Lab08-Shortest Path

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

∗ Name: Hongyi Guo Student ID: 516030910306 Email: guohongyi@sjtu.edu.cn

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. We can use $e' = (x, y)$ to loose the shortest path between any two vertices u and v in G. That is, we compare the path $u \to x \to y \to v$ with original shortest path and try to update it. For each pair of vertices u, v, we only compare once in undirected graphs or twice in directed graphs. So the time complexity is $O(n^2)$.

   (a) For directed graph G.

   Algorithm 1: For directed graphs

   input : An adjacent matrix of a directed graph G, original shortest path matrix D, vertex set V, edge $e = (x, y)$, w'_e.
   output: D, the updated shortest path matrix.

   ```
   for u in V do
       for v in V do
           if u ≠ v then
               D[i, j] = min{D[i, j], D[i, x] + w'_e + D[y, j]};
   ```

   (b) For undirected graph G.

   Algorithm 2: For undirected graphs

   input : An adjacent matrix of an undirected graph G, original shortest path matrix D, vertex set V, edge $e = (x, y)$, w'_e.
   output: D, the updated shortest path matrix.

   ```
   for u in V do
       for v in V do
           if u ≠ v then
               D[i, j] = min{D[i, j], D[i, x] + w'_e + D[y, j]};
               D[i, j] = min{D[i, j], D[i, y] + w'_e + D[x, j]};
   ```

   □

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. We use dynamic programming to solve this problem. We denote the distance of shortest path from s to other vertex u by $d(u)$. The state transition equation is as following.

$$d(u) = \begin{cases} \min\limits_{(v,u)\in E} \{d(v) + w(v,u)\}, & u \neq s \\ 0, & u = s \end{cases}$$

We can $depth - first - search$ the graph from vertex s and compute the shortest paths along the way. The time complexity of this is $O(|V| + |E|)$.

Algorithm 3: For undirected graphs

---

input : $G = (E, V)$, $n = |V|$, $w(u, v)$ as the weight of (u, v)
output: d

$d(1..n) \leftarrow \infty$;
DFS (u)
  foreach $(u, v) \in E$ **do**
    $d(v) \leftarrow \min\{d(v), d(u) + w(u, v)\}$;
    DFS (v);

$d(s) \leftarrow 0$;
DFS (s);

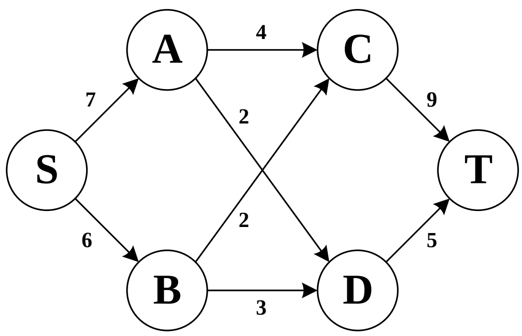
□

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

Solution. For this problem, we can simply change weights $w(u, v)$ on each edge to its opposite number and compute the shortest path $d[u]$ in this new graph. It's easy to know this current shortest path is the longest path in the original graph. In the end we just change $d[u]$ to its opposite number and that's the longest path from s to u. The time complexity of this algorithm is $O(|V| + |E|)$ which is linear.

□

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

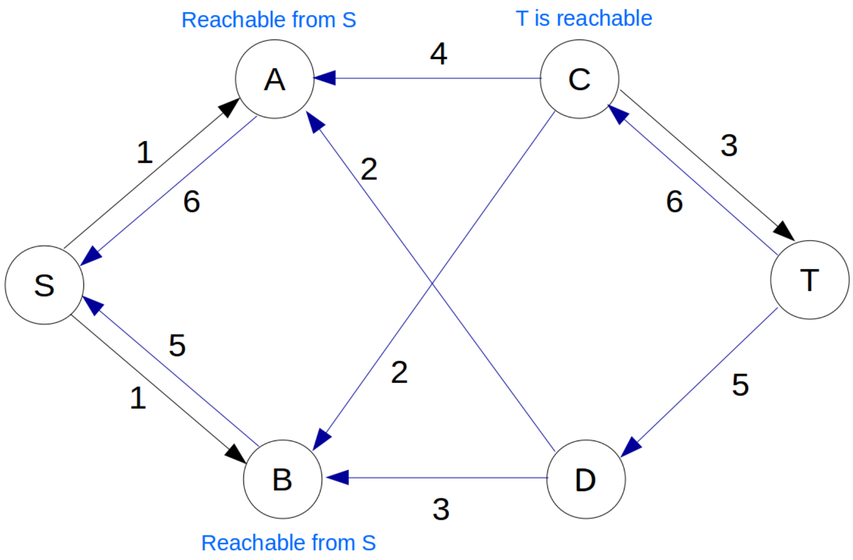


(a) Find the maximum flow f and a minimum cut.

Solution. The maximum flow $f = 11$. A minimum cut is $(S, A, B), (T, C, D)$.

□

(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.



(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.

Solution. All bottleneck edges in the above network are $(A, C), (B, C)$.

Algorithm: all edges that connect vertices reachable from S and vertices from which T is reachable are bottleneck edges, because if we add an edge (u, v) that satisfies that condition in the residual graph, then there exists an augment path: $s \to u \to v \to t$. Otherwise, no augment could be found.

We denote $G = (E, V)$ as the original graph, $G_R = (E_R, V)$ as the residual graph, and $G_R^T = (E_R^T, V)$ as the reversed residual graph which changed the edges in G_R to the opposite direction.

Algorithm 4: find-bottleneck-edges

---

input : G, G_R
output: the set of all bottleneck edges *bottlenecks*

$bottlenecks \leftarrow \emptyset$;

// A: points reachable from S
$A \leftarrow \emptyset$;

// B: points from which T is reachable
$B \leftarrow \emptyset$;

$depth - frist - search$ G_R from S and put the visited points into A;
$depth - frist - search$ G_R^T from T and put the visited points into B;

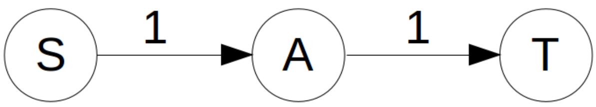
foreach $u \in A$ **do**
  foreach $(u, v) \in E$ **do**
    if $v \in B$ **then**
      $bottlenecks \leftarrow bottlenecks \cup \{(u, v)\}$;

output bottlenecks

□

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

Solution.



□

(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. We denote $G = (E, V)$ as the original graph, $G_R = (E_R, V)$ as the residual graph, and $G_R^T = (E_R^T, V)$ as the reversed residual graph which changed the edges in G_R to the opposite direction. Also, we denote A as the set of vertices reachable from S, and B as the set of vertices from which T is reachable.

Algorithm: critical edges are the edges (u, v) that $u \notin B, v \notin A, (u, v) \notin G_R$.

Algorithm 5: find-bottleneck-edges

input : G, G_R
output: the set of all bottleneck edges *bottlenecks*

bottlenecks $\leftarrow \emptyset$;

// A: points reachable from S
$A \leftarrow \emptyset$;

// B: points from which T is reachable
$B \leftarrow \emptyset$;

$depth - frist - search$ G_R from S and put the visited points into A;
$depth - frist - search$ G_R^T from T and put the visited points into B;

foreach $u \notin B$ **do**
  foreach $(u, v) \in E$ **do**
    if $v \notin A$ *and* $(u, v) \notin E_R$ **then**
      $bottlenecks \leftarrow bottlenecks \cup \{(u, v)\}$;

output bottlenecks

□