Lab08-Shortest Path

CS214-Algorithm and Complexity, Xiaofeng Gao, Spring 2018.

* If there is any problem, please contact TA Jiahao Fan.
* Name:Juncheng Wan   Student ID:516021910620   Email: 578177149@qq.com

1. Let D be the shortest path matrix of weighted graph G. It means that $D[u, v]$ is the length of the shortest path from u to v for any pair of vertices u and v. Graph G and matrix D are given. Now, assume the weight of a particular edge e is decreased from w_e to w'_e. Design an algorithm to update matrix D with respect to this change. The time complexity of your algorithm should be $o(n^2)$. Describe the details and write down your algorithm in the form of pseudo-code.

Solution. Set G' the weighted graph deriving from G by decreasing w_{st} to w'_{st}, $D' = (d'_{ij})_{n\times n}$ the the shortest path matrix of weighted graph G'.

I think that if $d_{ij} \neq d'_{ij}$, the new shortest path should include e_{st}. Therefore, I have the following relationship:

$$d'_{ij} = min\{d_{ij}, d'_{is} + d'_{tj}, d'_{it} + d'_{sj}\} \tag{1}$$

Note that if there is not an edge between v_i and v_j, $d_{ij} = \infty$.

By using Dijkstra Algorithm, it costs $O(|V|log|V|+|E|)$ to the shortest paths starting from s or t. Updating D to D' costs $O(|V|^2)$ by the above equation. Therefore, the time complexity of this algorithm is $O(|V|log|V| + |E| + |V|^2) = O(|V|^2)$.

I give the pseudo-code as follows:

Algorithm 1: Updata-G

Input: : The shortest path matrix D; decreasing weight w'_{st} of one edge e_{st}.
Output: : The new shortest path matrix D'.

```
D' ← An empty n × n matrix ;
A_s, A_s^t, A_t, A_t^t ← An empty 1 × n array ;
A_s = DIJKSTRA(G, s) ;
A_t = DIJKSTRA(G, t) ;
G^t ← Inverse the direction of all edges of G ;
A_s^t = DIJKSTRA(G^t, s) ;
A_t^t = DIJKSTRA(G^t, t) ;
for i = 1 to n do
    for j = 1 to n do
        D'[i][j] = min{D[i][j], A_s^t[i] + A_t[j], A_t^t[i] + A_s[j]}
return D' ;
```

□

2. Suppose $G = (V, E)$ is a *directed acyclic graph* (DAG) with positive weights $w(u, v)$ on each edge. Let s be a vertex of G with no incoming edges and assume that every other node is reachable from s through some path.

   (a) Give an $O(|V| + |E|)$-time algorithm to compute the shortest paths from s to all the other vertices in G. Note that this is faster than Dijkstra's algorithm in general.

Solution. Because G is a DAG, we can use $O(|V|+|E|)$ time to Topological Sort G, where s is the source of G. Now, start at s, each time we Relax vertices adjacent to the vertice in the front of the link list. This will also cost $O(|V|+|E|)$ time. Therefore, the time complexity of this algorithm is $O(|V|+|E|)$.
The pseudo-code is as follows:

Algorithm 2: DAG's shortest paths

Input: : A DAG G and the source vertice s.
Output: : The shortest path from s other vertices in G.

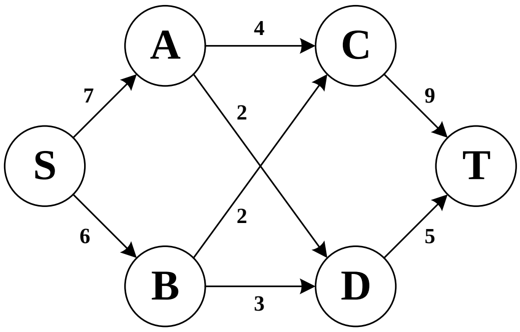
Topological-Sort(G) ;
$S \leftarrow \emptyset$;
$Q \leftarrow G.V$;
while $Q \neq \emptyset$ **do**
  $u =$ EXTRACT-HEAD(Q) ;
  $S = S \cup \{u\}$;
  for *each vertice* $v \in u.next$ **do**
    RELAX(u, v, w_G) ;

$\square$

(b) Give an efficient algorithm to compute the longest paths from s to all the other vertices.

Solution. I change the RELAX in the above algorithm. Each time we select the edges with greater weight and finally get the longest paths from s to all other vertices. $\square$

3. Consider the following network (the numbers are edge capacities).



(a) Find the maximum flow f and a minimum cut.

Solution. The maximum flow $f = 11$ and detailly $(S,A).f = 6, (S,B).f = 5, (A,C).f = 4, (A,D).f = 2, (B,C).f = 2, (B,D).f = 3, (C,T).f = 6, (D,T).f = 5$.
The minimum cut $c(X,Y) = 11$ and detailly $X = \{S,A,B\}, Y = \{C,D,T\}$. $\square$

(b) Draw the residual graph G_f (along with its edge capacities). In this residual network, mark the vertices reachable from S and the vertices from which T is reachable.

