Chapter 1

Maximum Flow

Flow in a network can be:

a.) flow of oil or water through a system of pipelines

b.) flow of electricity

c.) flow of phone calls

d.) flow of email messages

e.) flow of commodities being transported across truck routes

The flow of the material at any point in the system is the rate at which the material moves.

Directed edges in flow networks are conduits for the material. Each edge has a capacity which is the maximum rate at which the material can flow through the conduit.

Vertices are conduit junctions. There are two special vertices: the source and the sink.

Material flows through all other vertices without stopping at the vertices.

The study of network flow bridges several diverse and seemingly unrelated areas of combinatorial optimization, such as matchings used to solve scheduling and assignment problems, the edge and vertex connectivity problems, and the minimum cut problem.

1.1 Flow networks

A flow network is a directed graph $G = (V, E)$ with two special vertices: the source $s \in V$ and the sink $t \in V$ [CLR90]. Each edge $(u, v) \in E$ has a nonnegative capacity $c(u, v) \geq 0$. We assume that:

a.) $c(u, v) = 0$ if $(u, v) \notin E$

b.) every vertex lies on some path from s to t (so the graph is connected)

A flow in G is a real-valued function $f : V \times V \to \mathbf{R}$ such that for all $u \in V$, $v \in V$:

a.) $f(u, v) \leq c(u, v)$ – capacity constraint

b.) $f(u, v) = -f(v, u)$ – skew symmetry
Note that this implies that $f(u, u) = 0$, because $f(u, u) = -f(u, u)$

c.) $\sum\limits_{v \in V} f(u, v) = 0$ for all $u \in V \setminus \{s, t\}$ – flow conservation

The quantity $f(u,v)$ is called the net flow from vertex u to vertex v. The value of a flow f is the total net flow out of the source and is denoted by $|f|$. In other words:

$$|f| = \sum_{v \in V} f(s,v)$$

The maximum-flow problem consists in finding a flow of maximum value from s to t for a given flow network G with source s and sink t. The positive net flow entering a vertex $v \in V$ is given by:

$$\sum_{u \in V, f(u,v)>0} f(u,v)$$

The positive net flow leaving a vertex $v \in V$ is given by:

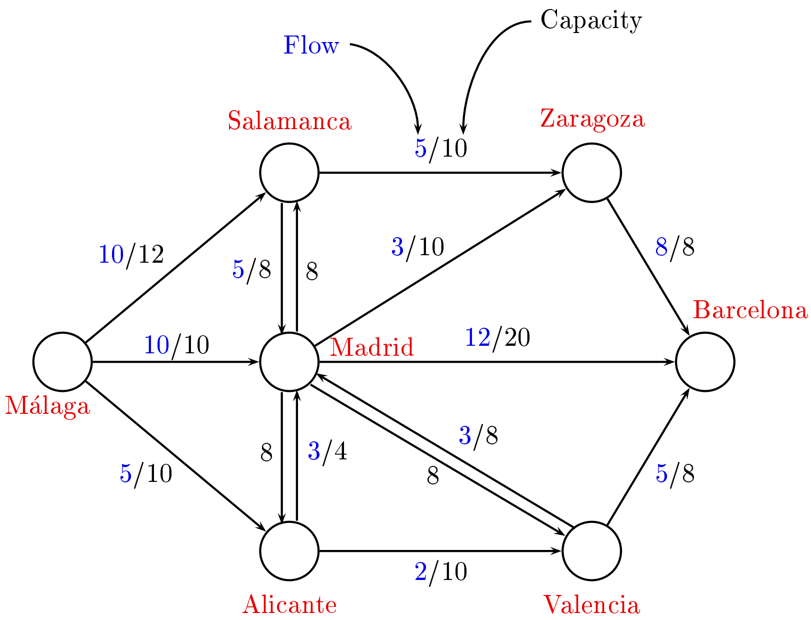
$$\sum_{u \in V, f(v,u)>0} f(v,u)$$

By the flow conservation property, the positive net flow entering $v \in V$ where $v \neq s$ and $v \neq t$, is equal to the positive net flow leaving vertex $v \in V$.

EXAMPLE: The ANDALUCIA ACEITUNA COMPANY (AAC) buys black and green olives form the farmers in southern Spain and puts them in jars in Málaga. The jars that are not destined for local consumption are placed in one-ton containers, called crates, and are then sent to Barcelona to be shipped to the rest of Europe.

