INFO 6205 – Assignment 3

Student Name: Yuetong Guo

Professor: Nik Bear Brown

Q1(10 Points)

Write a code with an algorithm explaining the solution for following problem:
There is a ball in a maze with empty spaces (represented as 'o') and walls (represented as 'x'). The ball can go through the empty spaces by rolling up, down, left or right, but it won't stop rolling until hitting a wall. When the ball stops, it could choose the next direction.
Given the (m x n) maze, the ball's start position and the destination, where start = [start_row, start_col] and destination = [destination_row, destination_col], return true if the ball can stop at the destination, otherwise return false. [You may assume that the borders of the maze are all walls]

Ans:

When designing the algorithm to solve the ball-in-maze problem, the primary goal is to determine if the ball can reach the destination by following the maze's constraints. We can use a Breadth-First Search (BFS) approach to explore the maze, considering the fact that the ball can only change direction when it hits a wall. Here's a step-by-step breakdown of the algorithm design:

1. **Initialize data structures**: Create a queue (**q**) for BFS and a set (**visited**) to store the visited positions in the maze. Initialize the queue with the starting position and add the starting position to the visited set.
2. **BFS traversal**: While the queue is not empty, perform the following steps:

   a. Dequeue the first position from the queue.

   b. Check if the dequeued position is the destination. If it is, return **True**.

   c. For each direction (up, down, left, right), perform the following steps:

      i. Simulate the ball rolling in the current direction until it hits a wall, updating the position at each step.

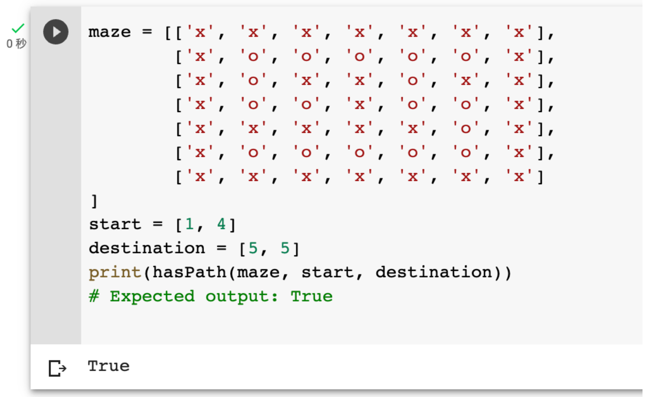
      ii. If the updated position has not been visited, add it to the visited set and enqueue it for further exploration.

3. **Determine the result**: If the BFS traversal finishes without finding the destination, return **False**. Otherwise, return **True**.

By using a BFS approach, the algorithm explores the maze in layers, considering all possible directions from each valid position. When the ball hits a wall, it can choose the next direction, and the BFS traversal ensures that all possible paths are explored. If the ball can reach the destination, the traversal will find it and return **True**. Otherwise, if all paths have been explored and the destination has not been reached, the algorithm returns **False**.

The related code and test code are in Colab document.

The output is as right show:



```
maze = [['x', 'x', 'x', 'x', 'x', 'x', 'x'],
        ['x', 'o', 'o', 'o', 'o', 'o', 'x'],
        ['x', 'o', 'x', 'x', 'o', 'x', 'x'],
        ['x', 'o', 'o', 'x', 'o', 'o', 'x'],
        ['x', 'x', 'x', 'x', 'x', 'o', 'x'],
        ['x', 'o', 'o', 'o', 'o', 'o', 'x'],
        ['x', 'x', 'x', 'x', 'x', 'x', 'x']
]
start = [1, 4]
destination = [5, 5]
print(hasPath(maze, start, destination))
# Expected output: True
```

```
True
```

Q2 (10 points)

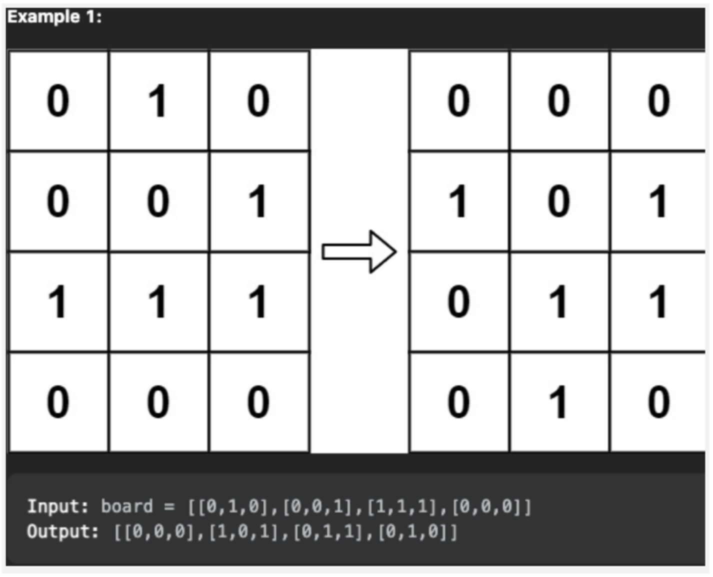
According to Wikipedia's article: "The Game of Life, also known simply as Life, is a cellular automaton devised by the British mathematician John Horton Conway in 1970."

