CS800, Practice Final – Solutions

Note: This file contains more than five problems – it is a collection of several final-exam-like problems.

Name: ____________________________

• This exam is worth 40 points. It consists of five problems, each worth 10 points. The sum of the four highest scored problems defines the final grade.

• If you want to ask a question, write it on the provided sheet of paper and raise your hand. I will collect it, write down the answer and return it back to you.

• If you need more scrap paper, raise your hand. Use only the scrap paper provided.

• Please turn off your cell-phones and other electronic devices.

• When time is called, stop writing immediately and remain seated until all exams have been collected.

• Hand in also your scrap paper and cheat sheet. Stop by my office next quarter to get your cheat sheet back.
Problem 1

Given is a tree \( T = (V, E) \). Give a linear-time algorithm which determines whether \( T \) has a perfect matching.

Recall: A matching is a collection of edges that do not share vertices. A matching is perfect if every vertex is covered by an edge from the matching.

a) Give the pseudocode of your algorithm. You may use any algorithms from class as subroutines.

1. For every vertex \( v \), let \( \text{seen}[v] = \text{false} \) and \( \text{matched}[v] = \text{false} \).
2. Run \( \text{ModifiedDFS}(s) \) for some vertex \( s \).
3. If \( \text{matched}[v] = \text{false} \) for some vertex \( v \), then output “NO perfect matching”.
4. Else, output “Perfect matching exists”.

\[ \text{ModifiedDFS}(v) \]
1. Let \( \text{seen}[v] = \text{true} \).
2. For every neighbor \( u \) of \( v \):
3. If \( \text{seen}[u] == \text{false} \), then run \( \text{ModifiedDFS}(u) \).
4. Else, let parent = \( u \).
5. If \( \text{matched}[v] == \text{false} \):
6. If \( \text{matched}[\text{parent}] == \text{true} \):
7. Output “NO perfect matching” and exit the computation.
8. Let \( \text{matched}[v] = \text{true} \) and \( \text{matched}[\text{parent}] = \text{true} \).

b) Briefly explain your algorithm.

Idea: Keep matching vertices that have only one neighbor that has not been matched yet. Hence, we start with vertices with only one neighbor (these vertices are known as the leaves), then we match vertices that have all but one neighbors matched to leaves, etc.

This can be done while performing DFS – if a vertex is not matched to any of its children, we need to match it to its parent.

Note: there are other possible algorithms for this problem.

c) Argue your algorithm’s running time.

This algorithm modifies DFS, where the modification adds only a constant number of steps per recursive call. Thus, the running time is the same as DFS, i.e. \( O(m + n) = O(n) \) where \( n \) is the number of vertices of \( T \) and \( m = n - 1 \) is the number of edges of \( T \).
Problem 2

Given is a set of \( n \) four-letter words \( w_1, w_2, \ldots, w_n \) (all spelled in lower-case letters). Given \( i, j \), we need to change \( w_i \) to \( w_j \) by changing one letter at a time or permuting the letters in the word and along the way we have to use only words from \( \{ w_1, w_2, \ldots, w_n \} \).

For example, for words “ahoy”, “algo”, “foal”, “goal”, “leaf”, “leap”, “loaf”, “luck”, “pale”, “plea”, we can get from “algo” to “leaf” as follows: “algo”, “goal”, “foal”, “loaf”, “leaf”. We cannot get from “good” to “luck”.

Give an \( O(n) \) algorithm that outputs POSSIBLE if it is possible to get from \( w_i \) to \( w_j \), otherwise it outputs impossible.

a) Give the pseudocode of your algorithm. You may use any algorithms from class as subroutines.

