HOMEWORK 4

Charles Haithcock, `cehaith2`

CSC 505 - Design and Analysis of Algorithms
Steffen Heber

Due: 21 April 2017

Homework should be submitted using WolfWare Submit Admin in PDF, or plain text. To avoid reduced marks, please submit **word/latex-formated PDF file, NOT scanned writing in pdf format**. Scanned writing is hard to read, takes longer to grade, and produces gigantic files. **To simplify grading, please make sure that each problem starts on a new page**. All assignments are due on 9 PM of the due date. Late submission will result in 10%/40% point reduction on the first/second day after the due date. No credit will be given to submission that are two or more days late. Please try out Submit Admin well before the due date to make sure that it works for you.

All assignments for this course are intended to be individual work. Turning in an assignment which is not your own work is cheating. The Internet is not an allowed resource! Copying of text, code or other content from the Internet (or other sources) is plagiarism. Any tool/resource must be approved in advance by the instructor and identified and acknowledged clearly in any work turned in, anything else is plagiarism.

General instruction about how to "give/describe/..." an algorithm, taken from Erik Demaine. **Try to be concise, correct, and complete.** To avoid deductions, you should provide (1) a textual description of the algorithm, and, if helpful, pseudocode; (2) at least one worked example or diagram to illustrate how your algorithm works; (3) a proof (or other indication) of the correctness of the algorithm; and (4) an analysis of the time complexity (and, if relevant, the space complexity) of the algorithm. Remember that, above all else, your goal is to communicate. If a grader cannot understand your solution, they cannot give you appropriate credit for it.

Question 1 (24 pts, 3 pts each)

- Problem 22-2 a-h (3 points each)

22-2 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 teo edges in the set lie on a common simple cycle. Figure 22.10 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.

a. Prove that 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. Let v be a nonroot vertex of G_π. Prove that v is an articulation point of G if and only if v has a child s such that there is no back edge from s or any descendant of s to a proper ancestor of v.

c. Let

$$v.low = \min \begin{cases} v.d, \\ w.d : (u, w) \text{ is a back edge for some descendant } u \text{ of } v. \end{cases} \tag{1}$$

Show how to compute $v.low$ for all vertices $v \in V$ in $O(E)$ time

d. Show how to compute all articulation points in $O(E)$ time.

e. Prove that an edge of G is a bridge if and only if it does not lie on any simple cycle of G.

f. Show how to compute all the bridges of G in $O(E)$ time.

g. Prove that the biconnected components of G partition the nonbridge edges of G.

h. Give an $O(E)$-time algorithm to label each edge e of G with a positive integer $e.bcc$ such that $e.bcc = e'$ if and only if e and e' are in the same biconnected component.

- Problem 22-3 a (4 points) on pages 622 and 623

22-3 Euler tour

An ***Euler tour*** of a strongly connected, directed graph $G = (V, E)$ is a cycle that traverses each edge of G exactly once, although it may visit a vertex more than once.

a. Show that G has an Euler tour if and only if in-degree(v) = out-degree(v) for each vertex $v \in V$.

Purpose Learn about articulation points, bridges, biconnected components, and Euler tours

Problem 22-2 a

The root, r, of a Depth-First Search (DFS) tree, G_π, can have thee different kinds of amounts of children; (a) zero children, (b) one child, (c) and two or more children. For (a), if r has no children, r is part of a graph of only one node, so removal of r from G_π will not disconnect the graph so r is not an articulation point. For (b), if r has only one child, then r can reach all nodes and all nodes can reach each other without travelling through r. Removal of r will never prevent any node from reaching any other node (except of course r itself) so r is not an articulation point.

For (c), assume r has two children (the roots of subtrees s_1 and s_2) and removal of r would not disconnect the tree. In such a scenario where removal of r does not disconnect G_π, a back edge must exist which connects the subtrees. Two problems exist with this however; by definition, trees do not contain cycles so G_π would never have been a DFS tree to begin with. The second, more important aspect of such a contradiction is, when building G_π, the required back edge would have been traversed first while exploring s_1 before going through s_2 and created a different DFS tree from G_π. This contradiction would extend for any number of subtrees $s_1, s_2, \ldots s_n$ as back edges would need to connects some node in s_k to $s_{k-1} : 1 < k \leq n$. As such, removal of r would cause a disconnect graph meaning r is an articulation point. QED.

