Graphs (Γράφοι)

Κ08 Δομές Δεδομένων και Τεχνικές Προγραμματισμού
Κώστας Χατζηκοκολάκης
Graphs

• **Graphs** are collections of nodes in which various pairs are connected by line segments. The nodes are usually called *vertices* (κορυφές) and the line segments *edges* (ακμές).

• Graphs are **more general than trees**. Graphs are allowed to have cycles and can have more than one connected component.

• Some authors use the terms *nodes* (κόμβοι) and *arcs* (τόξα) instead of vertices and edges.
Example of Graphs (Directed)
Example of Graphs (Undirected)
Examples of Graphs

• Transportation networks

• **Interesting problem**: What is the path with one or more stops of shortest overall distance connecting a starting city and a destination city?
Examples

• A network of oil pipelines

• **Interesting problem:** What is the maximum possible overall flow of oil from the source to the destination?
Examples

- The Internet

- **Interesting problem**: Deliver an e-mail from user A to user B
Examples

• The Web

• Interesting problem: What is the PageRank of a Web site?
Examples

• The Facebook social network

• Interesting problem: Are John and Mary connected? What interesting clusters exist?
Formal Definitions

- A **graph** $G = (V, E)$ consists of a set of **vertices** $V$ and a set of **edges** $E$, where the edges in $E$ are formed from pairs of **distinct** vertices in $V$.

- If the edges have directions then we have a **directed graph** (κατευθυνόμενο γράφο) or **digraph**. In this case edges are ordered pairs of vertices e.g., $(u, v)$ and are called **directed**. If $(u, v)$ is a directed edge then $u$ is called its **origin** and $v$ is called its **destination**.

- If the edges do not have directions then we have an **undirected graph** (μη-κατευθυνόμενος γράφο). In this case edges are unordered pairs of vertices e.g., $\{u, v\}$ and are called **undirected**.

- For simplicity, we will use the directed pair notation noting that in the undirected case $(u, v)$ is the same as $(v, u)$.

- When we say simply graph, we will mean an undirected graph.
Example of a Directed Graph

\[ G = (V, E) \]
\[ V = 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11 \]
\[ E = (1, 2), (1, 3), (2, 5), (3, 4), (5, 4), (5, 6), (6, 7), (8, 9), (8, 10), (10, 11) \]
Example of an Undirected Graph

\[ G = (V, E) \]
\[ V = 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11 \]
\[ E = (1, 2), (1, 3), (2, 5), (3, 4), (5, 4), (5, 6), (6, 70, (8, 9), (8, 10), (10, 11) \]
More Definitions

• Two different vertices \( v_i, v_j \) in a graph \( G = (V, E) \) are said to be adjacent (γειτονικές) if there exists an edge \((v_i, v_j) \in E\).

• An edge is said to be incident (προσπίπτουσα) on a vertex if the vertex is one of the edge's endpoints.

• A path (μονοπάτι) \( p \) in a graph \( G = (V, E) \), is a sequence of vertices of \( V \) of the form \( p = v_1v_2 \ldots v_n \), \((n \geq 2)\) in which each vertex \( v_i \), is adjacent to the next one \( v_{i+1} \) (for \( 1 \leq i \leq n - 1 \)).

• The length of a path is the number of edges in it.

• A path is simple if each vertex in the path is distinct.

• A cycle is a path \( p = v_1v_2 \ldots v_n \) of length greater than one that begins and ends at the same vertex (i.e., \( v_1 = v_n \)).
Definitions

- A **directed path** is a path such that all edges are directed and are traversed along their direction.

- A **directed cycle** is similarly defined.
Definitions

- A **simple cycle** is a path that travels through three or more **distinct** vertices and connects them into a loop.
Four simple cycles: (1,2,3,1) (4,5,6,7,4) (4,5,6,4) (4,6,7,4)
Example

Two non-simple cycles: (1,2,1) (4,5,6,4,7,6,4)
Example

A path that is not a cycle: (1, 2, 4, 6, 8)
Connectivity and Components

• Two vertices in a graph \( G = (V, E) \) are said to be connected (συνδεδεμένες) if there is a path from the first to the second in \( G \).

