CSC373H1S 2014 **Assignment 3**
Due: Thursday, March 20th, 10PM Total Score: 10 points

Please follow the instructions provided on the course website to submit your assignment. You may submit the assignments in pairs. Also, if you use *any* sources (textbooks, online notes, friends) please cite them for your own safety.

For each of the problems below you will be asked either to write a linear program or prove that the problem is NP-Complete.

Questions

1. **Oh hi Mark!**

   Here is a generalization of the roommate-chore matching question from the previous assignment. As before, there are n roommates and m chores, and we want to distribute the work from each of the chores across the roommates. For simplicity, we will label each of the m chores with a unique integer from $\{1, 2, \dots, m\}$. This time, however, for each chore i you will receive a positive real number c_i representing how many hours of work this chore will require each week.

   For any chore i the n roommates can split up the work arbitrarily as long as the c_i hours of work will be completely finished. For each j, $1 \leq j \leq n$, the jth roommate is defined by $r_j = (P_j, z_j, h_j)$. The set $P_j \subseteq \{1, 2, \dots, m\}$ is a collection of chores, representing the chores that this roommate prefers to do. However, each roommate *is willing* to do the chores not in his set of preferred chores, but you must sweeten the deal with some hard cash: for each hour of work roommate j spends on a chore not in P_j, you must pay him z_j dollars (for some positive real number z_j). You do not have to pay a roommate any amount of money for them to do a preferred chore. Finally, h_j is a positive real number representing how many hours of work roommate r_j is willing to do in total (that is, including the contributions from every chore that they are working on).

   Greedy Roommates

   Input: Two positive integers m, n. For each i, $1 \leq i \leq m$, a positive integer c_i representing how many hours of work the ith chore will require this week. For each j, $1 \leq j \leq n$, a subset of preferred chores P_j for roommate r_j, the roommate's hourly price z_j for doing chores they do not prefer, and the number of hours h_j they are able to spend on doing chores in total.

   Output: The minimum amount of money that you need to pay each of your roommates in order to complete all of the hours of work for each chore, or infeasible if it is impossible to assign your roommates to chores within the given constraints.

   Give a linear programming formulation which solves the Greedy Roommates problem.

   Solution

   Here is a linear programming formulation for the problem. For each of the n roommates and m chores we introduce a variable a_{ij} which represents the number of hours that roommate i will spend on chore j. Here is a linear program for this problem:

$$\begin{aligned}
\text{min} \quad & \sum_{j=1}^{n} \sum_{i \notin P_j} z_j a_{ji} \\
\text{subject to} \quad & \sum_{j=1}^{n} a_{ji} = c_i, i = 1, 2, \ldots, m \\
& \sum_{i=1}^{m} a_{ji} \le h_j, j = 1, 2, \ldots, n \\
& a_{ji} \ge 0, i = 1, 2, \ldots, m, \text{ and } j = 1, 2, \ldots, n
\end{aligned}$$

The first constraint, for each chore i, requires that the total number of hours spent on that chore to be at least c_i. The second constraint, for each roommate j, requires that the roommate does not spend more than h_j hours on all of his chores. The final constraint restricts the number of hours on each chore to be positive. Finally, the objective function minimizes the number of hours that each roommate spends on chores that they do not prefer.

2. **Talk about MY generation**

The *weighted generation* problem is defined as follows. You are given a finite set of n points P, with two distinguished points s and t in P (and note that $s \neq t$). As well, you are given a set of *triples* $T \subseteq P^3$, where a *triple* consists of any three (ordered) points drawn from P: for example, the triple (x, y, z) is distinct from the triple (x, z, y). There is also a weight function $w : T \to \mathbb{R}^+$, which weights each triple with a positive real number.

Given P and T, we say that a point $z \in P$ can be *generated* from T if either:

(a) $z = s$ (that is, s is always generated), or

(b) there is a triple $(x, y, z) \in T$, and both x and y can be generated.

