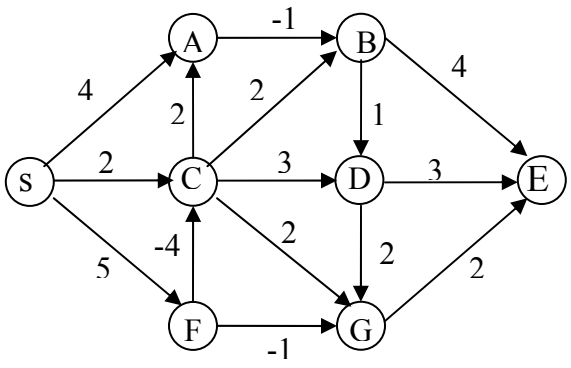
CS4231: Analysis of Algorithms I, Fall 2018

Final Exam Solutions, December 6, 2018

Problem 1 [20 points] Graded by Niloofar

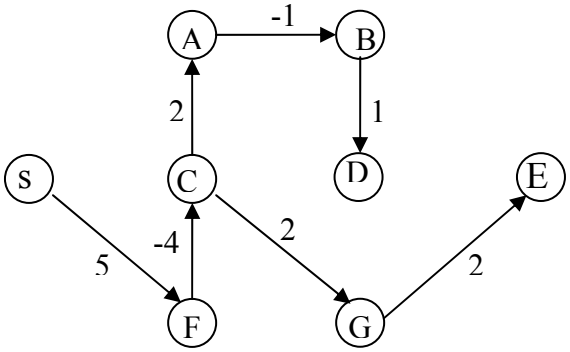
1. [10 points] Consider the following weighted directed graph with the edges weighted as indicated in the figure. (Note, some of the weights are negative.)



Write down in the following table the shortest (minimum weight) distances from node s to all the other nodes, and draw below it a shortest path tree from s. You do not need to provide any justifications.

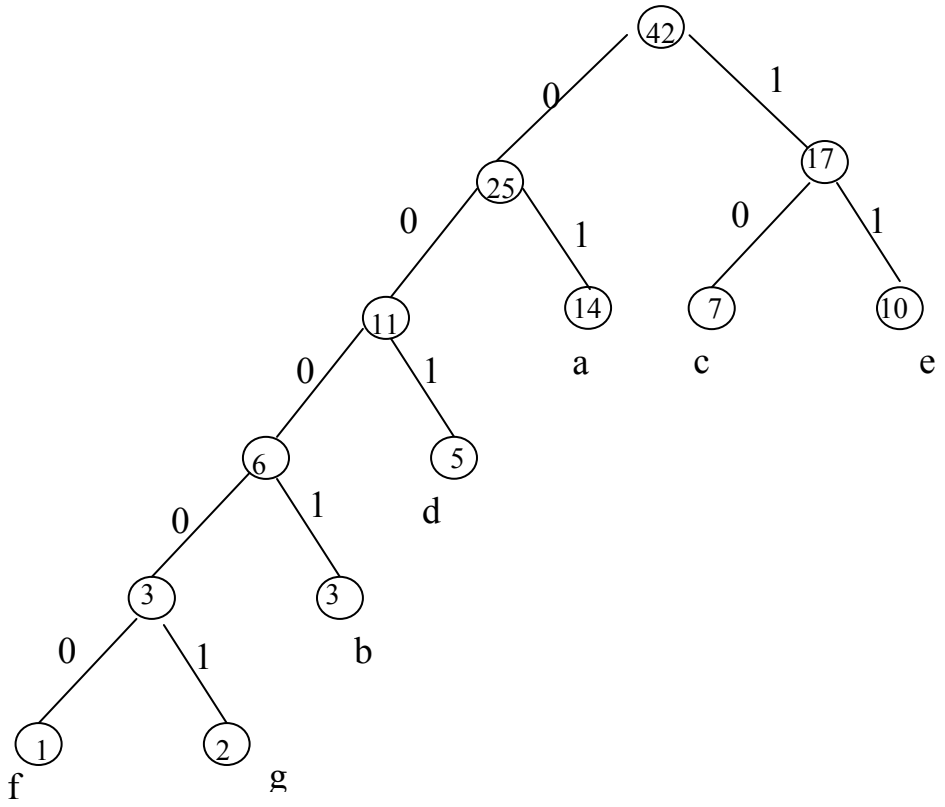
A	B	C	D	E	F	G
3	2	1	3	5	5	3

Shortest path tree:



2. [10 points] Give an optimal Huffman code for the alphabet {a,b,c,d,e,f,g} with the following frequencies: a:14, b:3, c:7, d:5, e:10, f:1, g:2. Show the code in terms of the corresponding optimal tree. You do not need to provide any justification.

