Lab08-Graph Exploration

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

∗ If there is any problem, please contact TA Yiming Liu.
∗ Name:Hanzhang Yang Student ID:518030910022 Email: linqinluli@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 $0 \le Dist(v) \le Dist(u)$.

   (c) For undirected graph, each cross edge (u, v) yields $Dist(v) = Dist(u)$ or $Dist(v) = Dist(u) + 1$, while for directed graph, each cross edge (u, v) yields $Dist(v) \le Dist(u) + 1$.

   Proof.

   (a) By contradiction. We assume there exists a forward edge. And node i is the ancestor, j is the descendant.

   If the forward edge exists, when building the BFS tree, j must be the child of i. Thus there will be a circle which contains the edge (i, j)that it's contrary to it's a tree.

   Therefore no forward edges exist in the graph.

   (b) By contradiction. Assume there exists a back edge (u, v)

   Undiredted graphs:

   For it's a undirected graph, when building BFS tree, u will be the child of v. Thus it's contrary to that (u, v) is a back edge.

   Therefore, there are no back edges in undirected graph.

   Directed graphs:

   If $Dist(v) > Dist(u)$, v will be the descendant of u. Thus when building BFS tree, v will be searched earlier than u and u will be the child of v, whih is contrary to that (u, v) is a back edge.

   Therefore, in directed graph each back edge(u, v) yields $0 \le Dist(v) \le Dist(u)$.

   (c)

   When building BFS tree, every $Dist()$ yields a level. And the search order is according to the number of $Dist()$.

   Undiredted graphs:

   By contradiction. Assume $|Dist(v) - Dist(u)| \ge 2$.

   When building the level according to $min\{Dist(u), Dist(v)\} + 1$. For there exists edge (u, v), u and v will be parent and child. Thus it's contrary to that (u, v) is a cross edge and $|Dist(v) - Dist(u)| = 1$.

   Directed graphs:

   By contradiction. Assume $Dist(v) - Dist(u) \ge 2$.

   The same as before, When building the level according to $min\{Dist(u), Dist(v)\} + 1$, v hasn't been added another node except u. For there exists edge (u, v), u will be the parent of v which causes the contradiction.

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.

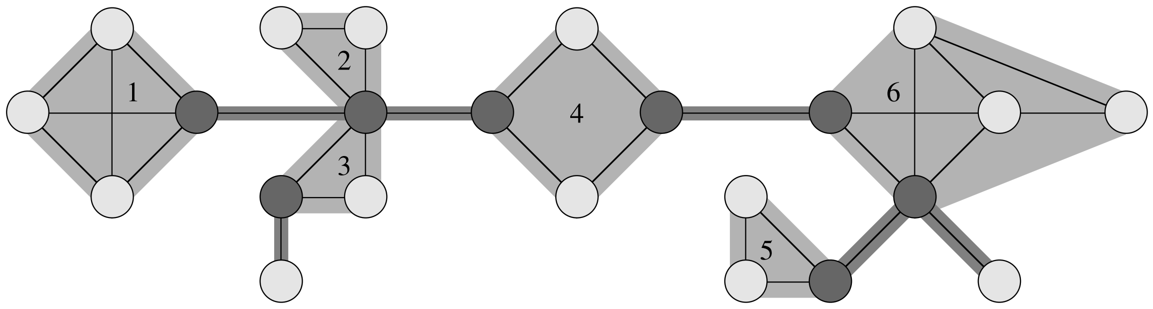

图 1: The definition of articulation points, bridges, and biconnected components. The articulation points are the heavily shaded vertices, the bridges are the heavily shaded edges, and the biconnected components are the edges in the shaded regions, with a *bcc* numbering shown.

Proof.

(a)

Necessity.

For G_π is a tree, assume the root of G_π has at least two children. After removing the root, its child trees will be disconnected. Thus the root is an articulation point.

Sufficiency.

If G_π is an articulation point of G, its removal disconnects G. If it has only one child, its removal only causes G become a node and another *tree*. Therefore, it has at least two children.

(b)

Necessity.

If an edge does not lie on any simple cycle of G, the nodes according to the edge cannot be connected after removing the edge. Thus the whole graph G disconnects too. Therefore, the edge is a bridge.

Sufficiency.

Assume an edge lies on a simple cycle of G. After removing the edge, the circle still connects and G doesn't disconnect too. Thus it cannot lie on any simple cycle of G.

(c) Consider a biconnected component of G. For G is connected graph, the component must connect with at least one edge. Because a biconnected component of G is the maximal set of edges such that any two edges in the set lie on a common simple cycle. The connected edge cannot be added to the biconnected component and it does not lie on any simple cycle. If it lies, it will be in a biconnected component. Thus according to (b), the edge is an bridge. And for all edges, it can be either nonbridge or an edge in a biconnected component.

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

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 vertices 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 vertices.

   Solution.

   For it's a DAG, we can use Dynamic Programming to solve this problem.

   (a) We can define $dp[i]$ stores the shortest path weight from node s to i.

$$dp(i) = \begin{cases} 0 & \text{if i=s} \\ min\{dp(j) + weight(j,i)\}(for\ all\ j \to i) & \text{else} \end{cases} \tag{1}$$

   However it needs topologically sort and store all the node in the dp array, which enables the prvious node been visited before the current node.

   Here is the pseudo code

```
Algorithm 1: shortest path in DAG
  Input: $G = (V, E)$ and the weights $W(i, j)$
  Output: the shortest paths weights of all nodes
1 $dp[n] = \{0\}$;
2 $node[n]$;
3 $node[1] = s$;
4 topologically sort and store nodes in node[ ];
5 for $i = 2$ to $n$ do
6     $dp[node[i]] = min\{dp[node[j]] + w(j,i)\}$(for all $node[j] \to node[i]$)
```

   The time complexity consists of topologically sort which costs $O(|V| + |E|)$ and the traversal which costs $O(|V| + |E|)$.

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

(b) We can define $dp[i]$ stores the longest path weight from node i to other nodes.

$$dp(i) = \begin{cases} 0 & \text{if node i's out degree is 0} \\ max\{dp[j] + weight(i,j)\}(\text{for all } i \to j) & else \end{cases} \tag{2}$$

Algorithm 2: longest path in DAG

Input: $G = (V, E)$ and the weights $W(i, j)$

Output: the longest paths weights

```
1  $dp[n] = \{0\}$;
2  $node[n]$;
3  $node[1] = s$;
4  topologically sort and store nodes in node[ ];
5  for $i = n$ to 1 do
6      if node i's out degree is 0 then
7          $dp[node[i]] = 0$;
8      else
9          $dp[node[i]] = max\{dp[node[j] + w(i,j)]\}$(for all $node[i] \to node[j]$);
10 Output $dp[node[1]]$;
```

The result is stored in $dp[node[1]]$. The time complexity is the same as before which is $O(|V| + |E|)$.

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