Problem 22-2 b

Assume removal of v would not disconnect v's child s or some descendant of s, no back edge exists connecting s or a descendant of s to an ancestor of v, and v is a non-root vertex of G_π. In order for such a disconnect to be prevented, (1) an edge would need to exist connecting s or the descendant of s to some ancestor of v or (2) to another node of G_π not part of the subtree whose root is v. The second scenario is impossible as such an edge would be traversed in an earlier visited subtree connected to via the cross edge. In order for (1) to operate, an edge would need to exist connecting s or some descendant of s to some ancestor of v. Such an edge is by definition a back edge, creating a contradiction. QED.

Problem 22-2 c

Textual description of the algorithm with pseudocode

Because the two main cases in the formula provided are either $v.d$, the current vertex's discovery time, or consideration of backedges of descendants of v if they have back edges, the algorithm recurses down children before considering the $v.low$ by starting with leaves of subtrees and propagating the minimum values upward.

On each recursion, the algorithm begins by assuming the minimum (`min`) is the current node, v's discovery time, $v.d$ (case one in the formula). From here, backedges from v are checked (if any) and compared to the current `min`, meaning the algorithm performs $\min(v.d, w.d)$ where $w.d$ is the vertex on the other end of the back edge starting at v. All backedges starting with v are checked updating `min` along the way. The algorithm assumes multiple backe dges can exist and assumes a function `back_edges(v)` exists which takes a vertex `v` as a parameter and returns a list of backedges starting with `v` each with a pointer to the node on the other end called `w`, similar to the formula.

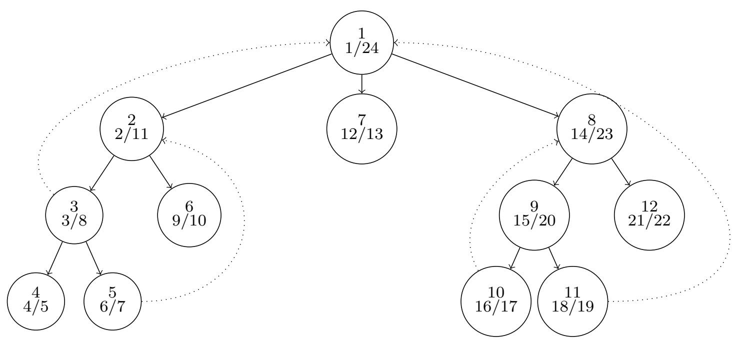
After checking all backedges starting from v, the children of v are checked. In order to accomplish the task of checking for the discovery time to descendants of v to a backedge to w, the algorithm eventually returns `min` for v which allows for checking of descendants of v which many have backedges. The section above which checks for backedges and stores the minimum of such (if applicable) allows propagating the value upward. If a backedge does not exist, then a descendant u of v should have $u.d > v.d$ because the DFS algorithm discovers u after v. For each child of v, the `min` is appropriately updated if the returned d time is lower than `min`.

Once all backedges and children of v are checked, `min` is returned as it should contain $v.d$. Returning $v.d$ allows for parent recursions to check descendants.

The algorithm is made recursive and given an "entry point" to facilitate recursion.

A worked example

Consider the following example DFS Tree where dotted edges are back edges and the vertices contain the vertex label and corresponding discovery/finish times.



Below are the recursions in the algorithm:

v	$v.d$	Recurse	`back_edges(v)`	$v.low$	
1	1	$\{ \cancel{2: 1}, \cancel{7: \varnothing}, 8: (11, 1).\text{d} = 1\}$	1	$\varnothing$	$\varnothing$
2	2	$\{ 3: (3, 1).\text{d} = 1, \cancel{6: \varnothing}\}$	1	$\varnothing$	(3, 1)
3	3	$\{ \cancel{4: \varnothing}, 5: (5, 2).\text{d} = 2\}$	2	$\{(3, 1)\}$	(3, 1)
4	4	$\varnothing$	4	$\varnothing$	$\varnothing$
5	6	$\varnothing$	6	$\{(5, 2)\}$	(5, 2)
6	9	$\varnothing$	9	$\varnothing$	$\varnothing$
7	12	$\varnothing$	12	$\varnothing$	$\varnothing$
8	14	$\{\cancel{9: 1}, \cancel{12: 21}\}$	1		
9	$\cancel{15}$	~~b. e.~~	rec $\implies$ $\{\cancel{10: 14}, 11: 1\}$	1	
10	$\cancel{16}$	b. e. $\implies$ 14	rec $\implies$ $\cancel{\{\}}$	14	
11	$\cancel{18}$	b. e. $\implies$ 1	rec $\implies$ $\cancel{\{\}}$	1	
12	21	~~b. e. ?~~	rec $\implies$ $\cancel{\{\}}$	21	

Note the table is layed out as if after having ran the entirety of the algorithm. As values are checked, if it does not cause an update to `min` or if the prior `min` is replaced, its value will be crossed out.

