Estructures de dades i algorismes

Tema 6
Grafs

Curs 2018-2019

Objectius

- Aprendre els conceptes bàsics del grafs
- Aprendre a representar grafs mitjançant EDAs
- Aprendre els recorreguts bàsics sobre grafs
- Aprendre a obtindre el camí de pes mínim en un graf ponderat.
- Aprendre a obtindre l'arbre de recobriment mínim cerca

Continguts

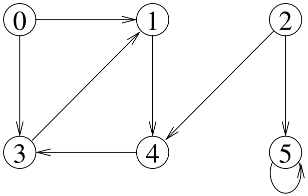
Què és un graf?

- Una ferramenta matemàtica per a modelar relacions binàries entre elements d'un conjunt.
- Permeten expressar d'una forma visualment molt senzilla i efectiva les relacions que es donen entre elements de molt diversa índole.
- Des d'un punt de vista pràctic, els grafs permeten estudiar les interrelacions entre unitats que interactuen unes amb unes altres.
- El problema dels ponts de Königsberg, és un conegut problema matemàtic resolt per Leonhard Euler en 1736, la resolució del qual va donar origen a la teoria de grafs.

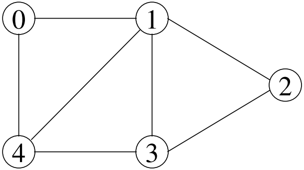
2. GRAFS: APLICACIONS

- Indexació de pàgines web. Cada pag es un vèrtex i els links són les arestes
- Xarxes socials.
- Internet
- Testeig de models: chips, codi, autòmates, etc.

- **Graf dirigit:** és un parell $G = (V, A)$ on V és un conjunt finit d'elements anomenats *vèrtexs* i $A \subseteq V \times V$ és un conjunt de "parells ordenats" de vèrtexs anomenats *arestes*.
  Si (u, v) és una aresta de G, es diu que el vèrtex v és *adjacent* a u.



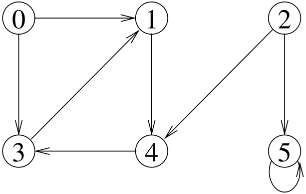
- **Graf no dirigit:** és un parell $G = (V, A)$ on V és un conjunt finit de *vèrtexs* y $A \subseteq \{\{u, v\} \mid u, v \in V \wedge v \neq u\}$ és un conjunt de "parells no ordenats" de vèrtexs.
  Si $a = \{u, v\}$ és una aresta no dirigida, es diu que a *uneix* a u i v i que a *incideix* en u y v.
  Si $\{u, v\}$ és una aresta de G, es diu que el vèrtex v és *adjacent* a u. La relació és simètrica.



3. GRAFS: DEFINICIONS BÀSIQUES

- **Grau:** per a tot vèrtex v,
  - **grau d'entrada** és el nombre d'arestes que incideixen en v;
  - **grau d'eixida** és el nombre d'arestes que emergeixen de v;
  - **grau** és la suma dels graus d'entrada i eixida de v.

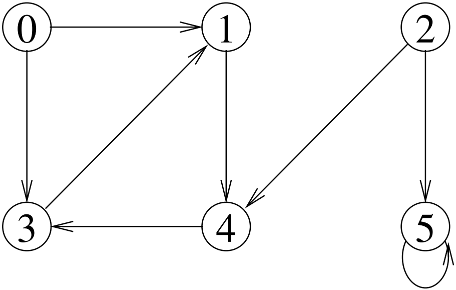
  El **grau d'un graf** és el màxim grau dels seus vèrtexs.



Exemple.: El grau d'entrada del vèrtex 1 és 2; el grau d'eixida és 1; el grau del vèrtex és 3. El grau del graf és 3.

3. GRAFS: DEFINICIONS BÀSIQUES

- **Camí** des d'un vèrtex $o in V$ a un vèrtex $v \in V$: és una seqüència $\langle v_0, v_1, \ldots, v_k \rangle$ de vèrtexs de $G = (V, A)$ tal que $v_0 = o$, $v_k = v$, $(v_i, v_{i+1}) \in A$, $0 \leq i < k$
- La **longitud d'un camí** $\langle v_0, v_1, \ldots, v_k \rangle$ és el nombre d'arestes que ho formen.
- **Camí simple:** és un camí $\langle v_0, v_1, \ldots, v_k \rangle$ en el qual tots els seus vèrtexs són diferents, excepte potser el primer i l'últim.
- **Cicle:** és un camí simple $\langle v_0, v_1, \ldots, v_k \rangle$ tal que $v_0 = v_k$ i el camí conté almenys una aresta.
- Un **bucle** és un cicle de longitud 1.
- **Graf acíclic:** és un graf sense cicles.

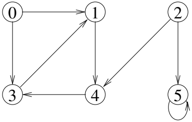


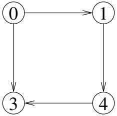
Exemple.: El camí $\langle 0, 3, 1, 4 \rangle$ és simple i té longitud 3. El camí $\langle 0, 1, 4, 3, 1 \rangle$ no és simple.

Exemple.: El camí $\langle 1, 4, 3, 1 \rangle$ és un cicle de longitud 3. El cicle $\langle 5, 5 \rangle$ és un bucle.

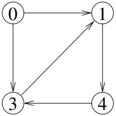
3. GRAFS: DEFINICIONS BÀSIQUES

- **Subgraf:** $G' = (V', A')$ es un subgraf de $G = (V, A)$ si $V' \subseteq V \wedge A' \subseteq A$.
- **Subgraf induït:** Donat $V' \subseteq V$, el subgraf de G **induït** per V' es $G' = (V', A')$ tal que $A' = \{(u, v) \in A \mid u, v \in V'\}$.