The *cost* of generating a point z from x, y using the triple $\ell = (x, y, z)$ is

$$cost(z) := \min \{cost(x), cost(y)\} + w(\ell).$$

The cost of generating s is 0. There may be many ways of generating each point in P, and in this problem you will be interested in finding the way of generating the point t with the minimum overall cost.

Weighted Generation

Input: A set P of n points, with two distinguished points s and t. A set of triples $T \subseteq P^3$, along with a function $w : T \to \mathbb{R}^+$ weighting each triple with a positive real number.

Output: The minimum cost necessary for generating the point t.

Give a linear programming formulation which solves the Weighted Generation problem.

Solution

For the linear programming formulation of this problem we follow the maximization linear program for the s-t shortest-path problem. For each point j in the input set P we introduce a variable d_j which represents the minimum cost to generate the point j. For the same point we also introduce a parallel variable f_j, which is used to determine if t can be feasibly generated or not.

$$\begin{aligned}
\text{max} \quad & d_t + f_t \\
\text{subject to} \quad & \forall (i,j,k) \in T : d_k \le d_i + w(i,j,k), \quad d_k \le d_j + w(i,j,k) \\
& \forall (i,j,k) \in T : f_k \le f_i + f_j + w(i,j,k) \\
& d_s = 0 \\
& f_s = 0 \\
& d_i \ge 0, i \in P \\
& f_i \ge 0, i \in P
\end{aligned}$$

For each point j and triple (i,j,k) which generates j, the first constraint minimizes the cost to generate k using either the point i or the point j. The second constraint is used only to detect if the point k can be generated: if there is no triple bounding the value of f_k, then it will grow unboundedly. If this is the case and there is no way to generate t, then f_t will also grow unboundedly, and the linear program will return "unbounded". The third and fourth constraints says that the cost to generate s is 0, which is always true. The final constraints restrict the distances to be non-negative (which, strictly speaking, is not necessary since this is a maximization problem and all weights are positive).

3. **Take that, you stupid string!**

In this problem we will be considering strings of bits. If y is a string of bits (for example, $y = 00001$) then we will use the notation $y[i]$ to denote the ith character in y. (In the example, $y[1] = 0$ and $y[5] = 1$.) We define the *Hamming weight* of a bit string y, denoted $|y|$, to be the number of 1s in the string y. In the example above the Hamming weight of the string y is $|y| = 1$.

Hitting String

Input: A positive integer k. A set A of strings of bits, where each string in A has length n.

Output: Return 1 if and only if there is a string x of n bits, with $|x| \le k$, such that for every string $y \in A$ there exists an index i where $x[i] = y[i] = 1$. (Such a string x is called a *hitting string*.)

Part a.

Show that this problem is in NP by a reduction to one of the NP-Complete problems discussed in the lectures or tutorials. Then show that it is NP-Hard by a reduction from the Vertex Cover problem.

Part b.

The *search version* of the Hitting String problem instead asks for you to actually return the hitting string x, if it exists (rather than simply return 1). Give a polynomial time search-to-decision reduction for the Hitting String problem. That is, assuming that you have access to a polynomial time algorithm H which solves the Hitting String problem, give a polynomial time algorithm which solves the search version of the Hitting String problem.

Solution for Part a.

We give a reduction from the Hitting String problem to the Circuit-SAT problem. Let A be a set of m strings $y_1, y_2, \ldots, y_m$, each with n bits, and let k be a positive integer. For each $i = 1, 2, \ldots, k$ and $j = 1, 2, \ldots, n$ our Circuit-SAT instance will have a variable a_{ij}, which will be 1 if and only if the jth bit of the target string x is the ith 1. That is, we imagine constructing the hitting string x as an assignment problem, where we have n possible places in the string x to assign each of our k 1s. We have to add subformulas to the Circuit-SAT instance which prevent things like multiple 1s being assigned to the same place in the string x, and the same 1 being assigned to multiple indices. We describe each of these subformulas one at a time, and our final circuit will simply take the "AND" of all of these formulas.

