CS180 HW7

Xilai Zhang

TOTAL POINTS

100 / 100

QUESTION 1

1 Problem 1 25 / 25

✓ - **0 pts** **Correct**

- **10 pts** failed to union points
- **10 pts** run time not correct
- **20 pts** wrong answer but showed efforts
- **25 pts** wrong answer or no answer

QUESTION 2

2 Problem 2 25 / 25

✓ - **0 pts** **Correct**

- **10 pts** failed to transfer the problem space from G to G'
- **10 pts** failed to use max flow min cut
- **20 pts** wrong answer but showed efforts
- **25 pts** wrong answer or no answer
- **0 pts** Click here to replace this description.

QUESTION 3

3 Problem 3 25 / 25

✓ - **0 pts** **Correct**

- **25 pts** No answer found

QUESTION 4

4 Problem 4 25 / 25

✓ - **0 pts** **Correct**

- **6 pts** Need more explanation
- **4 pts** Fail to prove time complexity
- **25 pts** No answer found

1.
Sort all flow based on their magnitude. We will later process them in descending order.
Initialize an array A of length 2n that contains the sorted start and ending times. Each entry of A is associated with a specific flow.
Each component of A contains the value of the break point and a pointer that points to another component.
Initially all components in A point to itself.
Initialize an array B of length 2n that document the size of each component, initially all element of B has size 1.
Initialize an array C of length 2n that document the endpoint of each component, initially all element of C ends at itself.
Initialize an array D of length 2n that document the start point of each component, initially all element of D starts at itself.
We define a function Find() that takes in a break point and returns the component the breakpoint belongs to.
We define a function Union() that unions components together.
For each interval being processed in descending order of flow:

```
    Let component u= Find(start point)
    Let component v= Find(end point)
    If u==v:
        Continue;
    Else:
        Follow the pointer of u, for each break point s in between u and v in
        A starting from the endpoint of u and before the start point of v:
            let t= Find(s)
                If t points to itself:
                    Set the value of t to be the value of current flow being
                    processed. Union(u,t).
                Else:
                    Perform union(u,t). Jump to the endpoint of t.
```

```
Function Find( break point w)
{
Follow the pointer of component w, until we reach a component x that points to itself.
Return component x;
}

Function Union ( component c, component d){
If c==d:
    Return;
Let e= size of component c found in array B.
Let f=size of component d found in array B.
Let the pointer of the component with a smaller size point to the component with a
```

bigger size. Update array B to be the sum of the size. Update array C to be the later of the endpoint. Update array D to be the earlier of the start point.
}

Correctness:
The correctness of this algorithm is based on union find structure. For each break point being processed in descending order, if it already belongs to a component of size more than one, its value is already at maximum, we simply perform union. If we encounter a break point whose value has not yet been set, in other words, whose component only contains itself, we set its value to the current flow value and then perform union.

Complexity:
Sorting the values will cause O(NlogN). Since each time we union, we make the component of smaller size pointing to the component with a bigger size, the size of the smaller component is increased at least by 2. Thus when we perform a find operation, the number of times we need to follow the pointer equals the number of time the smaller component is being incorporated by a bigger component. We know this incorporation can happen at most logN times because each incorporation increments the size by at least 2. Thus for each find operation the complexity is bounded by O(logN). Since each time we perform a union, the total number of components is reduced by 1, given there are n components at the start, we can perform at most n union operations. In each union operation, we only change the pointers and update array BCD, which is O(1) complexity. In total union operations will cost O(N) complexity. Now for each time interval being processed, we need 2 find operations for its start and ending time. This will need O(NlogN) complexity. For each component in between the start and ending time, we perform a find and a union operation. This will happen at most n times because each time we union the total number of components is reduced by 1. So in total this part is bounded by O(N)*(O(logN)+O(1))=O(NlogN)+O(N)=O(NlogN). So combining the two parts we will have the complexity to be O(NlogN)+ O(NlogN)= O(NlogN).

1 Problem 1 **25 / 25**

✓ **- 0 pts Correct**

- 10 pts failed to union points

- 10 pts run time not correct

- 20 pts wrong answer but showed efforts

- 25 pts wrong answer or no answer

2. We can prove Menger's theorem the same way as the book does.
Let us first look at the case of directed graph.

Let F be the set of edges such that they are the minimum number of edges to disconnect s and t. If the removal of F separate s from t, then each edge disjoint s-t path must use at least one edge from F, and thus the number of edge disjoint path is at most F.

(1) If there are k edge-disjoint paths, then the maximum flow is at least k, because we can simply let each edge disjoint path carry a flow of 1.

