CSE 5311 Notes 9: Maximum Flows

(Last updated 3/1/17 12:28 PM)

Goldberg and Tarjan, `http://dl.acm.org.ezproxy.uta.edu/citation.cfm?doid=2632661.2628036`

CLRS, Chapter 26

FORD-FULKERSON (review)

Network Flow Concepts:

Maximum Flow

Goal

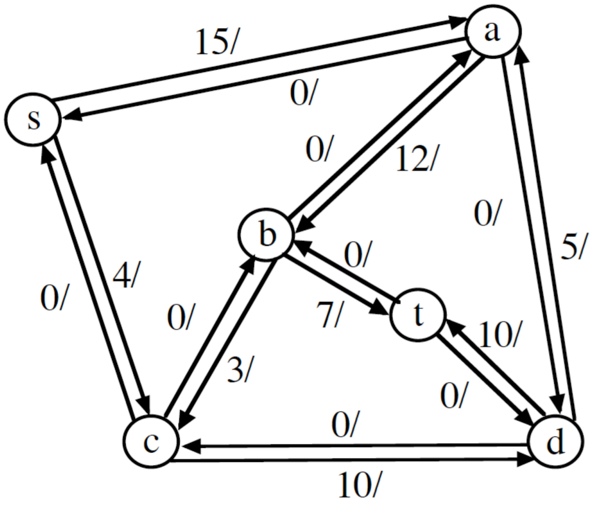
Constraint

Augmenting Path

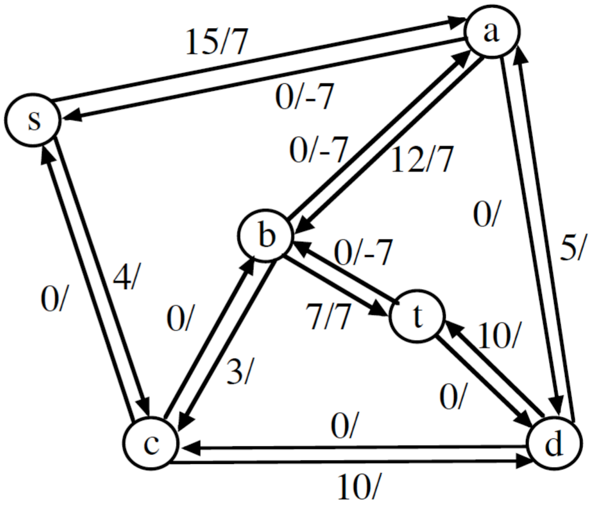
Residual Graph

Example:

Initial residual network:

