Lecture 37: Graphs III

CS 62

Spring 2018

Alexandra Papoutsaki & William Devanny

DFS/BFS traversal

- Can be performed in $O(n + m)$, where $n = |V|, m = |E|$
- Can :
  - Test if G is connected
    - If traversal visited all vertices, then graph is connected
  - Compute a spanning tree of G, if G is connected
  - Find a path between two vertices, if it exits
  - Compute the connected components of G
    (needs to loop over all vertices and run DFS/BFS again)

Connectivity in Digraphs

- **reachable vertices**: when there is a directed path from one to another.
- **strongly connected vertices**: if mutually reachable
- **strongly connected digraph**: directed path from every vertex to every other vertex
- **weakly connected graph**: a digraph that would be connected if all of its directed edges were replaced by undirected edges.

Testing connectivity

- For an undirected graph:
  - Run DFS/BFS from any vertex without restarting and see if all vertic es are marked

- For strong connectivity on a directed graph:
  - 1. Initialize all vertices are not visited
  - 2. Run DFS/BFS from an arbitrary vertex v.
    - If traversal does not visit all vertices return false
  - 3. Reverse all edges
  - 4. Start from same vertex v and perform DFS/BFS. Graph is strongly connected iff all vertices are marked as visited again.

Single Source Shortest Path Problem

- From a starting node **s**, find the shortest path (and its length) to all other (reachable) nodes
- The collection of all shortest paths form a tree, called... the *shortest path tree*!
- If all edges have the same weight, we can use *BFS*.
- Otherwise …

Single Source Shortest Path Problem

- If all edges have weights ≥ 0 then use Dijkstra's algorithm
- Essentially BFS with priority queue
- Priorities are best known distance to a node from s
- We can keep track of parent nodes to get shortest path
- Example of a **greedy** algorithm

Dijkstra’s algorithm (1956) pseudocode

```
Q = {}; //set with unvisited vertices
for(every vertex v in V) {
   dist[v] = Infinity;
   parents[v] = null;
   Q.add(v);
}
   dist[s] = 0;
   while (!Q.isEmpty()) {
      u = vertex in Q with min dist[u];
      Q.remove(u);
      for(every edge (u,v)) {
         tentative = dist[u] + weight(u,v);
         if (tentative < dist[v]) {
            dist[v] = tentative;
            parents[v] = u;
         }
      }
   }
```

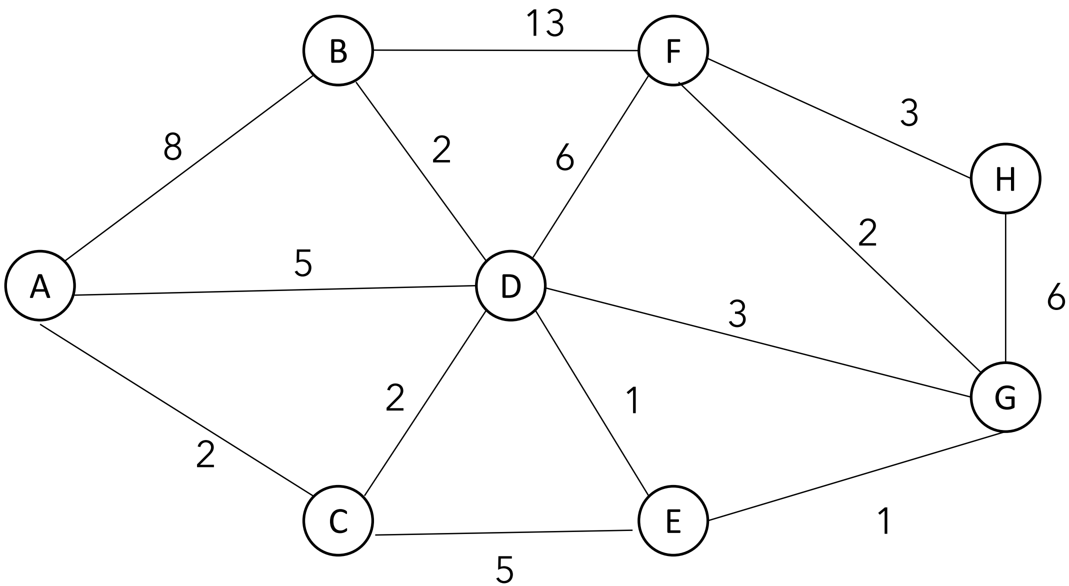
Dijkstra’s algorithm (1984) pseudocode

```
Q = new PriorityQueue();
for(every vertex v in V) {
   dist[v] = Infinity;
   parents[v] = null;
   Q.addWithPriority(v,dist[v]);
}
   dist[s] = 0;
   Q.addWithPriority(s, 0);
   while (!Q.isEmpty()) {
      u = Q.extractmin();
      Q.remove(u);
      for(every edge (u,v)) {
         tentative = dist[u] + weight(u,v);
         if (tentative < dist[v]) {
            dist[v] = tentative;
            parents[v] = u;
            Q.reducePriority(v, tentative);
         }
      }
   }
```

Run-time of Dijkstra

- Adding and removing from priority queue: $O(\log n)$
  - Each goes on and off once, so $O(n \log n)$
- `reduce_priority`: $O(\log n)$
  - Worst case, once for each edge, so $O(m \log n)$
- Total time: $O((m + n) \log n)$

Dijkstra on sample graph



Dijkstra on sample graph

	A	B	C	D	E	F	G	H
Init	0_A	∞	∞	∞	∞	∞	∞	∞
A	0_A	8_A	2_A	5_A	∞	∞	∞	∞
C	0_A	8_A	2_A	4_C	7_C	∞	∞	∞
D	0_A	6_D	2_A	4_C	5_D	10_D	7_D	∞
E	0_A	6_D	2_A	4_C	5_D	10_D	6_E	∞
B	0_A	6_D	2_A	4_C	5_D	10_D	6_E	∞
G	0_A	6_D	2_A	4_C	5_D	8_G	6_E	12_E
F	0_A	6_D	2_A	4_C	5_D	8_G	6_E	11_F
H	0_A	6_D	2_A	4_C	5_D	8_G	6_E	11_F

Follow the subscripts to find shortest path from start to any vertex

Practice Time