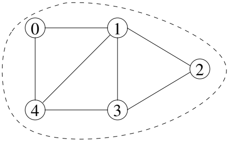
Subgraf

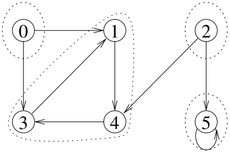


Subgraf induït per $V' = \{0, 1, 3, 4\}$

3. GRAFS: DEFINICIONS BÀSIQUES

- **Vèrtex assolible des d'un vèrtex** u**:** és qualsevol vèrtex v per al qual existeix un camí de u a v.
- Les **components connexes en un graf no dirigit** són les classes d'equivalència de vèrtexs segons la relació *"ser assolible"*.
  - Un graf no dirigit és **connex** si $\forall u, v \in V$, v és assolible des de u. És a dir, si té una única component connexa.
- Les **components fortament connexes en un graf dirigit** són les classes d'equivalència de vèrtexs segons la relació *"ser mútuament assolible"*.
  - Un graf dirigit és **fortament connex** si $\forall o, v \in V$, v és assolible des de o.





3. GRAFS: DEFINICIONS BÀSIQUES

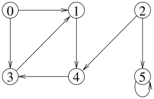
- **Graf complet:** és un graf $G = (V, A)$ en el qual $\forall u, v \in V,\ u \neq v$, $(u, v) \in A$.
- **Graf etiquetat:** és un graf $G = (V, A)$ acompanyat d'una funció $f : A \rightarrow E$, on E és un conjunt les components del qual es denominen *etiquetes*.
- **Graf ponderat:** és un graf etiquetat amb nombres reals ($E \equiv real$).
- Un graf es considera **dens** si $|A| \approx |V|^2$.
- Un graf es considera **dispers** si $|A| <<< |V|^2$.

4. REPRESENTACIÓ DE GRAFS: Llista i Matriu d'adjacència

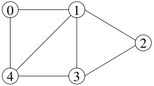
- Existeixen dues formes fonamentals de representar un graf:
  - Si el graf és molt *dispers* ($|A| <<< |V|^2$): Llistes de adjacència
  - Si el graf és molt *dens* ($|A| \approx |V|^2$): Matriu de d'Adjacències
- Si el graf és *ponderat*:
  - llista (node amb pes)
  - matriu (matriu amb pesos)
- Saber si una aresta pertany al graf:
  - llista (recórrer tota la llista $O(grau(G))$
  - matriu $O(1)$

4.1. REPRESENTACIÓ DE GRAFS: Matriu d'adjacència

- Un graf $G = (V, A)$ es representa com a G: matriu$[V, V]$ de booleans.
- La component $G[u, v]$ és 1 si $(u, v) \in A$; sinó $G[u, v] = 0$.
- Memòria: $O(|V|^2) \rightarrow$ grafs densos $|A| \approx |V|^2$.
- Temps d'accés: $O(1)$. Llista adjacents: $O(|V|)$. Llista incidents: $O(|V|)$.



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



0	0	1	0	0	1
1	1	0	1	1	1
2	0	1	0	1	0
3	0	1	1	0	1
4	1	1	0	1	0

4.1. REPRESENTACIÓ DE GRAFS: Matriu d'adjacència

```
class GrafoM {
    private boolean m[][];
    private int nvert;
    GrafoM(int nvert) {
        m=new boolean[nvert][nvert];
        this.nvert=nvert;
        for(int i=0;i<nvert;i++)
      for(int j=0;j<nvert;j++)
          m[i][j]=false;
    }
    void inserta_arista(int u,int v) {m[u][v]=true;}
    void elimina_arista(int u,int v) {m[u][v]=false;}
    public String toString() {
        String s="";
        for(int i=0;i<nvert;i++)
            for(int j=0;j<nvert;j++)
          if (m[i][j]) s=s+i+"-->"+j+"\n";
        return s;
    }
}
```

4.1. REPRESENTACIÓ DE GRAFS: Matriu d'adjacència

- Exercici 1: Implementar el càlcul del grau d'entrada

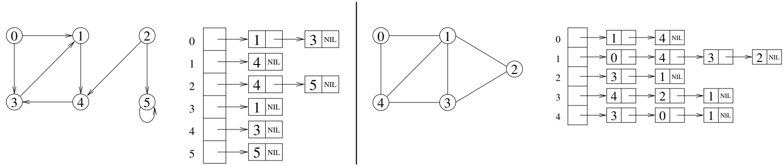
```
class GrafoM {
    private boolean m[][];
    private int nvert;
    //...
    GrafoM(int nvert) {

    int grado_entrada() {
     int s,max;
     max=0;
     for(int i=0;i<nvert;i++) {
         s=0;
         for(int j=0;j<nvert;j++)
          if (m[j][i]) s++;
         if (s>max) {max=s;}
     }
     return max;
    }
}
```

- Exercici 2: Implementar el graf traspost

```
class GrafoM {
    private boolean m[][];
    private int nvert;
    //...

    GrafoM traspuesto() {
        GrafoM g=new GrafoM(nvert);

        for(int i=0;i<nvert;i++)
            for(int j=0;j<nvert;j++)
             if (m[i][j]) g.inserta_arista(j,i);
        return g;
    }
}
```

4.2. REPRESENTACIÓ DE GRAFS: Llistes d'adjacència

- Un graf $G = (V, A)$ es representa com un *vector de llistes de vèrtexs* indexat per vèrtexs; és a dir, G: vector$[V]$ de V.
- Cada component $G[v]$ és una llista dels vèrtexs emergents i/o incidents de/a $v \in V$.
- Memòria: $O(|V| + |A|) \rightarrow$ grafs dispersos $|A| <<< |V|^2$.
- Temps d'accés: $O(grau(G))$. Llista adjacents: $O(grau(G))$. (Llista incidents: $O(grau(G))$ amb llistes d'incidència.)



