Fundamental Algorithms, Assignment 9 Solutions

1. (*) Suppose that the Huffman Code for $\{v, w, x, y, z\}$ has 0 or 1 as the code word for z. *Prove* that the frequency for z cannot be less than $\frac{1}{3}$. *Give an example* where the frequency for z is 0.36 and z does get code word 0 or 1.

   `Solution:` For an example, let the frequencies be $0.16, 0.16, 0.16, 0.16, 0.36$ in order. Then v, w merge to, say, a with 0.32; x, y merge to b with 0.32; a, b merge to c with 0.64 and finally c, z merge, so z gets only a single bit. For the proof, the first two merges cannot involve z and so after them we have some a, b, z. (a, b might be among the original letters.) These add to 1 so that when z has frequency less than 1/3 it cannot have the biggest frequency. Hence it will be involved in the penultimate merge.

   (a) What is an optimal Huffman code for the following code when the frequencies are the first eight Fibonacci number?

   $$a : 1, b : 1, c : 2, d : 3, e : 5, f : 8, g : 13, h : 21$$

   (b) The Fibonacci sequence is defined by initial values $0, 1$ with each further term the sum of the previous two terms. Generalize the previous answer to find the optimal code when there are n letters with frequencies the first n (excluding the 0) Fibronacci numbers.

   `Solution:` The Huffman encoding tree is given in the picture on the website: In general, the encoding for the character with the first Fibonacci number will be $h_1 = 1^{n-1}$. For the character with the kth Fibonacci number will be $h_k = 1^{n-k}0$

2. Suppose that in implementing the Huffman code we weren't so clever as to use Min-Heaps. Rather, at each step we found the two letters of minimal frequency and replaced them by a new letter with frequency their sum. How long would that algorithm take, in Thetaland, as a function of the initial number of letters n.

   `Solution:` To find the letter of minimal frequency takes time $O(n)$, doing it twice, adding frequencies, and replacing them by a new letter all takes $O(n)$. We do this n times so the total time is $O(n^2)$. It is actually $\Theta(n^2)$ – as the number of letters decreases the time decreases but the first $n/2$ times there are at least $n/2$ letters and so finding the minimum takes $\Omega(n)$ and so the total time for the first $n/2$ times (that is, starting with n letters until there are $n/2$ letters left) is $\Omega((n/2)\cdot(n/2)) = \Omega(n^2)$. We've sandwiched upper and lower bounds so this gives total time $\Theta(n^2)$.

3. Consider the undirected graph with vertices $1,2,3,4,5$ and adjacency lists (arrows omitted) $1:25$, $2:1534$, $3:24$, $4:253$, $5:412$. Show the d and π values that result from running BFS, using 3 as a source. Nice picture, please!
   `Solution:`
   BFS: $3,2,4,1,5$
   $d[3]=0,\ \pi[3]=nil$
   $d[2]=1,\ \pi[2]=3$
   $d[4]=1,\ \pi[4]=3$
   $d[1]=2,\ \pi[1]=2$
   $d[5]=2,\ \pi[5]=2$

4. Show the d and π values that result from running BFS on the undirected graph of Figure A, using vertex u as the source.
   `Solution:`
   $d[U]=0,\ \pi[U]=nil$
   $d[T]=1,\ \pi[T]=U$
   $d[X]=1,\ \pi[X]=U$
   $d[Y]=1,\ \pi[Y]=U$
   $d[W]=2,\ \pi[W]=T$
   $d[S]=3,\ \pi[S]=W$
   $d[R]=4,\ \pi[R]=S$
   $d[V]=5,\ \pi[V]=R$

5. We are given a set V of boxers. Between any two pairs of boxers there may or may not be a rivalry. Assume the rivalries form a graph G which is given by an adjacency list representation, that is, $Adj[v]$ is a list of the rivals of v. Let n be the number of boxers (or nodes) and r the number of rivalries (or edges). Give a $O(n+r)$ time algorithm that determines whether it is possible to designate some of boxers as `GOOD` and the others as `BAD` such that each rivalry is between a `GOOD` boxer and a `BAD` boxer . If it is possible to perform such a designation your algorithm should produce it.

   Here is the approach: Create a new field `TYPE`$[v]$ with the values `GOOD` and `BAD`. Assume that the boxersare in a list L so that you can program: For all $v \in L$. The idea will be to apply `BFS[v]` – when you hit a new vertex its value will be determined. A cautionary note: `BFS[v]` might not hit all the vertices so, just like we had `DFS` and `DFS-VISIT` you should have an overall `BFS-MASTER` (that will run through the list L) and, when appropriate, call `BFS[v]`.

