Assignment 4

Bhanu Verma (903151012)

QUESTION 1: Branch & Bound and Local Search

Part 1: Branch and Bound

Answer 1: To start with let's define the given problem, MINIMUM SET COVER, first. Given a set of elements (number of elements, n), called the universe U, and total number of m sets, S, whose union equals to universe, we need to find the smallest sub-collection of sets, S^, whose union is equal to universe U.

We need to devise a branch and bound algorithm for Minimum Set Cover. To design our branch and bound, we will use an array of size m (number of sets), $Z_1, Z_2, Z_3 \ldots \ldots \ldots Z_m$, to store our partial solutions. For each subset till i, we evaluate if S_i is part of solution (Z_i becomes 1) or not part of solution (Z_i becomes 0). If we haven't looked at a subset yet, then Z_i will contain -1.

a). What is a sub-problem for branch and bound algorithm?

Sol a). If we have a partial solution $Z_1, Z_2, Z_3 \ldots \ldots \ldots Z_m$ then as per our above definition, let S^* be the subsets for the given partial solution i.e. sets that are included in the set cover. S^* can be formally defined as:

$$S^* = \{S_i \; where \; Z_i = 1, i = 1,2 \ldots \ldots m\}$$

For the partial solution, we have U^* as the set of unique elements covered by subsets, S^* in the partial solution. Now, our sub-problem can be defined as smaller set cover problem with a reduced universe $U' = U - U^*$ i.e. universe obtained by removing the elements that have been covered at-least once by the subsets in the partial solution. Sets for our sub-problem will be the sets, S' for which no decision has been made yet i.e. all the sets for which Z_i = -1. We have defined out sub-problem in terms of S' and U'.

b). How do you choose a sub-problem to expand?

Sol b). Intuitively if think about this, to minimize our number of sets in the solution, at each step we would need to cover the most number of elements that we can. Another way of doing this is picking a sub-problem which leaves us with minimum number of uncovered elements, U'.

c). How do you expand a sub-problem?

Sol c). Expanding a sub-problem given our solution is pretty straight-forward. We select all elements i for which Z_i = -1 and branch out two sub-problems for each element, one each for Z_i = 1 and Z_i = 0.

d). What is an appropriate lower-bound?

Sol d). An appropriate lower-bound should be the **number of subsets used so far** in the partial solution.

The choices that we have made above should work well on typical instances of the problem because we have built a generic solution approach for the given problem and have not based our solution on any particular instance.

Part 2: Outline a simple greedy heuristic for the SET COVER problem, and explain why it finds a valid solution and its running time.

Answer 2:

A simple greedy heuristic can be to add that subset to our SET COVER solution that covers the maximum number of remaining uncovered elements from the universe.

Here is the pseudo code for the algorithm:

$Input: Universe\ in\ the\ form\ of\ set\ of\ elements, U = \{u_1, u_2, u_3 \ldots \ldots u_n\}$
$Sets, S = \{S_1, S_2, S_3 \ldots \ldots S_m\}\ such\ that\ union\ of\ all\ S_i\ is\ equal\ to\ U$

$S' = \emptyset$
$U' = U$
$while\ U' \neq \emptyset:$
$S_{temp} = S_i \in (S - S') which\ gives \max |S_i \cap U'|$
$S' = S' \cup S_{temp}$
$U' = U' - S_{temp}$

Please note that it is safe to assume that there is some subset of subsets S_i, union for which covers the universe U.

This algorithm will always find a valid solution (if one exists) because it does not stop iterating till all the elements of the given universe have been visited.

Time complexity:

First line inside the while loop computes intersection which can take $O(m * n)$ as we can have at-most n elements and m sets. Next two lines in the while loop requires linear operations in terms of m and n, so out time complexity is still $O(m * n)$. Final step is to compute the number of iterations of the while loop. We see that while loop runs till U' becomes an empty set. If our subsets are such that one elements get removed from U' one at a time, then while loop runs for n number of iterations, otherwise it runs for m number of iterations. So, our complexity becomes $O(min(m, n) * m * n)$.

Part 3: Local Search

a). What could be a possible scoring function for such candidate solutions?

