Answer the questions in the boxes provided on the question sheets. If you run out of room for an answer, add a page to the end of the document.

Name: Gurusharan Kunusoth Wisc id: kunusoth@wisc.edu

Randomization

1. *Kleinberg, Jon. Algorithm Design (p. 782, q. 1).*

3-Coloring is a yes/no question, but we can phrase it as an optimization problem as follows.

Suppose we are given a graph $G = (V, E)$, and we want to color each node with one of three colors, even if we aren't necessarily able to give different colors to every pair of adjacent nodes. Rather, we say that an edge (u, v) is *satisfied* if the colors assigned to u and v are different. Consider a 3-coloring that maximizes the number of satisfied edges, and let c^* denote this number. Give a polynomial-time algorithm that produces a 3-coloring that satisfies at least $\frac{2}{3}c^*$ edges. If you want, your algorithm can be randomized; in this case, the expected number of edges it satisfies should be at least $\frac{2}{3}c^*$.

Initialize a random 3-coloring for all nodes in the graph G. Calculate the number of satisfied edges edges, denoted as c. Repeat the following steps for a predefined number of iterations or until a termination condition is met: a) Randomly select a node u from the graph G. b) save the current color of u. c) For each color in the remaining two colors: i) Assign the selected color to node u. ii) Calculate the number of satisfied edges, denoted as c'. iii) If c' is greater than c, update c to c' and save the current coloring. iv) If c' is equal to c, flip a biased coin (eg: with a probability of 2/3) to decide whether to save the current coloring. v) If c' is equal to c, flip a biased coin. v) If c' is equal to c, flip a restore the original color of node u. d) If the termination condition is met, exit the loop. e) Output the final saved coloring. The algorithm works by iteratively exploring different colorings. At each iteration, it randomly selects a node & tries assigning each of the remaining two ~~c~~ colors to it. If assigning a new color increases the number of satisfied edges, it updates the count & saves the new coloring. If the count remains the same, it has a biased probability of the saving the new coloring.

2. *Kleinberg, Jon. Algorithm Design (p. 787, q. 7).*

In lecture, we designed an approximation algorithm to within a factor of 7/8 for the MAX 3-SAT Problem, where we assumed that each clause has terms associated with three different variables. In this problem, we will consider the analogous MAX SAT Problem: Given a set of clauses $C_1, \cdots, C_k$ over a set of variables $X = \{x_1, \cdots, x_n\}$, find a truth assignment satisfying as many of the clauses as possible. Each clause has at least one term in it, and all the variables in a single clause are distinct, but otherwise we do not make any assumptions on the length of the clauses: There may be clauses that have a lot of variables, and others may have just a single variable.

(a) First consider the randomized approximation algorithm we used for MAX 3-SAT, setting each variable independently to true or false with probability 1/2 each. Show that in the MAX SAT, the expected number of clauses satisfied by this random assignment is at least $k/2$, that is, at least half of the clauses are satisfied in expectation.

Each clause may have a different no. of terms, and there are no assumptions on the length of the clauses. Our goal is to find a truth assignment that satisfies as many clauses as possible. To we follow similar approach to Max 3-SAT. We set each variable independently to true or false with a probability of 1/2 each. Lets denote the number of clauses satisfied by the random assignment as X. We want to show that the expected value of X is atleast k/2.

$E(x) = E(x_1 + x_2 \cdots x_k] = E[X_1] + E[x_2] \cdots + E[X_k]$

$= 1/2 \wedge |c_1| + 1/2 \wedge |c_2| \cdots + 1/2 \wedge |c_k|$.

$E[X] \geq 1/2 + 1/2 \cdots 1/2$ (k terms)

$= k/2$.

