Lab08-Graph Exploration

CS214-Algorithm and Complexity, Xiaofeng Gao, Spring 2020.

∗ If there is any problem, please contact TA Yiming Liu.

∗ Name:Zehao Wang Student ID:518021910976 Email: davidwang200099@sjtu.edu.cn

1. **BFS Tree.** Similar to DFS, BFS yields a tree, (also possibly forest, but **just consider a tree** in this question) and we can define **tree, forward, back, cross** edges for BFS. Denote $Dist(u)$ as the distance between node u and the source node in the BFS tree. Please prove:

   (a) For both undirected and directed graphs, no forward edges exist in the graph.

   (b) There are no back edges in undirected graph, while in directed graph each back edge (u, v) yields $0 \leq Dist(v) \leq Dist(u)$.

   (c) For undirected graph, each cross edge (u, v) yields $|Dist(v) - Dist(u)| \leq 1$, while for directed graph, each cross edge (u, v) yields $Dist(v) \leq Dist(u) + 1$.

   Solution. (a) Forward edge is an edge (u, v) where v is a non-child descendant of u.

   According to the implementation of BFS, any vertex u and any vertex v, assumes that there is an edge (u, v) connecting u and v, and v is discovered later than u.

   Then v will be pushed into the queue when visiting u.

   If one of the u's descendants w, which also has an edge connected to v, is visited, v will not be pushed into the queue again, which otherwise will generate a forwarding edge.

   Therefore in the BFS tree there is no forward edge.

   (b) A backward edge is an edge (u, v), where u is a descendant of v.

   According to the implementation of BFS, descendants are discovered later than ancestors and thus visited later than ancestors.

   Moreover, in an undirected graph, if there is an edge connected a descendant with its ancestor, the ancestor will not be visited because BFS does not visit an already-visited vertex. Therefore this back edge will not be included into the BFS tree.

   Also, in BFS, $Dist(u)$ depends on how late u is visited. the distance of an ancestor is smaller than its descendants.

   For a back edge (u, v), u is the descendant of v, so $0 \leq Dist(v) \leq Dist(u)$.

   (c) According to the implementation of BFS, BFS discovers the shortest path from source vertex to all other vertexes.

   If $Dist(v) > Dist(u) + 1$, then there is a path $src \to u \to v$, the length of which is $Dist(u) + 1$, shorter than $Dist(u)$, which contradicts to the assumption that $Dist(v)$ is the length of the shortest path from source vertex to v.

   Similarly, we can also prove that $Dist(u) > Dist(v) + 1$ is impossible.

   Therefore for an undirected graph, $|Dist(v) - Dist(u)| \leq 1$.

   As for directed graph, because it is not sure whether there is an edge from v to u, we can only make sure $Dist(v) \leq Dist(u) + 1$.

   □

2. **Articulation Points, Bridges, and Biconnected Components.** Let $G = (V, E)$ be a connected, undirected graph. An articulation point of G is a vertex whose removal disconnects G. A bridge of G is an edge whose removal disconnects G. A biconnected component of G is a maximal set of edges such that any two edges in the set lie on a common simple cycle. Figure1

illustrates these definitions. We can determine articulation points, bridges, and biconnected components using depth-first search. Let $G_\pi = (V, E_\pi)$ be a depth-first tree of G. Please prove:

(a) The root of G_π is an articulation point of G if and only if it has at least two children in G_π.

(b) An edge of G is a bridge if and only if it does not lie on any simple cycle of G.

(c) The biconnected components of G partition the nonbridge edges of G.

Solution. (a) According to the definition of DFS, DFS first visit a source vertex which serves as the root, then calls $Explore(v)$ on each v of the root's neighbors if v is not visited.

It has been proved that $Explore(v)$ will visit all the vertexes to which there is a path from v.

Therefore in the DFS tree, all the subtrees, whose roots are neighbors of the source vertex, are disconnected.

Therefore if the root has at least two children, then the two subtrees with them as roots are disconnected. And removing the root will cause these subtrees disconnected.

If there is only one child in the DFS tree, then calling $Explore(u)$ on the root will make all the vertexes in the graph get visited, therefore the root is not an articulation point.

Therefore, The root of G_π is an articulation point of G if and only if it has at least two children in G_π.

(b) We can prove this statement by proving

i. If an edge lies on a simple cycle of G, it is not a bridge.

ii. If an edge is a bridge, it does not lie on any simple cycle of G.

- Assumes that an edge (u, v) lies on a simple cycle and vertexes $<u_1, u_2, \cdots, u_n, u, v>$ forms this simple cycle.
  if we remove (u, v), then $< u, u_n, u_{n-1}, \cdots, u_1, v >$ is another path connecting u and v.
  All the paths including (u, v) or (v, u) can include the paths above instead, and the connectivity does not change.
  Therefore removing (u, v) does not disconnect this graph, and thus (u, v) is not a bridge.
  Therefore if an edge lies on a simple cycle of G, it is not a bridge.
- Assumes that an edge (u, v) is a bridge but it also lies on a certain simple cycle $<u_1, u_2, \cdots, u_n, u, v>$.
  According to the definition of bridge, removing (u, v) can disconnect this graph.
  However, $< u, u_n, u_{n-1}, \cdots, u_1, v >$ is another path connecting u and v.
  All the paths including (u, v) or (v, u) can include the paths above instead, and the connectivity does not change.
  This is contradictory to the assumption that (u, v) is a bridge.
  Therefore if an edge is a bridge, it does not lie on any simple cycle of G.