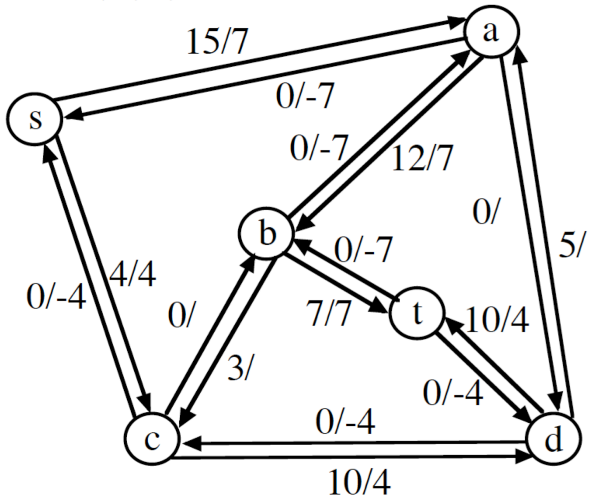


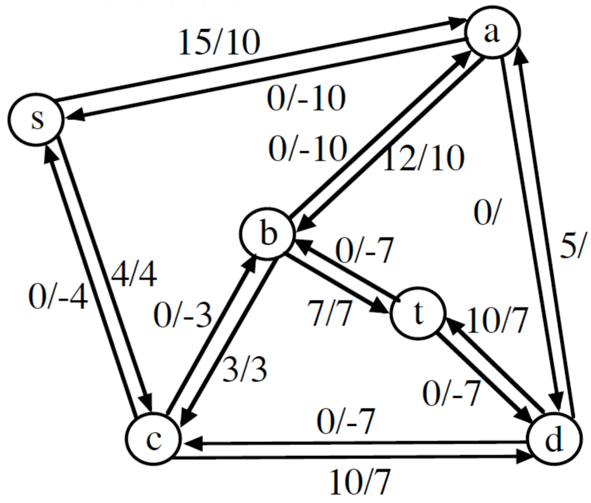
A.P. s, a, b, t / 7 units



A.P. s, c, d, t / 4 units

A.P. s, a, b, c, d, t / 3 units





Search on the residual network yields the minimum cut S = {s, a, b} T= {t, c, d} with capacity 14.

More Concepts (Notes 17.E, CSE 2320):

Termination?

Minimum Cut:

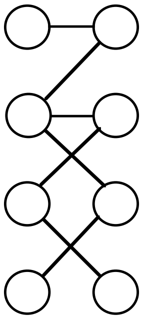
S: Vertices reachable from source in residual graph (using unsaturated edges)

T: *V* - S

Capacity of minimum cut = maximum flow

APPLICATIONS

Bipartite Matching (unweighted) concepts - Alternating path



(Aside: Problem 26-6 introduces the Hopcroft-Karp method, a simplification of Dinic's layered network technique.)

Transportation/Communications Modeling

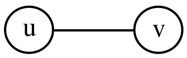
Connectivity

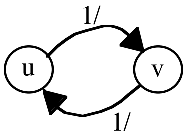
Vertex - Minimum number of vertices to remove to disconnect graph

Edge - Minimum number of edges to remove to disconnect graph

These cannot exceed the degree of any vertex.

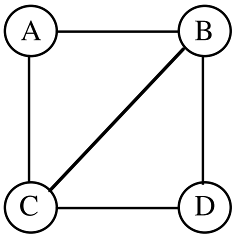
Edge connectivity - undirected

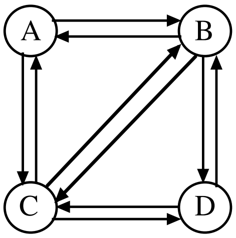
Replace 

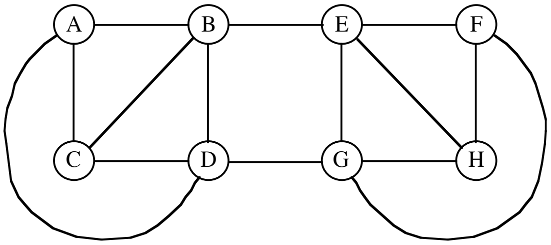
by 

Try every pair of vertices - one as source, one as sink. (Reversing roles is unnecessary.)

Minimum max-flow over all such pairs is the edge connectivity.



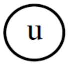


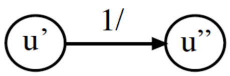


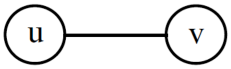
Can speed up by using a fixed source, a vertex of minimum degree ($\leq 2E/V$), along with $V-1$ instances with each of the other vertices as the sink. If Edmonds-Karp is used, the total time is in $\mathrm{O}\left(E^2\right)$ since the number of augmenting paths for each instance is bounded by $2E/V$ and each A.P. is found using breadth-first search taking $\mathrm{O}(E)$.

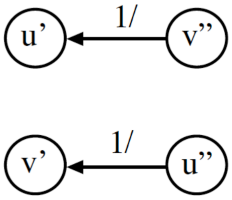
(Aside: `https://en.wikipedia.org/wiki/Gomory–Hu_tree` and a *very* simple, elegant implementation: `http://epubs.siam.org.ezproxy.uta.edu/doi/abs/10.1137/0219009`)