AAC does not own trucks and therefore leases space on trucks from the BARCELONA CAMIÓN COMPANY (BCC) trucking company. The trucks of BCC travel over specified routes between major cities in Spain and have only limited capacity for AAC's crates since BCC also does business with other manufacturers.

So AAC has no control over the routes and capacities of the trucks of BCC. AAC can ship at most $c(u,v)$ crates per day between each pair of cities u and v. The goal of AAC is to ship the largest number p of crates per day that can reach Barcelona (that same day). Recall that olives cannot pile up in intermediate cities. The rate at which olives enter an intermediate city is equal to the rate at which olives leave. In our EXAMPLE we have: $s =$ Málaga and $t =$ Barcelona. The intermediate cities are: Salamanca, Zaragoza, Madrid, Alicante, and Valencia. Only $c(u,v)$ crates per day can go from u to v. For example: AAC can move a maximum number of 8 crates of olives from Madrid to Alicante per day. Only positive netflows are shown in the diagram. If $f(u,v) > 0$, then edge (u,v) is labeled with $f(u,v)/c(u,v)$.



The flow in the above example is:

$$|f| = \sum_{v \in V} f(\text{Málaga}, v) = 25$$

Is this the maximum possible flow ?

For the above figure, please consider: edges with one number only, such as Madrid–Alicante, have zero flow. Also note that in the example:

$$\begin{aligned} f(\text{Salamanca, Madrid}) &= 5 \\ f(\text{Madrid, Salamanca}) &= -5 \quad \text{by skew symmetry} \end{aligned}$$

likewise:

$$\begin{aligned} f(\text{Madrid, Valencia}) &= 3 \\ f(\text{Valencia, Madrid}) &= -3 \end{aligned}$$

and

$$\begin{aligned} f(\text{Madrid, Zaragoza}) &= 3 \\ f(\text{Zaragoza, Madrid}) &= -3 \\ f(\text{Madrid, Barcelona}) &= 12 \\ f(\text{Barcelona, Madrid}) &= -12 \end{aligned}$$

So that

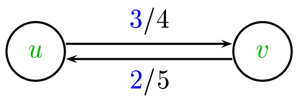
$$\begin{aligned} f(\text{Madrid}, v) &= \sum_{v \in V} f(\text{Madrid}, v) \\ &= f(\text{Madrid, Salamanca}) + f(\text{Madrid, Zaragoza}) \\ &+ f(\text{Madrid, Barcelona}) + f(\text{Madrid, Valencia}) \\ &+ f(\text{Madrid, Alicante}) + f(\text{Madrid, Málaga}) \\ &= (-5) + (+3) + (+12) + (+3) + (-3) + (-10) \\ &= 0 \quad \text{as expected by flow conservation} \end{aligned}$$

$\square$

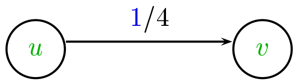
Remark: Without loss of generality, we can say that positive flow goes either from u to v, or from v to u but not both.

If we have positive flow in both directions, we can transform the scenario into an equivalent one which has only positive flow.

Example:



We can transform it into the following scenario by "cancelling" 2 units of flow in each direction:



We do not show the edge from v to u since the flow is zero ($f(v,u) = 0$). Note that the flow from u to v in both cases is one unit. Note that the capacity constraint and flow conservation are still satisfied $\square$

Notation: For $X \subseteq V$ and $Y \subseteq V$ we write:

$$f(X,Y) = \sum_{u \in X} \sum_{v \in Y} f(u,v)$$

Remarks: Let $G = (V,E)$ be a flow network and f a flow in G.

1. $f(X,X) = 0$ for all $X \subseteq V$
2. $f(X,Y) = -f(Y,X)$ for all $X \subseteq V$ and $Y \subseteq V$
3. for $X \subseteq V$, $Y \subseteq V$, $Z \subseteq V$ and $X \cap Y = \emptyset$, we have:
   (a) $f(X \cup Y, Z) = f(X,Z) + f(Y,Z)$
   (b) $f(Z, X \cup Y) = f(Z,X) + f(Z,Y)$

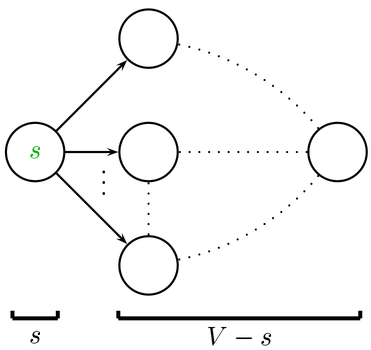
Result :

$$|f| = f(V,t)$$

for any flow f in a flow network

Proof: $|f| = \sum_{v \in V} f(s,v)$ by definition

$$= \sum_{u \in V} \sum_{v \in V} f(u,v) - \sum_{u \in V \setminus s} \sum_{v \in V} f(u,v)$$