1. Create a graph with vertices \( \{1, 2, \ldots, n\} \).
2. For \( x = 1 \) to 4:
3. Remove the \( x \)-th letter from every \( w_k \), getting \( w'_k \), for \( k \in \{1, 2, \ldots, n\} \).
4. Sort \( w'_1, w'_2, \ldots, w'_n \) alphabetically using RADIXSORT (keep track of the original index of the words, i.e., for the \( k \)-th word in the sorted order, \( a[k] \) is the original index – the \( k \)-th word in the sorted order is \( w'_{a[k]} \)).
5. Let \( k = 1 \).
6. While \( k \leq n \):
7. Find the largest \( \ell \geq k \) such that \( w'_{a[k]} = w'_{a[\ell]} \).
8. Add an edge between every pair of vertices from \( \{a[k], a[k+1], \ldots, a[\ell]\} \).
9. Let \( k = \ell + 1 \).
10. Reorder the letters in every \( w_k \) alphabetically, getting \( w''_k \).
11. Sort \( w''_1, w''_2, \ldots, w''_n \) alphabetically using RADIXSORT (keep track of the original index of words via \( b[k] \)).
12. Let \( k = 1 \).
13. While \( k \leq n \):
14. Find the largest \( \ell \geq k \) such that \( w''_{b[k]} = w''_{b[\ell]} \).
15. Add an edge between every pair of vertices from \( \{b[k], b[k+1], \ldots, b[\ell]\} \).
16. Let \( k = \ell + 1 \).
17. Run BFS from vertex \( i \).
18. If vertex \( j \) has been reached (seen[j] is true), then output “POSSIBLE”.
19. Else, output “IMPOSSIBLE”.

3
b) Briefly explain your algorithm.

We create a graph with vertices corresponding to the words and edges corresponding to the described operations on words. Then, the problem asks whether there exists a path from the vertex corresponding to \( w_i \) to the vertex corresponding to \( w_j \). The tricky thing is to create the graph in time \( O(n) \). If two words differ in one letter, we include the corresponding edge on line a8 (for \( x \) equal to the position of the letter that differs). Since after removing the letter the two words are equal, we will find them after sorting on line a4. Similarly we add edges between two words that use the same letters in different permutations, see line a15.

c) Argue your algorithm’s running time.

The RADIXSORTs take \( O(4n) \) time each since the length of the words is 4 and they use only up to 26 different letters. For five RADIXSORTs, the running time is \( 5O(4n) = O(n) \). Lines a8 and a15 take as much time as is the number of edges in the final graph. Notice that every vertex can be connected to at most \( 4 \times 25 \) other vertices by swapping a single letter of the word, and to at most \( 4! \) vertices by permuting the letters. Hence, every vertex has at most 124 neighbors, thus \( m \leq 124n/2 \). Thus, \( m = O(n) \) and lines a8 and a15 take \( O(n) \) time each (when counted cumulatively across the entire execution of the algorithm). BFS then takes time \( O(n + m) = O(n) \) for our graph. Overall, the running time is \( O(n) \).
Problem 3

a) Given is the following weighted graph $G$. Trace the Dijkstra’s algorithm to compute the shortest distances from vertex $s$ to every other vertex. For every vertex mark the shortest path leading to it.

b) Find the minimum spanning tree of $G$ (mark it on the graph below).

c) Given is the following simple algorithm:

1. Let $T$ be the minimum spanning tree of $G$.
2. For every vertex $v \in V$ do
3. Compute the length of the path from $s$ to $v$ along the edges of $T$.
4. Output the computed lengths as the shortest distances from $s$ to all other vertices.

Does the algorithm work? Argue your answer.

No, the algorithm does not work. E.g., for the graph on the right, the min spanning tree contains the two edges of weight 2 but the shortest distance from vertex $s = A$ to vertex $C$ is 3, not $2 + 2$. 
Problem 4

Given is a computer network with $n$ computers. Given are also $m$ direct connections specifying pairs of connected computers and for each connection we know the bandwidth (i.e. how many gigabytes per second can be sent through the connection). Two of the computers, $s$ and $t$, are very important and a lot of data is being sent from $s$ to $t$. We want to invest and increase the bandwidth on one of the connections so that we can send more gigabytes per second from $s$ to $t$.

