Assignment-3

1) a) The Greedy-Balance algorithm ensures that the makespan T is at most twice the optimal makespan T^*:

1. Greedy Assignment:
   - At each step, the job is assigned to the machine with the current minimum load.
2. Maximum load Before Last Job:
   - Before assigning the last job t_j, the load on the selected machine M_i was at most $T - t_j$.
3. Bounding the makespan:
   - After assigning the last job:
     $T \leq T^* + t_{max}$
   - The worst case is when $t_{max} \leq T^*$:
     $T \leq 2T^*$

Thus, $T \leq 2T^*$.

b)

Given jobs [12, 13, 14, 16, 12, 12] and m = 3 machines:

1. Initial loads:
   - $T_1 = 0, T_2 = 0, T_3 = 0$
2. Assign jobs:
   - Job 1 (12): $M_1 \rightarrow T_1 = 12$
   - Job 2 (13): $M_2 \rightarrow T_2 = 13$
   - Job 3 (14): $M_3 \rightarrow T_3 = 14$
   - Job 4 (16): $M_1 \rightarrow T_1 = 28$
   - Job 5 (12): $M_2 \rightarrow T_2 = 25$
   - Job 6 (12): $M_3 \rightarrow T_3 = 26$
3. Final loads:
   - $T_1 = 28, T_2 = 25, T_3 = 26$
4. Makespan:
   - The maximum load is $T = \max(28, 25, 26) = 28$.

So, the resulting makespan is 28.

2)

Case 1: Single Center C_1

1. Best Location:
   - Mid point of 0 and 20:

$$c_1 = \left(\frac{x_3 + x_2}{2}, \frac{y_3 + y_2}{2}\right)$$

2. Optimal Covering Radius $r(c_1)$:

$$r(c_1) = d/2$$

Where d is the distance between 3 and 2.

Case 2: Two Centers c_1 and c_2

1. Best locations:
   - At 3 and 2:
     $c_1 = 3$ and $c_2 = 2$
2. Optimal Covering Radius $r(c_2)$:

$$r(c_2) = 0$$

3)

Problem Instance
- Universe U: {1, 2, 3, 4, 5, 6, 7, 8}
- Subsets and weights:
  - $S_1 = \{1, 3, 5, 7\}$, $w_1 = 1.1\epsilon$
  - $S_2 = \{2, 4, 6, 8\}$, $w_2 = 1.1\epsilon$
  - $S_3 = \{1\}$, $w_3 = 1$
  - $S_4 = \{2\}$, $w_4 = 1$
  - $S_5 = \{3, 4\}$, $w_5 = 1$
  - $S_6 = \{5, 6, 7, 8\}$, $w_6 = 1$

Greedy-Set-Cover Steps
1. Initialization:
   - R = {1, 2, 3, 4, 5, 6, 7, 8}
   - S = $\emptyset$
2. First Iteration:
   - Select S_6 (min 1/4)
   - S = {S_6}, R = {1, 2, 3, 4}
3. Second Iteration:
   - Select S_5 (min 1/2)
   - S = {S_6, S_5}, R = {1, 2}
4. Third Iteration:
   - Select S_1 (min $1.1\epsilon/2$)
   - S = {S_6, S_5, S_1}, R = {2}

5. Fourth Iteration:
   - Select S_4 (min 1)
   - $O = \{S_6, S_5, S_1, S_4\}$, $R = \emptyset$

Greedy Solution
- $O = \{S_6, S_5, S_1, S_4\}$
- Total weight: $1+1+(1+\epsilon)+1 = 4+\epsilon$

Optimal Solution
- $O^* = \{S_6, S_5, S_3, S_4\}$
- Total weight: 4

Comparison
- Greedy solution weight: $4+\epsilon$
- Optimal solution weight: 4

The greedy solution is very close to the optimal solution, differing by a small ϵ.

4)

a) Graph and Vertex weights
- Vertices: $V = \{1,2,3,4\}$
- Weights: $w(1) = 6$, $w(2) = 5$, $w(3) = 7$, $w(4) = 5$
- Edges: $E = \{(1,2), (1,3), (2,4), (3,4)\}$

Algorithm Steps
1. Initialization:
   - $P_e = 0$ for all $e \in E$.
2. First Iteration:
   - Select (1,2): Increase $P(1,2)$ to 2.5 (Vertex 2 tight).
   - $S = \{2\}$
3. Second Iteration:
   - Select (1,3): Increase $P(1,3)$ to 3.5 (Vertex 1 tight).
   - $S = \{1,2\}$
4. Third Iteration:
   - Select (2,4) and (3,4): Increase $P(2,4)$, $P(3,4)$ to 2.5 (Vertex 4 tight).
   - $S = \{1,2,4\}$

Greedy Solution
- Selected Set: $S = \{1,2,4\}$
- Total weight: $w(1) + w(2) + w(4) = 6+5+5 = 16$

Optimal Solution

- Optimal Set: $S^* = \{2, 3\}$.
- Total weight: $w(2) + w(3) = 5 + 7 = 12$

b)

For any vertex cover S and any nonnegative and fair prices p_e:

$$\sum_{e \in E} p_e \leq w(S)$$

Proof:-

- fair Prices: p_e are increased without exceeding the vertex weights.
- Vertex Cover: S covers all edges.
- Sum of Prices: The total price of edges incident to vertices in S is less than or equal to the sum of vertex weights:

$$\sum_{e \in E} p_e \leq \sum_{v \in S} w(v) = w(S)$$

c)

The set S returned by the algorithm is a vertex cover, and its cost is at most twice the minimum cost of any vertex cover.