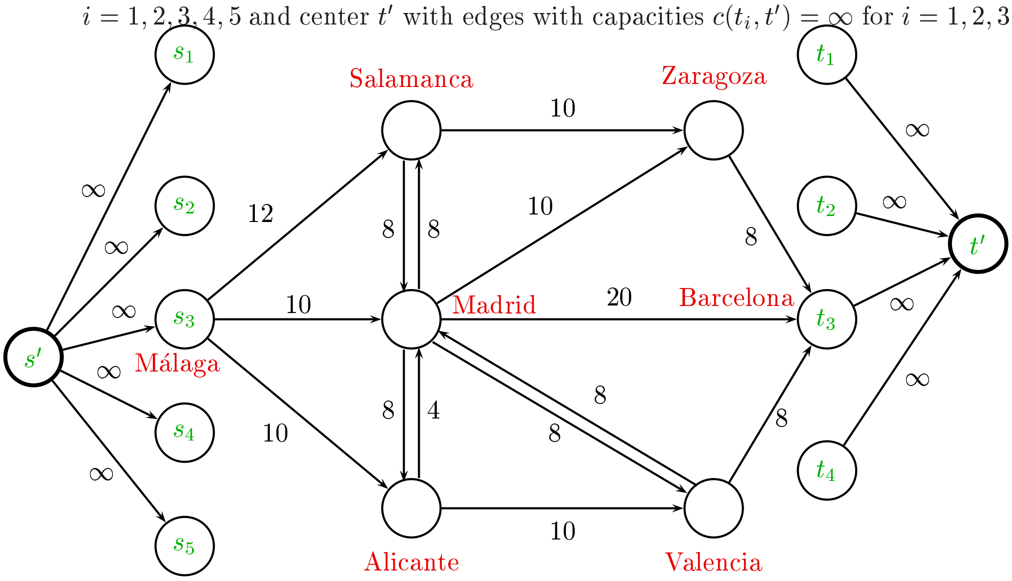
$$
\begin{aligned}
|f| &= \sum_{u\in V}\sum_{v\in V} f(u,v) - \sum_{u\in V\backslash s}\sum_{v\in V} f(u,v) \\
&= f(V,V) - f(V\backslash\{s\},V) \text{ by definition of } f(X,Y) \\
&= -f(V\backslash s, V) \text{ by Remark 1} \\
&= f(V, V\backslash s) \text{ by Remark 2} \\
&= f(V, V-\{s\}+\{t\}\cup\{t\}) \text{ by Remark } 3b) \\
&= f(V,t) \text{ by flow conservation}
\end{aligned}
$$

□

1.1.1 Multiple sources and sinks

A maximum flow problem may have more than one source, s_1, s_2, ..., s_m, and more than one sink, t_1, t_2, ..., t_3. Reduce the problem with multiple sources and sinks to one with only one source s' and only one sink t' by adding edges in the following way: (s', s_i) and costs $c(s', s_i) = \infty$ for $i = 1, 2, \ldots, m$ and (t_i, t') and costs $c(t_i, t') = \infty$ for $i = 1, 2, \ldots, n$.

Example: If AAC puts olives in jars in centers s_1, s_2, s_3, s_4, s_5 and ships them to t_1, t_2, t_3, t_4; create a center s' and edges with capacities $c(s', s_i) = \infty$ for $i = 1, 2, 3, 4, 5$ and center t' with edges with capacities $c(t_i, t') = \infty$ for $i = 1, 2, 3, 4$.



□

1.2 The Ford-Fulkerson method

Solving the maximum flow problem consists in finding paths from s to t and pushing the maximum flow over these paths.

The Ford-Fulkerson method is based on three ideas:

- residual network
- augmenting paths
- cuts

It is called a method and not an algorithm because Ford-Fulkerson's method consists of several different algorithms with different running times.

```
FORD-FULKERSON-METHOD
initialize flow f to 0
while there exists an augmenting path P
      do augment flow f along P
return f
```

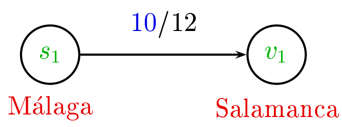
An augmenting path is a path from s to t along which we can push more flow.

