CS170–Spring 2015 — Solutions to Homework 04

Chia-Hao Chiao, SID 23364418, `cs170-ht`

Collaborators: Kevin Chau, Ian Fox, Yi-Lin Zhao

1.

(a)
- A, B, G, D, E, F, C, H, I
-

	A	B	C	D	E	F	G	H	I
Pre	1	2	13	4	5	8	3	14	16
Post	12	11	18	7	6	9	10	15	17

- $\overrightarrow{AB}$: T, $\overrightarrow{AE}$: F, $\overrightarrow{BG}$: T, $\overrightarrow{CH}$: T, $\overrightarrow{CI}$: T, $\overrightarrow{DB}$: B, $\overrightarrow{DE}$: T, $\overrightarrow{ED}$: B, $\overrightarrow{GD}$: T, $\overrightarrow{GF}$: T, $\overrightarrow{IH}$: C

(b) 6

2.

(a) **Main idea.** Determine the minimum number of undirected edges needed.

Pseudocode.

```
global int edges
exploreGraph(G):
  for each vertex $v \in G$:
    if not visited(v):
      visited(v) = true
      explore(v)
  return edges
explore(G, v):
  visited(v) = true
  for each edge $(v, u) \in G$:
    if not visited(u):
      edges += 1
      explore(u)
```

Proof of correctness. Since this is a undirected graph, for a connected graph with n nodes, we only need $n-1$ edges to connect each vertex. In this method, we add one to the result whenever we explore a new node except for the root, which gives us exactly what we need.

Running time. $O(|V| + |E|)$

Justification. We visit each node and edge exactly once in the method and the operation of each visit takes constant time.

(b) **Main idea.** The same as part (a) except for directed graphs.

Pseudocode.

1. Run the algorithm introduced in textbook 3.4.2 to find out the strongly connected components as well as the meta graphs.

2. While producing the meta graphs, sum up each number of vertices of strongly connected components, say the sum is `edges`.

3. Do a DFS on the meta graph. Whenever a tree edge or a cross edge is visited, increment `edges` by 1.

4. Return `edges`.

Proof of correctness. We know that for each strongly connected component, if there are n vertices, then n edges are required at minimal to form a cycle and connect all the vertices. After forming the meta graph, since it is a DAG, there is no backedge, we only consider tree edges, cross edges and forward edges. Forward edges do not need to be considered because tree edges have already provide the same functionality. Thus, by summing up the number of vertices in the strongly connected components, number of tree edges and cross edges, we get the minimal edges required.

Running time. $O(|V| + |E|)$

Justification. We visit each vertex and edge at most twice during forming the meta graph and exploring the meta graph. The operation in each visit takes constant time, so the run time is $O(|V| + |E|)$.

3.

Main idea. Takes a DAG and determines whether it contains a directed path that touches every vertex exactly once.

Pseudocode.

1. Look for a source by checking the incoming edges of each vertex. If there is no incoming edges, the vertex is a source. If there are multiple sources or a vertex without any edges attached to it, return false.

2. Do a depth first traversal from the source. Keep two stacks (push and pop at the same time), one is for the number of vertices visited in the path, and the other one for the vertex to explore. We do not use the visited feature as usual. Instead, we check if the corresponding number of vertices visited is the same as the number of total vertices. If yes, then return true.

3. If both stacks are empty, return false.

Proof of correctness. First, we narrow down the cases to exactly one source since it is impossible to have a directed path that touches every vertex exactly once otherwise. By avoiding the visited feature, we are able to explore every possible path from the only source. The number of vertices on the path indicates whether the path touches every vertex once.

Running time. $O(|V| + |E|)$

Justification. Since looking for the source takes $O(|V|)$ time, and the depth first traversal takes $O(|V| + |E|)$ steps while each step takes constant time for operations, the running time is $O(|V|) + O(|V| + |E|) = O(|V| + |E|)$.

4.

Main idea. Find whether there is a unique shortest path from s to u for each $u \in G$.

