INFO 6205
Program Structure and Algorithms
Assignment 6

Student Name: Vidhi Bharat Patel
Professor: Nik Bear Brown

Q1 (20 Points) Suppose you're acting as a consultant for the Port Authority of a small Pacific Rim nation. They're currently doing a multi-billion-dollar business per year, and their revenue is constrained almost entirely by the rate at which they can unload ships that arrive in the port.

Here's a basic sort of problem they face. A ship arrives, with n containers of weight $w1, w2, \ldots, wn$. Standing on the dock is a set of trucks, each of which can hold K units of weight. (You can assume that K and each wi is an integer.) You can stack multiple containers in each truck, subject to the weight restriction of K; the goal is to minimize the number of trucks that are needed in order to carry all the containers. This problem is NP-complete (you don't have to prove this).

A greedy algorithm you might use for this is the following. Start with an empty truck, and begin piling containers 1, 2, 3, . . . into it until you get to a container that would overflow the weight limit. Now declare this truck "loaded" and send it off; then continue the process with a fresh truck. This algorithm, by considering trucks one at a time, may not achieve the most efficient way to pack the full set of containers into an available collection of trucks.

a) Give an example of a set of weights, and a value of K, where this algorithm does not use the minimum possible number of trucks.

b) Show, however, that the number of trucks used by this algorithm is within a factor of 2 of the minimum possible number, for any set of weights and any value of K.

Solution:

a) Consider that there are the weights of the containers as followed: -
w1 = 4, w2 = 4, w3 = 3, w4 = 2, w5 = 2, w6 = 1
The weight limit of the truck K = 6

By using the greedy algorithm, the trucks will be loaded as follows:
Truck 1: w1 = 4, w2 = 4
Truck 2: w3 = 3
Truck 3: w4 = 2, w5 = 2
Truck 4: w6 = 1

However, there is a better solution to load the containers into the truck, this can be given as:
Truck 1: w1 = 4, w6 = 1
Truck 2: w2 = 4, w5 = 2
Truck 3: w3 = 3, w4 = 2

This way, there are only three trucks used which basically makes the packing more efficient using the minimum possible number than the greedy algorithm.

b) We can prove that the number of trucks used by the greedy algorithm is within a factor of 2 of the minimum possible number using a proof by contradiction.

Suppose there exists a set of weights and a value of K for which the greedy algorithm uses more than twice the minimum number of trucks needed to transport all the containers. Let OPT be the minimum number of trucks needed to transport all the containers, and let Greedy be the number of trucks used by the greedy algorithm.
Then we have: Greedy > 2 * OPT

Let T1, T2, ..., T(Greedy) be the trucks used by the greedy algorithm, where Greedy is the number of trucks. For each truck Ti used by the greedy algorithm, let wi be the total weight of containers loaded onto Ti.
Note that wi <= K, since the greedy algorithm never overloads a truck. Since Greedy > 2 * OPT, there must exist at least one truck Tj such that: wj > K/2

Consider the set of containers loaded onto Tj, and let S be a subset of this set of containers such that the total weight of the containers in S is greater than K/2. We can form a new truck T' by taking all the containers in S and putting them onto T'.
Since the total weight of the containers in S is greater than K/2, we have: wi < K/2 for all trucks Ti except Tj.
Therefore, we can load all the remaining containers onto the remaining trucks, and we need at most (OPT - 1) additional trucks.

Thus, we have shown that we can transport all the containers using at most (OPT + 1) trucks, which contradicts the assumption that Greedy > 2 * OPT.
Therefore, the number of trucks used by the greedy algorithm is within a factor of 2 of the minimum possible number, for any set of weights and any value of K.

Q2 (20 Points) You are asked to consult for a business where clients bring in jobs each day for processing. Each job has a processing time *ti* that is known when the job arrives. The company has a set of ten machines, and each job can be processed on any of these ten machines.

At the moment the business is running the simple Greedy-Balance Algorithm we discussed in Section 11.1 of K&T. They have been told that this may not be the best approximation algorithm possible, and they are wondering if they should be afraid of bad performance. However, they are reluctant to change the scheduling as they really like the simplicity of the current algorithm: jobs can be assigned to machines as soon as they arrive, without having to defer the decision until later jobs arrive.