Proof

FUCK

Time complexity analysis

Assuming all operations are constant time actions, the loops at lines 10-12 and 15-18 are thus multiple iterations of constant time actions. Likewise, these loops are ran for each recursion. A recursion occurs for each node in the graph. As such, for the recursions alone, the algorithm is guaranteed to run at $\Theta(|V|)$.

In general, for every node, all possible edges are eventually considered; when checking all children of a node, all edges connected via tree edges are considered followed up with a check against all back edges. As such, since the $|E|$ is typically greater than $|V|$, checking all edges dominates the running time over $|V|$. Thus the running time is bound at $O(|E|)$.

Problem 22-2 d

Fortunately, computing all articulation points is all but done with the above problems. The steps then are effectively layed out already;

1. Run DFS against the graph, G, to generate G_π. This action requires $O(|V| + |E|)$, however, since G is connected, $|E| > |V| - 1$, so generating G_π is only $O(|E|)$.
2. Run the above algorithm to calculate `low` for each vertex $v \in V$.
3. From here, with G_π, we can apply some of the above points to determine articulation points in G. For each vertex $v \in V$:
   - if v is a root vertex and has more than one child, then v is an articulation point.
   - if v a non-root vertex, has descendants, and has no backedge to a proper ancestor, v is an articulation point.
   - if $v.d <$ any child of v, u's *low* value, then a back edge must exist from u to some proper ancestor of v. As such, if $u.low \leq v.d$, then a backedge does not exist meaning v is an articulation point.

Because of the above, articulation points can be determined as the algorithm is performed. For example, each vertex can include an addition member in their structs named "is_art_point", a boolean value where true means the vertex is an articulation point (and false otherwise). On each iteration, simply check the current vertex, v, to see if v has no parent pointer and has multiple children (case 1 above); if v has a parent pointer, has children, and does not contain a back pointer (case 2); and, on checking children, simply store the maximum *low* value in all children and compare the $v.d$ with this maximum low value (case 3). If any of the cases are true, then v is an articulation point and `v.is_art_point = True`.

This modification adds no additional loops or recursions or any particulat actions which can not be represented in a single operation. As such, calculating all articulation points does not change the running time of the algorithm.

Problem 22-2 e

Assume some edge, $e \in V$ is a bridge edge but is also part of a cycle of G. This implicates some vertex v_i is able to reach some other vertex v_j via e. If e is indeed a bridge edge, then removal of e from the graph will cut the path between v_i and v_j. However, despite the original path being destroyed by the removal of e, because e lies in a cycle, an alternate path exists for v_i to reach v_j.

Inherently, if a path lies does not lie in a cycle, then some set of vertices have a path that must include said path to reach a separate set of vertices in the graph. Removal of the edge would destroy this path and any path which constitute the edge. However, if an edge lies on a cycle then every vertex can reach every other vertex in the cycle via the edges which constitute the cycle. If an edge is removed, every vertex can still reach every other vertex. As such, any additional vertices or subtrees attached to the cycle before the edge removal will still be able to reach each other via different paths if an edge is removed from the cycle.

Problem 22-2 f

Fortunately, G_π can be used rather than G for determining all bridges of G; a bridge, when removed, disconnects a graph. In a DFS, if a tree edge is removed, the vertices on either end, u, v can not longer reach each other, disconnecting the graph. So all bridges within G will also exist within G_π.

With the algorithm to calculate `low` for all vertices, the vertices, $v.d$ and $v.low$ can be used to determine bridges. In general, a series of edges with the same `low` score will end up being part of a simple cycle of G due to the `low` value being propagated up from a descendant with a back edge. When traversing G_π looking for cycles, series of ancestrally related vertices with the same `low` value are part of a cycle until a parent vertex which has a different `low` value. The last child vertex in question then is also the end point of the back edge which caused all intermediate vertices to have the same `low` value. As such, the vertex, v, should have $v.d = v.low$.