Sol a). Our solution tries to maximize the number of elements with as few subsets as possible. We will need to capture this criterion in our scoring function. One possible way of doing this can be building a scoring function by **calculating the number of elements captured by the subsets of a candidate solution and then dividing it by the number of sub-sets it contains**. Such a scoring function will favor those solutions which accumulate the most of elements from the universe with minimal of subsets.

b). What would be a Neighborhood (or Moves) you would consider using for your local search to move from one candidate solution to other 'nearby' solutions? How many potential neighbors can a candidate solution have under your Neighborhood (using Big-Oh)?

Sol b). A very good criterion for selecting a neighborhood can be to iteratively remove or add subsets, one at a time. In this way, our candidate solutions will differ by one subset. Addition will be for cases when we have to add a new subset as that subset contains elements which are not covered by other subsets in the partial solution so far. Removal will be for cases when we see a subset that covers all the elements covered by a particular subset in the partial solution and covers extra elements on top of that.

As for the terminology that we have used above, we can have at-most m number of subsets, so a candidate solution can have $O(m)$ potential neighbors.

c). Why would you consider adding Tabu Memory and what would be remembered in your Tabu Memory?

Sol c). Tabu memory is used for facilitating Tabu search. Tabu search is used for enhancing the performance of local search by relaxing the basic rule of local search i.e. accepting worsening moves or neighbors if no better neighbors are available and additionally we use prohibitions (done by using Tabu memory) to stop the search from coming back to already visited solutions. This is usually done **to prevent local search from getting stuck at local minimum**.

A very simple idea can be to keep a certain number of visited (added or removed) subsets, say p, in the memory for last p solutions and this memory can then be used to choose candidate solutions which contains subsets that have not already been visited.

QUESTION 2: MAKE AMERICA CONNECTED AGAIN

a). Devise a greedy algorithm that has an approximation ratio of 2. Give both the high level idea, and a pseudocode implementation of it.

Sol a). Given problem states that we have to partition the graph into two parts such that connections or edges between the two parts are maximized. Please note that this problem is nothing but a max-cut problem.

Consider each person a vertex and each friendship friendship as an edge between two vertices. Then we get a graph for the given community, C, with n vertices. Now, we need to build a greedy algorithm for partitioning the graph into two sets such that number of edges between the two sets are maximized.

Let's say there are n vertices, then each vertex or person can be assigned an integer id and it will be used for ordering the vertices. After assigning ids, order the vertices in increasing order, $p_1, p_2, p_3, \ldots\ldots, p_n$. **We will use a specific terminology, for any edge between p_i & p_j, where $i < j$, the edge will belong to vertex with a higher integer id, vertex p_j for this case.** Our greedy algorithm starts with two empty partitions S & S' and it starts by putting vertex v_1 in S. For each of the subsequent vertex, we check the number of edges that vertex make with S and S', then we put the vertex in the partition which forms the less number of edges with the vertex. This step ensures that we are maximizing the number of edges between two partitions at each step.

Here is the psuedocode for the greedy algorithm:

Start with two empty partitions S & S'.
Sort the vertices in increasing order of their ids.
Put vertex v_1 in S

for every subsequent vertex v_i:
    check $|\ v_i - S\ |$ and $|\ v_i - S'\ |$, (number of edges v_i has with vertices in S & S')

$if\ |\ v_i - S\ |\ >\ |\ v_i - S'\ |$:
    $put\ vi\ in\ S'$
$if\ |\ v_i - S\ |\ <\ |\ v_i - S'\ |$:
    $put\ v_i\ in\ S$
$if\ |\ v_i - S\ |\ ==\ |\ v_i - S'\ |$:
    $put\ v_i\ in\ either\ of\ S\ or\ S'$

b). Proof that greedy algorithm has an approximation ratio of 2.

Sol b). Let r_i be the number of edges that belongs to each vertex, v_i as per our terminology (edge belongs to that vertex which has the higher node id). Now, each edge can belong to exactly one vertex, we know that

$$\Sigma\, r_i = m, \quad \text{where } m \text{ is number of edges.}$$

Now, our greedy claim is that **adding each vertex, v_i will add at least $r_i/2$ to the cut**.

