Algorithms and Data Structures

Graphs: basic concepts and algorithms

(c) Marcin Sydow

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 $V' \subset V$: $G' = (V', E \cap (V' \times V'))$
- Weights on arcs/edges $(w : E \rightarrow R)$
- Path $(v_0, ..., v_k)$ of length k. Simple path: nodes do not repeat.
- Cycle (path: $v_0 == v_k$). 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

How to represent graphs in computer?

Various graph representations are possible. The choice depends on which *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).

Algebraic Graph Theory

Many interesting connections between linear algebra and graphs, for example:

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

Algebraic Graph Theory: studies such connections between matrices and graphs, etc.

Binary Tree

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:

```
traverse(v):
 previsit(v)
 for each child w of v: traverse(w)
 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


```
inorderVisit(BinTree currentNode){
  if currentNode == null return

  inorderVisit(currentNode.left)
  visit(currentNode)
  inorderVisit(currentNode.right)
}
```

pre-order order

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

  visit(currentNode)
  preorderVisit(currentNode.left)
  preorderVisit(currentNode.right)
}
```

post-order order

```
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):
 if isLeaf(r) return number(r)
 a = eval(leftChild(r))
 b = eval(rightChild(r))
 return a operator(r) b
```

Example: how to avoid recursion with a stack?

in-order in binary trees:

```
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 - distance from s, p - parent

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

s.color = gray; s.d = 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
        node.d = currNode.d + 1
        node.p = 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 – discovery time; f – finishing time

```
DFS(){
  time = 0
  for-each v in V:
     v.color = white; v.parent = null
  for-each v in V:
     if (v.color == white):
        recursiveDFS(v)
}
recursiveDFS(GraphNode v){
  v.d = time++
  v.color = gray
  process(v)
  for-each u in v.adjList:
     if (u.color == white):
        u.parent = v
        recursiveDFS(u)
  v.color = black
  v.f = 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 $(u.d, u.f), (v.d, v.f)$ 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)

Topological Sort

- G - a DAG. "Sort" the nodes so that all the edges point from left to right
- application in scheduling: (V – set of tasks, (u, v) – task u must be done before v)

Topological Sort:

1. compute finishing times $v.f, v \in V$ with DFS on G
2. sort decreasingly by $v.f$

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 $v.f, v \in V$ with DFS on G
2. "reverse" the arcs in G (transposed adjacency matrix)
3. run DFS on the reversed graph; apply decreasing order of $v.f$ 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