1.

(a) Aprove

Given that we need to calculate the MST, we can use Kruskal's Algorithm to do that.

In Kruskal Algorithm, every step we choose the minium edge, and check whether the edge make the result become a cycle, if it isn't, then add the edge to the result. After traversal all of the edge, we get the MST.

Kruskal can prove the statement. For the first step, the result is null, so when we choose the minium edge, it must not make result become a cycle, so we can say that the minimum edge must exists in MST. When it comes to multiple minimum edges, MST will have at least one minium edge due to the first step.

(b) Disaprove

Example : imagining a graph having node A,B,C and edge AB = 1, AC = 1, BC = 2. The MST will only have 1 result that is {AB,AC}!

Reason : Considering about this situation, when the Kruskal is doing on a specific stage, there are 2 edges with the same values (we can call it e1 and e2), if the sequence is important ( which means that if e1 in the result then e2 will make the cycle, and if e2 in the result then e1 make the cycle), then it has multiple MST because e1 and e2 are conflicting to each other, otherwise it only make 1 MST.

(c) Using Prim Algorithm to deal with that.

In Prim Algorithm, start at a random node, and keep 1 list or array to store the result, value is the distance of node and the parent of the node(could be a list of tuple(parent,weight)) when connecting to the MST. Also, maintain a minimum heap to store the minimum distance from the unchosen node to the result (each node store the distance and the node name). The initial value of the list is all positive infinity, meaning that no node is in the MST, and the heap have no value.

At the beginning, randomly choose a node, build a heap to store the the minimum distance. On the first step, set result[first_node] = (null, 0), then check all the edge of the node, put them all in the minimum hap.

At each step, pop the minimum element from the heap.

Finding that whether the node is in the result or MST.

- If it is, continually pop the minimum element until we find the node that isn't in the result or MST. Then, put the node and distance to the result and check the edge from the node.

  Assuming the node call u, and the besides node v has:

  - v is not the parent of u in MST
  - v in the MST (result)
  - result[u] = the distance between u,v

  When it happened, we just return the result : MST is not unique.

Else, if the node v has:

- v is not in the MST

put v and the distance between u and v into the heap, and retotate the heap.

Otherwise we do nothing.

If we traversal all the node, and put them into MST, which means MST is unique.

(d) The Algorithm is correct.

The Algorithm is based on the observation that:

Given a temporary MST, if a node u will be added to the MST and there are several minimal path that can be added to MST, the final MST should be multiple.

For example, the node v is a new added node, and u, k is the node in MST.

The distance from v to u and v to k is the same and is the minimal.

That means that it is both correct to choose any of the edge, but we cannot choose them together, which makes the multiple MST.

In c), I use result tuple to store : the node in MST, the parent, the distance in inserting operation. And use heap to store the current minimum distance that out of MST. Every step we pop the minimum element in heap, in order to find the next node that will be insert into the MST. Then we traversal the edge of that node for finding whether there are another edge connecting to the MST having a same value as the current one. When it is, we can say that there are multiple MST, and after getting the final MST without any question, we can say that the MST is unique.

(e) Time comlexity is O(mlogn)

There are several step in this procedure.

- Initalization
  Taking O(n).
- Insert heap
  Every step takes O(logn) and given that we need to put all of the edge in it for at least 2 * m times, it is O(mlogn).
- Pop the heap
  Every step takes O(logn) and given that we need to pop all of the edge in it for n times, it is O(nlogn).
- Traversal all the result[] and check the duplicate minimal edge.
  Every step takes O(1) and m times, so it takes O(m)

Overall, the Time complexity is O(mlogn + nlogn), given that it is a connected graph, m should >= n – 1, it takes O(mlogn).

2

(a)

Considering a graph, where every node store : (x ,y ,v_x, v_y, temp_step), where x, y is the position, and v_x, v_y is the current speed, temp_step is the current step. According to the statement, we can make v_x + 1 or -1 or do nothing, v_y + 1, -1 or do nothing, so it totally have 9 forward situations. So the next generation of the node have 9 children. We can generate the graph step by step and the minimum temp_step is the right answer.

Firstly, initialize all the variable.

- min_step = positive infinity : the minimum step from start to finish.
- visited_array : a 4D array indexed by (x, y, vx, vy), which is n*n*n*n, all initialize None, recording the min_step to the node.
- node_list : the list of the starting node, could be several. Scanning all track[x][0] == True, and for each such cell, we initialize a node (x, 0, 0, 0, 0) and add it to the initial queue. These are our possible starting positions. Also, set all of the (x,0,0,0) in visted_array in 0.

