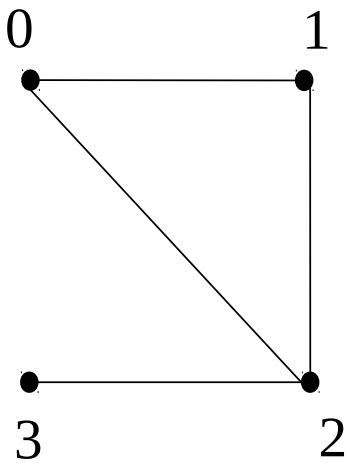
Grafuri

SD 2020/2021

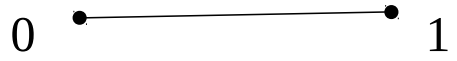
Conținut

Grafuri

- $G = (V, E)$
  - V mulțime de vârfuri
  - E mulțime de muchii; o muchie = o pereche neordonată de vârfuri distincte



$V = \{0, 1, 2, 3\}$
$E = \{\{0, 1\}, \{0, 2\}, \{1, 2\}, \{2, 3\}\}$
$u = \{0, 1\} = \{1, 0\}$

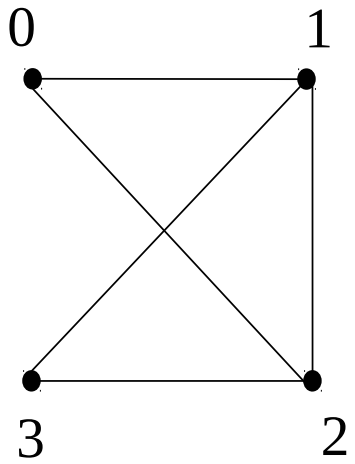


0,1 - extremitățile lui u
u este incidentă în 0 și 1
0 și 1 sunt adiacente (vecine)

Grafuri

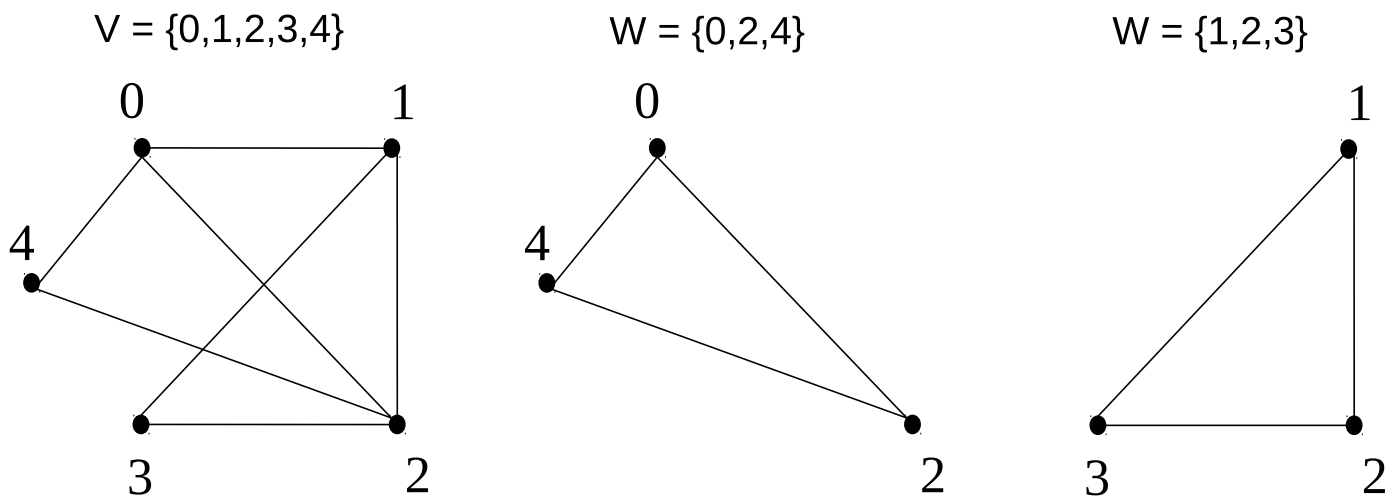
- Mers de la u la v (*walk*): $u = i_0, \{i_0, i_1\}, i_1, \cdots, \{i_{k-1}, i_k\}, i_k = v$
  3, {3,2}, 2, {2,0}, 0, {0,1}, 1, {1,3},3, {3,2}, 2
- parcurs (*trail*): mers în care oricare două muchii sunt distincte
- circuit = parcurs închis în care oricare două muchii intermediare sunt distincte

- drum (*path*): mers în care oricare două vârfuri sunt distincte
- ciclu (*cycle*): drum închis $i_0 = i_k$



Subgraf indus

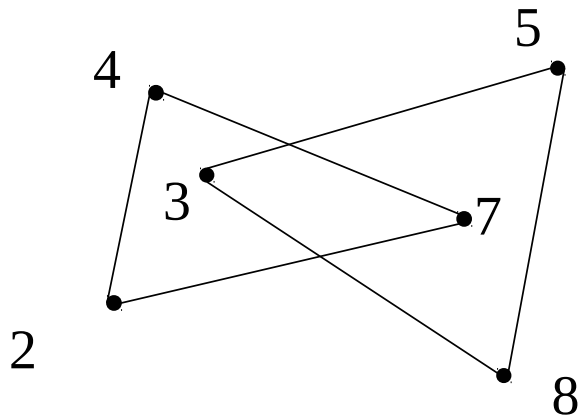
- $G = (V, E)$ – graf, W – submulțime a lui V
- Subgraf indus de W: $G'(W, E')$, unde $E' = \{\{i, j\} | \{i, j\} \in E \text{ și } i \in W, j \in W\}$



Grafuri - Conexitate

Orice graf poate fi exprimat ca fiind reuniunea disjunctă de subgrafuri induse, conexe și maximale (componente conexe).