The condition, $v.d = v.low$ can be used to determine bridges. The edge connecting v with its predecessor then is an edge which connects a vertex in a simple cycle or as a leaf vertex to another vertex. This can be determined just at the end of a recursion where $v.low$ is assigned. If $v.d = v.low$, then the edge connecting v to its predecessor (if v is not the root of G_π) is a bridge edge. This adds no extra complexity to the algorithm beyond a check for equivalencies for two prior known values and an assignment statement of a single value to a single variable.

Problem 22-2 g

Problem 22-2 h

Problem 22-3 a

Assume G indeed already has a cycle involving vertices $\{v_j, v_{j+1}, \ldots, v_k\}$. Such a cycle implies each vertex in this cycle has an edge to its sibling. For a directed graph, in order for a cycle to exist, each node must be reachable to every other node in the cycle via some path made of some or all the edges that constitutes the cycle. For here, the edges leading from a source vertex are "out-edges" while edges leading to a node are "in-edges". "In-degree" then is the count of incoming edges whie "out-degree" is the count of outgoing edges.

In the most simple example of a cycle within a directed graph, $G = \{v_1, v_2\}$ such that $e_1 = edge(v_1, v_2), e_2 = edge(v_2, v_1)$. The cycle created herein is then an Euler Tour because all edges in G are able to be visited exactly once. For each vertex in G, the in-degree is 1 while out-degree is also 1 thus fitting the restriction of Euler's Tour where the in-degree for some vertex v must equal the out-degree for that same vertex v for all vertices in G.

This example can be extended with arbitrarily large amounts of paths and vertices. In continuing the example, let $e_3 = edge(v_1, v_2) : e_3 \not\simeq e_1$. Here, while the Euler Tour still exists between v_1 and v_2 via e_1 and e_2, e_3 can not be a part of the same Euler Tour; to maintain a cycle, the path must traverse through the previously visited edge e_2 and doing so violates a condition of Euler Tours. However, if $e_4 = edge(v_2, v_1), e_4 \not\simeq e_2$, then the properties of Euler Tours are not violated. In the violation scenario, $in\_degree(v_1) = 1, out\_degree(v_1) = 2$ and vice-versa for v_2. Adding the forth path remedied the situation causing $in\_degree(v_1) = out\_degree(v_1)$.

Inherently, for any node, $v \in V$, for a directed graph G, if G is a Euler Tour, $in\_degree(v) =$

$out\_degree(v)$ otherwise a path exists to leave a node and never return to the node. Consider the violation example above, the out-degree did not match the in-degree for v_1 so, while the cycle persisted due to e_1 and e_2, taking e_3 means taking a path that would prevent returning to v_1 via edges which have yet to be traversed.

In general, when $in\_degree(v) < out\_degree(v)$, a possible path exists where you can never return to v if all edges are traversed in G.

The other scenario is $in\_degree > out\_degree(v)$. In such a scenario, if all edges are to be traversed, eventually, v would be reached via one of the in-edge but all of its out-edges would be exhausted. So the path terminates with v.

Question 2 (8 pts total)

Please sole problem 23-4 on page 641. You do NOT have to describe the most efficient implementation of each algorithm.

> ***23-4 Alternative minimum-spanning-tree algorithms***
>
> In this problem, we give pseudocode for three different algorithms. Each one takes a connected graph and a weight function as input and returns a set of edges T. For each algorithm, either prove that T is a minimum spanning tree or prove that T is not necessarily a minimum spanning tree. Also describe the most efficient implementation, whether or not it computes a minimum spanning tree.
>
> ***a.*** MAYBE-MST-A(G, w)
>
> ***b.*** MAYBE-MST-B(G, w)
>
> ***c.*** MAYBE-MST-C(G, w)

Purpose Reinforce your understanding of minimum spanning trees

Question 3 (6 pts)

Suppose you have a weighted, undirected graph G with positive edge weights and a start vertex s. Describe a modification of Dijkstra's algorithm that runs (asymptotically) as fast as the original algorithm, and assigns a label $usp[u]$ to every vertex u in G, so that $usp[u]$ is true if and only if there is a unique shortest path from s to u. By definition $usp[s]$ is true. In addition to your modification, be sure to provide arguments for both the correctness and time bound of your algorithm, and an example.

Purpose Reinforce your understanding of Dijkstra's shortest path algorithm, and practice algorithm design

Textual description of the algorithm with pseudocode

A worked example

Proof

Time complexity analysis