

Graphs

DS 2018/2019

Abstract data type Graph

Abstract data type Digraph

The implementation with adjacency matrices

The implementation with adjacency linked lists

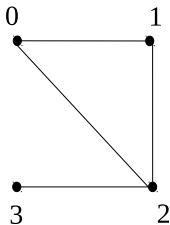
Graph traversal algorithms (DFS, BFS)

Finding the (strongly) connected components

Graphs

► $G = (V, E)$

- V a set of **vertices**
- E a set of **edges**; an **edge** = a non-ordered pair of distinct vertices



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

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

$$u = \{0, 1\} = \{1, 0\}$$



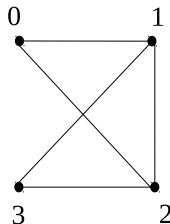
0,1 - the **ends** of u

u is **incident** with 0 and 1

0 and 1 are **adjacent (neighbors)**

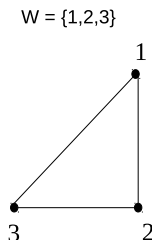
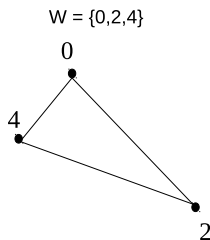
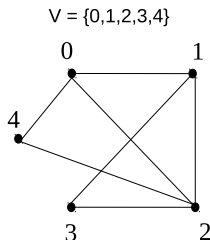
Graphs

- ▶ **Walk from u to v :** $u = i_0, \{i_0, i_1\}, i_1, \dots, \{i_{k-1}, i_k\}, i_k = v$
 $3, \{3,2\}, 2, \{2,0\}, 0, \{0,1\}, 1, \{1,3\}, 3, \{3,2\}, 2$
- ▶ **Trail:** a walk where any two edges are distinct
- ▶ **Path:** a walk where any two vertices are distinct
- ▶ **Cycle:** closed walk ($i_0 = i_k$)
- ▶ **Circuit** = a closed walk where any two intermediate vertices are distinct



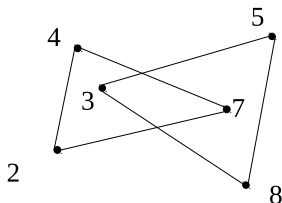
Induced subgraph

- ▶ $G = (V, E)$ – a graph, W – a subset of V
- ▶ **Induced subgraph:** $G'(W, E')$, where
 $E' = \{\{i, j\} | \{i, j\} \in E \text{ și } i \in W, j \in W\}$



Graphs - Connectivity

- ▶ $i R j$ if and only if there is a path from i to j
- ▶ R is an equivalence relation
- ▶ V_1, \dots, V_p equivalence classes
- ▶ $G_i = (V_i, E_i)$ a subgraph induced by V_i
- ▶ G_1, \dots, G_p – **connected components**
- ▶ **connected graph** = a graph with a single connected component



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

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

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

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

Abstract data type **Graph**

- ▶ **objects:**

- ▶ graphs $G = (V, E)$, $V = \{0, 1, \dots, n-1\}$

- ▶ **operations:**

- ▶ **emptyGraph()**

- ▶ input: nothing
 - ▶ output: the empty graph (\emptyset, \emptyset)

- ▶ **isEmptyGraph()**

- ▶ input: $G = (V, E)$,
 - ▶ output: true if $G = (\emptyset, \emptyset)$, false other way

- ▶ **insertEdge()**

- ▶ input: $G = (V, E)$, $i, j \in V$
 - ▶ output: $G = (V, E \cup \{i, j\})$

- ▶ **insertVertex()**

- ▶ input: $G = (V, E)$, $V = \{0, 1, \dots, n-1\}$
 - ▶ output: $G = (V', E)$, $V' = \{0, 1, \dots, n-1, n\}$

Abstract data type **Graph**

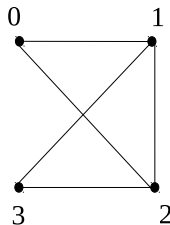
► removeEdge()

- input: $G = (V, E)$, $i, j \in V$
- output: $G = (V, E - \{i, j\})$

► removeVertex()

- input: $G = (V, E)$, $V = \{0, 1, \dots, n-1\}$, k
- output: $G = (V', E')$, $V' = \{0, 1, \dots, n-2\}$

