Massachusetts Institute of Technology
6.046J/18.410J: Introduction to Algorithms
Professors Piotr Indyk and Bruce Tidor
Handout 22
May 17, 2003

Problem Set 7 Solutions

Problem 7-1. Integral Flows

The first observation we make about the max flow network is that no vertex can have exactly one fractional flow edge incident on it. This follows from flow conservation and the integrality of the maximum flow. Thus, if we look at the subgraph formed by the fractional flow edges and interpret it as an undirected graph by ignoring the orientation of edges, each connected vertex has degree at least 2. This implies the existence of a cycle in the fractional flow subgraph.

Next, observe that the residual network corresponding to this cycle would be a a pair of directed cycles in opposite directions (see Fig 1). This follows from the fact that each edge (u, v) with fractional flow f_{uv} creates two edges in the residual network: (u, v) with capacity $c_{uv} - f_{uv}$ and (v, u) with capacity f_{uv}. Since f_{uv} is a fraction > 0 and c_{uv} is integral, both the forward and backward residual capacities are non-zero.

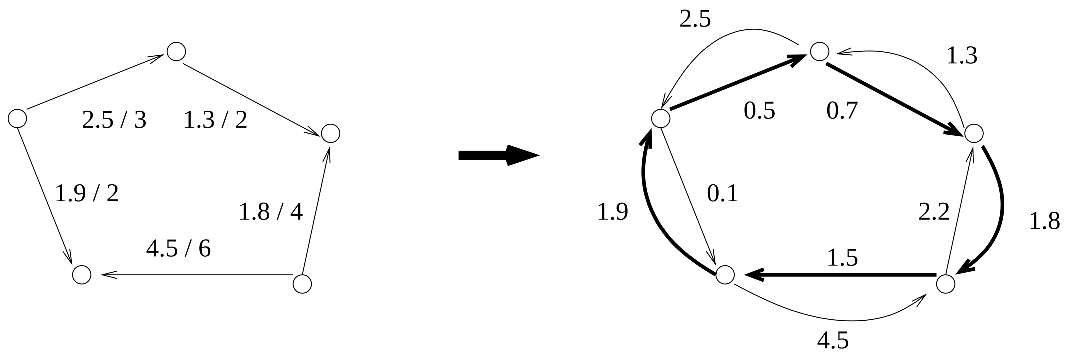

Figure 1: Fractional flow cycle and its residual graph

Once we have the residual network for the cycle, we can treat any one of the directed cycles as an "augmenting path" from a source to itself. Augmenting the flow along this path will make the flow on the critical edge integral. Furthermore, this augmentation leaves the overall value of the flow unchanged since the new flow coming into a vertex is exactly balanced by the new flow going out of it. Repeating this augmentation gives us the desired integral flow. The following algorithm formalizes the idea outlined above:

```
Convert_Flow(G = (V, E), f)
G' ← subgraph of G with fractional flow edges
repeat
        C ← find_undirected_cycle(G')
        C_f ← residual graph corresponding to C
        f' ← critical flow along augmenting cycle
        f ← f + f' ; Update G'
  until (No fractional flow edges left in f)
```

A cycle can be found in an undirected graph in $O(V)$ time. Each of the remaining steps $4-7$ requires time linear in the size of the cycle, which again is $O(V)$. Every iteration of the loop converts at least one fractional flow edge into an integral one. Since the number of fractional flow edges can be at most E, the overall running time of the algorithm is $O(VE)$.

Problem 7-2. Intersection Counting

Recall the algorithm described in lecture for **reporting** all intersection points between orthogonal segments.

