CS 170 Homework 11

Due **Monday 4/15/2024, at 10:00 pm (grace period until 11:59pm)**

1 Study Group

List the names and SIDs of the members in your study group. If you have no collaborators, you must explicitly write "none".

Solution: I worked on this homework with the following collaborators:

- Lakshya Nagal, SID: 3037935253

2 Some Sums

Given an array $A = [a_1, a_2, \ldots, a_n]$ of nonnegative integers, consider the following problems:

1 **Partition**: Determine whether there is a subset $S \subseteq [n]$ ($[n] := \{1, 2, \cdots, n\}$) such that $\sum_{i \in S} a_i = \sum_{j \in ([n] \setminus S)} a_j$. In other words, determine whether there is a way to partition A into two disjoint subsets such that the sum of the elements in each subset equal.

2 **Subset Sum**: Given some integer x, determine whether there is a subset $S \subseteq [n]$ such that $\sum_{i \in S} a_i = x$. In other words, determine whether there is a subset of A such that the sum of its elements is x.

3 **Knapsack**: Given some set of items each with weight w_i and value v_i, and fixed numbers W and V, determine whether there is some subset $S \subseteq [n]$ such that $\sum_{i \in S} w_i \leq W$ and $\sum_{i \in S} v_i \geq V$.

For each of the following clearly describe your reduction and justify its correctness.

(a) Find a linear time reduction from Subset Sum to Partition.

Solution:

Reduction Algorithm: Given an instance of Subset Sum and its solution, define the sum of A to be $s = \sum_{i=1}^{n} a_i$, then we check if $s - 2x \geq 0$. If $s - 2x \geq 0$ then we can create a new array $A' = A \cup \{s - 2x\}$, otherwise, if $s - 2x < 0$, we can define $A' = A \cup \{2x - s\}$.

Justification:

- If $s - 2x \geq 0$, then the sum of elements in A' is $s + (s - 2x) = 2s - 2x$ and each subset's sum is $\frac{2s-2x}{2} = s - x = x$, which is exactly half of the total sum of A'.
- If $s - 2x < 0$, then the sum of elements in A' is $s + (2x - s) = 2x$. and each subset's sum should be $\frac{2x}{2} = x$.

Therefore we can partition A' into two disjoint sets such that each subsets sum is x.

(b) Find a linear time reduction from Subset Sum to Knapsack.

Solution:

Reduction Algorithm: Given an instance of Subset Sum, we can assign every element $a_i \in A$ a corresponding weight $w_i = a_i$ and value $v_i = a_i$. We can set $W = V = x$.

Justification: If we have a subset S in A such that the sum of $S = x$, then the total weight of these items will be $x = W$. Additionally, the total value of these items will also be $x = V$. This reduction can be performed in linear time, since we need only iterate through the original array once.

3 k-XOR

In the k-XOR problem, we are given n boolean variables $x_1, x_2, \ldots, x_n$, a list of m clauses each of which is the XOR of exactly k distinct variables (that is, the clause is true if and only if an odd number of the k variables in the clause are true), and an integer c. Our goal is to decide if there is some assignment of variables that satisfies at least c clauses.

(a) In the Max-Cut problem, we are given an undirected unweighted graph $G = (V, E)$ and integer α and want to find a cut $S \subseteq V$ such that at least α edges cross this cut (i.e. have exactly one endpoint in S). Give and argue correctness of a reduction from Max-Cut to 2-XOR.

Hint: every clause in 2-XOR is equivalent to an edge in Max-Cut.

Solution:

Reduction Algorithm: We can assign each vertex in the Max-Cut problem a boolean variable, for each edge in G create a 2-XOR clause between the adjacent vertices, we can assign the variables True or False values depending on whether they are on the left or right of the cut.

Justification: By assigning the variables True or False values depending on which side of cut they are on, we can create 2-XOR clauses for every edge. For any clause, if the vertices are on the same side, the clause will evaluate to False, and if they are on opposite sides of the cut, the clause will evaluate to True. If a clause evaluates to True it implies that the edge between the two vertices in the clause crosses the cut. All that is left to do is to check if at least c of these clauses are satisfiable. Conversely, given a instance of a 2-XOR problem and it's satisfying assignment, create a graph such that true literals are one side of the graph, and false literals are the opposite side of the graph. For each clause, add an edge from x_i to x_j if they are in the same clause. For each clause if the assignment evaluates to True, then we know that the edge between the two literals crosses the cut. Finally we can check if the number of True clauses is at least α.

(b) Give and argue correctness of a reduction from 3-XOR to 4-XOR.

Solution:

Reduction Algorithm: For every clause in the 3-XOR problem, we create a 4-XOR clause with the same variable as the 3-XOR clause and add a new variable y to the 4-XOR clause. For example if we had the 3-XOR clause $(x_i \oplus x_j \oplus x_k)$ the corresponding 4-XOR clause would be $(x_i \oplus x_j \oplus x_k \oplus y)$. We now have 2 possible cases:

- if y is false, the 3-XOR clause and the 4-XOR clause are equivalent in output.
- if y is true, then this turns out to be equivalent to flipping the assignment of every clause which means we now have $m - c$ true clauses.