In particular, they have heard that this algorithm can produce solutions with makespan as much as twice the minimum possible; but their experience with the algorithm has been quite good: They have been

running it each day for the last month, and they have not observed it to produce a makespan more than 20 percent above the average load.

Solution:

We can prove that on the type of inputs the company sees, the Greedy-Balance Algorithm will always find a solution whose makespan is at most 20 percent above the average load.

Let OPT be the optimal makespan, and let m be the average load per machine (i.e., m = (1/10) * sum(ti) for all jobs i). We want to show that the Greedy-Balance Algorithm produces a solution with makespan at most 1.2m.

Consider the set of machines M1, M2, ..., M10, and let S be the set of jobs processed on each machine by the Greedy-Balance Algorithm. Let Mmax be the machine with the most processing time, i.e., Mmax is such that sum(ti) for all jobs i in S(Mmax) is greater than or equal to sum(ti) for all jobs i in S(Mi) for all i != Mmax. Let Tmax be the total processing time on machine Mmax, i.e., Tmax = sum(ti) for all jobs i in S(Mmax).

Then we have: Tmax <= 2m
since the Greedy-Balance Algorithm assigns each job to a machine that has at most twice the average load.
Let T be the total processing time for all jobs, i.e, T = sum(ti) for all jobs i. Then we have: T = sum(Ti) for all machines Mi
where Ti is the total processing time for jobs assigned to machine Mi by the Greedy-Balance Algorithm.
By definition of the average load, we have: T/10 <= m
Multiplying both sides by 2, we get:
T/5 <= 2m
Using the fact that Tmax <= 2m, we have: T/5 <= Tmax

Adding up the processing times of jobs assigned to all machines, we get:
T = sum(Ti) for all machines Mi
<= Tmax * 10
<= 2m * 10
= 20m

Therefore, we have:
T/5 <= Tmax <= 2m
T <= 20m
Dividing both sides by 10, we get:
T/10 <= 2m
which is the same as the average load per machine.
Thus, we have shown that Greedy – Balance Algorithm produces a solution with makespan at most 20 percent above the average load, on the type of inputs the company sees.

Q3 (20 Points) Consider the problem of finding a stable state in a Hopfield neural network, in the special case when all edge weights are positive. This corresponds to the Maximum-Cut Problem that we discussed earlier in the chapter: For every edge *e* in the graph *G,* the endpoints of *G* would prefer to have opposite slates.

Now suppose the underlying graph G is connected and bipartite; the nodes can be partitioned into sets *X* and *Y* so that each edge has one end in *X* and the other in *Y.* Then there is a natural "best" configuration for the Hopfield net, in which all nodes in *X* have the state +1 and all nodes in *Y* have the state –1. This way, all edges are *good,* in that their ends have opposite states.

The question is: In this special case, when the best configuration is so clear, will the State-Flipping Algorithm described in the text [as long as there is an unsatisfied node, choose one and flip its state) always find this configuration? Give a proof that it will, or an example of an input instance, a starting configuration, and an execution of the State-Flipping Algorithm that terminates at a configuration in which not all edges are good. is to minimize the overall cost $\sum_{s \in A} f_s + \sum_{u \in U} \min_s \in A d_{us}$. Give an $H(n)$ approximation for this problem.
(Note that if all service costs d_{us} are 0 or infinity, then this problem is exactly the Set Cover Problem: f_s is the cost of the set named *s,* and d_{us} is 0 if node *u* is in set *s,* and infinity otherwise.)

Solution:
In the special case when the underlying graph G is connected and bipartite, the State-Flipping Algorithm will always find the best configuration where all nodes in X have the state +1 and all nodes in Y have the state –1.

Suppose we start with an arbitrary configuration of the Hopfield net where not all edges have their endpoints with opposite states. Let's assume there is at least one unsatisfied node. Since the graph is bipartite, any unsatisfied node must have neighbors with the opposite state. Otherwise, if all of its neighbors have the same state, it would be satisfied. Therefore, flipping the state of an unsatisfied node will satisfy that node and all its neighbors, making them all satisfied. We repeat this process until all nodes are satisfied, and thus all edges have their endpoints with opposite states.