Solution: The following figure shows an optimal tree.



Attached to each node in the figure is the sum of the frequencies of the letters in the subtree rooted at the node.
The codewords are the strings formed by the edge labels on the paths from the root to the leaves. That is, the codes for the different letters are as follows.
a:01, b:0001, c:10, d:001, e:11, f:00000, g:00001

Other optimal trees and corresponding codes can be obtained by switching left and right children at any subset of internal nodes in the above tree, or equivalently by switching the 0 and 1 labels of the edges from any subset of internal nodes.

Problem 2 [30 points, 6 points per part] Graded by Ananth and Rabia

For each of the claims below circle True or False, and justify your answer. *Your justifications are more important than the True/False designations, and must be* ***succinct, precise and convincing****.*
All the parts can be adequately answered and justified in a few lines.

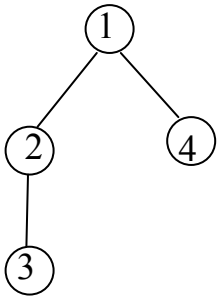
All the times below refer to worst-case time.

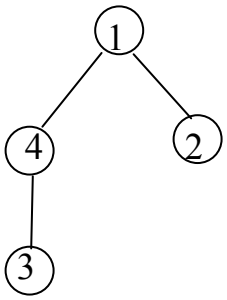
a. Recall that a graph *G* has in general many adjacency list representations because the adjacency list of each node contains the adjacent nodes in arbitrary order. Suppose that we perform Breadth First Search of an undirected graph from a given source node *s*.

Claim: The BFS tree does not depend on the adjacency list representation of the graph, i.e. all representations yield the same BFS tree.

False

Suppose for example that the graph is a cycle with 4 nodes 1,2,3,4, and we perform a BFS starting from node 1. If the adjacency list of node 1 is <2,4> then we will obtain the left tree below (where parent(3)=2), whereas if it is <4,2> we will obtain the right tree (where parent(3)=4).





For larger graphs, the BFS trees in general may be non-isomorphic; for example, if the graph consists of the above 4-cycle with an additional edge (2,5).

b. We are given a weighted undirected graph G with positive integer weights on the edges, and a minimum spanning tree T.

Claim: If we square the weights of all the edges, then T is still a minimum spanning tree of the graph with the new weights.

True

As mentioned in class, the minimum spanning tree depends on the relative order of the edge weights and not on the exact values of the weights. For example, we showed that a spanning tree T is a minimum spanning tree if and only it has the property that every edge (u,v) that is not in the tree has weight greater than or equal to the weight of every edge on the tree path from u to v. Since T is a minimum spanning tree, it has this property with respect to the original weights. After we square all the edge weights, the new weights obviously still satisfy this property since the weights are positive: the new weight of every edge (u,v) that is not in the tree is still greater than or equal to the new weight of every edge on the tree path from u to v. Therefore, T is a minimum spanning tree with respect to the new weights.

c. We are given a directed graph $G=(N,E)$ with node set $N=\{1,\ldots,n\}$ and edge set E, by its adjacency list representation.

Claim: We can compute all cycles of length 2 of G in $O(|N|+|E|)$ time and space.

True

A cycle of length 2 consists of two opposite edges (i,j) and (j,i). Thus, we need to find all edges (i,j) such that the graph contains also the reverse edge (j,i).
We can do this as follows.

1. Scan the adjacency list of all nodes to form a list L that contains all the edges and their reverses. That is,

```
L = empty list
for each i in N do
   for each j in Adj[i]  { add (i,j) and (j,i) to L }
```

2. Radix Sort L
3. Scan the sorted list L and for each pair (i,j) that is equal to the previous pair and i<j output the cycle {(i,j), (j,i)}. (We do this only if i<j, so that we print every 2-cycle once.)

