INFO 6205 – Program Structures and Algorithms
Assignment 6

Student Name: Yuetong Guo
Professor: Nik Bear Brown

Q1 (20 Points)

Ans:

A:

Let's consider a case with 4 containers with weights w1 = 4, w2 = 4, w3 = 2, and w4 = 2, and each truck can carry a maximum weight of K = 6.

Using the greedy algorithm:

- First truck: Add container 1 (weight 4).
  Adding container 2 would overflow the weight limit (4 + 4 > 6)
  So we stop and send the truck off.
- Second truck: Add container 2 (weight 4).
  We could add container 3 (weight 2), but not able to add container 4 (weight 2), which would overflow the weight limit (4 + 2 + 2 > 6), so we stop and send the truck off.
- Third truck: Add container 4 (weight 2)

The greedy algorithm uses 2 trucks in this example.

However, we could have loaded the containers more efficiently using only 2 trucks:

- Truck 1: Add containers 1 (weight 4) and 3 (weight 2).
- Truck 2: Add containers 2 (weight 4) and 4 (weight 2).

B:

Let OPT be the optimal (minimum) number of trucks required, and let GREEDY be the number of trucks used by the greedy algorithm.

The total weight of all containers is W = w1 + w2 + ... + wn. In the optimal solution, each truck carries at most K weight. So, OPT * K >= W. This means that OPT >= W / K.

The greedy algorithm fills each truck to at least half its capacity (except possibly the last one). Otherwise, we could have added another container to the truck without exceeding the weight limit, contradicting the greedy algorithm's condition. So, each of the (GREEDY - 1) trucks carries at least K / 2 weight.

Therefore, (GREEDY - 1) * (K / 2) <= W.

Combining the two inequalities above, we have (GREEDY - 1) * (K / 2) <= W <= OPT * K. Divide both sides by K / 2, we get (GREEDY - 1) <= 2 * OPT.

Rearranging the inequality, we have GREEDY <= 2 * OPT + 1.

Thus, the greedy algorithm uses at most 2 times the minimum number of trucks, plus 1 extra truck. In practice, this extra truck is negligible when the number of containers is large, so the greedy algorithm provides a reasonable approximation for this problem.

Q2 (20 Points)

Ans:

To prove that the Greedy-Balance Algorithm will always find a solution whose makespan is at most 20 percent above the average load for the given constraints, let's first state the average load:

$$\text{The average load} = (1/10) * \sum_0^i ti$$

Since there are 10 machines, let M be the machine that has the largest load, or makespan, after running the Greedy-Balance Algorithm.

Let's first consider the worst-case scenario. Suppose a job with processing time tmax = 50 arrives, and the Greedy-Balance Algorithm assigns it to machine M. Since the minimum total load is 3000, the average load is at least 300 (3000/10). Before the arrival of the job with processing time 50, machine M had a load of L. After adding the job, the new load becomes L' = L + tmax.

Now we need to show that the makespan L' is at most 20% larger than the average load:

$$L' \leq (1 + 0.2) * (1/10) * \sum_0^i ti$$

Substitute L' with the expression we found:

$$L + \text{tmax} \leq (1 + 0.2) * (1/10) * \sum_0^i ti$$

Notice that before the arrival of the last job with processing time 50, the load of the machine M was at most the average load:

$$L \leq (1/10) * \sum_0^i ti$$

Now, we will use this inequality to show that the Greedy-Balance Algorithm meets the condition:

$$L + \text{tmax} \leq (1 + 0.2) * (1/10) * \sum_0^i ti$$

$$L + 50 \leq (1.2) * (1/10) * \sum_0^i ti$$

Recall that $L \leq (1/10) * \sum_0^i ti$:

$$(1/10) * \sum_0^i ti + 50 \leq (1.2) * (1/10) * \sum_0^i ti$$

Now divide both sides by $(1/10) * \sum_0^i ti$:

$$1 + 50/((1/10) * \sum_0^i ti) \leq 1.2$$

Since the minimum total load $\sum_0^i ti$is 3000:

$$1 + 50/(300) \leq 1.2$$

$$1 + 1/6 \leq 1.2$$

This inequality holds true, so the Greedy-Balance Algorithm will always find a solution whose makespan is at most 20 percent above the average load for the given input constraints.

