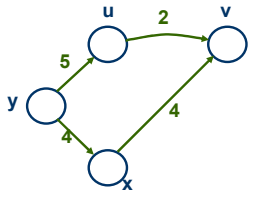
Tirgul 12

(and more)

sample questions

Q. We've seen that solving the shortest paths problem requires O(VE) time using the Belman-Ford algorithm or O((V+E)logV) using the Dijkstra algorithm (might be even O(VlogV+E) when using Fibonacci heaps). Suppose you know that your graph is a DAG, can you do better?



A. Yep, the algorithm is:

```
                Dag-Shortest-Paths(G,w,s)
Θ(|V+E|)    1. topologically sort the vertices of G
Θ(|V|)      2. Initialize-Single-Source(G,s)
Θ(|E|)      3. For each vertex u (taken in topologically sorted order)   } (Θ(|V|))
            4.    For each vertex v ∈ Adj[u]
            5.          Relax(u,v,w)
```

This procedure runs in Θ(V+E)

We first run topological sort (O(|V+E|))

Reminder:

A ***topological sort*** takes a directed acyclic graph, or ***DAG***, and returns an ordered list of the vertices such that if there is an edge (v,u) in the graph, then v will appear before u in the list.

- If there are a few paths from s to v, we will relax all the vertices along these paths before calculating the distance from $v \rightarrow$ When we relax the distances of all the neighbors of v, the distance to v is the shortest path (since we relax vertices in topologically ordered manner, we will relax all vertices that have incoming edges to v, before relaxing all those reachable from v)

(a bit) More formally:

We want to prove: The algorithm calculates the shortest paths between the source vertex and all other vertices

- The proof of correctness is by induction on the length of the number of nodes on the shortest path between s and a node v: $p=(v_0,..,v_k)$ with $v_0 = s$, $v_k = v$.
- We run through the nodes in topological order: the edges of p are relaxed in the order of the path (we did not have this property in Bellman-Ford).
- Assuming inductively $d[v_{i-1}]=\delta[s, v_{i-1}]$ when the for loop reached v_{i-1} the relaxation of (v_{i-1}, v_i) has the effect of setting $d[v_i]=D[s, v_i]$.
- Initially $d[v_0]=0=D[s,s]$.

Modified MST

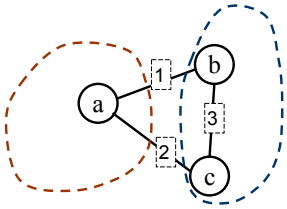
Your friend suggests an improved recursive MST algorithm:

Given a graph G = (V,E), partition the set V of vertices into two sets V_1 and V_2 such that $|V_1|$ and $|V_2|$ differ by at most 1. Let E_1 be the set of edges that are incident only on vertices in V_1, and let E_2 be the set of edges that are incident only on vertices in V_2. Recursively solve a minimum-spanning tree problem on each of the two subgraphs $G_1 = (V_1,E_1)$ and $G_2 = (V_2,E_2)$. Finally, select the minimum-weight edge in E that crosses the cut (V1; V2), and use this edge to unite the resulting two minimum spanning trees into a single spanning tree.

Will it work?

Modified MST

No, it won't work, counter example:



Most Reliable path

Assume you are given a directed graph $G = (V, E)$ on which each edge $(u, v) \in E$ has an associated value $R(u, v)$ which is a real number in the range $0 \leq R(u, v) \leq 1$ that represents the reliability of a communication channel from vertex u to vertex v. Interpret $R(u, v)$ to be the probability that the channel from u to v will not fail, and assume that these probabilities are independent. Give an efficient algorithm to find the most reliable path between two given vertices.

Most Reliable path

1. Modify the initialization routine to initialize the reliability estimate (formerly the distance estimate), $d[u]$, to zero for each vertex
2. Let vertex u be the source vertex and assign $d[u] = 1$
3. Modify the RELAX routine to maximize the reliability estimate, where the reliability estimate estimate along any path is the product of the reliabilities of each link (due to independent probabilities). Relaxing an edge (a; b) will update the reliability estimate $d[b] = \max(d[b]; d[a]*R(a,b))$
4. Use a priority queue based on a Max-Heap (most reliable vertex on top)

Counting paths

Q. Give an efficient algorithm to count the total number of paths in a *DAG* G between a source vertex s and each other vertex

A. The algorithm exploits the fact that G is a DAG and can therefore be topologically sorted.

1. Set the path counts of all vertices to 0, and set s's path count to one.
2. Topologically sort the vertices.
3. For each vertex u in topological order, do the following:
4. For each neighbor v of u, add u's path count to v's path count.

Counting paths

Correctness: Since the vertices are topologically ordered, a vertex v is not reachable until the path count to all vertices $u \in V$ such that $(u, v) \in E$ have been updated.

(The proof is by induction on the topological order of the vertices)

Prim with adjacency matrix

Reminder: Prim's algorithm (G, w, v)

```
Q ← V[G]                          ← build heap O(|V|)
for each u in Q do                } O(|V|)
    key [u] ← ∞
key [r] ← 0
π[r] ← NIl
while queue is not empty do       } O(|V|log|V|)
    u ← EXTRACT_MIN (Q)
for each v in Adj[u] do           } loop executed a total of O(|E|) times
    if v is in Q and w(u, v) < key [v]
        then π[v] ← w(u, v)
            key [v] ← w(u, v)     ← decrease_key O(log|V|)
```

Prim with adjacency matrix

The time complexity of the algorithm using a min heap is $O(|V|\log|V| + |E|\log|V|) = O(|E|\log|V|)$

We will now see a variant that works with graphs represented using adjacency matrices in $O(|V|^2)$ time

Prim with adjacency matrix

Reminder: Prim's algorithm (G, w, v)

```
A ← V[g] // Array        ← build an array Θ(|V|)
For each vertex u in A do
   key [u]= ∞            } Θ(|V|)
key [r] ← 0
π[r] ← NIL
while Array A is not empty do
    scan over A find the node u with smallest      } O(|V|)
    key, and remove it from array A
    for each vertex v in Adj[u]                    } *|V|
        if v in A and w[u,v] < key[v] then         } O(|V|)
            π[v] ← u
            key[v] ← w[u,v]
```