Solutions to Midterm 2 - Monday, July 11th, 2009

CPSC320, Summer2009. Instructor: Dr. Lior Malka. (liorma@cs.ubc.ca)

---

1. **Dynamic programming.** Let A be a set of n integers $A_1, \ldots, A_n$ such that $1 \leq A_i \leq n^2$ for each $1 \leq i \leq n$. Let S be a subset of A. We use the following notation:

   - $\bar{S}$ is the set of all A_is that are not in S. That is, $S \cup \bar{S} = A$, and $S \cap \bar{S} = \emptyset$.
   - $\Sigma(S)$ is the sum of all the elements in S. That is, $\Sigma(S) = \sum_{A_i \in S} A_i$.
   - Similarly, we define $\Sigma(\bar{S}) = \sum_{A_i \in \bar{S}} A_i$, and $\Sigma(A) = \sum A_i$.

   (a) Find a subset S of $A = \{1, 20, 4, 9\}$ such that $\Sigma(S) - \Sigma(\bar{S})$ is minimal.

   `Answer.` $S = \emptyset$.

   (b) Find a subset S of $A = \{1, 20, 4, 9\}$ such that $|\Sigma(S) - \Sigma(\bar{S})|$ is minimal.

   `Answer.` $S = \{20\}$ or $S = \{1, 4, 9\}$.

   (c) Find a subset S of $A = \{11, 16, 7, 3, 4, 19\}$ such that $|\Sigma(S) - \Sigma(\bar{S})|$ is minimal. *Hint: find S such that $\Sigma(S)$ is as close to $\frac{\Sigma(A)}{2}$ as possible.*

   `Answer.` $S = \{11, 19\}$ or $S = \langle 16, 7, 3, 4 \rangle$

   (d) Let S such that $|\frac{\Sigma(A)}{2} - \Sigma(S)|$ is minimal. Prove that $|\Sigma(S) - \Sigma(\bar{S})|$ is also minimal.

   `Answer.` If S minimizes $|\frac{\Sigma(A)}{2} - \Sigma(S)|$, then it minimizes $2|\frac{\Sigma(A)}{2} - \Sigma(S)| = |\Sigma(A) - 2\Sigma(S)| = |\Sigma(S) + \Sigma(\bar{S}) - 2\Sigma(S)| = |\Sigma(S) - \Sigma(\bar{S})|$. Notice that the opposite is also true.

   (e) We define $m[i, j]$ to be `TURE` if there is a subset S of $\{A_1, \ldots, A_i\}$ such that $\Sigma(S) = j$, and `FALSE` otherwise. Describe an algorithm that on input the set A and the values $m[i, j]$ would compute the minimum value of $|\Sigma(S) - \Sigma(\bar{S})|$. Do not write pseudocode, but be concise.

   `Answer.` The algorithm computes $s = \frac{\Sigma(A)}{2}$ by adding the values of the elements in A and dividing by 2. It then finds the minimum of $|m[n, j] - s|$ by ranging over all the possible j. It outputs the j that minimizes $|m[n, j] - s|$.

(f) Define $m[i,j]$ recursively, in terms of smaller values of i or j. *Hint: in terms of m, under what conditions would a subset of $\{A_1,\ldots,A_i\}$ sum to j?*

`Answer.` $m[i,j]$ =`TRUE` if $m[i-1,j]$ is true or $m[i-1,j-A_i]$ is true. Otherwise, $m[i,j]$ =`FALSE`. The definition covers all cases. That is, if $m[i,j]$ =`TRUE`, then either A_i is part of the sum, which implies that a subset of $A_1,\ldots,A_{i-i}$ sums to $j-A_i$, or A_i is not included in the sum, which means that a subset of $A_1,\ldots,A_{i-1}$ sums to j.

(g) What is the time and memory complexity of an algorithm that on input A outputs the minimum value of $|\Sigma(S)-\Sigma(\bar{S})|$?