Vertex connectivity - undirected

Replace 

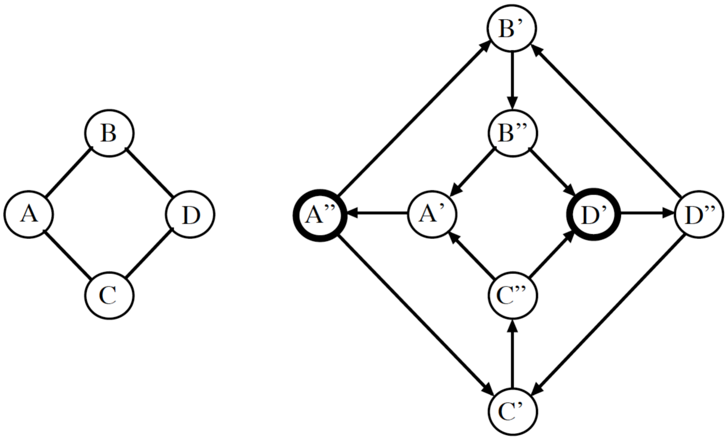
by 

Replace 

by 

Try every non-adjacent pair (x'', y') as (source, sink) for max-flow. (Again, reversing roles is unnecessary)

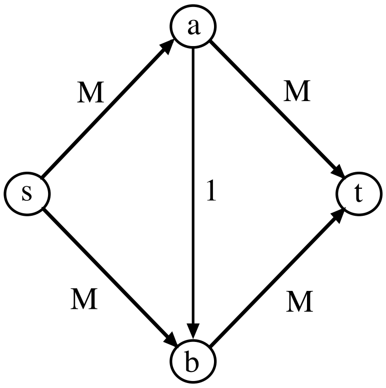
Minimum max-flow over all non-adjacent pairs is the vertex connectivity.



FORD-FULKERSON: Choosing an augmenting path

The original technique makes no assumption about choosing an A.P.

Classic bad case (unlikely in practice).



Can lead to 2M A.P.s that each contribute one unit of flow.

Since the number of bits (in the input file) to represent a number of magnitude M is Θ(log M), Ford-Fulkerson has the theoretical potential to take exponential time.

1. Breadth-First Search (Edmonds-Karp variant) - take an A.P. with smallest number of edges (takes $O(E)$ time to find one)

2. Maximum Capacity Path (MCP) - find an A.P. that maximizes incremental flow

   Found using algorithm similar to Dijkstra (shortest path) and Prim (minimum spanning tree).

   Algorithm uses max heap. (Fibonacci heap version takes $O(E + V \log V)$ time)

How many augmenting paths for each method?

Edmonds-Karp - $O(VE)$ A.P.s to give overall $O\left(VE^2\right)$ time.

Critical edge on A.P.

Limits flow on chosen A.P.

Based on min { capacity – flow }

No capacity will remain on a critical edge after A.P. is recorded

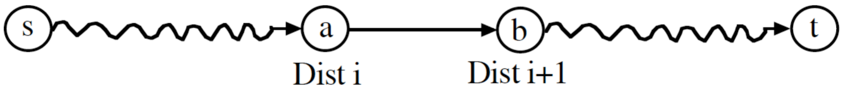
Observations

Edge may become critical several times.

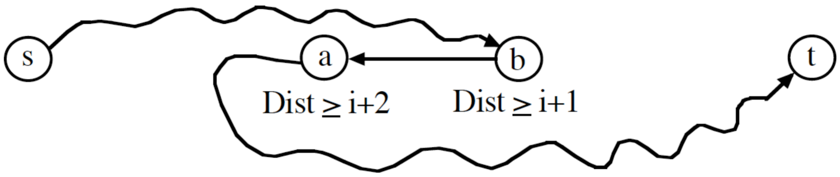
A vertex cannot get closer to source in later rounds of BFS. (Appendix to CSE 2320 Notes 17)

Flow must be sent in opposite direction by another A.P. before an edge can become critical again.

