GA HW 06

Piyush Hinduja

December 2023

Question 1

Our aim here is to convert the non-linear objective to a linear one.
Given objective function is:
Minimize $max_{1 \leq i \leq n} |y_i - < u_i, x > |$
Max function and the absolute function make it non-linear.

For the max function, we know the value it will return will be greater than or equal to all other values from 1 to n.
If we denote it by M, we can say that, M $\geq$ $|y_j - < u_j, x > |$ for all j.

And we can get rid of this absolute value using an inequality of absolute values rule as follows,

$$-M \leq (y_j - < u_j, x >) \leq M$$

Hence, our linear program formulation will be,

Variables:
$M \epsilon R^d$
$u_1, u_2, ..., u_n \epsilon R^d$
$y_1, y_2, ..., y_n \epsilon R^d$
$x \epsilon R^d$

Constraints:
For all j, $(y_j - < u_j, x >) \leq M$
For all j, $(y_j - < u_j, x >) \geq -M$

Optimize:
Minimize M

Question 2

a)

For the independent set problem, our original problem is to achieve a largest set of nodes which aren't connected directly.

In the given formulation, the constraint $x_u^2 = x_u$ for all nodes, states that for each node we can have either 0 or 1 associated to it, where 0 indicates the node is not a part of independent set and 1 indicates the node is a part of independent set.

The constraint $x_u + x_v \leq 1$ for all edges ensures that from 2 nodes of an edge, at most one node is taken, that is 2 nodes in the independent set aren't connected. Hence, we can say that any feasible solution to the formulation will yield a feasible solution to the original problem.

Also, given an independent set, for each edge, it will have at most 1 node in the set, hence, our constraint $x_u + x_v \leq 1$ will be satisfied.
So, we can say that any feasible solution to the original problem yields a feasible solution to the formulation too.

b)

Now, using LP relaxation, our constraint $x_u^2 = x_u$ for all nodes becomes $0 \leq x_u \leq 1$.
Hence in order to maximize $\sum_u x_u$, the formulation will assign each node with value 0.5.
For a clique with n nodes,

$$\sum_{1 \leq u \leq n} x_u = \frac{n}{2}$$

Whereas the value of true independent set for a clique will be 1.

c)

We can observe differences between both the values because in the integer or binary LP, each node can take values either 0 or 1 and it has to satisfy other constraints.
Whereas, in case of fractional LP, nodes can take values between 0 and 1 and they need to satisfy the same constraints.

No, we could not use formulation in part a) to solve Maximum Independent Set quickly because Integer LPs are NP hard problems and they take a lot of time. Therefore, we relax them to fractional values so that they can be solved in polynomial time.

Question 3

a)

Decision version of Minimum vertex cover problem could be:
Can the number of vertices in the set cover problem (nodes with cameras) be equal to k in the optimal solution?

b)

Verification Algo:

```
function verifyvertexcover(G, S):
    for all edges in G:
        if both the nodes of an edge are NOT present in S:
            return False
    return True
```

Certificate:
The certificate for the minimum vertex cover problem is simply the subset S of vertices. Given the graph G=(V,E) and a subset S of vertices, the certificate tells that S is a vertex cover for G. The certificate size is O(V), and it can be easily verified using the verification algorithm above.

c)

For this problem, we can perform rounding in the following manner (x_u is the fractional solution and y_u is the binary solution):
If $x_u \epsilon [0, 0.2)$, then $y_u = 0$.
If $x_u \epsilon (0.8, 1]$, then $x_u = 1$.

Observation: We can observe that if $x_u \epsilon [0, 0.2) U (0.8, 1]$, then $1.25 * x_u \epsilon [0, 0.25) U (1, 1.25]$.
For all values of $1.25 * x_u$, $y_u \leq 1.25 * x_u$.
Hence we will also have,

$$\sum_u y_u \leq 1.25 * \sum_u x_u$$

Question 4

Here, we are given that all items have lengths in $s_1, s_2, ..., s_r$ and we have n items in total.
Hence, we can write down all possible configurations of how a box can be filled (with length at most 1) using r lengths and n items.

Then we can maintain a 'dp' table of size n x n, in which we can store 'different combinations of items' and their corresponding 'number of boxes required to store those items'.

Then we can apply a recursive procedure which returns the minimum number of boxes required to pack i items, until all items are covered, as follows:

$$h(A) = 1 + h(A - I_1) = 1 + 1 + h(A - I_1 - I_2) = ... = k + h(0)$$

Here, h(n) represents number of boxes required to pack n elements optimally and k at the last step represents number of boxes required to pack all elements of list A.

Algorithm:

1) Store all the possible configurations in a look up table.
2) Then starting with the first item and go on increasing till last item.
3) At each step, we will have a changing set of items and a constant configuration look up table from which we select the optimal most configuration for those set of items.
4) Finally when we reach last step, we will have the optimal number of boxes to store all n items.

Running time:
To check all possible configurations of 'r' elements which could take at most 'n' size, it will take $O(n^r)$ time.
Whereas in Dynamic Programming procedure we are traversing all configurations for each element, which in worst case becomes $O(n^2 * n^r)$.
Hence, overall running time of algorithm will be $O(n^r)$.

Correctness:
Applying Dynamic Programming ensures that we go through all the possible sub-problems.
And at each step we are picking the optimal most configuration for our items which will give us optimal solution at the end.

Question 5

We know that, $0 < a_i < 1$.
We can multiply all the elements with p such that we covert fractional values to integer values.
Hence our range of elements become, $0 < p * a_i < p$.
i.e., $-np \leq \sum_{i=1}^{n} s_i p a_i \leq np$

Hence, we can say that our required sum will have values between -np and np.

We will make a dp table of size, (2np + 1) x (n + 1), which will have rows numbered from -np to np and columns from n to 0, with all elements as 0 initially.

We will now see all elements from a_1 to a_n one by one and hence the size of array will decrease sequentially from n to 0 (so are our column values).
And while processing each element, taking both possible values (positive and negative) of element into consideration with the previous sum values and set the new sum values (which will be the row number) as 1.

So, while moving forward, our number of possible sum values (or answers) will go on increasing but they will be in range -np to np.

Finally, the entry in dp table with last column (0) and row closest to middle row (0) which has entry set to 1 will be returned.

Algorithm:

```
Initialize dp[2np+1][n+1] with all elements 0.
A = [$a_1, a_2, ....., a_n$]

For i in [n-1, .., 0]: // Iterating over columns by processing one element at a time and reducing size of A
    For j in [-np, .., 0, .., np]: //Iterating over all rows
        if dp[j][i-1] == 1: //Checking for sum values until previous element
            dp[j+A[i]] = 1 //Set s = +1 instance
            dp[j-A[i]] = 1 //Set s = -1 instance

return dp[row value for which entry is 1 and is closest to 0][0]
```

Running time:
Multiplying n elements with a constant takes O(n) time.
Maintaining a dp table of size 2np x n takes O($2n^2p$) time.
Hence, overall algorithm takes O(n^2p) time with is polynomial in n and p.

Correctness:
For each element we are storing sum values of both it's positive and negative instances with the previous sum values, which ensures that we don't miss out any possible sub-problem.
Also, at the end we are returning the sum value closest to 0 which ensures we are minimizing absolute value of our sum.