Priority Quque (continued)

We finally give the pseudo-code for implementing the binary heap. We not only maintain the array S but also the number of elements in S, denoted as n.

```
function empty($PQ$)
    if $n = 0$: return true;
    else: return true;
end function;

function insert($PQ$, $x$)
    $n = n + 1$;
    $S[n] = x$;
    bubble-up ($S$, $n$);
end function;

function find-min($PQ$)
    return $S[1]$;
end function;

function delete-min($PQ$)
    $S[1] = S[n]$;
    $n = n - 1$;
    sift-down ($S$, 1);
end function;

function decrease-key($PQ$, $k$, new-key)
    $S[k].key$ = new-key;
    bubble-up ($S$, $k$);
end function;
```

Both bubble-up and sift-down procedures runs in $O(\log n)$ time. This is because, a complete binary tree with n vertices has a height of $\log n$, while the worst case of either procedure traverses along a path between the root and a leaf. Formally, in each recursive call, k is either halved or doubled, and hence the number of recursive calls is $\log n$. The empty and find-min procedures takes $\Theta(1)$ time; the other 3 procedures takes $O(\log n)$ time, as they are dominated by either bubble-up or sift-down.

Single-Source Shortest Path Problem with Unit Edge Length

In many applications, edges of graphs are associated with *lengths*. We will consider three cases.

1. unit edge length: the length of each edge is 1;
2. positive edge length: the length of each edge is a positive number;
3. length of edge might be a negative number.

Let $G = (V,E)$ be a graph; let $l(e)$ be the length of $e \in E$ (in any of above cases). Let p be a path of G. We define the length of path p, still denoted as $l(p)$, as the sum of the length of all edges in p, i.e., $l(p) := \sum_{e\in p} l(e)$. Given $u, v \in V$, the *shortest path* from u to v, is the path from u to v with smallest length. We use notation $distance(u,v)$ to denote the length of the shortest path from u to v.

There are different variants of shortest path problems. One category is *single-source shortest path problem*, where we are given graph $G = (V,E)$ with edge length (one of the three cases) and a *source* vertex $s \in V$, and we seek to find the shortest path from s to *all* vertices.

We first solve the easiest version of shortest path problem: given $G = (V,E)$ with unit edge length and a *source* vertex $s \in V$, to find the shortest path from s to all vertices, i.e., to find $distance(s,v)$ for any $v \in V$.

The breadth-first search (BFS) can solve above problem. The idea of BFS is to traverse the vertices of graph in increasing order of their distance from s. Formally, we define V_k, $k = 0,1,2,\cdots$, as the subset of vertices whose distance (from s) is exactly k, i.e., $V_k = \{v \in V \mid distance(s,v) = k\}$. BFS will first traverse vertices in V_0, then vertices in V_1, then V_2, and so on, until all vertices reachable from s are traversed.

Notice that $V_0 = \{s\}$ as $distance(s,s) = 0$. How about V_1? They are the vertices reachable from s with one edge (but cannot be s), i.e., $V_1 = \{v \in V \mid (s,v) \in E\} \setminus V_0$. How about V_2? They are the vertices reachable from some vertex in V_1 with one edge, but not in V_0 or V_1, i.e., $V_2 = \{v \in V \mid u \in V_1 \text{ and } (u,v) \in E\} \setminus (V_0 \cup V_1)$. The general form is that $V_{k+1} = \{v \in V \mid u \in V_k \text{ and } (u,v) \in E\} \setminus (V_0 \cup V_1 \cup \cdots \cup V_k)$. This suggests an iterative framework of BFS: to find V_{k+1}, explore the out-edges of V_k to collect the *newly* reached vertices (i.e., those not in $V_0 \cup V_1 \cup \cdots \cup V_k$). To realize this idea, BFS uses two data structures; the complete algorithm follows.

1. Array *dist* of size $|V|$, initialized as $dist[v] = \infty$ for any $v \in V$. Array *dist* serves as two purposes: indicating a vertex has been reached or not, and storing the distance from s (after it has been reached). Formally, before v is reached, $dist[v] = \infty$; after v is reached, $dist[v] = distance(s,v)$.

