4/18

Single-source shortest Path:

Problem: Given a directed graph $G=(V,E)$ with weight(length) associated with the edges, find shortest paths from v to all other vertices of G.

Here we talk about lengths instead of weights, since the problem traditionally is called the shortest path (rather than the lightest path problem). We sometime call weight costs as well.

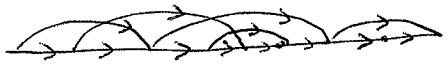
The length of a path is the sum of the lengths of its edges.

★First DAGS:

We know how we can compute a topological sorting of a DAG in $O(|V|+|E|)$ with this feature:

(★) if z is a vertex with label k, then 1) there are no paths from z to vertices with labels $<k$

2) there are no paths from vertices with labels $>k$ to z.



Thus without loss of generality, we assume v is the vertex with label 1.

Now we use induction as follows:

Induction hypothesis: Given a topological ordering, we know how to find the lengths of the shortest paths from v to the first $n-1$ vertices.

Thus, we remove the nth vertex, we solve the reduced problem by induction, then take the minimum of the values $sp[w]+length(w,z)$ over all edges $(w,z)\in E$ as the value of $sp[z]$.

The algorithm can be written with ~~shortest~~ for loops only:

```
Algorithm shortest_path(G,V);
begin sp[v]=0; sp[w]=∞ for w≠v;
   for i=1 to n do
      for all w such that (w,z) ∈ E do
         if sp[w]+length(w,z) < sp[z] then
            sp[z] := sp[w]+length(w,z);
end;
```

Time complexity: Topological sort needs $O(|V|+|E|)$. We check every vertex and every edge only once. Thus the total running time is $O(|V|+|E|)$.

★The general directed graphs: Note we can replace each edge u—v with two directed edges $u \rightleftarrows v$. Thus directed graphs are more general than undirected graphs. This fact is usually true. Since (★) above is not correct in general directed graphs, the IH above does not work but a similar idea works:

IH: Given a graph and a vertex v, we know the k vertices that are closest to v and the lengths of the shortest paths to them.

Denote the set containing v and k closest vertices to v by V_k. The problem is to find a vertex w that is closest to v among the vertices not in V_k, and to find the shortest path from v to w can go through only the vertices in V_k. It cannot include vertices not in V_k

since they would then be closer to v than w. Therefore to find w, it is sufficient to consider only edges connecting vertices from V_k to vertices not in V_k; all other edges can be ignored for now. Let (u,z) be an edge such that u is in V_k and z is not. Such an edge corresponding to a path from v to z, which consists of the shortest path from v to u (already known by induction) and the edge (u,z). We only need to compare all such paths, and take the shortest among them.

The algorithm implied by the induction hypothesis is the following. At each iteration, a new vertex w is added such that $\min_{u \in V_k}(sp[u] + length(u,w))$ is the minimal over all w not in V_k. By the argument above, w is indeed the (k+1)-th closest vertex to v; thus adding it extends the induction hypothesis. This greedy algorithm is known as Dijkstra's algorithm.

```
Algorithm Dijkstra(G,v);
begin sp[v]=0; (sp[w]=∞ and mark[w]=false) for all w≠v
    while there exists an unmarked vertex do
        let w be an unmarked vertex such that sp[w] is minimal;
        mark[w]:=true;
        for all edges (w,z) such that z is unmarked do
            if sp[w]+length(w,z)<sp[z] then
                sp[z]=sp[w]+length(w,z)
end;
```

Implementation and complexity: A heap is a good data structure for finding minimum elements and updating lengths of elements. We always keep all vertices not yet in V_k in the heap. Initially all vertices there with v is on top (all other vertices have shortest path ∞). We find the minimum easily and delete it in $O(\log|V|)$ with total $O(|V|\log|V|)$. Say w taking the minimum. Now we update all $(w,z) \in E$. If sp[z] is updated and changed, z's position may change in the heap. First we need to locate z in the heap by another data structure, say an array with pointers to their location in the heap. We can access z in the heap in O(1) and update its position in the heap by exchanging and moving up until its appropriate position is found (note that the path length only decrease). This is essentially the same as heap insert and can be done in $O(\log|V|)$. Since there are at most |E| updates and each takes $O(\log|V|)$ leading to $O(|E|\log|V|)$ comparisons in the heap. Hence the total running is $O((|V|+|E|)\log|V|)$ instead of $O(|V|+|E|)$ that we had for DAGs. All edges selected in the process (from z to its parent w) form a shortest path tree (similar to BFS-tree).