Algorithms and Data Structures

Graphs (1)

Marcin Sydow

Web Mining Lab
PJWSTK
Topics covered by this lecture:

- Graphs - Reminder
- Trees
- Visiting Trees (in-order, post-order, pre-order)
- Graph Traversals (BFS, DFS)
Graph Definitions: Reminder

- Directed graph (digraph): \( G = (V, E) \), \( V \) - vertex (node) set, \( E \subseteq V \times V \) - arc set (each arc is an ordered pair \( (u, v) \), where \( u, v \in V \). \( u \) - source, \( v \) - target of the arc)

- Undirected graph - the only difference is that edge (undirected arc) \( (u, v) \) is an un-ordered pair, \( u, v \in V \).

- Another variant: bidirected graphs.

- Self-loops usually not allowed. If multiple arcs(edges) possible - multi-graph. Generalisation: arc are not pairs, but n-tuples - hypergraph.

- \( e = (u, v) \) is incident to \( u, v \), and \( u, v \) are adjacent.

- In-degree, out-degree of a node (directed); degree (undirected)

- Sum of degrees is always even
Graph Definitions: Reminder (2)

- Subgraph $G' \subseteq G$.
- Subgraph $G'$ induced by a subset of nodes $V' \subseteq V$: $G' = (V', E \cap (V' \times V'))$
- Weights on arcs/edges ($w : E \rightarrow R$)
- Path $(v_0, \ldots, v_k)$ of length $k$. Simple path: nodes do not repeat.
- Cycle (path: $v_0 \equiv= v_k$). Hamiltonian cycle, Euler Cycle.
- Connected graph (there is a path between any pair of nodes), weakly connected: path can ignore the arc direction
- strongly connected component (SCC): a maximum strongly connected subgraph
- SCCs partition $V$ (no intersections)
Number of edges

- if $|V| == n$ and $|E| = m$ then $m = O(n^2)$
- by graph size we usually mean $m + n$
- if $m = o(n^2)$ the graph is called sparse
- full graph (has all possible edges): maximum number of edges
  - Undirected full graph has exactly $(n - 1)n/2$ edges
- empty graph - no edges (only nodes)
- how many edges a connected graph must have at least?
Number of edges

- if $|V| == n$ and $|E| = m$ then $m = O(n^2)$
- by graph size we usually mean $m + n$
- if $m = o(n^2)$ the graph is called sparse
- full graph (has all possible edges): maximum number of edges
- Undirected full graph has exactly $(n - 1)n/2$ edges
- empty graph - no edges (only nodes)
- how many edges a connected graph must have at least? $n - 1$
Trees: Reminder

- Undirected tree: a connected graph without cycles (acyclic)
- undirected tree $\Leftrightarrow$ connected and has exactly $(n - 1)$ edges
- leaf, interior node
- Forest: an acyclic graph (not necessarily connected)
- rooted tree: ancestor, descendant, child, sibling, subtree,
- height (of tree or node): maximum distance to a leaf,
- depth (of node) distance to the root
- ordered tree (rooted tree with order on children)
- binary tree (ordered tree with max of 2 children of each node)
- DAG: Directed Acyclic Graph
Various graph representations are possible. The choice depends on which \textit{operations} should be fast and how much memory is available.

The most important operations on graphs:

- node/edge information access (weights, existence, etc.)
- navigation (typically: list of all outgoing arcs/edges)
- update (adding/removing nodes or edges/arcs)
- input/construction/conversion/output
Computer Representations of Graphs

- unordered sequence of edges (fast update, good as input/output format)
- adjacency matrices (extremely fast access, much memory, very slow extension; can be adapted to sparse graphs)
- adjacency arrays (good for static graphs)
- adjacency lists (least memory, easy update, relatively easy navigation)

Except few cases (which?), translation from one representation to another is linear (fast).
Many interesting connections between linear algebra and graphs, for example:

$A$ - adjacency matrix: $A_{i,j}^k = \text{how many paths from } i \text{ to } j \text{ of length exactly } k$

Algebraic Graph Theory: studies such connections between matrices and graphs, etc.
A **rooted tree** - some node is distinguished and is called **root**. 
(On picture, the root is at the top). Case: complete tree.

A binary tree is a rooted tree, and each node has maximum of 2 nodes, which are distinguishable (left and right).

A binary tree can be represented as a linked structure:
- each node has links to its children
- the only access to the whole tree is a pointer to the root
Traversing Trees

A general scheme:

\[
\text{traverse}(v) : \\
\quad \text{previsit}(v) \\
\quad \text{for each child } w \text{ of } v: \text{traverse}(w) \\
\quad \text{postvisit}(v)
\]

If postvisit is empty we call it pre-order, if previsit is empty – post-order.

Post-order can be used to compute height, pre-order for computing depth in trees.
Visiting Binary Trees

In a special case of binary tree we have 3 important variants:
- in-order
- pre-order
- post-order
in-order order

```java
inorderVisit(BinTree currentNode){
    if currentNode == null return
    inorderVisit(currentNode.left)
    visit(currentNode)
    inorderVisit(currentNode.right)
}
```
pre-order order

```java
preorderVisit(BinTree currentNode){
    if currentNode == null return

    visit(currentNode)
    preorderVisit(currentNode.left)
    preorderVisit(currentNode.right)
}
```
post-order order

```java
postorderVisit(BinTree currentNode){
    if currentNode == null return

    postorderVisit(currentNode.left)
    postorderVisit(currentNode.right)
    visit(currentNode)
}
```
Example: expression trees

Evaluate an expression: \(2,+,3,/,6\)

A *parser* first transforms it to an **expression tree**. The root is the “last” operator, numbers are in leaves, interior nodes are the other operators.