Correctness: If {(i,j), (j,i)} is a cycle of length 2, then step 1 will add two copies of the pair (i,j) to L (and of (j,i)), the duplicates will be next to each other after the radix sorting of step 2, and then step 3 will print the cycle (once, only for i<j). Conversely, if the algorithm prints the cycle {(i,j), (j,i)}, i.e. L has a duplicate pair (i,j) with i<j, this implies that the graph contains both the edge (i,j) and the reverse edge (j,i) (because the graph

contains only one copy of an edge), hence G contains the cycle {(i,j), (j,i)}.

Time: Step 1 takes O($|N|+|E|$) time; Step 2 radix sorts the $2|E|$ pairs of the List L whose elements are drawn from a domain of size $|N|$, thus takes also O($|N|+|E|$) time , and step 3 that scans the sorted list L takes time O($|E|$). Thus, the total time is O($|N|+|E|$) .

d. We are given a directed graph $G=(N,E)$ and two disjoint subsets U, V of the nodes. We want to find a minimum-size subset F of edges whose deletion disconnects U from V, i.e., a subset F with as few edges as possible such that the graph $G\text{-}F=(N,E\text{-}F)$ does not contain any path from any node of U to any node of V.

Claim: This problem can be solved in polynomial time.

True

1. Create a flow network G' as follows: Give capacity 1 to all edges of G, add a source node s, a sink node t, and add edges of infinite capacity from s to all nodes of U, and edges of infinite capacity from all nodes of V to t.
2. Compute a minimum s-t cut (S,T) in the network G'.
3. Output the set F of forward edges of the cut, i.e. the edges that are directed from S to T.

The algorithm runs in polynomial time since the max flow/min cut can be computed in polynomial time.

Correctness: Every subset of edges of G that disconnects U from V induces an s-t cut of the network G' of capacity equal to the number of edges in the subset. Conversely, every s-t cut of G' with finite capacity gives a set of edges of G that disconnects U from V, since all nodes of U must be on the same side as s and all nodes of V must be on the same side as t because of the infinite capacity edges. Therefore, a minimum s-t cut gives a minimum size subset F of edges of G whose deletion disconnects U from V.

e. *Claim.* If problems A and B are NP-complete and A is in P then B is also in P.

True

Problem B is in NP, since it is NP-complete. Since A is NP-complete, all problems in NP reduce in polynomial time to A, therefore $B \leq_p A$. Since $B \leq_p A$ and A is in P, it follows that B is also in P.

Problem 3. [15 points] Graded by Aditya and Malhar

We are given a directed graph G=(N,E) by its adjacency list representation, and a partition of its nodes into red and blue nodes, i.e., sets R, B, where $R \cup B = N$ and $R \cap B = \emptyset$. We want to determine whether there exists a simple cycle in G that contains both a red node and a blue node, i.e. a simple cycle C such that $C \cap R \neq \emptyset$ and $C \cap B \neq \emptyset$.

Give an algorithm that solves this problem in time O(|N|+|E|). Justify the correctness of your algorithm and the running time.

If you cannot achieve linear time, give the most efficient algorithm you can, for partial credit, and state explicitly its running time.

Solution:

Claim: There is a simple cycle that contains both a red and a blue node if and only if there is a strongly connected component (scc) that contains both a red and a blue node. In proof, suppose first that there is a simple cycle C that contains red node u and blue node v. Then u and v (and all nodes of the cycle) belong to the same scc. Conversely, suppose that there is a scc K that contains both a red and a blue node. Then there is clearly an edge from some node u in $K \cap R$ to some node v in $K \cap B$. Since K is strongly connected, there is a path from v to u; this path together with the edge (u,v) form a simple cycle that contains both a red and a blue node.

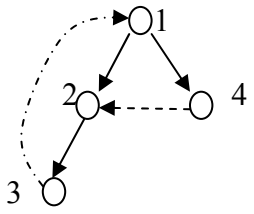
The claim leads to the following algorithm.

1. Find the strongly connected components of *G*, and compute an array SCC indexed by the nodes which records the component of each node.
2. For every node u in R give the component SCC[u] color red and for every node v in B give SCC[v] color blue. If a scc receives both colors then return "Yes", else return "No".

Correctness follows from the Claim.