• Formally, if \( x \in V \) and \( y \in V \), where \( x \neq y \), then \( x \) and \( y \) are connected if there exists a path \( p = v_1 v_2 \ldots v_n \in G \) in such that \( x = v_1 \) and \( y = v_n \).
Connectivity and Components

- In the graph $G = (V, E)$, a **connected component** (συνεκτική συνιστώσα) is a subset $S$ of the vertices $V$ that are all connected to one another.

- A connected component $S$ of $G$ is a **maximal connected component** (μέγιστη συνεκτική συνιστώσα) provided there is no bigger subset $T$ of vertices in $V$ such that $T$ properly contains $S$ and such that $T$ itself is a connected component of $G$.

- An undirected graph $G$ can always be separated into maximal connected components $S_1, S_2, \ldots , S_n$ such that $S_i \cap S_j = \emptyset$ whenever $i \neq j$. 
Example of Undirected Graph and its Separation into Two Maximal Connected Components
Connectivity and Components in Directed Graphs

- A subset \( S \) of vertices in a directed graph \( G \) is **strongly connected** (ισχυρά συνεκτικό) if for each pair of distinct vertices \((v_i, v_j)\) in \( S \), \( v_i \) is connected to \( v_j \) and \( v_j \) is connected to \( v_i \).

- A subset \( S \) of vertices in a directed graph \( G \) is **weakly connected** (ασθενώς συνεκτικό) if for each pair of distinct vertices \((v_i, v_j)\) in \( S \), \( v_i \) is connected to \( v_j \) or \( v_j \) is connected to \( v_i \).
Example: A Strongly Connected Digraph
Example: A Weakly Connected Digraph
Degree in Undirected Graphs

• In an undirected graph $G$ the degree (βαθμός) of vertex $x$ is the number of edges $e$ in which $x$ is one of the endpoints of $e$.

• The degree of a vertex $x$ is denoted by $\text{deg}(x)$. 
The degree of node 1 is 2.
The degree of node 4 is 4.
The degree of node 8 is 1.
Predecessors and Successors in Directed Graphs

• If \( x \) is a vertex in a directed graph \( G = (V, E) \) then the set of predecessors (προηγούμενων) of \( x \) denoted by \( \text{Pred}(x) \) is the set of all vertices \( y \in V \) such that \( (y, x) \in E \).

• Similarly the set of successors (επόμενων) of \( x \) denoted by \( \text{Succ}(x) \) is the set of all vertices \( y \in V \) such that \( (x, y) \in E \).
In-Degree and Out-Degree in Directed Graphs

• The **in-degree** of a vertex $x$ is the number of predecessors of $x$

• The **out-degree** of a vertex $x$ is the number of successors of $x$

• We can also define the in-degree and the out-degree by referring to the **incoming** and **outgoing** edges of a vertex.

• The in-degree and out-degree of a vertex $x$ are denoted by $\text{indeg}(x)$ and $\text{outdeg}(x)$ respectively.
Example

The in-degree of node 4 is 2. The out-degree of node 4 is 1.
Proposition

• If $G$ is an undirected graph with $m$ edges, then

$$\sum_{v \in G} \deg(v) = m$$

• Proof?
  - Each edge is counted twice
Proposition

- If is a directed graph with edges, then
  \[ \sum_{v \in G} \text{indeg}(v) = \sum_{v \in G} \text{outdeg}(v) = m \]

- Proof?
  - Each edge is counted once
Proposition

• Let $G$ be a graph with $n$ vertices and $m$ edges. If $G$ is undirected, then $m \leq \frac{n(n-1)}{2}$ and if $G$ is directed, then $m \leq n(n - 1)$.

• Proof?
  - If $G$ is undirected then the maximum degree of a vertex is $n - 1$. Therefore, from the previous proposition about the sum of the degrees, we have $2m \leq n(n - 1)$.
  - If $G$ is directed then the maximum in-degree of a vertex is $n - 1$. Therefore, from the previous proposition about the sum of the in-degrees, we have $m \leq n(n - 1)$. 
More definitions

• A subgraph (υπογράφος) of a graph $G$ is a graph $H$ whose vertices and edges are subsets of the vertices and edges of $G$ respectively.

• A spanning subgraph (υπογράφος επικάλυψης) of $G$ is a subgraph of $G$ that contains all the vertices of $G$.

• A forest (δάσος) is a graph without cycles.

