PROBLEM 1

(i) Prove directly from the primal-dual solution to the Hitchcock problem (trans-shipment problem) that only an empty arc can become inadmissible after modification of the dual variables. Construct an example in which an arc does in fact become inadmissible. (10pts)

i)

To prove that only an empty arc can become inadmissible after modification of the dual variables in the primal-dual solution of the Hitchcock transportation problem, we'll first define the problem and the primal-dual method.

- The Hitchcock transportation problem (trans-shipment problem) is a classical optimization problem that involves minimizing the cost of shipping goods from multiple origins to multiple destinations while satisfying supply and demand constraints. The problem can be formulated as a linear programming problem with primal variables representing the flow on each arc (origin-destination pair) and dual variables associated with supply and demand constraints.
- The primal-dual method is an iterative technique used to solve linear programming problems. It involves updating the primal variables (flows) and dual variables (shadow prices) in each iteration to approach the optimal solution. The primal variables are updated by finding a feasible direction that reduces the objective function, while the dual variables are updated to maintain complementary slackness and feasibility of the dual problem.

Now, let's prove that only an empty arc can become inadmissible after modification of the dual variables.

Let (i, j) be an arc in the transportation network, and let x_ij be the flow on this arc, c_ij be the cost per unit flow, and u_i and v_j be the dual variables associated with supply at node i and demand at node j, respectively. An arc (i, j) is admissible if and only if:

$c_{ij} - u_i - v_j = 0$.

Let's consider modifying the dual variables u_i and v_j. Without loss of generality, suppose that u_i increases by some amount Δu_i and v_j decreases by the same amount Δu_i to maintain feasibility of the dual problem:

$u_i' = u_i + \Delta u_i$, $v_j' = v_j - \Delta u_i$.

The new reduced cost for arc (i, j) is:

$c_{ij} - u_i' - v_j' = c_{ij} - (u_i + \Delta u_i) - (v_j - \Delta u_i) = c_{ij} - u_i - v_j + 2\Delta u_i$.

Now, we have two cases:

- If the arc (i, j) is empty ($x_{ij} = 0$), then the reduced cost may become positive, i.e., inadmissible, after modifying the dual variables.

- If the arc (i, j) is non-empty (x_ij > 0), then the reduced cost remains non-positive, i.e., admissible, since the primal-dual method ensures that non-empty arcs have non-positive reduced costs to maintain complementary slackness.

Thus, only an empty arc can become inadmissible after modification of the dual variables.

EXAMPLE:

Consider a simple transportation problem with two origins (O1, O2), two destinations (D1, D2), and unit shipping costs as follows:

O1 -> D1: 1 O1 -> D2: 2 O2 -> D1: 3 O2 -> D2: 4

Suppose the initial dual variables are u_O1 = 0, u_O2 = 0, v_D1 = 1, and v_D2 = 2. Then, the reduced costs for all arcs are non-positive, and the solution is admissible. Now, let's increase u_O1 by 1 and decrease v_D1 by 1:

u_O1' = 1, v_D1' = 0.

The new reduced costs after updating the dual variables are:

O1 -> D1: 1 - 1 - 0 = 0 (admissible) O1 -> D2: 2 - 1 - 2 = -1 (admissible) O2 -> D1: 3 - 0 - 0 = 3 (inadmissible) O2 -> D2: 4 - 0 - 2 = 2 (admissible)

In this example, the arc (O2 -> D1) becomes inadmissible after modifying the dual variables. Notice that this arc was empty (i.e., had a flow of 0) before the dual variables were updated, which is consistent with the proof that only an empty arc can become inadmissible after modification of the dual variables.

(ii) Run the algorithms on the following problem: There are 4 warehouses (indexed with i) with supply (3, 4, 2, 1) and 4 clients (indexed with j) with requirements (3, 2, 3, 2). The costs are cij = 2i−3j + 100. (10pts)

ii)

AUCTION METHOD

To solve this combinatorial optimization problem, we need to find the optimal assignment of warehouses to clients that minimizes the total cost. This is a transportation problem that can be solved using the Hungarian algorithm.

