Parker Phillips

Worked Alone

1)

a. Used ChatGPT 5.1 for help

```
def compute_distance(p1: Tuple[float, float], p2: Tuple[float,
float]) -> float:
      return math.hypot(p1[0] - p2[0], p1[1] - p2[1])


def build_edges(
    towns: List[Tuple[float, float]],
    C: float,
    m: float,
    c: float,
    M: float
) -> List[Edge]:
    n = len(towns)
    edges: List[Edge] = []

    # break-even distance
    Break_even = (C - c) / (M - m)

    for i in range(n):
        for j in range(i + 1, n):
            d = compute_distance(towns[i], towns[j])
            road_cost = c + M * d
            air_cost = C + m * d

            if d <= break_even:
                # road is cheaper
                cost = road_cost
                kind = "road"
            else:
                cost = air_cost
                kind = "air"

            edges.append(Edge(u=i, v=j, cost=cost, kind=kind))

    return edges


def minimum_transportation_network(
    towns: List[Tuple[float, float]],
    C: float,
    m: float,
    c: float,
    M: float
```

```
) -> Tuple[float, List[Edge]]:
    n = len(towns)
    if n == 0:
        return 0.0, []

    # Build all candidate edges with their cheapest mode
(road/air)
    edges = build_edges(towns, C, m, c, M)

    # Sort edges by cost
    edges.sort(key=lambda e: e.cost)

    uf = UnionFind(n)
    chosen_edges: List[Edge] = []
    total_cost = 0.0

    for e in edges:
        if uf.union(e.u, e.v):
            chosen_edges.append(e)
            total_cost += e.cost
            # Stop when we have n - 1 edges
            if len(chosen_edges) == n - 1:
                break

    return total_cost, chosen_edges
```

This requires a few different methods. The first one `compute_distance` simply computes the Euclidean distance between two towns. So this is just how we can determine their distance. `build_edges` first calculates the breakeven distance to determine whether air or road is cheaper. If a distance is less than the breakeven distance, then road is cheaper. Otherwise air is cheaper. We can then go through all of the edges and give their edge the appropriate cost, and then a label if it is road or air. `minimum_transportation_network` first calls `build_edges` so we can get the edge information. We then want to sort the edges by the cost so we can use Kruskal's algorithm for MST. We then just loop over the edges from cheapest to most expensive, and if the e.u and e.v can't be unioned yet then we add them to the edges list and increase our total cost. From here we get the total cost once it is done and the edges for our MST. Our build_edges has nested loops that run n times so we get O(n^2). For minimum_transportation_network, we have to sort all of the edges and our edges could be n^2 so our runtime would be O(n^2log(n^2) or O(n^2log(n)). This is our worst complexity overall so it is O(n^2log(n)).

b. So we have 4 variables, C = fixed cost for air, m = per-mile cost for air, c = fixed cost for road, and M = per-mile cost for a road. Air(d) = C + md and Road(d) = c + Md. Planes are more expensive to own but cheaper per mile so our air travel should be more efficient. So we can then state C + md < c + Md => C – c < (M – m)d =>
d > (C – c)/(M – m). We then use this as our breakeven method to determine if d > than that then our air is cheaper and less, then our road is cheaper.

2)
```
Shortest_paths_from_source(G, s):
        init dist obj to infinity for each node
        init parent obj to None for each node
        init edge_count obj to infinity for each node
        dist[s] = 0;

        init Q (queue) with s, 0 (dist), 0(edges)

        while Q is not empty:
              cur_dist, cur_edges, u = pop(Q)

              for each neighbor v and weight of u:
                    new_dist = dist[u] + w
                    new_edges = edge_count[u] + 1

                    if (new_dist < dist[v]) or (new_dist == dist[v] and
                                                new_edges < edge_cunt[v]):

                          dist[v] = new_dist
                          edge_count[v] = new_edges
                          parent[v] = u
                          q.push(new_dist, new_edges, v)

        return dist, edge_count, parent

  get_path(parent, source, target):
        init path[]
        cur = target

        while cur:
              path.append(cur)
              cur = parent[cur]

        reverse path
        return path

  main:
        dist, edges, parent = shortest_paths_from_source(G, s)
        for each node v in G besides s:
```

```
path = get_path(parent, s, v)
```

This algorithm uses Dijkstra's algorithm but with a slight alteration. We keep track of the best dist from s to each node we find. We also keep track of the parent and the number of edges on the best path from s to v. We then go through the queue until it is empty which starts with just s. For each node in the queue we then loop through all of it's neighbors and relax the distance by increasing the distance by the weight and the edge count by 1. We then check if we improved the total distance by seeing if the new distance is less than the previous distance or if it is the same with less edges overall. If it is then we update the distance and edge count fo that neighbor and updated the parent to have the previous u node. We can then push this new node onto the queue and do this all again till we can return the final distance, edge_count, and parent. In our main we then call this function to get the shortest path from s to every single node. We can then loop through all of the nodes in the graph and recreate the path from each node back to s. To analyze the runtime of this algorithm we first look at how many nodes we have to visit. We will end up touching every node for initializing the data so that gives us O(|V|). For our queue, assuming it is a min-heap priority queue, requires us to push at most once for every single edge since every edge to a node may be better than the last so add it to the queue. For pushing onto a heap it is O(logn), so this would give us O(|E|log(n)). Then we have to pop the nodes, and so if we have the worst case where we pop every node then we would have |V| pops which again is O(logn). So this would be O(|V|log(|V|). This combined gives us O((|V| + |E|)log(|V|)).

