به نام خدا

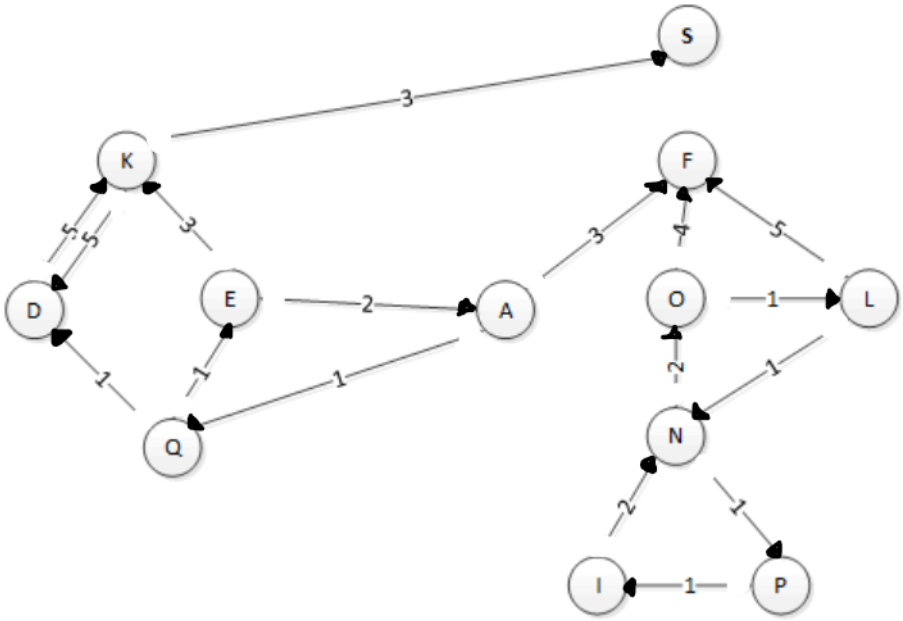
آنالیز الگوریتم ها
نیم‌سال دوم ۰۱-۰۲

دانشکده علوم ریاضی

تمرین۲ - امیرحسین محمودی - ۹۸۱۰۸۷۷۹

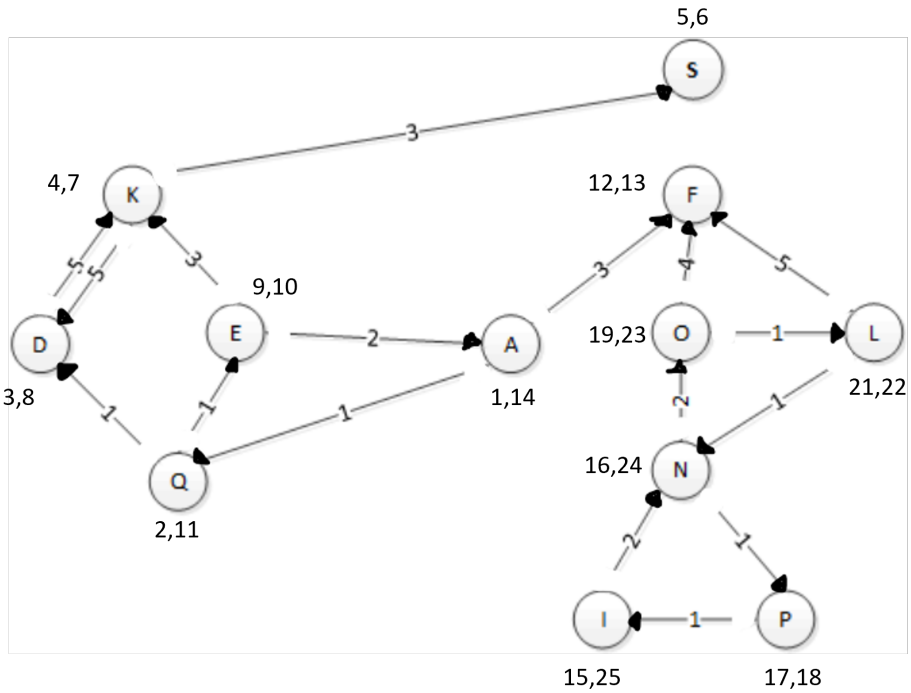
1

First we Make a graph reverse graph of G called G' by reversing the direction of G edges.