`Answer.` Each of $A_1,\ldots,A_n$ is a number between 1 to n^2. The sum of these numbers is at most $n\cdot n^2=n^3$. Thus, the table $m[i,j]$ has n^4 entries, each requiring a constant amount of time to compute. As we saw in Section (e), finding the minimum requires examining $m[1,j]$ for all j. Thus, both time and memory complexity are $\Theta(n^4)$.

2. **Huffman encoding.** Let F be a file containing n characters.

(a) Draw the Huffman tree for the case where F has 8 different characters with the following frequencies: $\langle 10, 10.25, 10.5, 11, 12, 13, 13.25, 20\rangle$.

`Answer.` The tree is a perfect binary tree with four levels (height 3). The leaves, from left to right, are $20, 13.25, 13, 12, 11, 10.5, 10.25$, and 10. At each level of the tree, two adjacent nodes at location $2k$ are called *a pair*. Thus, $\langle 10.25, 10\rangle$ is a pair, $\langle 11, 10.5\rangle$ is a pair, and so on. The next level up contains nodes summing pairs of leaves. From left to right, the weights of these nodes are $33.25, 25, 21.5$, and 20.25. Pairs of these nodes yield the next level up, with nodes 58.25 and 41.75. The root is 100. The edges of the tree are labeled 0 (for left edges) and 1 (for right edges)

(b) What is the **byte size** of F from Section (a) after you encode it using your tree?

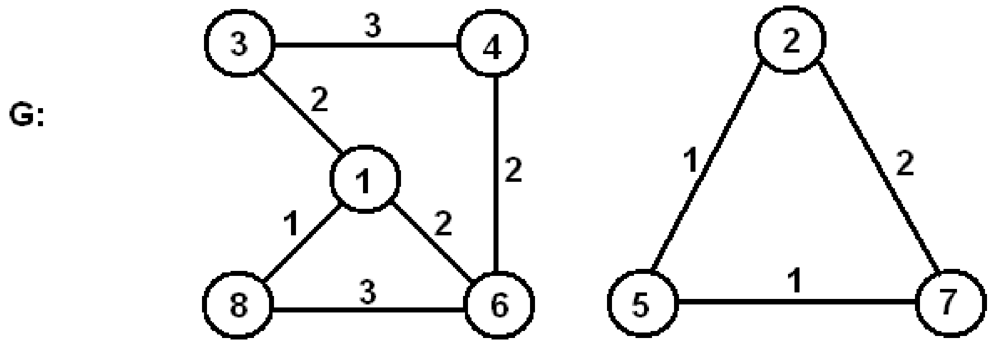
`Answer.` Each character is encoded using 3 bits, and there are n characters. Since a byte is 8 bits, the byte size of F after encoding is $\frac{3n}{8}$.

(c) Consider the case where each character in F is an ASCII symbol, and all the characters have "close" frequencies. Formally, if the lowest frequency is x, then the highest frequency is $2x$. Prove that in this case the the size of F after encoding equals its size before encoding.

`Answer.` Since there are 128 ASCII characters, the tree is a perfect binary tree with eight levels (height $7=\log(128)$). Proof: at each level of the tree, two adjacent nodes at location

$2k$ are called *a pair*. Since the lowest frequency is x and the highest is $2x$, the sum of a pair of leaves is between $2x$ and $4x$. Thus, the first 128 nodes created by the algorithm join pairs of leaves. The next level up has 128 nodes. Since each one of them contains frequency between $2x$ and $4x$, the same argument applies, and pairs of these nodes are joined to form a new node with frequency between $4x$ to $8x$. This continues until the root. Since the tree is perfect and has height 7, each codeword is of length 7. Thus, when we encode F and replace each ASCII character with its encoding, we use the same number of bits. We conclude that the size of F before encoding equals its size after encoding. Remark: this proof works for any number 2^k of characters (in this case $k = 7$).

3. **Representation of graphs.** Consider the graph G described below. Recall that given an *undirected* graph G, we say that H is a *component* of G if H is a subgraph of G and for any two vertices u and v in H there is a path from u to v. Thus, G has two components.