The board is made up of an m x n grid of cells, where each cell has an initial state: live (represented by a 1) or dead (represented by a 0). Each cell interacts with its eight neighbors (horizontal, vertical, diagonal) using the following four rules (taken from the above Wikipedia article):

Any live cell with fewer than two live neighbors dies as if caused by under-population. Any live cell with two or three live neighbors lives on to the next generation.

Any live cell with more than three live neighbors dies, as if by over-population.

Any dead cell with exactly three live neighbors becomes a live cell, as if by reproduction.



```
Input: board = [[0,1,0],[0,0,1],[1,1,1],[0,0,0]]
Output: [[0,0,0],[1,0,1],[0,1,1],[0,1,0]]
```

The next state is created by applying the above rules simultaneously to every cell in the current state, where births and deaths occur simultaneously. Given the current state of the m x n grid board, return the next state.

Write a code and explain your approach to solving the problem with the help of an algorithm.

Ans:

When designing the algorithm for Conway's Game of Life, the primary goal is to simulate the next state of the given board by applying the four rules simultaneously to each cell. The key challenge is to ensure that the next state of the board is calculated based on the current state, without any intermediate updates affecting the outcome. To accomplish this, we can use a two-pass approach:

1. Calculate next state: For each cell in the board, calculate its next state based on the rules and the current state of its neighboring cells. To avoid modifying the original state of the board directly, we can use temporary values to represent the next state of each cell:
   - Dying cells (alive cells that will be dead in the next state) are marked with -1.
   - Becoming alive cells (dead cells that will be alive in the next state) are marked with 2.

This step requires counting live neighbors for each cell, which can be done using a helper function count_live_neighbors(x, y).

2. Update the board: After determining the next state for all cells, update the board in place. If a cell has a positive value (1 or 2), it should be alive (set its value to 1). If a cell has a non-positive value (0 or -1), it should be dead (set its value to 0).

Here's a step-by-step breakdown of the algorithm design:

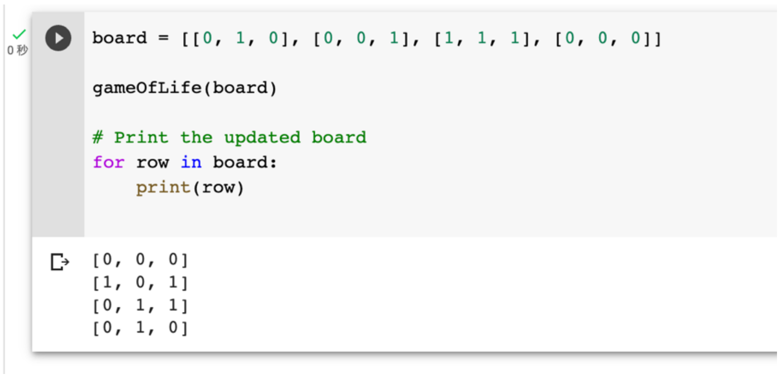
1. Define a helper function count_live_neighbors(x, y) that takes the coordinates of a cell and returns the count of its live neighbors. This function can use a list of possible directions (directions) to traverse the neighboring cells.
2. Loop through each cell of the board and call the count_live_neighbors function to get the count of live neighbors for that cell.

3. Apply the four rules of the Game of Life based on the count of live neighbors:
    - If the cell is alive and has fewer than 2 or more than 3 live neighbors, mark it as dying (change its value to -1).
    - If the cell is dead and has exactly 3 live neighbors, mark it as becoming alive (change its value to 2).
4. Loop through each cell of the board again and update the cell values based on the markings:
    - If a cell's value is positive (1 or 2), it is now alive (set it to 1).
    - If a cell's value is non-positive (0 or -1), it is now dead (set it to 0).

By using this two-pass approach, we can effectively simulate the next state of the board while maintaining the original state during the calculation. The temporary values (-1 and 2) allow us to keep track of the previous states while updating the board in place.

The related code and example test code are in Colab document.

The output is as below:



```
board = [[0, 1, 0], [0, 0, 1], [1, 1, 1], [0, 0, 0]]

gameOfLife(board)

# Print the updated board
for row in board:
    print(row)
```

```
[0, 0, 0]
[1, 0, 1]
[0, 1, 1]
[0, 1, 0]
```

Q3 (10 points)

A (5 points): Give an algorithm to find the position of largest element in an array of n numbers using divide-and-conquer. Write down the recurrence relation and find out the asymptotic complexity for this algorithm.
B (5 points): Give a divide-and-conquer algorithm to find the largest and the smallest elements in an array of n numbers. Write down the recurrence relation and find out the asymptotic complexity for this algorithm.