4.2. REPRESENTACIÓ DE GRAFS: Llistes d'adjacència

```java
public class Adyacente {
    protected int destino;
    protected double peso;

    public Adyacente(int v, double peso){
       destino = v;
       this.peso = peso;
    }
    public int getDestino(){
       return this.destino;
    }
    public double getPeso(){
       return this.peso;
    }
    public String toString() { return destino + "("+ peso+ ") "; }
}
```

4.2. REPRESENTACIÓ DE GRAFS: Llistes d'adjacència

```
public class GrafoDirigido extends Grafo {
    protected int numV, numA;
    protected ListaConPI<Adyacente> elArray[];

    public GrafoDirigido(int numVertices){
        numV = numVertices; numA=0;
        elArray = new ListaConPI[numVertices];
        for ( int i=0; i<numV; i++ )
           elArray[i]= new LEGListaConPI<Adyacente>();
    }
    public int numVertices(){ return numV; }
    public int numAristas(){ return numA; }

    // Comprueba si la arista (i,j) esta en un grafo
    public boolean existeArista(int i, int j){
       ListaConPI<Adyacente> l = elArray[i];
       for ( l.inicio(); !l.esFin(); l.siguiente() )
            if ( l.recuperar().getDestino()==j ) return true;
       return false;
    }
```

4.2. REPRESENTACIÓ DE GRAFS: Llistes d'adjacència

```
// Devuelve el peso de la arista (i,j) de un grafo,
// 0 si dicha arista no esta en el grafo.
public double pesoArista(int i, int j){
    ListaConPI<Adyacente> l = elArray[i];
    for ( l.inicio(); !l.esFin(); l.siguiente() )
        if ( l.recuperar().getDestino()==j )
          return l.recuperar().getPeso();
   return 0.0;
}

// Si no esta, inserta la arista (i, j) en un grafo
// no Ponderado
public void insertarArista(int i, int j){
    if ( !existeArista(i, j) ) {
       elArray[i].insertar(new Adyacente(j, 1));
       numA++;
   }
}
```

4.2. REPRESENTACIÓ DE GRAFS: Llistes d'adjacència

```
    //...
    // Si no esta, inserta la arista (i, j) de peso p
    // en un grafo Ponderado
    public void insertarArista(int i, int j, double p){
         if ( !existeArista(i, j) ) {
            elArray[i].insertar(new Adyacente(j, p));
            numA++;
        }
    }

    // Devuelve una Lista Con PI que contiene
    // los adyacentes al vertice i de un grafo.
    public ListaConPI<Adyacente> adyacentesDe(int i){
        return elArray[i];
    }
}
```

4.2. REPRESENTACIÓ DE GRAFS: Llistes d'adjacència

```java
public class GrafoNoDirigido extends GrafoDirigido {
    public GrafoNoDirigido(int numVertices){ super(numVertices); }
    // No Dirigido y No Ponderado;
    public void insertarArista(int i, int j){
        if ( !existeArista(i, j) ){
            elArray[i].insertar(new Adyacente(j, 1));
            elArray[j].insertar(new Adyacente(i, 1));
            numA++;
        }
    }
    // Si no esta, inserta la arista (i, j) de peso p en un grafo
    public void insertarArista(int i, int j, int p){
        if ( !existeArista(i,j) ) {
            elArray[i].insertar(new Adyacente(j, p));
            elArray[j].insertar(new Adyacente(i, p));
            numA++;
        }
    }
}
```

4.2. REPRESENTACIÓ DE GRAFS: Llistes d'adjacència

- Exercici 1: Implementar el càlcul del grau d'entrada d'un vèrtex en un graf no
- Exercici 2: Implementar el graf traslladat en un graf no dirigit
- Exercici 3: Implementar com a mètode estàtic la intersecció de dos grafs no dirigits. La intersecció de dos grafs és un altre graf amb només aquelles arestes que estan en tots dos grafs.

4.3 Resum: Representació de grafs

	Espai
Matriu d'adjacència	$\Theta(\|V\|^2)$
Llistes d'adjacència	$\Theta(\|V\| + \|A\|)$

	Construcció del graf	$(u, v) \in A$
Matriu d'adjacència	$\Theta(\|V\|^2)$	$\Theta(1)$
Llistes d'adjacència	$\Theta(\|V\| + \|A\|)$	$\Theta(grau\_eixida(u))$

	Recorregut successors	Recorregut predecessors
Matriu d'adjacència	$\Theta(\|V\|)$	$\Theta(\|V\|)$
Llistes d'adjacència	$\Theta(grau\_eixida(u))$	$\Theta(\|A\|)$
Llistes d'adjacència	$\Theta(\|A\|)$	$\Theta(grau\_entrada(u))$

5. RECORREGUTS DE GRAFS

- Mètode per a recórrer de forma sistemàtica i eficient un graf.
  - Recorregut en profunditat: generalització del recorregut en preordre d'un arbre
  - Recorregut en amplitud: generalització del recorregut en nivells d'un arbre
- En tots els algorismes de recorregut de grafs suposarem que el graf està implementat **amb llistes de adjacència**.

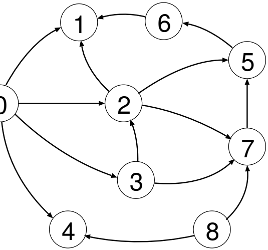
Recorregut en profunditat d'un graf

- Donat un graf $G = (V, A)$ i un vèrtex $v \in V$, l'estratègia de recorregut en profunditat (*Depth-First Search (DFS)*), explora sistemàticament les arestes de G de manera que primer es visiten els vèrtexs adjacents als visitats més recentment. D'aquesta forma, es va *aprofundint* en el graf; és a dir, allunyant-se progressivament de v.
- Aquesta estratègia admet una implementació simple de forma recursiva, utilitzant globalment un *comptador* `n` i un *vector de naturals* `visita` per a *marcar* els vèrtexs ja visitats `visita[v] > 0` i emmagatzemar l'ordre de recorregut.