• A free tree (ελεύθερο δένδρο) is a connected forest i.e., a connected graph without cycles. The trees that we studied in earlier lectures are rooted trees (δένδρα με ρίζα) and they are different than free trees.

• A spanning tree (δένδρο επικάλυψης) of a graph is a spanning subgraph that is a free tree.
The thick green lines define a spanning tree of the graph.
Example

The thick green lines define a forest which consists of two free trees.
Graph Representations: Adjacency Matrices

• Let $G = (V, E)$ be a graph. Suppose we number the vertices in $V$ as $v_1, v_2 \ldots v_n$

• The adjacency matrix (πίνακας γειτνίασης) corresponding to $G$ is an $n \times n$ matrix such that $T[i, j] = 1$ if there is an edge $(v_i, v_j) \in E$, and $T[i, j] = 0$ if there is no such edge in $E$. 
Example

A graph $G$

The adjacency matrix for graph $G$

\[
\begin{array}{cccc}
1 & 2 & 3 & 4 \\
1 & 0 & 1 & 0 & 0 \\
2 & 0 & 0 & 1 & 1 \\
3 & 1 & 0 & 0 & 1 \\
4 & 1 & 0 & 0 & 0 \\
\end{array}
\]
Adjacency Matrices

- The adjacency matrix of an **undirected graph** $G$ is a **symmetric matrix** i.e., $T[i, j] = T[j, i]$ for all and in the range $1 \leq i, j \leq n$

- The adjacency matrix for a **directed graph** need not be symmetric.
Adjacency Matrices

• The **diagonal entries** in an adjacency matrix (of a directed or undirected graph) **are zero**, since graphs as we have defined them are not permitted to have looping self-referential edges that connect a vertex to itself.
Example

An undirected graph G

The adjacency matrix for graph G:

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
Adjacency Sets

- Another way to define a graph $G = (V, E)$ is to specify **adjacency sets** (σύνολα γειτνίασης) for each vertex in $V$.

- Let $V_x$ stand for the set of all vertices **adjacent** to $x$ in an undirected graph $G$ or the set of all vertices that are **successors** of $x$ in a directed graph $G$.

- If we give both the vertex set $V$ and the collection $A = \{V_x | x \in V\}$ of adjacency sets for each vertex in then we have given enough information to define the graph $G$. 
Graph Representations: Adjacency Lists

- Another family of representations for a graph uses adjacency lists (λίστες γειτνίασης) to represent the adjacency set $V_x$ for each vertex $x$ in the graph.
Example Directed Graph

A directed graph $G$

<table>
<thead>
<tr>
<th>Vertex Number</th>
<th>Out Degree</th>
<th>Adjacency list</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>2 3</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>3 4 5</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

The **sequential** adjacency lists for graph $G$. Notice that vertices are listed in their **natural order**.
Example Directed Graph

A directed graph $G$

The **linked** adjacency lists for graph $G$. Notice that vertices in a list are organized according to their **natural order**.
Example Undirected Graph

An undirected graph $G$

<table>
<thead>
<tr>
<th>Vertex Number</th>
<th>Degree</th>
<th>Adjacency list</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3</td>
<td>2 3 5</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>1 3 4 5</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>1 2 4</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>2 4</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td>1 2</td>
</tr>
</tbody>
</table>

The sequential adjacency lists for graph $G$
Graph Searching

• To search a graph $G$, we need to visit all vertices of $G$ in some systematic order.

• Each vertex $v$ can be a structure with a `bool` valued member `visited` which is initially `false` for all vertices of $G$. When we visit $v$, we will set it to `true`. 
An Algorithm for Graph Searching

// Ψευδοκώδικας, επίσκεψη όλων των κόμβων του γράφου

void graph_search(G) {
    Let G = (V,E) be a graph
    Let C be an empty container

    for (each vertex x in V) {
        x.visited = false;
    }
    Insert v into C;

    while (C is non-empty) {
        Remove a vertex x from container C;
        if (!x.visited) {
            visit(x);
            x.visited = true;
            for (each vertex w adjacent to x) {
                if (!w.visited))
                    Insert w into C;
            }
        }
    }
}
Graph Searching

Interesting case: the container $C$ is a stack.

In what order vertices are visited?
Graph Searching

Eg. the container $C$ is a stack.