A Ans:

To find the position of the largest element in an array of n numbers using a divide-and-conquer approach, we can recursively divide the array into two halves until we reach the base case (a single element) and then compare the largest elements in the two halves to find the global maximum.

Here's a step-by-step breakdown of the algorithm:

1. If the array has only one element, return its position as the largest element's position.
2. Divide the array into two halves: left and right.
3. Recursively find the position of the largest element in the left half and the right half.
4. Compare the elements at the positions found in step 3 and return the position of the larger element.

Recurrence relation:

Let T(n) be the time complexity of finding the position of the largest element in an array of n numbers. The array is divided into two halves at each level, so the recurrence relation can be written as:

T(n) = 2 * T(n / 2) + O(1)

The O(1) term comes from the constant time spent comparing the largest elements in the left and right halves.

Asymptotic complexity:

Using the Master theorem, we can find the asymptotic complexity of the algorithm:

a = 2, b = 2, and f(n) = O(1)

Since log_b(a) = log_2(2) = 1 and f(n) = O(n^c) with c = 0, the case 2 of the Master theorem applies:

T(n) = O(n^(log_b(a)) * log(n)) = O(n * log(n))

Thus, the asymptotic complexity of this divide-and-conquer algorithm for finding the position of the largest element in an array is O(n * log(n)).

B Ans:

To find the largest and smallest elements in an array of n numbers using a divide-and-conquer approach, we can recursively divide the array into two halves until we reach the base case (a single element) and then compare the largest and smallest elements in the two halves to find the global maximum and minimum.

Here's a step-by-step breakdown of the algorithm:

1. If the array has only one element, return this element as both the largest and smallest elements.
2. Divide the array into two halves: left and right.
3. Recursively find the largest and smallest elements in the left half and the right half.
4. Compare the largest and smallest elements found in step 3 and return the maximum and minimum of these elements.

Recurrence relation:

Let T(n) be the time complexity of finding the largest and smallest elements in an array of n numbers. The array is divided into two halves at each level, so the recurrence relation can be written as:

T(n) = 2 * T(n / 2) + O(1)

The O(1) term comes from the constant time spent comparing the largest and smallest elements in the left and right halves.

Asymptotic complexity:

Using the Master theorem, we can find the asymptotic complexity of the algorithm:

a = 2, b = 2, and f(n) = O(1)

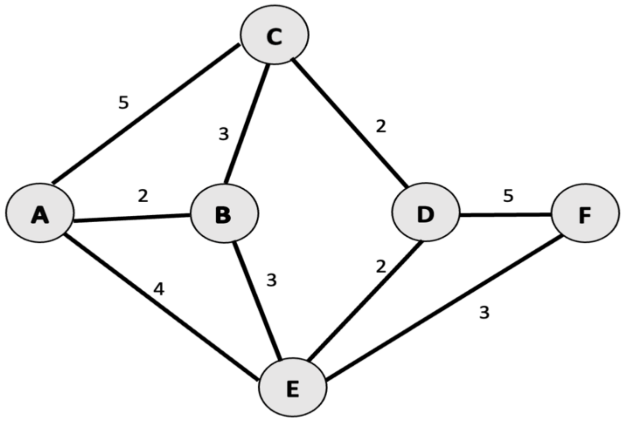
Since log_b(a) = log_2(2) = 1 and f(n) = O(n^c) with c = 0, the case 2 of the Master theorem applies:

T(n) = O(n^(log_b(a)) * log(n)) = O(n * log(n))

Thus, the asymptotic complexity of this divide-and-conquer algorithm for finding the largest and smallest elements in an array is O(n * log(n)).

Q4 (5 Points)

Use Kruskal's algorithm to find a minimum spanning tree for the connected weighted graph below: What is the Time Complexity of Kruskal's algorithm?



Ans:

Kruskal's algorithm is a greedy algorithm that is used to find the minimum spanning tree of a connected, weighted graph. The minimum spanning tree is a subgraph that connects all the vertices of the graph with the minimum possible total edge weight.

