Hermmy Wang
704978214
Discussion 1B

Homework 7

1.

The idea is to first sort the demands in descending order of flow. Initialize an array of bp which stores the start time and end time of the sorted demands. For each entry in the array, the breakpoint is a pair which contains its time, and a pointer to the next entry. graph[i] stores a triple, which represents an interval like (start_time, end_time, flow).

For each demand, we first look at the start time to see if it is inside another interval with higher flow; if it is, we union the end breakpoint and all the points in between to this interval, because now we committed the graph between the previous interval and the current demand. Similarly, if the demand ends within another interval, we union its start time and all the points in between to this interval. Every time we union two breakpoints, we update the graph within the two breakpoints to the current flow for the part where has not yet been set. If a part of the interval has already been set, we skip that part by jumping to the next pointer of the interested breakpoint.

After a breakpoint is unioned into a set, the set is stored in a binary minheap tree in O(logn). Extracting the min value (earliest time of the interval) is constant.

Time complexity:

> Since insertion of a binary heap tree takes O(logn) time, unioning takes O(logn) time and finding the root that contains a breakpoint takes O(1). We at most union for n demands. The total time complexity is O(nlogn).

union(a, b, flow)

```
root_a <-- find_root(a)
root_b <-- find_root(b)
if root_a != root_b
      insert b to the rooted binary heap tree containing a
      //starts within another interval
      if root_a != a and root_b == b
            add(graph, root_a.next, b, flow)
      //ends within another interval
      else if root_a == a and root_b != b
            add(graph, a, root_b, flow)
      //starts and ends within two intervals
      else if root_a != a and root_b != b
            add(graph, root_a.next, root_b, flow)
```

```
        //not start or end within intervals
              add(graph, a,b,flow)
```

flow(demands)

```
    sort demands in descending order of flow
    initialize bp[2n] <-- start and end time for each interval from
    highest flow to lowest flow
    parent[2n] <-- 1, 2, 3, ..., 2n
    graph[] <-- (0,0,0)
    union(bp[0], bp[1])
    next pointer of bp[0] to bp[1] <-- bp[1]
    for i <-- 1 to n-1
          start_root <-- find_root(bp[2i])
          end_root <-- find_root(bp[2i+1])
          if start_root != end_root
                union(bp[2i], bp[2i+1], demands[i].flow)

    return graph
```

find_root(a)

```
    heap <-- parent[a] //set parent during insertion
    return heap[0]
```

append(start,end,flow)

```
    i <-- start
    for j <-- start to end:
          if find_root(j) != j and i != j  //encounter an interval
                graph.append(i, j, flow) //update graph
                i <-- j.next
                j <-- j.next
```

2. Menger's theorem (edge)

We need to prove that the minimum number of egdes to disconnect two vertices s and t is the maximum number of edge disjoint paths between s and t.

(1) Suppose J is a set of edge disjoint paths between s and t, and C is a set of edges to disconnect s and t. Every path in J must use an edge in C; otherwise, C can't disconnect s and t. Also, in order for J to be edge-disjoint, J cannot share one edge in C. By pigeonhole principle, we can conclude |J| <= |C|. (Every pigeon has to have a pigeonhole and no two pigeons share the same hole).

(2) In order to show |J| = |C|, we also need to show |C| <= |J|.

Replace G with parallel edges between every two vertices and assign each edge a capacity of 1. By assigning capacity to 1, we ensure that units of flow from s to t corresponds to edge-joint paths. If we try to find a max flow for G, then for every flow into an edge, the value must be 1. No two flows of value 1 can pass through the same edge, because doing so will exceed the edge capacity. Therefore, every path from s to t is mutually disjoint. Let f be a max flow in G. The max flow yields a set of paths from s to t that are edge-disjoint, since any edge has at most one unot of flow and therefore can appear in only one path. |J| is the maximum number of edge-disjoint paths in G. f <= |J|.

Let C be a minimum s,t cut so that C=[S,T] (S contains s, T contains t). capacity(C) is the number of edges in K because each edge has a capacity of 1. In other words, if we remove capacity(C) edges from G, s and t will be disconnected. Since this set of edges with capacity (C) disconnects s and t, it is also a edge-connectivity in the undirected graph. This implies that |C| <= capacity(C). By max flow min cut theorem,|C| <= |J|.

