

Design and Analysis of Algorithms: Homework #6

Due in class on April 24, 2018

Professor Kasturi Varadarajan

Alic Szecei

Problem 1

For any flow network G and any vertices u and v let $bottleneck_G(u, v)$ denote the maximum, over all paths π in G from u to v , of the minimum-capacity edge along π .

(a) Describe and analyze an algorithm to compute $bottleneck_G(s, t)$ in $O(E \log V)$ time.

Solution

Part A

```

1: function BOTTLENECK( $G, u, v$ )
2:    $queue \leftarrow$  an empty heap
3:    $current \leftarrow u$ 
4:   for all  $vertex$  in  $G.vertices$  do
5:     if  $vertex = u$  then
6:        $vertex.weight \leftarrow \infty$ 
7:     else
8:        $vertex.weight \leftarrow -\infty$ 
9:     end if ▷ We'll use these weights in our heap
10:    insert  $vertex$  into  $queue$ 
11:  end for
12:  while  $\neg v.seen$  do ▷ Find the path with the bottleneck by building a tree
13:     $current.seen \leftarrow \text{True}$ 
14:    for all  $d$  in  $current.neighbors$  do
15:      if  $\neg d.seen$  then
16:        if  $edge(current, d).weight > d.weight$  then
17:          delete  $d$  from  $queue$ 
18:           $d.weight \leftarrow edge(current, d).weight$ 
19:           $d.parent \leftarrow current$ 
20:          insert  $d$  into  $queue$ 
21:        end if
22:      end if ▷ Update the vertex weight of a neighbor of  $current$ 
23:    end for
24:     $current \leftarrow$  dequeue the max vertex in  $queue$ 
25:  end while
26:   $c \leftarrow v$ 
27:   $min \leftarrow \infty$ 
28:  while  $c \neq u$  do ▷ Find the minimal edge in the bottlenecking path
29:    if  $edge_{c,c.parent}.weight < min$  then
30:       $min \leftarrow edge_{c,c.parent}.weight$ 
31:    end if
32:     $c \leftarrow c.parent$ 
33:  end while
34:  return  $min$ 
35: end function

```

This assumes the function should return the *weight* of the bottlenecking edge; however, it can trivially be modified to return the edge itself.

This algorithm is essentially a modification of Prim's algorithm, where instead of searching for minimally-weighted edges, we instead search for maximally-weighted edges. To do this, we use a priority queue (using a binary heap).

We can split up our algorithm into three sections. The first is where we add each vertex into the priority queue. This is a total of V insertions into the queue, each of which takes $O(\log V)$ time, for a total runtime of $O(V \log V)$.

Second, we build a spanning tree in G until we have found a route from u to v . To do this, we continually dequeue vertices from our queue. For each unvisited neighboring vertex, we check if we've found a better edge to connect it to our tree. If so, we update the weight of the vertex, and also update its "parent" in the tree. We continue in this fashion until v has been added to the tree. Each time we attempt to update a vertex, it means we have traversed a new edge in the graph; updating the vertex involves a successive deletion and insertion, both of which are $O(\log V)$. Since this can happen at most E times, the total running time for this section is $O(E \log V)$.

Last, we determine the smallest edge in the path from v to u . This is more straightforward, as it is a simple $O(E)$ loop.

Our total running time is thus $O((E + V) \log V)$; since V is $O(E)$ for any connected graph, we can say that it is $O(E \log V)$.

Problem 2

Suppose you have already computed a maximum flow f^* in a flow network G with *integer* edge capacities.

- Describe and analyze an algorithm to update the maximum flow after the capacity of a single edge is increased by 1.
- Describe and analyze an algorithm to update the maximum flow after the capacity of a single edge is decreased by 1.

Both algorithms should be significantly faster than recomputing the maximum flow from scratch.

Solution

Part A

```

1: function UPDATEMAXFLOW( $G, source, sink$ )
2:    $path \leftarrow \text{BFS}(G, source, sink)$            ▷ Assume this works similarly to original Ford-Fulkerson algorithm
3:   if  $path$  exists then                           ▷ Check if there's an augmenting path
4:      $r \leftarrow$  the minimum residual capacity of  $(u, v) \forall (u, v) \in path$ 
5:     for all  $edge$  in  $path$  do
6:       if  $edge$  is a forward edge then
7:          $G.flow(edge) \leftarrow G.flow(edge) + r$ 
8:       else
9:          $G.flow(edge) \leftarrow G.flow(edge) - r$ 
10:      end if
11:    end for
12:  end if
13:  return  $G.flow$ 
14: end function

```

The idea for this algorithm is straightforward. We assume the edge has already been increased before the algorithm is run; we then search for an augmenting path, which runs in $O(V + E)$ time using breadth-first search. If an augmenting path exists, we run through a single iteration of Ford-Fulkerson, updating edges along the path; this runtime is clearly bounded by $O(E)$. If an augmenting path does not exist, no alterations are required. Our algorithm thus runs in $O(V + E)$ time.