Solution. The graph G_f is in the following figure. $\square$

(c) An edge of a flow network is called a *bottleneck edge* if increasing its capacity results in an increase in the maximum flow. List all bottleneck edges in the above network and give an efficient algorithm to identify all bottleneck edges in a flow network. You need to give the notations and write down your algorithm in the form of pseudo-code.

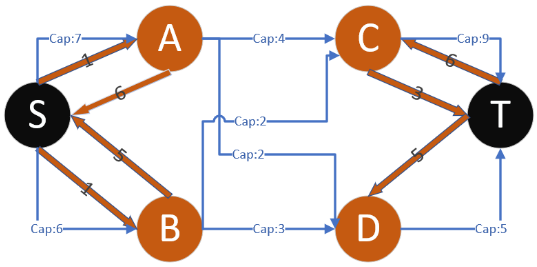


图 1: Residual Graph (3.b)

Solution. Bottleneck edges: $(A, C), (B, C)$.
To identify all bottleneck edges in a flow network, I give the following algorithm:

Algorithm 3: Search Bottleneck Edges

Input: : A graph G and the capacity c_G.
Output: : All the bottleneck edges in G.

1 $G_f \leftarrow \emptyset$;
2 Capacity-Scaling(G, c_G, G_f) ;
3 $A \leftarrow \emptyset$;
4 **foreach** *edge e in G* **do**
5   **if** *e not in G_f* **then**
6   G_f.add(e) ;
7   **if** *there is a path from s to t in G_f* **then**
8   A.add(e) ;
9   G_f.delete(e) ;
10 return A ;

The time complexity in the line 2 is $O(|E|logC)$, where C is the maximal edge capacity in G. The **for** loop will excute $|E|$ times. The time complexity in line 7 by DFS is $O(|V| + |E|)$. Therefore, the time complexity of this algorithm is $O(|V||E| + |E|^2 + |E|logC)$. However, it is just my train of thought. I think that maybe there is no need to use Capacity-Scaling to calculate the G_f of the maximum flow. But I am not sure. □

(d) Give a very simple example (containing at most four nodes) of a network which has no bottleneck edges.

Solution. The simple example is in the following figure. □

(e) An edge of a flow network is called *critical* if decreasing the capacity of this edge results in a decrease in the maximum flow. Give an efficient algorithm that finds all critical edges in a flow network. Again, you need to give the notations and write down your algorithm in the form of pseudo-code.

Solution. To find all critical edges in a flow network, I give the pseudo-code as follows:

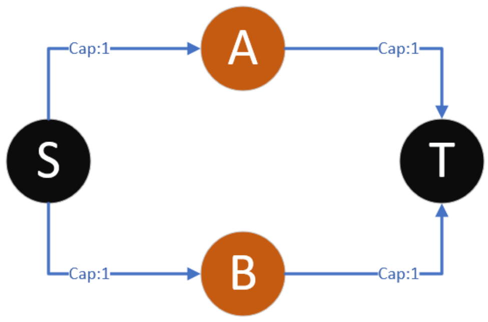


图 2: Example of network without bottleneck edges (3.d)

Algorithm 4: Search Critical Edges

Input: : A graph G and the capacity c_G.
Output: : All the critical edges in G.

$G_f \leftarrow \emptyset$;
$UsedEdges \leftarrow \emptyset$;
Capacity-Scaling$(G, c_G, G_f, UsedEdges)$;
$A \leftarrow \emptyset$;
foreach *edge* $e(u,v)$ *in* $UsedEdges$ **do**
  if $e(u,v).capacity == e(v,u).f_p$ **then**
    $tmp \leftarrow$ all $e \in \{UsedEdges - e(u,v)\} \cap$ {In the paths from s to u} ;
    G_f.add(tmp) ;
    if *there is no path from* s *to* t *in* G_f **then**
      A.add$(e(u,v))$;
    G_f.delete(tmp) ;
return A ;

Because if $e(u,v).capacity == e(v,u).f_p$, it means that in the maximum flow $e(u,v)$ is fully used. If decreasing the capacity of this edge, it means that the flow will decrease.

Assume that we decrease the flow a little. All the used edges in Capacity-Scaling and in the paths from s to u may reappear in G_f (if the used edges is already existed in G_f, G_f.add() will do nothing). After adding these edges in G_f, if there is still no path from s to t in G_f, it means we still need $e(u,v)$ to connect s and t.

The paths from s to u may be recorded in the Capacity-Scaling Algorithm. Other analysis is similar to question 3.c. The time complexiyt of this algorithm is $O(|V||E| + |E|^2 + |E|logC)$. □

Remark: You need to include your .pdf and .tex files in your uploaded .rar or .zip file.