Now, the evaluation is very easy:

\[
eval(r):
\begin{align*}
    &\text{if } \text{isLeaf}(r) \text{ return } \text{number}(r) \\
    &a = eval(\text{leftChild}(r)) \\
    &b = eval(\text{rightChild}(r)) \\
    &\text{return } a \text{ operator}(r) b
\end{align*}
\]
Example: how to avoid recursion with a stack?

**In-order in binary trees:**

```python
stack = empty; v = root

1:
if (v.left != null):
    stack.push(v)
    v = v.left
    goto 1

2:
visit(v)
if (v.right != null):
    v = v.right
    goto 1

if (!stack.empty())
    v = stack.pop()
    goto 2
```
Graph Traversal: a General Scheme

Systematic traverse through the whole graph: start visiting from a single node and moving along edges from already visited nodes, visit each node and edge available from s exactly once

In each iteration: select next already visited node and visit all its outgoing, non-visited edges, and non-visited end-nodes

In the above general scheme, by specifying the way of selecting the next visited node we obtain various refinements of the algorithm
BFS and DFS - Important Variants of Graph Traversal

Two particularly important graph traversals:

- **BFS** (breadth-first search) (next nodes to visit are kept on queue)
- **DFS** (depth-first search) (next nodes to visit are kept on stack)

Both produce resulting forest and (as a side product) classify each edge into one of four categories:

- **tree (T)** edge (edge of the resulting forest)
- **forward (F)** edge (in the same branch of the forest)
- **backward (B)** (as above but counter-directed)
- **cross (C)** (between two different branches or trees)
Breadth-first search (BFS)

graph G<V,E>; s - start node; d - distances, p - parents

for-each node in V:
    node.color = white; d[u] = infinity; p[u] = null

s.color = gray; d[s] = 0; queue.in(s)

while(!queue.empty()){
    currNode = queue.out()

    process(currNode)

    for-each node in currNode.adjList:
        if (node.color == white):
            queue.in(node)
            node.color = gray
            d[node] = d[currNode] + 1
            p[node] = currNode

    currNode.color = black
}

white - untouched; gray - waiting for processing; black - processed
Properties of BFS

- O(m + n) (dom. op.: set or update distance)
- the resulting tree (recorded in the parent array) specifies the shortest paths from s to other nodes
Depth-first search (DFS) – a Recursive Version

\[d - \text{first visit times;} \ f - \text{finishing times}\]

DFS()
\[
\text{time} = 0
\]
\[
\text{for-each } v \text{ in } V:
\]
\[
\quad v.\text{color} = \text{white}; \ \text{parent}[v] = \text{null}
\]
\[
\text{for-each } v \text{ in } V:
\]
\[
\quad \text{if } (v.\text{color} == \text{white}):
\]
\[
\quad \quad \text{recursiveDFS}(v)
\]
}

recursiveDFS(GraphNode v){
\[
\quad v.\text{color} = \text{gray}
\]
\[
\quad \text{process}(v)
\]
\[
\quad d[v] = \text{time}++
\]
\[
\quad \text{for-each } u \text{ in } v.\text{adjList}:
\]
\[
\quad \quad \text{if } (u.\text{color} == \text{white}):
\]
\[
\quad \quad \quad \text{parent}[u] = v
\]
\[
\quad \quad \quad \text{recursiveDFS}(u)
\]
\[
\quad u.\text{color} = \text{black}
\]
\[
\quad f[u] = \text{time}++
\]
}

white - before \(d\) is set; gray between \(d\) is set and \(f\) is set; black after \(f\) is set.
Properties of DFS

- O(m + n) time complexity
- DFS can be obtained by modification of BFS - Queue should be replaced by Stack
- for any \( u, v \in V \) either the intervals \((d[u], f[u]), (d[v], f[v])\) are disjoint or one is completely included in the other (so called: “parentheses” structure)
- when DFS first visits an edge \((u, v)\): T if \( v \) white, B if \( v \) gray, F or C if \( v \) black
- undirected DFS: only T or B edges may happen (no C or F)
- DAG DFS: only T may happen (a good test for acyclicity)
Applications of DFS

DFS has many important applications, for example:

- directed: topological sort
- directed: finding SCCs
- undirected: finding BCC (bi-connected components: maximum subsets of edges, so that any 2 edges in BCC lie on a common simple cycle; bridges or articulation points connect different BCCs; bridge: an edge which removed increases the number of SCCs; articulation point - a node with such property)
G - a DAG. “Sort” the nodes so that all the edges point from left to right

application in scheduling: \((V - \text{set of tasks}, (u, v) - \text{task } u \text{ must be done before } v)\)

Topological Sort:

1. compute finishing times \(f[v], v \in V\) with DFS on \(G\)
2. sort decreasingly by \(f[v]\)

Remark: If cycles are present ideal sorting impossible, but it minimises the “backward” edges
Computing SCCs

Application of DFS to compute strongly connected components:

1. compute finishing times $f[v], v \in V$ with DFS on $G$
2. “reverse” the arcs in $G$ (transposed adjacency matrix)
3. run DFS on the reversed graph; apply decreasing order of $f[v]$ in the main loop of DFS
4. result: separate trees $==$ separate SCCs
Questions/Problems:

- all basic graph definitions
- trees (definitions)
- graph computer representations (differences/advantages, etc.)
- binary trees and visiting them: in-order, pre-order, post-order
- classification of edges in BFS and DFS
- BFS + properties
- DFS + properties
- compare DFS and BFS
- Topological Sort (high-level idea)
- (*) Other applications of DFS
Thank you for your attention