We can represent the problem as a cost matrix where each row corresponds to a warehouse and each column corresponds to a client. The element in row i and column j represents the cost of assigning warehouse i to client j. The cost matrix is:

C = [99, 96, 93, 90] 3 [101, 98, 95, 92] 4 [103, 100, 97, 94] 2 [105, 102, 99, 96] 1 3 2 3 2

We can apply the Hungarian algorithm to find the optimal assignment. The algorithm proceeds as follows:

- Subtract the smallest cost in each row from all the elements in that row.

- Subtract the smallest cost in each column from all the elements in that column.
- Cover the matrix with minimum number of lines (either horizontal or vertical) so that all zeros are covered. If the number of lines equals the size of the matrix, we have found the optimal assignment and can stop.
- If the number of lines is less than the size of the matrix, find the smallest uncovered element and subtract it from all uncovered elements. Add it to all elements covered by two lines.
- Repeat steps 3-4 until the optimal assignment is found.

But I prefer to use the AUCTION METHOD due to its interpretability

AUCTION METHOD

C =
[99, 96, 93, 90] 3
[101, 98, 95, 92] 4
[103, 100, 97, 94] 2
[105, 102, 99, 96] 1
3 2 3 2

Pick the smallest (90) and assign the warehouse to the client. Warehouse 1 --> 2 units for client 4

C =
[99, 96, 93, --] 1
[101, 98, 95, --] 4
[103, 100, 97, --] 2
[105, 102, 99, --] 1
3 2 3 0

Pick the smallest (93) and assign the warehouse to the client Warehouse 1 --> 1 units for client 3

C =
[--, --, --, --] 0
[101, 98, 95, --] 4
[103, 100, 97, --] 2
[105, 102, 99, --] 1
3 2 2 0

Pick the smallest (95) and assign the warehouse to the client Warehouse 2 --> 2 units for client 3

C =
[--, --, --, --] 0
[101, 98, --, --] 2
[103, 100, --, --] 2

[105, 102, --, --] 1
3 2 0 0

Pick the smallest (98) and assign the warehouse to the client Warehouse 2 --> 2 units for client 2

C =
[--, --, --, --] 0
[--, --, --, --] 0
[103, --, --, --] 2
[105, --, --, --] 1
3 0 0 0

Pick the smallest (103) and assign the warehouse to the client Warehouse 3 --> 2 units for client 1

C =
[--, --, --, --] 0
[--, --, --, --] 0
[--, --, --, --] 0
[105, --, --, --] 1
1 0 0 0

Pick the smallest (105) and assign the warehouse to the client Warehouse 4 --> 1 units for client 1

C =
[--, --, --, --] 0
[--, --, --, --] 0
[--, --, --, --] 0
[--, --, --, --] 0
0 0 0 0

Once clients requirements are met, we stop. In this problem, as total supply is the same as total demand, once we reached to suply the demand, we also consume all of the supply, that is why there is not any leftover unit in the warehouses.

Therefore, the optimal assignment that minimizes the total cost is:

Warehouse 4 --> 1 units for client 1
Warehouse 3 --> 2 units for client 1
Warehouse 2 --> 2 units for client 2
Warehouse 2 --> 2 units for client 3
Warehouse 1 --> 1 units for client 3
Warehouse 1 --> 2 units for client 4

Therefore, the clients will be provided exactly (3, 2, 3, 2) units

C =
[99, 96, 93, 90] 3

[101, 98, 95, 92] 4
[103, 100, 97, 94] 2
[105, 102, 99, 96] 1
3 2 3 2

C =
[--, --, 1, 2] 3
[--, 2, 2, --] 4
[2, --, --, --] 2
[1, --, --, --] 1
3 2 3 2

And the total cost of this assignment is 1 * 93 + 2 * 90 + 2 * 98 + 2 * 95 + 2 * 103 + 1 * 105 = 970

LINEAR PROGRAM

Finally, as a second solution, we will run the algorithm for the following transhipment problem, approaching it using linear programming. The result should be the same.

LINEAR PROGRAM

min sum(c_ij x_ij)

s.t.
sumj from i = 1 .... to i = n sumi from j = 1 .... to i = m x_ij => 0

DUAL LINEAR PROGRAM

max sum(alpha_i a_i) + sum(beta_j b_j)

s.t.
alpha_i + beta_j <= c_ij
alpha_i, beta_j != 0

Ther, using the EXCEL SOLVER tool...

