Problem 1 - Productivity in a messy office

This recurrence is going to look pretty similar to the type of stuff we did in class - I used a two-dimensional type of dynamic programming to get the optimal solution here, but it's still case analysis. The optimal solution for all (h, k) pairs in the table is one of the following statements: if $h \neq k$, which is to say, if the hour h isn't being examined as an hour used to clean, $OPT(h, k)$ is equivalent to $OPT(h-1, k) + w(h, k)$ - the last hour we cleaned is the same so we just move "forward" in time, so to speak. The real interesting bit here is when $h = k$, or in those situations where we are cleaning during hour h. In this case, $OPT(h, k)$ is equivalent to $max(OPT(h-1, k) for\ k \in \{1, 2...h-1\}) + w(h, h)$. So, in these cases, on a "reset" of cleaning the room, the best optimal solution from the previous hour should be selected - essentially partitioning off h and above and choosing the best way to utilize the hours that come before h. Once the algorithm has computed $OPT(h, k)$ for all h and k values, it selects the best among the solutions available for hour n, and then using the memoized previous solutions, collects the set C. This can be algorithmically formalized as

```
Input: $n, w[h][k]$
$C = \{0\}$
$k = 0, h = 1$
Initialize $M[n][n]$ to store $(h, k)$ pairs, and for all instances where $k > h$ mark $M[h][k]$ NULL.
while $h \leq n$ do
    while $k \leq h$ do
        if $k \neq h$ then $M[h][k] = M[h-1][k] + w[h][k]$
        else $M[h][k] = max(M[h-1][k] for\ k \in \{1, ...h-1\}) + w[h][h]$
        end if
        $k = k + 1$
    end while
    $h = h + 1$
end while
The $n$th column now contains the value of the best solution.
$k_{OPT} = max(M[n][k])$
Use the above recursion starting at $M[n][k_{OPT}]$ to step back from $h = n$ through the maximums where necessary (and to just retrieve the value where $h$ is one less where not), adding each element to $C$ until $h = 1$. Return $C$.
```

First, let's just state off the bat that this runs in polynomial time - naively, in $O(n^3)$. I'm sure there's some way we could have the individual columns for each h be max-heaps so as to be able to access the maximum more easily and drop the time to something better, but I like my array. Filling in M would take $O(n^2)$ if there weren't any maximum calculation - h and k effectively circle n events each, and are nested - but n times during that entire calculation, the $O(n)$ operation of getting the maximum for the previous column is calculated, bringing us to $O(n^3)$ish. Finding the maximum in the nth column is then $O(n)$ as well, and stepping back through the solution to actually compile the set is at most n maximum calculations at $O(n^2)$.

The recurrence here does evaluate the best possible solution for all (h, k) pairs - if it doesn't change k, as in if the last time cleaned is staying the same, it forces the (h, k) pair to keep the same k as $(h-1, k)$ - which makes sense, considering that you can't change the last time you've cleaned except by cleaning moving forward into the future. Otherwise, which is where the actual complexity lies, in cleaning during hour h and setting $k = h$, the recurrence cordons off all time between the previous k and h by finding the optimum solution for that particular "interval", which is, of course, $max(M[h-1][k]) for\ k \in \{1, ...h-1\}$, considering that all hours between h and the last time the office was cleaned *must* have the value of the previous hour + working without cleaning - you must work if you're not cleaning. This just essentially means that every time a best solution is chosen, it's chosen based on previous best solutions, so that once n is reached, an answer is clear. A value is always computed from previous optimal solutions, all the way back to $h = 1$, where the best solution is deterministic (assuming $w(h, h)$ is 0, the best option is always to work immediately after cleaning the office at $h = 0$).

And the above algorithm does compute the best solution given the correctness of the above recurrence. The nth column at the end contains one optimal solution (based on columns of previous optimal solutions) which can be quickly determined from a max search - the set that's actually required as the algorithmic output is recoverable via "tracebacks" from the recurrence - from the optimal $h = n$ solution, find the optimal $h = n-1$ solution (moving "left" one box from $M[h][k]$ to $M[h-1][k]$, or finding the maximum previous value once a cleaning value is hit at n from the column to the "left" and continuing from the optimal solution before that point) - so on and so forth until $h = 1$.

