1 Introduction

The shortest augmenting path flow algorithm repeatedly finds a shortest augmenting path (A-path) in the residual graph $G_f$. A simple approach uses BFS to find these shortest paths. Below we describe an incremental approach which does not require redoing everything from scratch each time.

We use distance labels $d(v)$ which are lower bounds on the number of arcs from $v$ to $t$ on a shortest path. The basic properties we use are that:

- $d(t) = 0$
- $d(u) \leq d(v) + 1$ if $(u, v) \in G_f$.

Initialization

We can use a BFS from $t$ in the reverse graph of $G_f$ to compute each vertex’s true distance from $t$ initially. Or we can simply set all $d()$ values to zero (both work, but the BFS will usually make the algorithm faster).

2 Algorithm

The high level idea is to start at $s$ and look for an A-path of length $d(s)$: this would mean going successively to vertices of distance label $d(s) - 1, d(s) - 2, \ldots , 0$. We also record for each vertex a value $\text{pred}(v)$ which is the vertex we reached $v$ from in our current A-path (this allows us to backup both to find the eventual path, and to backtrack).

Basic routines:

- **Relabel(u)**
  
  if $u$ has no neighbors in $G_f$, $d(u) \leftarrow n$
  else, $d(u) \leftarrow k + 1$ where $k$ is the minimum $d()$ value of a neighbor of $u$ in $G_f$.

- **Advance(v)** // moves towards $t$ till gets stuck, or hits $t$
  
  $cv \leftarrow v$; // $cv$ represents the current vertex in our path.
  
  While (exists $u$ with $(cv, u) \in G_f$ AND $d(u) = d(cv) - 1$ )
  
  { $\text{pred}(u) \leftarrow cv$;
    $cv \leftarrow u$; // move to next vertex in path, $u$
  }

  return($cv$); // return last vertex reached

- **Findpath()** // Looks for an A-path, success if hits $t$, fail if backup to $s$
  
  last $\leftarrow$ Advance($s$); // move forward from $s$ till get stuck at last
  
  While (last $\neq s, t$) // stuck partway, so backup
  
  { $v \leftarrow \text{pred}($last$)$;
    relabel($last$); // can’t move forward from last, so needs relabel
    last $\leftarrow$ Advance($v$); // continue looking from $v$
  }

  If (last $= t$) return(1) else return(0);

Main

Start with any legal flow $f$. Create $G_f$ and set initial $d()$ values.

While($d(s) < n$) // if ($d(s) \geq n$) no A-path exists.

If (Findpath() ) // Looks for an A-path
  
  Update $G_f$ using $s \leftarrow t$ path found
Else Relabel($s$); // no path from $s$ of length $d(s)$ so relabel.

3 Analysis

Labels are always greater than zero, no more than $n$, and always go up. Thus there at most $n$ relabel operations per vertex. Each relabel operation takes time $O(\text{degree}(v))$, so relabeling each vertex once takes $O(m)$. Thus all
relabel operations take $O(mn)$ for the entire algorithm. More generally, if we can bound the maximum $d()$ value of a node to be some $r < n$, than the total work for relabels is $O(rn)$.

Most of the other real work takes place in Advance. To find a vertex $u$ such that $v, u \in G_f$ and $d(u) = d(v) - 1$ we have to scan the adjacency list of $v$ (for the graph representing $G_f$). As we noted in class, we keep track of our current position in each adjacency list, so when we start a new scan, we don’t revisit old vertices on the adjacency list (which all have the wrong $d()$ value.). If we find an A-path with $k$ vertices we can consider that the work associated with this path is $O(k)$: to find the path (not counting bookkeeping work already counted in the $O(mn)$ bound), augment, and update $G_f$.

Each A-path kills off at least one critical arc $(v, u)$ which determines the path capacity. When removed from $G_f$ we have $d(v) = d(u) + 1$. To use this arc on an A-path again, we must augment in the reverse direction first, so the $d(u)$ value must go to at least $d(v) + 1$. For example, if when we first kill $(v, u)$ we have $d(u) = 6, d(v) = 7$, then we can augment on $(u, v)$ with $d(u) = 8$, finally to use $(v, u)$ again we must raise $d(v)$ to 9.

In general, between uses of $(v, u)$ as a critical arc the distance labels must go up by at least 2. Thus each arc is critical at most $n/2$ times, and the total number of augmenting paths is thus at most $mn$. With $O(n)$ work per A-Path, we get an $O(mn^2)$ general time bound.

Special cases: for unit graphs all arcs on an A-path are critical. Thus if an A-path has $k$ arcs it takes $O(k)$ work but kills $k$ arcs. Thus the total work for Augmenting is $O(mn)$, the total number of arcs killed over all augmenting paths. The total time for unit networks is then $O(mn)$.

Note: a better algorithm (a variant of the shortest path algorithm) can improve this to $O(mn^{2/3})$ for unit networks.