- $i\ R\ j$ dacă și numai dacă există drum de la i la j
- R este relație de echivalență
- $V_1, \cdots, V_p$ clasele de echivalență
- $G_1, \cdots, G_p$ componente conexe, $G_i = (V_i, E_i)$ subgraful indus de V_i



$V_1 = \{2, 4, 7\}$

$E_1 = \{\{2, 4\}, \{4, 7\}, \{2, 7\}\}$

$V_2 = \{3, 5, 8\}$

$E_2 = \{\{3, 5\}, \{5, 8\}, \{8, 3\}\}$

- graf conex = graf cu o singură componentă conexă

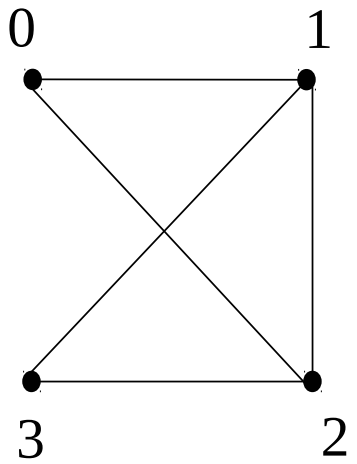
Tipul de date abstract **Graf**

- obiecte:
  - grafuri $G = (V, E)$, $V = \{0, 1, \cdots, n-1\}$
- operații:
  - grafVid()
    - intrare: nimic
    - ieșire: graful vid $(\emptyset, \emptyset)$
  - esteGrafVid()
    - intrare: $G = (V, E)$,
    - ieșire: true daca $G = (\emptyset, \emptyset)$, false în caz contrar
  - insereazaMuchie()
    - intrare: $G = (V, E)$, $i, j \in V$
    - ieșire: $G = (V, E \cup \{i, j\})$
  - insereazaVarf()
    - intrare: $G = (V, E)$, $V = \{0, 1, \cdots, n-1\}$
    - ieșire: $G = (V', E)$, $V' = \{0, 1, \cdots, n-1, n\}$

Tipul de date abstract **Graf**

- eliminaMuchie()
  - intrare: $G = (V, E)$, $i, j \in V$
  - ieșire: $G = (V, E \setminus \{i, j\})$
- eliminaVarf()
  - intrare: $G = (V, E)$, $V = \{0, 1, \cdots, n-1\}$, k
  - ieșire: $G = (V', E')$, $V' = \{0, 1, \cdots, n-2\}$

$$\{i', j'\} \in E' \Leftrightarrow (\exists \{i, j\} \in E)\ i \neq k, j \neq k,$$
$$i' = \textit{if}\ (i < k)\ \textit{then}\ i\ \textit{else}\ i - 1,$$
$$j' = \textit{if}\ (j < k)\ \textit{then}\ j\ \textit{else}\ j - 1$$



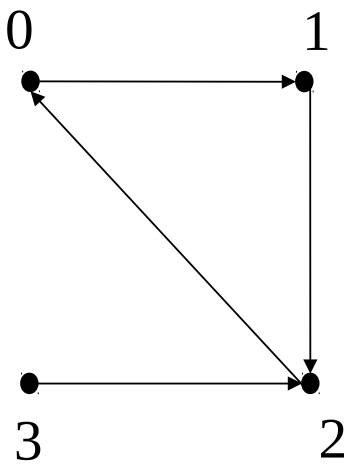
Tipul de date abstract **Graf**

- listaDeAdiacenta()
  - intrare: $G = (V, E)$, $i \in V$
  - ieșire: lista vârfurilor adiacente cu i
- listaVarfurilorAccesibile()
  - intrare: $G = (V, E)$, $i \in V$
  - ieșire: lista vârfurilor accesibile din i

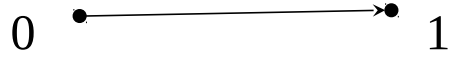
Conținut

Digraf (graf orientat)

- $D = (V, A)$
  - V mulțime de vârfuri
  - A mulțime de arce; un arc = o pereche ordonată de vârfuri distincte



$V = \{0, 1, 2, 3\}$
$A = \{(0, 1), (2, 0), (1, 2), (3, 2)\}$
$a = (0, 1) \neq (1, 0)$

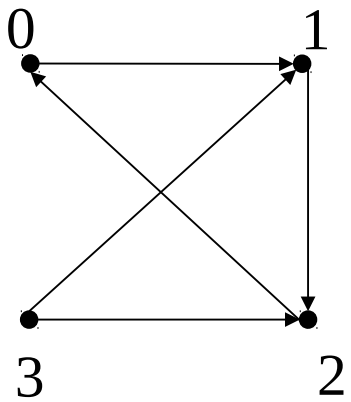


0 – sursa lui a
1 – destinația lui a

Digraf

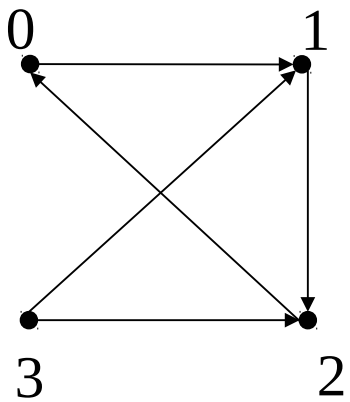
- mers (*walk*): $i_0, (i_0, i_1), i_1, \cdots, (i_{k-1}, i_k), i_k$
  3, (3,2), 2, (2,0), 0, (0,1), 1, (1,2), 2, (2,0), 0
- parcurs (*trail*): mers în care oricare două arce sunt distincte
- circuit = parcurs închis în care oricare două arce intermediare sunt distincte

- drum (*path*): mers în care oricare două vârfuri sunt distincte
- ciclu (*cycle*): drum închis $i_0 = i_k$



Digraf - Conexitate

- i R j dacă și numai dacă există drum de la i la j și drum de la j la i
- R este relație de echivalență
- $V_1, \cdots, V_p$ clasele de echivalență
- $G_1, \cdots, G_p$ componente tare conexe, $G_i = (V_i, A_i)$ subdigraful indus de V_i
- digraf tare conex = digraf cu o singură componentă tare conexă
- digraf conex



$$V1 = \{0, 1, 2\}$$
$$A1 = \{(0, 1), (1, 2), (2, 0)\}$$

$$V2 = \{3\}$$
$$A2 = \emptyset$$

Tipul de date abstract **Digraf**

- obiecte: digrafuri $D = (V, A)$
- operații:
  - digrafVid()
    - intrare: nimic
    - ieșire: digraful vid $(\emptyset, \emptyset)$
  - esteDigrafVid()
    - intrare: $D = (V, A)$,
    - ieșire: true dacă $D = (\emptyset, \emptyset)$, false în caz contrar
  - insereazaArc()
    - intrare: $D = (V, A)$, $i, j \in V$
    - ieșire: $D = (V, A \cup (i, j))$
  - insereazaVarf()
    - intrare: $D = (V, A)$, $V = \{0, 1, \cdots, n-1\}$
    - ieșire: $D = (V', A)$, $V' = \{0, 1, \cdots, n-1, n\}$

Tipul de date abstract **Digraf**

- eliminaArc()
  - intrare: $D = (V, A),\ i, j \in V$
  - ieșire: $D = (V, A \setminus (i, j))$
- eliminaVarf()
  - intrare: $D = (V, A),\ V = \{0, 1, \cdots, n-1\},\ k$
  - ieșire: $D = (V', A'),\ V' = \{0, 1, \cdots, n-2\}$

$$\{i', j'\} \in A' \Leftrightarrow (\exists \{i, j\} \in A)\ i \neq k, j \neq k,$$
$$i' = \textit{if}\ (i < k)\ \textit{then}\ i\ \textit{else}\ i - 1,$$
$$j' = \textit{if}\ (j < k)\ \textit{then}\ j\ \textit{else}\ j - 1$$

