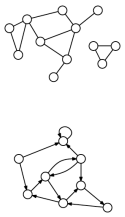
CS 340 - Analysis of Algorithms Graph Theory Review

Dianna Xu

1

Graph

- A graph $G = (V, E)$ represents a set of vertices V and a set of edges E.
- E may consist of unordered or ordered pairs of vertices and the resulting graph is undirected/directed.
- Edges and vertices may have weights



2

2

Terminology

- Given an edge $e = (u, v)$, u and v are *endpoints* of e and e is *incident* on u and v. u and v are *adjacent*.
- The *degree* of a vertex $\deg(v)$ is the number of incident edges on v in an undirected graph, or the number of outgoing edges in a directed graph.
- $|V| = n$
- $|E| = m$

3

3

Basic Graph Combinatorics

- Undirected graph
- $|E| = 0 \leq m \leq \binom{n}{2}$
- $= \frac{n(n-1)}{2} \in O(n^2)$
- $\sum_{v \in V} \deg(v) = 2m$

- Directed graph
- $|E| = 0 \leq m \leq n^2 - n$
- $\in O(n^2)$
- $\sum_{v \in V} \text{indeg}(v) = m$
- $\sum_{v \in V} \text{outdeg}(v) = m$

4

4

Terminology

- A *path* in a graph is a sequence of vertices $\langle v_0, \ldots, v_k \rangle$ such that (v_{i-1}, v_i) is an edge for $i = 1, \ldots, k$
- The *length* of a path is the number of edges, k.
- A *cycle* is a path containing at least one edge and for which $v_0 = v_k$
- A cycle is simple if its edges and vertices (except for v_0 and v_k) are distinct.

5

5

Terminology

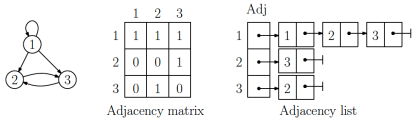
- An *acyclic* graph contains no simple cycles
- An acyclic connected graph is a *tree*
- The vertices of a *bipartite* graph can be partitioned into two disjoint subsets, V_1 and V_2 such that all edges have one endpoint in V_1 and the other one in V_2

6

6

Representation

- Adjacency matrix: An $n \times n$ matrix defined for $1 \le v, w \le n$: $A[i,j] = 1$ if $(u,v) \in E$
- Adjacency list: An array of pointers where for $1 \le v \le n$, $Adj[v]$ points to a list containing the vertices that are adjacent to v

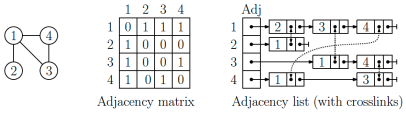


7

7

Undirected Graphs

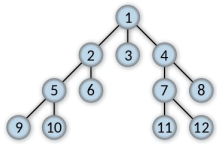
- Adjacency matrix: store the edges twice
- Adjacency list: cross-links



8

8

Graph Traversals: BFS

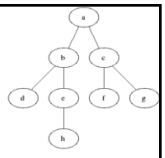


- Given a graph $G = (V, E)$, breadth-first search starts at some vertex s and visits vertices reachable from s in layers.
- Define the distance between a vertex v and s to be the minimum number of edges on a path from s to v.
- BFS visits the vertices in increasing order of distance.

9

9

BFS



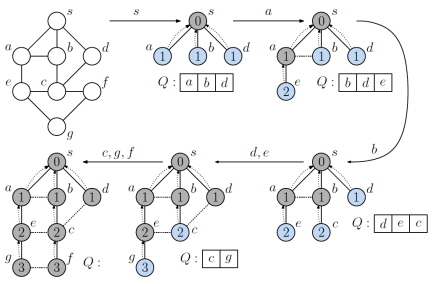
```
BFS(G, s) {
  set mark to all false
  mark[s] = true, Q = {s}
  while (Q is not empty) {
    u = dequeue of Q
    for each (v in Adj[u]) {
      if (!mark[v]){
        mark[v]=true
        append v to Q
      }
    }
  }
}
```

- `mark` array is used to track which vertices have been visited
- `Q` is a FIFO queue
- `Q` contains the frontier/level the BFS is currently searching

10

10

BFS on a Graph



11

11

BFS Time Analysis

```
BFS(G, s) {
  set mark to all false
  mark[s] = true, Q = {s}
  while (Q is not empty) {
    u = dequeue of Q
    for each (v in Adj[u]) {
      if (!mark[v]){
        mark[v]=true
        append v to Q
      }
    }
  }
}
```

- $|V| = n, |E| = m$
- Initialization requires $O(n)$
- Traversal loop
  - while: we never visit a vertex twice
  - for each: depends on the degree of vertex
- $T(n,m) = n + \sum_{u \in V}(\deg(u) + 1)$
- $= n + \sum_{u \in V} \deg(u) + \sum_{u \in V} 1$
- $= n + \sum_{u \in V} \deg(u) + n$
- $= 2n + \sum_{u \in V} \deg(u)$
- $= 2n + 2m$
- $O(n+m)$

