CS341 Final S2017

1. Assume the length of each word is less than M. We define our subproblems as follows. D(j) is the minimum sum of squares of number of extra spaces for the first j words. Then D(1) = $(M - l_1)^2$. We have the recurrence

D(j) = min { D(i-1) + $(M - j + i - \sum_{k=i}^{j} l_k)^2$ }, for all $0 < i < j$ if $\sum_{k=i}^{j} l_k \leq$ M

In other words, since word j is the last word, it is on the last line. We compute all possible starting points for this line and calculate their minimum sum of squares of number of extra spaces. The minimum sum of squares of extra spaces for all words is D(n) - using all the words. To obtain the actual line break locations, for each D(j) we keep track of which i we used as the line break. Then to obtain the line breaks, we start at D(n), find the last line break, and so on.

Runtime: There are n subproblems. For each subproblem, we iterate at most M times, since M is the maximum number of characters per line. The summation in the recurrence equation can be calculated in O(1) time in each iteration by keeping a running sum as we iterate starting from j. Hence our runtime is O(nM).

Correctness: We prove the correctness of our algorithm on j, the correctness of the minimum sum of squares of number of extra spaces of the first j words (D(j)), using induction. Base case: j=1. Since we only have one word, we only have one line. The square of number of extra spaces is $(M - l_1)^2$, which is what our algorithm calculates. Inductive step: Assume D(k) is correct for all 0 < k < j. Since word j is the last word, it is on the last line. We iterate through all possible starting locations for this last line. The minimum sum of squares of number of extra spaces for the previous lines is D(i-1), where i is the start index of the last line. Hence by the recurrence relation, our algorithm calculates D(j) correctly. Hence our algorithm is correct.

2. We can model this problem as a graph G, where each submission is a vertex and each committee member is a vertex. There is an edge from submission to a member if the member is willing to review that submission. The graph is bipartite, since edges only go between vertices and committees. We model the problem as a max flow problem. Edges from submissions to members have capacity 1. We add a source vertex s and an edge from s to every submission with capacity 3. We add a sink vertex t and an edge from every member to t with capacity 3m/n. An ideal assignment corresponds to a flow of 3m.

Hence our algorithm works as follows. The max flow problem can be solved with the Ford Fulkerson algorithm. If the max flow returned is 3m, we return the submission-member edges in the graph with a flow of 1. Otherwise, no idea assignment exists.

Runtime: It takes O(mn) time to construct the graph, since there are at most nm edges. Ford-Fulkerson takes O(M|E|) time, where M is the maximum value of a flow. There are at most nm edges in the graph, and flow of at most 3m (since there is an edge from s to every submission with capacity 3). Hence our algorithm runs in O(nm^2) time.

Correctness: If our algorithm returns an assignment, it must be valid and ideal. Since edges between submissions and members have capacity 1, a flow of 1 means we assign the member to this paper, and a 0 means the member does not review this paper. Flow can only be 0 or 1 on these edges since the capacity is 1. If we have a max flow of 3m, every paper just be assigned to 3 members, since each

s to submission edge has a capacity of 3, totalling to exactly 3m. Likewise, every member must be assigned 3m/n papers, since each member to t edge has a capacity of 3m/n, totalling to exactly 3m.

3. **A)** We first prove that this problem is in NP. Given an s-t path, we can check if it is a Hamiltonian path by traversing the path once, and checking that every vertex was visited exactly once, and that every edge in the path is in G. This takes polynomial time.

[You can also use DHP but similar to the assignment but then you have to prove NP-completeness of DHP] We prove that this problem is a reduction of the Hamiltonian path (HP) problem. Given an instance of HP with undirected graph G, we create an instance of our problem as follows. We create G' with the same vertices as G. For each edge (u, v) in G, we create two directed edges (u, v) and (v, u) in G'. We add a vertex s, as well as an edge from s to every vertex. We also add a vertex t, and an edge from every vertex (except s) to t.