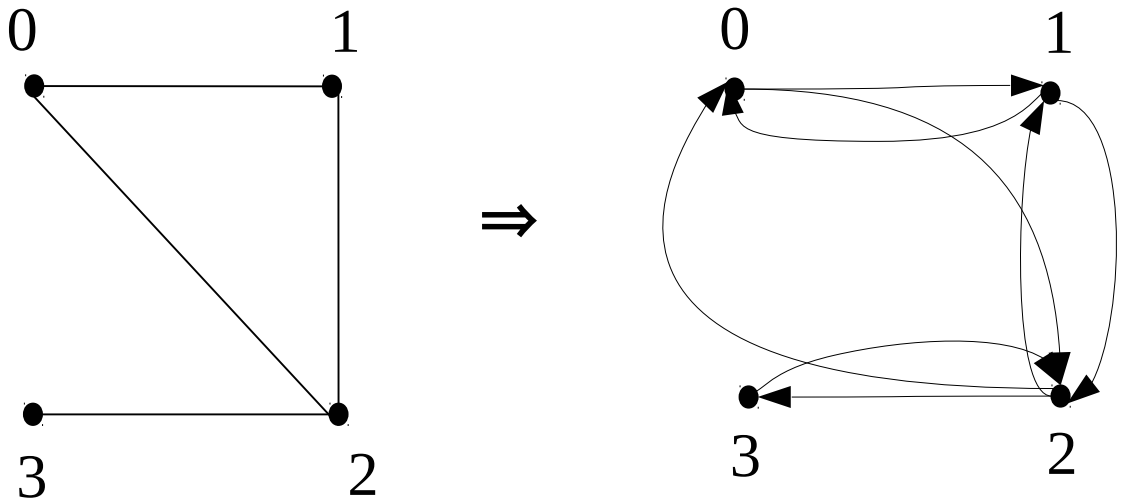
Tipul de date abstract **Digraf**

- listaDeAdiacentaExterioara()
  - intrare: $D = (V, A),\ i \in V$
  - ieșire: lista vârfurilor destinatare ale arcelor care pleacă din i
- listaDeAdiacentaInterioara()
  - intrare: $D = (V, A),\ i \in V$
  - ieșire: lista vârfurilor sursă ale arcelor care sosesc în i
- listaVarfurilorAccesibile()
  - intrare: $D = (V, A),\ i \in V$
  - ieșire: lista vârfurilor accesibile din i

Reprezentarea grafurilor ca digrafuri

$G = (V, E) \implies D(G) = (V, A)$
$\{i, j\} \in E \implies (i, j),\ (j, i) \in A$

- topologia este păstrată
  - lista de adiacență a lui i în G = lista de adiacență exterioară (=interioară) a lui i în D



Conținut

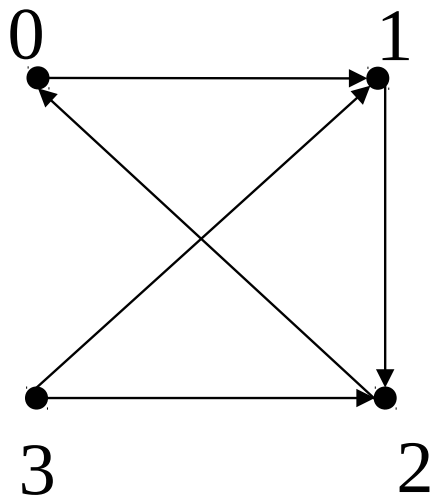
Implementarea cu matrici de adiacență a digrafurilor

- reprezentarea digrafurilor
  - n numărul de vârfuri
  - m numărul de arce (opțional)
  - o matrice $(a[i,j] |\ 1 \leq i,\ j \leq n)$
    $a[i,j] = \mathit{if}\ (i,j) \in A\ \mathit{then}\ 1\ \mathit{else}\ 0$

  - dacă digraful reprezintă un graf, atunci $a[i,j]$ este simetrică
  - lista de adiacență exterioară a lui $i \subseteq$ linia i
  - lista de adiacență interioară a lui $i \subseteq$ coloana i

Implementarea cu matrici de adiacență

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



Implementarea cu matrici de adiacență

- operații
  - digrafVid
    $n \leftarrow 0;\ m \leftarrow 0$
  - insereazaVarf: $O(n)$
  - insereazaArc: $O(1)$
  - eliminaArc: $O(1)$

Implementarea cu matrici de adiacență

- eliminaVarf()

Procedure *eliminaVarf(a, n, k)*
begin
    for $i \leftarrow 0$ **to** $n-1$ **do**
        for $j \leftarrow 0$ **to** $n-1$ **do**
            if $(i > k)$ **then**
                $a[i-1, j] \leftarrow a[i, j]$
            if $(j > k)$ **then**
                $a[i, j-1] \leftarrow a[i, j]$
    $n \leftarrow n-1$
end

timp de execuție: $O(n^2)$

Implementarea cu matrici de adiacență

- listaVarfurilorAccesibile()
  - Dacă $i = j$ atunci j este accesibil din i
    Dacă $i \neq j$ atunci există drum $i \rightsquigarrow j$ dacă există arc $i \rightarrow j$ sau există k:
    $\exists i \rightsquigarrow k, k \rightsquigarrow j$