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



Abstract data type **Graph**

▶ `adjacencyList()`

- ▶ input: $G = (V, E)$, $i \in V$
- ▶ output: the list of vertices adjacent with i

▶ `listOfReachableVertices()`

- ▶ input: $G = (V, E)$, $i \in V$
- ▶ output: the list of vertices reachable from i

Abstract data type Graph

Abstract data type Digraph

The implementation with adjacency matrices

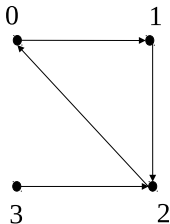
The implementation with adjacency linked lists

Graph traversal algorithms (DFS, BFS)

Finding the (strongly) connected components

Digraph (directed graph)

- ▶ $D = (V, A)$
 - ▶ V a set of **vertices**
 - ▶ A a set of **arcs (directed edges)**; an **arc** = an ordered pair of distinct vertices



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

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

$$a = (0, 1) \neq (1, 0)$$

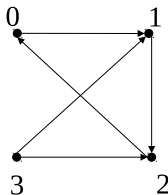


0 – the **tail** of a

1 – the **head** of a

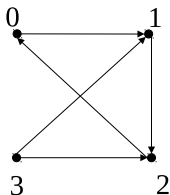
Digraph

- ▶ **Walk:** $i_0, (i_0, i_1), i_1, \dots, (i_{k-1}, i_k), i_k$
 $3, (3,2), 2, (2,0), 0, (0,1), 1, (1,2), 2, (2,0), 0$
- ▶ **Trail:** a walk where any two edges are distinct
- ▶ **Path:** a walk where any two vertices are distinct
- ▶ **Cycle:** closed walk ($i_0 = i_k$)
- ▶ **Circuit** = a closed walk where any two intermediate vertices are distinct



Digraph - Connectivity

- ▶ $i R j$ if and only if there is a path from i to j and a path from j to i
- ▶ R is an equivalence relation
- ▶ V_1, \dots, V_p the equivalence classes
- ▶ $G_i = (V_i, A_i)$ the subdigraph induced by V_i
- ▶ G_1, \dots, G_p – **strongly connected components**
- ▶ **digraph strongly connected** = digraph with a single strongly connected component



$$V1 = \{0, 1, 2\}$$

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

$$V2 = \{3\}$$

$$A2 = \emptyset$$

Abstract data type **Digraph**

- ▶ **objects**: digraphs $D = (V, A)$
- ▶ **operations**:
 - ▶ **emptyDigraph()**
 - ▶ input: nothing
 - ▶ output: the empty digraph (\emptyset, \emptyset)
 - ▶ **isEmptyDigraph()**
 - ▶ input: $D = (V, A)$,
 - ▶ output: true if $D = (\emptyset, \emptyset)$, false other way
 - ▶ **insertArc()**
 - ▶ input: $D = (V, A)$, $i, j \in V$
 - ▶ output: $D = (V, A \cup (i, j))$
 - ▶ **insertVertex()**
 - ▶ input: $D = (V, A)$, $V = \{0, 1, \dots, n-1\}$
 - ▶ output: $D = (V', A)$, $V' = \{0, 1, \dots, n-1, n\}$

Abstract data type **Digraph**

▶ removeArc()

- ▶ input: $D = (V, A)$, $i, j \in V$
- ▶ output: $D = (V, A - (i, j))$

▶ removeVertex()

- ▶ input: $D = (V, A)$, $V = \{0, 1, \dots, n-1\}$, k
- ▶ output: $D = (V', A')$, $V' = \{0, 1, \dots, n-2\}$

$$\begin{aligned}\{i', j'\} \in A' &\Leftrightarrow (\exists \{i, j\} \in A) \ i \neq k, j \neq k, \\ i' &= \text{if } (i < k) \text{ then } i \text{ else } i - 1, \\ j' &= \text{if } (j < k) \text{ then } j \text{ else } j - 1\end{aligned}$$

Abstract data type **Digraph**

▶ outAdjacencyList()

- ▶ input: $D = (V, A)$, $i \in V$
- ▶ output: the list of direct successors of i

▶ inAdjacencyList()

- ▶ input: $D = (V, A)$, $i \in V$
- ▶ output: the list of direct predecessors of i

▶ listOfReachableVertices()