Problem 2 - Sketching with outliers

On the advice of solving the problem of each possible interval first, I'll present an algorithm that does this - essentially, for all $i \leq j \leq n$, how many outliers are required to create a single interval between i and j. As it turns out, the solution to this problem is pretty simple - for each $i = 1$ to n, set $k = 0$, and then for each $j = i$ to n, if $y_j \leq y_i + \epsilon$ AND $y_j \geq y_i + \epsilon$, then j fits in i's interval and nothing needs to be done. Otherwise, $k = k + 1$, as j won't fit into i's interval. Regardless, this k value can then be stored in a table indexed by (i, j). Once this $O(n^2)$ algorithm is complete, a table of $(i, j), i \leq j$ pairs will be filled where applicable with the number of outliers required to create a single interval from i to j. This is correct for the same reason problem 1 on pset 2 was correct - in a single interval, no point can be greater than 2ϵ from any other point, and keeping i's height as the 'boundary' for inclusion into the interval holds this true - the maximum height is $y_i + \epsilon$, the minimum is $y_i - \epsilon$, the difference is 2ϵ. Otherwise, the point is marked as an outlier and the algorithm moves on.

Now, onto the actual dynamic programming. Assume our results - the number of outliers required to create an interval from i to j - from the above were kept in some table $S[i][j]$, undefined if $i > j$ but full otherwise. The dynamic programming algorithm for solving the problem of finding the shortest set of intervals using no more than a outliers should look like this:

```
$J_0 = [0]$
while $j = 1$ to $n$ do
    Create table $J_j[min(j-1, a)+1][j]$, indexed by row from 0 to $min(j-1, a)$ and by col from 1 to $j$.
    while $\alpha = 0$ to $min(j-1, a)$ do
        while $i = 1$ to $j$ do
            if $S[i][j] < \alpha$ then
                $J_j[\alpha][i] = NULL$
            else
                $J_j[\alpha][i] = min(J_{i-1}[(\alpha - S[i][j])][k]$ for any $k \leq i-1)$ +1
                NB: If $\alpha - S[i][j]$ is less than 0, use 0. If $\alpha - S[i][j]$ is greater than $i-2$, use $i-2$ (or more
  specifically, the maximum $\alpha$ value for table $J_{i-1}$, which is either $a$ or $i-1-1 = i-2$).
            end if
            $i = i + 1$
        end while
        $\alpha = \alpha + 1$
    end while
    $j = j + 1$
end while
```

The minimum value in $J_n[a][i]$ for any $i \leq j$ is the minimum value for the full data set. Use the above recurrence to trace back through the recurrence, using the newly obtained i and j in each step as the start and finish values for each interval, which is then $\in C$.

This is rather complex, but essentially, the values in $S[i][j]$ represent the outlier 'cost' - the number required - to make a single interval from i to j. The actual dynamic programming above computes $J_j[\alpha][i]$, the number of intervals required to get from i to j with α outliers to spare (and using $NULL$, whether or not that's possible). The outlier 'cost' to make an i to j interval then becomes helpful - the α available for any given combination has the 'cost' for moving from i to j in one step subtracted from it, meaning that there are $\alpha - S[i][j]$ outliers 'left' to use in the interval previous to the current one. Once the minimum value in the final table is found, this recurrence can be used to compile the set of intervals from there, taking the optimum solution and figuring out what i to j intervals it 'spent' its outliers on.

As for proving that this recurrence actually works, I'll give it a vague shot - this recurrence actually calculates the optimal solution, period, for all values of j from 1 to n using at most a outliers. You could stop at the end of any given iteration of the j loop and find the optimal set of intervals. If the basis for this proof of induction is that at $j = 1$, the value calculated $(J_1[0][1])$ is $J_0[0]$, or $0, +1$, (i.e. a single interval is required to go from a single point to itself - clearly true), and we assume that we have all the solutions - including the optimal solution - for $J = m$, that is, all values in $J_m[min(a, m-1)][m]$, for all solutions in J_{m+1} one of the following will be true. Either for some $i \leq m+1, \alpha \leq a$, $S[i][m+1] < \alpha$, in which case that particular interval cannot be included in any solution using α or fewer outliers, or it can be used in a solution, $J_{m+1}[\alpha][i] = min(J_m[(\alpha - S[i][m+1])][k$ for any $k \leq m+1]$. All the values that could be generated here have already been computed in tables $J_{j=1 \text{ to } m}$ - all that's left is to increment by 1 and assign - and any of them could be incorporated into the optimal solution, including that optimal solution from the previous problem where $j = m$. These solutions can increase the number of intervals being used, drawing upon previous optimal solutions given the outlier 'cost' of the current step and the length of the current interval, or it can eschew them entirely and attempt to use the outliers in all one go and create a single interval, using J_0. One way or another - when new values are computed and inserted into J_{m+1}, they are based off of previous optimal values and current required costs, making them themselves optimal values for the given step and given i and α conditions. The optimal solution, of course, is then among them, as the entire solution space is being thoroughly searched. This is then findable with a quick minimum search at J_{m+1} on α - the current solution with the most usable outliers. J_m implies J_{m+1} - the recurrence does its job and finds an optimal solution.

