CS/ECE 374 Spring 2017
Homework 7 Problem 3

Renheng Ruan (rruan2)
Lanxiao Bai (lbai5)
Ho Yin Au (hoyinau2)

Given an undirected connected graph $G = (V, E)$ an edge (u, v) is called a cut edge or a bridge if removing it from G results in two connected components (which means that u is in one component and v in the other). The goal in this problem is to design an efficient algorithm to find *all* the cut-edges of a graph.

What are the cut-edges in the graph shown in the figure?

- Given G and edge $e = (u, v)$ describe a linear-time algorithm that checks whether e is a cut-edge or not. What is the running time to find all cut-edges by trying your algorithm for each edge? No proofs necessary for this part.
- Consider *any* spanning tree T for G. Prove that every cut-edge must belong to T. Conclude that there can be at most $(n-1)$ cut-edges in a given graph. How does this information improve the algorithm to find all cut-edges from the one in the previous step?
- Suppose T is a spanning tree of G rooted at r. Prove that an edge (u, v) in T where u is the parent of v is a cut-edge iff there is no edge in G, other than (u, v), with one end point in T_v (sub-tree of T rooted at v) and one end point outside T_v.
- Use the property in the preceding part to design a linear-time algorithm that outputs all the cut-edges of G. You don't have to prove the correctness of the algorithm but you should point out how your algorithm ensures the desired property. *Hint:* Consider a DFS tree T and some additional information you can compute during DFS. You may want to run DFS on the example graph with the cut edges identified.

Solution:

(a) (e, g), (f, j), and (h, l) are the cut-edges.

(b) By removing e from G and use whatever-first search to determine whether u is still connected to v. If u still connected to v, then e is not a cut-edge, and e is a cut-edge if u not connected to v. Whatever-first search takes $O(m)$ time, and take $O(m^2)$ if we go though each of the m edges.

(c) For any edge e $\notin$ T, removing e would not makes the graph to be disconnected because there is always a path in the spanning tree that connects any two vertices in G.
There can be at most $n-1$ cut-edges in a given graph because only the edges in the spanning tree can be cut-edges, and the spanning tree has n-1 edges.
Since the number of candidate cut-edges drop from m to $n-1$, then the algorithm takes $O(mn)$ time by going though each of the $n-1$ edges.

(d) Proving by Contrapositive: We assume that there is an edge e in G with one end point in T_v (sub-tree of T rooted at v) and one end point outside T_v. We know that T is connected, then removing (u, v) from T leaves it in two connected components: one contains u and another

contains v (T_v). The graph will be $G-(u,v)$ which is a single connected component in G by the edge e connecting those two components. Hence, (u,v) is not a cut-edge.
By removing the edge (u,v) from T. We have two connected components: one contains u and another contains v (T_v). T_v is a connected component of $G-(u,v)$ because T_v does not have any edge leaving it in $G-(u,v)$. And u is also a connected component which is not connect T_v in $G-(u,v)$. Hence, (u,v) is a cut-edge.

(e) Let T be a DFS tree.
We define each node u:

$$low(u) = min \begin{cases} \text{pre(u)} \\ \text{pre(w)} & (v,w)\text{is a back edge for } v \notin T_u \end{cases}$$

The following procedure records the previsit time $pre(u)$ and predecessor $pred(u)$, and compute $low(u)$ for each u in DFS.

```
LowDFS(G):
  for all u ∈ V
      pred(u) ← NULL
  chose an arbitrary vertex u ∈ V
  LowDFSwizU(G,u)
```

```
LowDFSwizU(G,u):
  change u to a visited vertex
  time ← time + 1
  pre(u) ← time
  low(u) ← pre(u)
  for each edge (u,v) in Adj(u)
      if v is not marked as visited
          LowDFSwizU(G,v)
          low(u) ← min{low(u), low(v)}
          pred(v) ← u
      else
          low(u) ← min{low(u), pre(v)}
```

The procedure above is DFS with predecessor and previsit time, and with constant time at every step to calculate $low(u)$. So the running time is same as DFS which is $O(m+n)$.
And the AllCuts function returns the set of all of the cut-edges:

```
AllCuts(G):
  pre(), low(), pred() ← LowDFS(G)
  S ← ∅
  for each vertex v ∈ V
      if low(v) = pre(v) and pred(v) ≠ NULL
          add (pred(v), v) to S
  return S
```

The algorithm we have takes $O(m+n)$ time, and $Allcuts(G)$ takes $O(n)$ time. Thus, the total running time is $O(m+n)$.

Proving by induction to show how $low(u)$ is correctly computed:

Base Case:
u has no children, then $low(u) = pre(u)$, which is correct. Or there are back edges from u, which calculate the min $\{pre(w) : (u, w) is a back edge\}$.

Inductive hypothesis:
Assume $low(u)$ is calculated correctly for all children v of u.

Inductive Step:
If there are no back edges leaving T_u then $low(u) = pre(u)$. If $low(u)$ is achieved by $pre(w)$ where (u, w) is a back edge, then the "else" clause will calculate $low(u)$ correctly. If $low(u)$ is achieved by $pre(w)$ where (v, w) is a back edge for $v \in T_u$, $v \notin u$, then the line after the recursive call LowDFSwizU(G,v) will calculate $low(u)$ correctly by the inductive hypothesis.

The solution template is taken from
https://www.coursehero.com/file/13235458/hw7-solutionspdf/.

■