Design an $O(nm(n + m))$ algorithm that finds one connection such that increasing the bandwidth on the connection would increase the number of gigabytes per second that can be sent from $s$ to $t$.

a) Give the pseudocode of your algorithm. You may use any algorithms from class as subroutines.

1. Create a flow network with $n$ vertices corresponding to the computers, edges corresponding to the connections between computers, and capacities equal to the corresponding bandwidth.
2. Using the algorithm from class, compute a minimum $(s, t)$-cut $S$ in the flow network. Let $f$ be the corresponding maximum flow.
3. For every cut edge $(u, v)$, i.e., $u \in S$ and $v \notin S$:
   4. Increase the capacity on $(u, v)$ by one, i.e., let $c(u, v) = c(u, v) + 1$.
   5. Construct a new residual graph $G'$ for flow $f$ and the new capacities.
   6. If there is an $s$-$t$ path in $G'$:
      7. Output “Increase the bandwidth on connection $(u, v)$” and exit the computation.
   8. Else:
      9. Change the capacity on $(u, v)$ back to its original value, i.e., let $c(u, v) = c(u, v) - 1$.
10. Output “No such connection exists.”

b) Briefly explain your algorithm.

Every minimum $(s, t)$-cut creates a “bottleneck” for the computation. Thus, we need to increase the bandwidth of an edge on a minimum $(s, t)$-cut. We find a minimum $(s, t)$-cut and try increasing the bandwidth of each edge on this cut. For each such edge, we create the residual graph (notice that the flow $f$ is still a valid flow) and if now there is an $s$-$t$ path, then we can increase the flow. Hence, this is the edge we are looking for. However, if there is no $s$-$t$ path, we set the capacity back to the original value and try another cut edge. If no cut edge works, then there is another minimum $(s, t)$-cut that is causing a “bottleneck”. Thus, we would need to increase the capacity on an edge of this other cut too – in this case there is no solution if we are allowed to increase the bandwidth of only one connection.
c) Argue your algorithm’s running time.

Finding a minimum $(s, t)$-cut $S$ takes time $O(nm(n + m))$ if we use the Edmonds-Karp algorithm to find the flow $f$. We have at most $m$ cut-edges out of $S$ and for each such edge we need to construct the residual graph (time $O(n + m)$) and search for an $s$-$t$ path (again, $O(n+m)$). Hence, overall the running time is $O(nm(n+m) + m(n+m)) = O(nm(n+m))$. 
Problem 5

Recall that a clique of size $\ell$ in an undirected graph is a set of $\ell$ vertices such that every pair of these vertices is connected by an edge.

Also recall the CLIQUE problem where, given an undirected graph $G$ and a number $k$, the answer is YES if and only if $G$ contains a clique of size $k$. This problem is known to be NP-complete.

Consider the following problems:

A. Given a graph $G$ and a number $k$, does there exist a clique of size $k$?
B. Given a graph $G$ and a number $k$, does there exist a clique of size smaller than $k$?
C. Given a graph $G$ and a number $k$, does there exist a clique of size larger than $k$?
D. Given a graph $G$, does there exist a clique of size 5?
E. Given a graph $G$, does there exist a clique of size $n - 5$ (where $n$ is the number of vertices)?
F. Given a graph $G$, does there exist a clique of size $n/2$ (where $n$ is the number of vertices)?
G. Given an acyclic graph $G$ and a number $k$, does there exist a clique of size $k$?
H. Given a graph $G$, is the largest clique of size $k$?
I. Given a graph $G$, is the smallest clique of size $k$?

List all the problems that are in P, all the problems that are in NP, and all the problems that are NP-complete. You do not have to give any reasoning.