Lastly, let me just specify that this algorithm is in fact running in polynomial time: after the initial $S[n][n]$ computation of $O(n^2)$, the dynamic programming algorithm runs in three loops - creating tables $J_{j=i \ through \ n}[n][n]$, and for each value inserted into a table, there may be a need of a search of n elements - the particular row with $\alpha = \alpha - S[i][j]$ in the previous table has $i = 1$ through j elements in its row, which is at max n elements. At some points, then, this min search is an $O(n)$ operation inside an $O(n^3)$ loop, for a grand total of $O(n^4)$. Whoops. This is pretty terrible optimization, but hey, at least it's polynomial.

Problem 3 - Parties & trees

The algorithm to do this has three steps - a sort of 'inward' motion, an 'outward' motion, and then a quick min comparison. First I want to set some definitions down: define $f(x, y)$ as the function representing the value of a subtree rooted at y and not including x or anything in a path to x that does not include y. Essentially, $f(x, y)$ represents a subtree rooted at y and 'cut off' at x. This function's domain is only on the set of edges in the graph - x and y need to be adjacent. The value of a subtree, $f(x, y).v$, is the numerical value *val* of all 'relevant' - I'll explain in a second - segments, and $f(x, y).s$, the number of nodes $s \in S$ that are inside the subtree. The other definition I need to establish is that of $SUM(x)$, which is the sum of all $f(x, y)$ pairs 'cut off' at x - essentially, the summation of all subtrees that surround a vertex.

Now, let's get on to the algorithm itself. We'll want to start the algorithm by working our way inward from the leaves of the tree, then once we've reached the center, taking the paths back out - in the process, getting the SUMs of all nodes in S, and comparing them along the way.

```
Step through the adjacency list and mark each node with the original cardinality of its adjacency set, and
the same number for variation a second time.
Form a queue. Enqueue all $(x, y)$ pairs where $y$ is a vertex of degree 1 - retrievable by checking the
cardinality of each node - and $x$ is $y$'s sole neighbor.
Create a table $SUM$ of all nodes, with fields $SUM(x).v$ and $SUM(x).s$. Initialize all those fields to 0.
Lastly, create a table $f$ of all edges, in both directions (i.e. $(x, y)$ and $(y, x)$ are separate entries). Since
there are only $n - 1$ edges in a tree, this is just $2n - 2$ operations, or still $O(n)$.
while queue is not empty of $(x, y)$ pairs do
    if $y \in S$ then
        $\gamma = 1$
    else
        $\gamma = 0.$
    end if
    if the true degree of $y = 1$, so if $y$ is a leaf at the start then
        $f(x, y).v = d(x, y) * \gamma$
        $f(x, y).s = \gamma$
        $SUM(y).v = 0$
        $SUM(y).s = \gamma$
        $SUM(x).v = f(x, y).v + SUM(x).v$
        $SUM(x).s = f(x, y).s + SUM(x).s$
    else
        $f(x, y).v = d(x, y) * (SUM(y).s + \gamma) + SUM(y).v$
        $f(x, y).s = SUM(y).s + \gamma$
        $SUM(x).v = f(x, y).v + SUM(x).v$
        $SUM(x).s = f(x, y).s + SUM(x).s$
    end if
    Reduce the variable marked cardinality of $x$ by 1.
    if the variable marked cardinality of $x = 1$ then
        Enqueue all pairs of the form $(w, x)$, but only if $(x, w)$ has not been encountered yet - for example,
if the leaf $c$ was attached to $b$, and $b$ to $a$, $f(b, c)$ currently precludes $f(c, b)$ from being analyzed, but
enqueues $f(a, b)$. NB: If this is not at all possible - i.e, all pairs have been exhausted, the center has been
found and a variable center should be set to the number of that node, from here on marked $\rho$. If $\rho \in S$,
$SUM(\rho).s = SUM(\rho).s + 1$ (as this is not accounted for above).
    end if
    Dequeue $(x, y)$.
end while
```