Cost = c_ij = 2i - 3j + 100

C = [99, 96, 93, 90] 3 [101, 98, 95, 92] 4 WAREHOUSE SUPPLIES [103, 100, 97, 94] 2 [105, 102, 99, 96] 1 3 2 3 2

CLIENTS REQUIREMENTS

FINAL DECISIONS

[--, --, 1, 2] 3
[--, 2, 2, --] 4
[2, --, --, --] 2
[1, --, --, --] 1
3 2 3 2

Both, for the LINEAR PROBLEM and the DUAL LINEAR PROBLEM, the objective function optimized value is: 970

PROBLEM 2

2. Show that in a bipartite graph the maximum matching size is the same as the minimum cover size. A cover is a set S of vertices where every edge in the graph is incident onto a vertex in S. (15pts)

SEE THE DEMONSTRATION SCANNED IN PAPER

NUMERICAL EXAMPLE

Let's consider a simple bipartite graph G with vertex sets A and B as follows:

A: {a1, a2, a3} B: {b1, b2, b3}

And let the edges in the graph be:

E: {(a1, b1), (a1, b2), (a2, b2), (a2, b3), (a3, b3)}

Now, let's find a maximum matching and a minimum vertex cover for G.

Maximum matching: M = {(a1, b1), (a2, b2), (a3, b3)}
In this case, we can see that every vertex is matched, so the size of the maximum matching |M| = 3.

Minimum vertex cover: X = {a1, a2, a3} or X = {b1, b2, b3}
We can see that by choosing all vertices from either set A or set B, we cover all edges in the graph. No smaller subset of vertices can cover all edges. Thus, the size of the minimum vertex cover |X| = 3.

In this example, we see that the maximum matching size (|M| = 3) is the same as the minimum vertex cover size (|X| = 3), which is consistent with the result we proved using Konig's theorem.

PROBLEM 3

a) Detail the proof of correctness (in particular show why compressing the blossom is correct). Analyze the complexity of the matching algorithm on general graphs. (10pts)

The Blossom Algorithm, introduced by Jack Edmonds in 1965, is an algorithm for finding maximum matchings in undirected graphs. A matching in a graph is a set of edges such that no two edges share a vertex. A maximum matching is a matching that contains the largest possible number of edges.

The core concept in the Blossom Algorithm is the identification and compression of blossoms. A blossom is a cycle in the graph with an odd number of vertices, where there exists an unmatched vertex that has an even level in the alternating level structure (also called the "base" of the blossom). The algorithm compresses the blossom into a single

"pseudo-vertex" to simplify the graph structure and then proceeds to search for augmenting paths in the compressed graph.

Here's an outline of the proof of correctness for the Blossom Algorithm, specifically focusing on why compressing the blossom is correct:

Invariant: Let G be the original graph, and let G' be the graph with the blossom B compressed into a single vertex. An alternating path P in G' corresponds to an alternating path P' in G such that:
a. If P does not use the pseudo-vertex in G', then P' = P.
b. If P uses the pseudo-vertex in G', then P' is obtained by replacing the pseudo-vertex with the corresponding path in the blossom B.

Correctness: If there's an augmenting path P in G', then there's an augmenting path P' in G, preserving the invariant.
a. If P does not use the pseudo-vertex in G', then P' = P is an augmenting path in G.
b. If P uses the pseudo-vertex in G', then P' is obtained by replacing the pseudo-vertex with the corresponding path in the blossom B. Since P is an augmenting path in G', the endpoints of the path in the blossom B must be unmatched, and thus P' is an augmenting path in G.

Completeness: If there's an augmenting path P' in G, then there's an augmenting path P in G', preserving the invariant.
a. If P' does not use any vertex in the blossom B, then P = P' is an augmenting path in G'.
b. If P' uses vertices in the blossom B, then P can be obtained by replacing the path in the blossom B with the corresponding pseudo-vertex. Since P' is an augmenting path in G, the endpoints of the path in the blossom B must be unmatched, and thus P is an augmenting path in G'.

Now, let's analyze the complexity of the Blossom Algorithm on general graphs:
The algorithm begins by initializing a forest structure with unmatched vertices as roots, taking O(n) time. The search for augmenting paths is done using a breadth-first search, which takes O(m) time in the worst case, where m is the number of edges in the graph. When a blossom is found, it is compressed in O(n) time, since the worst-case scenario is a blossom with n vertices. The algorithm then recurses on the compressed graph. The number of recursion levels is bounded by n/2 (since each compression reduces the number of vertices by at least 2), and at each level, the time complexity is O(n+m).