(b) Give an example to show that there are MAX SAT instances such that no assignment satisfies more than half of the clauses.

clauses: C_1:(x_1 or x_2) C_2:(NOT x_1 OR x_2) C_3:(x_1 or NOT x_2) C_4:(NOT x_1 OR NOT x_2)

1. If we set both x_1 & x_2 to true, C_1 & C_3 are satisfied, but C_2 & C_4 are not satisfied. So, only 2 out of 4 clauses are satisfied.
2. If we set both x_1 & x_2 to false. C_2 & C_4 are satisfied, but C_1 & C_3 are not satisfied. Only 2 out of 4 satisfied.
3) If we set x_1 to true & x_2 to false, C_1 & C_4 are satisfied but C_2 & C_3 are not.
4) If we set x_1 to false & x_2 to true, C_2 & C_3 are satisfied but C_1 & C_4 are not satisfied.

This shows that there are instances where no assignment satisfies more than half of the clauses.

(c) If we have a clause that consists only of a single term (e.g., a clause consisting just of x_1, or just of $\overline{x_2}$), then there is only a single way to satisfy it: We need to set the corresponding variable in the appropriate way. If we have two clauses such that one consists of just the term x_i, and the other consists of just the negated term $\overline{x_i}$, then this is a pretty direct contradiction. Assume that our instance has no such pair of "conflicting clauses"; that is, for no variable x_i do we have both a clause $C = \{x_i\}$ and a clause $C' = \{\overline{x_i}\}$. Modify the randomized procedure above to improve the approximation factor from 1/2 to at least 0.6. That is, change the algorithm so that the expected number of clauses satisfied by the process is at least $0.6k$.

Initialize a random truth assignment for all variables in the instance. Initialize a count of the number of satisfied clauses, denoted as satisfied_count to 0. For each clause Ci in the instance: a. If Ci consists of a negated term, check if the variable in the term is set opposite to the corresponding assignment. If satisfied, increment satisfied_count by 1. c. If Ci has more than one term, proceed with the original randomized assignment for that clause shown below.
a. Randomly select a clause Ci from the instance.
b. Save the current truth assignment for all variables.
c. Set the variables in Ci according to a random assignment. d. Calculate of the number of satisfied clauses, denoted as new_satisfied count to new_satisfied_count.
f. If new_satisfied_count ~~is greater~~ flip a biased coin to decide whether to update satisfied_count.
g. If new_satisfied_count is less than satisfie_cnt, restore the original truth assignment for all variables
h. If the termination condition is met, exit all loop.
5. Output the final satisfied_count, representing the number of clauses satisfied by the modified procedure.
This modification guarantees that the expected number of clauses satisfied by the modified procedure is at least 0.6k. where k is the total number of clauses in the instance

(d) Give a randomized polynomial-time algorithm for the general MAX SAT Problem, so that the expected number of clauses satisfied by the algorithm is at least a 0.6 fraction of the maximum possible. (Note that, by the example in part (a), there are instances where one cannot satisfy more than $k/2$ clauses; the point here is that we'd still like an efficient algorithm that, in expectation, can satisfy a 0.6 fraction of the maximum that can be satisfied by an optimal assignment.)

start with a random truth assignment for all variables in the instance. Calculate the number of clauses satisfied by the initial assignment, denoted as satisfied_count. Repeat the following steps for a predefined number of iterations:

a) Randomly select a clause Ci from the instance.
b) Save the current truth assignment for all variables,
c) set the variables in Ci according for all variables
d) calculate the number of satisfied clauses for the new assignment,
e) If new_satisfied_count is equal to satisfied_count, flip a biased coin.
g) If new_satisfied_count is less than satisfied_count, restore the original truth assignment ∀ variables
4) Output the final satisfied count, representing the number of clauses satisfied by the algorithm.

The expected number of clauses satisfied by this algorithm is at least 0.6 times the maximum number of clauses that can be satisfied at most K/2 claues, the algorithm can still achieve an expected approximation ratio of at least 0.6 of the maximum possible satisfaction.

