ESO211 (Data Structures and Algorithms) Lectures 33-34

Shashank K Mehta

1 Some definitions about graphs

Definition 1 *An* undirected *graph is (V, E) where the elements of set V are called* vertices *and E is a collection of some vertex 2-sets (sets containing two vertices) called the edges of the graph. Usually we denote an edge $\{x, y\}$ as (x, y). A* subgraph *of $G = (V, E)$ is any graph $G' = (V', E')$ where $V' \subseteq V$ and $E' \subseteq E$.*

Definition 2 *A* walk *in a graph $G = (V, E)$ is a sequence of vertices $v_{i_1} v_{i_2} \ldots v_{i_k}$ where $(v_{i_j}, v_{i_{j+1}}) \in E$ for all j. A path is a walk $v_{i_1} v_{i_2} \ldots v_{i_k}$ where $v_{i_s} \neq v_{i_t}$ if $s \neq t$. A* closed walk *is a walk $v_{i_1} v_{i_2} \ldots v_{i_k} v_{i_1}$. A* cycle *is a closed walk $v_{i_1} v_{i_2} \ldots v_{i_k} v_{i_1}$ where $v_{i_1} v_{i_2} \ldots v_{i_k}$ is a path.*

consider a walk from v_{i_1} to v_{i_s}: $v_{i_1} \alpha v_{i_r} \beta v_{i_r} \gamma v_{i_s}$ where α, β, and γ are sub-walks. Then $v_{i_1} \alpha v_{i_r} \gamma v_{i_s}$ is also a walk. Repeating this step we can show that if there is a walk from v_{i_1} to v_{i_s} in a graph G, then there must be a path from v_{i_1} to v_{i_s} in G. As a special case of this observation, we can claim that if there is closed walk passing through a vertex v, then there is a cycle passing through v. Note that the last claim is due to the fact that a closed walk can be written in the way such that any of its vertex becomes the first vertex.

Definition 3 *A graph is* connected *if it contains a path between each pair of vertices.*

Definition 4 *A* tree *is a connected graph without a cycle. A* forest *is any cycle-free graph. A rooted tree is just a tree but one of its vertices is identified as its* root*. A* spanning tree *of a graph $G = (V, E)$ is a subgraph (V', E') which is a tree and $V' = V$.*

Definition 5 *A* directed *graph is (V, E) where each member of E is an edge with a specified direction. So its elements are not 2-sets but are ordered 2-tuples. So a graph may have either one or both or neither of the edges (u, v) and (v, u), for each $u, v \in V$. We visualize (u, v) as an arrow from u to v. In a directed graph, a walk is $v_{i_1} v_{i_2} \ldots v_{i_k}$ only if the arrow is from $(v_{i_j}$ to $v_{i_{j+1}})$ for all j.*

Observation 1 *In a tree there exists exactly one path between each pair of vertices.*

Observation 2 *Graph $G = (V, E)$ is a tree if and only if it is connected and $|E| = |V| - 1$.*

Definition 6 Length *of a walk/path/cycle is the number of edges in it. Let $G = (V, E)$ be a graph and $x, y \in V$, then the* distance *between x and y in G is the length of the shortest path between x and y in G, denoted by $\delta_G(x, y)$. Let G be a connected graph and s be a vertex in it. Then a spanning tree T of $G = (V, E)$ is called a* shortest path tree *of s if $\delta_G(s, x) = \delta_T(s, x)$ for each $x \in V$.*

Observation 3 *Let $v_{i_1} v_{i_2} \dots v_{i_k}$ be a shortest path between v_{i_1} and v_{i_k}. Then $v_{i_j} v_{i_{j+1}} \dots v_{i_{j+r}}$ is a shortest path between v_{i_j} and $v_{i_{j+r}}$.*

Corollary 1 *Let G be a connected graph. Then there exists a shortest path tree in G for each of its vertices.*

Proof left as an exercise.

2 Problem

2.1 Problem Statement

Given a connected graph and a specific vertex s in it. Compute a shortest path tree for s in it.

2.2 BFS Algorithm

Following algorithm computes the shortest-path-tree based on BFS (Breadth-First-Search). Here Q is a queue data-structure. The idea behind this algorithm is capturing the propagation of light from a single point source in a discrete manner using Huygen's principle. In a BFS we visit all the neighbors of the starting vertex s. These vertices collectively form the first wave-front (assuming s as the wavefront 0). Then we visit all the neighbors of each vertex of wavefront 1. All the new vertices visited constitute wavefront 2. This is how we complete the visit to the entire graph. This order of visiting all the vertices is known as breadth-first search/visit. In the algorithm we compute two additional parameters for each vertex which helps us compute the shortest path tree for s.

The algorithm outputs shortest-path tree of s, as we will show later. It is also known as BFS-tree. The edges in this spanning tree are $(x, parent(x))$ for all $x \in V \setminus \{s\}$.