`Note:` The cognescenti will recognize that we are determining if a graph is bipartite!
`Solution:`The idea here is to call the first boxer`GOOD`. When someone is adjacent to someone `GOOD` they are called `BAD` and if they are adjacent to someone `BAD` they are called `GOOD`. But if in the adjacency list you come upon someone who has already been labelled (that is, not white) then you must check if there is a contradiction. A further problem: `BFS[v]` will only explore the connected component of v, if that is labelled with no contradiction then you must go on to the other vertices. So we start with everything white. The "outside" program is:
For all $v \in L$
If $COLOR[v] = WHITE$ (*else skip*) then `BFSPLUS[v]`.

`BFSPLUS[v]` starts by setting `TYPE`$[v] = GOOD$. Then it runs `BFS[v]` with two additions. When $u \in Adj[w]$ and u is white you define `TYPE`$[u]$ to be the opposite of `TYPE`$[w]$. When u is not white you check if `TYPE`$[w]$ = `TYPE`$[u]$. If not, ignore. But if so exit the entire program with `NO DESIGNATION POSSIBLE` printout.

6. Show how DFS works on Figure B. All lists are alphabetical, except that we put R before Q so it is the first letter. Show the discovery and finishing time for each vertex.
`Solution:`
$Discovery\ order : RUYQSVWTXZ$
$Finishing\ order : WVSZXTQYUR$
$Stack : push(R)\ push(U)\ push(Y)\ push(Q)\ push(S)\ push(V)\ push(W)$
$pop(W)\ pop(V)\ pop(S)\ push(T)\ push(X)\ push(Z)\ pop(Z)$
$pop(X)\ pop(T)\ pop(Q)\ pop(Y)\ pop(U)\ pop(R)$

7. Show the ordering of the vertices produced by `TOP-SORT` when it is run on Figure C, with all lists alphabetical.
`Solution:`We apply `DFS` to the graph. The first letter is M so we apply `DFS-VISIT(M)`

v	s[v]	f[v]
M	1	20
Q	2	5
T	3	4
R	6	19
U	7	8
Y	9	18
V	10	17
W	11	14
Z	12	13
X	15	16

Note, for example, that though X is in Adj[M] it doesn't affect DFS. At time 19 R finishes and returns control to M. M looks at X in its adjacency list but it is no longer white and so ignores it. At this stage all vertices are black except N, O, P, S which as white. In this particular example N is the letter right after M but in the general case DFS would skip over those vertices which weren't white. Indeed, right after DFS-VISIT all vertices are white or black. So next we do `DFS-VISIT(N)`. Note that the time does *not* restart! Note also that the now black vertices, such as $U \in Adj(N)$ and $R \in Adj(O)$, do not play a role

v	s[v]	f[v]
N	21	26
O	22	25
S	23	24

Finally we do `DFS-VISIT(P)`. This one is quick. The adjacency list of P consists only of S which is already black. So

v	s[v]	f[v]
P	27	28

The sort is the list of vertices in the reverse order of their finish. In the algorithm when a vertex finishes we place it at the *start* of a linked list, initially nil. At the end, with negligible extra time, we have the list:

$$PNOSMRYVXWZUQT$$

8. Let G be a DAG with a specific designated vertex v. Uno and Dos play the following game. A token is placed on v. The players alternate moves, Uno playing first. On each turn if the token is on w the player moves the token to some vertex u with (w, u) an edge of the DAG. When a player has no move, he or she loses. Except for the first part below, we assume Uno and Dos play perfectly.

   (a) Argue that the game *must* end.
   `Solution:`Let G have V vertices. If the game went on for V moves the chip would hit $V + 1$ positions $v = v_0, v_1, \ldots, v_V$ and so some position would be hit twice – some $i < j$ with $v_i = v_j$ – but that gives a cycle $v_i v_{i+1} \cdots v_{j-1} v_i$.

   (b) Define `VALUE[z]` to be the winner of the game (either Uno or Dos) where the token is initially placed at vertex z and Uno plays first. Suppose the `VALUE[w]` are known for all $w \in Adj[z]$. How do those values determine `VALUE[z]`.
   `Solution:`Suppose there is some $w \in Adj[z]$ with `VALUE[w]` equal Dos. Uno makes that move. Now, as the roles are reversed and Dos must move first so Uno wins. Therefore `VALUE[z]` is Uno. If there is no such w then whatever move Uno makes a position w is reached with `VALUE[w]` equal Uno. But this means the the player making the first move will win, and that player is Dos. Therefore `VALUE[z]` is Dos.

   (c) Using the above idea modify DFS to find who wins the original game. Give an upper bound on the time of your algorithm.
   `Solution:`Apply `DFS-VISIT[v]` with an additional field `VALUE` We can implement the previous part in several ways. The easiest is to wait until a vertex z has become black. At that time check the `VALUE` (they will already have been determined) of all $w \in Adj[z]$. If any is Dos, set `VALUE[z]` to be Uno, otherwise (this includes the case where $Adj[z]$ is empty!) set `VALUE[z]` to be Dos. The time is $O(V + E)$. It could be considerably smaller than $V + E$ as `DFS-VISIT[v]` might only reach a small part of the graph.