CS570

Analysis of Algorithms

Summer 2013

Exam III

Name: ________________________

Student ID: ____________________

____On Campus ____DEN

	Maximum	Received
Problem 1	20	
Problem 2	20	
Problem 3	20	
Problem 4	20	
Problem 5	10	
Problem 6	10	
Total	100	

2 hr exam
Close book and notes

If a description to an algorithm is required please limit your description to within 150 words, anything beyond 150 words will not be considered.

1) 20 pts

Mark the following statements as **TRUE** or **FALSE**. No need to provide any justification.

[TRUE/FALSE] True

Assume P !=NP. Let A and B be decision problems. If A is in NP-Complete and $A \leq_P B$, then B is not in P.

[TRUE/FALSE] True

There exists a decision problem X such that for all Y in NP, Y is polynomial time reducible to X.

It is in fact the Cook-Levin theorem that proves that there are problems that are NP-complete. Picking X to be 3-SAT (or any other in problem in NP-complete) ensures that ensures that every Y in NP is reducible to X.

[TRUE/FALSE] True

If P equals NP, then NP equals NP-complete.

True. A problem X is NP-hard iff any problem in NP can be reduced in polynomial time to X. If P equals NP, then we can reduce any problem in NP to any other problem by just solving the original problem.

[TRUE/FALSE] False

The running time of a dynamic programming algorithm is always theta(P) where P is the number of sub-problems.

Solution: False. The running time of a dynamic program is the number of subproblems times the time per subproblem. This would only be true if the time per subproblem is O(1).

[TRUE/FALSE] True

A spanning tree of a given undirected, connected graph G=(V,E) can be found in O(| E|) time. You can just walk on the graph.

[TRUE/FALSE] True

To find the minimum element in a max heap of n elements, it takes O(n) time

[TRUE/FALSE] True

Kruskal's algorithm for finding the MST works with positive and negative edge weights.

[TRUE/FALSE] False

If a problem is not in P, then it must be in NP.

[TRUE/FALSE] False

If an NP-complete problem can be solved in linear time, then all NP-complete problems can be solved in linear time.

[TRUE/FALSE] True

Linear programming problems can be solved in polynomial time.

2) 20 pts
Consider the following heuristic to compute a vertex cover of a connected undirected graph G. Pick an arbitrary vertex as the root and perform depth first search. Output the set of non-leaf vertices in the resulting depth first search tree.

(i) Show that the output is a vertex cover for G.
(ii) How good an approximation to a minimum vertex cover does this heuristic assure? That is, upper bound on the ratio of the number of vertices in the output to the number of vertices in a minimum vertex cover of G.

2) Let G=(V,E) denote the graph, T=(V,E') the resulting depth first search tree, L the set of leaf vertices in the depth first search tree and N (=V - L) the set of non leaf vertices.

(i) Assume there is an edge e=(u,v) in E that is not covered by N. This implies that both u and v are in L. Without loss of generality, assume that DFS explored u first. At this stage since e was available to DFS to leave u, the DFS would have left u to explore a new vertex thereby making u a non leaf. Hence our assumption is incorrect and N does indeed cover every edge in E.

(ii) If a vertex cover of G contains a vertex u in L, then u can be replaced with its parent in the DFS tree while still covering every edge in E without increasing the number of vertices. Hence there exists a min vertex cover (call A) of G that does not contain a vertex in L. In particular, A is also a vertex cover of the DFS tree T.

We next create a matching M as follows. Recall that a matching is a set of edges such that no two distinct edges in the set share a vertex.

For every vertex u in N, pick one edge that connects u to one of its descendents and call it e_u. We call the set of odd level non leaf vertices in the DFS tree ODD and the set of even level non leaf vertices as EVEN. If the set ODD is bigger or equal to the set EVEN, set BIG:=ODD. Else set BIG:=EVEN.

Since the total number of non leaf vertices in the tree T is |N|, BIG has size at least |N|/2. Now the edge set M = {u_e | u is in BIG} is a matching of size at least |N|/2.

Since M is a matching, to cover every edge in the matching the optimal vertex cover A has to contain at least |M| vertices. (This is because in a matching no two distinct edges share a vertex). Thus A has to contain at least |N|/2 vertices while our solution has |N| vertices. Hence our solution is at worst a 2-approximation.

It can be shown that our solution can be indeed twice as bad by considering E={(a,b), (b,c)} with DFS rooted at a.

3) 20 pts
There is a precious diamond that is on display in a museum at m disjoint time intervals. There are n security guards who can be deployed to protect the precious diamond. Each guard has a list of intervals for which he/she is available to be deployed. Each guard can be deployed to at most A time slots and has to be deployed to at least B time slots. Design an algorithm that decides if there is a deployment of guards to intervals such that each interval has either exactly one or exactly two guards deployed.