Recorregut en profunditat d'un graf

```
public int[] toArrayDFS() {
    int res[] = new int[numVertices()];
    visitats = new int[numVertices()];
    ordreVisita = 0;
    for ( int i=0; i<numVertices(); i++ )
        if ( visitats[i]==0 ) res = toArrayDFS(i, res);
    return res;
}
// Recorregut DFS del vertex origen d'un graf
protected int[] toArrayDFS(int origen, int[] res){
    res[ordreVisita++] = origen;
    visitats[origen] = 1;
    ListaConPI<Adjacent> l = adjacenttesDe(origen);
    for ( l.inicio(); !l.esFin(); l.siguiente() ){
       int destino = l.recuperar().getDestino();
       if ( visitats[destino]==0 ) res = toArrayDFS(destino, res);
    }
    return res;
}
```

Recorregut en profunditat d'un graf: exemple



nodos	*visita*								
v/*w*	0	1	2	3	4	5	6	7	8
	0	0	0	0	0	0	0	0	0
0/1,2,3,4	1	-	-	-	-	-	-	-	-
1/–	-	2	-	-	-	-	-	-	-
2/1,5,7	-	-	3	-	-	-	-	-	-
5/6	-	-	-	-	-	4	-	-	-
6/1	-	-	-	-	-	-	5	-	-
7/5	-	-	-	-	-	-	-	6	-
3/2,7	-	-	-	7	-	-	-	-	-
4/–	-	-	-	-	8	-	-	-	-
8/4,7	-	-	-	-	-	-	-	-	9
visita	1	2	3	7	8	4	5	6	9

Ordre de Visita de Nodes: 0, 1, 2, 5, 6, 7, 3, 4, 8

Recorregut en profunditat d'un graf: exemple

- Exercici 1: Analitza el cost temporal del recorregut en profunditat DFS.
- Exercici 2: Es demana implementar el mètode **assolibles** que calcule el nombre de vèrtexs que són assolibles (amb algun camí de longitud major o igual a 0) des del vèrtex que es passa com a argument.

Recorregut en amplària d'un graf

- Donat un graf $G = (V, A)$ i un vèrtex $v \in V$, l'estratègia de recorregut en amplitud o amplària (*Breadth-First Search (BFS)*), explora sistemàticament les arestes de G de manera que primer es visiten els vèrtexs *més propers* a v.
- La funció `recorregut_ampl‘aria` obté un recorregut en amplària de G cridant a la rutina `bfs`. S'utilitza globalment un *comptador* `n` i un *vector de naturals* `visita` per a *marcar* els vèrtexs ja visitats `visita[v] > 0` i emmagatzemar l'ordre de recorregut. `bfs` utilitza una cua auxiliar `Q` per a gestionar els vèrtexs no visitats.

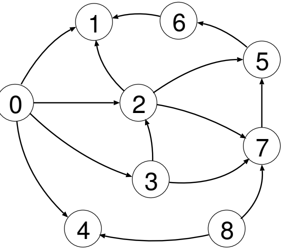
Recorregut en amplària d'un graf

```
public int[] toArrayBFS(){
    int res[] = new int[numVertices()];
    visitats = new int[numVertices()];
    ordreVisita = 0;
    q = new ArrayCola<Integer>();
    for ( int i=0; i<numVertices(); i++ )
        if ( visitats[i]==0 ) res = toArrayBFS(i, res);
    return res;
}
// ...
```

Recorregut en amplària d'un graf

```java
// Recorregut BFS del vertex origen d'un graf
protected int[] toArrayBFS(int origen, int[] res) {
    res[ordreVisita++] = origen;
    visitats[origen] = 1;
    q.encolar(new Integer(origen));
    while ( !q.esVacia() ){
        int u = q.desencolar().intValue();
        ListaConPI<Adjacent> l = adjacenttesDe(u);
        for ( l.inicio(); !l.esFin(); l.siguiente() ){
            int v = l.recuperar().getDestino();
            if ( visitats[v]==0 ){
          res[ordreVisita++] = v;
          visitats[v] = 1;
          q.encolar(new Integer(v));
            }
        }
    }
    return res;
}
```

Recorregut en amplària d'un graf: exemple



v	u	w	0	1	2	3	4	5	6	7	8	Q
0			1	0	0	0	0	0	0	0	0	< 0 >
	1		-	-	-	-	-	-	-	-	-	<>
		1	-	2	-	-	-	-	-	-	-	< 1 >
		2	-	-	3	-	-	-	-	-	-	< 1, 2 >
		3	-	-	-	4	-	-	-	-	-	< 1, 2, 3 >
		4	-	-	-	-	5	-	-	-	-	< 1, 2, 3, 4 >
	1		-	-	-	-	-	-	-	-	-	< 2, 3, 4 >
	2		-	-	-	-	-	-	-	-	-	< 3, 4 >
		1	-	-	-	-	-	-	-	-	-	—
		5	-	-	-	-	-	6	-	-	-	< 3, 4, 5 >
		7	-	-	-	-	-	-	-	7	-	< 3, 4, 5, 7 >
	3		-	-	-	-	-	-	-	-	-	< 4, 5, 7 >
		2	-	-	-	-	-	-	-	-	-	—
		7	-	-	-	-	-	-	-	-	-	—
	4		-	-	-	-	-	-	-	-	-	< 5, 7 >
	5		-	-	-	-	-	-	-	-	-	< 7 >
		6	-	-	-	-	-	-	8	-	-	< 7, 6 >
	7		-	-	-	-	-	-	-	-	-	< 6 >
		5	-	-	-	-	-	-	-	-	-	—
	6		-	-	-	-	-	-	-	-	-	<>
		1	-	-	-	-	-	-	-	-	-	<>
8			-	-	-	-	-	-	-	-	9	< 8 >
	8		-	-	-	-	-	-	-	-	-	<>
		4	-	-	-	-	-	-	-	-	-	—
		7	-	-	-	-	-	-	-	-	-	—
			1	2	3	4	5	6	8	7	9	

Ordre de Visita de Nodes: 0, 1, 2, 3, 4, 5, 7, 6, 8

6. CAMINS DE MÍNIM PES EN GRAFS DIRIGITS

Donat un graf $G = (V, A)$ dirigit i ponderat per una funció $p : A \rightarrow real^{\geq 0}$, es defineix el *pes* d'un camí $v_0, v_1, \ldots, v_k$ com la suma dels pesos de les seues arestes:

$$p(v_0, v_1, \ldots, v_k) = \sum_{i=1}^{k} p(v_{i-1}, v_i)$$

6. CAMINS DE MÍNIM PES
Consideracions

- Si el graf és no dirigit, podem obtenir un graf dirigit sense més que duplicar cada aresta $\{o, v\}$ en cada adreça: (o, v) i (v, o) ambdues amb el mateix pes. Per tant, reduïm el problema al cas de grafs dirigits.
- Si no hi ha una aresta entre dos vèrtexs, podem assumir que és "equivalent" al fet que existisca una aresta amb pes infinit.
- És possible que no existisca el camí de menor cost entre dos vèrtexs s i t si podem arribar de s a t per un camí que incloga un cicle de pes negatiu. En tal cas, donant suficients voltes al cicle podem obtenir camins de pes arbitràriament baix.