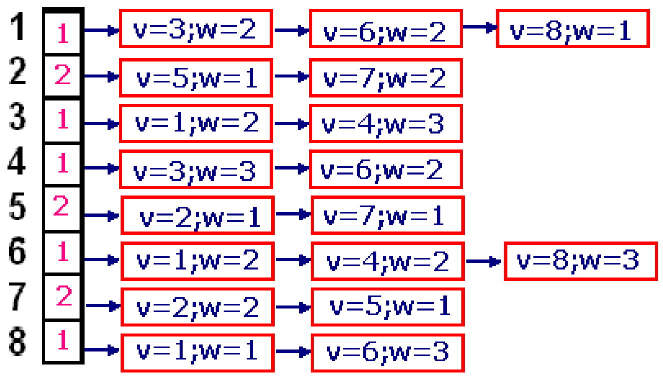
(a) Represent G using an adjacency matrix.

`Answer.` The following matrix A has $A[i, j] = w$ if and only if there is an edge between vertex i and vertex j whose weight is w. Otherwise, $A[i, j] =$`null`. Other conventions, like $A[i, j] = 0$ if there is no edge between vertex i and vertex j, are also possible.

1			2			2		1
2					1		2	
3	2							
4						2		
5		1					1	
6	2			2				3
7		2			1			
8	1					3		
	1	2	3	4	5	6	7	8

(b) Represent G using an adjacency-list. In your array A you should also store in $A[i]$ the number of the component to which vertex i belongs. You can number the components in any order, starting at 1. Use the convention that if $i < j$, then vertex i appears in the list before vertex j.

`Answer.` In the following adjacency list A, there is a list item v= j; w= k in $A[i]$ if and only if there is an edge between vertex i and vertex j whose weight is k. In addition to storing a pointer, $A[i]$ also stores the component in which vertex i resides.


1 | 1 → v=3;w=2 → v=6;w=2 → v=8;w=1
2 | 2 → v=5;w=1 → v=7;w=2
3 | 1 → v=1;w=2 → v=4;w=3
4 | 1 → v=3;w=3 → v=6;w=2
5 | 2 → v=2;w=1 → v=7;w=1
6 | 1 → v=1;w=2 → v=4;w=2 → v=8;w=3
7 | 2 → v=2;w=2 → v=5;w=1
8 | 1 → v=1;w=1 → v=6;w=3

4. **Minimum Spanning Trees.** Let H be a graph represented using an adjacency-list A such that $A[i]$.`component` stores the number of the component to which vertex i belongs. Just like we defined spanning trees, we define a *spanning forest* of H to be a forest of H that includes all the vertices of H. Similarly, we say that F is a *minimum* spanning forest of H if F is a spanning forest of H with the minimal weight.

(a) Draw the minimum spanning forest of G from Question 3.

`Answer.` The spanning forest of G is all the vertices of G with the edges $\langle 2, 5\rangle$, $\langle 5, 7\rangle$, $\langle 1, 3\rangle$, $\langle 1, 6\rangle$, $\langle 1, 8\rangle$, and $\langle 6, 4\rangle$.

(b) Suppose that someone modifies H into a graph H' such that in H' each component is connected to exactly two other components. They connect components i and j by:

- taking the vertex u with the smallest index from component i,
- taking the vertex v with the smallest index from component j,
- adding the edge $\langle u, v\rangle$ to the adjacency-list representing the graph H, and
- picking a random number r which is smaller than the weight of any edge in H, and assigning to $\langle u, v\rangle$ weight r.

Except for the above, nothing else in the representation of H is modified. Given H' and the weight of a minimum spanning forest in H, describe an efficient algorithm that computes the weight of a minimum spanning tree in H'. Do not write pseudocode, but be concise.

