Christopher Moroney

Arias

CSC 3430

Due March 11, 2020

Dynamic Programming Homework

Exercise 6.1.a)

An example of a graph that will show that the “heaviest-first” greedy algorithm does not return the maximum total weight of an independent set is this:



According to the algorithm, the first node that we would add to S, the returning independent set, is node weight 20. We then delete 20 as well as both nodes on either side of 20 from graph G, which are nodes weighted 14 and 14. The nodes that are remaining are (left) node weight 1 and (right) node weight 1. In this case, it doesn’t really matter which node 1 we pick, so we just add (left) node 1 to S, and we delete both nodes on the graph G. The independent set consists of {20, 1}, but this isn’t the maximum weight. The two independent sets that have more weight are {20, 1, 1} and {14, 14}, so this algorithm doesn’t always find the maximum weight independent set.

Exercise 6.1.b)

An example of a graph that will show that this algorithm does not return the maximum total weight of an independent set is this:



The algorithm will compare {9, 1, 2} and {3, 4, 3}. The algorithm will pick {9, 1, 2} because the weight of this set is 12, whereas the weight of the other set is 10. However, another independent set that is larger than this is v1, v4, and v6, which is {9, 4, 3} The total weight of this is 16, and so this algorithm does not always pick the maximum weighted independent set.

Exercise 6.1.c)

S is an empty set

G is a graph of n-node path

set findMaxIndependentSet(graph G, set S){

if number of nodes in G == 0{

return S;

break;

} else if number of nodes in G <= 2 {

put larger node in S;

return S;

break;

} else {

find largest sum of weights of two nodes that are not

neighbors;

put the two nodes in S;

remove the two nodes from G;

return findMaxIndependentSet(G, S);

}

}

Longest-Common-Subsequence)

Dynamic programming can solve the LCS problem easily, including with this DNA styled question. What we can do is we can take both strands of DNA, and compare them to one another, letter by letter. If we start with both strands complete, dynamic programming would take these strands, compare them either by their last letter or first letter, and then break the letters down subsequently. Suppose we set up a counter that represents the number of letters that match both sets, starting at 0. If there is a letter that matches the letters in their position of their sets, then we simply remove the letter from both sets, and add 1 to the counter. If the letters differ, then we split the problem into two subproblems, one that removes the letter from that position for the first strand, and one for the other. Essentially, there will be a tree of subproblems as more and more letters are matching or differ. The dynamic programming element comes in (compared to just recursion and thus increasing run time) by when we know that a previous subproblem containing all of the letters is a match, then we can automatically add 1 instead of just dividing the problem again while in the same recursion loop. We just simply tag one, and then move on. This is how the LCS problem can be solved easily with Dynamic Programming.