Solution 7

1. In class, we have roughly discussed how to implement the idea by Siddharth to solve the midterm water flow problem. His idea is creating the pressure function committing sections growing of it by considering intervals in descending order of flow. Whatever we committed will never be changed.

   We first create an array of the 2n breakpoints which are either the start of an interval or the end of an interval. We think of each entry of this array as a pair: A value of a break point, followed by a pointer to the next break point "of interest."

   Initially a pointer of entry j in the array points to break point in position $j+1$ in the array. And the graph now at the beginning is of height 0 between break points.

   The observation of Siddharth is that now we take the interval with the max flow demand say *maxflow1*. Say it starts at break point i and ends at break point $j > i$. Then the observation is that the final graph will have value *maxflow1* between break points i and j.

   So now, we would like to have the pointer at i point to j, and now continuing inductively. This will be easy if intervals are disjoint. But they are not. So the next interval in the order of flow with *maxflow2* may start say between i and j but end after at $k > j$. So now we would like to have the pointer in i point now to k because now we committed the graph between i and k.

   But how do we know that the beginning of the interval of *maxflow2* is in between i and j unless we "memorize" i and j which will be expensive!

   So, we want a mechanism that if a new interval has a start or a finish falling in a committed segment we can not too expensively find this segment.

   Here comes handy the Union-Find algorithm discussed in earlier homework (review). At the beginning, each break point is a tree consisting of itself. When we put in the first interval of *maxflow1* we Union all the break points between i and j, to a rooted tree that includes all of them. Now when i take the beginning of interval *maxflow2* I just follow that breakpoint to its root. Its root tells me the interval it belongs to. The idea is that when we Union we hang small height tree off the big one, so that going from any node in a tree to its root is less that $\log 2n$ steps.

   Develop this idea into an algorithm and argue its complexity is $O(n \log n)$. And if Siddharth worked the details of his idea correctly on the midterm...Prof Gafni will do an "end-zone NFL dance" for him.

   Solution:

   The problem is equivalent to the skyline problem. Assume each building i has the input (s_i, e_i, h_i), and the buildings are sorted by height in decreasing order. Let arrary I store the processed interval information for all $2n$ start points and end point., where $I[c] = (\text{start}_{I[c]}, \text{end}_{I[c]},)$, where c could be either a starting point or a end point of some building. And c is a representive of an interval $[\text{start}_{I[c_i]}, \text{end}_{I[c_i]})$ which the height is fixed in the skyline. If a point c is not associated with any interval, $I[c] = (c, c)$. We then sort all the starting points and end points in increasing order and put them into array $B[1..2n]$. Each $B[i]$ store either a start point or an end point, and it is equipped with a pointer to a later position in B. Initially, $B[i]$ points to iteself. When an interval $(\text{start}_{I[c]}, \text{end}_{I[c]})$ is fixed, we let the pointer for $\text{start}_{I[c]}$ points to $\text{end}_{I[c]}$. The array B is used when we scan and union points that falls in a building. If a point is not in a previous interval, we

union the point with the current interval. Otherwise, we find the root of the previous interval, and union that with the new interval. To ouput the skyline, we just add a skyline point every time we union. In the Union find datastrucre, every point is a node. Initially, every node points to itself. When we union some points into an interval, we will use some starting point in the interval as the representive node , and make it to be the roots of all other points in the same interval. So

```
SKYLINE(Building1..n)
for building i ← 1 to n
  start ← I[r_{s_i}].start, where r_{s[i]} is the root in Union-Find of s_i
    // try to find the start position of any overlapping interval.
  end ← I[r_{e_i}].end, where r_{e[i]} is the root in Union-Find of e_i
    // try to find end position of any overlapping interval.
    //we are processing new interval I_i with position [start, end),
    //the points in this interval will have a fix height in skyline
  if start = s_i    // s_i is not in a previous interval
      add skyline point (s_i, h_i)
  p ← the point after start in B    //a pointer used to scan all the points between [start, end)
  while p ≠ end
      r_p ← the root of p in the union find
      if I[r_p] is an empty interval    //the height of position p is not determined before
          add the skyline line point (p, h_i)
          union p and s_i
          move p to the right in B
      else
          union r_p and s_i    // union the interval of r_p and I_i.
          p ← I[r_p].end    //skip the points in interval I[r_p] since we already union the whole interval.
  r ← find the root point of s_i
  update I[r] = (start_{I[c]}, end_{I[c]}, )
```

The complexity is determined by the time cost for scan points and check intervals for a point. Every point will checked constant times since one it is inside an interval, it will skipped later. Each check of interval cost $O(\log n)$ time since the size of the union find datastructure is $O(n)$. Hence, with the sort cost, the total time complexity is $O(n \log n)$.