The first formula requires that for each $i = 1, 2, \ldots, k$, the ith 1 is assigned to at least one of the n indices in the string x:

$$\bigvee_{j=1}^{n} a_{ij}.$$

The second formula says that the ith 1 cannot be assigned to two different indices j_1, j_2 in x:

$$\bigwedge_{j_1 \neq j_2} \neg a_{ij_1} \vee \neg a_{ij_2}.$$

The next formula says that no pair of distinct 1s can be assigned to the same index j:

$$\bigwedge_{i_1 \neq i_2} \neg a_{i_1 j} \vee \neg a_{i_2 j}.$$

Finally we add a collection of m formulas, one for each string y_ℓ in the input set A, which requires that at least one of the 1s in the string y will be in the same place as a 1 in the string x. For each $\ell = 1, 2, \ldots, m$:

$$\bigvee_{j=1}^{n} \left(y_\ell[j] \wedge \left(\bigvee_{i=1}^{k} a_{ij} \right) \right).$$

The final circuit will simply AND each of the formulas together.

Now we prove that the reduction is correct: that is, there exists a hitting string x for the input set A with $|x| \leq k$ if and only if there exists an assignment to the constructed circuit which will make the circuit output a 1. Suppose that x is a hitting string, and let $d_1, d_2, \ldots, d_k$ be the k distinct indices in x which are set to 1. For $i = 1, 2, \ldots, k$ we set $a_{id_i} = 1$ (so the ith 1 is set to be the index d_i), and the rest of the variables to 0. Since the circuit takes the AND of each of the subformulas, we simply need to show that each subformula is satisfied.

The first formula is clearly satisfied, as all k of the 1s are assigned to indices in the string x. The assignment also implies that no 1 is assigned to multiple indices and no index is assigned multiple 1s, so the second and third formulas are satisfied. The final family of formulas is satisfied for each string y_ℓ as x is a hitting string, and there must be an index j such that $y_\ell[j] = x[j]$.

Now assume that there is a satisfying assignment to the variables of the circuit. The first three subformulas must be satisfied, which implies that we can treat each of the variables $a_{1j_1}, a_{2j_2}, \ldots, a_{kj_k}$ which have value 1 in the satisfying assignment as a mapping of k 1s to indices in a hitting string x. The last family of formulas must also be satisfied, which implies that for each string y in the input instance the constructed hitting string x must have a 1 in the same index as y.

The reduction is complete, and since Circuit SAT is in NP it follows that Hitting String is in NP.

Hardness

Now we show that Hitting String is NP-Hard by a reduction from the Vertex Cover problem. Let $G = (V, E)$ be an undirected graph and let c be a positive integer. We construct a set of strings A and an integer k such that the set A has a hitting string x with at most k 1s if and only if G has a vertex cover with at most c vertices.

Let $n = |V|$, $m = |E|$, and write $V = \{1, 2, \ldots, n\}$. The set A will have m strings, each with exactly n bits. For each edge $e = (i, j)$, introduce a string y_e into A which has exactly two 1s: the first 1 will be at index i and the second 1 will be at index j. This set A can clearly be constructed in $O(nm)$ time by just iterating through each of the edges and then constructing an n bit string for each edge in the natural way. Finally we set $k = c$.

Now we prove that the reduction is correct. Assume that $S \subseteq V$ is a vertex cover with $\ell \leq c = k$ vertices. If $S = \{i_1, i_2, \ldots, i_\ell\}$, we construct a hitting string x by setting $x[i_j] = 1$ for all $j = 1, 2, \ldots, c$, and every other bit in x to 0. Clearly x will have at most $k = c$ bits set to 1. If $e = (i, j)$ is any edge in the input, it follows that either i or j is in S, and so either $x[i] = 1$ or $x[j] = 1$. Therefore the string y_e must be covered, since

$y_e[i] = y_e[j] = 1$. Since this is true for every edge in the input, and every string in A has a corresponding edge in E, then every string in A must be hit by x.