Now, let's suppose that we start with the configuration where all nodes in X have state -1 and all nodes in Y have state +1. We claim that this configuration is already stable, meaning there is no unsatisfied node. To see why, note that any node in X has only neighbors in Y, and vice versa. Since all edges have their endpoints with opposite states in the best configuration, any node in X has all its neighbors in Y with the opposite state. Thus, it is impossible for any node in X to be unsatisfied, and the same holds for any node in Y. Therefore, the State-Flipping Algorithm will terminate immediately without making any state flips, and the best configuration is already achieved. In conclusion, the State-Flipping Algorithm will always find the best configuration in the special case when the underlying graph G is connected and bipartite.

The Set Cover problem is NP-complete, so any polynomial-time algorithm for it cannot be guaranteed to give optimal solutions. Therefore, we seek an approximation algorithm that can guarantee a certain approximation ratio. Here, we will present a simple greedy algorithm that gives an H(n) approximation, where n is the number of elements in the universe.

Algorithm:

1. Let S be the empty set of selected sets.
2. While there exists an uncovered element:
3. Select the set Sj that covers the largest number of uncovered elements.
4. Add Sj to the set S.
5. Return the set S.

This algorithm repeatedly selects the set that covers the largest number of uncovered elements until all elements are covered. It is easy to see that this algorithm is a polynomial-time algorithm. Now, we will show that it guarantees an approximation ratio of H(n). Let OPT be the optimal cost of a set cover, and let the cost of the set cover produced by our algorithm be C. We will show that C ≤ H(n) · OPT.

Let m be the number of sets in the optimal solution, and let Xi be the number of elements covered by the ith set in the optimal solution. Since the greedy algorithm selects the set that covers the largest number of uncovered elements, we have Xj ≥ n/m for any j. Therefore, the cost of the solution produced by the greedy algorithm is:
C = Σ fs for all sets s in the solution
≤ Σ OPT · (Xi/n) for all sets in the optimal solution (because the optimal solution covers all elements)
= OPT · Σ (Xi/n) for all sets in the optimal solution
= OPT · Σ (1/(ni/m)) for all sets in the optimal solution (because Xi/n ≤ 1)
= OPT · Σ (m/ni) for all sets in the optimal solution
≤ OPT · Σ (m/1) for all elements in the universe (because each element is covered by at least one set)
= m · OPT
≤ H(n) · OPT

The third inequality follows from the fact that the harmonic mean of a set of numbers is always less than or equal to the arithmetic mean of the same set of numbers. The last inequality follows from the fact that m ≤ n (because each set in the optimal solution covers at least one element). Therefore, our algorithm is an H(n) approximation algorithm for the weighted set cover problem.

Q4 (20 Points) Let $G = (V, E)$ be an undirected graph with n nodes and m edges. For a subset $X \subseteq V$, we use $G[X]$ to denote the subgraph *induced* on X —that is, the graph whose node set is X and whose edge set consists of all edges of G for which both ends lie in X.

We are given a natural number $k \leq n$ and are interested in finding a set of k nodes that induces a "dense" subgraph of G; well phrase this concretely as follows. Give a polynomial-time algorithm that produces, for a given natural number $k \leq n$ a set $X \subseteq V$ of k nodes with the property that the induced subgraph $G[X]$ has at least $\frac{mk(k-1)}{n(n-1)}$ edges.

You may give either (a) a deterministic algorithm, or (b) a randomized algorithm that has an expected running time that is polynomial, and that only outputs correct answers.

Solution:

To solve this problem, we can use a randomized algorithm that is similar to the one used for the Max-Cut problem. The algorithm proceeds as follows:

1. Choose a random subset X of k nodes from V.
2. Calculate the number of edges in G[X].
3. Repeat steps 1 and 2 T times, where T is a constant to be determined later.
4. Output the subset X that had the most edges in G[X] over all the T iterations.