Proof: Now adding a vertex v_i, we get r_i edges. It means that we have r_i other vertices as well. Now, the partition that contains the most number of these r_i other vertices will have **atleast** $r_i/2$ vertices (because both the partitions cannot contain less than the average of total number of vertices). Hence, adding v_i to the partition which contain lesser number of those r_i other vertices will **atleast** add $r_i/2$ vertices to the cut.

So, final cut-size or number of edges crossing two partitions will be:

$= \Sigma$ Number of edges added by each vertex
$\geq \Sigma\ r_i/2$
$\geq m/2$ (because $\Sigma\, r_i = m$)

Since $opt(I) \leq m$ (number of edges, m), so our approximation ratio becomes 2.

c). Why isn't your greedy solution optimal? Give an example where your greedy algorithm does not achieve the optimal, but achieves twice the optimal.

Sol c). Optimal solution can at-max have a cut of size m, where m is the number of edges. So, our greedy solution is definitely not optimal. Main reason being, adding each vertex only adds **atleast** $r_i/2$ vertices to the cut, this lower bound is less than the bound we should have had for an optimal solution.

Here is an example of the graph, where our greedy algorithm achieves **twice** the optimal:

(1,3), (1,4), (2,3), (2,4) are the edges and 1,2,3 and 4 are the vertices.

We can get a cut size of 4 if we make two partitions, (1,2) and (3,4). But using our greedy algorithm, we can get the following partitions:

- (1,3) & (2,4)
- 1 & (2,3,4)
- (1,3,4) & 2

And cut size for each of these three partitions is 2. Hence, our greedy algorithm achieves twice the optimal.

Bonus. Prove that decision version of MACA is NP-complete.

Sol). Before we start proving this, we need to first state the decision version of max-cut problem:

For a given graph G, and given some number k, does there exist a cut of size atleast k in G.

We will use the standard 3 steps conversion for proving NP-completeness. Here are the three steps:

Step 1 ➔ Show that MACA or max-cut is in NP.

This can be done by showing that given a candidate solution for max-cut problem, it can be verified in polynomial time whether or not candidate solution is the actual solution for max-cut problem.

If we are given a candidate solution, in the form of a list of nodes say S then we automatically get two partitions, S and S' where $S' = V - S$. Now, let's see how we can verify whether or not this is the correct solution to max-cut problem. For each vertex in S, we can calculate the number of edges we have that are going to S' i.e. number of edges that are crossing the cut. So, time complexity for verifying the candidate solution will be $O(m)$, where m is the number of edges.. Hence, a candidate solution can be verified in polynomial time.

Step 2 ➔ We choose minimum vertex cover as problem X and it is known to be a NP-complete problem.

Step 3 ➔ Reduce minimum vertex cover to max-cut problem (problem Y) and then prove that if an input instance of minimum vertex cover is a solution then reduced instance is also a solution for max-cut problem and vice-versa.

Let us transform the graph G to G' by adding a new vertex x to it in such a way that every vertex u in G is connected by $deg(u) - 1$ parallel edges to x. New graph G' will have |V|+1 vertices and $|E| + \sum \deg(u) - 1$ (summation over $u \in V$). Please note that this transformation can be done in polynomial time.

Now, let U and $V - U + x$ be a cut in G'. We now need to prove that if U is a minimum size vertex cover in G then U and $V - U + x$ is a max-cut in G' and vice-versa.

Proof: if U is a minimum vertex cover in G then U and $V - U + x$ is a max-cut in G'.

An edge in the graph is said to be incident on a set of vertices B if the edge has one of its vertices in B. The number of edges incident on U in G', $N_G'(U)$ is given by the following equation:

$N_G'(U) = N_G(U) + \sum (\deg(u) - 1)$,
where $N_G(U)$ is the number of edges incident on U with the vertex w removed.

Above equation can also rewritten as:

$N_G'(U) = N_G(U) + \sum \deg(u) - |U|$

It can be noted that $N_G(U) + \sum \deg(u)$ equates to twice the number of edges with atleast one vertex in U. Rewrite the above equation, we get:

$N_G'(U) = 2\{ edges\ in\ G\ with\ atleast\ one\ vertex\ in\ U\} - |U|$

Now, if U is a vertex cover for G, above equation becomes:

$N_G'(U) = 2|E| - |U|$ → **Equation 1**

If U is a minimum vertex cover then our $N_G'(U)$ is maximized for the given U, in other words,

$N_G'(U) > N_G'(T)$, where T is any other set. Also, note that $N_G'(U)$ is nothing but number of edges crossing U and $V - U + x$. Hence, U results in a max-cut for G' if U is a minimum vertex cover for G.