Now assume that x is a hitting string for the set A, and let $i_1, i_2, \ldots, i_f$ be the $f \leq k$ indices of x which contain a 1. If y_e is a string in the set A corresponding to an edge $e = (i, j)$, then $y_e[i] = y_e[j] = 1$, and so in x either $x[i] = 1$ or $x[j] = 1$. Therefore, if we define $S \subseteq V$ to be the set of vertices $S = \{i_1, i_2, \ldots, i_f\}$ then the corresponding edge $e = (i, j)$ must have one of its endpoints in S. So S is a vertex cover, and since $f \leq k = c$ we have that S is a solution to the vertex cover instance.

Solution for Part b.

Let H be a polynomial time algorithm solving the Hitting String problem, and we show how to use this algorithm to construct a hitting string x for an arbitrary instance.

Algorithm 1: HittingString

```
Input: A set of $m$ strings $A$, each with $n$ bits. A positive integer $k$.
Output: A hitting string $x$ for $A$ with $|x| \leq k$, if it exists.
if $H(A, k) = 0$ then
    return No hitting string exists
end
Let $A = \{y_1, y_2, \ldots, y_m\}$;
Let $x$ be an array of length $n$, initialized to 0s;
$\ell = k$;
for $i = 1, 2, \ldots, n$ do
    $S = A$;
    /* Check if setting $x[i] = 1$ will result in a valid hitting string */
    for $j = 1, 2, \ldots, m$ do
        if $y_j[i] = 1$ then
            Remove $y_j$ from $S$;
        end
    end
    if $H(S, \ell - 1) = 1$ then
        $A = S$;
        $\ell = \ell - 1$;
        $x[i] = 1$;
    end
end
return $x$
```

Our construction works as follows. First, it checks if any hitting string exists using the algorithm H. If not, it halts and rejects. Otherwise, it iterates through each index $1, 2, \ldots, n$ of the potential string x, and tries to set $x[i] = 1$. It removes all of the strings in A which would be covered if $x[i] = 1$, and then tests if the set of strings that remain has a hitting string. If it does not then it assumes that $x[i] = 0$ and continues. If it does have a hitting string, then it sets $x[i] = 1$ and removes the covered strings from A permanently. The algorithm clearly runs in $O(nm + nh)$ time, where h is the running time of the algorithm H, and this is a polynomial time algorithm assuming H is a polynomial time algorithm.

Why does the algorithm correctly return a hitting string? If the set does not have a hitting string, then the algorithm immediately rejects. Otherwise, as the algorithm constructs the string x bit by bit, it will only set a bit in x to 1 if the remaining set of uncovered strings has a hitting string within the new bound $\ell - 1$. Since we remove all strings that are covered by the partially constructed hitting string x, we can be guaranteed that for each index i that has already been examined, either $x[i] = 1$ and any string y with $y[i] = 1$ was removed, or $x[i] = 0$ and so any hitting string will have to hit y at a later index. Thus, it will never set a bit in x to 1 if doing so would prevent the rest of the hitting string x from being constructed.

4. **Puzzles are fun!**

Let m and n be positive integers and consider the following game – called Pebble Up – played on an $n \times n$ grid. We will denote the square in the ith row of the grid and the jth column of the grid by the pair (i, j). For each $1 \leq i, j \leq n$, the square (i, j) will be given a *colour* (either black or white) and a *number* (some positive integer $k \leq m$). The grid, along with a given colour and integer for each square, will be referred to as a *game board.*

Pebble Up is played on the game board as follows. You have a bag of black and white pebbles, and for each integer k between 1 and m, you must choose to place either a black or a white pebble on all of the squares labelled with the integer k (so, squares with different numbers can have different coloured pebbles, but if two squares have the same number then they must have the same coloured pebble). After choosing a coloured pebble for each integer k, the game moves on to the next phase.