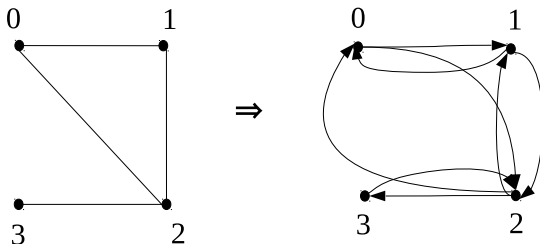
- ▶ input: $D = (V, A)$, $i \in V$
- ▶ output: the list of vertices reachable from i

The representation of graphs as digraphs

$$G = (V, E) \implies D(G) = (V, A)$$

$$\{i, j\} \in E \implies (i, j), (j, i) \in A$$

- ▶ the topology is preserved
 - ▶ the adjacency list of i in G = the out (=in) adjacency list of i in D



Content

Abstract data type Graph

Abstract data type Digraph

The implementation with adjacency matrices

The implementation with adjacency linked lists

Graph traversal algorithms (DFS, BFS)

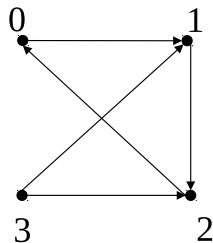
Finding the (strongly) connected components

The implementation of digraphs with adjacency matrices

- ▶ the representation of digraphs
 - ▶ n the number of vertices
 - ▶ m the number of arcs (optional)
 - ▶ a matrix $(a[i,j] \mid 1 \leq i, j \leq n)$
 $a[i,j] = \text{if } (i,j) \in A \text{ then } 1 \text{ else } 0$
 - ▶ if the digraph is a graph, then $a[i,j]$ is symmetric
 - ▶ the out adjacency list of $i \subseteq \text{line } i$
 - ▶ the in adjacency list of $i \subseteq \text{column } i$

The implementation with adjacency matrices

| | 0 | 1 | 2 | 3 |
|---|---|---|---|---|
| 0 | 0 | 1 | 0 | 0 |
| 1 | 0 | 0 | 1 | 0 |
| 2 | 1 | 0 | 0 | 0 |
| 3 | 0 | 1 | 1 | 0 |



The implementation with adjacency matrices

- ▶ operations

- ▶ emptyDigraph

- $n \leftarrow 0; m \leftarrow 0$

- ▶ insertVertex: $O(n)$

- ▶ insertArc: $O(1)$

- ▶ removeArc: $O(1)$

The implementation with adjacency matrices

► removeVertex()

Procedure *removeVertex*(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

the execution time: $O(n^2)$

The implementation with adjacency matrices

► listOfReachableVertices()

Procedure *reflTransClosure*(*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

the execution time: $O(n^3)$

Abstract data type Graph

Abstract data type Digraph

The implementation with adjacency matrices

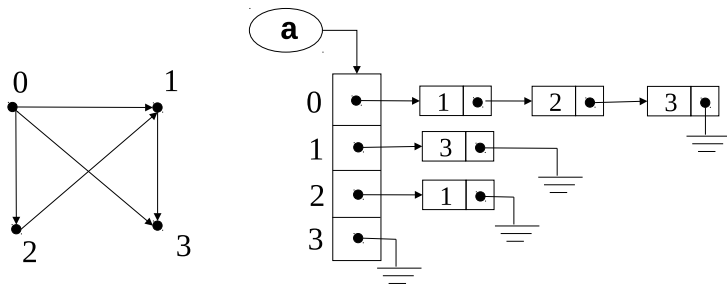
The implementation with adjacency linked lists

Graph traversal algorithms (DFS, BFS)

Finding the (strongly) connected components

The implementation with adjacency lists

- the representation of digraphs with adjacency lists

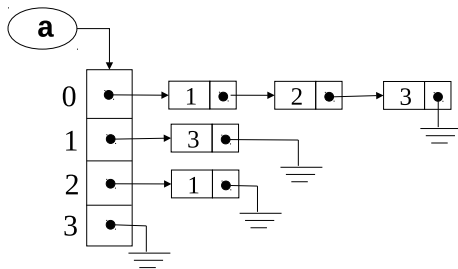


- a vector $a[0..n-1]$ of linked lists (pointers)
- $a[i]$ is the out adjacency list corresponding to i

The implementation with adjacency lists

► operations

- emptyDigraph
- insertVertex: $O(1)$
- insertArc: $O(1)$
- removeVertex: $O(n + m)$
- removeArc: $O(m)$