Thus, the overall time complexity of the Blossom Algorithm is O(n^2(m+n)).

It is worth noting that there are more efficient implementations of the Blossom Algorithm with better complexity bounds. For example, the most efficient known implementation has a time complexity of O(n^3), which is significantly faster than the simple implementation outlined here.

b) Is the following statement true or false:: If G = (V, E) is a graph, M a matching and B a blossom, then there is an augmenting path in G with respect to M if there is one in G|B where blossom B is compressed into one vertex. (15pts)

The statement is true. If G = (V, E) is a graph, M a matching, and B a blossom, then there is an augmenting path in G with respect to M if there is one in G|B where blossom B is compressed into one vertex.

As explained in the previous response, the correctness of the Blossom Algorithm relies on the fact that compressing a blossom into a single vertex preserves the existence of augmenting paths.

In the compressed graph G|B, if there is an augmenting path with respect to M, then there is a corresponding augmenting path in the original graph G. The correspondence between the augmenting paths in G and G|B is established by replacing the pseudo-vertex in G|B with the corresponding path in the blossom B in G.

The proof of correctness for the Blossom Algorithm, as outlined in the previous response, demonstrates that if there's an augmenting path in G|B, then there's an augmenting path in G. Similarly, if there's an augmenting path in G, then there's an augmenting path in G|B. Thus, the statement is true.

PROBLEM 4

Suppose we generalize the matching requirement in bipartite graphs such that each node has not one but can have at most b edges incident onto it. Give an algorithm for maximum b-matching in bipartite graphs. (20pts)

A b-matching is a generalization of the matching problem where each node can have at most b edges incident onto it. To find the maximum b-matching in a bipartite graph, we can transform the problem into a regular maximum flow problem and then solve it using a standard max-flow algorithm like the Ford-Fulkerson algorithm or the Edmonds-Karp algorithm.

Given a bipartite graph G = (U, V, E) with node sets U and V, and an edge weight matrix W = [w(u,v)] representing the weight of the edge between nodes u ∈ U and v ∈ V, the algorithm proceeds as follows:

- Create a new directed graph G' = (V', E') by adding a source node s and a sink node t to the original graph G.
- Connect the source node s to each node u ∈ U with an edge (s, u) and capacity c(s, u) = b.
- Connect each node v ∈ V to the sink node t with an edge (v, t) and capacity c(v, t) = b.
- For each original edge (u, v) ∈ E, add a directed edge (u, v) in G' with capacity c(u, v) = w(u, v) (or 1, if the graph is unweighted).
- Find the maximum flow in G' using a max-flow algorithm such as the Ford-Fulkerson algorithm or the Edmonds-Karp algorithm.
- The maximum b-matching in G corresponds to the flow on the edges between U and V in the max-flow solution of G'.

The time complexity of this algorithm depends on the max-flow algorithm used. The Ford-Fulkerson algorithm has a time complexity of O(max_flow * |E'|), where max_flow is the value of the maximum flow in G' and |E'| is the number of edges in G'. In the worst case, the maximum flow is O(b * |U|) and the number of edges in G' is O(|U| + |V| + |E|), so the time complexity of Ford-Fulkerson is O(b * |U| * (|U| + |V| + |E|)).

If we use the Edmonds-Karp algorithm, the time complexity is O(|V'|^3) = O((|U| + |V| + 2)^3) since it only depends on the number of nodes in the graph G'

PROBLEM 5

(i) Prove the correctness of the algorithms for weighted bipartite matching and analyze the time complexity. (5pts)

THE STATEMENT OF PROBLEM 4 IS WRONG WRITTEN. I DO NOT UNDERSTAND IF THE PROBLEM 4 IS THE CONTEXT FOR THE 5 (i) OR NOT, SO, I HAVE DONE THE EXERCISE 5 TAKING INTO ACCOUNT BOTH SCENARIOS

a) CASE IN WHICH PROBLEM 4 IS THE CONTEXT FOR 5 (i)

Assuming you are referring to the Hungarian algorithm, which is a popular algorithm for solving the weighted bipartite matching problem.

