Dijkstra's Algorithm (continued)

We rewrite the formula in Corollary 1 (last Lecture) into an equivalent form by separating the minimization into two levels:

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

Now we define the inner level of minimization with a new term $dist_k(v)$:

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

Then we have

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

Intuitively, $dist_k(v)$ gives the length of the shortest path from s to v by only using vertices in R_k. This means that $dist_k(v)$ is always an upper bound of the distance, formally $dist_k(v) \geq distance(s,v)$. This is because, $dist_k(v)$ represents the length of the optimal path of a subset of all possible paths from s to v, hence giving an upper bound.

With $dist_k$ available, the next closest vertex, i.e., v^*_{k+1} can be easily calculated by picking the one with smallest $dist_k$ value. And the distance is equal to the corresponding dist value: $distance(s, v^*_{k+1}) = dist_k(v^*_{k+1})$.

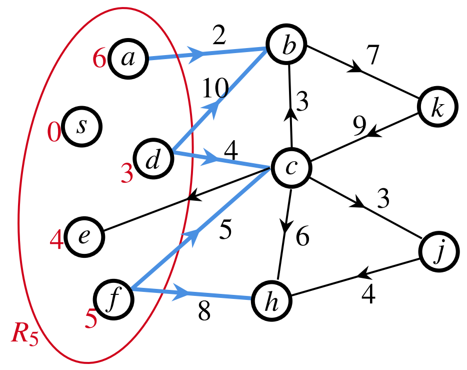

Figure 1: Assume that we already know $R_5 = (s,d,e,f,a)$; the distance to each vertex in R_5 is marked in red. Try above formulas in Figure 1. Answer: $dist_5(b) = \min\{6+2, 3+10\} = 8$, $dist_5(c) = \min\{3+4, 5+5\} = 7$, $dist_5(h) = 5+8 = 13$, $dist_5(k) = \infty$, $dist_5(j) = \infty$; hence $v^*_6 = c$ and $distance(s, v^*_6) = dist_5(c) = 7$.

The reason we introduce $dist_k$ is that, $dist_{k+1}$ can be calculated in an incremental way by largely reusing $dist_k$. This is much more efficiently than directly calculating $dist_{k+1}$ using the definition. To see the details, recall its definition, for any $v \in V \setminus R_{k+1}$, we have

$$dist_{k+1}(v) = \min_{u \in R_{k+1}, (u,v) \in E} (distance(s,u) + l(u,v)).$$

Note that $R_{k+1} = R_k \cup \{v^*_{k+1}\}$. Hence

$$dist_{k+1}(v) = \begin{cases} \min\{dist_k(v), distance(s, v^*_{k+1}) + l(v^*_{k+1}, v)\} & \text{if } (v^*_{k+1}, v) \in E \\ dist_k(v) & \text{if } (v^*_{k+1}, v) \notin E \end{cases}$$

In other words, when calculating $dist_{k+1}$, we only need to examine the out-edges of v^*_{k+1} and update only if the use of v^*_{k+1} leads to a shorter path. The pseudo-code of calculating $dist_{k+1}$ from $dist_k$ is given below.

Before calling this updating procedure, we assume $dist_{k+1}$ is initialized as $dist_k$; that is the update-dist procedure below just update for vertices whose dist values might be reduced. (In the complete algorithm, you will see that we will use one *dist* array for all k, hence such initialization is not necessary.)

```
procedure update-dist ($v^*_{k+1}$)
    for $(v^*_{k+1}, v) \in E$
        if $(dist_k(v^*_{k+1}) + l(v^*_{k+1}, v) < dist_{k+1}(v))$
            $dist_{k+1}(v) = dist_k(v^*_{k+1}) + l(v^*_{k+1}, v)$;
        end if;
    end for;
end procedure;
```

Try above procedure with the example below.

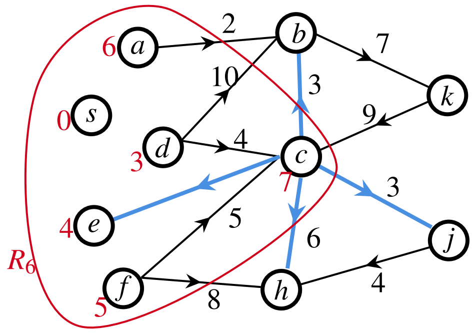

Figure 2: Following Figure 1, we have that $v^*_6 = c$. We now want to calculate $dist_6$ using $dist_5$. We consider the out-edges of c, marked as thick blue edges. We have, $dist_6(b) = \min\{dist_5(b), 7+3\} = \min\{8, 10\} = 8$, $dist_6(h) = \min\{dist_5(h), 7+6\} = \min\{13, 13\} = 13$, $dist_6(k) = dist_5(k) = \infty$, $dist_6(j) = \min\{dist_5(j), 7+3\} = \min\{\infty, 10\} = 10$.

The last piece of Dijkstra's algorithm comes with the use of *priority queue* to quickly pick the next closest vertex, i.e., to calculate $v^*_{k+1} = \arg\min_{v \in V \setminus R_k} dist_k(v)$. To this end, the priority queue PQ always stores $V \setminus R_k$, and for each vertex v that is stored in PQ, its priority is $dist_k(v)$. In this way, every time we call find-min (PQ), it gives us $\min_{v \in V \setminus R_k} dist_k(v)$.