Proof: If U and $V - U + x$ is a max-cut in G' then U is a minimum vertex cover in G

We first need to prove that U is a vertex cover then equation 1 can be used to prove that it is a minimum vertex cover in G.

We will prove this using proof by contradiction. If U is not a vertex cover in G, then it means that there is an edge (p, q) for which both the end point lies in $V - U + w$. Now we can make a new cut by adding one of the vertices of this edge say q to U. This increases the size of the cut by $deg_G(q) - 1 + 1$, this also decreases the size of the cut by **atmost** $deg_G(q) - 1$ (for the case, when all the edges involving q were in U but for one). Hence, there will a net increase of **atleast** 1 in the cut-size. It contradicts the max-cut property of U, hence U is a vertex cover in G. Using equation 1, if U is a vertex cover then U is also a minimum vertex cover. Hence, proved.

QUESTION 3: Modeling with NLP

a). Formulate minimum set cover problem as Integer Linear Program.

Sol a). Given an input (U, S_1, S_2, ……, S_n) of the set cover problem, we introduce a variable x_i for every set S_i, this variable will take value 1 when set S_i is selected in the vertex cover and 0 otherwise. Using this notation, set cover problem can be expressed as the following integer linear program:

Minimize $\sum x_i$ (summation over i, where $i = 1, 2, \dots, n$)
Subject to:

- $\sum_{i:v \in S_i} x_i \geq 1 \quad \forall v \in U$ (cover every element in the universe)
- $x_i \in \{0,1\}$ → a
- OR
- $x_i \leq 1$ → b
- $x_i \geq 0$ → c

We use one extra variable (which can take 2 values for n sets, hence 2*n) and three constraints or two constraints (either a or b&c).

b). Caleb and Karl camping trip.

Sol b). We have a set of k, s_1, s_2,……, s_k essential items for the trip such that $\sum s_i < 2C$. We introduce a variable x_i which will take value 1 if Caleb is carrying that item and 0 if Karl is carrying that item. Using this notation, this problem can be expressed as the following integer linear program:

Minimize $\sum x_i^1 - x_i^0$ (summation over i, where i = 1,2, … , k), minimize the number of sets
Subject to:

- $\sum x_i^1 \leq C$ & $\sum x_i^0 \leq C$ (each bag should have less than C capacity)

- $\sum x_i^1 \geq \sum x_i^0$ (each specific case explained below, Caleb should carry more than Karl)
- $\sum s_i < 2C$
- if $\sum s_i \ \% \ 2 == 0$, (If sizes are equal, both carry same weight)
    - $\sum x_i^1 = \sum x_i^0$
- $else$,
    - $\sum x_i^1 > \sum x_i^0$ (else, Caleb carry more weight)

We have one variable (which can take 2 values for k sets, hence 2*k) and 7 constraints.

c). Emily's hummingbird cake muffins

Sol c). We have a matrix where each row represents the items each stall (from 1 to n) is going to get for a day and each column represents the items a specific stall is going to get over n specific days. We are going to use a variable m to represent the matrix, d to represent the row index capturing the day, s to represent the column index capturing the stall, i to represent the item.

Here are the constraints:

- $m[d_1, s_1] = i_3$ (given constraint)
- $m[d_2, s_2] = i_2$ (given constraint)
- $m[d_3, s_1] = i_4$ (given constrain)
- $m[d_a, s] \neq m[d_b, s]$ (no two days can have the same item for a specific stall)
- $m[d, s_a] \neq m[d, s_b]$ (no two stalls can have the same item for a specific day)
- $m[d, :] \in \{1, n\}$ (each stall for a day will have 1 to n items)
- $m[:, s] \in \{1, n\}$ (each day, we will have 1 to n items assigned to n different stalls)
- $i \in \{1, n\}$ (i can take 1 to n values)
- $d \in \{1, n\}$ (d can take 1 to n values)
- $s \in \{1, n\}$ (s can take 1 to n values)

We have 4 variables (s can take n values, i can take n values, d can take n values, m will have n*n values) and 10 constraints.