**به نام خدا**

**آنالیز الگوریتم ها**
**نیم‌سال دوم ۰۱-۰۲**

**دانشکده علوم ریاضی**

**تمرین ۵- امیرحسین محمودی - ۹۸۱۰۸۷۷۹**

## 1

This problem is NP-complete. To show this, we can reduce the 3-dimensional matching problem, which is a known NP-complete problem, to this problem in polynomial time.

The 3-dimensional matching problem is: given a set of triples of elements from three disjoint sets X, Y, and Z, is there a subset of triples that covers each element exactly once? For example, if X = a, b, c, Y = 1, 2, 3, Z = x, y, z, and the set of triples is (a, 1, x), (a, 2, y), (b, 2, x), (b, 3, z), (c, 1, z), then the answer is yes, because we can choose (a, 2, y), (b, 3, z), (c, 1, x).

To reduce this problem to the lowercase uppercase letter selection problem, we can do the following:

- For each element in X, Y, and Z, create a sequence of letters that contains both the lowercase and uppercase versions of that element. For example, for a in X, we create the sequence Aa.

- For each triple in the set of triples, create a sequence of letters that contains only the lowercase versions of the elements in that triple. For example, for (a, 1, x), we create the sequence a1x.

- The resulting set of sequences is an instance of the lowercase uppercase letter selection problem. The answer is yes if and only if there is a 3-dimensional matching for the original problem.

To see why this reduction works, suppose there is a 3-dimensional matching for the original problem. Then we can select a letter from each sequence as follows:

- For each sequence that corresponds to an element in X, Y, or Z, select the uppercase version of that element.

- For each sequence that corresponds to a triple in the matching subset, select any letter from that sequence.

- This way, we avoid selecting both the lowercase and uppercase versions of any letter.

Conversely, suppose there is a way to select a letter from each sequence without selecting both the lowercase and uppercase versions of any letter. Then we can construct a 3-dimensional matching for the original problem as follows:

- For each sequence that corresponds to a triple in the set of triples, if we selected a letter from that sequence, then include that triple in the matching subset.

- This way, we cover each element in X, Y, and Z exactly once.

Therefore, this reduction shows that the lowercase uppercase letter selection problem is

NP-hard. It is also easy to see that it is in NP, because we can verify a solution by checking that no letter appears in both lowercase and uppercase forms. Hence, it is NP-complete.

## 2

The maximum satisfiability problem (MAX-SAT) is the problem of determining the maximum number of clauses, of a given Boolean formula in conjunctive normal form, that can be made true by an assignment of truth values to the variables of the formula. It is a generalization of the Boolean satisfiability problem, which asks whether there exists a truth assignment that makes all clauses true. It is also NP-hard and APX-complete, which means it is difficult to find an exact or an approximate solution in polynomial time.

One possible algorithm that always gets at least half of the answer of MAX-SAT is to randomly assign each variable to be true with probability 1/2. This gives an expected 2-approximation, meaning that the expected number of satisfied clauses is at least half of the optimal number . This algorithm can be derandomized using the method of conditional probabilities.

One possible non-randomized algorithm is to use the method of conditional probabilities to derandomize the random algorithm. The idea is to choose the truth value for each variable that maximizes the expected number of satisfied clauses, given the previous assignments. This can be done by computing the conditional probabilities for each variable and clause, and then picking the value that gives the highest expectation. This algorithm also gives a 2-approximation, but it is deterministic.

The time complexity of the algorithm depends on how the probability of a clause is computed. One possible way to compute the probability of a clause is to use dynamic programming, as described in . This takes $O(nm)$ time, where n is the number of variables and m is the number of clauses. Therefore, the overall time complexity of the algorithm is $O(n^2m)$, since it iterates over n variables and m clauses for each variable.

## 3

### 3.1

If an investor, represented by $y_j$, is selected, then all of the investor's actors, represented by $x_i$ for all i in $L_j$, must also be selected. The profit in this situation is the residual income that remains after paying the actors.

