CS5050 Advanced Algorithms

Philip Nelson

Spring Semester, 2018

Assignment 6: Graph Algorithms I

Worked with: Ammon H, Raul R, Hailee M, Kyle H, and Jack K

Note: In this assignment, we assume that all input graphs are represented by adjacency lists.

1. We say that a **directed** graph G is *strongly connected* if for every pair of vertices u and v, there exists a path from u to v and there also exists a path from v to u in G.

   Given a directed graph G of n vertices and m edges, let s be a vertex of G.

   (a) Design an $O(m + n)$ time algorithm to determine whether the following is true: there exists a path from s to v in G for all vertices v of G. **(10 points)**

   Solution

   To determine if there is a path from s to v for all v in G we can use a breadth first search. As nodes are visited, keep a count of how many are reachable from s. If the count is equal to the number of nodes in G then return true, else return false.

   A breadth first search can be accomplished in $O(m + n)$

   (b) Design an $O(m + n)$ time algorithm to determine whether the following is true: there exists a path from v to s in G for all vertices v of G. **(10 points)**

   Solution

   The method to determine whether there exists a path from all v in G to s is similar to the previous except we want to perform a breadth first search following the reverse direction of all the edges.

   In order to be able to follow the edges in the reverse direction, we need to construct the reverse adjacency list. This can be done by iterating through the adjacency list of each vertex and adding the vertex v to the new list of each vertex in its list. For example, if the list for vertex A is $[B \to D \to E]$, then add A to the reverse adjacency lists of B, D, and E. This can be done in $O(m+n)$.

   Once the reverse adjacency list is constructed we can use it to perform a breadth first search as described in part a. This takes $O(m+n)$. Together with construing the reverse adjacency list, the time complexity is $O(2(m + n)) \Rightarrow O(m + n)$.

(c) Prove: *G is strongly connected if and only if there exists a path in G from s to v and there is also a path from v to s for all vertices v of G.* **(10 points)**

Proof

If there exists a path in G from s to v and from v to s for all vertices v of G, then G is strongly connected. This is true because of the transitive property; if you can go from any vertex v to s, and from s you can go to any other vertex u, then you can always go from v to u through s and vice versa. In this fashion, you can start from any vertex and visit every other vertex which makes G strongly connected.

2. Given an undirected graph G of n vertices and m edges, suppose s and t are two vertices in G. We have already known that we can find a shortest path from s to t by using the breadth-first-search (BFS) algorithm. However, there might be multiple different shortest paths from s to t.

   Design an $O(m+n)$ time algorithm to compute the number of different shortest paths from s to t (your algorithm does not need to list all the shortest paths, but only report their number). (Hint: Modify the BFS algorithm.) **(20 points)**

Solution

To find the number of shortest paths from s to t, we will use a modified breadth first search. Begin by setting the path length at each node to ∞ and the path length at s to 0. We need another point of data, the number of paths that go through the node, set this value to zero, for all vertices and one for s. Then begin a breadth first search from s. When visiting a node for the first time, enqueue its unvisited neighbors. Update the path length of the neighbors to the current node's path length + 1, and add the number of paths from the vertex to the number of paths in the neighbor. However, when a node has already been visited, if the path length + 1 is equal to the neighboring nodes path length, just add the number of paths in the vertex to the paths in the neighbor. Since the paths have equal length through the vertex, it would not matter which pat was chosen, they are equivalent. When the queue is empty, the number of paths from s to t will be stored in t.

Since breadth first search can be accomplished in $O(m+n)$ therefore the time complexity is $O(m+n)$.

3. Given a directed-acyclic-graph (DAG) G of n vertices and m edges, let s and t be two vertices of G. There might be multiple different paths (not necessarily shortest paths) from s to t. Design an $O(m+n)$ time algorithm to compute the number of different paths in G from s to t. **(20 points)**

Solution

To count the paths from s to t in a DAG, we need to follow a topological order for G. Then we use a similar algorithm as described in problem 2. Label all nodes with the number of paths that pass through them, initialized to zero, with s starting at one. Then, beginning with s, add the paths from s to the paths of s's adjacent neighbors. Then repeat the process with the next node in the list until t is processed. t will contain the number of paths to it from s.

Following the topological order, all nodes and edges are used once, therefore the time complexity is $O(m+n)$

4. In class we have studied an algorithm for computing shortest paths in DAGs. Particularly, the *length* of a path is defined to be the total sum of the weights of all edges in the path. Sometimes we are not only interested in a shortest path but also care about the number of edges of the path.

Let G be a DAG (directed-acyclic-graph) of n vertices and m edges. Each edge (u,v) of E has a weight $w(u,v)$ (which can be positive, zero, or negative). Let k be a positive integer, which is given in the input. A path in G is called a *k-link* path if the path has **no more than** k edges. Let s and t be two vertices of G. A *k-link shortest path* from s to t is defined as a k-link path from s to t that has the minimum total sum of edge weights among all possible k-link s-to-t paths in G.

Design an $O(k(m+n))$ time algorithm to compute a k-link shortest path from s to t. For simplicity, you only need to return the length of the path and do not need to report the actual path. **(20 points)**

Solution

To find the *k-link shortest path* from s to t we can find the shortest paths of edge count e s.t. $1 \leq e \leq k$ using topological ordering. Once the shortest paths are known, take the shortest one. Since it takes $O(m+n)$ time to find the shortest path of length e and we will need to perform this operation k times, the complexity is $O(k(m+n))$.

To find the shortest path of edge length e, we will use the topological ordering and two data points, path length and edge number length. Initialize them to ∞ for all v and for s, 0 and 1 respectively. Starting with s, push the path length plus the adjoining edge weight from s to the adjacent vertices of s if the edge number length + 1 is less than or equal to the edge number length at the adjacent vertex, and the path length plus the adjoining edge length is less than the path length is less than the path length in the vertex. If at any point the edge length number is greater than e, stop pursuing the path. At the end, if there exists a path with e edges, its length will be stored in t.