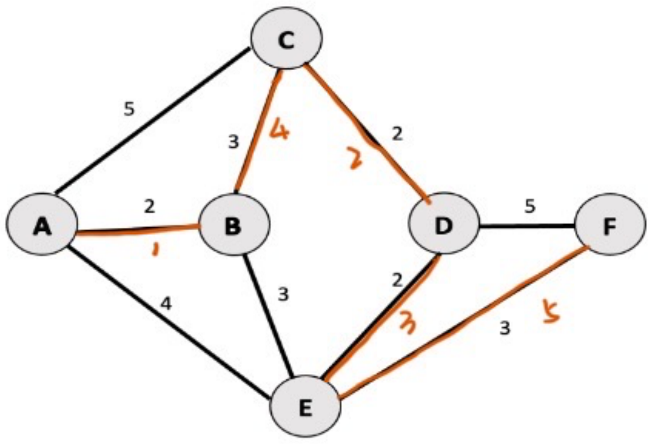
To apply Kruskal's algorithm to the given graph, we follow these steps:

1. Sort all the edges of the graph in non-decreasing order of their weights:
   - [("A", "B", 2), ("C", "D", 2), ("D", "E", 2), ("B", "C", 3), ("B", "E", 3), ("E", "F", 3), ("A", "E", 4), ("A", "C", 5), ("D", "F", 5)]
2. Initialize an empty set to store the edges of the MST:
   - mst = []
3. Iterate through the sorted edges, and for each edge:
   - Check if adding the edge to the MST set will form a cycle.
   - If adding the edge does not form a cycle, add it to the MST set.
4. Continue iterating through the edges until the MST set contains n - 1 edges, where n is the number of vertices in the graph.

Using the above steps, we can find the minimum spanning tree of the given graph:

- Add edge ("A", "B", 2) to the MST set.
- Add edge ("C", "D", 2) to the MST set.
- Add edge ("D", "E", 2) to the MST set.
- Add edge ("B", "C", 3) to the MST set. Now, vertex B and C are already connected, so adding the edge ("B", "E", 3) would create a cycle; skip it.
- Add edge ("E", "F", 3) to the MST set. Now we have n - 1 edges in the MST set, where n is the number of vertices (6).

The MST set is now complete:



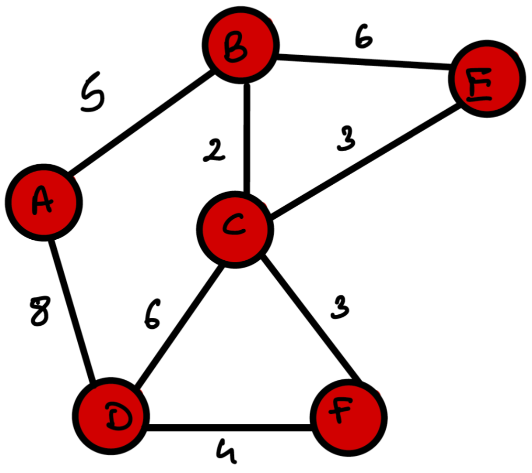
- mst = [("A", "B", 2), ("C", "D", 2), ("D", "E", 2), ("B", "C", 3), ("E", "F", 3)]

The time complexity of Kruskal's algorithm is O(E * log(V)), where E is the number of edges in the graph and V is the number of vertices.

The time complexity is determined by sorting the edges (O(E * log(E))) and iterating through the sorted edges and checking for cycles using a Union-Find data structure (O(E * log(V))). Since in a connected graph, E can be at most V^2, the time complexity can be written as O(E * log(V)).

Q5 (5 Points)

Use Prim's algorithm to find a minimum spanning tree for the connected weighted graph below. Show your work. What is the Time Complexity of Prim's algorithm?



Ans:

Prim's algorithm is a greedy algorithm that is used to find the minimum spanning tree of a connected, weighted graph. The minimum spanning tree is a subgraph that connects all the vertices of the graph with the minimum possible total edge weight.

To apply Prim's algorithm to the given graph, we follow these steps:

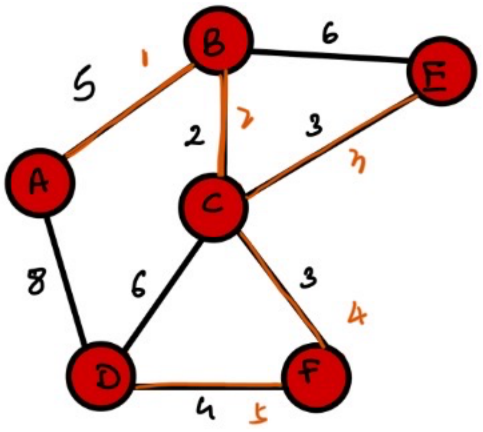
1. Choose an arbitrary vertex to start from (in this example, we'll start from vertex A).
2. Initialize the MST set, and a set containing unvisited vertices:
   - mst = []
   - unvisited = {"A", "B", "C", "D", "E", "F"}
3. As long as there are unvisited vertices, continue the following steps:
   - From the set of unvisited vertices, find the edge with the minimum weight connected to the vertices already in the MST. Initially, the MST contains only vertex A.
   - Add the minimum weight edge to the MST set and remove the corresponding vertex from the unvisited set.
4. Repeat step 3 until all vertices have been visited and added to the MST set.