Two most suitable data-structures for storing a graph are (i) adjacency matrix and (ii) adjacency lists. In the former we have a $|V| \times |V|$ matrix having ij-th entry 1 if $(x_i, x_j) \in E$ else it is 0. In the latter case we have lists $Adj[u]$ pointing to a list of vertices which are adjacent to u, for each $u \in V$. The space complexity of the matrix is $O(n^2)$ and that of the latter is $O(m)$ where $n = |V|$ and $m = |E|$.

```
**Data**: $G = (V, E)$ is a connected graph and $s \in V$
**Result**: BFS tree $T$
initialize_queue($Q$);
**for** *each* $v \in V$ **do**
    $status[v] \leftarrow$ "*unvisited*";
    $parent[v] \leftarrow undefined$;
    $d[v] \leftarrow \infty$;
**end**
$status[s] \leftarrow$ "*currently_visiting*";
$d[s] \leftarrow 0$;
enqueue($s, Q$);
**while** $Q \neq \phi$ **do**
    $u \leftarrow dequeue(Q)$;
    **for** *each* $v \in Adj[u]$ **do**
        **if** $status[v] =$ "*unvisited*" **then**
            $status[v] \leftarrow$ "*currently_visiting*";
            enqueue($v, Q$);
            $d[v] \leftarrow 1 + d[u]$;
            $parent[v] \leftarrow u$;
        **end**
    **end**
    $status[u] \leftarrow$ "*visit_complete*";
    **return** $(V, \{(x, parent(x)) | x \in V \setminus \{s\}\})$;
**end**
```

Algorithm 1: BFS(s,G)

2.3 Proof of Correctness

Claim 1 *Each vertex enters and exists Q exactly once. If x and y co-existed in Q any point in time and if x was ahead of y, then $d(x) \leq d(y) \leq d(x) + 1$.*

Proof Every vertex, encountered in *unvisted* state, is placed in Q and its state is changed to *current*. Since we never change the state of any vertex back to *unvisited*, it is impossible for any vertex to enter Q more than once. Suppose a vertex x is never encountered in BFS and hence never entered in Q. Since the graph is connected, there must be a path from s to x, say $sy_1y_2 \ldots y_k(= x)$. We do place s into Q initially. If there is any vertex in the path that is never placed in Q, then it must be y_i for some i. Let i_0 is the least index such that y_{i_0} was never placed in Q but y_{i_0-1} was placed in Q. Since y_{i_0} is adjacent to y_{i_0-1} (means, $\{y_{i_0}, y_{i_0-1}\}$ is an edge in the graph), while expanding y_{i_0-1} must have been encountered. At that time either y_{i_0} was already visited or that was the first visit. But in either case y_{i_0} must have been entered in Q. So we conclude that every vertex enters Q at least once. Putting together two facts we see that every vertex enters and exists Q exactly once.

To prove the second part of the claim we will use induction on time. Initially Q only contained s so the claim was vacuously true. Suppose at some time Q

contains one or more vertices with $d()$ value t in front and zero or more vertices with d value $t+1$ are at the back. Now u dequeued from Q. So $d(u) = t$. Now as we expand u all the new vertices added to Q (at the back) have d value $t+1$. Finally u will be deleted. Once again, after the expansion of u, we will find that vertices of at most two wavefronts will be present in Q and all the vertices of earlier wavefront will be ahead of all the vertices of the next wavefront. ■

Claim 2 *$\delta_G(s, x) \leq d[x]$ for all $x \in V$.*

Proof Sketch $d[x]$ is the length of some path from s to x, namely, $x, parent(x), parent^2(x), \ldots, s$. But $\delta_G(s, x)$ is the length of the shortest path from x to s. ■

Claim 3 *$\delta_G(s, x) = d[x]$ for all $x \in V$.*

Proof We will that the claim is valid for k when $(\delta_G(s, x) = k) \Rightarrow (d(x) = k)$ holds for all x. By induction we will show that the claim holds for all k.

As the base case claim holds for $k = 0$ because s is the only vertex such that $\delta_G(s, s) = 0$ and we know that $d(s) = 0$.

Now for induction step, suppose the claim holds for $k = k_0 - 1$. Let $\delta_G(s, x) = k_0$. The there must exist a path from s to x of length k_0. Suppose the second last vertex on the path (before x) is y. Since the subpath from s to y is also a shortest path, $\delta_G(s, y) = k_0 - 1$. From induction hypothesis, $d(y) = k_0 - 1$. During the expansion of y we must have encountered x. If x was in unvisited sate at that time then $d(x)$ must have been set to $d(y) + 1 = k_0 - 1 + 1 = k_0$. On the other hand, if x was encountered earlier, then its d value can be k_0 or less because d values of the successively emerging vertices can only increase. Hence $d(x) \leq k_0$. But we know that d value of x is the length of the path $P = x, parent(x), parent^2(x), \ldots, s)$ so $d(x) \geq \delta_G(s, x) = k_0$. Hence $d(x) = k_0$. This completes the proof. ■

Claim 4 *T is the shortest-path-tree for s.*

Proof Since G is connected, we have seen that each vertex enters and exits Q exactly once. Thus every vertex is visited in the graph. Since $parent(x)$ is defined for all $x \in V - \{s\}$ and x is in the neighborhood of $parent[x]$, $(x, parent[x])$ is an edge of G. So number of edges in T is $|\{(x, parent(x)) | x \in V - \{s\}\}| = |V| - 1$. To show that T is a shortest-path-tree now we only need to show that T is connected.

Consider the path $P_x = x, parent(x), parent^2(x), \ldots.$ the d value of each vertex in the path is 1 less than that of the previous vertex, if exist. Hence no vertex can repeat. Thus this path must eventually end. But the only vertex that does not have a parent is s. So it must terminate at s. Thus we have shown that every vertex has a path to s. That establishes that T is connected. Thus T is a tree. Besides every vertex of the graph is on it so it must be spanning vertex. Finally we see that P_x has length $d(x) = \delta_G(s, x)$ so it is a shortest path in G from s to x. Thus T is a shortest path tree of s. ■

2.4 Time and Space Complexity

Since each edge is visited exactly twice (once from each side) and each vertex enters and exits exactly once, the time complexity of this algorithm is $O(n+m)$. The algorithm requires the graph to be stored as adjacency lists of its vertices and the queue requires to store at most $O(n)$ space. So the space complexity is also $O(n+m)$.