Buffalo Hird
CS124
Assignment 6

Problem 1: We determine the largest $w(S)$ achievable over seperated sets by using a dynamic programming algorithm such that we solve this problem for subtrees of a tree and use these to determine the answer for the whole tree. For easier computing we have our function return a tuple of $w(S)$ for the heaviest separated set including our root node and the heaviest separated set not including our root node. We can then in our final answer return the max of these two values. We then have a simple base case for a tree of size 1, where we return $w(s)$ for s is our one node and 0 as there are no other separated sets.

We otherwise return either (0) our root and the largest $w(S)$ for all its grandchildren, as it is connected to its children or (1) the largest $w(S)$ for any of its children. In this way we will recursively scan down the tree building up bigger separated sets. We therefore have final recurrence:

$$f(s) = \left\{ \begin{array}{cc} [w(s), 0] & |s| = 1 \\ [w(s) + \sum_{c \in children(s)} f(c)[1] & , max_{c \in children(s)} max(f(c)[0], f(c)[1]) \end{array} \right\}$$

Runtime Analysis: Our function has only one argument s which we can store for all n nodes in our tree such that we use $\theta(n)$ space where n is the number of nodes in our tree. We note that we will call $f()$ for all n nodes which will view each edge only once such that we have $\theta(n)$.

Proof of Correctness:

We consider our two cases from which return values: (0) our set containing our root and (1) our set not containing our root.

(0) Our separated set includes our root s.

We note that it is illogical for any of s' children to be contained in the separated set as they are connected to s but any of s' grandchildren can. We can then compute $w(s)$+each separated set of s' grandchildren as these are not connected to each other (or it wouldn't be acyclic) and they all must contain elements not connected to s. We can guarantee this is the heaviest separated set as we cannot produce this set in any other strategy as we must include s if we know it is in the set and we only exclude its children in our non-recursive element of the function as we know they are not separated from s.

(1) Our separated set does not include our root s.

As noted, each of s' children could be in the heaviest separated set as the graph is acyclic such that these nodes are not connected to each other. We can therefore compute $f()$ for each of these children and sum these values to produce the magnitude of the heaviest separated set. We note the correctness

of this as we are doing this heaviest separated set problem for all children such that it is the same logic but summed recursively. We find the heaviest separated set for each of these subtrees by returning the max of the heaviest separated set (in the subtree) containing the sub-root and the heaviest separated set not including the sub-root such that we get correct logic building up from our base case.

Problem 2: We solve the problem of finding the shortest well-nested string that contains our input string s as a subsequence. We note that to do this, we will need to match any left item (i.e. $<.$ ( ) with a right item (i.e. $>$)). We then design a dp algorithm to dynamically determine the shortest string which satisfies this property:

We define a function $f(i, j)$ which sould return the shortest well-nested string that contains our input string s as a subsequence. We then note that we return $f(0, n-1)$ for a string of length n. We note that our base case is when $i = j$. In this case we have a string of length 1 and since we must follow this closure property of pairing left and right elements, the shortest string must have length 2. We also note that we have an invalid input if $i > j$.

We know that a concatenation of two well-nested strings mn is also well-nested such that we can subdivide these into two separate subproblems when determining $f()$ and combine these two solutions to determine the min total well-nested string. We also note that if mn is not well-nested, that we can add an additional 2 characters to close both m and n.

We therefore have:

$$f(i,j) = \left\{ \begin{array}{cc} 0 & i > j \\ 2 & i = j \vee j = i+1(\text{ ij } \mathit{well-nested}) \\ min_{i<m<j}[2 + f(i+1, m) + f(m+1, j-1)] & ij \text{ well-nested} \\ min_{i<m<j}[f(i, m) + f(m+1, j)] & ij \text{ otherwise} \end{array} \right\}$$

Runtime Analysis:

We run this algorithm with i and j where intial values are $f(0, n-1)$ and continuing recursively. As this algorithm can take on any value $[0, n]$ for i and j we will have a space complexity of $\theta(n^2)$. We note that since we store each of these n^2 values we only compute each once each in $O(n)$ time as checking if they are well-nested is $O(n)$ we then have our runtime is $\theta(n^3)$.

Proof of Correctness:

We inductively show that this recursive function returns the correct value. We can define any input for $f(i, j)$ as input string mn. We therefore have two cases: (1) mn is well-nested (2) mn is not well nested.

1.ij is well-nested

Given that we can recursively determine the shortest well-nested string for the subsequences in each of our intervals contained in $[i, j]$ then we can sum

these shortest values as any well-nested string can, as defined in the problem, be concatenated with other well-nested strings (our contain them entirely). We note that we have substrings $(2+f(i+1,m)+f(m+1,j)$. We note that eventually these will hit our base case, otherwise we will continue to sum shortest values in subintervals.

2. ij is not well-nested

Given that we can recursively determine the shortest well-nested string for the subsequences in each of our intervals contained in $[i,j]$ then we can sum these shortest values as any well-nested string can, as defined in the problem, we can use the same approach as case (1) but modified slightly. We modify such that even though ij is not well-nested, we calculate the shortest subsequence that is well-nested in $f(i,m)+f(m+1,j)$ as we note in the problem definition that well-nested strings can be concatenated together and it could be the case that this well-nested substring could form the larger string which is the solution to the whole of the problem.

Problem 3:

We define a dynamic programming solution to find if y is a mixing of s and t, building up from a base case to determine if it is possible for our current string to be a mixing of s and t. We define our function as $f(a,b,c)$ where a= our current position in s, b= our current position in s and c= our current position in t. We can then compute $f(0,0,0)$ which should return 1 if y is a mixing of s and t checking the whole of these 3 strings.