For each square (i, j), with $1 \leq i, j \leq n - 1$, examine the set of four squares

$$S_{ij} = \{(i,j), (i+1,j), (i,j+1), (i+1,j+1)\}.$$

This set of four squares is said to be *winning* if at least one of squares has the same colour as the pebble lying on it. You *win* the game if every set S_{ij} is winning.

Figure 1 depicts a game board (with $n = 3, m = 5$) along with two configurations of pebbles. The middle configuration is a winning configuration: any set S_{ij} of four squares has at least one pebble which has the same colour as its square. If we switch the colour of the "5" pebble from white to black, we get the configuration on the right. This is not a winning configuration since the set S_{12}, consisting of the four squares in the top-right corner of the game board, has each square and pebble differing in colour.

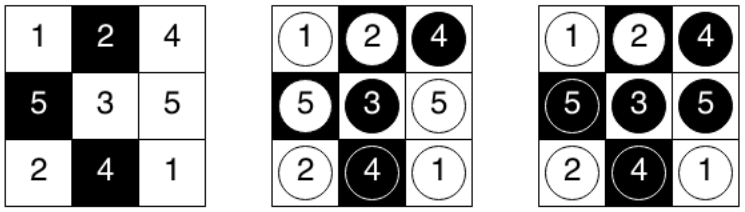

Figure 1: A game board and two possible configurations of pebbles

Pebble Up

Input: Two positive integers m, n, and for every (i, j) a colour b_{ij} (either black or white) and a positive integer $k_{ij} \leq m$.

Output: 1 if and only if it is possible to win the Pebble Up game on the given game board.

Prove that Pebble Up is NP-Complete.

First we show that Pebble-Up is in NP by giving a polynomial-time verifier for it. The certificate for the Pebble-Up problem will be interpreted by the verifier as an assignment of pebbles to squares of the board. The verifier then iteratively checks all sets of 4 squares of the board (which takes $O(n^2)$ time) to see if this assignment of pebbles causes each set to be winning. If so, the verifier returns 1, and otherwise it returns 0.

Now we prove that Pebble-Up is NP-Hard by a reduction from 4-SAT. Let $C_1, C_2, \ldots, C_m$ be a collection of clauses, each containing at most four literals, defined over the variables $x_1, x_2, \ldots, x_n$. The first step is to pad each of the clauses in the input which have less than 4 variables. This is easy: introduce 6 boolean variables $h_1, t_1, t_2, o_1, o_2, o_3$. We use h_1 to pad the clauses with 3 variables, t_1 and t_2 to pad the clauses with 2 variables, and o_1, o_2, o_3 to pad the clauses with 1 variable. If $C = z_1 \vee z_2 \vee z_3$ is a clause with three variables, we replace it with the two clauses

$$(z_1 \vee z_2 \vee z_3 \vee h_1) \wedge (z_1 \vee z_2 \vee z_3 \vee \neg h_1),$$

and both of these clauses are satisfiable if and only if the original clause is. Similarly, if $C = z_1 \vee z_2$ then we replace it with four clauses

$$(z_1 \vee z_2 \vee t_1 \vee t_2) \wedge (z_1 \vee z_2 \vee \neg t_1 \vee t_2) \wedge (z_1 \vee z_2 \vee t_1 \vee \neg t_2) \wedge (z_1 \vee z_2 \vee \neg t_1 \vee \neg t_2).$$

These four clauses are satisfiable if and only if the original clause is. If C only has one variable, we replace it with 8 clauses in the same way (by adding every combination of the three variables o_1, o_2, o_3 to C). The resulting 4-SAT instance will have at most $8m$ clauses (each with 4 literals) and 6 more variables, which is certainly within a polynomial of the size of the original instance.