Step 1 takes time O(|N|+|E|) and step 2 takes time O(|N|), thus the total time is O(|N|+|E|).

Remark: Note that doing a DFS of the graph and checking only the cycles formed by a back edge and tree edges (these are called fundamental cycles) is not sufficient. There are less than |E| fundamental cycles, while in general there can be exponentially many simple cycles in a graph. It is possible that all fundamental cycles have nodes of only one color, but there is another bicolored simple cycle. For example, this is the case in the following graph, where the solid edges are the edges of the DFS tree, nodes 1,2,3 are blue, and node 4 is red.



Problem 4. [15 points] Graded by Shenzhi and Ziqing

We are given n points $x_1 < x_2 < \ldots < x_n$ on the real line in sorted order, and a positive integer $k \le n$. We want to find k closed intervals whose union contains the n given points $x_1, x_2, \ldots, x_n$, such that the sum of the lengths of the intervals is as small as possible. For example, if k=1, the optimal solution is the interval $[x_1, x_n]$; if k=n, the optimal solution consists of the n trivial intervals $[x_i, x_i]$, i=1,…,n.

Give an algorithm that solves this problem in O(n) time. Justify the correctness of your algorithm and the running time. (Note that k is part of the input, not a constant.)

If you cannot achieve O(n) time, give the most efficient algorithm you can, for partial credit, and state explicitly its running time.

(Hint: What is the optimal solution if k=2? If k=3?)

Solution.

Let S^* be an optimal solution with k intervals. We note first that every interval of S^* must contain some point x_i , otherwise it can be discarded. Second, we observe that the endpoints of all the intervals of S^* must be among the given points $x_1, x_2, \ldots, x_n$. Suppose to the contrary that an endpoint of an interval $[a,b]$ of S^* is not one of the input points. If $a \notin \{x_1,...,x_n\}$ and the smallest input point greater than a is x_i, then we can replace $[a,b]$ by the interval $[x_i ,b]$ in S^* to get a better solution. Similarly if $b \notin \{x_1,...,x_n\}$. A third observation is that the intervals of S^* do not overlap, for if two intervals overlap, then we can reduce one of them and still cover all the input points. Thus, the optimal solution S^* is a set of k disjoint intervals $[x_1, x_{i_1}],[x_{i_1+1}, x_{i_2}],[x_{i_2+1}, x_{i_3}],...,[x_{i_{k-1}+1}, x_n]$ for some indices i_1, i_2, …, i_{k-1}. Their union is the interval $[x_1, x_n]$ minus the k-1 gaps $(x_{i_1}, x_{i_1+1}),(x_{i_2}, x_{i_2+1}),...,(x_{i_{k-1}}, x_{i_{k-1}+1})$. The cost of the solution S^* (the sum of the lengths of its intervals) is $x_n - x_1$ (the length of the interval $[x_1, x_n]$) minus the sum of the lengths of the k-1 gap intervals $(x_{i_1}, x_{i_1+1}),(x_{i_2}, x_{i_2+1}),...,(x_{i_{k-1}}, x_{i_{k-1}+1})$. Minimizing the cost of the solution is equivalent to maximizing the sum of the lengths of the k-1 gap intervals. Since S^* has minimum cost it must be the case that the corresponding k-1 gap intervals must be the k-1 longest intervals among the intervals (x_1,x_2), (x_2,x_3),… (x_{n-1},x_n), because if there was another such interval (x_j, x_{j+1}) that was longer than one of the gap intervals of S^*, then we could exchange them and obtain a better solution.

Thus, we can compute an optimal solution by finding the k-1 longest intervals among the intervals (x_1,x_2), (x_2,x_3),…, (x_{n-1},x_n), and then forming the corresponding solution that leaves these as the gap intervals.

1. Form the multiset $F = \{x_{j+1} - x_j \mid j = 1,...,n-1\}$ of the lengths of the intervals between consecutive points of the given list of n points. (Note: F may have duplicate elements because there may be intervals (x_j, x_{j+1}) with the same length.)

2. Use the linear time selection algorithm to select the (k-1)th largest element v of F, and let r be the number of elements of F that are strictly greater that v.