Proof:-

1. Vertex Cover: S covers all edges since the algorithm stops when all edges have a tight endpoint.
2. Cost Bound: for any edge e, the price p_e is bounded by 2.

$$\sum_{e \in E} p_e \leq w(S^*)$$

3. Approximation Ratio: The total cost of S:

$$w(S) \leq 2 \sum_{e \in E} p_e \leq 2w(S^*)$$

Thus, the cost of S is at most twice the cost of the optimal vertex cover.

)

- Nodes: $V = \{a, b, c, d, e, f\}$
- Edges: $E = \{(a,b), (a,e), (a,f), (b,f), (b,c), (c,d), (e,d), (f,d)\}$
- Edge weights: $w = \{-1, 4, 6, -5, 1, 2, -3, 9\}$

- Node Assignments:
  - $S_a = +1$
  - $S_b = +1$
  - $S_c = +1$
  - $S_d = +1$
  - $S_e = -1$
  - $S_f = -1$

Stability Check:

Check if S_v matches the sign of the weighted sum of incident edges for each node.

1. Node a:
   - $S_a = +1$
   - Sum: $-1 - 4 - 6 = -11$
   - Not Stable
2. Node b:
   - $S_b = +1$
   - Sum: $-1 + 5 + 1 = 3$
   - Stable
3. Node c:
   - $S_c = -1$
   - Sum: $1 + 2 = 3$
   - Not Stable
4. Node d:
   - $S_d = +1$
   - Sum: $-2 + 3 - 9 = -8$
   - Not Stable
5. Node e:
   - $S_e = +1$
   - Sum: $4 - 3 = 1$
   - Not Stable

Node f:

- $S_f = -1$
- Sum: $6 - 5 + 9 = 10$
- Not Stable

Flip the states of unstable nodes:

1. flip S_a to -1
   flip S_c to $+1$
   flip S_d to -1
   flip S_e to $+1$
   flip S_f to $+1$

New Configuration

$S_a = +1$

$S_b = +1$

$S_c = +1$

$S_d = -1$

$S_e = +1$

$S_f = +1$

Recalculate the sums for the new configuration to ensure stability:

Node a:
Sum: 9 (Stable)

Node b:
Sum: -3 (Stable)

Node c:
Sum = -1 (Stable)

Node d:
Sum = 8 (Stable)

Node e:
Sum = (-1) (Stable)

Node f:
Sum = -20 (Stable)

The configuration is now stable.

a)

In a stable configuration of a hopfield neural network, there can't be any bad edges. A bad edge occurs when the weight of the edge multiplied by the nodes assignments results in a negative value. For example, if $S_u = +1$ and $S_v = -1$, and the weight of the edge $e = (u, v)$ is -3, then $w_e \cdot S_u \cdot S_v = 3$, which is positive. Therefore, there are no bad edges in this case.

b)

In a Hopfield neural network, a non-zero weight on an edge $e = (u, v)$ doesn't guarantee that nodes u and v will always be in the same state. For example, if $w_e = -2$ and initially $S_u = +1$ and $S_v = +1$, their states may not remain the same due to the influence of the negative weight, which encourages state changes to reduce energy. This illustrates that configurations may not always adhere to constraints imposed by edge weights and initial states.

8)
In the given complete bipartite graph, selecting all vertices from one partition forms a minimum vertex cover. While the gradient descent local search algorithm efficiently finds local minima by iteratively reducing the objective function (here, minimizing vertex cover size), it doesn't guarantee finding the global optimum. It may get stuck in local minima or plateaus, potentially missing equally optimal solutions.

9)
In the given graph, the gradient descent local search algorithm starts with an initial vertex cover $C=\{a,b\}$. By iteratively adding vertices that cover uncovered edges, it finds the final vertex cover $C=\{a,b,c,d\}$. Possible local solutions include $\{a,b\}$, $\{a,b,c\}$, $\{a,b,d\}$ and $\{a,b,c,d\}$.
However, gradient descent doesn't guarantee the global optimum, as it may get stuck in local minima or plateaus, potentially missing equally optimal solutions.

10)
a) If H is 2-universal because any pair of distinct keys has an equally likely hash output.

b) The given hash function $h_a(x)$ over $\mathbb{Z}_p$ isn't 2-universal because distinct keys may have unequal hash probabilities.

c) Modifying the function to $h'_{a,b}(x)$ by adding an offset makes H' 2-universal.

11)
a)
False. The statement is false. The algorithm returns the kth largest element only if the size of the set S is k-1. Otherwise, it continues the recursion based on whether k-1 is greater than or equal to K or less than k-1.

b) If the median is always chosen as the splitter,

the size of S^- and S^+ would be approximately $n/2$ each. Let $T(n)$ be the running time for the select() function without the recursive call. In each step, the size of the set reduces by half. So, we can express the running time as:

$$T(n) = T(n/2) + O(n)$$

This is a recurrence relation for the time complexity of the algorithm. By the master theorem, the time complexity would be $O(n) \log n$.