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.

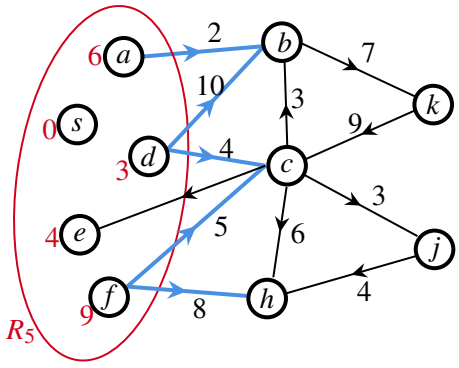

Figure 1: Try above formulas in Figure 1. Answer: $dist_5(b) = \min\{8, 13\} = 8$, $dist_5(c) = \min\{7, 14\} = 7$, $dist_5(h) = 17$, $dist_5(k) = \infty$, $dist_5(j) = \infty$; hence $v^*_6 = c$ and $distance(s, v^*_6) = dist_5(c) = 7$.

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. More importantly, we can design a more efficient procedure to calculate $dist_{k+1}$ in the next iteration by largely reusing $dist_k$. To see that, 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

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

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, $dist_{k+1}$ is first copied from $dist_k$.

```
procedure update-dist (dist_k, v*_{k+1})
    for (v*_{k+1}, v) ∈ E
        if (dist_k(v) > distance(s, v*_{k+1}) + l(v*_{k+1}, v))
            dist_{k+1}(v) = distance(s, v*_{k+1}) + l(v*_{k+1}, v);
        end if;
    end for;
end procedure;
```

Try above procedure with the example below.

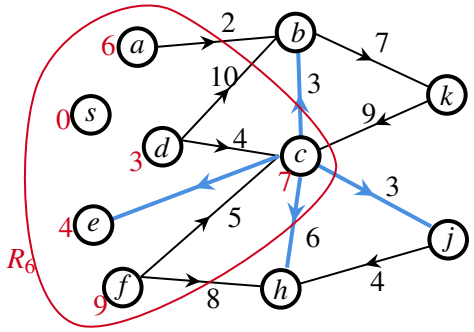

Figure 2: Following Figure 1, we have that $v_6^* = c$. We now want to calculate $dist_6$ using $dist_5$. We consider the out-edges of c, marked as thick blue edges. Eventually, $dist_6(b) = 8$, $dist_6(h) = 13$, $dist_6(k) = \infty$, and $dist_6(j) = 10$.

The above procedure enables fast calculation of $dist_k$. 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 $\min_{v \in V \setminus R_k} dist_k(v)$. In the complete Dijkstra's algorithm, we don't need to implicitly maintain R_k, as PQ is always complement to R_k. In order to maintain this invariant, we delete v_{k+1}^* from PQ, by calling delete-min, at the time of adding v_{k+1}^* to R_{k+1}. In order to guarantee that the priority of v is always $dist_k(v)$, we call decrease-key every time we update $dist_k$ of a vertex.

```
Algorithm Dijkstra (G = (V,E), l(e) for any e ∈ E, s ∈ V)
    dist[v] = ∞, for any v ∈ V;
    init an empty priority queue PQ;
    for any v ∈ V: insert (PQ, v), where the priority of v is ∞;
    dist[s] = 0;
    decrease-key (PQ, s, 0);
    while (empty (PQ) = false)
        u = find-min (PQ);
        delete-min (PQ);
        for each edge (u,v) ∈ 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 pseudo-code for complete Dijkstra's algorithm is given above. We use array $dist$ of size n to store $dist_k$. Where are the distances for those vertices in R_k (i.e., those are not in PQ) stored? They are in array $dist$ as

well. This is because, at the time v^*_{k+1} is identified and added to R_{k+1} (i.e., removed from PQ), $dist_k$ value for this vertex is exactly its distance. In fact, at any time of the algorithm, $dist[v] = distance(s,v)$ for any vertex v that is not in PQ. Finally, the algorithm don't explicitly maintain an index k: this index implicitly increases in every iteration of the while loop.

Here are some facts about Dijkstra's algorithm. First, $dist[v]$ is always an upper bound of the distance.

Fact 1. At any time of the algorithm, $dist[v] \geq distance(s,v)$.

This is because, by definition, $dist_k(v)$ is the length of shortest path from s to v using only vertices in R_k. In other words, $dist_k(v)$ represents the length of the optimal path of a subset (of all possible paths from s to v).

Second, once v is removed from PQ, $dist[v]$ won't change and remain as $distance(s,v)$.

Fact 2. At any time of the algorithm, if v is not in PQ, then $dist[v] = distance(s,v)$.

This is because, at the time v is deleted from PQ, $dist[v] = distance(s,v)$. Following Fact 1, it will remain this minimized value. Notice though, in Dijkstra's algorithm, a vertex v not in PQ might be "touched" by an edge (v^*_{k+1}, v), where v^*_{k+1} is the newly determined closest vertex, but the actual update will not happen. See an example in Figure 2 of Lecture 17, edge (c,e).

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 $\Theta((|V|+|E|)\log|V|)$.

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

Properties of Shortest Path Problem

To prepare to solve the shortest path problem with negative edge length, we first see some properties.

A negative cycle C in a graph is a cycle with negative length, i.e., $l(C) := \sum_{e \in C} l(e) < 0$. In the presence of negative cycle, if we don't limit the number of edges in a path, then the length of a path could goes to negative infinity. In other words, the shortest path may not exist. Therefore, in a graph with negative edge length, we want to detect if there exists negative cycle. We will show an algorithm (the Bellman-Ford algorithm) can be used to detect negative cycles.

A path p in a graph is *simple* if p does not have repeating vertices. If a graph G does not contain negative cycle, then for any pair of vertices u and v, if u can reach v, then there always exists a simple shortest path

from u to v, as otherwise we can skip the cycle in it to get a better or same-length path. If all cycles in graph G are positive then every shortest path is simple.

Property 1. If G does not contain negative cycles, for every $u, v \in V$, there exists a shortes path from u to v with at most $(|V|-1)$ edges.

Shortest path admits the following *optimal substructure* property. Intuitively, this property states that, the shortest path from u to v contains the shortest path from u to any internal vertex on this path (formally described below). Essentially, this is why shortest path problem can be solved efficiently.

Property 2. Let $p = (v_1, v_2) \to (v_2, v_3) \to \cdots \to (v_{k-1}, v_k)$ be the shortest path from v_1 to v_k. Then for any $1 \le i \le k$, $p_i := (v_1, v_2) \to (v_2, v_3) \to \cdots \to (v_{i-1}, v_i)$, i.e., the portion of p from v_1 to v_i, is the shortest path from v_1 to v_i.

Proof. Suppose that $p_i = (v_1, v_2) \to (v_2, v_3) \to \cdots \to (v_{i-1}, v_i)$ is not the shortest path from v_1 to v_i. Assume that q is the shorest path from v_1 to v_i. Then we can construct a path from v_1 to v_k shorter than p, by concatenating q and $(v_i, v_{i+1}) \to \cdots \to (v_{k-1}, v_k)$. This contradicts to the fact that p is the shortest path from v_1 to v_k. □

Note that the above property holds even for graphs with negative edge length (as in the proof we don't assume anything about edge length). This property also immediately implies the following fact.

Property 3. If we know that there exists one shortest path from s to v such that (w, v) is the last edge on this shortest path, then we have that $distanct(s, v) = distance(s, w) + l(w, v)$.