Fundamental Algorithms

Chapter 8: Graphs

Jan Křetínský

Winter 2017/18
Graphs

Definition (Graph)

A graph $G = (V, E)$ consists of a set $V$ of vertices (nodes) and a set $E$ of edges between the vertices.

- **undirected graph**: $(i, j) \in E$ an unordered pair – $(i, j) = (j, i)$
- **directed graph** (or shorter: “digraph”): $(i, j) \in E$ an ordered tuple, i.e. $(i, j) \in E$ independent of $(j, i) \in E$
Graphs

Definition (Graph)

A graph $G = (V, E)$ consists of a set $V$ of vertices (nodes) and a set $E$ of edges between the vertices.

- **undirected graph**: $(i, j) \in E$ an unordered pair -- $(i, j) = (j, i)$
- **directed graph** (or shorter: “digraph”):
  $(i, j) \in E$ an ordered tuple, i.e. $(i, j) \in E$ independent of $(j, i) \in E$

Some Terms

- two vertices $V_0$ and $V_n$ are connected by a path (of length $n$), if there is a sequence of edges $(V_0, V_1), (V_1, V_2), \ldots, (V_{n-1}, V_n)$
- a graph is **connected**, if there is a path between any two vertices
- a vertex $V$ has **degree** $d$, if $V$ has $d$ (outgoing) edges
Graphs in CSE – Unstructured Grids:

- in blue: $V =$ grid cells, $E =$ neighbours (“dual graph”)
- in black: $V =$ grid vertices, $E =$ cell edges
Trees

**Definition (Tree)**

A *tree* is a connected graph without cycles.

---

**Theorem**

A graph $T$ is a *tree* if and only if there is a unique path between any two distinct vertices of $T$.

**Implications:**

- There is only one connection from the root to any of the nodes.
- Any path between two nodes will run through the root of the respective subtree.
- Actually: which node is the "root"?
Trees

Definition (Tree)

A tree is a connected graph without cycles.

→ Question: is this consistent with our “naive” image of a tree?
Trees

**Definition (Tree)**

A *tree* is a connected graph without cycles.

→ *Question: is this consistent with our “naive” image of a tree?*

**Theorem**

A graph \( T \) is a *tree*, if and only if there is a unique path between any two distinct vertices of \( T \).
Trees

Definition (Tree)
A tree is a connected graph without cycles.

→ Question: is this consistent with our “naive” image of a tree?

Theorem
A graph $T$ is a tree, if and only if there is a unique path between any two distinct vertices of $T$.

Implications:
- there is only one connection from the root to any of the nodes
- any path between two nodes will run through the root of the resp. subtree
- actually: which node is the “root”?
Trees (2)

Theorem

A connected graph \((V, E)\) is a tree, if and only if \(|E| = |V| - 1\)
Trees (2)

Theorem

A connected graph \((V, E)\) is a tree, if and only if \(|E| = |V| - 1\)

Implications:

- if you “cut” one edge, a tree is no longer connected (child becomes an orphan)
- building a tree incrementally requires a root (one node, no edge) and one additional edge per added node
Trees (2)

**Theorem**

*A connected graph \((V, E)\) is a tree, if and only if \(|E| = |V| - 1\)*

**Implications:**

- if you “cut” one edge, a tree is no longer connected (child becomes an orphan)
- building a tree incrementally requires a root (one node, no edge) and one additional edge per added node

**Definition (Spanning Tree)**

\(T = (V, E)\) is called a *spanning tree* for the graph \(G = (V, E')\), if \(T\) is a tree, and \(E \subset E'\).

*Note: \(T\) has the same vertices as \(G\).*
Data Structures for Graphs

**Pointer-Based Data Structure:** (esp. for directed graphs)

```plaintext
Node := ( 
    key: Integer,  
    edges: List of Node );
```
Data Structures for Graphs

**Pointer-Based Data Structure:** (esp. for directed graphs)

\[
\text{Node} := (\text{key : Integer}, \text{edges : List of Node});
\]

**Adjacency Matrix:**

- \( n \times n \) matrix \( A \), where \( n = |V| \)
- \( a_{ij} = 1 \), if \((i, j) \in E\)
- \( A \) is symmetric for undirected graphs
Data Structures for Graphs

**Pointer-Based Data Structure:** (esp. for directed graphs)

```
Node := (  
    key: Integer,  
    edges: List of Node );
```

**Adjacency Matrix:**
- $n \times n$ matrix $A$, where $n = |V|$
- $a_{ij} = 1$, if $(i, j) \in E$
- $A$ is symmetric for undirected graphs

*Note: to store an adjacency matrix as an $n \times n$ array is a good idea, only if $|E| \in \Theta(n^2)$*
Graph Traversals

Definition (Graph Traversal:)

Input: a (connected!) directed or undirected graph \((V, E)\), and a node \(x \in V\).