Algorismes

- Si tots els arcs tenen el mateix pes i aquest és positiu, l'algorisme de cerca primer en amplària des de s ens proporciona els costos de s a tots els altres vèrtexs.
- En el cas de grafs acíclics, podem utilitzar tècniques de programació dinàmica
- En el cas de grafs amb cicles i pesos positius, podem utilitzar l'algorisme de Dijkstra, que veurem a continuació.

6.1. ALGORISME DE DIJKSTRA

Requereix pesos positius. Aconsegueix processar els vèrtexs i arestes una sola vegada. Utilitza un vector de cotes superiors de la distància des de s. A més es garantir un ordre de selecció de vèrtexs de manera que cada vèrtex seleccionat té en D la vertadera distància, no solament una cota.

És a dir, aquest algorisme manté un conjunt de vèrtexs S el pes de la qual del camí més curt des de l'origen s ja és conegut. L'algorisme va seleccionant el vèrtex $o \in V - S$ amb la millor estimació del camí mínim, ho insereix en S i utilitza les arestes que ixen de o per a actualitzar la cota dels vèrtexs de $V - S$.

Idea voraç: començant en el vèrtex origen s, construir incrementalment camins als altres vèrtexs seleccionant en cada pas un vèrtex v no seleccionat anteriorment tal que:

- Existisca algun vèrtex $o \in V$ ja seleccionat prèviament tal que $(o, v) \in A$.
- En afegir (o, v) al camí que acabava en o es produïsca el menor increment de pes possible.

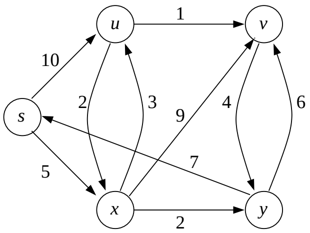
6.1. ALGORISME DE DIJKSTRA

```
protected void dijkstra(int origen){
    int D[]=new int[numVertices()];  // Distancies
    boolean F[]=new boolean[numVertices()]; // Fixats
    int u,v,min,fijados;

    for (u=0; u<numVertices(); u++) {
        D[u]=INFINITO;;
        F[u]=false;
    }

    D[origen]=0;
    fijados=0;
    //...
```

6.1. ALGORISME DE DIJKSTRA

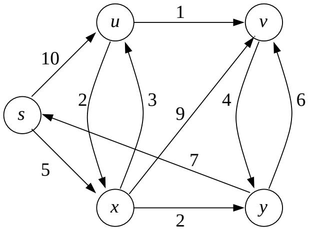
```
    while (fijados<numVertices()) {
        min = INFINITO; u=0;
        for (v=0; v<numVertices(); v++)
            if (!F[v] && D[v] < min) {
          min=D[v];
          u=v;
            }
        F[u]=true;
        fijados++;
        ListaConPI<Adjacent> l = adjacenttesDe(u);
        for ( l.inicio(); !l.esFin(); l.siguiente() ){
            Adjacent aU = l.recuperar();
            int   w = aU.getDestino();
            double pesoUW = aU.getPeso();
            if (!F[w] && D[w] > D[u] + pesoUW)
          D[w] = D[u] + a.peso;
        }//for
    }// while
}
```

6.2. ALGORISME DE DIJKSTRA: UN EXEMPLE



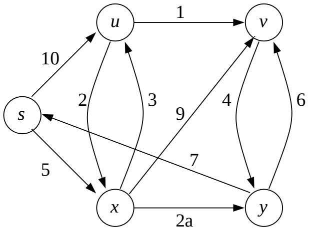
Iteració	u	Fixats	Per fixar	$D[s]$	$D[u]$	$D[v]$
0	-	$\emptyset$	$\{s, u, v, x, y\}$	0	∞	∞

6.2. ALGORISME DE DIJKSTRA: UN EXEMPLE



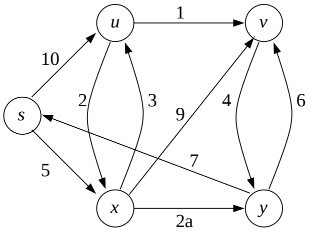
Iteració	u	Fixats	Per fixar	$D[s]$	$D[u]$	$D[v]$
0	-	$\emptyset$	$\{s, u, v, x, y\}$	0	∞	∞
1	s	$\{s\}$	$\{u, v, x, y\}$	0	10	∞

6.2. ALGORISME DE DIJKSTRA: UN EXEMPLE



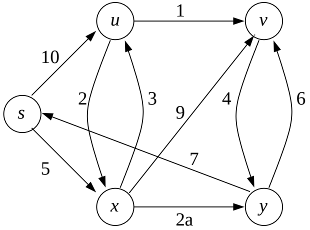
Iteració	u	Fixats	Per fixar	$D[s]$	$D[u]$	$D[v]$
0	-	$\emptyset$	$\{s,u,v,x,y\}$	0	∞	∞
1	s	$\{s\}$	$\{u,v,x,y\}$	0	10	∞
2	x	$\{s,x\}$	$\{u,v,y\}$	0	8	14

6.2. ALGORISME DE DIJKSTRA: UN EXEMPLE



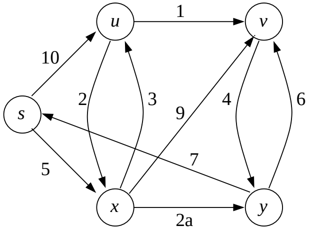
Iteració	u	Fixats	Per fixar	$D[s]$	$D[u]$	$D[v]$
0	-	$\emptyset$	$\{s, u, v, x, y\}$	0	∞	∞
1	s	$\{s\}$	$\{u, v, x, y\}$	0	10	∞
2	x	$\{s, x\}$	$\{u, v, y\}$	0	8	14
3	y	$\{s, x, y\}$	$\{u, v\}$	0	8	13