(c) We need to prove:

i. Bridge edges are not in any biconnected component in the graph.

ii. A non-bridge edge is in at least one biconnected component and in only one biconnected component, which is the meaning of "partition".

- From (b) we can know that bridge edges must not lie on a simple cycle of G and thus are not in any biconnected component in the graph.
- From the definition of non-bridge edges, we can know that they must lie on at least one simple cycle.
  Assume that a non-bridge edge (u, v) lies in two biconnected components, that is two simple cycles, which are $<u_1, u_2, \cdots, u_n, u, v>$ and $<v_1, v_2, \cdots, v_n, u, v>$.
  Then $<u_1, u_2, \cdots, u_n, u, v_1, v_2, \cdots, v_n, v>$ form a new simple cycle.
  This means that the two smaller cycles can form a new bigger simple cycle, aka a new biconnected component.
  Therefore a non-bridge edge is in at least one biconnected component and in only one biconnected component.

Therefore, The biconnected components of G partition the nonbridge edges of G.

□

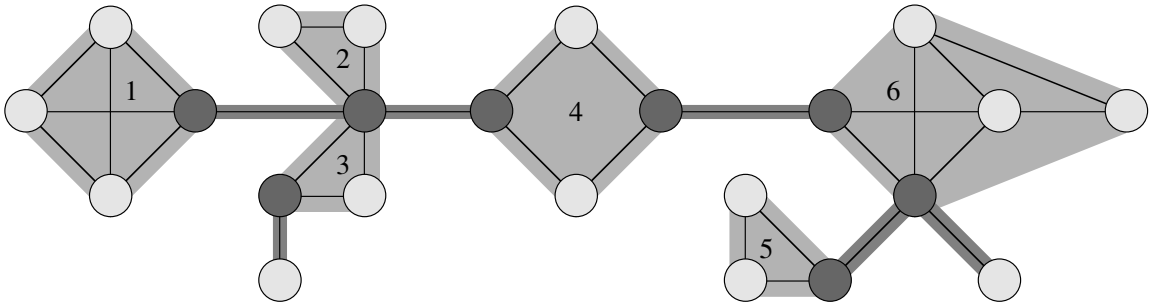

Figure 1: The definition of articulation points, bridges, and biconnected components. The articulation points are the heavily shaded vertexes, the bridges are the heavily shaded edges, and the biconnected components are the edges in the shaded regions, with a *bcc* numbering shown.

3. 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 vertexes in G. Note that this is faster than Dijkstra's algorithm in general.

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

   Solution.

   (a) An algorithm based on BFS will have a time complexity of $O(|V| + |E|)$. The space complexity is $O(|V|)$.

   When visiting a vertex u, try to update the distance of each of its neighbors v with $dist[u] + w(u, v)$. If $dist[u] + w(u, v) < dist[v]$, then the update is feasible. Otherwise $dist[v]$ must not be updated.

Algorithm 1: ShortestPaths(G,s)

Input: Graph $G = (V, E)$, which is a DAG.
Output: Distance from source:$dist[u]$ and predecessor:$pred[u]$ for all vertexes u reachable from s.

```
1  inqueue[|V|];
2  foreach u ∈ V do
3  |  dist[u] ← +∞;
4  |  inqueue[u] ← 0;
5  dist[s] ← 0;
6  inqueue[s] ← 1;
7  Q ← [s];
8  while Q is not empty do
9  |  u ← dequeue(Q);
10 |  inqueue[u] ← 0;
11 |  foreach (u, v) ∈ E do
12 |  |  if dist[u] ≠ +∞ and dist[v] > dist[u] + w(u, v) then
13 |  |  |  dist[v] ← dist[u] + w(u, v);
14 |  |  |  pred[v] ← u;
15 |  |  |  if inqueue[v] = 0 then
16 |  |  |  |  enqueue(Q, v);
17 |  |  |  |  inqueue[v] ← 1;
18 return dist[ ], pred[ ];
```

(b) An algorithm based on topological sort is an efficient algorithm to solve this problem. the topological sort has $O(|V|+|E|)$ time complexity and the relaxation has $O(|E|)$ time complexity.

Therefore the total time complexity is $O(|V| + |E|)$. The space complexity is $O(|V|)$.

Algorithm 2: LongestPaths(G,s)

Input: Graph $G = (V, E)$, which is a DAG.
Output: Distance from source:$dist[u]$ and predecessor:$pred[u]$ for all vertexes u reachable from s.

```
1  topo[|V|];
2  outDegree[|V|];
3  rank ← 0;
4  foreach u ∈ V do
5  |   dist[u] ← −∞;
6  |   inDegree[u] ← the number of edges pointing to u;
7  dist[s] ← 0;
8  topo[rank] ← s;
9  while rank ≠ |V| − 1 do
10 |   u ← topo[rank];
11 |   for (u, v) ∈ E do
12 |   |   inDegree[v] ← inDegree[v] − 1;
13 |   |   if inDegree[v] = 0 then
14 |   |   |   rank ← rank + 1;
15 |   |   |   topo[rank] ← v;
16 for rank = 0 to |V| − 1 do
17 |   u ← topo[rank];
18 |   foreach (u, v) ∈ E do
19 |   |   if dist[v] < dist[u] + w(u, v) then
20 |   |   |   dist[v] ← dist[u] + w(u, v);
21 |   |   |   pred[v] ← u;
22 return dist[ ], pred[ ];
```

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