Lista vârfurilor accesibile

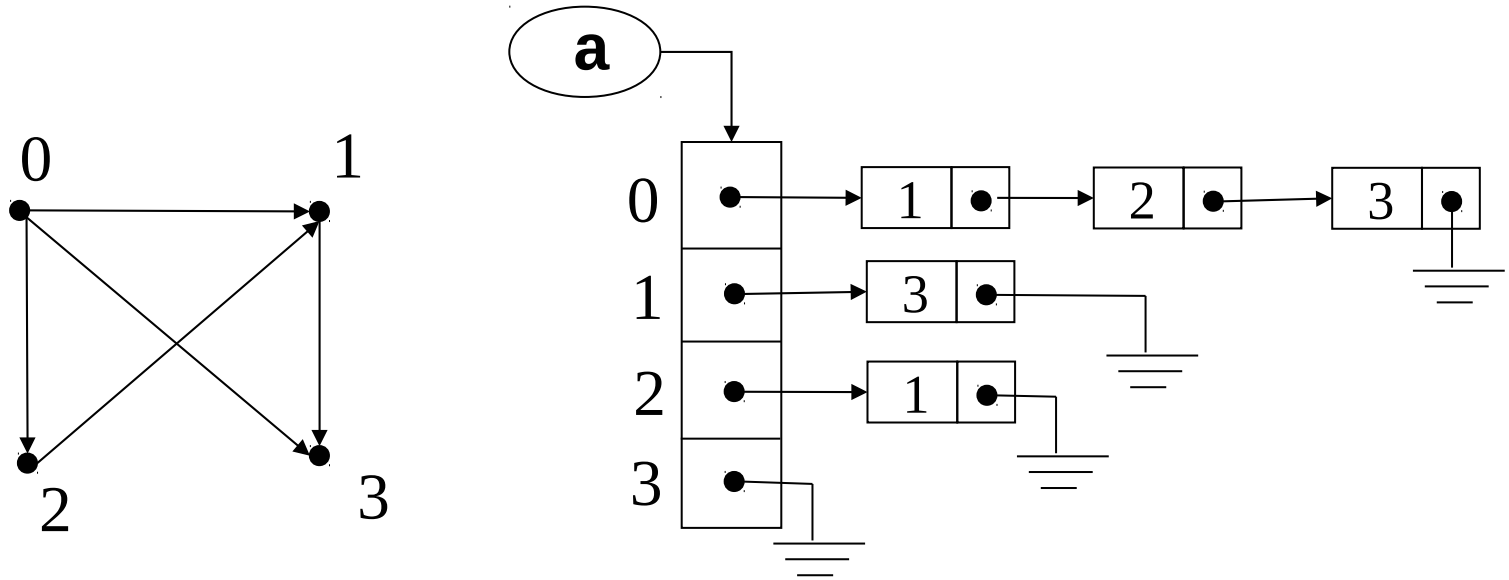
Procedure *inchReflTranz(a, n, b) // (Warshall, 1962)*
begin
    for $i \leftarrow 0$ **to** $n-1$ **do**
        for $j \leftarrow 0$ **to** $n-1$ **do**
            $b[i,j] \leftarrow a[i,j]$
            if $(i = j)$ **then**
                $b[i,j] \leftarrow 1$
    for $k \leftarrow 0$ **to** $n-1$ **do**
        for $i \leftarrow 0$ **to** $n-1$ **do**
            if $(b[i,k] = 1)$ **then**
                for $j \leftarrow 0$ **to** $n-1$ **do**
                    if $(b[k,j] = 1)$ **then**
                        $b[i,j] \leftarrow 1$
end

timp de execuție: $O(n^3)$

Conținut

Implementarea cu liste de adiacență

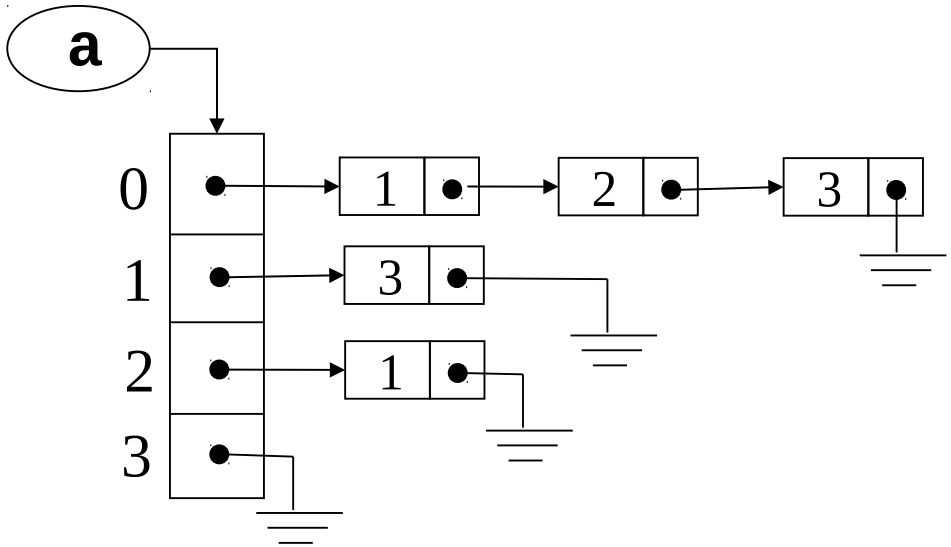
- reprezentarea digrafurilor cu liste de adiacență exterioară



  - un tablou $a[0..n-1]$ de liste înlănțuite (pointeri)
  - $a[i]$ este lista de adiacență exterioară corespunzătoare lui i

Implementarea cu liste de adiacență

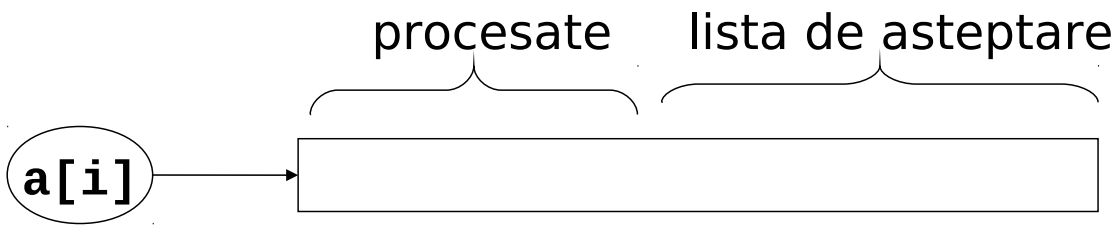
- operații
  - digrafVid
  - insereazaVarf: $O(1)$
  - insereazaArc: $O(1)$
  - eliminaVarf: $O(n + m)$
  - eliminaArc: $O(m)$



Conținut

Digrafuri: explorare sistematică

- se gestionează două mulțimi
  - S = mulțimea vârfurilor vizitate deja
  - $SB \subseteq S$ submulțimea vârfurilor pentru care există șanse să găsim vecini nevizitați încă
- lista de adiacență (exterioară) a lui i este divizată în două:



Digrafuri: explorare sistematică

- pasul curent
  - citește un vârf i din SB
  - extrage un j din lista de "așteptare" a lui i (dacă este nevidă)
  - dacă j nu este în S, atunci îl adaugă la S și la SB
  - dacă lista de "așteptare" a lui i este vidă, atunci elimină i din SB
- inițial
  - $S = SB = \{i_0\}$
  - lista de "așteptare a lui i" = lista de adiacenta a lui i
- terminare $SB = \emptyset$