We note we have a base case when $c=n$ as we have reached the end of the string so we return 1. Intuitively, our function iterates through our main strings, proceeding our pointer position for a and b whenever we see a string that matches the next element in s or t. The trick here is that we must restart our strings s and t when we reach the end of them as there can be multiple repititions of these strings in y. We therefore iterate through each of s and t as we iterate through y when we find the next string in either, if neither match we return 0. We therefore have recurrence:

$$f(a,b,c)=\left\{\begin{array}{cc} 1 & c=n \\ 0 & y[c]\neq s[a\%k]\wedge y[c]\neq t[b\%m] \\ f(a+1,b,c+1)\vee f(a,b+1,c+1) & y[c]=s[a\%k]\wedge y[c]=t[b\%m] \\ f(a+1,b,c+1) & y[c]=s[a\%k]\wedge y[c]\neq t[b\%m] \\ f(a,b+1,c+1) & y[c]\neq s[a\%k]\wedge y[c]=t[b\%m] \end{array}\right\}$$

Runtime Analysis:

We note that our recursive function $f(a,b,c)$ has 3 pointers for which we much keep track of. Therefore in memoizing these values we use at most $\theta(kmn)$ as each of these strings s,t,y have lengths k,m,n respectively. Since we save these kmn calculations for lookup and each function call includes at most 2 lookups, we have runtime $\theta(kmn)$.

Proof of Correctness:

We inductively show that this recursive function returns the correct value. We note from our non-base-cases that we have 3 real cases to consider. (1) when the next item in y is part of a subsequence for s and t. (2) when it is for just s and (3) when it is for just t.

(1) the next item could be next in a subsequence of s and t

We note that in this case we will call $f(a+1, b, c+1)$ and $f(a, b+1, c+1)$ and return 1 if either of these return 1. We note that we know that either y starting at $c+1$ is a proper mixing of s starting at $a+1$ and t starting at b *or* a proper mixing of s starting at a and t starting at $b+1$. We note that we only care if one of these is the case as we only need one valid pair of powerings to create a mixing such that if either is valid recursively we have a correct answer. We finally note that $y[c] = s[a\%k]$ and $y[c]] = t[a\%m]$ s.t. that we must have a valid mixing for a, b, c.

(2) the item could only next in a subsequence of s

We note that in this case we will call $f(a+1, b, c+1)$ such that if this returns 1 then y start at $c+1$ is a proper mixing of s starting at $a+1$ and t starting at b. We note that since $y[c] = s[a\%k]$ then we have a valid mixing for a, b, c

(3) the item could only next in a subsequence of t

We note that in this case we will call $f(a, b+1, c+1)$ such that if this returns 1 then y start at $c+1$ is a proper mixing of s starting at a and t starting at $b+1$. We note that since $y[c] = t[b\%k]$ then we have a valid mixing for a, b, c

Problem 4: We define a dynamic programming solution to determine a packing of nodes onto disk such that we minimize the sum of all cost of queries over the previous week. We define a function for a tree T $f(v, i, r)$ that returns the cost of said tree. We define $v =$ the root of this tree, $i=$ the first i children of v, $r=$ the remaining blocks in our current block. We then can cleverly calculate the cost of including / not including i building up dynamically for increasing tree size such that we determine which are best minimized by including or not including in our current block. We run this algorithm as $f(1, deg[1], B-1)$ to determine the result for our total tree T. We note that we have 3 base cases: we can either (1) hit an empty item, (2) hit an item with no children, (3) run out of space in our block such that we must use a new block. In case (1) we can just return 0, in case (2) we are just now returning the query value of v $T[v]$. Given we have no more space in the current block, we find that we regardless have to allocate a new block such that we have the current item in the current block, and then we run this recurrence on the next child of v and sum all the query values of the children of v as we will have to return to this block B for each of these.

We note that for our choice of whether or not we will place a child in the current block to minimize queries over the previous week, we either implement

base case (3), or we instead iterate through all possible values up to the remaining space in our block given we allocate any number up to the remaining available blocks.

Combining all these posibilities we get:

$$f(v,i,r) = \left\{ \begin{array}{cc} 0 & i = 0 \\ T[v] & deg[v] = 0 \\ f(v,i-1,0) + f(child_{v,i}, deg[child_{v,i}], B-1) + \sum_{\forall k \in child_v} T[k] & r = 0 \\ min \left(\begin{array}{c} f(u,i-1,r) = f(child_{v,i}, deg[child_{v,i}], B-1) + \sum_{\forall k \in child_v} T[k] \\ min_{0 \leq s \leq r-1} f(v,i-1,r-s) + f(child_{v,i}, deg[child_{v,i}], (r-s)-1) \end{array} \right) & otherwise \end{array} \right\}$$

Runtime Analysis:

We have three arguments to our function where v and r can collectively take $\theta(n)$ space as these are the number of edges in the tree. We then note that for each r for each v, we can have any value up to$B$ such that we have $\theta(Bn)$ total space. We note that we do Bn calculations equivalent to the space used and that these take at most linear time such that we $\theta(Bn^2)$

Proof of Correctness:

We inductively prove this function, noting as previously stated that there are 2 cases. We determine the minimum of these two cases of tree with i and v and pick the minimum allocation of the two:

(1) i is in the same block as v

Akin to our base case, the cost of having this tree with v's first i children is equivalent to the summing up the values of the tree with $i-1$ children with our space r, the minimum of filling the tree for this given i in a block size B and summing over these allocation values. This is equivalent to our base case above.

(2) i is in a new block after v's block

We take the minumum of any tree with root v that have the first $i-1$ children for all $r < B$ such that we get the minimum cost of these trees and any tree with root i that have space such that we get the minumum cost of these trees. Taking the combined, we should again have a tree v with its first i children.