6.2. ALGORISME DE DIJKSTRA: UN EXEMPLE



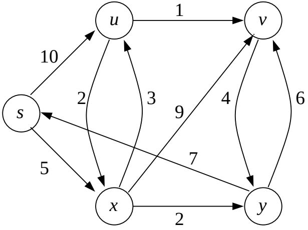
Iteració	u	Fixats	Per fixar	$D[s]$	$D[u]$	$D[v]$
0	-	$\emptyset$	$\{s, u, v, x, y\}$	0	∞	∞
1	s	$\{s\}$	$\{u, v, x, y\}$	0	10	∞
2	x	$\{s, x\}$	$\{u, v, y\}$	0	8	14
3	y	$\{s, x, y\}$	$\{u, v\}$	0	8	13
4	u	$\{s, x, y, u\}$	$\{v\}$	0	8	9

6.2. ALGORISME DE DIJKSTRA: UN EXEMPLE



Iteració	u	Fixats	Per fixar	$D[s]$	$D[u]$	$D[v]$
0	-	$\emptyset$	$\{s, u, v, x, y\}$	0	∞	∞
1	s	$\{s\}$	$\{u, v, x, y\}$	0	10	∞
2	x	$\{s, x\}$	$\{u, v, y\}$	0	8	14
3	y	$\{s, x, y\}$	$\{u, v\}$	0	8	13
4	u	$\{s, x, y, u\}$	$\{v\}$	0	8	9
5	v	$\{s, x, y, u, v\}$	$\emptyset$	0	8	9

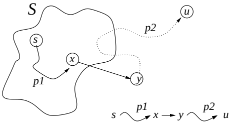
6.2. ALGORISME DE DIJKSTRA: UN EXEMPLE



Iteració	u	Fixats	Per fixar	$D[s]$	$D[u]$	$D[v]$
0	-	$\emptyset$	$\{s, u, v, x, y\}$	0	∞	∞
1	s	$\{s\}$	$\{u, v, x, y\}$	0	10	∞
2	x	$\{s, x\}$	$\{u, v, y\}$	0	8	14
3	y	$\{s, x, y\}$	$\{u, v\}$	0	8	13
4	u	$\{s, x, y, u\}$	$\{v\}$	0	8	9
5	v	$\{s, x, y, u, v\}$	$\emptyset$	0	8	9

6.3. ALGORISME DE DIJKSTRA:
Correcció de l'algorisme de Dijkstra

És immediat veure que $D[v] \geq d(s, v)$ en tot moment. N'hi ha prou amb veure que s'aconsegueix la igualtat en cada vèrtex que és "fixat". El cas inicial s és senzill. Per a la resta de casos, siga u el vèrtex a fixar. Per reducció a l'absurd sobre el criteri d'elecció del vèrtex, es demostra que el camí de s a u utilitza únicament vèrtexs fixats. És a dir, no pot haver-hi cap camí més curt fins a o que passada per vèrtexs (exemple i) que no estan fixats, doncs en tal cas s'elegiria i abans que u:



Aquesta demostració requereix que tots els pesos de les arestes siguen positius. Per tant, $D[u]$ és la distància del camí més curt des de l'origen s al vèrtex u.

6.5. ALGORISME DE DIJKSTRA: COST

G implementat com una llista de adjacència

- Existeixen $|V|$ operacions d'extracció del mínim en el vector, amb un cost $O(|V|)$.
- Cada vèrtex $v \in V$ es fixa exactament una vegada, de manera que cada aresta en la llista de adjacència s'examina una única vegada. A causa que el nombre total d'arestes en G és $|A|$, existeixen $|A|$ iteracions d'aquest bucle, i cada iteració té un cost $O(1)$.

Per tant, el cost total de l'algorisme és $O(|V|^2 + |A|) = O(|V|^2)$.

6.6. ALGORISME DE DIJKSTRA AMB UN HEAP: COST

G implementat com una llista de adjacència i utilitzant un *minheap*

(*algorisme de Dijkstra modificat*):

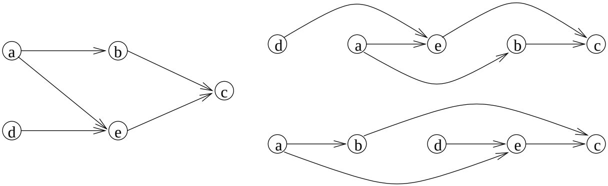
- Existeixen $|V|$ operacions d'extracció del mínim en un *heap*, amb un cost $O(\log|V|)$. Cal afegir el temps de construir el *heap*, $O(|V|)$.
- En cada iteració del bucle que recorre les arestes, si $D[v] > D[o] + pes(o, v)$, s'haurà de modificar $D[v]$ i, per tant, modificar el *heap*, amb un cost $O(\log|V|)$. El nombre de vegades que s'executa el bucle és $O(|A|)$.

Per tant, el temps total de l'algorisme serà $O((|V| + |A|)\log|V|) = O(|A|\log|V|)$.

7. ORDRE TOPOLÒGIC EN GRAFS ACÍCLICS

Un **ordre topològic** d'un graf dirigit acíclic $G = (V, A)$ és una ordenació dels vèrtexs de manera que si $(o, v) \in A$, llavors o apareix *abans* que v. (La solució no és única.) Exem..: prerequisits dels estudis.

No és possible l'ordenació topològica quan existeixen cicles!



Ordre no únic.
Ordenació de vèrtexs en eix horitzontal amb les arestes d'esquerra a dreta.

7. ORDRE TOPOLÒGIC EN GRAFS ACÍCLICS

Donat un graf acíclic $G = (V, A)$, el *recorregut en profunditat* pot usar-se directament per a ordenar els vèrtexs de V segons un ordre (parcial) $\prec$ tal que, $\forall o, v \in V$, si $(o, v) \in A$ llavors $o \prec v$.

