Graph Traversal: Breadth-First Search and Depth-First Search

Module 5: Graphs

Overview

We describe the graph traversal algorithms, BFS and DFS (breadth-first and depth-first search), explore their properties and look at problems we can solve with them.

Graph traversal

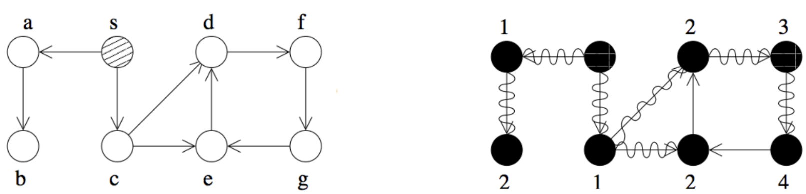
- Consider a graph, directed or undirected.
- The most basic graph problem is *traversing* the graph. There are two simple ways of traversing all vertices/edges in a graph in a systematic way: BFS and DFS
- Basic idea: over the course of the traversal the state of a vertex progresses from undiscovered, to discovered, to completely-discovered:
  - undiscovered or UNVISITED initially
  - discovered and IN-PROGRESS : after it's encountered, but before it's completely explored
  - completely explored or ALL-DONE the vertex after we visited all its incident edges
- Graph traversal starts with a single vertex and evaluate its outgoing edges:
  - If an edge goes to an UNVISITED vertex: we mark it as IN-PROGRESS and add it to the list of IN-PROGRESS vertices.
  - If an edge goes to a completely explored ALL-DONE vertex: we ignore it (we've already been there)
  - If an edge goes to an IN-PROGRESS vertex: we ignore it (it's already on the list).
- Depending on how we store the list of IN-PROGRESS vertices we get BFS or DFS:
  - queue: explore oldest vertex first. The exploration propagates in layers form the starting vertex.
  - stack: explore newest vertex first. The exploration goes along a path, and backs up only when new unexplored vertices are not available.
- Analysis: Each edge is visited once (for directed graphs), or twice (undirected graphs — once when exploring each endpoint) $\Rightarrow O(|V| + |E|)$. More on this later.

Breadth-first search (BFS)

- BFS uses a **queue** to hold the IN-PROGRESS vertices (which are the vertices we have seen but are still not done with).
- BFS can compute the following additional information for each vertex v: its parent and its distance from the source
  - parent[v]: this is the vertex that discovered v first
  - $d[v]$: the length of the path from s to v. Initially $d[s] = 0$. We'll see that $d[v]$ represents the length of the shortest path from s to v

BFS(vertex s)
    state[s] = IN-PROGRESS
    $d[s] = 0$
    Q.insertEnd(s)
    WHILE Q not empty DO
        u= Q.removeFront()
        FOR each $v \in adj[u]$ DO
            IF state[v] = UNVISITED THEN
                state[v] = IN-PROGRESS
                $d[v] = d[u] + 1$
                parent[v] = u //(u,v) is a tree-edge
                Q.insertEnd(v)
            //ELSE: v is not UNVISITED => (u,v) is non-tree edge
        state[u] = ALL-DONE //we are done exploring vertex v

- Example (for directed graph):



- Note that you can run BFS from an arbitrary vertex in the graph. BFS(s) will reach all vertices that are reachable from (are connected to) source vertex s.
- If graph is not connected: after BFS(s), some vertices in the graph will still be UNVISITED . To explore the whole graph, we start the traversal at all vertices until the entire graph is explored.

```
BFS (graph G)
for each vertex u ∈ V
      IF state[u] = UNVISITED : BFS(u)
```

Properties of BFS

- **Lemma:** On a directed graph, BFS(s) visits all vertices reachable from s. On an undirected graph, BFS(s) visits all vertices in the connected component (CC) of s.

  Proof sketch: Assume by contradiction that there is a vertex v in CC(u) that is not reached by BFS(u). Since u, v are in same CC, there must exist a path $v_0 = u, v_1, v_2, ..., v_k, v$ connecting u to v. Let v_i be the last vertex on this path that is reached by BFS(u) (v_i could be u). When exploring v_i, BFS must have explored edge (v_i, v_{i+1}),..., leading eventually to v. Contradiction.

- **Lemma:** BFS(s) runs in $O(|V_c| + |E_c|)$, where V_c, E_c are the number of vertices and edges in CC(s). When run on the entire graph, BFS(G) runs in $O(|V| + |E|)$ time. Put differently, BFS runs in linear time in the size of the graph.

  Proof: It explores every vertex once. Once a vertex is marked, it is not explored again. It traverses each edge (u, v) once (twice, on an undirected graph). Overall, this is $O(|V| + |E|)$.

- **Lemma:** Let x be a vertex reached in BFS(s). Its distance $d[x]$ represents the the shortest path from s to x in G.

  Proof sketch: All vertices v which are one edge away from s are discovered when exploring s and are set with $d[v] = 1$, which is correct. Now consider a vertex v whose shortest path from s is two edges, and let u be the intermediate vertex on the shortest path from s to v. Since there is an edge (s, u), vertex u will be discovered from s and set with $d[u] = 1$, and then when u is explored, it discovers vertex v and sets $d[v] = 2$.