2. Queue Q, which stores the vertices haven't been explored. Vertices will be added to Q in the order of $V_0, V_1, V_2, \cdots$, and vertices will be deleted from Q in the same order (as queue is first-in-first-out).

```
Algorithm BFS ($G = (V,E), s \in V$)
    $dist[v] = \infty$, for any $v \in V$;
    init an empty queue $Q$;
    $dist[s] = 0$;
    insert ($Q, s$);
    while (empty ($Q$) = false)
        $u$ = find-earliest ($Q$);
        delete-earliest ($Q$);
        for each edge $(u,v) \in E$
            if ($dist[v] = \infty$)
                $dist[v] = dist[u] + 1$;
                insert ($Q$, $v$);
            end if;
        end for;
    end while;
end algorithm;
```

Initially BFS has V_0 (i.e., s) in Q (right before the while loop), then BFS deletes s from Q and explores s (newly reached vertices—these are V_1, will have their *dist* set as 1 and be added to Q); at the time of finishing exploring V_0, V_1 will be in Q. Next, BFS will gradually delete and explore each of the vertices in Q (i.e., V_1); in this process, vertices in V_2 will be reached, their *dist* be set as 2, and be added to Q; after all of them are deleted and explored, Q will exactly consist of V_2.

In general, BFS keeps the following invariant: for every $k = 0, 1, 2, \cdots$, there is a time at which Q contains exactly V_k, $dist[v] = distance(s, v)$ for any $v \in V_0 \cup V_1 \cup \cdots V_k$, and $dist[v] = \infty$ for all other vertices. This invariant explains the behavior of BFS while also proves its correctness: when the Q becomes empty, $dist[v] = distance(s, v)$ for any $v \in V$.

Following above invariant and the pseudo-code, we know that the time that vertex v is reached for the first time (happend when $(u, v) \in E$ is checked and $dist[v] = \infty$), is also the time that the shortest path to v is found, that the $dist[v]$ gets assigned, and that v is added to Q.

BFS runs in $O(|V| + |E|)$ time, as each vertex will be explored at most once, and each edge will be examined at most once (for directed graph) and at most twice (for undirected graph). Note that BFS (G, s) only traverses those vertices in G can be reached from s.

Single-Source Shortest Path Problem with Positive Edge Length

We now study the single-source shortest path problem with positive edge length: given graph $G = (V, E)$ with edge length $l(e) > 0$ for any $e \in E$ and source vertex $s \in V$, to seek $distance(s, v)$ for any $v \in V$. We solve this problem with the *Dijkstra's algorithm*.

Similar to BFS, the idea of Dijkstra's algorithm is also to determine and calculate the distance of each vertex in increasing order of distance. More specifically, let $(v_1^*, v_2^*, \cdots, v_n^*)$, $n = |V|$, be the order of vertices with increasing distance, i.e., $distance(s, v_k^*) \leq distance(s, v_{k+1}^*)$, $1 \leq k < n$. In other words, we define v_k^* as the k-th closest vertex from s. We don't know this order in advance. But Dijktra's algorithm will identify vertices in this order and calculate their distance. For the sake of writing and notations, we define $R_k = \{v_1^*, v_2^*, \cdots, v_k^*\}$, i.e., the first k vertices in above list; R_k are also the closest k vertices from s.

Clearly, $v_1^* = s$, $distance(s, v_1^*) = distance(s, s) = 0$, and $R_1 = \{s\}$. The key question is: given R_k (i.e., the closest k vertices from s) and their distances (i.e., $distance(s, v)$ for every $v \in R_k$), how to find v_{k+1}^*? Below we show that v_{k+1}^* must be within *one-edge extension* of R_k.

Claim 1. There must exist vertex $u \in R_k$ such that $(u, v_{k+1}^*) \in E$.