Digrafuri: explorare sistematică

Procedure *explorare(a, n, i0)*
begin
    for $i \leftarrow 0$ **to** $n-1$ **do**
        $p[i] \leftarrow a[i]$
    $SB \leftarrow (i0)$
    $viziteaza(i0); S \leftarrow (i0)$
    while $(SB \neq \emptyset)$ **do**
        $i \leftarrow citeste(SB)$
        if $(p[i] = NULL)$ **then**
            $SB \leftarrow SB \setminus \{i\}$
        else
            $j \leftarrow p[i] \rightarrow varf$
            $p[i] \leftarrow p[i] \rightarrow succ$
            if $(j \notin S)$ **then**
                $SB \leftarrow SB \cup \{j\}$
                $viziteaza(j)$
                $S \leftarrow S \cup \{j\}$
end

Explorare sistematică: complexitate

Teorema

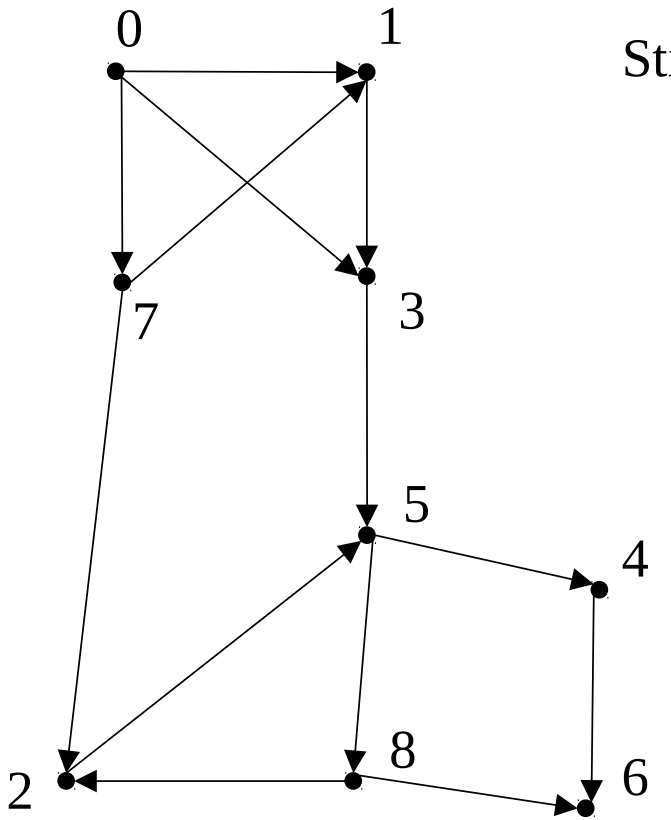
În ipoteza că operațiile peste S și SB precum și viziteaza() se realizează în $O(1)$, complexitatea timp, în cazul cel mai nefavorabil, a algoritmului explorare este $O(n + m)$.

Explorarea DFS (*Depth First Search*)

- SB este implementată ca stivă

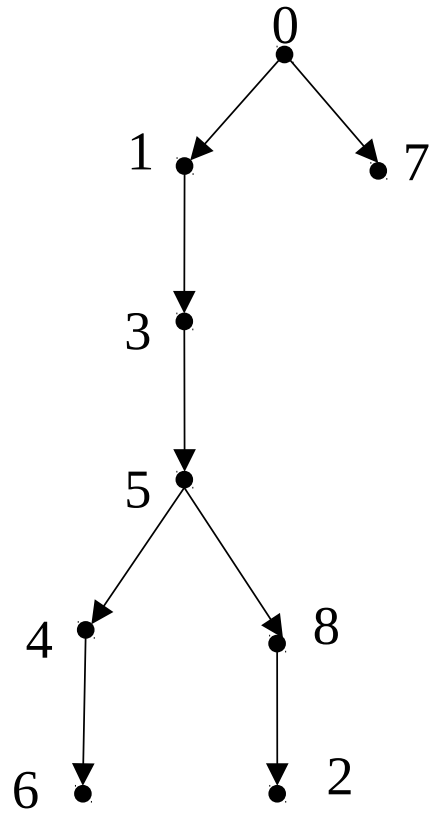
$$\begin{aligned} SB \leftarrow (i0) &\Leftrightarrow SB \leftarrow stivaVida() \\ &\quad\; push(SB, i0) \\ i \leftarrow citeste(SB) &\Leftrightarrow i \leftarrow top(SB) \\ SB \leftarrow SB \setminus \{i\} &\Leftrightarrow pop(SB) \\ SB \leftarrow SB \cup \{j\} &\Leftrightarrow push(SB, j) \end{aligned}$$

Explorarea DFS: exemplu



Stiva

6 2
4 8
5 5
3 3
1 1 7
0 0 0



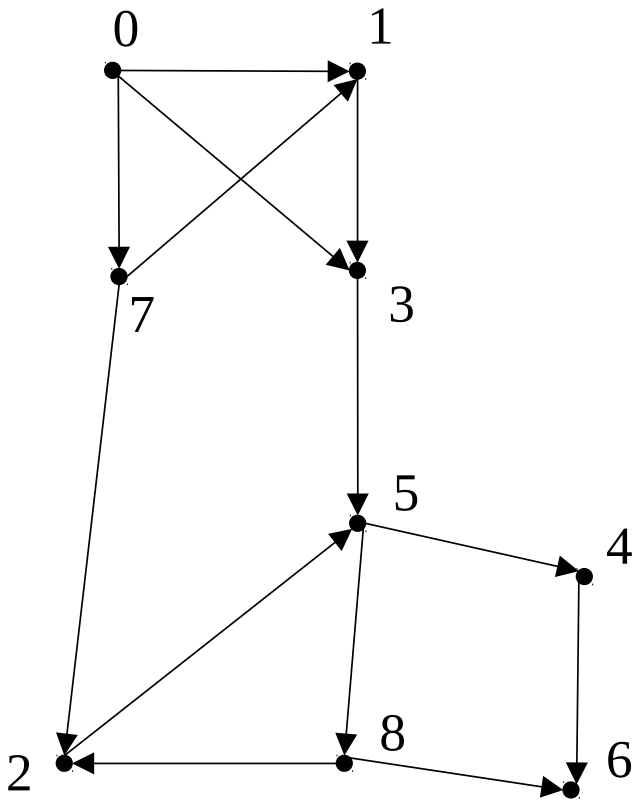
Arborele DFS

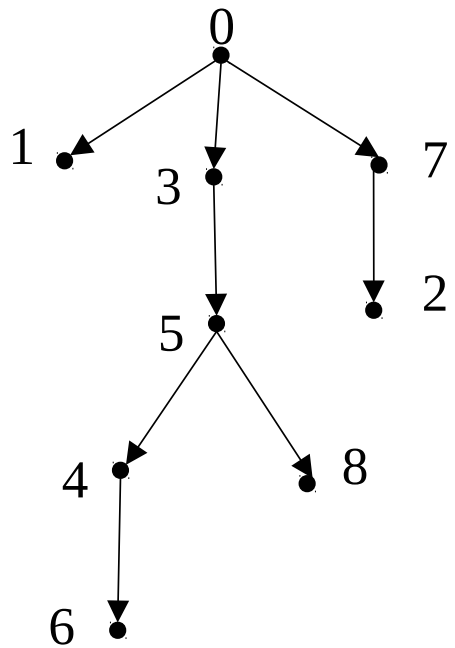
Explorarea BFS (*Breadth First Search*)