To find the maximum weighted b-matching in a bipartite graph, we can use the Hungarian Algorithm, which is a combinatorial optimization algorithm for solving the assignment problem in polynomial time. The assignment problem is a special case of b-matching where b = 1. We can extend the Hungarian Algorithm to handle b > 1 by iteratively finding maximum weighted 1-matchings and updating the graph accordingly.

Algorithm for maximum weighted b-matching in bipartite graphs using the Hungarian Algorithm:

Initialize the maximum weighted b-matching M as an empty set. For i = 1 to b:

a. Apply the Hungarian Algorithm to the current graph G to find the maximum weighted 1-matching M_i.

b. Add M_i to the maximum weighted b-matching M.

c. Remove the edges in M_i from the graph G. Return the maximum weighted b-matching M.

Proof of correctness:
In each iteration, the Hungarian Algorithm guarantees that we find the maximum weighted 1-matching M_i in the current graph G. By performing this process b times and updating the graph in each iteration, we ensure that each node has at most b edges incident onto it in the final matching. The algorithm returns the maximum weighted b-matching in the bipartite graph.

The Hungarian algorithm finds an optimal assignment by iteratively modifying the weight matrix while preserving the relative optimality of the assignments. This is based on two key observations:

- If a constant is added or subtracted from all elements of a row or column, the optimality of the assignment does not change.
- If the minimum number of lines required to cover all zeros in the reduced matrix is equal to n, there exists an optimal assignment with n independent zeros in the matrix.

By applying these observations, the Hungarian algorithm ensures that it converges to an optimal assignment. The final reduced matrix contains n independent zeros, which correspond to the optimal matching in the original bipartite graph.

Time complexity analysis:
The Hungarian algorithm has a time complexity of O(n^3), where n is the number of nodes in each partition of the bipartite graph. This bound arises from the fact that, in each iteration, the algorithm needs to find the minimum number of lines to cover all zeros in the matrix, which can be done in O(n^2) time using a bipartite graph matching algorithm (e.g., the Hopcroft-Karp algorithm). Since there can be at most n iterations before the algorithm converges to the optimal assignment, the total time complexity is O(n^3).

b) CASE IN WHICH PROBLEM 4 IS NOT THE CONTEXT FOR 5 (i)

As in this scenario, problem 5 has nothing to do with problem 4, I will prove the correctness of the main algorithm used in problem 1 (ii) for weighted bipartite matching: the AUCTION METHOD

The auction algorithm operates on a bipartite graph with two sets of vertices, A and B, and a set of weighted edges connecting them. Each vertex in A is an item to be assigned, and each vertex in B is a potential assignee. The weight of an edge represents the benefit of assigning an item to an assignee.

The algorithm proceeds in a series of rounds. In each round, unassigned vertices in A send "bids" to vertices in B, based on the edge weights and the current prices at vertices in B. Vertices in B then update their prices and assignments based on the received bids.

Proof of correctness:

To prove the correctness, we need to show that the auction algorithm always terminates and produces a valid assignment.

Termination: The algorithm terminates when all vertices in A have been assigned to vertices in B. Each round, at least one vertex in A is assigned, and there are no cycles, so the algorithm must eventually terminate.

Valid assignment: When the algorithm terminates, it produces a valid assignment because:

- Each vertex in A is assigned to exactly one vertex in B (by the algorithm's construction).
- Each vertex in B is assigned to at most one vertex in A (since each vertex in B updates its assignment only when it receives a higher bid).

Time complexity analysis:

The time complexity of the auction algorithm depends on the number of rounds and the amount of work done in each round. In the worst case, each round involves updating the prices of all vertices in B and reassigning vertices in A. Thus, each round takes O(n) time, where n is the number of vertices in A and B.

The number of rounds is more difficult to analyze. In the worst case, the algorithm may require O(n^2) rounds, leading to a worst-case time complexity of O(n^3). However, in practice, the algorithm often converges more quickly, and there are variants and optimizations that can significantly improve performance.

In conclusion, the auction algorithm is correct for solving the weighted bipartite matching problem and has a worst-case time complexity of O(n^3).