3. Compute an optimal solution S:

S = empty list;
t=0 [t is the number of gaps so far with length =v]
a=x_1 [a is the left endpoint of the current interval in the solution]
for j=1 to n-1 do
  { if ($x_{j+1} - x_j > v$) then { add [a,x_j] to S; a=x_{j+1} }
    if ($x_{j+1} - x_j = v$ and $t < k$-1-r) then { add [a,x_j] to S; a=x_{j+1} ; t=t+1 }
  }
add [a,x_n] to S
return S

The body of the loop terminates the current interval of S and creates a gap if the next interval (x_j, x_{j+1}) has length strictly greater than v (there are r such events), or if the next (x_j, x_{j+1}) has length equal to v and the counter t is < k-1-r (there are k-1-r such events); thus, the loop adds k-1 intervals to S, and then after that the algorithm finishes the last (the kth) interval and adds it to S.

Each step of the algorithm takes O(n) time, so the total time is O(n).

Problem 5. [20 points] Graded by Xingyu and Zhenchen

Given an unlimited supply of coins of denominations c_1, c_2,…, c_n, we wish to make change for a given value V, if possible; that is, we wish to find a set of coins whose total value is V, where we are allowed to use multiple coins of the same denomination if we wish. It might not be possible in general to make change for V; for example if the denominations are 5, 8 then we can make change for 18 (e.g. 18=5+5+8), but we cannot make change for 14.

Give an algorithm, which takes as input positive integers c_1, c_2,…, c_n, and V and solves the above coin-changing problem in O(nV) time. The algorithm should either determine that it is impossible to make change for V, or it should output a set of coins whose sum is equal to V. Justify the correctness and running time of your algorithm.

(Hint: Use dynamic programming.)

Solution:

Trivially, we can make change for value 0 by taking 0 coins.

Claim: We can make change for a value v>0 if and only if there is a denomination $c_i \le v$ such that we can make change for v- c_i.

The proof of the claim is straightforward: If we can make change for v- c_i then we can add a coin of c_i to get v. Conversely, if there is a sequence of coins that add to v, and the last coin has value c_i, then the other coins add up to v- c_i.

The claim leads to a dynamic programming algorithm for testing whether we can make change for v=0,1,2,…,V. If we let P[v] be the predicate which has value 1 (true) if we can make change for v and value 0 (false) otherwise, then P[0]=1, and P[v] for v>0 satisfies the recurrence $P[v] = \vee\{P[v - c_i] \mid c_i \le v, i = 1,...,n\}$. In order to reconstruct a solution (if we can make change for V), we record for every v for which P[v]=1 in another array C[v] a denomination c_i that satisfies the claim (we could have used one array for both purposes instead of two arrays P and C, but we keep them separate here for clarity). The DP algorithm is as follows.

```
P[0]=1
for v=1 to V do
    {  P[v]=0
       for i=1 to n do
           if (c_i ≤v and P[v-c_i]=1) then { P[v]=1;  C[v]=c_i ; break }
    }
if P[V]=0 then return "NO"
else
  {  S = empty list
     v = V
     while (v>0) do
         { add C[v] to S;  v = v – C[v] }
     return S
   }
```

Correctness follows from the claim.

Time: The first loop takes time O(nV). The second loop that constructs the solution (if there is one) takes time O(V). Thus, the total time is O(nV).

Another way to solve this problem is to reduce it to a graph reachability problem in a DAG. We can think of the process of making up an amount v using coins of the given denominations as follows: Start with 0 money. At any time, if we have an amount u, we can add for any i a coin c_i to form amount $u+c_i$. The question is whether we can get from the initial state of 0 value to the state of V value by performing a sequence of these operations.

Construct a graph G which has nodes 0, 1, 2,…., V. For every node u and each i=1,…,n, we include a directed edge $(u,u+c_i)$ if $u+c_i$ is a node, i.e., if $u+c_i \leq V$. Thus, the graph has $V+1$ nodes and O(nV) edges.

Performing a sequence of coin additions in the above process starting from 0 money corresponds to following a path in the graph G starting from node 0. Hence, we can make change for value V if and only if G has a path from node 0 to node V. Thus, we can test whether we can make change for V by performing a BFS or DFS from node 0; if node V is reached, then a path from 0 to V gives a sequence of coins which sum to V. This algorithm takes also time O(nV).