  In general, we use induction on the length of the shortest path. Assume inductively that any vertex u whose shortest path consists of $k-1$ edges is set correctly with $d[u] = k-1$. Let v be a vertex whose shortest path from s consists of k edges: $< s, v_1, v_2, ..., v_{k-1}, v_k = v >$. When vertex v_{k-1} is explored, it will discover v_k and set $d[v] = d[v_{k-1}] + 1$. Note that the shortest path from s to v_{k-1} consists of $k-1$ edges, and by induction hypothesis we have that $d[v_{k-1}] = k-1$. Then it follows that $d[v] = (k-1) + 1 = k$.

- Each vertex, except the source vertex s, has a parent; these edges $(v, parent[v])$ define a tree, called the *BFS-tree*. During BFS(v) each edge in G is classified as either: (1) a tree edge: an edge leading to an UNVISITED vertex; or (2) a non-tree edge: an edge leading to a vertex that's been discovered before (so the vertex is either IN-PROGRESS or ALL-DONE).

- **Lemma:** For undirected graphs, for any non-tree edge (x, y) in BFS(v), the level of x and y differ by at most one.

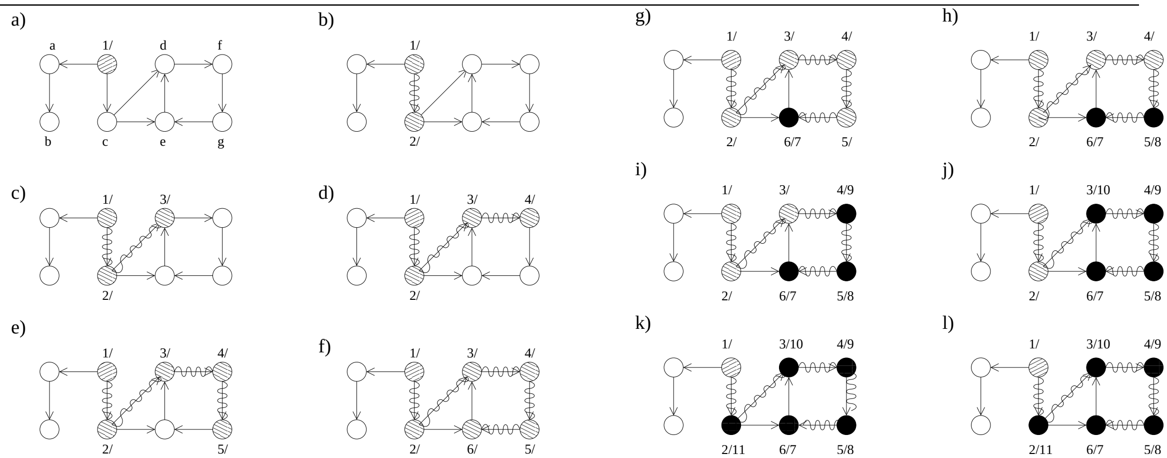
Proof idea: Observe that, at any point in time, the vertices in the queue have distances that differ by at most 1. Let's say x comes out first from the queue; at this time y must be already marked (because otherwise (x, y) would be a tree edge). Furthermore y has to be in the queue, because, if it wasn't, it means it was already deleted from the queue and we assumed x was first. So y has to be in the queue, and we have $|d(y) - d(x)| \leq 1$ by above observation.

Depth-first search (DFS)

- DFS uses a **stack** instead of queue to hold discovered (UNVISITED) vertices
- DFS computes the following additional information for each vertex:
  - start[u]: time when a vertex is first visited.
  - finish[u]: time when all adjacent vertices of u have been visited and v is set as ALL-DONE
- We can write DFS iteratively using the same algorithm as for BFS but with a STACK instead of a QUEUE;
- The standard DFS implementation is recursive:

```
DFS( VERTEX u)
    state[u] = IN-PROGRESS
    start[u] = time
    time = time + 1
    FOR each v ∈ adj[u] DO
        IF state[v] = UNVISITED THEN
            parent[v] = u
            DFS(v)
    state[u] = ALL-DONE
    finish[u] = time
    time = time + 1
```

- Example:



- Note that you can run DFS from an arbitrary vertex in the graph. DFS(s) will reach all vertices that are reachable from (are connected to) source vertex s.
- If graph is not connected: after DFS(s), some vertices in the graph will still be UNVISITED . To explore the whole graph, we start the traversal at all vertices until the entire graph is explored.

```
DFS (graph G)
for each vertex u ∈ V
      IF state[u] = UNVISITED : DFS(u)
```

Properties of DFS

- On a directed graph, DFS(u) reaches all vertices reachable from u. On an undirected graph, DFS(u) visits all vertices in CC(u).
- Analysis: DFS(s) runs in $O(|V_c| + |E_c|)$, where V_c, E_c are the number of vertices and edges in CC(s) (reachable from s, for directed graphs). When run on the entire graph, DFS(G) runs in $O(|V| + |E|)$ time. Put differently, DFS runs in linear time in the size of the graph.
- Each vertex, except the source vertex s, has a parent; these edges $(v, parent[v])$ define a tree, called the *DFS-tree*.
- Nesting of descendants: If u is a descendent of v in the DFS-tree then $d[v] < d[u] < f[u] < f[v]$. And, it can be shown that this is true the other way around as well: If $d[v] < d[u] < f[u] < f[v]$ then u is descendent of v in DFS-tree.