Content

Abstract data type Graph

Abstract data type Digraph

The implementation with adjacency matrices

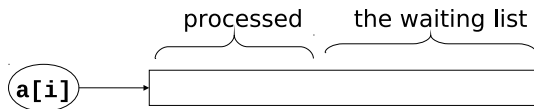
The implementation with adjacency linked lists

Graph traversal algorithms (DFS, BFS)

Finding the (strongly) connected components

Digraphs: systematic exploration

- ▶ it manages two sets
 - ▶ S = the set of already visited vertices
 - ▶ $SB \subseteq S$ the subset of vertices for which there are chances to find **neighbors** not visited yet
- ▶ the adjacency list of i is divided in:



Digraphs: systematic exploration

- ▶ the current step
 - ▶ read a vertex i from SB
 - ▶ extract j from the "waiting" list of i (if it is nonempty)
 - ▶ if j isn't in S , then add it to S and to SB
 - ▶ if the "waiting" list of i is empty, then remove i from SB
- ▶ initially
 - ▶ $S = SB = \{i_0\}$
 - ▶ the "waiting" list of i = the adjacency list of i
- ▶ termination $SB = \emptyset$

Digraphs: systematic exploration

Procedure *exploration*(*a*, *n*, *i0*, *S*)

begin

for *i* \leftarrow 0 **to** *n* - 1 **do**

$p[i] \leftarrow a[i]$

$SB \leftarrow (i0)$

$visit(i0); S \leftarrow (i0)$

while ($SB \neq \emptyset$) **do**

$i \leftarrow read(SB)$

if ($p[i] = NULL$) **then**

$SB \leftarrow SB - \{i\}$

else

$j \leftarrow p[i] \rightarrow varf$

$p[i] \leftarrow p[i] \rightarrow succ$

if ($j \notin S$) **then**

$SB \leftarrow SB \cup \{j\}$

$visit(j); S \leftarrow S \cup \{j\}$

end

Systematic exploration: complexity

Theorem

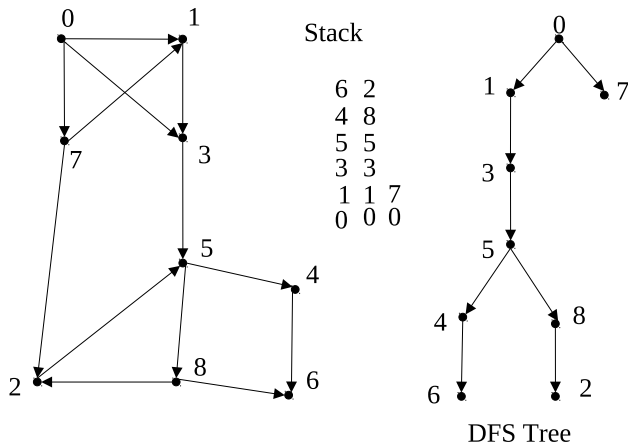
Assuming that the operations over S and SB as well as $\text{visit}()$ are achieved in $O(1)$, the time complexity, in the worst case, of the exploration algorithm is $O(n + m)$.

The DFS (*Depth First Search*) exploration

- ▶ SB is implemented as a stack

$$SB \leftarrow (i0) \Leftrightarrow SB \leftarrow \text{emptyStack}()$$
$$\text{push}(SB, i0)$$
$$i \leftarrow \text{read}(SB) \Leftrightarrow i \leftarrow \text{top}(SB)$$
$$SB \leftarrow SB - \{i\} \Leftrightarrow \text{pop}(SB)$$
$$SB \leftarrow SB \cup \{j\} \Leftrightarrow \text{push}(SB, j)$$

The DFS exploration: example

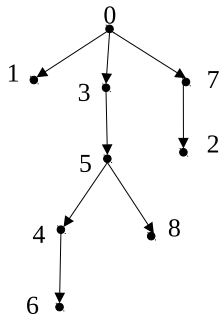
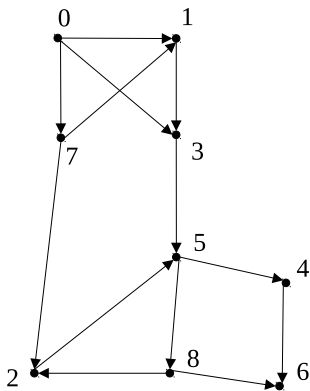