Using the above steps, we can find the minimum spanning tree of the given graph:

- Add edge ("A", "B", 5) to the MST set and remove B from the unvisited set.
- Add edge ("B", "C", 2) to the MST set and remove C from the unvisited set.
- Add edge ("C", "E", 3) to the MST set and remove E from the unvisited set.
- Add edge ("C", "F", 3) to the MST set and remove F from the unvisited set.
- Add edge ("D", "F", 4) to the MST set and remove D from the unvisited set.

The MST set is now complete:

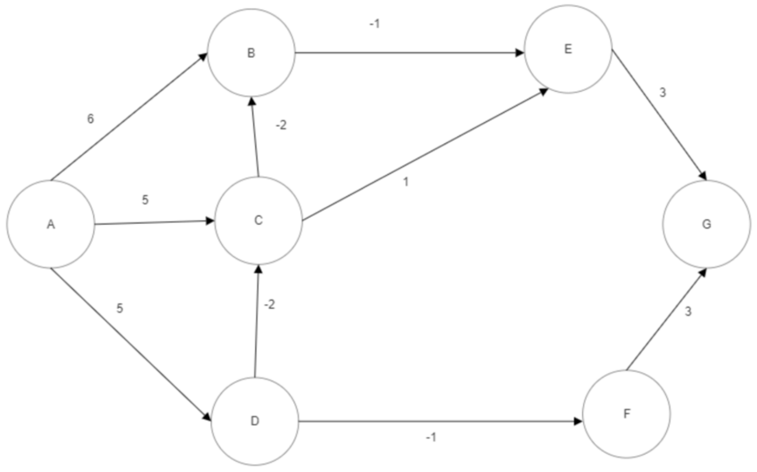
mst = [("A", "B", 5), ("B", "C", 2), ("C", "E", 3), ("C", "F", 3), ("D", "F", 4)]



The time complexity of Prim's algorithm depends on the data structures used for implementation. If you use an adjacency matrix representation and a linear search to find the minimum edge, the time complexity would be O(V^2), where V is the number of vertices in the graph. If you use an adjacency list representation and a binary heap or a Fibonacci heap, the time complexity can be reduced to O(E * log(V)) or O(E + V * log(V)), respectively, where E is the number of edges in the graph.

Q6(5 Points)

Use the Bellman-Ford algorithm to find the shortest path from node A to G in the weighted directed graph below. Show your work.
What is the time complexity of Bellman-Ford? And for what condition algorithm fail?



Ans:
The Bellman-Ford algorithm is used to find the shortest path between two vertices in a weighted directed graph, where the weights of the edges may be negative. To find the shortest path from vertex A to vertex G in the given graph using the Bellman-Ford algorithm, we perform the following steps:

1. Initialize the distance to every vertex to infinity, except for the start vertex which is initialized to 0.

```
dist = {
    "A": 0,
    "B": float("inf"),
    "C": float("inf"),
    "D": float("inf"),
    "E": float("inf"),
    "F": float("inf"),
    "G": float("inf")
}
```

2. Relax each edge in the graph V-1 times, where V is the number of vertices in the graph.

```
for i in range(len(graph)-1):
    for u in graph:
        for v,w in graph[u]:
            if dist[u] != float("inf") and dist[v] > dist[u] + w:
                dist[v] = dist[u] + w
```

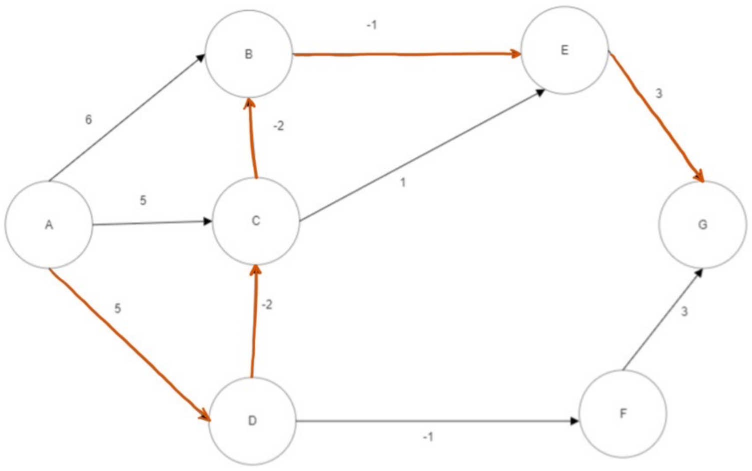
3. Check for negative weight cycles. If a negative weight cycle is found, the algorithm reports that no shortest path exists.

```
for u in graph:
    for v,w in graph[u]:
        if dist[u] != float("inf") and dist[v] > dist[u] + w:
            print("Negative weight cycle found")
            return
```

In this particular example, the shortest path from vertex A to vertex G is 3, and the path is A-D-C-B-E-G. Let’s use Matrix representation for the Bellman-Ford algorithm's process:

Right is step (i-1) Below is node	0	1	2	3	4	5	6
A	0	0	0	0	0	0	0
B	∞	6	3	1	1	1	1
C	∞	5	3	3	3	3	3
D	∞	5	5	5	5	5	5
E	∞	∞	5	2	0	0	0
F	∞	∞	4	4	4	4	4
G	∞	∞	∞	7	5	3	3

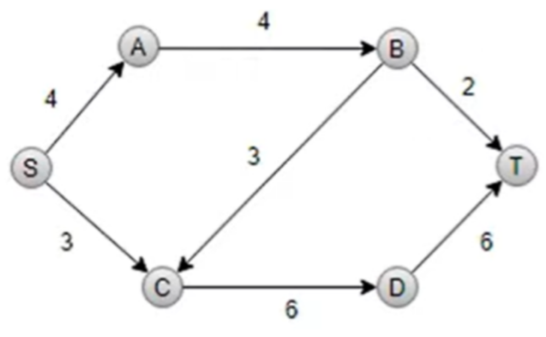
We can see that the distance from vertex A to vertex G is 3, which is the shortest distance. The path that corresponds to this distance is A-D-C-B-E-G, which has a total weight of 3.



The Bellman-Ford algorithm would fail if there's negative weight cycles in the graph. If a negative weight cycle exists in the graph, the algorithm will detect it and report that no shortest path exists. In this case, we do not have a negative weight cycle, so the algorithm is able to find the shortest path.

Q7(5 Points)

Use the Ford-Fulkerson algorithm to find the maximum flow from node S to T in the weighted directed graph below. Show your work.



Ans:

The Ford-Fulkerson algorithm is a method for computing the maximum flow in a flow network. To apply the algorithm to the given graph and find the maximum flow from node S to T, we can follow these steps:

1. Initialize the flow to 0.

   ```
   flow = 0
   ```

2. While there exists an augmenting path from S to T, find the bottleneck capacity of the augmenting path and update the flow along the path.

   ```
   while True:
   # Find an augmenting path from S to T using depth-first search
   path, bottleneck = find_augmenting_path(graph, "S", "T", capacities, flows)
   # If no augmenting path exists, we have found the maximum flow
   if not path:
       break
   # Update the flow along the augmenting path
   for i in range(len(path)-1):
       u, v = path[i], path[i+1]
       flows[u][v] += bottleneck
       flows[v][u] -= bottleneck
    # Update the total flow
   flow += bottleneck
   ```

3. To find an augmenting path, we can use depth-first search. We start at the source node S and follow any edge with positive residual capacity to a neighboring node. If we reach the destination

node T, we have found an augmenting path. Otherwise, we continue the search from the last visited node.

```
def find_augmenting_path(graph, start, end, capacities, flows):
# Use depth-first search to find an augmenting path
stack = [(start, [start])]
visited = set()
while stack:
    node, path = stack.pop()
    if node == end:
            # Found an augmenting path
            bottleneck = min(capacities[u][v] - flows[u][v] for u, v in zip(path, path[1:]))
            return path, bottleneck
    visited.add(node)
    for neighbor, capacity in graph[node]:
            if capacity - flows[node][neighbor] > 0 and neighbor not in visited:
            stack.append((neighbor, path + [neighbor]))
            # No augmenting path found
    return None, 0
```

In this particular example, the maximum flow from S to T is 7.

Here's how we can compute this:

- We start with a flow of 0.
- We find an augmenting path S-A-B-T with bottleneck capacity 2, and update the flow along the path: S->A->B->T = 2.
- We find another augmenting path S-C-D-T with bottleneck capacity 5, and update the flow along the path: S->C->D->T = 5.
- We find one more augmenting path S-C-B-T with bottleneck capacity 1, and update the flow along the path: S->C->B->T = 1.
- We cannot find any more augmenting paths, so we have reached the maximum flow.
- The total flow is the sum of the flow along the augmenting paths: flow = 2 + 5 + 1 = 7.

Therefore, the maximum flow from node S to T in the given graph is 7.

Q8(10 Points)

Suppose you live with n − 1 other people, at a popular off-campus cooperative apartment, the Ice-Cream, and Rainbows Collective. Over the next n nights, each of you is supposed to cook dinner for the co-op exactly once, so that someone cooks on each of the nights. Of course, everyone has scheduling conflicts with some of the nights (e.g., algorithms exams, Miley concerts, etc.), so deciding who should cook on which night becomes a tricky task. For concreteness, let's label the people, P {p1, . . . , pn}, the nights, N {n1, . . . , nn} and for person pi, there's a set of nights Si {n1, . . . , nn} when they are not able to cook. A person cannot leave Si empty. If a person isn't doesn't get scheduled to cook in any of the n nights, they must pay $200 to hire a cook.

