Miscellaneous Graph Algorithms

1 Counting Length k Paths

Given an unweighted graph with n vertices, how many paths of length k are there from one vertex to another? First, let us represent the graph by an adjacency matrix A where A_{ij} is equal to the number of edges from i to j. Let us define $B = A^2$. Then

$$B_{ij} = \sum_{k=1}^{n} A_{ik} * A_{kj} = \sum_{k=1}^{n} (\text{\# of length 1 path from i to k}) * (\text{\# of length 1 path from k to j})$$

Thus, B_{ij} is the number of length 2 paths from i to j! Continuing in this fashion, the answer we desire can be found by computing A^k! Using the method of repeated squaring, this can be found in $O(n^3 * log(k))$. An example of using this idea is to answer the number of triangles in the graph, which can be found by summing up the diagonals of A^3 and dividing by 3.

2 Euler Path/Cycle

An **Euler Path** is a path in the graph which uses each edges exactly once. An **Euler Cycle** is a special type of Euler path where the starting and the ending vertex of the path are the same. The verification of the existence can be done in linear time using the following theorems (in both of the theorem, we assume the graph is **connected**):

Theorem 2.1. *Given an undirected graph G = (V, E), define the degree of a vertex u to be the number of edges connected to u. Then an Euler cycle exists iff every vertex has even degree. An Euler path exists iff exactly two vertices has odd degree. In this case the start and ending vertex of the path are the two odd degree vertices.*

Theorem 2.2. *Given an directed graph G = (V, E), define the degree of vertex u to be the number of outgoing edges from u minus the number of incoming edges to u. Then an Euler cycle exists iff every vertex has degree 0. An Euler path exists iff exactly one vertex a has degree 1 and exactly one vertex b has degree -1. In this case, the Euler path must start from a and end at b.*

The above theorems gives the conditions for the existence of Euler path/cycle. However, it does not tell us how to construct the Euler path/cycle. Algorithms do exists to construct the Euler path/cycle, but we will not go into the details here. The main idea is to continuously remove edges to form cycles until there are none left. For more detail, Google **Fleury's Algorithm**.

3 Johnson's Reweighting Algorithm

Suppose we want to compute the all-pair shortest path for a sparse graph with negative weight edges. We can definitely use the Floyd Warshall algorithm, but the time complexity if $O(V^3)$. Instead, we can try **reweighting** the graph so that every edge weight is positive. After reweighting, we can then run Dijkstra's algorithm from each vertex to find all-pair shortest path. The question is then, how do we reweight the graph so that shortest path remain unchanged?

3.1 Shortest Path Preserving Reweighting

The first idea is to find the most negative weight edge in the graph and scale it up to 0 weight. At the same time, we will also increase every other edge by the same amount. While this may look sound, it does not preserve shortest path - the shortest path in the original graph may not be the same as the one in the reweighted graph. This is because a path with more edge will be ”penalized” more than a path with less edges.

Instead, we define a height function $h : V \rightarrow Z$ on each vertex. Furthermore, let us assume the height function satisfies that for any edge (u, v), $w(u, v) + h(u) - h(v) \geq 0$, where $w(u, v)$ is the cost of edge (u, v). For now, let us assume we have such a function available to us (we will worry about finding the function later). Given such a function, let us reweight each edge (u, v) such that the reweighted cost, $\overline{w}(u, v)$, is equal to $w(u, v) + h(u) - h(v)$.

By definition of h, it is clear that the reweighted graph has no negative weight edge. Now, let us consider a path P from u to v. It is easy to see that if $cost(P)$ is the cost of the path in the original graph and $\overline{cost}(P)$ is the cost in the reweighted graph, then $\overline{cost}(P) = cost(P) + h(u) - h(v)$. Thus, every path from u to v are scaled up by the same amount. So the shortest path from u to v in the original graph must be the same as in the reweighted graph. In fact, if P is the shortest path from u to v, we can recover $cost(P)$ from $\overline{cost}(P)$ by rearranging the equation $cost(P) = \overline{cost}(P) - h(u) + h(v)$.

3.2 Finding the Height Function

The only thing that remains is to find the height function itself. Rearranging the equations, we get that for every edge (u, v), $h(v) \leq w(u, v) + h(u)$. This equation should look familiar because it showed up in the `relax(u, v)` function when we are talking about shortest path! In fact, we can define h based on shortest path!

Let us add a node S to the graph and connect S to every vertex with a weight 0 edge. Let us define $h(u)$ to be the cost of the shortest path from S to u. Then by definition of shortest path, we would get $h(v) \leq w(u, v) + h(u), \forall (u, v) \in E$. Thus, to obtain the height function, all we need is to run Bellman Ford once. Note what we add a new node S and and all the 0 weight edges to make sure that $h(u) \leq 0$ for all vertices u; things get screwed up if $h(u) = \infty$.

3.3 Time Complexity

The reweighting algorithm requires us to run Bellman Ford once to find the height function, which take $O(VE)$. Reweighting each edges takes $O(E)$ and running a Dijkstra from each vertex takes $O(VElog(V))$. Finally, readjusting the cost of the shortest path from the reweighted graph to the original graph take $O(V^2)$. Overall, the time complexity is $O(VElog(V))$. For a sparse graph, this is much better than Floyd Warshall, which require $O(V^3)$. Of course, if the graph is dense, then running Bellman Ford itself is almost as costly as Floyd Warshall and we would be better off not using the reweighting algorithm.