If there is a solution for this instance of HP, there is a Hamiltonian path in G. Then there must be a simple path in G' using all vertices except s and t. Let x and y be the two endpoints of this path in G'. We add s to the path by adding the edge (s, x). We add t to the path by adding the edge (y, t). Then we have a Hamiltonian s-t path in G'.

If there is a solution for this instance of our problem, there is a Hamiltonian s-t path in G'. If we remove s and t from this path, it is a Hamiltonian path in G, since G has all the same edges and same vertices (except s and t) as G'.

It takes polynomial time to construct G', since it has the same vertices and edges, plus 2 vertices s and t as well as their edges. Since HP is NP-complete, our problem must also be NP-complete.

B) [Assignment 5 question] We first prove that this problem is in NP. We can construct a polynomial-time verification algorithm for this problem that takes a path in G from s to t. To check that the path is simple and has total length of at most L, we iterate through each vertex in the path once, updating our visited array, as well as calculating the sum of edge lengths on the path. We can then check the visited array to see if any vertices were visited more than once, as well as compare the path length to L. This algorithm takes linear time. Hence we have a polynomial-time verification algorithm for this problem, so it is in NP.

We prove that this problem is a reduction of Hamiltonian s-t path (HPST). Given an instance of HPST with graph G and vertices s and t, we construct an instance of our problem as follows. We create G' with the same vertices and edges as G, where each edge has length -1. We use the same vertices s and t, and set $L = -(|V| - 1)$.

If there exists a solution to our instance of HPST, there exists a Hamiltonian s-t path in G. Then there exists a Hamiltonian s-t path in G', since they are the same graph except G' has edge weights. This path has length $-(|V| - 1) = L$, since there are $|V| - 1$ edges and each has length -1. Then we have found a solution to our problem.

If there exists a solution to the instance of our problem, there is a simple s-t path of length at most L. This path must have $|V|$ vertices, since it's length must be at most L and every edge has length -1. Then this must be a Hamiltonian path, since it visits every vertex once. Since G' is the same graph as G except with edge weights, we have found a Hamiltonian s-t path in G.

It takes polynomial time to construct G', since it is a same graph as G except with edge weights. Since HPST is NP-complete, our problem must also be NP-complete.

4. We pick some vertex to be the root. We define our subproblems as follows. D(v) is the weight of the minimum weight vertex cover for the tree rooted at v. If v is a leaf, D(v) = 0. If v is not a leaf, we can either include v in the vertex cover or not include it. Let's call the minimum weight vertex cover for the subtree M.

- If v is in M, M - v is a minimum weight vertex cover for the forest consisting of all v's children. Then the cost of M is $w_v + \sum_{u\ child\ of\ v} D(u)$
- If v is not in M, then all of v's children must be in M, since we must cover each edge from v to its children. Then the cost of M is $\sum_{u\ child\ of\ v} w_u + \sum_{w\ grandchild\ of\ v} D(w)$

We can compute this by traversing every vertex of T once and using memoization. Our result is D(root).

Runtime: There are |V| subproblems. Each subproblem requires (#children + #grandchildren) lookups, which is

$$\sum_{v\ in\ V} \#children + \#grandchildren = \sum_{v\ in\ V} \#parent + \#grandparent \leq \sum_{v\ in\ V} 2 = 2|V|$$

Hence the time complexity is O(|V|).

Correctness: We use induction on the vertices v to prove the correctness of our algorithm. Base case: the algorithm is correct for leaves, since leaves have no children. Inductive step: Assume the algorithm computes D(u) correctly for all vertices u with subtree smaller than v. Then there are only 2 cases as we described above: v is in the vertex cover or v is not in the vertex cover. By the inductive hypothesis and our discussion above, we have that our algorithm must compute D(v) correctly.

5. Let S be the 3SAT formula. Let $S_{i=1}$ be the formula obtained from S by setting variable x_i to true. Then we can simplify the equation as follows: for each clause, if x_i is a literal in the clause, we remove it from the formula; if ! x_i is a literal in the clause, we remove the literal from the clause (if the clause only has ! x_i, we mark it is not satisfiable by any assignment). This takes O(m) time, since we just iterate through each clause once. Similarly, let $S_{i=0}$ be the formula obtained from S by setting variable x_i to false and simplifying (same method, but vice versa).