The vertices are visited in the order 1, 4, 8, 7, 3, 2, 6, 5.
Depth-First Search (DFS)

• When $C$ is a stack, the tree in the previous example is searched in depth-first order.

• Depth-first search (αναζήτηση πρώτα κατά βάθος) at a vertex always goes down (by visiting unvisited children) before going across (by visiting unvisited brothers and sisters).

• Depth-first search of a graph is analogous to a pre-order traversal of an ordered tree.
Graph Searching

Another interesting case: the container $C$ is a queue.

What is the order vertices are visited?
Graph Searching

Another interesting case: the container $C$ is a queue.

The vertices are visited in the order 1, 2, 3, 4, 5, 6, 7 and 8.
Breadth-First Search (BFS)

• When $C$ is a queue, the tree in the previous example is searched in breadth-first order.

• Breadth-first search (αναζήτηση πρώτα κατά πλάτος) at a vertex always goes broad before going deep.

• Breadth-first traversal of a graph is analogous to a traversal of an ordered tree that visits the nodes of the tree in level-order.

• BFS subdivides the vertices of a graph in levels. The starting vertex is at level 0, then we have the vertices adjacent to the starting vertex at level 1, then the vertices adjacent to these vertices at level 2 etc.
What is the order of visiting vertices for DFS?
Example

Depth-first search visits the vertices in the order 1, 4, 8, 6, 5, 7, 3 and 2
Example

What is the order of visit for BFS?
Example

Breadth-first search visits the vertices in the order 1, 2, 3, 4, 5, 6, 7 and 8.
Exhaustive Search

• Either the stack version or the queue version of the algorithm GraphSearch will visit every vertex in a graph $G$ provided that $G$ consists of a single strongly connected component.

• If this is not the case, then we can enumerate all the vertices of $G$ and run GraphSearch starting from each one of them in order to visit all the vertices of $G$. 
Exhaustive Search

```c
void graph_exhaustive_search(G) {
    Let G = (V,E) be a graph.
    for (each vertex v in G) {
        graph_search(G, v)
    }
}
```
Recursive DFS

- DFS can be also written recursively
- The stack is essentially replaced by the **function call stack**
Recursive DFS

```plaintext
// Ψευδοκώδικας, επίσκεψη όλων των κόμβων του γράφου

void graph_dfs(G) {
    for (each vertex x in V) {
        x.visited = false;
    }
    for (each vertex x in V) {
        if (!x.visited)
            traverse(G, x);
    }
}

void traverse(G, x) {
    visit(x);
    x.visited = true;
    for (each vertex w adjacent to v) {
        if (!w.visited)
            traverse(G, w);
    }
}
```
Example of Recursive DFS

What is the order vertices are visited?
Example

The vertices are visited in the order 1, 2, 5, 6, 3, 4, 7 and 8. This is different than the order we got when using a stack!
Complexity of DFS

• DFS as implemented above (with adjacency lists) on a graph with \( e \) edges and \( n \) vertices has complexity \( O(n + e) \).

• To see why observe that on no vertex is \texttt{traverse} called more than once, because as soon as we call \texttt{traverse} with parameter \( x \), we mark \( x \) visited and we never call \texttt{traverse} on a vertex that has previously been marked as visited.

• Thus, the total time spent going down the adjacency lists is proportional to the lengths of those lists, that is \( O(e) \)

• The initialization steps in \texttt{graph_dfs} have complexity \( O(n) \)

• Thus, the total complexity is \( O(n + e) \)
Complexity of DFS

• If DFS is implemented using an adjacency matrix, then its complexity will be $O(n^2)$.

• If the graph is dense (πυκνός), that is, it has close to $O(n^2)$ edges the difference of the two implementations is minor as they would both run in $O(n^2)$ time.

• If the graph is sparse (αραιός), that is, it has close to $O(n)$ edges, then the adjacency matrix approach would be much slower than the adjacency list approach.
Complexity of BFS

- BFS with adjacency lists has the same complexity as DFS i.e., $O(n + e)$. 
Readings

• T. A. Standish. *Data Structures, Algorithms and Software Principles in C.* Chapter 10


• A. V. Aho, J. E. Hopcroft and J. D. Ullman. *Data Structures and Algorithms.* Chapters 6 and 7