A (5 Points). Express this problem as a maximum flow problem that schedules the maximum number of matches between the people and the nights.

B (5 Points) Can all n people always be matched with one of the n nights? Prove that it can or cannot.

A Ans:

This problem can be formulated as a bipartite matching problem in a bipartite graph, where one set of vertices represents the people (P) and the other set represents the nights (N). To construct the bipartite graph, add an edge (pi, nj) if person pi is able to cook dinner on night nj.

Once you have constructed the bipartite graph, convert it into a flow network by directing all edges from "people" to "nights," adding a source vertex s with edges from s to each person, and a sink vertex t with edges from each night to t. Set the capacity of each edge to be equal to 1. Then, find a maximum flow in the network using a maximum flow algorithm, such as the Ford-Fulkerson algorithm.

A feasible dinner schedule exists if and only if the maximum flow value equals n, which means that all n people are matched to distinct nights. In this case, examine the edges in the maximum flow that join people to nights to determine the optimal schedule. If the maximum flow value is less than n, there isn't a feasible schedule where all n people cook exactly once, and one or more people will need to pay $200 to hire a cook.

Constructing the bipartite graph takes time O(n^2), as there are at most n^2 edges in the bipartite graph, with each of the n people potentially joined to n nights. Adding the additional edges to create the directed flow network takes time O(n). The overall complexity of the solution is dominated by the maximum flow algorithm, which has a complexity of O(n^3) in this case, using the Edmonds-Karp algorithm as an example. Therefore, finding a feasible dinner schedule or concluding that there isn't one can be done in time O(n^3).

B Ans:

We cannot guarantee that all n people can always be matched with one of the n nights, as it depends on the specific constraints (Si) for each person.

To prove that it is not always possible to match all n people with n nights, consider the following counterexample:

Suppose we have 3 people (n = 3): P = {p1, p2, p3} and 3 nights: N = {n1, n2, n3}. The constraints for each person are as follows:

- p1 can't cook on nights: S1 = {n1}
- p2 can't cook on nights: S2 = {n2}
- p3 can't cook on nights: S3 = {n3}

In this situation, we can construct the bipartite graph as follows:

1. p1 can be matched with nights: {n2, n3}
2. p2 can be matched with nights: {n1, n3}
3. p3 can be matched with nights: {n1, n2}

When we apply the maximum flow algorithm to the resulting flow network, we find the following matching:

- p1 cooks on night n2
- p2 cooks on night n1
- There is no available night for p3 to cook

In this case, the maximum flow value is 2, which is less than n (3). Consequently, not all 3 people can be matched with one of the 3 nights, and p3 would need to pay $200 to hire a cook.

This counterexample demonstrates that it is not always possible to match all n people with one of the n nights due to individual constraints.

Q9(10 Points)

In a standard s-t Maximum-Flow Problem, we assume edges have capacities, and there is no limit on how much flow is allowed to pass through a node. In this problem, we consider the variant of the Maximum-Flow and Minimum-Cut problems with node capacities. Let G = (V, E) be a directed graph, with source s V, sink t V, and nonnegative node capacities {cv ≥ 0} for each v V. Given a flow f in this graph, the flow through a node v is defined as f in(v). We say that a flow is feasible if it satisfies the usual flow-conservation constraints and the node-capacity constraints: fin(v) ≤ cv for all nodes. Give a polynomial-time algorithm to find an s-t maximum flow in such a node-capacitated network. Define an s-t cut for node-capacitated networks and show that the analog of the Max-Flow Min-Cut Theorem holds true.

Ans:

In this problem, we consider the variant of the Maximum-Flow and Minimum-Cut problems with node capacities. Let G = (V, E) be a directed graph, with source s ∈ V, sink t ∈ V, and nonnegative node capacities {cv ≥ 0} for each v ∈ V. We say that a flow is feasible if it satisfies the usual flow-conservation constraints and the node-capacity constraints: fin(v) ≤ cv for all nodes. Our goal is to find an s-t maximum flow in such a node-capacitated network and show that the analog of the Max-Flow Min-Cut Theorem holds true.

First, we construct a new graph H, where for every vertex v in G, we have two vertices vi and vo. We add an edge from vi to vo of capacity cv. If graph G has an edge (u, v), we add an edge (uo, vi) in H of infinite capacity. This construction ensures that the total amount of flow entering vi (or leaving vo) cannot exceed cv.

To find the s-t maximum flow in a node-capacitated network, we can modify the Ford-Fulkerson algorithm:

1. Start with a feasible flow f=0.
2. While there exists an s-t path in the residual graph G_f, find an augmenting path p that satisfies both edge and node capacity constraints.
3. Update the flow along p by the maximum amount possible, i.e., min{c_e - f_e, min{c_v - f_in(v) : v in p}}.
4. Repeat steps 2-3 until no more augmenting paths exist.