- N'hi ha prou amb anar imprimint els *vèrtexs completament explorats* per DFS
- S'obté un ordre *invertit*.
- Cost temporal: $O(|V| + |A|)$

7. ORDRE TOPOLÒGIC EN GRAFS ACÍCLICS

```
public int[] toArrayTopologico() {
    visitats = new boolean[numVertices()];
    Pila<Integer> pVRecorreguts = new ArrayPila<Integer>();

    // Recorregut dels vertex
    for (int vOrigen = 0; vOrigen < numVertices(); vOrigen++)
        if (!visitat[vOrigen])
            ordenacioTopologica(vOrigen, pVRecorreguts);

    // Copia resultat a un array
    int res[] = new int[numVertices()];
    for (int i = 0; i < numVertices(); i++)
        res[i] = pVRecorreguts.desapilar();
    return res;
}
```

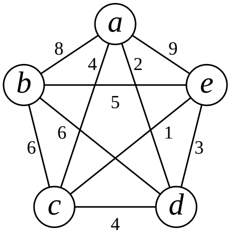
7. ORDRE TOPOLÒGIC EN GRAFS ACÍCLICS

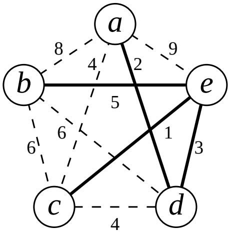
```java
// Funcio recursiva per a ordre topologic:
protected void ordenacioTopologica(int origen,
                  Pila<Integer> pVRecorreguts) {
    visitats[origen]=true;
    // Recorreguem els vertex adjacents
    ListaConPI<Adjacent> aux = adjacenttesDe(origen);
    for (aux.inicio(); !aux.esFin(); aux.siguiente()) {
        int destino = aux.recuperar().destino;
        if (!visitats[destino])
            ordenacioTopologica(destino, pVRecorreguts);
    }
    // Apilem el vertex
    pVRecorreguts.apilar(origen);
}
```

8. ARBRE DE RECOBRIMENT DE COST MÍNIM

Arbre de recobriment d'un graf no dirigit $G = (V, A)$: és un arbre lliure $T = (V', A')$ tal que $V' = V$ i $A' \subseteq A$.

Problema: Donat un graf connex ponderat no dirigit $G = (V, A, p)$, trobar un arbre de recobriment de G, $T = (V, A')$, tal que la suma dels pesos de les $|V| - 1$ arestes de T siga mínim.





8.1. ALGORISME DE KRUSKAL

Idea voraç: construir incrementalment un *bosc* (de recobriment), seleccionan

- *No* es cree cap cicle.
- Produïsca el menor increment de pes possible.

El resultat de l'algorisme és un arbre lliure (no arrelat) format pels mateixos v

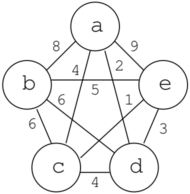
```
public GrafoPonderado* Kruskal (GrafoPonderado *G) { // Pseudo-codi
  // L'arbre de recubrimient te els mateixos vertex que G:
  GrafoPonderado *MST = new GrafoPonderado(G->vertices);
  // Cua  de prioritat, les arestes s'extrauen per menor pes:
  ColaPrioridad Q(G->aristas);
  arista a;
  while (MST->NumAristas() < MST->NumVertices()-1 &&
   Q.extraer(a)) {
    if (a no crea un ciclo en MST)
      MST->insertarArista(a);
  }
  return MST;
}
```

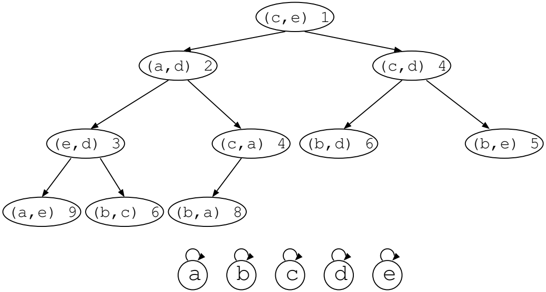
Problema: Com verificar eficientment la condició de *"no crear cicle"*?
Solució: Mantenir una col·lecció de subconjunts (disjunts) amb els vèrtexs de cada arbre del bosc: *Una aresta* (o, v) *no crearà cicle si o i v estan en diferents subconjunts* $\rightarrow$ components connexes $\rightarrow$ estructurar el conjunt d'arestes seleccionades com un mfset de vèrtexs.

Problema: Com seleccionar eficientment l'aresta de menor pes en cada iteració?
Solució: Mantenint les arestes en una cua de prioritat (per exemple, un *minheap*) organitzades segons el seu pes.