13

13

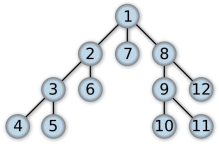
BFS with More Record Keeping

```
BFS(G, s) {
  for each (u in V) {
    mark[u] = false, d[u] = infinity, pred[u] = null
  }
  mark[s] = true, d[s] = 0, Q = {s}
  while (Q is not empty) {
    u = dequeue of Q
    for each (v in Adj[u]) {
      if (!mark[v]){
        mark[v] = true
        d[v] = d[u]+1
        pred[v] = u
        append v to Q
      }
    }
  }
}
```

14

14

Graph Traversals: DFS

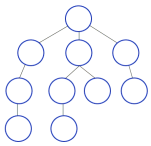


- Given a graph $G = (V, E)$, depth-first traversal strives for maximal depth and backtracks only when necessary.
- Recursive algorithm

15

15

DFS



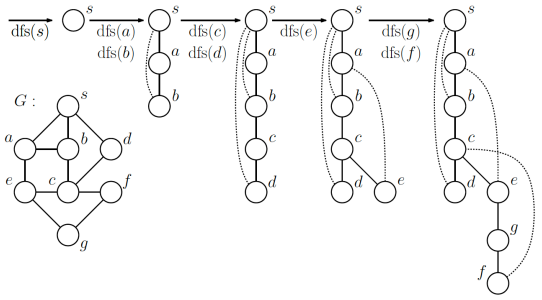
```
DFSG(G) {
  set mark to all false
  for each (v in V) {
    if (!mark(v))
      DFS(v)
  }
}
DFS(u) {
  mark[u] = true
  for each (v in Adj[u]) {
    if (!mark[v]){
      DFS(v)
    }
  }
}
```

- `mark` array used to track seen vertices
- The wrapper is only needed if not all vertices are reachable from source

16

16

DFS on Graph



17

17

Running Time

```
DFSG(G) {
  set mark to all false
  for each (v in V) {
    if (!mark(v))
      DFS(v)
  }
}
DFS(u) {
  mark[u] = true
  for each (v in Adj[u]) {
    if (!mark[v]){
      DFS(v)
    }
  }
}
```

- $|V| = n, |E| = m$
- Initialization $O(n)$
- DFS is called once per vertex (in wrapper or recursively)
- $T(n,m) = n + \sum_{u \in V}(\deg(u) + 1)$
- $= n + \sum_{u \in V} \deg(u) + \sum_{u \in V} 1$
- $= 2n + \sum_{u \in V} \deg(u)$
- $= 2n + 2m$
- $O(n + m)$

18

18

DFS Additional Record Keeping

- start times array: `s`
- finish times array: `f`
- predecessors array: `pred`

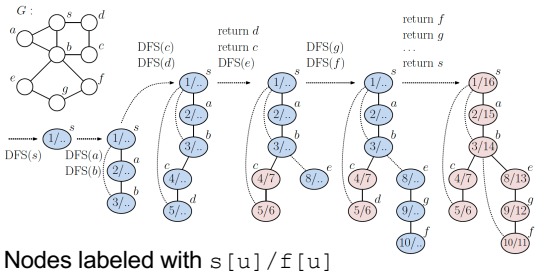
```
DFSG(G) {
  time = 0
  for each (u in V) {
    mark[u] = unseen
  }
  for each (u in V) {
    if (mark[u] == unseen)
      DFS(u)
  }
}
```

```
DFS(u) {
  mark[u] = seen
  s[u] = time++
  for each (v in Adj[u]) {
    if (mark[v] == unseen){
      pred[v] = u
      DFS(v)
    }
  }
  mark[u] = finished
  f[u] = time++
}
```

19

19

DFS on Undirected Graph

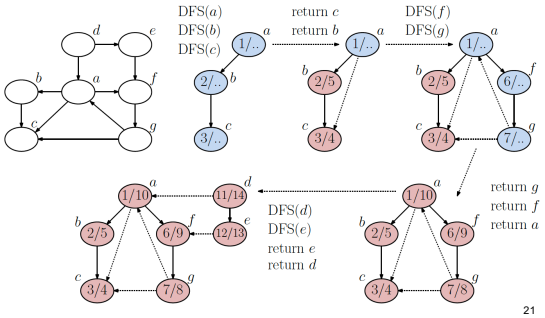


Nodes labeled with `s[u]/f[u]`

20

20

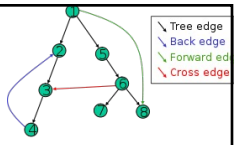
DFS on Directed Graph



21

21

DFS Edge Classification



- If v is visited for the first time as we traverse (u, v), then (u, v) is a tree edge
- else, v has already been visited
  - if v is an ancestor of u, (u, v) is a back edge
  - if v is a descendent of u, then (u, v) is a forward edge
  - if v is neither, then (u, v) is a cross edge

22

22

Read and Review

- Chapter 2 – review binary heaps and priority queues
  - insertion
  - deletion
  - deleteMin
  - changeKey
- Review binary search trees
  - AVL or Red/Black
  - search
  - insertion
  - deletion
  - traversal

23

23