Our algorithm works as follows.

```
function solution(S) {
    assignment = array of length n
    for i from 1 to n
        if B( S_{i=1})
            S = S_{i=1}
            assignment[i] = true
        else
            S = S_{i=0}
            assignment[i] = false
    return assignment
}
```

Runtime: We iterate through each variable once. In each iteration, we call B twice, and run the simplification twice. Hence our algorithm runs in polynomial time.

Correctness: We first prove that $S_{i=1}$ is satisfiable if and only if S has a satisfying assignment where x_i = true. If we set x_i to true in S, then every clause containing the literal x_i is satisfied. We have that $A\ OR\ false\ =\ A$ for any Boolean expression A, so for all clauses where ! x_i = false, we can just remove

! x_i from that clause, unless the clause only contains ! x_i in which case then the formula is not satisfiable. Then if $S_{i=1}$ has a satisfying assignment, we can augment that satisfying assignment by setting x_i = true to find a satisfying assignment in S. If S has a satisfying assignment where x_i = true, then the assignment restricted to the variables other than x_i will satisfy $S_{i=1}$. This proof is symmetric for $S_{i=0}$. Hence our algorithm is correct.

6. We first prove that this problem is in NP. Given a subset T, it takes polynomial time to iterate all the sets and check that T intersects the set. It takes polynomial time to verify that T has at most k elements.

We prove that this problem is a reduction of the decision version of vertex cover (is there a vertex cover of size at most k). Given an instance of vertex cover G = (V, E), we construct an instance of our problem as follows. We arbitrarily assign each vertex in G a unique number in [1, |V|]. We let n = |V| and m = |E|. For each edge (u, v), we create the set S which contains the numbers assigned to u and v. We keep k the same.

If there is a solution to this instance of vertex cover, we show there is a solution to this instance of our problem as follows. The vertex cover must intersect every edge. Then the numbers corresponding to the vertices in the vertex cover constitute T. T intersects every set S, since each edge in G corresponds to an edge, and if the vertex cover intersects an edge, this means T intersects that set. Hence T is a solution to our problem instance.

If there is a solution to this instance of our problem, we show there is a solution to this instance of vertex cover. T intersects every set. Since each set corresponds to an edge in G, if T intersects a set, it intersects that edge in G. Then the solution to the vertex cover instance is the vertices corresponding to the numbers in T.

It takes polynomial-time to construct the instance of our problem, since we iterate every vertex and edge once. Since vertex cover is NP-complete, our problem is also NP-complete.

7. A) We construct our subproblems as follows. D(m, i, j, k) is the minimum calories that fulfills at least i vitamin A, j vitamin B, and k vitamin C, using the first m items. We have D(0, 0, 0, 0) = 0. We have the recurrence

D(m, i, j, k) = min { D($m-1, i, j, k$), k_m + D($i - a_m, j - b_m, k - c_m$) }

At each iteration, we can either use food item m or not.

Runtime: We have nABC subproblems. Each subproblem checks 2 items. Hence the runtime of our algorithm is O(nABC).

B) We first prove that this problem is in NP. Given a subset S, we can check that the total calories is equal to K in polynomial time. We can check the vitamin requirement in polynomial time.

We prove that this problem is a reduction of decision subset sum. Given an instance of subset sum with numbers $x_1 \ldots x_n$ and target value V, we construct an instance of our problem as follows. We create n food items, where each item i has 1 unit of vitamins A, B, and C, and x_i calories. We let A, B, C = 0, and K = V.

If there is a solution to this instance of subset sum, then we have a subset of numbers that sum to V.

Then the corresponding subset of food items have calories that sum to V=K. They satisfy the vitamin constraints, since A, B, C = 0.

If there is a solution to this instance of our problem, we have that the calories of the food items sum to K. Then this is the subset of integers that sums to V.

It takes polynomial-time to construct the instance of our problem, since we can iterate every item once to construct the instance. Since subset sum is NP-complete, our problem is also NP-complete.