We only need to run through the Ford-Fulkerson algorithm at most once, since the maximum flow can only increase by one (the same amount as we increased an edge capacity). The augmenting path, then, must then increase the flow by 1, which satisfies our requirements.

Part B

```

1: function UPDATEMAXFLOW( $G, source, sink$ )
2:    $e \leftarrow \text{Null}$                                 ▷ We need to make our flow satisfy the new constraint
3:   for all  $edge$  in  $G$  do
4:     if  $G.\text{flow}(edge) - G.\text{capacity}(edge) > 0$  then
5:        $e \leftarrow edge$                                 ▷ Find an edge with negative residual flow
6:     end if
7:   end for
8:   if  $e = \text{Null}$  then
9:     return  $G.\text{flow}$                                 ▷ If the edge has non-negative residual flow, we didn't affect a bottleneck
10:  end if
11:   $path \leftarrow$  a path from  $source$  to  $sink$  which includes  $e$                                 ▷ DFS would likely be best here
12:  for all  $edge$  in  $path$  do                                ▷ Decrease the flow down the path from  $source$  to  $sink$ 
13:    if  $edge$  is a forward edge then
14:       $G.\text{flow}(edge) \leftarrow G.\text{flow}(edge) - 1$ 
15:    else
16:       $G.\text{flow}(edge) \leftarrow G.\text{flow}(edge) + 1$ 
17:    end if
18:  end for
19:   $path \leftarrow \text{BFS}(G, source, sink)$                                 ▷ Again, do a single iteration of Ford-Fulkerson
20:  if  $path$  exists then                                ▷ Check if there's an augmenting path
21:     $r \leftarrow$  the minimum residual capacity of  $edge \forall edge \in path$ 
22:    for all  $edge$  in  $path$  do
23:      if  $edge$  is a forward edge then
24:         $G.\text{flow}(edge) \leftarrow G.\text{flow}(edge) + r$ 
25:      else
26:         $G.\text{flow}(edge) \leftarrow G.\text{flow}(edge) - r$ 
27:      end if
28:    end for
29:  end if
30:  return  $G.\text{flow}$ 
31: end function

```

For this algorithm, we must first resolve the new constraints. Specifically, if an edge has had its capacity reduced such that its residual flow has become negative, we must decrease the flow through that edge so that its residual flow becomes zero. Finding an edge from $source$ to $sink$ which includes that negative edge is relatively easy; we can perform a depth-first search, attempting to follow the flow from this overloaded edge through the graph. In the worst case, this has a running time of $O(V + E)$.

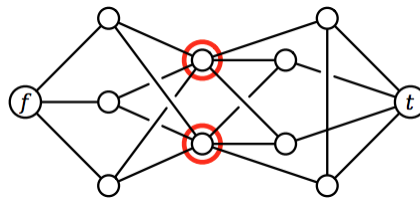
Now, all of our constraints have been satisfied, but the resultant flow may be sub-optimal. However, the flow can only be at most 1 less than the maximum flow, as we have decreased edges by 1; therefore, similar to part (a), we can simply try to find an augmenting path and run a single iteration of Ford-Fulkerson. As in part (a), this has a run-time of $O(V + E)$, and so our final algorithm also has a run-time of $O(V + E)$.

Problem 3

The Island of Sodor is home to a large number of towns and villages, connected by an extensive rail network. Recently, several cases of a deadly contagious disease (either swine flu or zombies; reports are unclear) have been reported in the village of Ffarquhar. The controller of the Sodor railway plans to close down certain railway stations to prevent the disease from spreading to Tidmouth, his home town. No trains can pass through a closed station. To minimize expense (and public notice), he wants to close down as few station as possible. However, he cannot close the Ffarquhar station, because that would expose him to the disease, and he cannot close the Tidmouth station, because then he couldn't visit his favorite pub.

Describe and analyze an algorithm to find the minimum number of stations that must be closed to block all rail travel from Ffarquhar to Tidmouth. The Sodor rail network is represented by an undirected graph, with a vertex for each station and an edge for each rail connection between two stations. Two special vertices f and t represent the stations in Ffarquhar and Tidmouth.