3. *Kleinberg, Jon. Algorithm Design (p. 789, q. 10).*

Consider a very simple online auction system that works as follows. There are n *bidding agents*; agent i has a bid b_i, which is a positive natural number. We will assume that all bids b_i are distinct from one another. The bidding agents appear in an order chosen uniformly at random, each proposes its bid b_i in turn, and at all times the system maintains a variable b^* equal to the highest bid seen so far. (Initially b^* is set to 0.) What is the expected number of times that b^* is updated when this process is executed, as a function of the parameters in the problem?

lets consider the case when there are only two bidding agents, agent 1 & Agent 2. We can calculate the expected number of updates of b^* in this scenario:

- Agent 1 will be b^* if its bid b_1 is higher than 0, which happens with a probability of 1.
- Agent 2 will update b^* if its bid b_2 is higher b^*, which with a probability of 1/2
- Thus in this case, the expected number of updates of b^* is $1+1/2 = 3/2$. Now consider the general case with n bidding agents. We observe the following:
- Agent 1 will always update b^* since its bid is the first one encountered, and it is always higher than 0. Agent 2 will update b^* if its bid b_2 is higher than the ~~current~~ b_1. probability of 1/2.
- Agent 3 will update b^* if its b_3 is higher than current b^*, probability of 1/3. Similarly, Agent i will update b^* if its bid b_i is higher than the current b^*, which happens with probability $1/i$.

$E[\text{number of updates}] = 1 + 1/2 + 1/3 + \dots + 1/n$.

$E[\text{number of updates}] \approx \ln(n) + \gamma$.

Therefore, the expected number of times that b^* is updated in the online auction system, as a function of the parameter n, is approximately $\ln(n) + \gamma$.

4. Recall that in an undirected and unweighted graph $G = (V, E)$, a cut is a partition of the vertices $(S, V\backslash S)$ (where $S \subseteq V$). The size of a cut is the number of edges which cross the cut (the number of edges (u, v) such that $u \in S$ and $v \in V\backslash S$). In the MAXCUT problem, we try to find the cut which has the largest value. (The decision version of MAXCUT is NP-complete, but we will not prove that here.) Give a randomized algorithm to find a cut which, in expectation, has value at least 1/2 of the maximum value cut.

1) Initialize the best cut as an empty cut ($S=\phi, V\backslash S = V$).
2) Repeat the steps for a predefined number of iterations or until a termination condition is met:
a. Randomly select a cut $(S, V\backslash S)$ by assigning each vertex independently to either S or $V\backslash S$ with probability $1/2$ each
b. calculate the value of the cut $(S, V\backslash S)$ by assigning each vertex independently no. of edges that cross the cut.
c. If the value of the current cut is greater than the value of the best cut found so far update the best cut to be the current cut. 3) Output the best cut found.
$E[\text{value of } c'] = Pr(c'=c) * \text{value}(c) + Pr(c' \neq c) * \text{value}(c')$.
$Pr(c'=c)$ represents the probability that the algorithm finds the maximum value cut, and $Pr(c' \neq c)$ represents the probability that the algorithm does not find the maximum value cut.
For the value of c', if it is not equal to c, the expected value of c' can be bounded by:
$E[\text{value of } c'] \leq Pr(c' \neq c) * \text{value}(c')$.
Since algorithm updates the best cut to be the current cut if it has a higher value, we have $\text{value}(c') \leq \text{value}(c)$, which implies:
$E[\text{value of } c'] \leq Pr(c' \neq c) * \text{value}(c)$.
combining the above equations, we get:
$E[\text{value of } c'] = Pr(c'=c) * \text{value}(c) + Pr(c' \neq c) * \text{value}(c')$
$\geq 1/2 * \text{value}(c) + Pr(c' \neq c) * \text{value}(c)$
$\geq 1/2 * \text{value}(c)$.
Therefore, the expected value of the cut found by the algorith is at least 1/2 of the maximum value cut, providing an expected approximation ratio of 1/2.