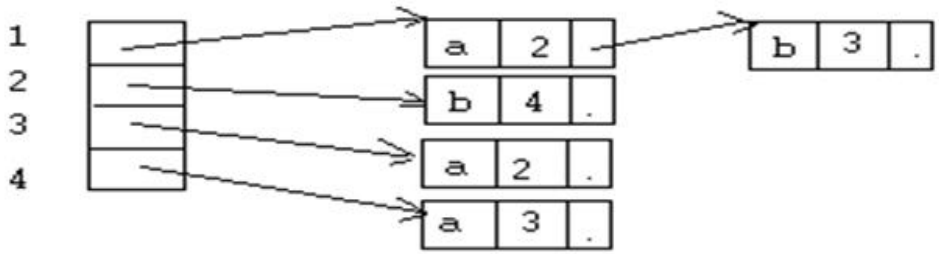
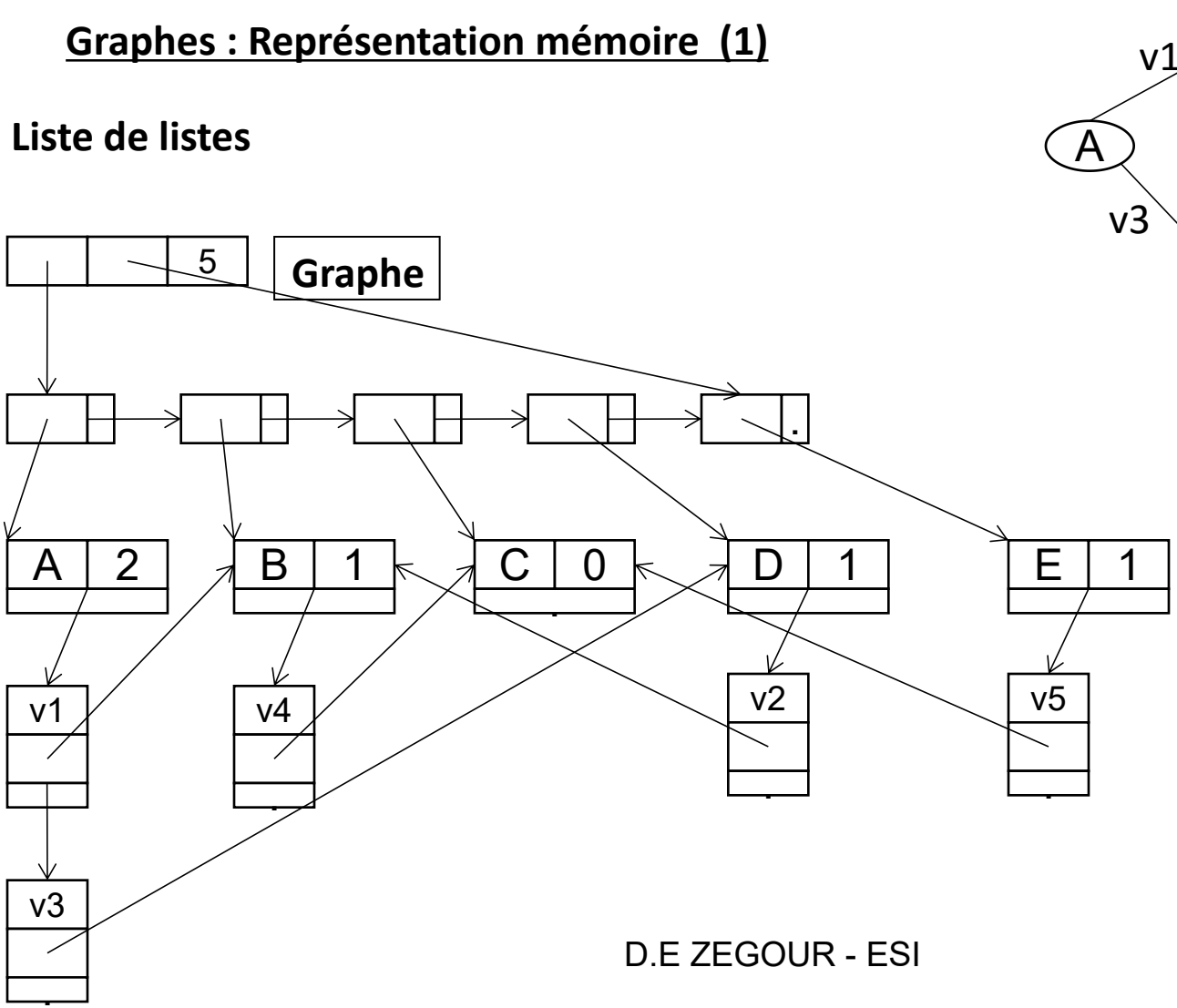


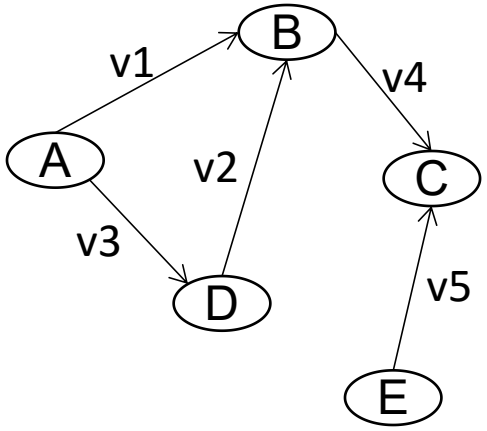
Tableau de listes

Les graphes

Graphes : Représentation mémoire (1)

Liste de listes





Les graphes

Implémentation des graphes en C (DYNAMIQUE / Listes de listes)

```
/* Graphes */
  struct Typegraphe
    {
                struct  Typeliste *Prem;
                struct  Typeliste *Dernier ;
                int Nbr;
    }
```

```
/* Listes */
   struct Typeliste
    {
                struct Typenoeud *Element;
                struct Typeliste *Suivant;
    }
```

Représentation mémoire (1)

```
struct Typelistearc
  {
                int Valarc ;    /* Valeur rattachée à l'arc */
                struct Typenoeud *Noeud;
                struct Typelistearc *Suivant ;
  }
```

```
/* Noeud */
   struct  Typenoeud
    {
                int Valnoeud ; /* Valeur rattach,e au noeud */
                int Degre ;     /* Degré du noeud */
                struct Typelistearc *List;  /* Liste des noeuds adjacent contenant les valeurs des arcs */
    }
```