Homework 2 Solutions

1  (Vulnerable edges)

(b) Fast algorithm for finding all the articulation points in $G$

This algorithm uses a single DFS tree as a starting point for identifying all the articulation points in
the graph. The following outer procedure calls a specialized version of DFS (explained below).

The value $status[v]$, stored for each node, represents the algorithm’s current assessment of each
node. “White” is unvisited, “grey” is visited. The field $parent[v]$ stores the parent of $v$ in the DFS
tree.

Algorithm 1  FindVulnerableEdges($G = (V, E)$)

1: $\forall v \in V \ status[v] \leftarrow$ white
2: $L \leftarrow \text{empty list}$ {global list of vulnerable edges}
3: $time \leftarrow 1$ {global timestamp}
4: while there exists a white vertex $v$ do
5: \hspace{1em} $parent[v] \leftarrow$ NULL
6: \hspace{1em} call DFS$\text{Mark}(G = (V, E), v)$
7: end while
8: Return $L$

We now turn to the recursive sub-procedure DFS$\text{Mark}$. This modified DFS algorithm computes
up to two auxiliary values for each node: $\text{Discover}[]$ and $\text{Backlink}[]$. The value $\text{Discover}[v]$ is a
timestamp indicating when the node $v$ was first explored (this is not a physical time, but simply the
value of a counter that is incremented with each new discovered node). This is ensured by Lines 2–3
in DFS$\text{Mark}()$. Note that we could have used the depth in the DFS tree instead of the discovery
time; we just need a quantity that increases as we go deeper in the tree.

The second value, $\text{Backlink}[v]$ is is computed by lines 10-13 and 15-17 in DFS$\text{Mark}()$. It is defined
by

$$Backlink[v] = \min \left( \{ \text{Discover}[v] \} \cup \left\{ \text{Discover}[w] : (u, w) \in E \text{ is a back edge and} \right. \right.$$
$$\left. \left. u \text{ is a descendant of } v \text{ (or } u = v \text{).} \right\} \right).$$

Claim 1  DFS$\text{Mark}()$ computes $\text{Backlink}[]$ as defined above.

Proof: Consider the following alternate characterization of $\text{Backlink}[]$:

$$\text{Backlink}[v] = \min \left( \{ \text{Discover}[v] \} \cup \{ \text{Discover}[w] : (v, w) \in E \text{ is a back edge} \right.$$
$$\left. \cup \{ \text{Backlink}[u] : u \text{ is a child of } v \} \right).$$

In Line 2, $\text{Backlink}$ is initialized to $\text{Discover}[v]$ and from then on can only decrease. Lines 15–17
ensure that $\text{Backlink}[]$ is computed correctly at the leaves (since for all back edges coming from a node,
Algorithm 2 DFS_Mark \((G = (V, E), v)\)

1: \(\text{status}[v] \leftarrow \text{grey}\)
2: \(\text{Discover}[v] \leftarrow \text{time}; \text{BackLink}[v] \leftarrow \text{time}\)
3: \(\text{time} \leftarrow \text{time} + 1\)
4: \(\textbf{for } u \text{ in } \text{adj-list}(v) \textbf{ do}\)
5: \(\textbf{if } \text{status}[u] = \text{white} \textbf{ then}\)
6: \(\text{parent}[u] \leftarrow v\)
7: \(\text{call DFS_Mark}(G = (V, E), u)\)
8: \(\textbf{if } \text{BackLink}[u] > \text{Discover}[v] \textbf{ then}\)
9: \(\text{add } (u, v) \text{ to list } L \text{ of vulnerable edges}\)
10: \(\textbf{end if}\)
11: \(\textbf{if } \text{BackLink}[u] < \text{BackLink}[v] \textbf{ then}\)
12: \(\text{BackLink}[v] \leftarrow \text{BackLink}[u]\)
13: \(\textbf{end if}\)
14: \(\textbf{else}\)
15: \(\textbf{if } \text{parent}[v] \neq u \text{ and } \text{Discover}[u] < \text{BackLink}[v] \textbf{ then}\)
16: \{\text{In this case, } (v, u) \text{ is a back edge}\}
17: \(\text{BackLink}[v] \leftarrow \text{Discover}[u]\)
18: \(\textbf{end if}\)
19: \(\textbf{end if}\)
20: \(\textbf{end for}\)

\(\text{Backlink} \) is updated). For internal nodes, assume inductively that \(\text{DFS\_Mark} \) correctly computes \(\text{Backlink}[]\) for the child \(u\). Then Lines 11–12 update \(\text{Backlink}[]\) appropriately. Lines 15–16 ensure that links coming directly from \(v\) are also counted.