The pseudo-code for complete Dijkstra's algorithm is given below. First, instead of maintaining a $dist_k$ array for each separate k, we just need to maintain a single *dist* array (the index k will increase implicitly when the next closest vertex gets identified and removed from the priority-queue). Second, we do not need to implicitly maintain R_k, as PQ is always complement to R_k. In order to maintain this invariant, we delete the next closed vertex u, by calling delete-min, at the time of identifying u. In the update-list procedure, in order to guarantee that the priority of v is always $dist(v)$, we call decrease-key every time we update $dist(v)$.

Recall that Dijkstra's algorithm sequentially identifies the closests vertices from s. Formally, if u is removed from PQ before v, then $distance(s,u) \le distance(s,v)$.

Where are the final distances from s? They are in array *dist*. This is because, at the time a vertex u is picked by find-min and removed from PQ, *dist* value for this vertex u is exactly its distance, i.e., $dist(u) = distance(s,u)$. This dist value for u will remain unchanged till the end of the algorithm. Notice though, later in the algorithm, when v is picked by find-min (and removed from PQ), there might be an edge (v,u) and therefore $dist(v) + l(v,u)$ will be compared with $dist(u)$ according to the algorithm, and if the former is smaller than the later, $dist(u)$ will be reduced/changed. (See an example in Figure 2, edge (c,e).) But such change will never happen. This is because, u is picked before v, implying that $distance(s,u) \leq distance(s,v)$. Hence $dist(v) + l(v,u) = distance(s,v) + l(v,u) > distance(s,v) \geq distance(s,u) = dist(u)$. So, the update to $dist(u)$ will not happen.

Another way to understand why such update to $dist(u)$ will not happen is that $dist(u)$ is always an upper bound of $distance(s,u)$. When u gets removed from PQ, this bound is reached, i.e., $dist(u) = distance(s,u)$. Hence after that $dist(u)$ will remain this minimized value and hence cannot get further reduced.

```
Algorithm Dijkstra ($G = (V,E)$, $l(e)$ for any $e \in E$, $s \in V$)
    $dist[v] = \infty$, for any $v \in V$;
    init an empty priority queue $PQ$;
    for any $v \in V$: insert ($PQ$, $v$), where the priority of $v$ is $\infty$;
    $dist[s] = 0$;
    decrease-key ($PQ, s, 0$);
    while (empty ($PQ$) = false)
        $u$ = find-min ($PQ$);
        delete-min ($PQ$);
        for each edge $(u,v) \in E$
            if ($dist[v] > dist[u] + l(u,v)$)
                $dist[v] = dist[u] + l(u,v)$;
                decrease-key ($PQ$, $v$, $dist[v]$);
            end if;
        end for;
    end while;
end algorithm;
```

The running time of Dijkstra's algorithmd depends on the specific implementation of priority queue used. Consider using binary heap. The break-down of running time is given below. Note that each vertex will be picked from the PQ at most once and each edge will be examined at most once (for directed graph) or at most twice (for undirected graph). The total running time is $O((|V| + |E|) \log |V|)$.

1. initialization: $\Theta(|V|)$;
2. insert (PQ): $|V| \times O(\log |V|)$;
3. empty (PQ): $|V| \times \Theta(1)$;
4. find-min (PQ): $|V| \times \Theta(1)$;
5. delete-min (PQ): $|V| \times O(\log |V|)$;
6. updating-dist: $|E| \times \Theta(1)$;
7. decrease-key (PQ): $|E| \times O(\log |V|)$;

Bellman-Ford Algorithm

Bellman-Ford algorithm can be used to solve the (single-source) shortest path problem with negative edge length, and its extension can also be used to detect if a graph contains negative cycle (reachable from the given source).

Bellman-Ford algorithm is quite simple. It only maintain an array, *dist* of size $|V|$, as its data structure. And it just does a bunch of "update" operations. An "update" function takes an edge $e = (u, v)$ as input, and updates $dist[v]$ as $dist[u] + l(u, v)$ if the former is larger than the latter.

```
procedure update(edge $(u,v) \in E$)
    if ($dist[v] > dist[u] + l(u,v)$)
        $dist[v] = dist[u] + l(u,v)$;
    end if;
end procedure;
```

Bellman-Ford algorithm iterates $(|V| - 1)$ rounds, and in each round, updates *all* edges, in an arbitrary order.

```
Algorithm Bellman-Ford ($G = (V,E)$, $l(e)$ for any $e \in E$, $s \in V$)
    init an array $dist$ of size $|V|$;
    $dist[s] = 0$; $dist[v] = \infty$ for any $v \neq s$;
    for $k = 1 \to |V| - 1$
        for each edge $(u,v) \in E$
            $update(u,v)$;
        end for;
    end for;
end algorithm;
```

Since update function takes constant time, clearly, Bellman-Ford algorithm runs in $\Theta(|V| \cdot |E|)$ time.