maximize :

$$\sum_{i}^{m} y_i \cdot p_i - \sum_{j}^{n} x_j \cdot s_j$$

subject to:

$$0 <= x_i <= 1 \; for \; all \; i$$

$$0 <= y_j <= 1 \; for \; all \; j$$

$$x_i >= y_j \; for \; all \; i \in L_j$$

**Algorithm 1** Derandomized Algorithm for half-MAX-SAT

```
1:  function Derandomized($F$)
2:      $A \leftarrow \emptyset$                                  ▷ Initialize an empty assignment
3:  for ( $x \in F.variables$ ) {
4:      $E_{\text{true}} \leftarrow 0$     ▷ Initialize the expected number of satisfied clauses for assigning $x$ to true
5:      $E_{\text{false}} \leftarrow 0$    ▷ Initialize the expected number of satisfied clauses for assigning $x$ to false
        for ( $c \in F.clauses(x)$ ) {
6:          $P_c \leftarrow$ probability($c, A$)       ▷ Compute the probability that $c$ is satisfied given $A$
7:      if $x \in c.positive$ then
8:              $E_{\text{true}} \leftarrow E_{\text{true}} + P_c$                 ▷ Increment $E_{\text{true}}$ by $P_c$
9:              $E_{\text{false}} \leftarrow E_{\text{false}} + (1 - P_c)$         ▷ Increment $E_{\text{false}}$ by $(1 - P_c)$
            end

            if $x \in c.negative$ then
10:             $E_{\text{true}} \leftarrow E_{\text{true}} + (1 - P_c)$           ▷ Increment $E_{\text{true}}$ by $(1 - P_c)$
11:             $E_{\text{false}} \leftarrow E_{\text{false}} + P_c$               ▷ Increment $E_{\text{false}}$ by $P_c$
            end
        }
        if $E_{true} \geq E_{false}$ then
12:         $A[x] \leftarrow$ True
        end
        else
13:         $A[x] \leftarrow$ False
        end
        }

14:     return $A$                                               ▷ Return the assignment
15: end function
```

### 3.2

the variables can take any value varying from 0 to 1.

## 4

### 4.1

Given a graph G and an integer n, the graph coloring problem asks whether there exists a way of coloring the vertices of G with n colors such that no two adjacent vertices have the same color. If such a coloring exists, it is called an n-coloring of G.

### 4.2

Let n be the number of vertices of G and let $x_{ij}$ be a binary variable that indicates whether vertex i is colored with color j or not. Then the ILP formulation is:

minimize z subject to:

$$z >= i \times x_{vi} \ for \ all \ i, v$$

$$\sum_i x_{vi} = 1 \ for \ all \ v \in V$$

$$x_{vi} + x_{ui} <= 1 \ for \ all \ u, v \ d(v, u) <= i$$

$$x_{vi} \in 0, 1 \ for \ all \ i, v$$

The objective function z represents the number of colors used in the coloring. The first constraint ensures that z is at least as large as the largest color used. The second constraint ensures that each vertex is assigned exactly one color. The third constraint ensures that no two adjacent vertices have the same color. The last constraint defines the domain of the variables.

The optimal solution of this ILP problem gives the chromatic number of G and a corresponding coloring.

## 5

we construct a polynomial-time reduction from the standard Hamiltonian cycle problem.

Start with an arbitrary graph G (without edge weights).

Create an edge-weighted graph H from G by assigning each edge weight 0.

H contains a heavy Hamiltonian cycle if and only if G contains a Hamiltonian path: If G has a Hamiltonian cycle C, then C is also a heavy Hamiltonian cycle in H. If H has a heavy Hamiltonian cycle C, then C is also a Hamiltonian cycle in G.

more detailed: Suppose G has a Hamiltonian cycle C. The total weight of C is at least half the total weight of all edges in H, because 0   0/2. So C is a heavy Hamiltonian cycle in H. Suppose H has a heavy Hamiltonian cycle C. By definition, C is also a Hamiltonian cycle in G.

## 6

### 6.1

The problem of determining whether a graph contains a clique of size 3 can be proven to be in NP using the verifier-based definition of NP. A certificate for the problem can be the nodes in a clique. A verifier can then check if these nodes form a clique by counting the number of edges between them. Given a clique in the graph, it is easy to verify in polynomial time that there is an edge between every pair of vertices. Hence, a solution to DEG3-CLIQUE can be checked in polynomial time. Since the maximum number of edges and nodes in a clique is polynomial, verifying each node takes polynomial time. Additionally, the number of cliques in a graph is also polynomial due to the limit of 3 edges on each vertex. Therefore, the verifier runs in polynomial time, and the DEG3-CLIQUE problem is in NP.

### 6.2

This reduction does not prove anything about the NP-completeness of DEG3-CLIQUE. Instead, it suggests that CLIQUE is at least as hard as DEG3-CLIQUE, not the other way around. The reduction is in the wrong direction. To show that DEG3-CLIQUE is at least as hard as CLIQUE, we must reduce CLIQUE to DEG3-CLIQUE.