3.) We create a circulation network as follows. For the ith guard introduce a vertex g_i and for the jth time interval introduce a vertex t_j. If the ith guard is available for the jth interval, then introduce an edge from g_i to t_j of capacity 1. Add a source s and a sink t. To every guard vertex add an edge from s of capacity A and lower bound B. From every interval vertex add an edge to t of capacity 2 and lower bound 1. Add an edge from s to t of infinite capacity. We claim that there exists a valid deployment if and only if the above network has a valid circulation. The proof of the claim is similar to the survey design problem in the text. The algorithm proceeds by determining if the network has a circulation (by reducing it to a flow problem and then applying Ford-Fulkerson) and answers "yes" if and only if there is a circulation.

4) 20 pts
Your input is a string S which is a sequence of n characters from the English alphabet. The string S is believed to be a corrupted version of a text where the spacing between the words have been erased. You have access to a program Dictionary(), which takes a string w as input and returns ``yes" if w is a valid word and ``no" otherwise. Design an algorithm that decides if S can be partitioned (by inserting spaces) into a sequence of valid words. The running time should be polynomial in n assuming that each call to Dictionary() takes polynomial time.

For example, if S = ``sortingiseasy", then your algorithm should output ``yes", since ``sorting”,``is",``easy" is a sequence of valid words.

Let the length of your input string of size N.

Let b(n) be a boolean: true if the document can be split into words starting from position n in the string.

b(N) is true (since the empty string can be split into 0 words). Given b(N), b(N - 1), ... b(N - k), you can construct b(N - k - 1) by considering all words that start at character N - k - 1. If there's any such word, w, with b(N - k - 1 + len(w)) set, then set b(N - k - 1) to true. If there's no such word, then set b(N - k - 1) to false.

Eventually, you compute b(0) which tells you if the entire document can be split into words.

In pseudo-code:

```
def try_to_split(string):
  N = len(string)
  b = [False] * (N + 1)
  b[N] = True
  for i in range(N - 1, -1, -1):
    for word starting at position i:
      if b[i + len(word)]:
        b[i] = True
        break
  return b
```

There's some tricks you can do to get 'word starting at position i' efficient, but you're asked for an O(N^2) algorithm, so you can just look up every string starting at i in the dictionary.

To generate the words, you can either modify the above algorithm to store the good words, or just generate it like this:

```
def generate_words(doc, b, idx=0):
  length = 1
  while true:
    assert b(idx)
    if idx == len(string): return
    word = doc[idx: idx + length]
    if word in dictionary and b(idx + length):
```

```
output(word)
idx += length
length = 1
```

Here b is the boolean array generated from the first part of the algorithm.

5) 10 pts
Show that the following problem is NP-complete:
For an undirected graph G=(V,E), does G have a spanning tree with at most 2 leaf vertices?

5.) Call the decision problem in question ST2L.

Given the graph G (instance) and a set of edges that forms a spanning tree (certificate), we can verify in polynomial time if the set of edges indeed forms a spanning tree and that spanning tree has at most two leaves. Thus the ST2L problem is indeed in NP.

All that is left is to prove that ST2L is NP-Hard. Recall that a leaf vertex in a tree is a vertex of degree 1. Thus a spanning tree has to have at least 2 leaves. A spanning tree with exactly 2 leaves is in one to one correspondence with a Hamiltonian path. Hence our problem is merely a restatement of the Hamiltonian Path problem. Hence you can use the reduction in the last HW to reduce Hamiltonian cycle to Hamiltonian Path (and then Hamiltonian Path to ST2L.)

(Note: If you assumed that the spanning tree is rooted and that the root is not a leaf even if it has degree one, then we need to slightly modify the argument that reduces Hamiltonian path to ST2L. Given a graph G as an instance of Ham-Path, if G has exactly 2 vertices and one edge output "yes", Else call the blackbox that solves ST2L with input G and return the result.)

6) 10 pts

The following are a few of the design strategies we learned in class to solve problems.

1. Dynamic programming.
2. Greedy strategy.
3. Divide-and-conquer.

For each of the following problems, mention which of the above design strategies (or combinations of strategies) we used in class to solve these problems:

1. Determining if a graph has a negative cycle
   Dynamic programming

2. Minimum spanning tree
   Greedy strategy

3. Closest pair of points on a plane
   Divide and conquer

4. Memory efficient sequence alignment
   Dynamic programming + divide and conquer

5. Stable matching
   Greedy Strategy