1 Introduction

In this lecture we will look at algorithms for Single-Source Shortest Paths problem.

2 The Problem

Given a directed graph $G = (V, E)$ with weights (length) associated with the edges, find the shortest paths from $v$ to all the other vertices of $G$. Here we talk about lengths instead of weights, since the problem is traditionally called as the shortest path (rather than the lightest path) problem. The length of a path is the sum of lengths of its edges.

3 Solution for DAGs

We know how to compute a topological sort for a DAG in $O(|V| + |E|)$ time. 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 that $v$ 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.
Algorithm 1 Shortest-Paths for DAGs

1: \( \text{SP}[v] = 0; \)
2: \( \text{SP}[w] = \infty \) for \( w \neq v; \)
3: for \( i = 1 \) to \( n \) do
4: for all \( w \) such that \( (w, z) \in E \) do
5: if \( \text{SP}[w] + \text{length}(w, z) < \text{SP}[z] \) then
6: \( \text{SP}[z] := \text{SP}[w] + \text{length}(w, z); \)

We remove the \( n \)-th vertex and solve the reduced problem by induction, then take the min of values \( \text{SP}[w] + \text{length}(w, z) \) over all edges \( (w, z) \in E \) as the value of \( \text{SP}[z] \). The algorithm can be written using only “for” loops.

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

4 Solution for General Directed Graphs

Note that we can replace each edge \( uv \) with bidirected edges. This directed graphs are more general than undirected graphs. This fact is usually true. Since general graphs may not have a topological order, the same induction hypothesis as before does not work. A similar idea works however.

**Induction Hypothesis**

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 the \( k \) closest vertices to \( v \) by \( v_k \). The problem is to find a vertex \( w \) that is closest to \( v \) from among the vertices not in \( v_k \), and to find the shortest path from \( v \) to \( w \) that can go through only the vertices in \( v_k \). It cannot included vertices outside of \( v_k \) since they would 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.

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.

**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 are there with \( v \) on the top (all other vertices have shortest paths \( \infty \)). We find the minimum easily.
Algorithm 2 Dijkstra(G,v)

1: \( \text{SP}[v] = 0; \)
2: \((\text{SP}[w] = \infty \text{ and mark}[w]=\text{false}) \text{ for } w \neq v;\)
3: \(\text{while There exists an unmarked vertex do}\)
4: \(\text{Let } w \text{ be an unmarked vertex such that } \text{SP}[w] \text{ is minimal};\)
5: \(\text{mark}[w]=\text{true};\)
6: \(\text{for All edges } (w, z) \text{ such that } z \text{ is unmarked do}\)
7: \(\text{if } \text{SP}[w] + \text{length}(w, z) < \text{SP}[z] \text{ then}\)
8: \(\text{SP}[z] := \text{SP}[w] + \text{length}(w, z);\)

and delete it in \(O(\log |V|)\) with total \(O(|V| \log |V|)\). Say \(w\) is the minimum. Now we update all \((w, z) \in E\). If \(\text{SP}[z]\) is updated and changed, then the position of \(z\) 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 lengths 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 \(|E| \log |V|\) comparisons in the heap. Hence the total running time is \(O\left(|V| + |E| \log |V|\right)\) instead of \(O(|V| + |E|)\) that we had for DAGs. All edges selected in the process (from \(z\) to its parent \(w\)) form the Shortest-Path-Tree (similar to BFS-Tree and DFS-Tree).

References