- SB este implementată ca o coadă

$$SB \leftarrow (i0) \Leftrightarrow SB \leftarrow coadaVida();$$
$$insereaza(SB, i0)$$
$$i \leftarrow citeste(SB) \Leftrightarrow citeste(SB, i)$$
$$SB \leftarrow SB \setminus \{i\} \Leftrightarrow elimina(SB)$$
$$SB \leftarrow SB \cup \{j\} \Leftrightarrow insereaza(SB, j)$$

Explorarea BFS: exemplu





Arborele BFS

Conținut

Determinarea componentelor conexe (grafuri neorientate)

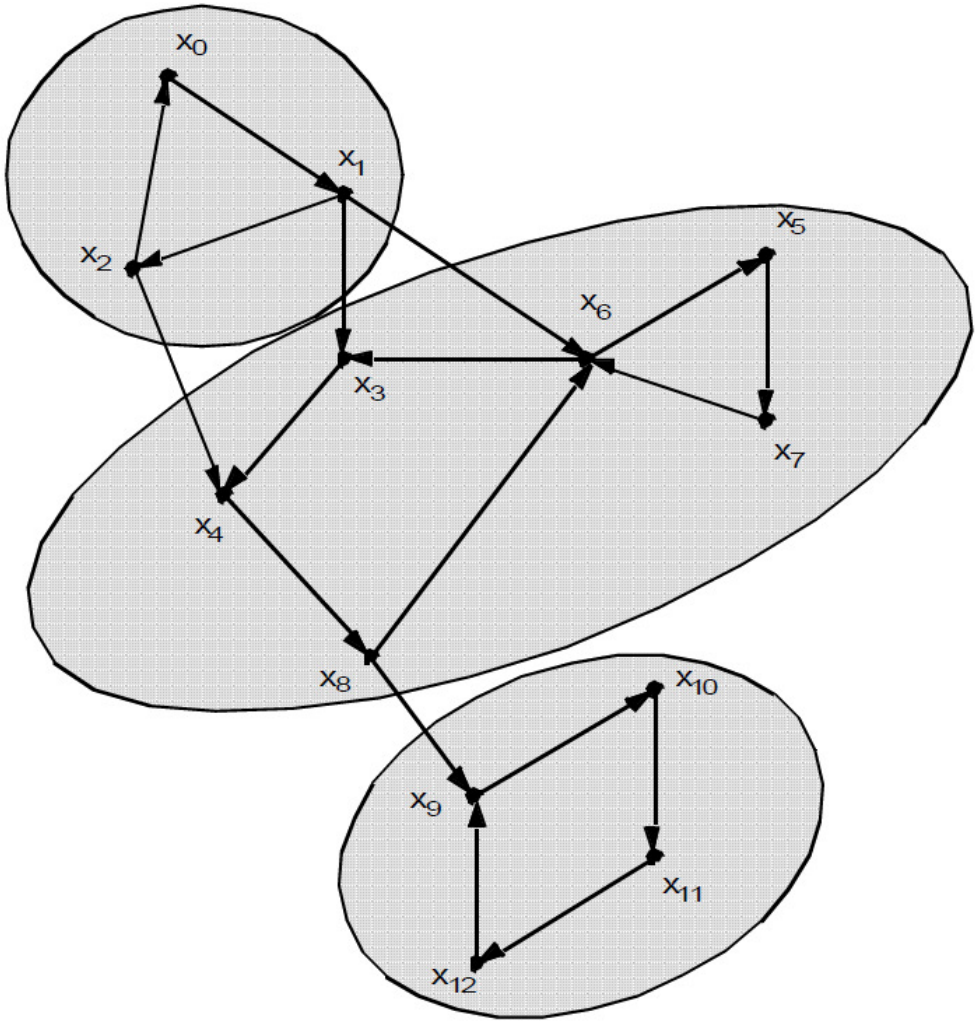
```
Function CompConexeDFS(D)
begin
    for i ← 0 to n − 1 do
        culoare[i] ← 0
    k ← 0
    for i ← 0 to n − 1 do
        if (culoare[i] = 0) then
            k ← k + 1
            DfsRecCompConexe(i, k)
    return k
end
```

Determinarea componentelor conexe (grafuri neorientate)

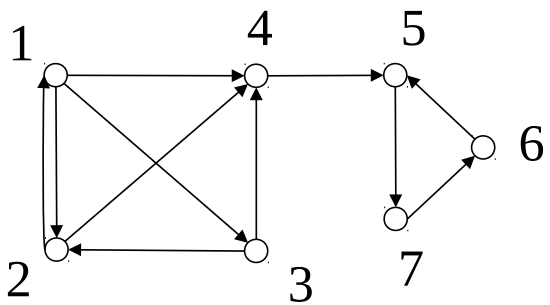
Procedure *DfsRecCompConexe(i, k)*
begin
    $culoare[i] \leftarrow k$
    for *(fiecare vârf j în listaDeAdiac(i))* **do**
        if $(culoare[j] = 0)$ **then**
            $DfsRecCompConexe(j, k)$
end

Componente tare conexe (digrafuri)

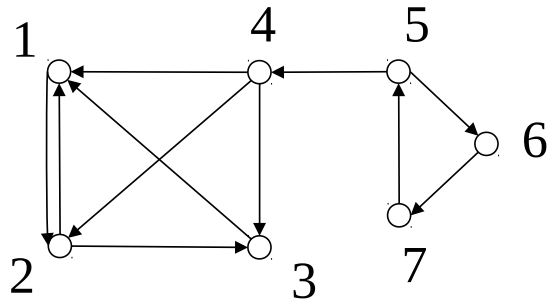
O componentă tare conexă este o mulțime maximală de vârfuri a.î. pentru fiecare $u, v : u \rightsquigarrow v, v \rightsquigarrow u$.