The idea of the reduction is as follows. We construct the game board so that for each clause $C = z_1 \vee z_2 \vee z_3 \vee z_4$, there will exist a set of four squares $S_{ij} = \{(i,j), (i+1,j), (i,j+1), (i+1,j+1)\}$ such that the clause C can be satisfied if and only if there is a winning assignment of pebbles to S_{ij}. For each z_i, let ℓ_i be the label of z_is variable (so, if $z_i = \neg x_3$, then $\ell_i = 3$) and let c_i be black if z_i is negated and white otherwise. We define

$$\begin{aligned} b_{ij} &= c_1, \quad k_{ij} = \ell_1 \\ b_{i,j+1} &= c_2, \quad k_{i,j+1} = \ell_2 \\ b_{i+1,j} &= c_3, \quad k_{i+1,j} = \ell_3 \\ b_{i+1,j+1} &= c_4, \quad k_{i+1,j+1} = \ell_4 \end{aligned}$$

Figure 2 presents this reduction. In this way, there is a natural correspondence between assignments to the variables in the clause and pebbles placed on squares: black pebbles correspond to a variable being assigned 0, and white pebbles correspond to a variable being assigned a 1.

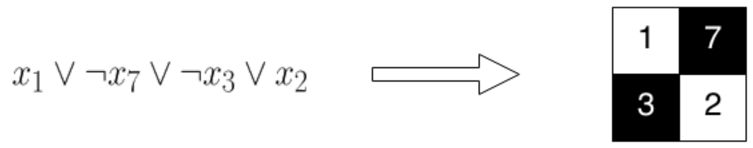

Figure 2: Reducing a clause to a set of 4 squares

To prevent neighbouring clauses from conflicting with one another, we further pad the game board with extra rows and columns that separate the clauses. We fix the same number in each square in all separating rows and columns, and alternate colours. Such a padding is shown in Figure 3 (where C_1, C_2, C_3, and C_4 are separated clauses). With the separating rows and columns labelled in this way, all of the intermediate sets of 4 squares

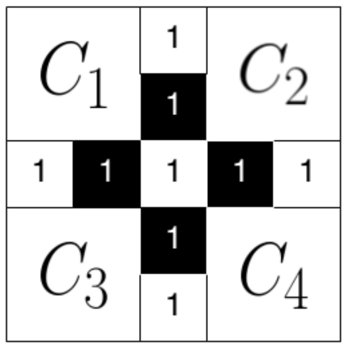

Figure 3: Padded clauses

which do not correspond to any fixed clause must overlap a separating row or column in at least 2 squares, and so it will be satisfied for any choice of pebble on the number.

We have shown how to construct each individual set of 4 squares from each clause, and also how to separate the clauses. Constructing the board is now simple: we are embedding m clauses into a square grid. Let $k = 2\lceil\sqrt{m}\rceil - 1$, and the game board will be a k by k grid. We can embed each of the clauses into the grid in any order, placing a separating row and column in between the clauses as described above. If there is leftover "room" in the grid, we pad it with more squares having a fixed number and alternating colour so that they will always be winning. This reduction can be performed in $O(k^2) = O(m^2)$ time.

Proving the reduction is correct is easy. Each number on the game board corresponds to a variable in the 4-SAT instance, and the colour of a pebble placed on all squares with that number corresponds exactly to an assignment to that variable. Assume that if a number receives a black pebble, the corresponding variable is set to 0 and vice-versa. Otherwise the variable is set to 1. If there is a satisfying assignment to the 4-SAT instance, we can construct a pebbling of the game board in exactly the way described above. Each set of 4-squares is either vacuously satisfied (as they are the result of padding), or correspond exactly to one of the 4-SAT clauses and thus must be satisfied. Conversely, if there is a winning configuration of pebbles on the game board, then we can construct a satisfying assignment using the method above and the 4-SAT instance must also be satisfied.