$P$: $B, D, E, G, I$

$NP$: $A, B, C, D, E, F, G, I$

$NP$-complete: $A, C, F$

Note: Problem $H$ is NP-hard. (Of course, problems $A$, $C$, and $F$ are also NP-hard.)
Problem 6

The BINPACKING problem is defined as follows. Given are positive integers \( n, k, B \), and a sequence of positive integers \( w_1, w_2, \ldots, w_n \). The number \( n \) represents \( n \) items numbered \( 1, 2, \ldots, n \), where the \( i \)-th object weights \( w_i \) pounds, for every \( i = 1, \ldots, n \). The number \( k \) represents \( k \) identical boxes, each capable of holding \( B \) pounds. The BINPACKING problem decides if it is possible to put every item in a box so that no box holds more than \( B \) pounds. This problem is known to be NP-complete.

(a) Your friend decided to try the following algorithm:

1. Sort the items by decreasing weight.
2. For \( b = 1 \) to \( k \) let \( \text{RemainingCapacity}[b] = B \).
3. For \( i = 1 \) to \( n \) do
4. Find bin \( b \) with the smallest \( \text{RemainingCapacity} \) that is at least \( w_i \).
5. If there exists such \( b \) then
6. Place the \( i \)-th item in the bin \( b \).
7. Set \( \text{RemainingCapacity}[b] = \text{RemainingCapacity}[b] - w_i \).
8. Else Return IMPOSSIBLE.
9. Return POSSIBLE.

Is the algorithm likely to work? Reason your answer.

This is not very likely since the algorithm runs in polynomial time and the problem is NP-complete. If the algorithm works, this would mean \( P=NP \)! (People tried showing that either \( P=NP \) or \( P \neq NP \) for several decades, without success.)

Indeed, it is possible to find a counterexample (try this!).

(b) Reason that BINPACKING is in NP.

Witness: numbers \( a_1, a_2, \ldots, a_n \), for every \( i \): \( a_i \in \{1, 2, \ldots, k\} \) (for every item, the box that the item is packed into)

Verify: for every \( b \in \{1, 2, \ldots, k\} \), sum the weights of all items packed into the \( b \)-th box and check whether this number is smaller than or equal to \( B \);
formally: verify that \( \sum_{i: a_i=b} w_i \leq B \) for every \( b \in \{1, 2, \ldots, k\} \)
This can be implemented in time \( O(nk) \), i.e., a polynomial time.
Problem 7

Given is a sequence of numbers $a_1, a_2, \ldots, a_n$. Cut the sequence into the minimum possible number of contiguous subsequences such that each subsequence starts and ends with the same number.

Example: Sequence 8, 2, 6, 7, 1, 2, 3, 4, 5, 4, 5, 3, 1 can be cut into four contiguous subsequences that start and end with the same number as follows: $8 | 2, 6, 7, 1, 2 | 3, 4, 5, 4, 5, 3 | 1$. It is not possible to cut the sequence into fewer such subsequences.

Give an $O(n^2)$ algorithm that computes the minimum number of such subsequences. Use dynamic programming with the following first part of the heart:

$$S[j] = \begin{cases} 
\text{the minimum number of contiguous subsequences} \\
\text{that start and end with the same number that the sequence} \\
a_1, a_2, \ldots, a_j \text{ can be cut into}
\end{cases}$$

Give the rest of the heart:

- The mathematical expression of $S[j]$:

$$S[j] = \min_{k \in \{1, 2, \ldots, j\}: a_k = a_j} S[k - 1] + 1,$$

where $S[0] = 0$.

In words, we go through every element equal to $a_j$ appearing before $a_j$ (or at the $j$-th position). For every such $a_k$, we find the minimum number of subsequences we need for elements $a_1, a_2, \ldots, a_{k-1}$, plus we add 1 for the subsequence $a_k, a_{k+1}, \ldots, a_j$.

- The return value of the algorithm:

$S[n]$ 

Pseudocode of your algorithm:

1. Let $S[0] = 0$.
2. For $j = 1$ to $n$:
3. Let $S[j] = \infty$.
4. For $k = 1$ to $j$:
5. If $a_k == a_j$ and $S[j] > S[k - 1] + 1$, let $S[j] = S[k - 1] + 1$

10