Let's analyze the expected number of edges in G[X] for a given iteration. We can assume without loss of generality that the nodes in V are labeled 1, 2, ..., n. Let e(i,j) be the indicator variable that takes value 1 if there is an edge between nodes i and j, and 0 otherwise. Then the expected number of edges in G[X] is given by:

E[|G[X]|] = E[Σe(i,j)] = Σ E[e(i,j)]
= Σ P(i,j in G[X])
= Σ P(i in X)P(j in X | i in X)
= Σ k/n * (k-1)/(n-1)
= k(k-1)/n(n-1) * n
= k(k-1)/n-choose-2

where P(i,j in G[X]) is the probability that both nodes i and j are in X and hence there is an edge between them in G[X].

Using linearity of expectation, the expected number of edges in the best subset X over T iterations is:

E[|G[X*]|] >= (1/T) Σ E[|G[X]|]
>= k(k-1)/(n-choose-2) * (1/T) Σ (k/n * (k-1)/(n-1))
= k(k-1)/(n-choose-2)

where X* is the best subset over all T iterations.

Therefore, if we set T = n^2, then the probability that the output of the algorithm has less than n(n-1) edges in G[X] is at most 1/n. Hence, the expected running time of the algorithm is polynomial, and the output is correct with high probability.

Algorithm:

```
RANDOM-DENSE-SUBGRAPH(G, k):
  X_best = {}
  E_best = 0
  T = n^2
  for i = 1 to T:
    X = random subset of k nodes from V
    E = number of edges in G[X]
    if E > E_best:
      X_best = X
      E_best = E
  return X_best
```

Q5 (20 Points) Consider a balls-and-bins experiment with 2*n* balls but only two bins. As usual, each ball independently selects one of the two bins, both bins equally likely. The expected number of balls in each bin is *n.* In this problem, we explore the question of how big their difference is likely to be. Let X_1 and X_2

denote the number of balls in the two bins, respectively. (X_1 and X_2 are random variables.) Prove that for any $\varepsilon > 0$ there is a constant $c > 0$ such that the probability $\Pr[X_1 - X_2 > c n] \leq \varepsilon$.

Solution:
We can start by using the Chernoff bound. Let p be the probability of a ball going into bin 1 (and hence, 1-p is the probability of it going into bin 2). Since each ball goes into either bin with equal probability, we have p = 1/2.

Then, the expected value of X1 is E[X1] = np = n/2, and the expected value of X2 is E[X2] = n(1-p) = n/2 as well. The difference between X1 and X2 is given by Y = X1 - X2, and so E[Y] = E[X1] - E[X2] = 0.

We can now use the Chernoff bound to bound the probability of Y deviating from its expected value by a certain amount. Let δ be a positive constant, and let t = c n for some constant c > 0. Then we have:
Pr[Y > t] = Pr[X1 - X2 > t]
= Pr[e^(s(Y - E[Y])) > e^(st)]
<= E[e^(s(Y - E[Y]))]/e^(st)
= E[e^(sY)]/e^(st)
= E[e^(s(X1 - X2))]/e^(st)
= E[e^(sX1)]*E[e^(-sX2)]/e^(st)
= (e^sp)^n * (e^s(1-p))^n / e^(st)
= e^(ns(p - 1/2)s) / e^(st)
= e^(-n(1/2 - p)s + st)
where we have used the fact that X1 and X2 are independent, and that they have the same distribution (since balls are equally likely to go into either bin).

Now, we want to choose s such that the exponent in the last expression is negative. That is, we want:
-n(1/2 - p)s + st < 0
s(n/2 - p) < t
s < t/(n/2 - p)

We can choose s = ε ln(1/δ), where ε > 0 is some constant to be determined later. Then, we have:

s < t/(n/2 - p)
ε ln(1/δ) < c n / (n/2 - 1/2)
ε ln(1/δ) < 2c
ln(1/δ) < 2c/ε
δ > e^(-2c/ε)

So if we choose c = ε ln(1/e), then we have:
δ > e^(-2c/ε) = e^(-2 ln(1/e)) = e^2

Thus, we have shown that for any ε > 0, there exists a constant c > 0 such that Pr[X1 - X2 > c n] ≤ e^(-2ε^2 n), using the Chernoff bound with s = ε ln(1/δ) and choosing c = ε ln(1/e).