Task: Starting from \(x\), “visit” all vertices in \(V\) (following edges only)

Examples:

• modify the key values of all vertices
• search a specific key value in a graph

Two main variants:

• depth-first traversal (depth-first search)
• breadth-first traversal (breadth-first search)
Graph Traversals

Definition (Graph Traversal:)

Input: a (connected!) directed or undirected graph \((V, E)\), and a node \(x \in V\).

Task: Starting from \(x\), “visit” all vertices in \(V\) (following edges only)

Examples:

- modify the key values of all vertices
- search a specific key value in a graph
Graph Traversals

Definition (Graph Traversal:)

**Input:** a (connected!) directed or undirected graph \((V, E)\), and a node \(x \in V\).

**Task:** Starting from \(x\), “visit” all vertices in \(V\) (following edges only)

**Examples:**
- modify the key values of all vertices
- search a specific key value in a graph

**Two main variants:**
- depth-first traversal (depth-first search)
- breadth-first traversal (breadth-first search)
Depth-First Traversal

DFTraversal(\(V:\text{Node}\) ) {
    ! mark current node \(V\) as visited:
    Mark[\(V.\text{key}\)] = 1;
    ! perform desired work on \(V\):
    Visit(\(V\));
    ! perform traversal from all nodes connected to \(V\)
    \textbf{forall} (\(V,W\)) in \(V.\text{edges}\) \textbf{do}
        \textbf{if} Mark[\(W.\text{key}\)] = 0 \textbf{then} DFTraversal(\(W\));
    \textbf{end do};
}

Assumptions:

- keys \(V.\text{key}\) numbered from 1, \ldots, \(n = |V|\)
- Mark : \textbf{Array}[1..n]
- \textbf{forall} loop executed sequentially
DF-Traversal – Stack-Based Implementation

StackDFTrav(X:Node) {
  ! uses stack of "active" nodes
  Stack active = { X }; Mark[X.key] = 1;
  while active <> {} do
    ! remove first node from stack
    V = pop(active);
    Visit(V);
    forall (V,W) in V.edges do
      if Mark[W] = 0 then {
        push(active, W); Mark[W.key] = 1;
      }
    end do;
  end while;
}

→ use stack as last-in-first-out (LIFO) data container
Breadth-First-Traversal
Queue-Based Implementation

BFTraversal(X:Node) {

! uses queue of "active" nodes
Queue active = { X }; Mark[X.key] = 1;
while active <> {} do

! remove first node from queue
V = remove(active);
Visit(V);
forall (V,W) in V.edges do

if Mark[W.key] = 0 then {
append(active, W); Mark[W.key] = 1;
}
end do;
end while;
}

→ use queue as first-in-first-out (FIFO) data container
Breadth-First Search

BFSearch(x:Node, k:Integer) : Node {
    Queue active = { x };
    while active <> {} do
        V = remove(active);
        if V.key = k then return V;
        if Mark[V.key] = 0 then
            Mark[V.key] = 1
            forall (V,W) in V.edges do
                append(active, W);
            end do;
        end if;
    end while;
}
Breadth-First Search

BFSearch(x:Node, k:Integer) : Node {
    Queue active = { x };
    while active <> {} do
        V = remove(active);
        if V.key = k then return V;
        if Mark[V.key] = 0 then
            Mark[V.key] = 1
            for all (V,W) in V.edges do
                append(active, W);
            end do;
        end if;
    end while;
}

Breadth-First Search as Shortest-Path Algorithm:
• breadth-first search will return the node with the shortest path from x
• requires modification of algorithm to return this path, as well
Breadth-First and Depth-First Traversal

DF/BF-Traversal and Connectivity of Graphs:

- DF- and BF-traversal will visit all nodes of a connected graph
- if a non-connected graph is traversed, both algorithms can be used to find the (maximum) connected sub-graph that contains the start node
- hence, DF- and BF-traversal can be extended to find all connectivity components of a graph
Breadth-First and Depth-First Traversal

DF/BF-Traversal and Connectivity of Graphs:
- DF- and BF-traversal will visit all nodes of a connected graph
- if a non-connected graph is traversed, both algorithms can be used to find the (maximum) connected sub-graph that contains the start node
- hence, DF- and BF-traversal can be extended to find all connectivity components of a graph

DF/BF-Traversal and Trees:
- DF- and BF-traversal will compute a spanning tree of a connected graph
- BF-traversal generates a spanning tree with shortest paths to the root