(ii) Show that in the Hungarian method, when weights are integers the dual variables are always half-integral (i.e. multiples of 1/2) (10pts)

In the Hungarian Algorithm, dual variables are associated with each node in the bipartite graph. The dual variables are used to update the weights of the edges and to maintain feasibility of the dual problem. Let u_i and v_j be the dual variables for nodes i and j, respectively. According to the complementary slackness condition, the following must hold for edges (i, j) in the optimal matching:

w(i, j) = u_i + v_j

Since w(i, j) are integers, the sum u_i + v_j must also be an integer. Now, let's show that the dual variables are always half-integral:

Initially, the dual variables u_i are set to the maximum weight of the edges incident to node i, and v_j are set to 0. Since the weights are integers, the initial dual variables are integers as well.

During the Hungarian Algorithm, the dual variables are updated according to the minimum value of slack variables:

min_slack = min {u_i + v_j - w(i, j)}

The slack variables are updated as follows:

u_i = u_i - min_slack v_j = v_j + min_slack

Since the weights are integers, the difference (u_i + v_j - w(i, j)) is always an integer. Therefore, min_slack is an integer. When updating the dual variables, we subtract min_slack from u_i and add it to v_j. If u_i and v_j were both integers, they would remain integers after the update. If u_i and v_j were both half-integral, the update would still result in half-integral values.

By induction, we can show that the dual variables in the Hungarian Algorithm are always half-integral (i.e., multiples of 1/2) when the edge weights are integers.

(iii) Run the algorithms on the problem in 1(ii) with requirements (1, 1, 1, 1). (5pts)

AUCTION METHOD

C =
[99, 96, 93, 90] 3
[101, 98, 95, 92] 4
[103, 100, 97, 94] 2
[105, 102, 99, 96] 1
1 1 1 1

Pick the smallest (90) and assign the warehouse to the client. Warehouse 1 --> 1 units for client 4

C =
[99, 96, 93, --] 2
[101, 98, 95, --] 4
[103, 100, 97, --] 2
[105, 102, 99, --] 1
1 1 1 0

Pick the smallest (93) and assign the warehouse to the client Warehouse 1 --> 1 units for client 3

C =
[99, 96, --, --] 1
[101, 98, --, --] 4
[103, 100, --, --] 2
[105, 102, --, --] 1
1 1 0 0

Pick the smallest (96) and assign the warehouse to the client Warehouse 1 --> 1 units for client 2

C =
[--, --, --, --] 0
[101, --, --, --] 4
[103, --, --, --] 2
[105, --, --, --] 1
1 0 0 0

Pick the smallest (101) and assign the warehouse to the client Warehouse 2 --> 1 units for client 1

C =
[--, --, --, --] 0
[--, --, --, --] 3
[--, --, --, --] 2
[--, --, --, --] 1
0 0 0 0

Once clients requirements are met, we stop. In this problem, the total supply is NOT the same as total demand. Therefore, as there is more supply than demand, there will still be leftovers in the warehouses.

Therefore, the optimal assignment that minimizes the total cost is:

Warehouse 2 --> 1 units for client 1 Warehouse 1 --> 1 units for client 2 Warehouse 1 --> 1 units for client 3 Warehouse 1 --> 1 units for client 4

Therefore, the clients will be provided exactly (1, 1, 1, 1) units

C =
[99, 96, 93, 90] 3
[101, 98, 95, 92] 4
[103, 100, 97, 94] 2
[105, 102, 99, 96] 1
1 1 1 1

[--, 1, 1, 1] 3
[1, --, --, --] 1
[--, --, --, --] 0
[--, --, --, --] 0
1 1 1 1

And the total cost of this assignment is 1 * 101 + 1 * 96 + 1 * 93 + 1 * 90 = 380

LINEAR PROGRAM

Finally, as a second solution, we will run the algorithm for the following transhipment problem, approaching it using linear programming. The result should be the same.

Then, using the EXCEL SOLVER tool...

Cost = c_ij = 2i - 3j + 100

C = [99, 96, 93, 90] 3
[101, 98, 95, 92] 4 WAREHOUSE SUPPLIES
[103, 100, 97, 94] 2
[105, 102, 99, 96] 1
1 1 1 1

CLIENTS REQUIREMENTS

FINAL DECISIONS

[--, 1, 1, 1] 3
[1, --, --, --] 1
[--, --, --, --] 0
[--, --, --, --] 0
1 1 1 1

Both, for the LINEAR PROBLEM and the DUAL LINEAR PROBLEM, the objective function optimized value is: 380