1.2.1 Residual networks

Given a flow network $G = (V, E)$ with source s, sink t, and a flow f in G. The residual capacity of edge (u, v) is the net flow we can push from u to v before exceeding the capacity $c(u, v)$:

$$c_f(u, v) = c(u, v) - f(u, v)$$

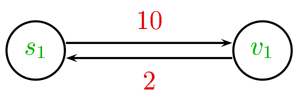
Example: Consider the edge joining Málaga to Salamanca in our previous example:



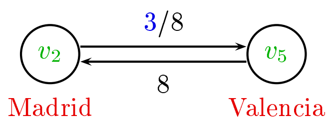
Let s_1 = Málaga, v_1 = Salamanca, $c(s_1, v_1) = 12$, and $f(s_1, v_1) = 10$. So $c_f(s_1, v_1) = 12 - 10 = 2$, and

$$\begin{aligned} c_f(v_1, s_1) &= c_f(v_1, s_1) - f(v_1, s_1) \\ &= 0 - (-10) \text{ since } f(s_1, v_1) = 10 \\ &= 10 \end{aligned}$$

So $c_f(v_1, s_1) = 10$. Let us replace edge (s_1, v_1) by edges (s_1, v_1) and (v_1, s_1) labeled with the residual capacities.



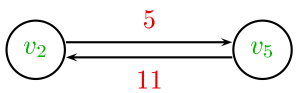
Similarly for edge (Madrid, Valencia).



$$\begin{aligned} c_f(v_2, v_5) &= c(v_2, v_5) - f(v_2, v_5) \\ &= 8 - 3 = 5 \\ c_f(v_5, v_2) &= c(v_5, v_2) - f(v_5, v_2) \\ &= 8 - (-3) = 11 \end{aligned}$$

(Note that $f(v_5, v_2) = 0 + (-f(v_2, v_5)) = -3$.)

By replacing the values between Madrid and Valencia by the residual capacities, we get:



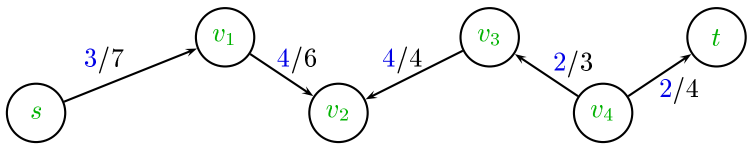
Note that when the net flow is negative, $f(v_5, v_2) = -3$ in the second example, then the residual capacity $c_f(v_5, v_2) = 11$ is greater than the capacity $c(v_5, v_2) = 8$. Are we violating the capacity from v_5 to v_2 ? No, since in reality we have 3 crates that go from v_2 to v_5, $f(v_2, v_5) = 3$, which cancel out 3 of the units that go from v_5 to v_2, and thus we have $11 - 3 = 8$ crates that travel from v_5 to v_2. □

The residual network of G, given a flow network $G = (V, E)$ and a flow f, is the graph $G_f = (V, E_f)$ induced by f, where:

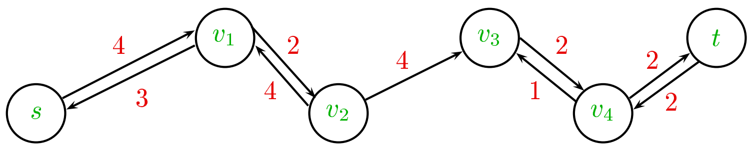
$$E_f = \{(u, v) \in V \times V \text{ such that } c_f(u, v) > 0\}$$

In other words, the edges of G are replaced by edges in G_f labeled with the residual capacities.

Example: Consider the following path through G. Note that this is only one path (conservation law does not hold since we don't see the rest of the flow network).