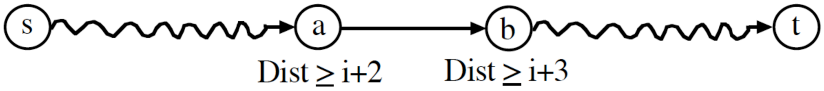
First time critical



Later



Second time critical

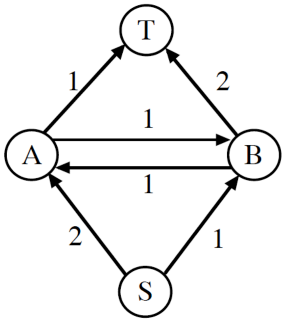


The number of distances available for the tail of a repeated critical edge is $(V\text{-}2)/2$. The number of edges that become critical is $\leq E$. Thus, $O(VE)$ A.P.s overall.

Maximum Capacity Augmenting Paths

All capacities integral. M = maximum capacity over all edges

NOTE! The sequence of MCP paths is *not* required to have incremental flows in descending order!



Flow Decomposition Theorem: Any flow *f* (maximum or otherwise) may be decomposed into no more than E augmenting paths.

Constructive Proof:

A. Use DFS to find cycles (e.g. back edges) in flow graph

1. Determine smallest flow in cycle.

2. Cancel smallest flow along all edges in cycle.

3. Repeat until flow graph is *acyclic*.

B. Find S-T paths

1. Determine smallest flow on path.

2. Cancel smallest flow throughout path. (Flow conservation and acyclicity guarantees this.)

Each edge may only be cancelled *once*.

Corollary: At least one of the paths in the decomposition contributes no less than $\frac{f}{E}$ units of flow.

Claim: The first A.P. found (by MCP) must carry at least $\frac{f}{E}$ units of flow.

Proof: Since the first A.P. found may only go “forward” on input edges, the path must be at least as good as any path in the flow decomposition.

Generalization: After each MCP is found, the effect is to have a new flow problem where the claim again applies. After *k* MCPs, the amount of remaining flow to find is no more than:

$$\left(1-\frac{1}{E}\right)^k f$$

but, due to integral capacities/flows, this amount cannot be < 1.

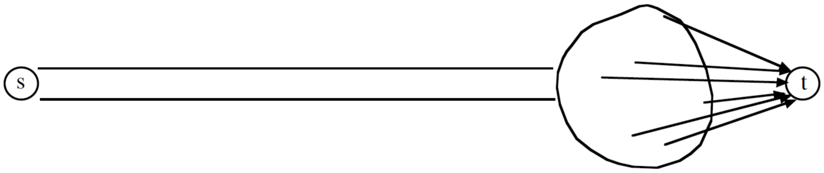
Unfortunately, this leads to dependence on the eventual maximum flow in bounding the number of A.P.s:

$$k \approx \frac{\ln f}{\ln \frac{1}{1-\frac{1}{E}}} \approx E \ln f \text{ since } \ln \frac{1}{1-\frac{1}{E}} = \ln \frac{E}{E-1} \approx \frac{1}{E}$$

SCALING - does not choose the MCP every time

```
Set $\Delta$ to $2^k$ where $k = \left\lfloor \log_2 \max_{v \in V} \{cap(s,v)\} \right\rfloor$
while $\Delta \geq 1$
    Search residual network for an A.P. that will increase flow by at least $\Delta$.
    if A.P. exists
        Record A.P. in residual network
    else
        $\Delta = \Delta / 2$
```

PUSH/RELABEL METHODS (concepts only)

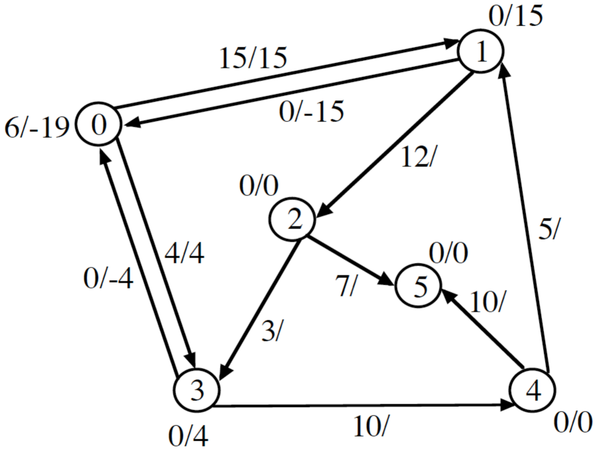


Conservation of flow - only at termination

Vertex status:

Excess - temporary flow tank
Height - controls movement of flow (source height is always V, sink height is always 0)
Will label vertices with height/excess

Example:



Operations - Lift & Push

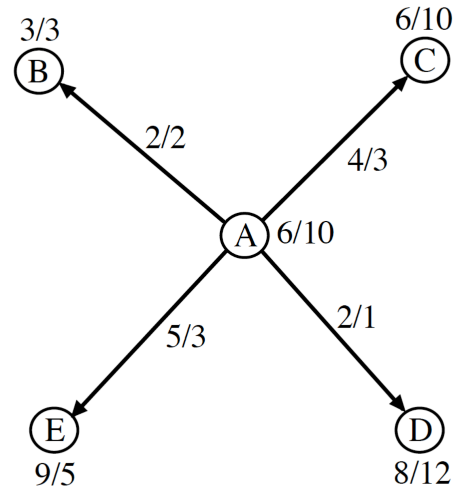
Lift a vertex

1. Vertex has excess > 0 (“overflowing”).
2. All exiting *unsaturated* edges have heads with height ≥ height of tail.

Change height on tail to 1 + minimum height on a head.

Example:

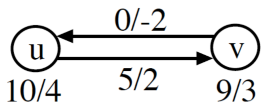
Lift at A from 6 to 7.

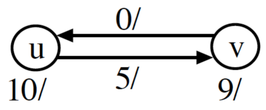


Push from u to v

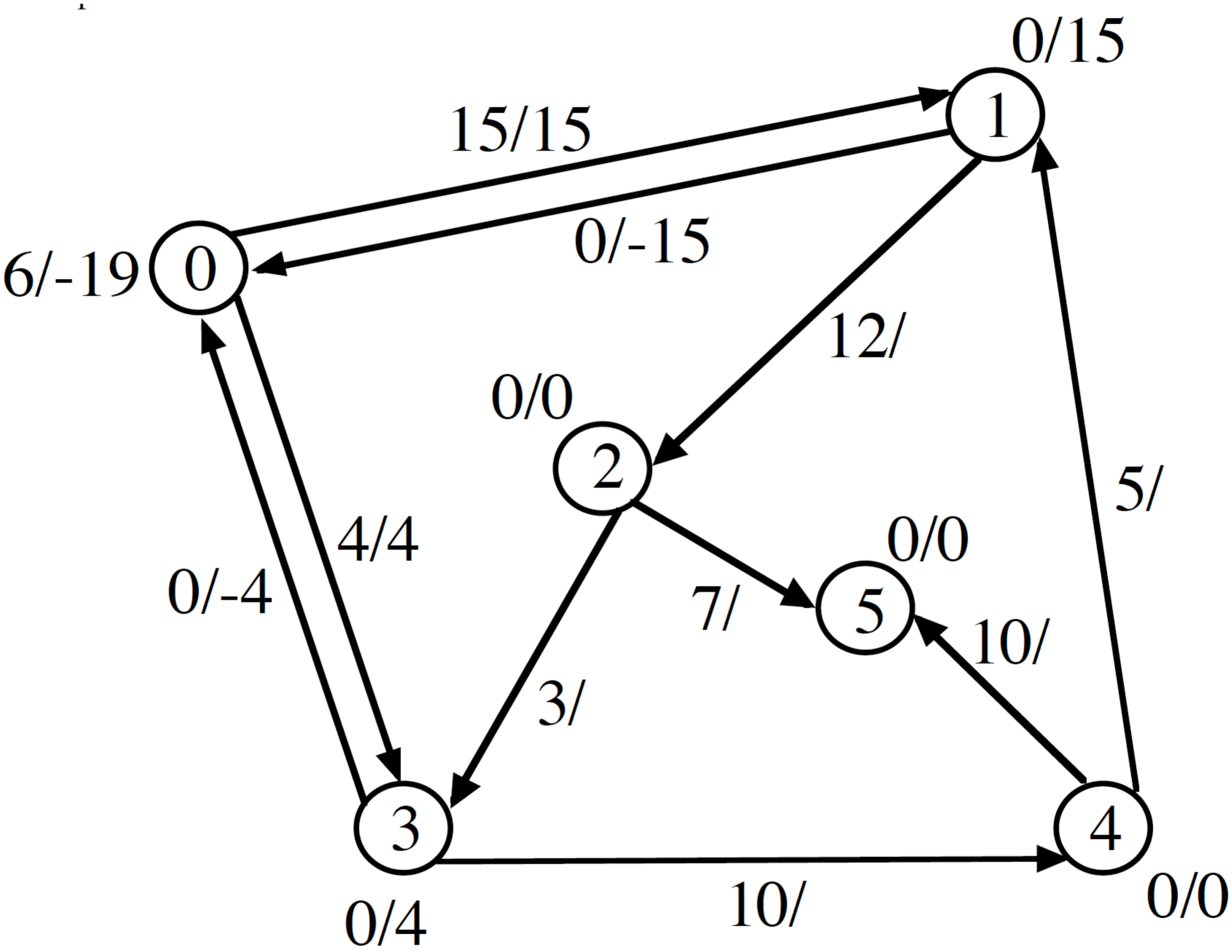
1. u is overflowing.
2. (u, v) has unused capacity.
3. height[u] == height[v] + 1