2. In an undirected graph $G = (V, E)$, the edge s-t connectivity is the minimum number of edges to disconnect two vertices s and t. Using max flow, prove that the s-t connectivity equals the maximum number of edge disjoint paths between s and t. (Hint: replace G by creating parallel edges and assign edge capacities 1. This observation is called Menger Theorem and will help you copying from the book or the internet if you are so inclined.)

   Solution:
   We construct a directed graph G' with same vertices as G, replace every edge $e = \{u, v\}$ in G by two parallel edges $e_1 = (u, v)$ and $e_2 = (v, u)$. With every edge capacity 1, we solve the maximum flow problem on $G', (s, t)$. The maximum flow f^* corresponds to edge disjoint paths between s and t in G since every edge has 1 capacity, there are f^* many unit flow such that they are edge disjoint. On the other hand, by max flow min cut theorem, there exits a cut in G' with capacity f^*. By the construction, the edges that contribute the cut capacity in G' corresponds to a cut in original G. Since the cut in G' is minimal, then the s-t connectivity is also f^*.

3. The s-t vertex-connectivity of undirected graph $G = (V, E)$, is the minimum number of nodes in order to disconnect two vertices s and t. Show that the s-t vertex-connectivity equals the number of vertex-disjoint paths between s and t. (Hint: split a vertex into two vertices)

   Solution:

   The statement is true. To prove it, construct a new graph G' from the existing graph G. Replace every vertex v (except s and t) of G with a pair of vertices v_1 and v_2, and add an edge from v_1 to v_2. Also for every edge (u, v) in G add an edge from u_2 to v_1 in G'. Every s-t path in G corresponds to an s-t path in G'. It's easy to see that the maximum number of vertex disjoint s-t paths in G is equal to the maximum number of edge disjoint s-t path in G'. Also the minimum number of vertices that need to be removed from G to disconnect s and t is equal to the minimum number of edges that need to be removed from G' to disconnect s and t. Thus the truth of the statement follows from the fact that the minimum number of edges that need to be removed to disconnect s and t in G equals the maximum number of edge disjoint paths between s and t.

4. Suppose we are given the maximum flow in a flow network $G = (V, E)$ with source s, sink t, and integer capacities. Suppose the capacity of a single edge is increased by one. Give an $O(n + m)$ algorithm for updating the maximum flow, where G has n vertices and m edges.

   Solution:
   Denote the maximum flow in G as f, the value of the maximum flow as $v(f)$, the edge with the increased capacity as e, and the modified graph as G'.
   We first show that the value of the maximum flow in G' is either $v(f)$ or $v(f) + 1$.

   Proof. (a) The value of the maximum flow in G' is at least $v(f)$ since f is still a feasible flow in G'.

   (b) The value of maximum flow in G' is at most $v(f) + 1$. By the Max-Flow Min-Cut theorem, there is some $s - t$ cut (A, B) in G with capacity $v(f)$. All the edges crossing (A, B) have the same capacity in G' that they did in G, with the possible exception of e (in case e crosses (A,B)). c_e only increased by 1, so the capacity of (A, B) in G' is at most $v(f) + 1$.
   It is also integer-valued. So the value of the maximum flow is either $v(f)$ or $v(f) + 1$.

   □

   To update the maximum flow for G', starting with the feasible flow f in G', we try to find a single augmenting path from s to t in the residual graph G'_f in O(m+n). If we fail to find augmenting path, we know that f is a maximum flow in G'. Otherwise the augmentation succeeds, producing a flow f' of value at least $v(f) + 1$. In this case, since the value of the maximum flow in G' can be at most $v(f) + 1$, f' must be a maximum flow. So either way, we produce a maximum flow after a single augmenting path computation. This takes O(m+n).