Traversal all the node in the node_list, for each node u , there are 9 situations or 9 possible next steps, the new node u` generated by u will have

(x` = x + v_x`, y` = y + v_y`, v_x`, v_y`,temp_step` = temp_step + 1)

Where v_x` = v_x +1 or -1 or v_x, so does v_y`. So that's why there will be 9 possible next steps because we need to cover all the situations.

Then, check whether the new generated node u`(there are 9 nodes actually) is in the track, if it is True, the go traversal the next 9 steps of the node, else means it is not in the track, we skip the node (stop generating). Also, for the new node, we need to vist vistied_array throught the current (x, y, v_x, v_y), if the temp_step >= min_step, which means that it is not the best path to the current place, so we can stop generating new node on this path. Else, temp_step < min_step, continually generating and set index (x,y,v_x,v_y) in vistied_array as temp_step.

When the y = n - 1, which means that it go to the finish line, we stop generating new node through the node and check if it is in the track, if it is, we do min_step = min(node.temp_step, min_turn), else we do nothing. When x or y > n - 1, meaning that it go out of the track, we do nothing and stop generating.

When we traversal all the possible situations, we return the min_step, which is the answer. If min_step = positive infinity, which means that the final line is not reachable, we return -1.

(b)

The algorithm is correct due to it traversal all the possible result or path to the end. Every situations, there will be 9 possible next steps, we can generate 9 next nodes through a node and check whether it is in the track, if it is not, that means that the path is illegal, we don't need to continually generate new node in this path and go back to previous node. After successfully get to the finish line through a path, we check whether the path is the minimum number, if it is, we update the min_step to make sure the min_step is minimum. The visited_array make sure that the

potential higher number of turns will be pruned, the choose of n*n*n*n array is that the speed and also position cannot bigger than n, if the speed is bigger, the step will be overshoot.

(c) The time complexity of the algorithm is O(n^4)
Other operation like min(), generate new node all takes O(1), the main dependence on time complexity should be the times of generate new node. We can see that every node generate 9 next node, and the stopping generation operation should be :

- Node reach the end.
- Node outshooting.
- Node is in the visted and it is not the best step to this position.

The times of generate new node depend on the times of statement, which is (x, y, v_x, v_y). There are maximum n^2 on x,y because that x,y is ranging from 0 to n, otherwise it will overshoot and v_x, v_y have totally n^2 times time complexity because the range of v_x and v_y is (-n, n), which will also overshoot in the next step if it do not in this range. So the time complexity is O(n^4).

3.

a)

Running 2 Dijkstra Algorithm to Sydney and Tamworth.

Dijkstra Algorithm:

- Firstly, maintain a min heap for the distance to the shortest road, we can call it PQ, and the result list to store the path.
- The initial PQ is the neighbor of the starting node, the initial result list is the starting node.
- Every time remove_min() from the PQ, and if it is not in the result list, add it in the result list and add all the neighbour and the distance in the PQ. If it is in the result list, then do nothing.
- When result list contains all the node, we finish the Algorithm and return result list.

The shortest path to Sydney we call it p_sydney and the shortest path to Tamworth we call it p_tamworth. We can get P through p_sydney[p_tamworth] or continually get parent() in p_sydney.

Result list initial value is positive infinity and length is the number of edge in P. The index is the position of P, for example, P (Sydney -> A -> B -> Tamworth), the index of Sydney -> A is 0 and A -> B is 1…

We can initialize a result list, also a bridge list, which list all the edge, where

- (u,v)
- u is node of P and v is node of P.
- (u,v) is not in the P.
- Index u < index v.

(u,v) also the can be changed to a range (x,y), which means that the edge that the bridge is covered.

Then, traversal all the bridge in bridge lists. we can update the result list for i ranging from x to y by result[i] = min(result[i], p_sydney[u] + l(u,v) + p_tamworth[v]).

After traversal all the edge in bridge list, we return the result.

b) The algorithm is correct.

The algorithm first use the Dijkstra to calculate P and also the distance from Sydney and Tamworth for the following steps.

Then get the P, the shortest path from Sydney to Tamworth through the result of Dijkstra. Then we can traversal all the edge to find the bridge, which bridge means the node in P but can cover other node, so when we remove the edge, we can go to the destination through the bridge. We can traversal the bridge and update the value of the result list that the bridge has covered.

c) Time complexity

It is O(|E|log|V|).

Dijkstra Algorithm takes |E| log|V| time, we traversal all the node take |E| time, and every step contains remove_min(), which takes log|V|.

Then calculate the bridge takes O(|E|) time to traversal all the bridge.

After that, traversal all the bridge, assuming there are M bridge and M << |E|, at the worth case, everytime the bridge cover |E| edge, so it takes M*|E| which is O(|E|) at all.

We add them together to get a final time complexity which is O(|E|log|V|).