Pseudocode.

```
for all $u \in V$
  dist($u$) $= \infty$
dist($s$) $= 0$
usp = boolean[]
$H$ = makequeue($V$)
while $H$ is not empty:
 $u$ = deletemin ($H$)
  for all edges $(u, v) \in E$:
   if dist ($v$) $>$ dist($u$) $+ l(u, v)$:
    usp[$v$] = false
    dist($v$) = dist($u$) $+ l(u, v)$
    decreasekey($H, v$)
   else if dist ($v$) $=$ dist($u$) $+ l(u, v)$:
    usp[$v$] = true
```

Proof of correctness. Since all the edges have positive length, the concept of Dijkstra's theorem can be applied. We make a twist to Dijkstra's shortest-path algorithm. Besides checking whether the current distance distance stored in the `dist` function is greater than the new distance calculated by the current path, we also check if it is equal. If it is equal, that means we have at least two shortest path at the moment, so we set `usp` to true. If it is greater, we set `usp` to false. Since Dijkstra's theorem finds the shortest path, the final output will be the result for shortest path at the final stage.

Running time. $O((|V| + |E|)log|V|)$

Justification. Since we only add another if statement with constant time operation to Dijkstra's algorithm, the running time is the same as Dijkstra's algorithm.

5.

(a) Proof by contraposition: If two biconnected components have an edge in common, since the graph is undirected, both biconnected components can form a cycle that contain each other through the common edge, which means that the two biconnected components are the same by definition. Thus, two distinct biconnected components cannot have any edges in common.

(b) Proof by contraposition: If two vertex sets corresponding to different biconnected components have more than one vertex in common, say two vertices, then there must be at least an edge in common (biconnected component is connected by cycle). From part(a) we know that this indicates the two biconnected components are identical. Thus, the vertex sets corresponding to two different biconnected components are either disjoint or intersect in a single vertex.

(c) ($\Rightarrow$): If the root of the DFS tree is a separating vertex, according to part(b), it must be the only intersecting vertex, so it must have more than a child.

($\Leftarrow$): If the root has more than one child in the tree, then there is not any edge connecting the two or more children (otherwise the potential child would be traversed by other descendent first). Since there are no edges connecting the children and their descendents, each of them has its own biconnected component and makes the root a separating vertex.

(d) ($\Rightarrow$): If a non-root vertex v of the DFS tree is a separating vertex, according to part(b), it corresponds to two different biconnected components, which mean that the edge between v and its ancestor cannot be in part of the biconnected component for at least one of the biconnected components its separating. Thus, at least one of its child which none of its descendants and itself has a backedge to an ancestor of v.

($\Leftarrow$): Since v has a child v' such that none of whose descendants has a backedge to a proper ancestor of v, the edge between v and v' itself is a biconnected component. Since there is another biconnected component between v and some of its ancestors, v is a separating vertex.

(e) **Main idea.** Computes all separating vertices and biconnected components of a graph.

Pseudocode.

1. Do an DFS with additional operations during previsits and postvisits. Say the current vertex getting explored is u. During previsits, we set $\texttt{low}(u) = \min(\text{pre}(u), \text{pre}(w))$ if there is for all backages (u, w). If there is no backages, then $\texttt{low}(u) = \text{pre}(u)$. For postvisits, say the parent of u is u'. Let $\texttt{low}(u') = \min(\texttt{low}(u), \texttt{low}(u'))$ if u has at least one backedge.

2. With all the `low` known in the graph, we do an additional DFS with additional operations during previsits and postvisits. Again, say the current vertex getting explored is u. During previsits, for each edge (u, v) connected to u, if $\texttt{pre}(u)$ ¡ $\texttt{low}(v)$ for all v, then mark u as a seperating vertex. In this case, when exploring each edge after previsit, add u to the stack at the beginning of each iteration and pop everything out until reaching another separating vertex or empty at the end of the iteration to form a biconnected component. If u is not a separating vertex, simply add itself into the stack before the for loop starts.

Proof of correctness. For the first part, the comparison for `low` in case of the current vertex having backedges is handled during previsits. The comparison for `low` in case of any child having backedges is handled. Thus, the `low` found in result of the first step fulfills the definition. For the second part, clearly separating vertices are found by comparing the value of `low` of each child. As for the biconnected components, since each biconnected component is separated by separating vertices in the stack, each time popping edges from the stack forms a biconnected component.

Running time. $O(|V| + |E|)$

Justification. There are two DFS and all operations including previsits and postvisits take constant time. Therefore, the running time is linear, $O(|V| + |E|)$.