A4_Report_120090542

Question 1.

1.1 Solution & Implementation

We can view the question from the aspect of shortest path problem. Ideally, there is a direct path from i to j. In this case, the number of changes needed is 0. If, the number of changes is 1, then there must exist one path that need to change one direction and no path can be made without any change. This motivates us to assign cost (weight) 0 to the direction (directed edge) which has the same orientation as required while assign 1 to other orientations. In this sense, to find the number of changes made, it is equivalent to find the shortest path (hence minimizes number of changes) of a weighted graph. It is natural to think of the Dijkstra's Algorithm.

For our graph object, we initialize the attribute pointingto by a n*m+1 vector with value to be -1 ((n * m) + 1, -1). This indicates that there is no direction has cost 0. Note that for every node, it will have at most one direction which has cost 0. Therefore, we use the vector to record the index of the next node where the weight between the current and next node is 0 and all other adjacent nodes has weight 1. Note that from the starting point, all the eligible directions should have weight 0. We have the following code to determine the weight:

```
int w = (pointingto[u] == v || u == source) ? 0 : 1;
```

To implement Dijkstra's Algorithm, we create a min-heap Q. Besides, to save the space, we do not explicitly store the neighbor nodes. Instead, we define a function

```
std::vector<int> getNeighbor(int index) {
        int m = nrow;
        int n = ncol;
        std::vector<int> result;
        if (index % n != 1) {result.push_back(index - 1);}
        if (index % n != 0) {result.push_back(index + 1);}
        if (index > n) {result.push_back(index - n);}
        if (index <= n * m - n) {result.push_back(index + n);}
        return result;
    }
```

This will return the neighbor nodes of each node. For each node, there is at most four neighbor nodes. Therefore, this operation will not cost much time.

Consider the ith line of the input and j th element, this represents the orientation of node with index index = i * n + j + 1. As discussed before, this node will have at most one eligible orientation. Therefore, we find whether it the orientation is legal. If so, we set pointingto[index] to be the pointing node index. If not true, we do not change and -1 means as long as the path passes through this node, the cost will increase by 1 (change the orientation).

We initialize the attribute shoetestd (typo in code) to be infinity and starting from the source node. Now, having the basic settings, we can do the Dijkstra's Algorithm:

1. push the source node into the heap. The element in heap is of shape (x,index), where the first element represents the shortest distance to source and the second index is the index of the node.
2. While the Q is not empty, we pop from the heap. For the children of the heap, we check whether $d(v) > d(u) + w(u, v)$, where w is defined before. If so, we update $d(v)$ to be $d(u) + w(u, v)$ and push v and the current shortest path value into heap for further implementation.

```
int STPA() {
        shortestd[source] = 0;
        MinHeap Q;
        Q.push(0, source);
        while (!Q.isEmpty()) {
            int u = Q.pop().second;
            visited[u] = true;
            for (int v : getNeighbor(u)) {
                int w = (pointingto[u] == v || u == source) ? 0 :
1;
```

```
                if (shortestd[v] > shortestd[u] + w) {
                    shortestd[v] = shortestd[u] + w;
                    Q.push(shortestd[v], v);}}}

        return shortestd[end];
    }
```

The output is the shortest path, which is the smallest changes needed as discussed before.

1.2 Advantages

1. The use of Dijkstra's Algorithm indeed helps to concretize the original abstract problem
2. Unlike the traditional graph, we do not store the neighborhood explicitly. Instead, we calculate them manually each time we need to get children. This is due to the limited number of children (at most four). This enormously decreases the memory usage without sacrificing the time.

Question 2.

2.1 Solution & Implementation

To deal this program, essentially we use BFS twice for the graph. Firstly, the graph structure is given by

```
def __init__(self,n_vertex):
        #initialize the graph so that index i is the node with
value i+1 in the array
        self.array = [None]
        for i in range(1,n_vertex+1):
            n = Node()
            n.value = i
            self.array.append(n)
```

For index i in self.array, it holds a node, and the node has three attributes: color (to check whether visited/visiting/to be visited), value (hold the value), and neighbor_index list to hold the neighborhoods. The trick to ensure performance is to maintain ordered list for the neighbor_index list. That is to say, each time we add a pair of link, based on the sorted list, we find the position it belongs to and insert at the correct position as below:

```
def add_pair(self,a,b):
        if b not in self.array[a].neighbor_index:
            posa = 0
            lena = len(self.array[a].neighbor_index)
            while posa < lena and
self.array[a].neighbor_index[posa] < b:
                posa += 1
            self.array[a].neighbor_index.insert(posa, b)
            posb = 0
            lenb = len(self.array[b].neighbor_index)
            while posb < lenb and
self.array[b].neighbor_index[posb] < a:
                posb += 1
            self.array[b].neighbor_index.insert(posb, a)
```

Therefore, to traverse the children node in ascending order, we no longer need to sort the child list. Besides, the sorted list could help us to find the multiple pairs much faster. Indeed, given a sorted list, we want to find all the pairs (u, v) such that $u = kv$ or $v = ku$, we can do:

```
def find_pairs_ordered(lst, k):
    result = []
    i, j = 0, 1
    while j < len(lst):
        u, v = lst[i], lst[j]
        if u == k * v or v == k * u:
            result.append((u, v))
        if v <= k * u:
            j += 1
        else:
            i += 1
    return result
```

The order guarantees that there is no missing, which takes $O(2n) = O(n)$ time. For unordered list, this takes $O(n^2)$. Therefore, the order of child list is of great importance.

Now, we have the graph and preparations, we can do edge-adding. To do this, we use BFS.

```
def bfs_edge_add(self,s,k):
        Q = Queue()
        start_node = self.array[s]
```

```
        Q.enqueue(start_node)
        start_node.color = 1
        while not Q.is_empty():
            v = Q.dequeue()
            children = v.neighbor_index
            eligible_list = find_pairs_ordered(children,k)
            for pair in eligible_list:
                a,b = pair
                if a!=b:
                    self.add_pair(a,b)
            for index in children:
                sub_node = self.array[index]
                if sub_node.color ==0:
                    Q.enqueue(sub_node)
                    sub_node.color = 1
            v.color = 2
```

Firstly, we use queue to hold the node that is to be visited. Starting from node 1, we mark the nodes in queue using color==1, each time we dequeue one node and enqueue the children of the node. Besides, we will also search for the neighbor_index list of the node and detect the multiple pairs and then add an edge between the pairs. After traversal, we mark the color to be 2. We only enqueue nodes with color 0 to avoid repeating.

Now, we have finished edge adding, we use function clear_node() to change all the nodes' color into 0 for the second BFS. The second BFS is indeed starting from s and do the same thing as above.

2.2. Advantages

1. As discussed before, for every node, the neighbor_index list is maintained in ascending order. This ensures:
    a. No sorting needed to traverse the children
    b. Finding multiple pairs of the children list costs only $O(n)$ instead of $O(n^2)$.
2. By using BFS twice, we separately adding the edges and do BFS, both are of $O(|V|)$. This ensures the time complexity to be low and the algorithm to be readable.