The BFS (*Breadth First Search*) exploration

- ▶ SB is implemented as a queue

$$SB \leftarrow (i_0) \Leftrightarrow SB \leftarrow \text{emptyQueue}();$$
$$\text{insert}(SB, i_0)$$
$$i \leftarrow \text{read}(SB) \Leftrightarrow \text{read}(SB, i)$$
$$SB \leftarrow SB - \{i\} \Leftrightarrow \text{remove}(SB)$$
$$SB \leftarrow SB \cup \{j\} \Leftrightarrow \text{insert}(SB, j)$$

The BFS exploration: example



BFS Tree

Content

Abstract data type Graph

Abstract data type Digraph

The implementation with adjacency matrices

The implementation with adjacency linked lists

Graph traversal algorithms (DFS, BFS)

Finding the (strongly) connected components

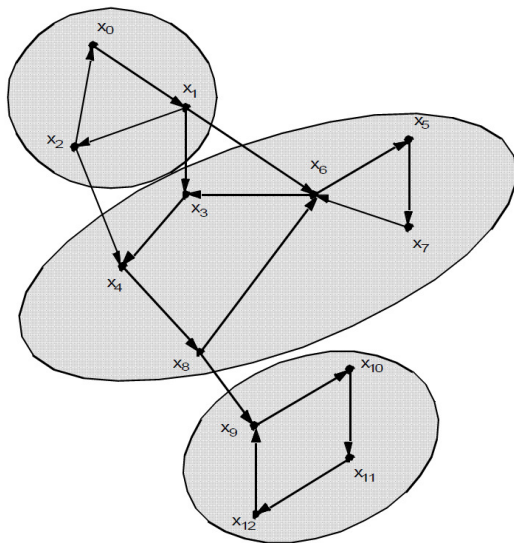
Finding the connected components (undirected graphs)

```
Function ConnectedCompDFS(D)  
begin  
  for  $i \leftarrow 0$  to  $n - 1$  do  
     $color[i] \leftarrow 0$   
   $k \leftarrow 0$   
  for  $i \leftarrow 0$  to  $n - 1$  do  
    if ( $color[i] = 0$ ) then  
       $k \leftarrow k + 1$   
      DfsRecConnectedComp( $i, k$ )  
  return  $k$   
end
```

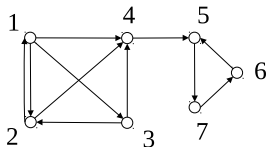
Finding the connected components (undirected graphs)

```
Procedure DfsRecConnectedComp(i, k)  
begin  
    color[i]  $\leftarrow k$   
    for (each vertex j in listaDeAdiac(i)) do  
        if (color[j] = 0) then  
            DfsRecConnectedComp(j, k)  
end
```

The strongly connected components (digraphs)

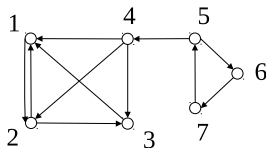


The strongly connected components: example



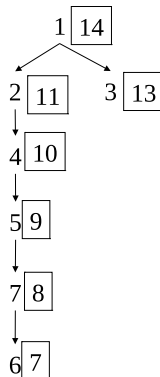
D

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



D^T

$1 \quad 4 \quad 5$
 $\downarrow \quad \quad \downarrow$
 $2 \quad \quad 6$
 $\downarrow \quad \quad \downarrow$
 $3 \quad \quad 7$



Finding the strongly connected components

Procedure *DfsStronglyConnectedComp*(*D*)

begin

for $i \leftarrow 0$ **to** $n - 1$ **do**

$color[i] \leftarrow 0$

$parent[i] \leftarrow -1$

$time \leftarrow 0$

for $i \leftarrow 0$ **to** $n - 1$ **do**

if ($color[i] = 0$) **then**

$DfsRecStronglyConnectedComp(i)$

end

Finding the strongly connected components

```
Procedure DfsRecStronglyConnectedComp(i)  
begin  
    time  $\leftarrow$  time + 1  
    color[i]  $\leftarrow$  1  
    for (each vertex j in adiacList(i)) do  
        if (color[j] = 0) then  
            parent[j]  $\leftarrow$  i  
            DfsRecStronglyConnectedComp(j)  
    time  $\leftarrow$  time + 1  
    finalTime[i]  $\leftarrow$  time  
end
```