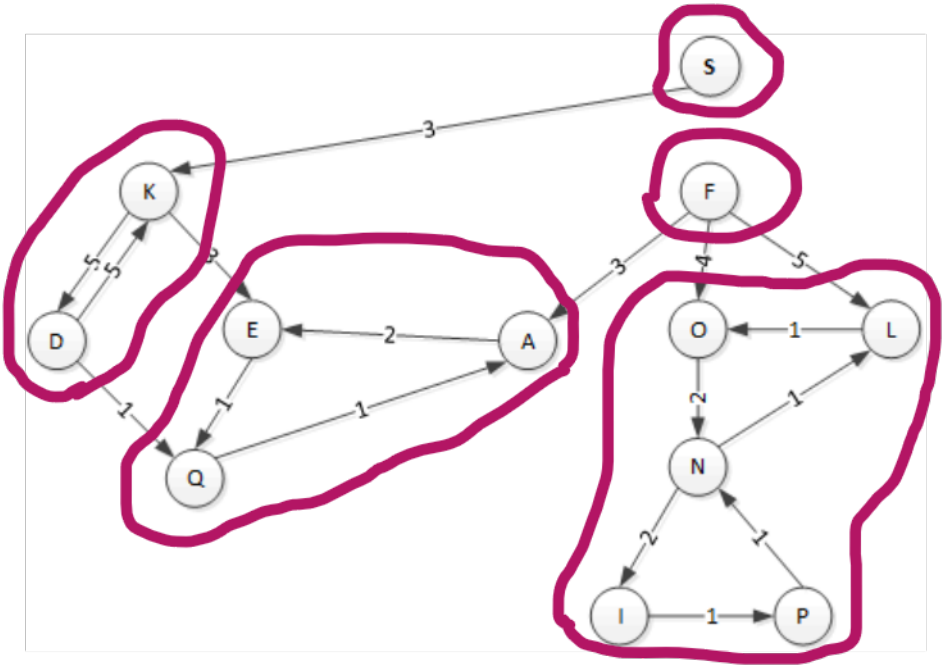
Then we run DFS on G' and insert each vertex on a stack after finishing it (or simply sort vertices by post number descending):

I,N,O,L,P,A,F,Q,E,D,K,S



now we start from the first node of this list and run DFS on the original graph starting from this vertex.the visited vertices would be strongly connected.

I:
I,P,N,L,O
A:
A,E,Q
F:
F
D:
D,K
S:
S



2

Suppose our graph is an undirected Tree.Depth of nodes and Levels are clearly defined in trees.

BFS traverse tree level by level and DFS traverse the tree by depth.both algorithms produce a tree with same root and containing all edges of the given tree.

We show that if the graph is a tree then every edge of it will be in DFS and BFS tree results.

Suppose there is an edge e which is in our graph but its not in the algorithm tree results.This edge has a vertex which is visited before (a back edge in DFS).but the BFS must have used it before.which is a contradiction.So the result trees contain all edges of the input tree.

also we can get this simply by a basic property of every tree:every tree has |V|-1 Edges.so both algorithms and the input must have the same number of edges.

Now we prove the other side: Suppose Both DFS and BFS algorithms are the same but input graph is not a tree.So there is an edge that belongs to input graph but it is not in the result tree.vertices of this edge must be ancestor and successor (back edge).also by BFS property the should only differ by one level.so this edge must be in the result tree which is contradiction.

also we look this in another way.suppose we have a cycle in our input graph. all vertices in this cycle will be on the same path starting from the first nodes visited in this cycle.In BFS tree these vertices will get in two branches when first node of this cycle get visited.(there maybe more branches after this.)So both trees cannot be the same which is a contradiction.

3

We will use dijkstra to find the shortest paths from source to sink.then we remove all edges between shortest pathes vertices.