3)

a. To determine if there is a feasible route from Bellingham to any other city in linear time would first involve looping through all of the edges and removing any edge that is greater than our max distance. We could do this by building an adjacency list and only adding the neighbors that have a distance less than the max distance. We could do this in O(|E|) since we are just looping through all the edges. We could then run BFS or DFS on the adjacency list to determine if there is a path from Bellingham to any city. This would just cost O(|V| + |E'|) where E' is the number of edges after removing any that were greater than D. This would allow us to find some path in linear time.

b.
```
Min_range(G, s, t):
        for each node v in V:
              best_len[v] = infinity
        best_len[s] = 0
        init empty priority Q
```

```
for each v in V:
      insert (v, best_len[v]) into Q

while Q is not empty:
      u = extract_min(Q)

      if u = t:
            break

      for each edge (u,v):
            new_best_len = max(best_len[u], len(u,v))

            if new_best_len < best_len[v]:
                  best_len[v] = new_best_len
                  decrease_key(Q, v, new_best_len)

return best_len[t]
```

This algorithm works by keeping track of the worst weight along the all possible paths, and we want to keep track of the path with the least worst weight. We do this by creating the best_len for each node which is the best worst weight. We then loop through all the nodes and do this by the min of cap, and go through it's neighbors and if we find an edge to it with a least worse edge than the current, we update it and add then decrease the cap key in the queue. For the analysis, extract_min runs in O(logn) time, and in the worst case we would have to extract every vertex at most one time. We then get that this would cost O(|V|log|V|). We also have our decrease_key method which runs in O(1) time. Our worst case scenario for this is that we see every edge once and each time it needs to be decrease. This would give us O(|E|). So with these combined we get an O(|V|log|V| + |E|) algorithm.

4) Used ChatGPT 5.1 for help

For this problem we would want an algorithm that finds a path from u to w and passes through v and only uses each vertex and edge once. To do this we need a subpath from v to u and a subpath from v to w. Then for every vertex we want to split it up into two different vertices such that we have an in vertex and out vertex connected with a directed edge from in to out and then if the vertex is not v give it capacity 1, and if it is v then give it capacity 2 since they can share v but not any other node. We then want to take all of the edge (x, y) and convert them into directed edges with a capacity. This would look lik x_out -> y_in and y_out -> x_in. We would then give these a capacity of 2 since we have 2 paths. We then want a stop point from u and w so we add a sink (t), where we connect u_out and w_out to t

with a capacity of 1 so only 1 path can go to it. We then set the source of the flow to be v_in since we want both of the paths to start at v and reach u and w. We could then run a max-flow algorithm on this network from v_in to t. If the max flow is equal to 2 then there are two disjoint paths from v to u and from v to w since u and w would be passing in 1. If it is less than 2 then there is not two paths.
Analyzing the algorithm. We first split up all of the vertices into x_out and x_in, this would require O(|V|) time since we have to visit all of them. We then go to each of the edges and and add x_out into y_in and vice versa. This would be O(|E|). We then add these together and get our construction runtime of O(|V| + |E|). For our maxflow we could then use Ford-Fulkerson algorithm, we would augment the path by increasing the total flow by at least 1 unit. Our max flow value is at most 2, so we would only perform at most 2 augmentations. Our iteration's cost O(|V| + |E|), so this would give us O(2(|V| + |E|). We can combine these two steps together and get our runtime of O(|V| + |E|).

5) Used ChatGPT 5.1 for help
So for this problem we know that each domino covers two adjacent undeleted squares. So here we could treat this as vertices are our undeleted squares, and the edges are two squares that are adjacent that a domino could cover. We then want to build the board, so we would give each square a label of either 'black' or 'white' if it is even or odd. We would then want to create a graph this is split up into two section, one section being all the undeleted black squares. Then the right side being all undeleted white squares. Then for every pair of adjacent undeleted squares, we would add an edge if one is black and one is white. We would then make a directed graph with a source node s, the sink t, and a single node for each undeleted square. We would want to then connect s to every black vertex with a capacity of 1. And then from every white vertex we connect it to t with capacity 1. We use 1 to ensure they can only be used once. For every adjacency edge between a black square and white square, we also add directed edge from black to white with capacity of 1. This flow from source to black to white to sink, corresponds to choosing the edge (black, white).
We then want to determine if a perfect tiling is possible or not. If the number of tiles is odd we immediately know that it is impossible since each space takes up 2 spots odd numbers won't work. Also if number of black tiles != number of white tiles, this also won't work for the same reason. We then run a max flow from s to t, and if the result is the total number of nodes / 2, then this means every node can be covered and we can return true.
Analysis: Here our board is an n x n board so we will have at most n^2 squares which I

will just call $N = n^2$. If we use Ford-Fulkerson for the max flow algorithm, this has a runtime of O(E' * maxFlow). Here our E' is equal to O(N) because we have all of the undeleted squares. Here we said our maxflow would be the number of tiles / 2. Our number of tiles is N and O(N/2) = O(N). This means we get $O(N * N) = O(n^2 * n^2) = O(n^4)$