Given this claim, we mark \(e = (v, u)\) as vulnerable only when \(\text{Discover}[v] < \text{Backlink}[u]\). This is justified by the following:

\textbf{Claim 2} An edge \(e\) is vulnerable if and only if either (a) it is a tree edge (not a back edge) and (b) \(\text{Backlink}[u] > \text{Discover}[v]\) where \((v, u)\) are the endpoints of the edge \(e\) and \(u\) is deeper in the tree than \(v\).

\textbf{Proof:} An edge is vulnerable if and only if it is not on any cycle. Back edges cannot be vulnerable since they are always part of a cycle (formed by using the path of tree edges to connect the endpoints).

So we’re left to decide which tree edges are vulnerable. Since the tree edges themselves cannot form a cycle, an edge \(e\) is on a cycle only if there is a back edge that “passes” it in the tree; that is, if there is back edge from one of its descendants to one of its ancestors. If \(e = (v, u)\) and \(u\) is deeper than \(v\), then we want to know if there is a back edge from a descendant of \(u\) to a node \(w\) with \(\text{Discover}[w] \leq \text{Discover}[v]\) (that is, a node \(w\) which is either \(v\) itself or an ancestor of \(v\)). Thus, we know \((v, u)\) is not vulnerable if \(\text{Backlink}[u]\) is less than or equal to \(\text{Discover}[u]\); it is vulnerable if \(\text{Backlink}[u] > \text{Discover}[v]\).

This completes the proof of correctness. The algorithm runs in time and space \(O(m + n)\) since it performs the same work as DFS, with only constant additional work for each edge in the graph.
2 El Goog

The correct greedy algorithm executes the jobs in decreasing order of \( f_i \). It takes \( O(n \log n) \) time to produce a schedule; the bottleneck is sorting.

The simplest proof of optimality is via an exchange argument. Order the jobs so that \( f_1 \geq \cdots \geq f_n \). Suppose there is some schedule \( S^* \) that does not follow this ordering. That means there must exist a pair of jobs \( i, j \) that are adjacent in \( S^* \) but out of order (that is, \( i \) comes before \( j \) but \( f_i < f_j \)).

We now show that swapping these two jobs will not increase the overall finishing time. Since jobs other than \( i \) and \( j \) are not affected, we only have to worry about the completion time of the later of these two jobs. Suppose that job \( i \) begins running on the supercomputer at time \( t \); then the later of the two jobs finishes at \( t + \max(p_i + f_i, p_i + pJ + f_j) \).

After the swap, job \( i \) will complete at time \( p_j + p_i + f_i \); this is less than the completion time of job \( j \) before the swap, since \( f_i < f_j \). Similarly, after the swap job \( j \) will complete at time \( p_j + f_j \); this is less than the completion time of job \( j \) before the swap since \( p_i > 0 \). Hence, the latest completion time cannot be increased by the swap.

Continuing to swap adjacent inverted pairs, we can eventually turn \( S^* \) into the greedy schedule without increasing the completion time. So the greedy is schedule is optimal.

Appendix

Given the many questions I received about asymptotics and basic proofs, I’ve attached solutions to a couple of the exercises.

A (More Asymptotics)

(a)

Recall L’Hospital’s rule: if \( f \) and \( g \) are differentiable functions with derivatives \( f', g' \), and \( L = \lim_{x \to \infty} f'(x)/g'(x) \) exists or is \( \pm \infty \), then \( \lim_{x \to \infty} f(x)/g(x) = L \). In particular, for an integer random variable \( n \), we get \( \lim_{n \to \infty} f(n)/g(n) = L \).
Consider real numbers $a, b > 0$ and let $t = \lceil a \rceil$:

$$\lim_{n \to \infty} \frac{n^b}{\log^a(n)} = \lim_{n \to \infty} \frac{b^{n-1}}{a \log^{a-1}(n) \cdot \frac{1}{n}} = \frac{b}{a} \lim_{n \to \infty} \frac{n^b}{\log^{a-1}(n)} = \cdots = \frac{b^a}{a(a - 1) \cdots (a - t + 1)} \cdot \lim_{n \to \infty} \frac{n^b}{\log^{a-t}(n)}.$$

This last limit must diverge to $+\infty$ since the numerator diverges and the denominator is at most 1. Hence $n^b = \omega(\log^a(n))$.

