Collaborators in Question 3: Manisha Dodeja, Mahima Pawar, Pranjal Patil

Question 1

(a) Algorithm description
Converting the given problem into graph problems.

(a) What are the vertices? What does each vertex represent? The vertices represent trailheads (which are all accessible by vehicle), trail junctions, and landmarks (e.g. lakes, summits). Let there be total of V vertices. Our friends are on one of the vertices- S.

(b) What are the edges? Are they directed or undirected? The edges represent the paths between the above mentioned vertices. Let these be represented by E. The edges don't cross and there are no cycles in the edges. Since the paths are not directed, we can assume that the Edges are undirected.

(c) If the vertices and/or edges have associated values, what are they? Each vertex has a probability associated with it which indicates whether an avalanche would occur on that vertex when visiting it. Similarly each edge is associated with a similar probability number indicating the avalanche occurence chance while traversing it.

(d) What problem do you need to solve on this graph? We need to find the path from our friends location (S) to any of the nearby trailheads such that the expected number of avalanches while traversing the path is minimum.

(e) What standard algorithm are you using to solve that problem? Explain how its solution solves your problem.
We are going to use Djikstras Algorithm to get the single source shortest path from the given source to all of the other vertices. In the end we will select the path that has the minimum expected number of avalanches.
Given the probablity of an event we can calculate the expected number of the event happening by multiplying No. of repetions of experiments * P(E). Here, Since the event(Traversing a path or Visiting a vertex) happens only once per edge/vertex. We would simply add the probabilities along the way to calculate minimum path.
Let u and v be any two vertices in the graph, p(u,v) be the probablity on the Edge E between the two and p(v) be the probability on the destination vertex. Then our relaxation code for Djikstras would be:
dist(v) $<-$ dist(u) + p(u,v) + p (v)
pred(v) $<-$ u
The strategy is quite simple, if an edge is tense then we just relax it with the above formula.

(b) Consider the following algorithm.

Algorithm 1: Determine if a Solution exists for the unlimited Knapsack problem

Input: An array A[1..n], containing n items
Knapsack of size M
Output: Is a solution possible or not (1 is yes, 0 is No)

```
/* Here's my algorithm.  */
```

```
Djikstras(S, E, V){
    for each vertex v in V do
        dist[v] <- Infinity
        pred[v] <- null
    end
    dist[S] <- 0
    Insert(S,0) to priority Q

    while(priority Q is not empty) do
        u <- ExtractMin() from Priority Q
        for all edges u->v do
            if(u->v is tense) then  //if dist[v] > dist[u] + p(u,v) + p(v)
                dist[v] <- dist[u] + p(u,v) + p(v)      //relax
                pred[v] <- u
                Update (v, dist[v]) to Priority Q
            end
        end
    end

    i <- 1
    min <- Infinity
    for each vertex v in V do
        if(v is a Trailhead) then
            if dist[v] < min then
                min <- dist[v]
                dest <- v
            end
        end
    end

    Stack.push(dest)
    while (dest != S) do
        Stack.push(prev[dest])
        dest = prev[dest]
    end

    print Stack

}
```

(c) Running time analysis.
There are V vertices so heap implementation for vertex removal from Queue takes time O(V log (V)) and similarly updation takes O(E log(V). the minimum edges can be V-1 but we consider upper bound. Hence Time complexity is O (E*log(V)).

Question 2

(a) Prove that the minimum spanning tree of G contains every reliable edge

We will prove this based on the principle that Spanning trees are connected sub-graphs containing every vertex.
Let T be the minimum spanning tree and e indicate a reliable edge.
If an edge is reliable (it is not contained in any cycle of G), this means that the edge is the only connecting edge between two vertices. If we disregard e from the spanning tree T, our tree would be disconnected at the point where we removed the edge as it was the only connecting edge. And by definition (as mentioned above) A spanning tree connects every vertex of the graph, removing e would contradict the statement. Also if there exists an $e^{'}$ that connects the two disconnected vertices other than e and our new spanning tree $T^{'}$ can use $e^{'}$ to become a spanning tree, then it contradicts the fact that e was a reliable edge as e and $e^{'}$ would have formed a cycle between the two vertices.