Q3 (20 Points)

Ans:

The State-Flipping Algorithm does not always find the "best" configuration in which all edges are good when the underlying graph G is connected and bipartite.

To illustrate this, let's consider the given example:

Let C be a graph consisting of a cycle of length four, with nodes v1, v2, v3, and v4, and edges (v1, v2), (v2, v3), (v3, v4), and (v4, v1). The graph is connected and bipartite, with partition sets X = {v1, v3} and Y = {v2, v4}.

If we start the State-Flipping Algorithm in a configuration where nodes v1 and v2 have state +1, and nodes v3 and v4 have state -1, then no improving move is possible. The initial configuration has the following states:

v1: +1

v2: +1

v3: -1

v4: -1

In this configuration, the edges (v1, v2) and (v3, v4) are not good because their endpoints have the same state. However, flipping the state of any single node will result in two other edges becoming not good while only improving one edge. For example, flipping the state of v1:

v1: -1

v2: +1

v3: -1

v4: -1

Now, edges (v1, v4) and (v1, v2) are not good. As a result, the State-Flipping Algorithm can never reach the "best" configuration in which all nodes in X have the state +1, and all nodes in Y have the state -1, because every single move creates new not good edges while fixing others.

Q4 (20 Points)

Ans:

Base on the given idea of professor's solution to this question, here is a complete randomized algorithm :

- Choose k nodes uniformly at random from G.
- For each pair of nodes i and j, calculate the probability P(i, j) that there is an edge between the i th and j th node.
- Compute the expected number of edges E(X) for the induced subgraph G[X] using the probabilities calculated in step 2.
- If the number of edges in G[X] is at least (mk(k-1))/(n*(n-1)), output X;
  otherwise, repeat from step 1.

To analyze the algorithm, let's define a random variable X_ij = 1 if there is an edge between nodes i and j in the subgraph G[X], and 0 otherwise. Since the subgraph is induced by k nodes, there are k(k-1)/2 pairs of nodes to consider. The expected number of edges E(X) in the subgraph G[X] can be calculated as:

$$E(X) = \sum\sum P(i, j) \text{ for all } i < j$$

It representing the expected number of edges in the induced subgraph G[X]. Here, the double summation is used to consider all pairs of distinct nodes i and j in the subgraph G[X]. The "for all i < j" condition ensures that we only consider each pair of nodes once (as i is always less than j).

As mentioned in the example solution, the probability that one iteration of this algorithm succeeds is p+. The overall running time will be the (polynomial) time for one iteration, multiplied by 1/p+. Since the algorithm keeps repeating until it finds a dense subgraph with the desired number of edges, the output will always be correct.

This randomized algorithm has an expected polynomial running time and only outputs correct answers, satisfying the requirements of the problem.

Q5 (5 Points)

Ans:

Let X1 and X2 denote the number of balls in the two bins, respectively. Let X = X1 - X2 be the difference between the number of balls in the two bins. We wish to find a constant c > 0 such that the probability Pr [X > c * (n^(1/2))] <= ε for any ε > 0.

Recall that the binomial distribution for 2n balls being placed into the two bins has a mean of n and a variance of n/2 for each bin. The central limit theorem states that, as the number of trials grows large, the distribution of the difference between the number of balls in the bins (X) will approach a normal distribution with a mean of 0 and a variance of n.

Now, let Y = (X1 - X2)/sqrt(n) be the normalized difference between the number of balls in the two bins. Then Y has a mean of 0 and a variance of 1.

We are interested in the probability Pr [Y > c]. Using the tail bounds for the normal distribution, we have:

$$\Pr[Y > c] <= e^{(-c^2/2)} \quad (1)$$

Since our goal is to find a constant c > 0 such that Pr [X > c * (n^(1/2))] <= ε, we can rewrite the probability in terms of X:

Pr[Y > c] = Pr[X > c * (n^(1/2))] <= ε

Now, substituting this expression for the probability in equation (1), we get:

e^(-c^2/2) <= ε

Taking the natural logarithm of both sides:

-ln(ε) >= c^2/2

Solving for c:

c >= sqrt(-2 * ln(ε))

Thus, for any ε > 0, there exists a constant c > 0 such that the probability Pr[X1 - X2 > c * (n^(1/2))] <= ε, where c >= sqrt(-2 * ln(ε)).