(b)

A counter example is:

$$f(n) = \begin{cases} n^n & \text{if } n \text{ is even} \\
(n-1) & \text{if } n \text{ is odd} 
\end{cases}$$

$$g(n) = \begin{cases} n^n & \text{if } n \text{ is odd} \\
(n-1) & \text{if } n \text{ is even} 
\end{cases}$$

For even $n \geq 2$, we have $f(n)/g(n) = \frac{n^n}{(n-1)(n-1)+1} > n$. Similarly, for odd $n \geq 3$, we have $g(n)/f(n) > n$. So neither function can be bounded by any constant times the other. On the other hand, both $f$ and $g$ are strictly increasing.

(c)

Consider $h(n) = \lceil n \log(n) \rceil$, and set $g(n) = n \log(n)$, so $g(n) \leq h(n) \leq g(n) + 1$. Since $g(n)$ grows to $\infty$ with $n$, we have $g(n) = h(n)(1 + o(1))$.

$$\frac{h(n)}{\log(h(n))} = \frac{g(n)(1 + o(1))}{\log(g(n))(1 + o(1))} = \frac{g(n)(1 + o(1))}{\log\log n + \log(\log n) + o(1)} = \frac{n \log(n)(1 + o(1))}{\log n + \log \log n + o(1)} = n \cdot \frac{1 + o(1)}{1 + \frac{\log \log n}{\log n} + o(1)}$$

Since $\log \log n/\log n \to 0$ as $n \to \infty$, the term multiplying $n$ in the right hand side above converges to 1 as $n \to \infty$. Thus $\lim_{n \to \infty} \frac{h(n)}{\log(h(n))} = 1$, and hence $h(n)/\log(h(n)) = \Theta(n)$. 

4
B (Basic proof techniques)

(a) (Induction) Chapter 3, problem 5.

Given a binary tree $T$, let $t$ be the number of nodes with two children in $T$. Let $\ell$ be the number of leaves in $T$. We will prove that $t + 1 = \ell$ by induction on the number of nodes in $T$. **Base case:** $T$ has one node. In this case $t = 0$ and $\ell = 1$, so the statement holds.

**Induction step:** suppose the statement holds for all trees with $n$ nodes, and prove it for a tree $T$ with $n + 1$ nodes. Consider a leaf $x$ in the tree $T$, and let $T_0$ be a tree obtained by removing $x$ from $T$. Let $t_0$ be the number of nodes with two children and $\ell_0$ be the number of leaves in $T_0$. Since $T_0$ has $n$ nodes, by induction hypothesis, $t_0 + 1 = \ell_0$.

Case 1: $x$ is the only child of its parent. Then $t = t_0$ and $\ell = \ell_0$, and therefore $t + 1 = \ell$, as required.

Case 2: $x$ is one of the two children of its parents. Then $t = t_0 + 1$ and $\ell = \ell_0 + 1$. Again, $t + 1 = \ell$, as required.

**Remark:** There is an equally valid and natural argument that proceeds by induction on the height of the tree.

(b) (Contradiction) Chapter 3, problem 7.

Proof by contradiction. Assume the opposite is true: every node in $G$ has a degree of at least $n/2$, but $G$ contains two or more disconnected components. Let $u$ and $v$ be two nodes from different connected components in $G$. Thus, there is no a path between between $u$ and $v$. Since the degree of $u$ is $n/2$, there are at least $n/2 + 1$ nodes that $v$ cannot be connected to (the nodes connects to $u$ plus $u$ itself). That leaves only $n/2 - 2$ nodes that $v$ can be connected to, which contradict with the fact that $v$ has a degree not less than $n/2$. 

5