`Answer.` The first observation is that the graph H' is like a chain of components. Specifically, component c is connected to component $c+1$, and the last component is connected to the first one. Thus, denoting the number of new edges by k, we conclude that the minimum spanning tree of H' must include at least $k-1$ of the new edges, or else it will not be a tree. The second observation is that, since the components are always connected by choosing the vertex with the smallest index, the new edges are actually connected to each other. It follows that by adding to the spanning forest in H all the new edges, except for the heaviest one, we get a spanning tree in H'. The problem is that we do not know how many new edges were added to H' and where. To find these edges, we use the information about the components. Specifically, only the new edges $\langle i, j\rangle$ have $A[i]$.`component`$\neq$ $A[j]$.`component`. Thus, to output the weight of a minimum spanning tree in H', we scan the entire adjacency list, add the weights of these edges, and at the end we reduce from that the weight of the heaviest new edge. The running time of this algorithm is $O(|E_{H'}|)$, where $E_{H'}$ is the set of edges of H'.

5. **Connected components.** In this question you will use the `DFS` algorithm (described in Appendix 1) to compute the connected components of an undirected graph. We assume that the input graph is represented using an adjacency-list A, and that we can store in $A[i]$.`component` the number of the component to which vertex i belongs. We also assume that in the edge list of the input graph, vertex i appears before vertex j if $i < j$.

   (a) Let G be the graph from question 3. If we call `DFS(`G`)`, what is the color of the vertices at the moment where we increase the time counter from 8 to 9?

   `Answer.` At $time = 0$ all vertices are colored white. The table below shows the changes to the colors as the $time$ variable increases. The order in which we visit the vertices depends on the representation of G, in this case the adjacency-list from Question 3 (b). When the $time$ variable is increased from 8 to 9, the vertices $2, 5, 7$ are white, the vertices $3, 4, 6, 8$ are black, and vertex 1 is grey.

DFS(G) at $time$=9		
Vertex No.	Color	Time
1	grey	1
3	grey	2
4	grey	3
6	grey	4
8	grey	5
8	black	6
6	black	7
4	black	8
3	black	9

   (b) Modify the `DFS` algorithm so that each vertex stores the component to which it belongs. That is, when the algorithm terminates, v.component= i if and only if v is in component i. Number the components starting at 1. Write your pseudo code on page 6.

`Answer.` If the `DFS-Discover` sub-routine starts at a vertex v, then it visits all the vertices in the component of v. When `DFS-Discover` terminates, the `DFS` sub-routine chooses another vertex from a component different from that of v, and invokes `DFS-Discover` on that vertex. Thus, we make two modifications: in `DFS` we increase the component counter, and in `DFS-Discover` we update the component variable of the vertex. The pseudocode is in bold (see the Appendix).

1 Appendix

The *Depth First Search (DFS)* algorithm takes a graph as an input, and recursively traverses all the vertices in the graph. Initially, the algorithm colors all vertices in white. When a vertex v is discovered for the first time, the algorithm changes the color of v from white to gray. The idea behind using colors is to prevent looping forever. The algorithm recursively discovers the neighbors of v, and when all recursive calls finish, the algorithm changes the color of v from grey to black. This continues until all vertices are colored black. In the following pseudocode there is also a time counter that records, in each vertex v, the time at which v is discovered, and the time at which v is finished.

```
DFS(G)
01    $time = 0$ // global variable
00    component = 0 // global variable
02    For each $v \in G$
03        $v$.color = white
04    For each $v \in G$ // scan vertices according to increasing index (that is, $1, 2, 3, ..$)
05        if $v$.color = white
00            component++
06            DFS-Discover(v)

DFS-Discover(v)
07    $v$.color = grey
00    $v$.component = component
08    $time{+}{+}$
09    $v$.discoveryTime = $time$
10    For each $u \in Adj(v)$ // scan neighbors $u$ of $v$, according to their order in the adjacency-list
11        if $u$.color = white
12            DFS-Discover($u$)
13    $v$.color = black
14    $time{+}{+}$
15    $v$.finishTime = $time$
```