Breadth-First Search

Breadth-First Search (BFS) is an algorithm for visiting every vertex in an undirected graph \( G = (V, E) \), starting from one node \( s \) in \( V \). BFS visits all nodes \( d \) hops from \( s \) before visiting any nodes \( d + 1 \) hops away from \( s \), for all \( d \geq 0 \). It only takes into account the presence (or absence) of a link in the graph: it does not take link costs into account.

The BFS search procedure below induces a tree rooted at \( s \) over \( G \); the predecessor dictionary records this tree. \( G.NEIGHBORS(u) \) is a function that returns a list of vertices in \( V \) neighboring \( u \).

\[
\text{BFS}(G, s) \quad  \\
1 \quad \text{visited} \leftarrow \emptyset  \\
2 \quad \text{predecessor} \leftarrow \emptyset  \\
3 \quad Q \leftarrow [s]  \\
4 \quad \text{while not } Q.	ext{EMPTY()}  \\
5 \quad \quad \text{do } u \leftarrow Q.	ext{DEQUEUE()}  \\
6 \quad \quad \quad \text{for } v \in G.NEIGHBORS(u)  \\
7 \quad \quad \quad \quad \text{do if not } \text{visited}[v]  \\
8 \quad \quad \quad \quad \quad \text{then } \text{visited}[v] \leftarrow \text{True}  \\
9 \quad \quad \quad \quad \quad \quad \text{predecessor}[v] \leftarrow u  \\
10 \quad \quad \quad \quad \quad Q.	ext{APPEND}(v)
\]

In lines 1–3, BFS initializes the set of visited vertices and the predecessor tree to empty, and the queue to contain only the starting vertex. BFS then visits all neighbors \((u, \text{line 6})\) of the vertex at the head of the queue \((u, \text{line 5})\) before any others, constructing the breadth-first tree (line 9).

Dijkstra’s Algorithm

Dijkstra’s algorithm is an algorithm for constructing a single-source shortest paths tree rooted at a vertex \( s \) in a graph \( G = (V, E) \). In contrast with BFS, Dijkstra’s algorithm uses the graph’s edge weights (a.k.a. link costs) to define the notion of ”shortest path.”

In the procedure below, \( G.COST(x, y) \) returns the edge weight between nodes \( x \) and \( y \). The dictionary \( d \) keeps track of the shortest path costs from \( s \) to every node, and the dictionary ”predecessor” records the shortest path tree that Dijkstra’s algorithm constructs.
Simplified-Dijkstra$(G, s)$

1. $nodes\_done \leftarrow [s]$
2. $d[s] \leftarrow 0$

3. while $nodes\_done$.LENGTH $<$ $G$.SIZE()
4.   do $next\_min\_cost \leftarrow \infty$
5.   for $u \in nodes\_done$
6.     do for $v \in G$.NEIGHBORS($u$)
7.         do $c \leftarrow d[u] + G$.COST($u, v$)
8.         if $c < d[v]$ and $c < next\_min\_cost$:
9.             then $next\_min\_cost \leftarrow c$
10.                $next\_node \leftarrow v$
11.                   $next\_parent \leftarrow u$
12.         $d[\text{next-node}] \leftarrow next\_min\_cost$
13.     $predecessor[\text{next-node}] \leftarrow next\_parent$
14. $nodes\_done$.APPEND($\text{next-node}$)

Comparing forwarding tables and shortest-path trees

In Lab 10 you don’t need to explicitly construct the shortest-paths tree as the Dijkstra’s pseudocode above does at line 13. Instead, you need to construct a forwarding table: information specifying for each destination, which outgoing link (proximal to itself) to use to send packets to any other destination. (Read the previous sentence again!) Therefore your Lab 10 code will differ in what information it records. Hint: once you know the correct proximal outgoing links for nodes $d$ hops away, you can use this information for nodes $d + 1$ hops away. (Note that “hops” refers to the number of links away from the source $s$, not the cost.)

Also note: the above pseudocode is less efficient than Dijkstra’s algorithms you will learn about in later courses.