We have proved the inequality in both directions, |C|=|J|.

3. Menger's theorem (vertex)

(1) Suppose J is a set of vertex disjoint paths between s and t, and C is a set of vertices disconnecting s and t. Every path in J must use a vertex in C; otherwise, C can't disconnect s and t. Also, in order for J to be vertex-disjoint, J cannot share one vertex in C. By pigeonhole principle, we can conclude |J| <= |C|. (Every pigeon has to have a pigeonhole and no two pigeons share the same hole).

(2) We want to show that |C| <= |J|.

Transform G to G': Split each vertex u except for s and t into two vertices: u+ and u-. Between each pair of u+ and u-, there is an internal edge u- --> u+. If there is an edge between u and v in G, now the edge is between u+ and v- in G'. For each edge coming from s, the destination is a minus edge. For each edge going into t, the source is a plus edge. Therefore, plus edges always have outgoing internal edges except for the internal edge, and minus edges always have incoming edges except for the internal edge.

Let f be the max flow value. If there is a flow into u-, then the value must be 1 because every edge only has a capacity of 1. If the max flow is f, there are f units flowing from s and t, and these paths cannot share a vertex because sharing vertex overflows the edge capacity. Therefore, the max number of vertex disjoint paths |J| is bounded by max flow. |J| >= maxflow.

Let the minimum set of internal edges that disconnect s,t into S and T be C in G'. Between these two sets S and T, the only connection should be internal edges. Proof by contradiction:

Assume we can find a non-internal edge connected from s to t in the vertex-connectivity C=[S,T]. The non-internal edge has a capacity of at most n, then capacity(C)<=n.

However, suppose there is another vertex connectivity C'=[S',T']. S' contains s and all the edges coming out of s. T' contains the rest of the edges. Since there are at most n-2 neighbors of s (all vertices except for s and t), capacity(C') <= n-2.

We can find a capacity(C')<=n-2<n. Contradict the minimality of C. Therefore, all the egdes going from s to t in C must be internal edges. Let P be a st path in G such that s-->a-->b-->t. In G', P' becomes s-->a- -->a+ -->b- -->b+ --> t. At some point, the path passes from S to T by an internal edge (u- --> u+). From C, if we delete an internal edge like this, we have broken the path and s will be separated from t. Let all these internal edges form the set C. There are |C| internal edges like this with the property that removing them will separate s and t. Transforming back to G, for a vertex-connectivity U, there are at least |C| vertices such that removing them will separate s and t.

Since maxflow = mincut,

$$|C| <= \text{mincut} = \text{maxflow} <= |J|.$$

In conclusion, |C| = |J|. The minimum number disconnecting s and t equals the maximum number of vertex-disjoint paths.

4. Altered edge capacity

We can implement the idea of the Ford-Fulkerson algorithm and particularly look for the edge with the increased capacity using BFS:

1. Find the residual flow graph. Increase 1 capacity to the increased edge.

2. BFS in O(m+n) to see if there is any path in the residual flow graph with a possible increase of flow. If there is one, we can update the flow. If there is no flow in the residual flow graph, the original max flow is unchanged.

Time complexity:

We only need to do one iteration of the Ford-Fulkerson algorithm becase the augmenting path can only possibly increase the flow by 1. Since there is only one iteration and the algorithm uses BFS, which dominates the total complexity is O(m+n).

BFS(residual, s, t, parent):

```
visited[n] <-- false
push s to queue, set visited
while queue not empty
      j = pop front
      for every vertex i
            if unvisited and residual[j][i]>0
                  q.push(i)
                  set parent and visited
return visited[t]==true
```

FordFulkerson(maxflow, s, t, edge(i,j)):

```
parent[n] //filled by BFS
for every vertex(u,v)
      residual[v][u] = maxflow[u][v] //back edge
      residual[u][v] = graph[u][v]-maxflow[u][v]  //forward edge
residual[i][j]++
if BFS(residual, s, t, parent):
      for (v=t; v!=s; v=parent[v])
            u=parent[v]
            residual[u][v]--
            residual[v][u]++
      max_flow++
return maxflow
```