Componente tare conexe: exemplu

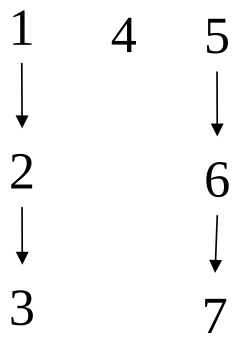


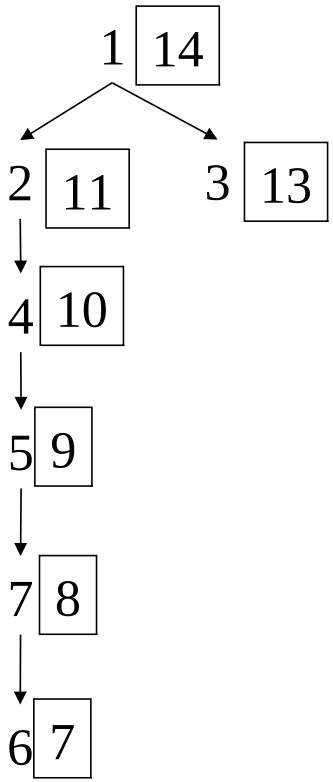
D



D^T

$1 \rightarrow (2, 3, 4)$
$2 \rightarrow (1, 4)$
$3 \rightarrow (2, 4)$
$4 \rightarrow (5)$
$5 \rightarrow (7)$
$6 \rightarrow (5)$
$7 \rightarrow (6)$





Determinarea componentelor tare conexe

Procedure *DfsCompTareConexe(D)*
begin
    for $i \leftarrow 0$ **to** $n-1$ **do**
        $culoare[i] \leftarrow 0$
        $tata[i] \leftarrow -1$
    $timp \leftarrow 0$
    for $i \leftarrow 0$ **to** $n-1$ **do**
        if $(culoare[i] = 0)$ **then**
            DfsRecCompTareConexe(i)
end

Determinarea componentelor tare conexe

Procedure *DfsRecCompTareConexe(i)*
begin
    $timp \leftarrow timp + 1$
    $culoare[i] \leftarrow 1$
    for *(fiecare vârf j in listaDeAdiac(i))* **do**
        if $(culoare[j] = 0)$ **then**
            $tata[j] \leftarrow i$
            DfsRecCompTareConexe(j)
    $timp \leftarrow timp + 1$
    $timpFinal[i] \leftarrow timp$
end

Determinarea componentelor tare conexe

Notație: $D^T = (V, A^T),\ (i,j) \in A \Leftrightarrow (j,i) \in A^T$

Procedure *CompTareConexe(D)*
begin

1. *DFSCompTareConexe*(D)
2. calculează D^T
3. *DFSCompTareConexe*(D^T) dar considerând în bucla *for* principală vârfurile în ordinea descrescătoare a timpilor finali de vizitare *timpFinal*[i]
4. returnează fiecare arbore calculat la pasul 3 ca fiind o componentă tare conexă separată

end

Determinarea componentelor tare conexe: complexitate

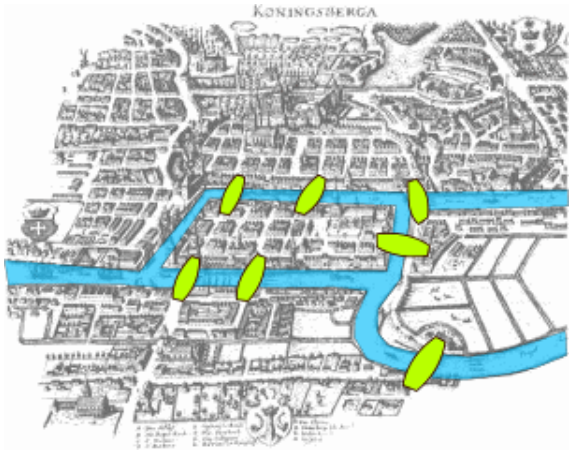
- *DFSCompTareConexe*(D): $O(n+m)$
- calculează D^T: $O(m)$
- *DFSCompTareConexe*(D^T): $O(n+m)$
- Total: $O(n+m)$

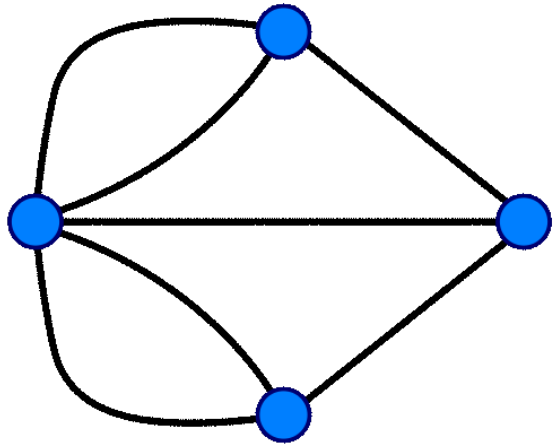
Aplicații

- Algoritmică, probleme de drum, rețele de calculatoare (rutare), genomică (rețele de aliniere, asamblarea genomului), *multi-relational data mining*, cercetări operaționale (planificare), inteligență artificială (satisfacerea restricțiilor), etc.

Aplicații

Problema celor șapte poduri din Konigsberg (1736): identificarea unui drum, pornind dintr-o zonă a orașului pentru a traversa cele 7 poduri o singură dată?





Zonele: vârfuri, podurile: muchii

Este posibil să alegem un vârf, să parcurgem muchiile și să ne intoarcem în varful ales, acoperind toate muchiile o dată?

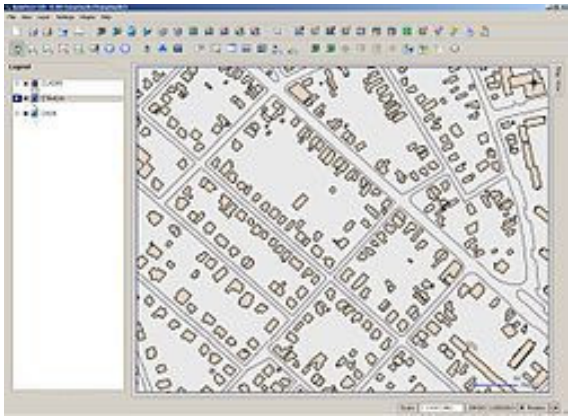
Aplicații

Probleme de planificare

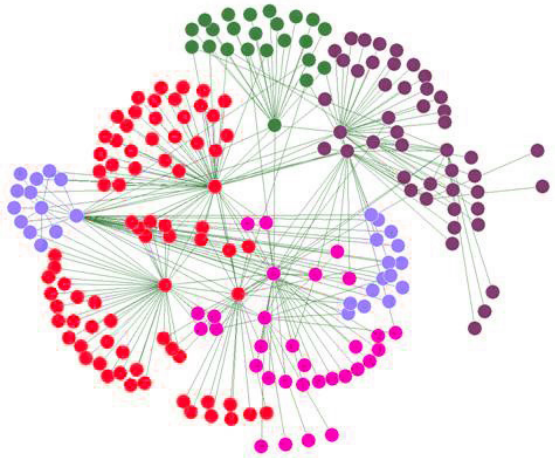
- Digrafurile aciclice pot modela precedența între evenimente
- O sortare topologică a unui digraf $D = (V, A)$ este o ordonare a vârfurilor sale astfel încât dacă $(i, j) \in A$, atunci i apare înaintea lui j.