Push the minimum of the unused capacity and the excess at u.



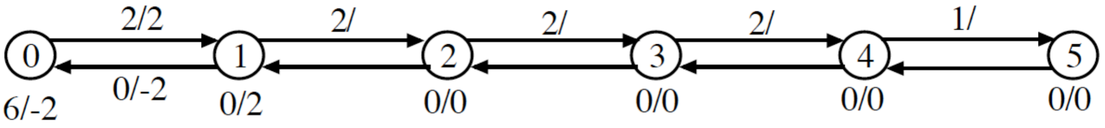


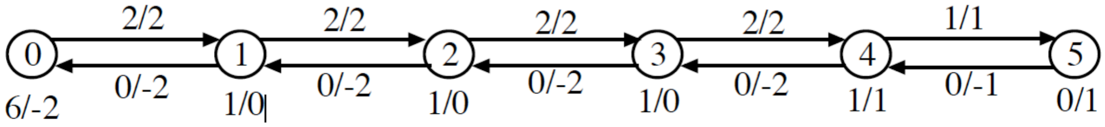
Example 1:



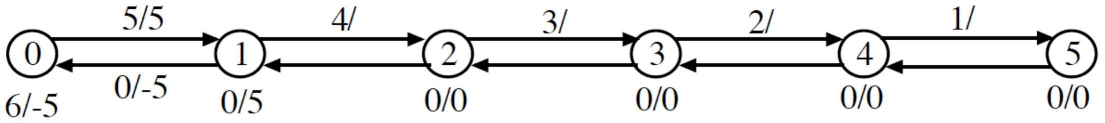
```
a.out < ppExample1.dat
debug: lifting 1 from 0 to 1
debug: pushing 12 units from 1 to 2
debug: lifting 1 from 1 to 7
debug: pushing 3 units from 1 to 0
debug: lifting 3 from 0 to 1
debug: pushing 4 units from 3 to 4
debug: lifting 2 from 0 to 1
debug: pushing 7 units from 2 to 5
debug: lifting 2 from 1 to 2
debug: pushing 3 units from 2 to 3
debug: lifting 2 from 2 to 8
debug: pushing 2 units from 2 to 1
debug: lifting 4 from 0 to 1
debug: pushing 4 units from 4 to 5
debug: lifting 3 from 1 to 2
debug: pushing 3 units from 3 to 4
debug: pushing 2 units from 1 to 0
debug: pushing 3 units from 4 to 5
total flow is 14
flows along edges:
0->1 has 10
0->3 has 4
1->2 has 10
2->3 has 3
2->5 has 7
3->4 has 7
4->5 has 7
```