]This is the first part of the algorithm - essentially, once this is done, the 'inward-facing' values of all the subtrees - from leaf inwards - have been calculated and are ready for use. Additionally, a center has been calculated - a node where all the subtrees that have been created come together and no more are findable from an 'inwards' approach. Now, the other part of the algorithm begins. Fortunately, this is a bit simpler.

```
$\sigma = |S| = SUM(\rho).s$
Enqueue all edges of the form $(x, \rho)$ onto a new queue.
$min.val = \infty, min.num = 0$, unless $\rho \in S$, in which case $min.val = SUM(\rho).v, min.num = \rho$.
while the queue is not empty of $(x, y)$ pairs do
    $f(x, y).v = SUM(y).v - (d(x, y) * (SUM(x).s)) + (d(x, y) * (\sigma - SUM(x).s))$
    The $f(x,y).s$ values need not be computed, as the entire mathematical process has already been computed through usage of $SUM$, and the second recurrence here doesn't actually need more data.
    If $x \in S$ and $f(x,y).v < min.val$, $min.val = f(x,y).v$ and $min.num = x$
    Enqueue all pairs of the form $(w,x)$ that have not already been seen - essentially, the $x$ here, which was originally all nodes adjacent to the center $\rho$, becomes the new "center" to build back outwards from.
    Dequeue $(x,y)$.
end while
Return $min.num$.
```

This bit of the algorithm essentially just works backwards from the defined center, 'trading' the number of times friends would have to cross particular edges and so in a sense finding the optimal set of paths for friends to follow. Having all the $SUMs$ computed already is the key here - the number of friends on either side of a node being available information immediately is a huge boon and allows for each of the edges processed in the first step to have their mirrored edges processed in $O(n)$ time.

The algorithm does run in $O(n)$ time. The initial setup of SUM and f each take $O(n)$, as any time "all edges" are being considered, this being a tree, $|E| = |V| - 1$, which is close enough to n to fudge it and even then, O is an upper bound. The important pieces of the time complexity here are the loops themselves. Notably, the queue in the first section of the algorithm, the moving inwards bit, only touches each edge once - you cannot enqueue (w,x) if you've enqueued (x,w), and as it moves from every leaf inward, it touches each edge of the undirected acyclic connected graph once. It does a constant-time data retrieval based on previous items that were processed in the queue, and then it discards that edge, not to pick it up again. This is thus $O(n)$. The second part of the algorithm has a similar structure - but instead of working from the leaves inward edge by edge, it works from the center outward, with the mirror versions of each of the edges previously seen in the first queue. Again, the constant-time data retrievals here from data that already has been computed gives dynamic programming the edge and allows for $O(n)$ processing a second time. After all the edges have been processed, some $x \in S$ will have been found with the minimum total walking distance for all other $y \neq x \in S$.

As for the correctness of the algorithm, the gritty details of the implementation don't actually matter that much - what matters is that on the 'inward' facing section, values of already-computed sub-subtrees which constitute (most of) the subtree currently in question are calculable in linear time using memoization, and that on the 'outward' facing section, the previously determined values of those subtrees are being stepped through - implicitly creating even larger subtrees than before - and their values are being examined. One of these will include the minimum for the entire set of friends. The former claim - that the inward facing section computes ever-increasing subtrees until it hits a center and does so in linear time - is a trivial one: the algorithm begins at leaves and forbids 'going back' in their direction. That is to say, for the subtrees composed of individual nodes at the end of leaves, the algorithm perfectly computes their values - and from there, expands inward, using the values of leaves (and whether or not they'll actually ever matter re: walking distance) to calculate the subtrees rooted at nodes one edge 'further in' - whose values are entirely composed of the number of friends in each subtree connected to the node in question times the distance of the edge, in addition to the distance traversed in each individual subtree under the node in question as well. The second claim is a little harder to justify - however, my algorithm, instead of continuing with the same process as in the first half, takes the second half into consideration by starting with the center node ρ and essentially making "trades" in one direction or the other - part of the purpose of calculating $f(x,y)$ for all 'outward-facing' (x,y) pairs is that dissolution of the full tree into two connected components on any edge is immediately available - we actually have it sort of already computed in SUM - and that means that should we want to examine a different node as the 'center', all that we have to do is consider: As a matter of difference from ρ, how many friends no longer have to cross this node? How many friends now have to cross this node, and didn't before? The combined value of all these can then be taken as the new 'center', and has its information stored so that $f(\rho,x)$ can be used to calculate $f(x,y)$, and so on. This is a fairly intuitive way of thinking about these things - it retains its linear time, however, only by virtue of having the solutions in the other direction pre-computed.