- Sort all V-segments and endpoints of H-segments by their x-coordinates
- Scan the elements in the sorted list:
  - Left endpoint: add segment to T
  - Right endpoint: remove segment from T
  - V-segment: report intersections with the H-segments stored in T

The last step involves searching T for the endpoints of the V-segment, y_{top} and y_{bottom} (or to be more precise, the successor of y_{top} and the predecessor of y_{bottom}) and then reporting all the nodes that lie between them. Since in this problem we are only required to **count** the number of intervening nodes, we can speed up this step by augmenting each node of the BST with the size of the subtree beneath it (as in Section 14.1 of CLRS on computing dynamic order statistics). Since the number of intervening nodes is just $Rank(successor(y_{top})) - Rank(predecessor(y_{bottom})) + 1$, it can be computed in time $O(\log n)$. The overall algorithm therefore runs in time $O(n \log n)$.

Problem 7-3. Painting Rectangles

This problem can be solved in different ways. The simplest method utilizes a sweep line approach. Since all rectangles are lined up on the x-axis, we only need to keep track of the height of the currently tallest rectangle. The following pseudocode describes this idea:

Paint_Rectangles(x_1, x_2, y)
1 Initialize an empty BST T
2 $Area \leftarrow 0$; $Height \leftarrow 0$; $Length \leftarrow 0$;
3 Sort the x-coordinates of rectangles into a single array of events
4 **for** each event in the sorted list
5   **do** $Length \leftarrow$ distance between current and previous event
6         $Area \leftarrow Area + Height * Length$
7   **if** event is Left Endpoint of rectangle i
8   Add height of rectangle, $y[i]$ to T
9   **else**
10   Remove height of rectangle, $y[i]$ from T
11         $Height \leftarrow Max(T)$
12 **return** $Area$

The sorting in step 3 takes $O(n \log n)$ time. The steps inside the For loop involve elementary BST operations, viz., insertion, deletion and finding the maximum. Each of them takes $O(\log n)$ in the worst case. Thus, the overall running time of the algorithm is $O(n \log n)$.

This problem can also be solved in $O(n \log n)$ time using a divide and conquer algorithm similar to merge sort. Here is a brief sketch of the algorithm. Let's call the union of a subset of our rectangles a *skyline*, which will be represented by a sequence of the form $(x_1, h_1), (x_2, h_2), \ldots, (x_k, h_k)$ where $x_1 < x_2 < \ldots x_k$. The x values specify the coordinates where the height changes as we sweep from left to right across the skyline, and the new height value at each of these points is given by the h values. The key insight is that we can merge two skylines in $O(n)$ time, where n denotes total combined complexity of both skylines. The algorithm to do this is very similar to the merge algorithm from merge-sort, and depends crucially on the fact that the x-coordinates in a skyline are sorted. Once we have the merge procedure, the remaining algorithm is straightforward: we split our input into two sets of $n/2$ rectangles each, recursively compute skylines from these, and merge the results. The running time will satisfy $T(n) = 2T(n/2) + \Theta(n)$, which solves to $T(n) = \Theta(n \log n)$.

Problem 7-4. Graduate student party

We shall solve this problem in a way similar to the close pair problem done in lecture, namely, partitioning the plane into a square grid and hashing points to corresponding buckets.

1. Impose a square grid onto the plane where each cell is a $\sqrt{2} \times \sqrt{2}$ square.
2. Hash each point into a bucket corresponding to the cell it belongs to.
3. If there is a bucket with ≥ 6 points in it, return YES.
4. Otherwise, for each point p, calculate distance to all points in the cell containing p as well as the neighboring cells (see Fig 2 for description of neighboring cells). Return YES if the number of points within distance 2 is ≥ 5.

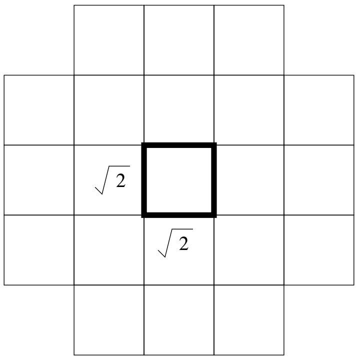

Figure 2: Neighboring cells for a given point

Clearly the algorithm will find a sociable person if one exists, either in step 3 or in step 4. By using a hash table of size $O(n)$, we can make the above algorithm run in expected linear time.