Proof. Suppose conversely that for every $u \in R_k$ we have $(u, v_{k+1}^k) \notin E$. Consider the shortest path from s to v_{k+1}^*. Let w be the vertex right before v_{k+1}^* in path p, i.e, (w, v_{k+1}^*) is the last edge of p. We have $distance(s, v_{k+1}^*) = distance(s, w) + l(w, v_{k+1}^*)$. As edges have positive edge length, we have $l(w, v_{k+1}^*) > 0$, and consequently $distance(s, w) < distance(s, v_{k+1}^*)$. Besides, according to the assumption, we have $w \notin R_k$. Therefore, v_{k+1}^* cannot be the $(k+1)$-th closest vertex from s, because w has a shorter distance than v_{k+1}^*. This contradicts to the definition of v_{k+1}^*. □

Note that, in above proof, we use the fact that edges have positive lengths. Hence, above claim may not be true for graphs with negative edge length. This also explains why Dijkstra's algorithm won't work for graphs with negative edge length.

The above claim shows that, the last edge (u, v) of the shortest path from s to v_{k+1}^* satisfies that $u \in R_k$ and

$v \notin R_k$. Which such edge is the correct one? We don't know, so we enumerate all such edges $(u,v) \in E$ with $u \in R_k$ and $v \notin R_k$. Suppose that (u,v) is the last edge of the shortest path from s to v^*_{k+1}, we know that $distance(s,v) = distance(s,u) + l(u,v)$. This leads to the following formula to calculate v^*_{k+1} and its predecessor in the shortest path. See an example in Figure 1.

Corollary 1. We have

$$distance(s, v^*_{k+1}) = \min_{u \in R_k, v \in V \setminus R_k, (u,v) \in E} (distance(s,u) + l(u,v)).$$

Let (u', v') be the optimal edge in the minimization, i.e.,

$$(u', v') := \arg\min_{u \in R_k, v \in V \setminus R_k, (u,v) \in E} (distance(s,u) + l(u,v)).$$

Then $v^*_{k+1} = v'$ and u' is the predecessor of v^*_{k+1} in the shortest path from s to v^*_{k+1}.

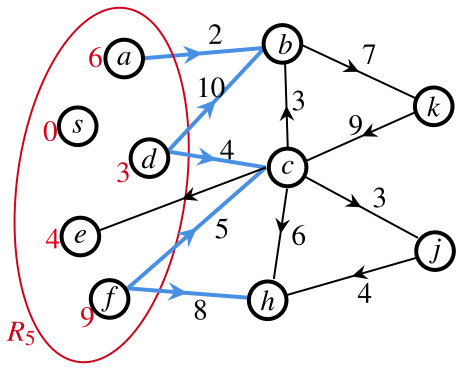

Figure 1: Example for Colollary 1. Suppose that we know $R_5 = \{s,a,d,e,f\}$ and their distances from s, marked next to vertices. To find v^*_6 and calculate $distance(s, v^*_6)$, we consider all one-edge extension of R_5, marked as thick blue edges. Following Corollary 1, $distance(s, v^*_6) = \min_{u \in R_5, v \notin R_5, (u,v) \in E}(distance(s,u) + l(u,v)) = \min\{6+2, 3+10, 3+4, 9+5, 9+8\} = 7$, and the optimal edge is (d,c). Hence, $v^*_6 = c$ and d is the predecessor of d in the shortest path.

We therefore have the following algorithm (framework), in which we follow above formula to iteratively construct R_k, $k = 1, 2, \cdots, n$. We again use array $dist$ of size n to store the distance for vertices in R_k.

```
Algorithm Dijkstra-Framework ($G = (V,E), s \in V$)
    let $R_1 = \{s\}$;
    $dist[s] = 0$; $dist[v] = \infty$ for any $v \neq s$;
    for $k = 1 \to n-1$
        calculate $(u', v') := \arg\min_{u \in R_k, v \in V \setminus R_k, (u,v) \in E}(dist[u] + l(u,v))$;
        $R_{k+1} = R_k \cup \{v'\}$;
        $dist[v'] = dist[u'] + l(u', v')$;
    end for;
end algorithm;
```

A naive implementation of above framework takes $O(|V| \cdot |E|)$ time, as for each k, the calculation of the minimization may take $O(|E|)$ time. Next lecture, we will show how to use efficient data structures to speed up, leading to the complete Dijkstra's algorithm with improved running time.