But we need a small change to find all shortest paths.Instead of storing only a single node in each entry of prev[] we would store all nodes satisfying the relaxation condition. For example, if both r and source connect to target and both of them lie on different shortest paths through target (because the edge cost is the same in both cases), then we would add both r and source to prev[target]. When the algorithm completes, prev[] data structure will actually describe a graph that is a subset of the original graph with some edges removed. Its key property will be that if the algorithm was run with some starting node, then every path from that node to any other node in the new graph will be the shortest path between those nodes in the original graph, and all paths of that length from the original graph will be present in the new graph. Then to actually find all these shortest paths between two given nodes we would use a path finding algorithm on the new graph, such as depth-first search.

now we use dijkstra another time to find the shortest path in the remained graph.this is a shortest path which surely is beautiful.

If the shortest path contains no bad vertices then it's ok.

if there are some bad vertices in the new shorest path then there is no edge between a pair of them cause we deleted them.

Time complexity: T(dijkstra) + T(DFS) + T(deleting edges)

If we use Adjacency matrix we can do it in O($|V|^2$).it also can takes $O(|E| + |V|log|V|)$ if use binary heap.

4

Break G's long edges into unit-length pieces, by introducing "dummy" nodes.

To construct the new graph G', For any edge $e = (u, v)$ of E, replace it by l_e edges of length 1, by adding $l_e - 1$ dummy nodes between u and v.

Graph G' contains all the vertices V that interest us, and the distances between them are exactly the same as in G. Most importantly, the edges of G' all have unit length. Therefore, we can compute distances in G' by running BFS on G'. if we use adjacency matrix then we can add new edges in O(Lm) and BFS will takes O(Lm+n) sowe have:

O(Lm)+O(Lm+n)=O(Lm+n)=O(L(m+n))

5

A graph G has a Topological order if and only if it is a DAG.So if our graph is not a DAG we should construct is component graph by replacing SCC of input graph with a vertex.(each SCC is satisfied with our desired property so we don't need to check them.).

So we continue by assuming input graph is DAG.

a DAG is semi-connected if and only if these conditions hold which are equivalent:

- There is a single path that goes trough all vertices (Hamiltonian path);
- There is and edge between every consecutive vertices in the linearized graph.
- There is exactly one linearization of the DAG.

we prove the first one:

let a linearization order of n vertices be $v_1, v_2, ..., v_n$.

for any vertex v_i there is no path from v_{i+1} to it and there is only one path from v_i to v_{i+1} which is an edge.so this edge should be in the topological order for the graph to be semi-connected.So there are edges between all consecutive pairs of vertices in the linearized order,there is a Hamiltonian path in this graph.

conversely if there is a Hamiltonian path in our graph,there is a path between any pair of vertices which is a part of the Hamiltonian path.

The second and thirds are similar.we prove one of the which encompass the other one:

say $\prec$ is a Topological order.if two vertices a and b are consecutive(there is no c that $a \prec c \prec b$) and there is no edge between a and b then we can swap a and b and result is still a topological order. So our topological order wouldn't be unique.

Conversely if every pair of consecutive vertices is connected by an edge,then $\prec$ is unique.

If there is only one source in our graph G then that is the unique vertex to start topological order.we can then remove this vertex and then again we have one source which is a unique second vertex and so on.

So we construct the below algorithm which we just proved its correctness.

Algorithm 1: Determine-Semi-Connected(G=(V,E))

```
Find strongly connected components of G using Kosaraju(G)
Construct G' by Replacing each SCC with a vertex
Topological-Sort(G')
for ( i=0;i+1</V|;i++ ) {
    if V[i] and V[i+1] has edge then
        Return True
    end
    Return False
}
```

Time Complexity: T(Kosaraju)+T(Topological-Sort)+C|V|=O(|V|+|E|)

6

By Robbins theorem it is possible to choose a direction for each edge of an undirected graph G, turning it into a directed graph that has a path from every vertex to every other vertex, if and only if G is connected and has no bridge. for a connected graph with no bridge we run an DFS on it and direct edges from top to down.then for back-edges we we direct them to ancestors thus we get a strongly connected graph. for a graph with bridge we need a small change.we can check for bridge while running DFS and when we get one we direct that edge randomly cause it will make a new SCC anyway.this is a simply changed DFS which takes O(|E|+|V|).