Finding the strongly connected components

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

Procedure *StronglyConnectedComp*(D)

begin

1. *DFSStronglyConnectedComp*(D)
2. compute D^T
3. *DFSStronglyConnectedComp*(D^T) but considering in the *for* main loop the vertices in descending order of their final times of visiting *finalTime*[i]
4. return each tree computed at step 3 as being a distinct strongly connected component

end

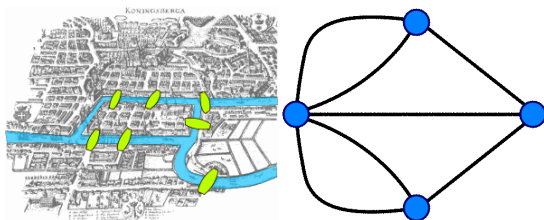
Finding the strongly connected components: complexity

- ▶ $DFSStronglyConnectedComp(D)$: $O(n + m)$
- ▶ compute D^T : $O(m)$
- ▶ $DFSStronglyConnectedComp(D^T)$: $O(n + m)$
- ▶ Total: $O(n + m)$

- ▶ Algorithms, path problems, computer networks (routing), genomics (alignment networks, genome assembly), multi-relational data mining, operations research (scheduling), artificial intelligence (constraint satisfaction), etc.

Applications

The **Konigsberg Bridge Problem** (1736): starting from one land masses, walk over each of the seven bridges just once

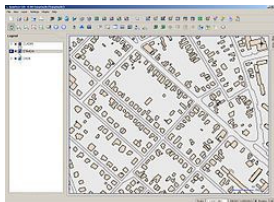


The land masses: vertices, the bridges: edges

It is possible to choose a vertex, proceed along the edges and return to the chosen vertex, covering each edge once?

Applications

- ▶ Google search engine: PageRank algorithm - to determine how important a given web page is
- ▶ Geographic Information Systems (GIS): Google Maps, Bing Maps



- ▶ Social networks

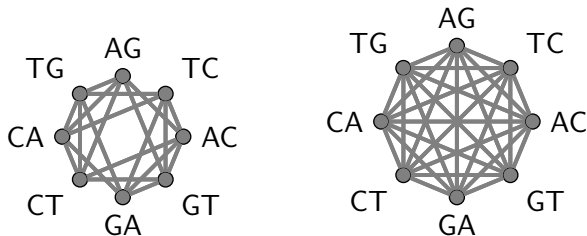


- ▶ The design of DNA codes that satisfy combinatorial constraints; use in biomolecular computing to store information, or to manipulate molecules in chemical libraries
- ▶ Find the largest set S of strings of length n over the alphabet $\{A, C, G, T\}$ s.t.:
 - ▶ *GC Content Constraint*: each word has 50% symbols from $\{C, G\}$
 - ▶ *Hamming Distance Constraint*: each pair of words, $w_1 \neq w_2$ differ in at least d positions: $H(w_1, w_2) \geq d$
 - ▶ *Reverse Complement Hamming Distance Constraint*:
 $H(R(w_1), C(w_2)) \geq d$; $R(w)$: the reversed of word w and $C(w)$ the complement of w ($C \leftrightarrow G, A \leftrightarrow T$)

Graph modeling

- ▶ every DNA word has an assigned vertex v_i
- ▶ $E = E_{HD} \cup E_{RC}$ (E_{HD} pairs of words with a HD conflict, E_{RC} pairs of words with a RC conflict)
- ▶ a solution: a maximum independent set

Figura: Graphs for words of size 2 and Hamming distance $d = 2$ (left) and $d = 3$ (right)



Solution of 136 words for $n = 8$, $d = 4$ instance

| | | | | | |
|----------|----------|----------|----------|----------|----------|
| AAACCACC | ACCAGTGT | ACCCAAGA | ACGTAGTG | ACTGACGT | AGGAAGCT |
| AGTCCTCT | AGTTGGCA | ATCCCGTT | ATGGGCTT | CAAACCTC | CAAGAGAC |
| CAAGCAGT | CACAGTTG | CACCAATC | CAGATGGT | CAGGATCT | CATCGTGT |
| CATGACTG | CATTGCTT | 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 |
| TGTTGCTC | TTAAGGGC | TTACCAGG | TTAGTCCC | TTCAACGG | TTCCTTGC |
| TTCGCCAT | TTCGGGTA | TTCTGACC | TTGACTCC | TTGCCCTA | TTGCGGAT |
| TTGTTGGG | TTTCAGCC | TTTGGTGG | TTTTCCCC | | |