8.2. ALGORISME DE KRUSKAL: Un exemple

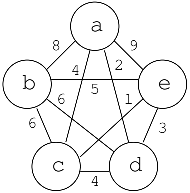
Inicialització

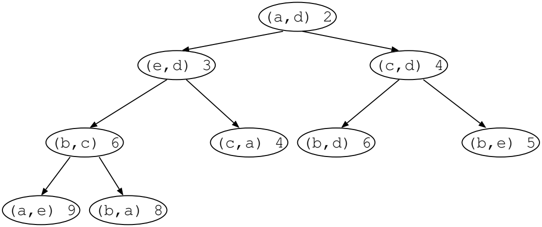


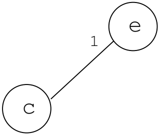


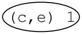
8.2. ALGORISME DE KRUSKAL: Un exemple

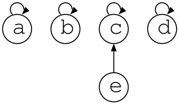
NumAristas = 1





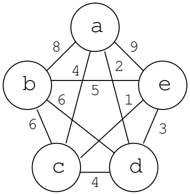


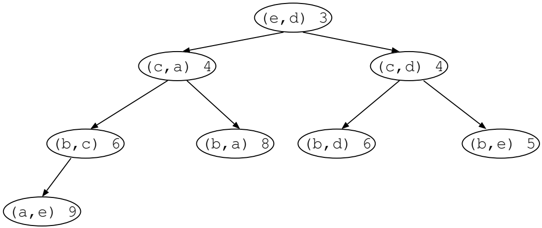


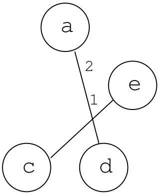


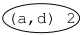
8.2. ALGORISME DE KRUSKAL: Un exemple

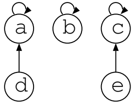
NumAristas = 2





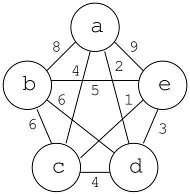


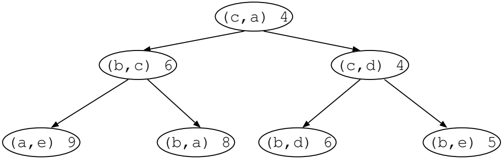


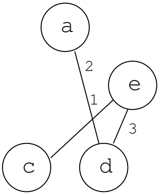


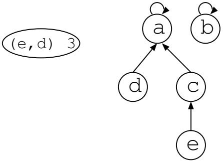
8.2. ALGORISME DE KRUSKAL: Un exemple

NumAristas = 3



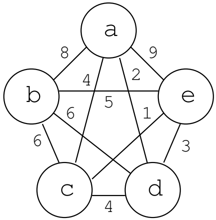


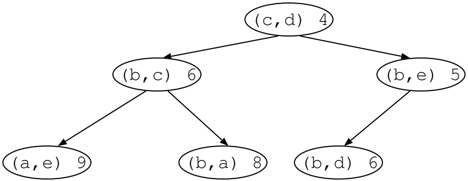


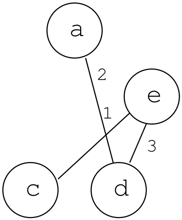


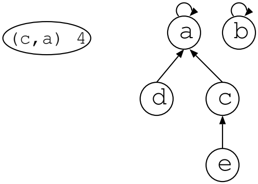
8.2. ALGORITMO DE KRUSKAL: Un exemple

NumAristas = 3



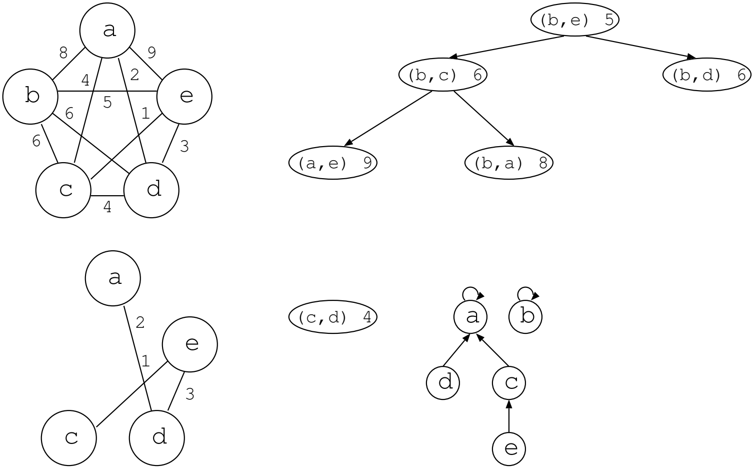






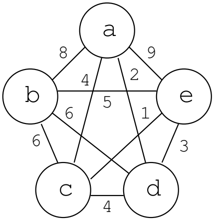
8.2. ALGORISME DE KRUSKAL: Un exemple

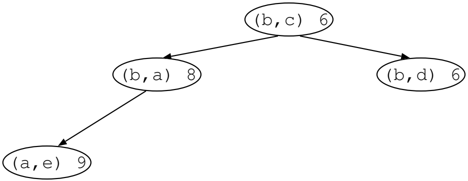
NumAristas = 3

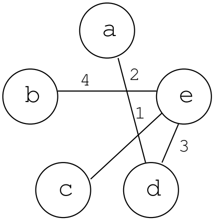


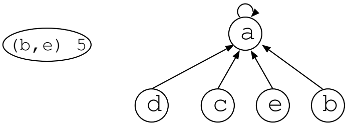
8.2. ALGORISME DE KRUSKAL: Un exemple

NumAristas = 4









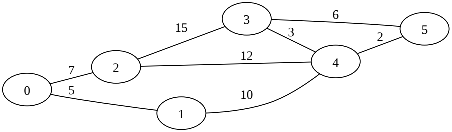
8.2. ALGORISME DE KRUSKAL: Un exemple

```java
public class PesAresta implements Comparable<PesAresta>{
    int origen, desti;
    double pes;

    public PesAresta(int u, int v, double p){
       orige = u;
       desti = v;
       pes = p;
    }

    public int compareTo(PesAresta altre){
       return pes - altre.pes;
    }
}
```

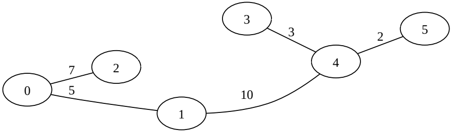
8.2. ALGORISME DE KRUSKAL: Un exemple

```
public GrafoDirigido kruskal() {
    GrafoDirigido res = new GrafoDirigido(numV);
    ColaPrioridad<PesAresta> qPrior = new MinHeap<PesAresta>();
    MFset m = new ForestMFset(numV);
    for (int i = 0; i < numV; i++)
       for (elArray[i].inici(); !elArray[i].esFi();
            elArray[i].seguent()){
           Arista a = elArray[i].recuperar();
           qPrior.inserir(new PesAresta(i, a.desti, a.pes));
       }
    while (res.numA < res.numV - 1 && !qPrior.esBuida()) {
       PesAresta a = qPrior.eliminarMin();
       if (m.find(a.origen ) != m.find(a.desti)) {
           m.merge(a.origen, a. desti);
           res.inseriAresta(a.origen, a.desti, a.pes);
       }
    }
    return res;
}
```

- Graf d'entrada:



- Arbre de cobriment mínim:



8.3. ALGORISME DE KRUSKAL: ANÀLISI DE COSTOS

El coste es $O(|A| \log |A|)$

construir *mfsets* de talla $\|V\|$	$O(\|V\|)$
construir *minheap*	$+O(\|A\|)$
$O(\|A\|)$ esborrats en *minheap*	$+O(\|A\| \log \|A\|)$
$O(\|A\|)$ operacions *mfset*	$+O(\|A\|)$

En general, el cost és prou inferior a la cota $O(|A| \log |A|)$:

Si m és el nombre d'iteracions del bucle **while**, típicament $m \approx |V|$ y si $|V| \ll |A| \leq |V|^2$, en la pràctica, el cost està més prop a

$$|A| + |V| \log |V|$$