Graph Traversal
Section 4.1–4.2

Dr. Mayfield and Dr. Lam

Department of Computer Science
James Madison University

Oct 30, 2015
Reminder

“Portfolio 7” due in two weeks

▶ Submit four new problems (seven total)
▶ You may replace/improve the other three

See Port3 feedback in Canvas!

▶ Avoid problems from the first three weeks
▶ Double check formatting, style, spacing, etc.
LaTeX tips

- Use dollar signs for math mode (use $n$ for $n$)
- If you want to say $n^{th}$ occurrence: $n^{\text{th}}$
- Use tilde for abbreviations (e.g., Dr.~Mayfield)
- Add `tabsize=4` to `lstdefinestyle` (or use spaces)
- Use `--` for – (n-dash) and `---` for — (m-dash)
- Curvy quotes are ‘‘different’’ on the left/right
- `lstlisting` style should match the language
- Don’t forget to change the date (on first page)
Section 4.1–4.2

The following slides are by Steven Halim

https://sites.google.com/site/stevenhalim/home/material
Graph Terms – **Quick Review**

- Vertices/Nodes
- Edges
- Un/Weighted
- Un/Directed
- In/Out Degree
- Self-Loop/Multiple Edges
  (Multigraph) vs Simple Graph
- Sparse/Dense
- Path, Cycle
- Isolated, Reachable
- (Strongly) Connected Component
- Sub Graph
- Complete Graph
- Directed Acyclic Graph
- Tree/Forest
- Euler/Hamiltonian Path/Cycle
- Bipartite Graph
Depth-First Search (DFS)
Breadth-First Search (BFS)
  Reachability
  Finding Connected Components
Flood Fill
Topological Sort
Finding Cycles (Back Edges)
Finding Articulation Points & Bridges
Finding Strongly Connected Components

GRAPH TRAVERSAL ALGORITHMS
Graph Traversal Algorithms

• Given a graph, we want to traverse it!
• There are 2 major ways:
  – Depth First Search (DFS)
    • Usually implemented using recursion
    • More natural
    • Most frequently used to traverse a graph
  – Breadth First Search (BFS)
    • Usually implemented using queue (+ map), use STL
    • Can solve special case* of “shortest paths” problem!
Depth First Search – Template

- $O(V + E)$ if using Adjacency List
- $O(V^2)$ if using Adjacency Matrix

```cpp
typedef pair<int, int> ii; typedef vector<ii> vi;

void dfs(int u) { // DFS for normal usage
    printf(" %d", u); // this vertex is visited
    dfs_num[u] = DFS_BLACK; // mark as visited
    for (int j = 0; j < (int)AdjList[u].size; j++) {
        ii v = AdjList[u][j]; // try all neighbors v of vertex u
        if (dfs_num[v.first] == DFS_WHITE) // avoid cycle
            dfs(v.first); // v is a (neighbor, weight) pair
    }
}
```

CS3233 - Competitive Programming,
Steven Halim, SoC, NUS
Breadth First Search (using STL)

• Complexity: also $O(V + E)$ using Adjacency List

```cpp
map<int, int> dist; dist[source] = 0;
queue<int> q; q.push(source); // start from source

while (!q.empty()) {
    int u = q.front(); q.pop(); // queue: layer by layer!
    for (int j = 0; j < (int)AdjList[u].size(); j++) {
        ii v = AdjList[u][j]; // for each neighbours of u
        if (!dist.count(v.first)) {
            dist[v.first] = dist[u] + 1; // unvisited + reachable
            q.push(v.first); // enqueue v.first for next steps
        }
    }
}
```
1st Application: Connected Components

- DFS (and BFS) can find connected components
  - A call of dfs(u) visits only vertices connected to u

```c
int numComp = 0;
dfs_num.assign(V, DFS_WHITE);
REP (i, 0, V - 1) // for each vertex i in [0..V-1]
    if (dfs_num[i] == DFS_WHITE) { // if not visited yet
        printf("Component %d, visit", ++numComp);
        dfs(i); // one component found
        printf("\n");
    }
printf("There are %d connected components\n", numComp);
```
Graph Traversal Comparison

- **DFS**
  - **Pros:**
    - Slightly easier to code
    - Use less memory
  - **Cons:**
    - Cannot solve SSSP on unweighted graphs

- **BFS**
  - **Pros:**
    - Can solve SSSP on unweighted graphs (discussed later)
  - **Cons:**
    - Slightly longer to code
    - Use more memory
BFS (unweighted)
Dijkstra’s (non–ve cycle)
Bellman Ford’s (may have–ve cycle), not discussed
Floyd Warshall’s (all-pairs)

SHORTEST PATHS
BFS for **Special Case SSSP**

- SSSP is a classical problem in Graph theory:
  - Find shortest paths from one source to the rest


- Problem Description:
  - Given an un-weighted & un-directed Graph, a starting vertex \( v \), and an integer TTL
  - Check how many nodes are un-reachable from \( v \) or has distance > TTL from \( v \)
    - i.e. \( \text{length}(\text{shortest\_path}(v, \text{node})) > \text{TTL} \)
Example (1)

$$Q = \{5\}$$

$$D[5] = 0$$
Example (2)

Q = {5}
Q = \{1, 6, 10\}

D[5] = 0
Example (3)

Q = \{5\}
Q = \{1, 6, 10\}
Q = \{6, 10, 0, 2\}
Q = \{10, 0, 2, 11\}
Q = \{0, 2, 11, 9\}

D[5] = 0
D[0] = D[1] + 1 = 2
Example (4)

\( Q = \{5\} \)
\( Q = \{1, 6, 10\} \)
\( Q = \{6, 10, 0, 2\} \)
\( Q = \{10, 0, 2, 11\} \)
\( Q = \{0, 2, 11, 9\} \)
\( Q = \{2, 11, 9, 4\} \)
\( Q = \{11, 9, 4, 3\} \)
\( Q = \{9, 4, 3, 12\} \)
\( Q = \{4, 3, 12, 8\} \)

\( D[5] = 0 \)
\( D[1] = D[5] + 1 = 1 \)
\( D[10] = D[5] + 1 = 1 \)
\( D[0] = D[1] + 1 = 2 \)
\( D[4] = D[0] + 1 = 3 \)
This is the BFS = SSSP spanning tree when BFS is started from vertex 5