5. Return the final flow f.

To show that the Max-Flow Min-Cut theorem holds true for node-capacitated networks, we need to define an s-t cut. In this case, an s-t cut is a partition of the nodes V into two sets, S and T=V-S, such that s is in S and t is in T. The capacity of an s-t cut is defined as the sum of the capacities of the edges leaving S, plus the sum of the capacities of the nodes in S.

The analog of the Max-Flow Min-Cut theorem states that the maximum s-t flow in a node-capacitated network is equal to the capacity of the minimum s-t cut.

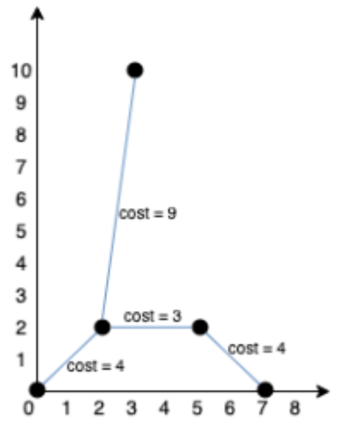
Proof:

1. Show that the maximum s-t flow in a node-capacitated network is less than or equal to the capacity of the minimum s-t cut.
2. Show that the maximum s-t flow in a node-capacitated network is greater than or equal to the capacity of the minimum s-t cut.

Combining these two parts, we conclude that the maximum s-t flow in a node-capacitated network is equal to the capacity of the minimum s-t cut. This shows that the analog of the Max-Flow Min-Cut theorem holds true for node-capacitated networks.

Q10(10 Points)

You are given an array of points representing integer coordinates of some points on a 2D-plane, where points[i] = [x_i, y_i]. The cost of connecting two points [x_i, y_i] and [x_j, y_j] is the manhattan distance between them: $|x_i - x_j| + |y_i - y_j|$, where |val| denotes the absolute value of val. Return *the minimum cost to make all points connected.* All points are connected if there is exactly one simple path between any two points. Use an appropriate algorithm to find the minimum cost. Support solution using runnable code.(Language of your choice)



Input: points = [[0,0],[2,2],[3,10],[5,2],[7,0]]

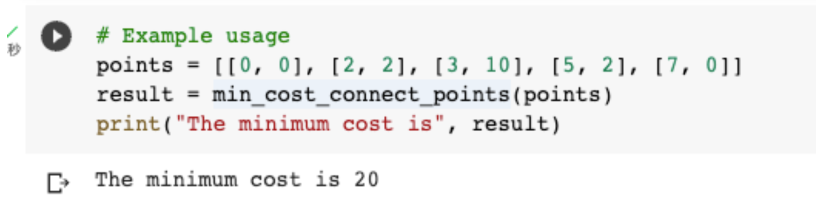
Ans:

To solve this problem, we could use Kruskal's algorithm to find the minimum spanning tree of the given graph. The minimum cost to make all points connected will be the sum of the edge weights in the minimum spanning tree.

The algorithm firstly calculates the manhattan distance between all pairs of points and sorts the edges by their distances. Then, it uses Kruskal's algorithm to find the minimum spanning tree by iterating through the edges and adding them to the tree if they don't create a cycle. Finally, the function returns the sum of the edge weights in the minimum spanning tree, which represents the minimum cost to connect all points.

The related code and test code are in Colab document.

The output is as right show:



```
# Example usage
points = [[0, 0], [2, 2], [3, 10], [5, 2], [7, 0]]
result = min_cost_connect_points(points)
print("The minimum cost is", result)
```

```
The minimum cost is 20
```

Q11(5 Points)

Consider using a simple linked list as a dictionary. Assume the client will never provide duplicate elements, so we can insert elements at the beginning of the list. Now assume the peculiar situation that the client may Perform any number of insert operations but will only ever perform atmost one lookup operation.

A. What is the worst-case running time of the operations performed on this data structure under the assumptions above? Briefly justify your answer.

B. What is the worst-case amortized running time of the operations performed on this data structure under the assumptions above? Briefly justify your answer.

A Ans:

The worst-case running time is O(n).

Under the given assumptions, insert operations always take O(1) time since elements are inserted at the beginning of the linked list. However, the lookup operation can take O(n) time in the worst case, as it may need to traverse the entire list to find the desired element. Since the client can perform any number of insert operations but only one lookup, the worst-case running time of the operations performed on this data structure is dominated by the lookup operation, which is O(n).

B Ans:

The worst-case amortized running time is O(1).

The insert operation always takes O(1) time. For the single allowed lookup operation, the worst-case time complexity is O(n). Given that there can be any number of insert operations but at most one lookup operation, the total cost of n operations (where n is the number of elements in the list) is at most O(n). Therefore, when we consider the amortized time across all n operations, the worst-case amortized running time is O(n)/n, which simplifies to O(1).