(2) if the flow f is 0-1 valued flow of value v, then the set of edges that carry flow contains a set of v edge-disjoint paths.

To prove (2), we can trace an outgoing edge from s that carries flow, and there can be two situations: (i) we will reach t. (ii) we will reach a node v for the second time. If the first case happens, we get an s-t path. We then decrease all flow along this s-t path by 1, and induct for flow v-1 and v-1 edge disjoint path. If the second case happens, we decrease the flow value along the cycle to 0, and continue to trace the flow. Decreasing a flow to 0 in both cases will ensure the paths we get are edge disjoint. Since there are v such flows we can trace, we will get v edge disjoint paths.

Combine (1) and (2) we will get (3): There are k edge-disjoint paths in a directed graph form s to t if and only if the value of the maximum value of an s-t flow in the graph is at least k.

Thus we know the maximum number of edge disjoint path is the value v of the s-t flow. And as we have proved in class, the maximum value of the flow is the minimum capacity of an s-t cut. In our case, since each edge has capacity at most 1, the minimum capacity is the minimum number of edges we needed to remove to disconnect s and t.

Now if the graph is undirected, we replace each undirected edge (u,v) by two directed edge(u,v) and (v,u). We can ensure that the max flow in this case will not use both (u,v) and (u,v). If (u,v) and (v,u) are both greater than 0, we can decrease them by the smaller value of the two until one of them becomes 0. In this case, at most one of the two oppositely directed edges can cross from s-side to t-side, because if one crosses, the other must go from t-side to s-side. Thus the maximum number of edge disjoint paths in this case is the maximum value of the s-t flow, which is minimum capacity of the s-t cut, which is the minimum number of edges needed to remove to disconnect s and t.

Thus we have proved that in an undirected graph, minimum number of edges to disconnect s and t equals the maximum number of edge disjoint paths between *s* and *t*.

2 Problem 2 **25 / 25**

✓ **- 0 pts Correct**

- 10 pts failed to transfer the problem space from G to G'

- 10 pts failed to use max flow min cut

- 20 pts wrong answer but showed efforts

- 25 pts wrong answer or no answer

- 0 pts Click here to replace this description.

3.

We first represent the undirected graph by directed graph, replacing each undirected edge (u,v) by two directed edge(u,v) and (v,u). Now for each vertex x, we split the vertex into x1 and x2. All the incoming edges to x are connected to x1, and all the outgoing edges from x2 are connected to x2. We then add the edge (x1, x2) to the graph and give it capacity 1. Because the capacity on any (x1,x2) edge is at most 1, the incoming flow to any vertex x can at most be 1, and the outgoing flow from any vertex x can at most be 1. This guarantees us that the flow across any vertex x can at most be 1, namely, each vertex will be visited at most once given any flow. Now if we perform max flow on this new graph, according to Menger's theorem, the maximum number of edge disjoint path corresponds to the maximum number of vertex disjoint path, which is the value of the maximum flow, which is the minimum capacity of the s-t cut, which is the minimum number of edges needed to remove to disconnect s and t. Since each removed edge corresponds to a flow of unique vertices, it is also the minimum number of vertices needed to remove to disconnect s and t.

Thus the minimum number of nodes in order to disconnect two vertices s and t equals the number of vertex-disjoint paths between s and t.

3 Problem 3 25 / 25

✓ **- 0 pts** **Correct**

- 25 pts No answer found

4.

Since only one edge is incremented by 1, we can update the value in the residual graph to reflect this change. If we don't have a residual graph, we can create a residual graph G. If we already have the residual graph G, we simply increase the capacity of this edge by 1. Now we perform DFS starting from s to t, on the residual graph G. If an s-t path exists on G, we simply update all edge values along this path, if the edge is forward we increase the flow on the edge by 1, if the edge is reverse direction we decrease the flow on the edge by 1. This will give us the updated max flow.

Correctness:

We know this update requires at most one iteration on the residual graph. Since all the capacities are integer values, in each iteration we improve the max flow at least by 1. In this case the max flow can be incremented at most by 1, because we only incremented the capacity of one edge by 1. Thus if max flow needs to be updated, it will be done within one iteration on the residual graph.

Complexity:

As explained in correctness section, we only need one iteration of DFS. Thus the complexity is that of DFS, which is O(M+N). If we need to create a residual graph, the complexity to create a residual graph is also O(M+N). So the total complexity will always be O(M+N).

4 Problem 4 25 / 25

✓ **- 0 pts** **Correct**

- 6 pts Need more explanation

- 4 pts Fail to prove time complexity

- 25 pts No answer found