Justification: Given an instance of the 3-XOR problem and its assignment, create new variable y in the 4-XOR instance. If y is false we know that by adding a false literal to the 4-XOR clause, we yield the same result as in the 3-XOR problem since XORing a number with 0 returns the same number. Additionally the number of true literals remains the same, so if there were an even number of true literals or an odd number of true literals, the 4-XOR clause would maintain that number. If y is true, we can negate the clause,

resulting in the same output as in the 3-XOR problem, with the same number of even or odd true literals as in the 3-XOR problem. Conversely, given a 4-XOR problem and its solution, for each clause evaluate the first two literals, create a new variable based on the result, and replace the evaluated literals with the new variable. Thus creating an instance of a 3-XOR problem, with the same number of true literals as in the 4-XOR problem.

4 Survivable Network Design

Survivable Network Design is the following problem:

> We are given two $n \times n$ matrices: a cost matrix d_{ij} and a connectivity requirement matrix r_{ij}, both of which are symmetric. We are also given a budget b. Our goal is to find an undirected graph $G = (\{1, ..., n\}, E)$ such that the total cost of all edges (i.e. $\sum_{(i,j)\in E} d_{ij}$) is at most b and there are exactly r_{ij} edge-disjoint paths between any two distinct vertices i and j; if no such G exists, output "None". (A set of paths is edge-disjoint if no edge appears in more than one of them)

Show that Survivable Network Design is NP-Complete.
Hint: Reduce from a NP-Hard problem in Section 8 of the textbook.

Solution:

We will show that Survivable Network Design is NP-Complete by reducing Rudrata Cycle to Survivable Network Design.

Reduction Algorithm Rudrata Cycle to Survivable Network Design:
Given an instance of Rudrata Cycle and its solution

- Set budget $b = n$
- $\forall (i, j) \in E$ set $d_{ij} = 1$ if $(i, j) \in$ Rudrata Cycle
- $\forall (i, j) \in E$ set $d_{ij} = \infty$ if $(i, j) \notin$ Rudrata Cycle
- $\forall (i, j)$ Set $r_{ij} = 2$

Proof Survivable Network Design is NP:
We can verify the solution to Survivable Network by running max flow on the solution graph G and checking that for any two pair of vertices (as our sources and sink) the r_{ij} and b constraints are met.

Proof $\rightarrow \exists$ solution to Rudrata Cycle $\Rightarrow \exists$ solution to Survivable Network Design:
We set the budget equal to the number of vertices so we account for every vertex in the Rudrata Cycle instance. For the cost matrix we assign the cost of 1 if the edge is part of the Rudrata cycle solution, and set it equal to ∞ otherwise. This ensures that if the edge is not in the Rudrata cycle, it will be greater than the budget and be ignored. r_{ij} is set to 2 for all pairs of vertices, indicating that there must be exactly two disjoint paths between any two pairs of vertices in the cycle. We now have a cycle such that the cost of the edges is equal to b and the r_{ij} requirements are met.

Proof $\leftarrow \exists$ solution to Survivable Network Design $\Rightarrow \exists$ solution to Rudrata Cycle:
Given an instance of Survivable Network Design and its solution, a graph G such that the total cost of all edges is $\leq b$ and there are exactly $r_i j$ $\forall (i, j)$. If we have a satisfying solution to Survivable Network Design problem, then this implies that in for every pair of vertices in G there are 2 edge disjoint paths. If there are 2 edge disjoint paths for every pair of vertices this implies that we have cycle such that the sum of all edge weights is $\leq b = n$, which means we have a Rudrata Cycle

5 Orthogonal Vectors

In the 3-SAT problem, we have n variables and m clauses, where each clause is the OR of (at most) three of these variables or their negations. The goal of the problem is to find an assignment of variables that satisfies all the clauses, or correctly declare that none exists.

In the orthogonal vectors problem, we have two sets of vectors A, B. All vectors are in $\{0,1\}^m$, and $|A| = |B| = n$. The goal of the problem is to find two vectors $a \in A, b \in B$ whose dot product is 0, or correctly declare that none exists. The brute-force solution to this problem takes $O(n^2 m)$ time: compute all $|A||B| = n^2$ dot products between two vectors in A, B, and each dot product takes $O(m)$ time.

Show that if there is a $O(n^c m)$-time algorithm for the orthogonal vectors problem for some $c \in [1, 2)$, then there is a $O(2^{cn/2} m)$-time algorithm for the 3-SAT problem. For simplicity, you may assume in 3-SAT that the number of variables must be even.

Hint: Try splitting the variables in the 3-SAT problem into two groups.

Solution: We can split the variables into groups of size $n/2$, call these groups A' and B'. For each clause we create a vector such that if the variable is in A' we add its assignment to the corresponding vector in set A, and if the variable is in B' we add its assignment to the corresponding vectors in B. Before we assess a new clause and create a new vector, we ensure that the size of the vectors in A and B are of size m by padding with 1's if needed. At the end we are left with two sets of vectors A and B such that they are sets of size n, with vectors of size m. We can use the $O(n^c m)$ algorithm to find two orthogonal vectors. If we have an $O(n^c m)$ algorithm for orthogonal vectors, then we use this to solve 3-SAT problem. By splitting variables into two groups we have $n/2$ variables in each group and each variable can be assigned to one of two values, so our $n = 2^{n/2}$, giving us a runtime of $O(2^{cn/2} m)$.