(b) Prove that the MST of G does not contain any treacherous edge

We will prove this based on the proof that we did in class - A minimum spanning tree contains every safe edge.
Let T be the minimum spanning tree and e indicate a safe edge between two vertices u and v. Suppose there exists an edge $e^{'}$ such that w($e^{'}$) > w(e). We can say that the two edges form a cycle between u and v and $e^{'}$ is the treacherous edge in the cycle. Suppose our new spanning tree $T^{'}$ contains $e^{'}$ instead of e. In this case, our new spanning tree $T^{'}$ has a higher weight than T which itself contradicts the statement that $T^{'}$ is the minimum spanning tree. Hence, A minimum-spanning tree would never contain a treacherous edge in a cycle and we can take a smaller weight edge in the same cycle(safe edge) to be part of the minimum spanning tree with the graph being connected.

(c) Algorithm description. The concept is similar to reversing Kruskal's algorithm for finding the minimum-spanning tree. Here we sort the edges in descending order and start with picking the highest-weight edge. If by removing this edge from the graph our new sub-graph stays connected then delete the edge else keep the edge. Do this till all edges are covered or we have v-1 edges remaining in the graph.

Algorithm 2: Calculate the minimum spanning tree from a graph by removing treacherous edges

Input: A set of all the edges in the graph
Output: Minimum spanning tree)

```
/* Here's my algorithm.  */
```

```
MSTRemoveEdges(vertices V, Edges E){

    Sort E in descending order
    for each v in V
        create Matrix to keep track of neighbours

    Let i <- 1
    while(i < e.size) do
        uv <- ith largest edge from E
        remove v from u's matrix entry
        remove u from v's matrix entry

        if(connected) then          //perform DFS to check (returns true if connected)
            remove uv from E
        else
            add v to u's matrix entry
            add u to v's matrix entry
        end

    end

    return E
}
```

Time Complexity: It takes E log(E) time to sort the edges, V for iterating over each vertex, we are performing DFS at most E times. Hence O (E * (E + V)) for that. The upper bound time complexity becomes E*E.

Question 3

(a) Algorithm description
Converting the given problem into graph problems.

(a) What are the vertices? What does each vertex represent? Each region where the player currently resides can be considered as the vertices in this graph problem.

(b) What are the edges? Are they directed or undirected? Edges are the paths from a Vertex to its neighboring nodes such that it can send meeples along these paths. The edges will be undirected as a vertex can send meeples to any of its neighbors.

(c) If the vertices and/or edges have associated values, what are they? Each vertex (region) has the number of meeples currently present there. The meeple count is the value associated with each vertex.

(d) What problem do you need to solve on this graph? We want to move our meeples so that the minimum number of meeple in any of your vulnerable regions is maximized. This means that if we consider our vulnerable regions as the sinks we need to flow max meeples to these sinks to strengthen them. The constraint is that each region should have minimum of one meeple present.

(e) What standard algorithm are you using to solve that problem? Explain how its solution solves your problem.
The basic idea is similar to Edmonds-Karp algorithm. We can assume the sink to be the most vulnerable region initially. And our source can be the region with most meeples. So we first find the shortest path from source to sink using BFS and then perform max flow with the constraint that a vertex can send only those meeples that they contain. Also we will follow the constraint that each vertex should contain atleast 1 meeple. Now the question is How many meeples are enough in the vulnerable regions? we need to maximize this at the boundaries. So to make this procedure efficient we can find a target meeple count that the vulnerable vertices can achieve. We will use binary search to find this target. Our low will be zero and our high will be (Total no. of meeples from all regions - no. of non-vulnerable vertices) / vulnerable vertices. This would give us the max meeple count that a vulnerable region can contain. Now we can start our search at the above mentioned value. Perform max flow using Edmonds-Karp algo and check if its possible to attain that number. Keep on doing so for all the vulnerable regions untill we receive our final value that indicates the minimum number of meeple in any of our vulnerable regions maximized. If we are unable to attain then we go to that respective side of binary tree and start for a new target. Keep doing this until we are able to localize our maximized value.

(b) Running time analysis.
We perform Edmonds for all vulnerable regions(sinks) making the time complexity to O(V*V*E*E) and if we apply our binary search logic. Our final time complexity will be bounded by $O(V^2 * E^2 * log(t))$ where t is the max meeple on vulnerable regions.