Lab08-Graph Exploration

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

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

$*$ Name: Futao Wei   Student ID: 518021910750   Email: weifutao@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) Suppose for contradiction that (u, v) is a forward edge in the graph, and that t is u's child as well as v's ancestor in the BFS tree. Then v should be u's child in the BFS tree — a contradiction!

   (b) The undirected case can be proved as in (a).
   According to the definition of the back edge, v is an ancestor of u in the BFS tree. Hence it's obvious that $0 \leq Dist(v) \leq Dist(u)$.

   (c) For undirected graph, assume for contradiction that some cross edge (u, v) yields $|Dist(v)-Dist(u)| \geq 2$. Then we get a contradiction that (u, v) cannot be a cross edge, since u would have been a child of v or vice versa.
   For directed graph, suppose for contradiction that some cross edge (u, v) yields $Dist(v) \geq Dist(u) + 2$. Then we have a contradiction that (u, v) cannot be a cross edge, since v would have been a child of u.

   □

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.

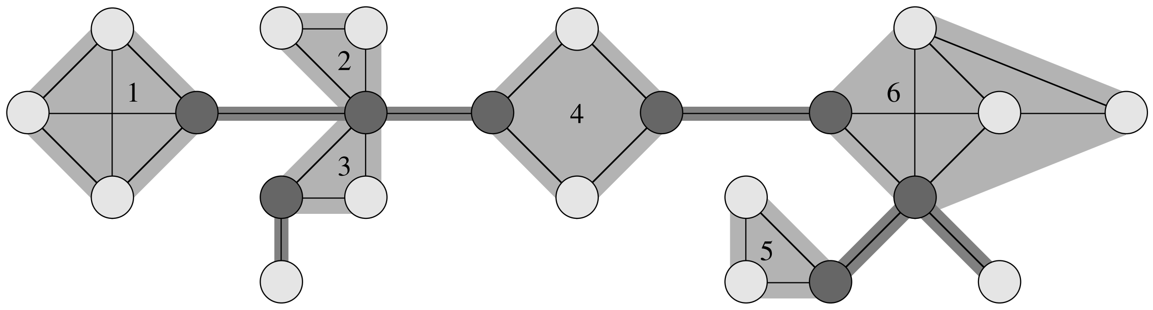

图 1: The definition of articulation points, bridges, and biconnected components. The articulation points are the heavily shaded vertices, the bridges are the heavily shaded edges, and the biconnected components are the edges in the shaded regions, with a *bcc* numbering shown.

(a) $\Longrightarrow$:
Suppose for contradiction that the root of G_π has only one child. Then its removal does not disconnect G, i.e., it's not an articulation point of G — a contradiction!
$\Longleftarrow$: Trivial.

(b) $\Longrightarrow$:
Suppose for contradiction that a bridge edge of G lies on some simple cycle of G. Then its removal will not disconnect G — a contradiction!
$\Longleftarrow$: Trivial.

(c) The biconnected components of G partition the nonbridge edges of G. $\Longleftrightarrow$ Any of the non-bridge edges in G is in only one biconnected component of G.
The contrapositive of (b) shows that any non-bridge edge of G lies on some simple cycle of G, i.e., on at least one biconnected component.
Suppose that some non-bridge edge of G lies on two biconnected components. Then the combination of the two components is actually a larger biconnected component. Hence any non-bridge edge of G is in only one biconnected component of G.

$\square$

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.

(a)

```
Algorithm 1: Sort($G, s$)
  Input: $G = (V, E)$, vertex $s \in V$, $stack = \emptyset$
  Output: $stack$
1 $visited[s] = True$;
2 foreach $(s, v) \in E$ do
3     if not $visited[v]$ then
4         Sort($G, v$);
5 $stack$.push($s$);
6 return $stack$;
```

```
Algorithm 2: SP($G, s$)
  Input: $G = (V, E), s$
  Output: $dist[1, \cdots, n]$
1 for $i = 1$ to $n$ do
2     $dist[i] = \infty$;
3 $stack$ = Sort($G, s$);
4 while not $stack$.empty() do
5     $v = stack$.pop();
6     foreach $(v, u) \in E$ do
7         if $dist[u] > dist[v] + w(v, u)$ then
8             $dist[u] = dist[v] + w(v, u)$;
9 return $dist[1, \cdots, n]$;
```

This algorithm is very similar to Dijkstra's algorithm except for using a stack rather than a heap. Hence it's faster than Dijkstra's algorithm.

(b)

```
Algorithm 3: LP($G, s$)
  Input: $G = (V, E), s$
  Output: $dist[1, \cdots, n]$
1 for $i = 1$ to $n$ do
2     $dist[i] = 0$;
3 $stack$ = Sort($G, s$);
4 while not $stack$.empty() do
5     $v = stack$.pop();
6     foreach $(v, u) \in E$ do
7         if $dist[u] < dist[v] + w(v, u)$ then
8             $dist[u] = dist[v] + w(v, u)$;
9 return $dist[1, \cdots, n]$;
```

$\square$

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