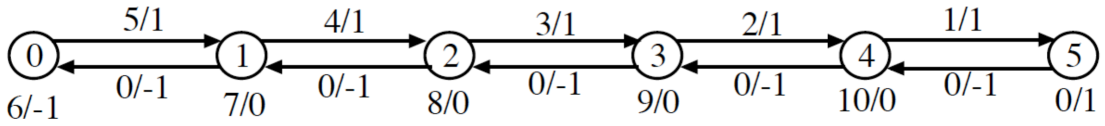
Example 2:





Example 3:





Implementation - Must avoid scanning of vertices and adjacency lists (especially in dense graphs).

1. Maintain queue of overflowing vertices:

   a. by initialization

   b. by pushes

2. For each vertex, separate edges:

   a. saturated (flow = capacity)

   b. unsaturated (flow < capacity, includes inverses)

3. Processing - driven by queue.

```
int preflowPushLoop()
{
int i,j,k,dF,minHeight;
int *fifo,*inQueue,tail=0,head=0;

. . .

while (tail!=head)
{
  i=fifo[head];
  head=(head==n) ? 0 : head+1;
  inQueue[i]=0;
  while (1)
  {
    minHeight=oo;
    for (k=firstEdge[i];k<firstEdge[i+1];k++)
    {
      j=edgeTab[k].head;
      if (edgeTab[k].capacity-edgeTab[k].flow>0)
        if (height[i]==height[j]+1)
        {
          dF=min(excessFlow[i],edgeTab[k].capacity-edgeTab[k].flow);
          //printf("debug: pushing %d units from %d to %d\n",dF,i,j);
          edgeTab[k].flow+=dF;
          edgeTab[edgeTab[k].inverse].flow=(-edgeTab[k].flow);
          excessFlow[i]-=dF;
          excessFlow[j]+=dF;
          if (!inQueue[j])
          {
            fifo[tail]=j;
            tail=(tail==n) ? 0 : tail+1;
            inQueue[j]=1;
          }
          if (excessFlow[i]==0)
            break;
        }
        else if (height[j]<minHeight)
          minHeight=height[j];
    }
    if (excessFlow[i]>0)
    {
      //printf("debug: lifting %d from %d to %d\n", i,height[i],1+minHeight);
      height[i]=1+minHeight;
    }
    else
      break;
  }
}
free(fifo);
free(inQueue);
return excessFlow[n-1];
}
```

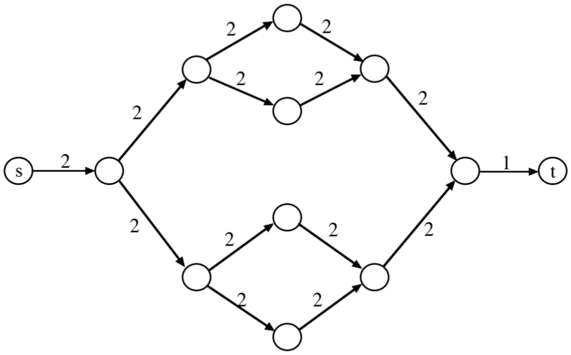
Correctness:

"Augmenting paths" are found by having paths with vertices with decreasing heights.

Excess is not stranded - must reach sink or return to source.

Maximum height is _____________.

Lift operation is intended to allow excess to move to alternate paths. Try:



Time: $O\left(V^2E\right)$ (Not responsible for details)

Can improve vertex processing order (CLRS 26.5) to achieve $O\left(V^3\right)$.

Can speed up in practice by occasionally using BFS to increase height of some vertices (also see CLRS problem 26.5-5).

Consider graph whose edges are the *inverses* of the unsaturated edges:

1. BFS starting from sink. Distance from sink is new height.

2. BFS for vertices not reached by first BFS (can flag these by initializing new heights to 2*V*) starting from source. Distance from source + *V* is new height.

First BFS makes it easier to find "paths". Second BFS forces excess to return quickly to source.