Aplicații

- Motorul de căutare Google: algoritmul *PageRank* - pentru a determina cât de importantă este o anumită pagină
- Sistem informațional geografic (*GIS*): *Google Maps*, *Bing Maps*



- Rețele sociale



DNA word design

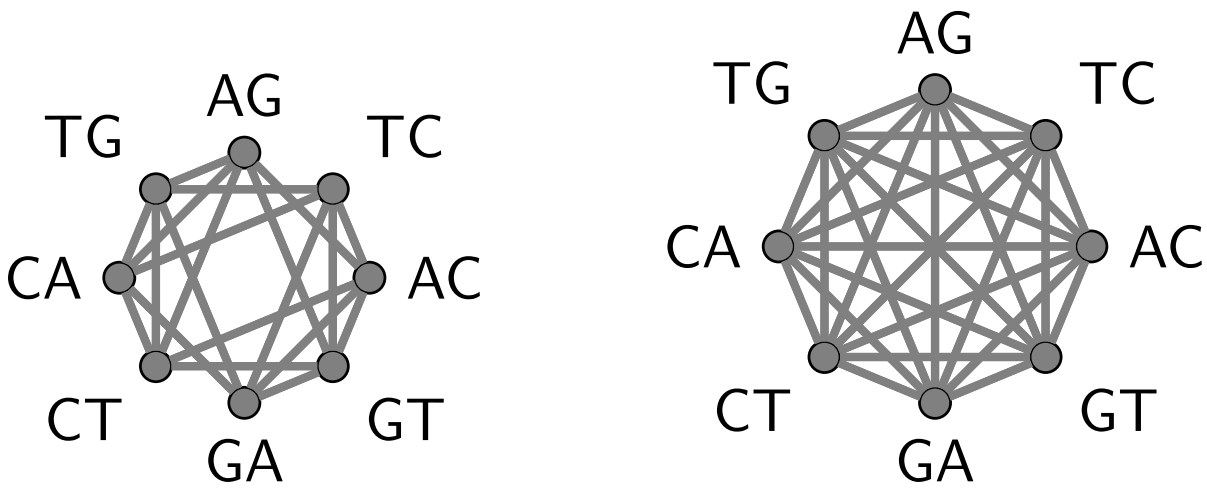
- Proiectarea de coduri ADN care satisfac anumite restrictii combinatoriale; utilizare: in calculul biomolecular pentru a memora informatii sau pentru manipularea moleculelor

- Determina cea mai mare multime S de cuvinte de lungime n peste alfbetul $\{A, C, G, T\}$ a.i.:
  - *GC Content Constraint*: fiecare cuvant are 50% simboluri din $\{C, G\}$
  - *Hamming Distance Constraint*: fiecare pereche de cuvinte $w_1 \neq w_2$ difera in cel putin d pozitii: $H(w_1, w_2) \geq d$
  - *Reverse Complement Hamming Distance Constraint*: $H(R(w_1), C(w_2)) \geq d$, unde $R(w)$ inversul cuvantului w si $C(w)$ complementul lui w ($C \leftrightarrow G$, $A \leftrightarrow T$)

Modelare

- pentru fiecare cuvant ADN asignam un varf v_i
- $E = E_{HD} \cup E_{RC}$ (E_{HD} perechi de cuvinte care au un conflict HD, E_{RC} perechi de cuvinte cu un conflict RC)
- solutie: o multime independenta maximala

Figura: Grafurile pentru cuvinte de dimensiune 2 si distanta Hamming $d = 2$ (stanga) si $d = 3$ (dreapta)



Solutie de 136 cuvinte pentru instanta $n = 8, d = 4$

AAACCACC	ACCAGTGT	ACCCAAGA	ACGTAGTG	ACTGACGT	AGGAAGCT
AGTCCTCT	AGTTGGCA	ATCCCGTT	ATGGGCTT	CAAACCTC	CAAGAGAC
CAAGCAGT	CACAGTTG	CACCAATC	CAGATGGT	CAGGATCT	CATCGTGT
CATGACTG	CATTCGCT	CCAGTCTT	CCCTGATT	CCGACTTT	CCTCAGTT
CGAAGGTT	CGACACAT	CGATTTGG	CGCACAAT	CGCCTTTT	CGCTAGTA
CGGTGTAT	CGTAAAGG	CGTGTGAT	CTATGCCT	CTCGTACT	CTGAAGAG
CTGCAAGT	CTTACCGT	CTTCCTAG	GAAAGCGT	GAACAGCT	GAACGTAG
GAAGGATC	GACATGAG	GACCTAGT	GACTGTCT	GAGAAGTC	GAGACACT
GAGTACAG	GATGCAAG	GATGTCCT	GCAATAGG	GCAGCTAT	GCCTAGAT
GCGATCAT	GCGGAATT	GCTCGAAT	GCTTATGG	GGAAATGC	GGACCATT
GGATAACG	GGCAACTT	GGGTTGTT	GGTATTCG	GGTTCCAT	GGTTTAGC
GTAACCAG	GTAGAGTG	GTATCGGT	GTCAGTAC	GTCCAAAG	GTCGATGT
GTGAGATG	GTGCTTCT	GTTAGGCT	GTTCTCTG	GTTGACAC	TAACACGC
TAAGCTCG	TACACAGC	TACCGCTT	TAGATCCG	TAGGAAGG	TAGGCGTT
TAGTGTGC	TATCGACG	TATGTGGC	TCAACGTG	TCACGTCT	TCAGACAG
TCATGCTC	TCCATGCT	TCCCATTG	TCCGTATC	TCCTCAAG	TCGAAGGA
TCGAGTAG	TCGCAAAC	TCGGTTGT	TCGTACCT	TCTACCAC	TCTCCTGA
TCTCTGAG	TCTGCACT	TGAACCCT	TGACCTAC	TGAGAGGT	TGATGGAG
TGCAGTCA	TGCGTTAG	TGCTACAC	TGCTCTGT	TGGAGAGT	TGGATGAC
TGGCTATG	TGGGATTC	TGTAGCTG	TGTCTCGT	TGTGACCA	TGTGGAAC
TGTTCGTC	TTAAGGGC	TTACCAGG	TTAGTCCC	TTCAACGG	TTCCTTGC
TTCGCCAT	TTCGGGTA	TTCTGACC	TTGACTCC	TTGCCCTA	TTGCGGAT
TTGTTGGG	TTTCAGCC	TTTGGTGG	TTTTCCCG		