For example, given the following input graph, your algorithm should return the number 2.



Solution

```

1: function MINSTATIONCLOSURES( $G, f, t$ )
2:    $N \leftarrow$  a new graph
3:   for all  $v$  in  $G.vertices$  do
4:     if  $v = f \vee v = t$  then
5:       add  $v$  to  $N$ 
6:     else
7:       add  $v_i$  and  $v_o$  to  $N$ 
8:       connect  $v_i \rightarrow v_o$  with weight 1
9:     end if
10:  end for
11:  for all  $v$  in  $G.vertices$  do
12:    for all  $w$  in  $v.neighbors$  do
13:      if  $\neg(v = t)$  then
14:        connect  $v_o \rightarrow w_i$  with weight  $\infty$ 
15:      end if
16:      if  $\neg(v = f)$  then
17:        connect  $w_o \rightarrow v_i$  with weight  $\infty$ 
18:      end if
19:    end for
20:  end for
21:  return the minimum cut of  $N$ 
22: end function

```

The minimum-cut algorithm is here assumed to determine the lowest-weighted edges to cut; an inverse assignment can be used if the min-cut algorithm instead prioritizes the highest-capacity edges.

This algorithm is designed to duplicate all vertices in G which are not s or t . For all $v \in G$ (again, aside from s or t), we construct two vertices, v_i and v_o . We then construct edges in a directed graph, such that for all nodes w connected to v , there exists a connection from w_o to v_i and a connection from v_o to w_i . Without loss of generality, we assume s as a source and t as destination; thus, for all neighbors n of s , we construct an edge from s to n_i ; conversely, for all neighbors n of t , we construct an edge from n_o to t .

Having split each vertex into two, we can construct a connection from v_i to v_o , and thus all paths from s to t are preserved. Now, we can perform a standard min-cut algorithm; our edge weightings will ensure that the algorithm only selects our newly-created “vertex” edges (those from v_i to v_o); cutting this edge has the equivalent effect to removing v from the graph, as now all paths from s to t passing through v are broken.

There are two stages to the algorithm to analyze for run-time: graph construction and then the min-cut algorithm.

Graph construction is straightforward; we re-create the original graph G , doubling vertices and edges as we go; since V is $O(E)$, our total run-time here is simply $O(E)$.

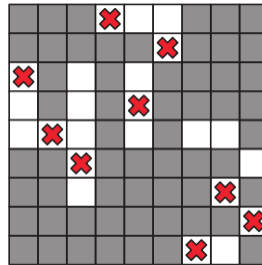
For the min-cut algorithm, the fastest algorithm for computing maximum flows is $O(VE)$, as discovered by James Orlin. Since the min-cut is equivalent to the maximum flow, we can use this algorithm to determine the min-cut of our new graph.

Thus, the total runtime of our algorithm is $O(VE)$.

Problem 4

Suppose we are given an $n \times n$ square grid, some of whose squares are colored black and the rest white. Describe and analyze an algorithm to determine whether tokens can be placed on the grid so that

1. every token is on a white square;
2. every row of the grid contains exactly one token; and
3. every column of the grid contains exactly one token



Your input is a two dimensional array $IsWhite[1..n, 1..n]$ of booleans, indicated which squares are white. Your output is a single boolean. For example, given the grid above as input, your algorithm should return **TRUE**.

Solution

Solution

Problem 5

Ad-hoc networks are made up of low-powered wireless devices. In principle, these networks can be used on battlefields, in regions that have recently suffered from natural disasters, and in other hard-to-reach areas. The idea is that a large collection of cheap, simple devices could be distributed through the area of interest (for example, by dropping them from an airplane); the devices would then automatically configure themselves into a functioning wireless network.

These devices can communicate only within a limited range. We assume all the devices are identical; there is a distance D such that two devices can communicate if and only if the distance between them is at most D .

We would like our ad-hoc network to be reliable, but because the devices are cheap and low-powered, they frequently fail. If a device detects that it is likely to fail, it should transmit its information to some other *backup* device within its communication range. We require each device x to have k potential backup devices, all within distance D of x ; we call these k devices the **backup set** of x . Also, we do not want any device to be in the backup set of too many other devices; otherwise, a single failure might affect a large fraction of the network.

So suppose we are given the communication radius D , parameters b and k , and an array $d[1..n, 1..n]$ of distances, where $d[i, j]$ is the distance between device i and device j . Describe an algorithm that either computes a backup set of size k for each of the n devices, such that no device appears in more than b backup sets, or reports (correctly) that no good collection of backup sets exists.

Solution

Solution