path in G



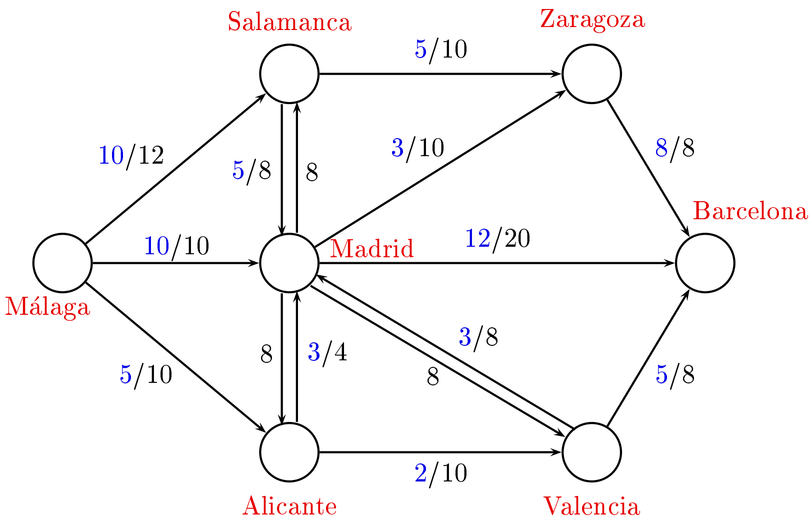
edges in the residual graph G_f

Note that the path in G can be augmented by 2 units. □

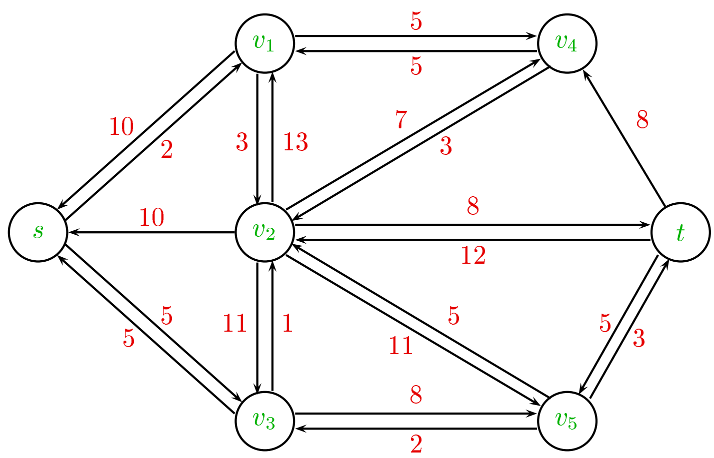
As pointed out in the example, new edges may emerge in G_f and thus E_f is rarely equal to E.

Example: Let us consider the flow network of the AAC example with flow function f given by the BCC.

So we construct the residual network G_f induced by f.



Given flow network with flow f



residual network induced by f

Since an edge (u, v) can appear in G_f only if at least one of (u, v) and (v, u) appears in the original network G, we have:

$$|E_f| \leq 2|E|$$

In our example:

$$|E| = 15 \text{ and } |E_f| = 22$$

Also note that G_f is itself a flow network with capacities given by c_f. There exists a relationship between a flow in G and a flow in G_f.
Result: $G = (V, E)$ is a flow network with flow f. $G_f = (V, E_f)$ is the residual network induced by f. Let f' be a flow in G_f. Define the flow sum $f + f'$ to be $f + f' : (V \times V) \to \mathbf{R})$, where

$$(f + f')(u, v) = f(u, v) + f'(u, v).$$

Then:

a.) $f + f'$ is a flow in G

b:) $|f + f'| = |f| + |f'|$

Proof:

a:) We show that $f + f'$ is a flow in G:

i.) skew symmetry:

$$\begin{aligned}(f+f')(u,v) &= f(u,v)+f'(u,v)\\ &= -f(v,u)-f'(v,u)\\ &= -(f(u,v)+f'(u,v))\\ &= -(f+f')(v,u)\end{aligned}$$

ii.) capacity constraints:

$$\begin{aligned}(f+f')(u,v) &= f(u,v)+f'(u,v)\\ &\leq f(u,v)+c_f(u,v)\\ \text{but } c_f(u,v) &= c(u,v)-f(u,v)\\ \Rightarrow (f+f')(u,v) &\leq f(u,v)+[c(u,v)-f(u,v)]\\ &= c(u,v)\\ \text{i. e. } (f+f')(u,v) &\leq c(u,v)\end{aligned}$$

iii.) flow conservation: for all $u \in V \backslash \{s,t\}$:

$$\begin{aligned}\sum_{v\in V}(f+f')(u,v) &= \sum_{v\in V}(f(u,v)+f'(u,v))\\ &= \sum_{v\in V} f(u,v)+\sum_{v\in V} f'(u,v)\\ &= 0+0\\ &= 0\end{aligned}$$

b.) We show that $|f + f'| = |f| + |f'|$

$$\begin{aligned}|f+f'| &= \sum_{v\in V}(f+f')(s,v)\\ &= \sum_{v\in V}(f(s,v)+f'(s,v))\\ &= \sum_{v\in V} f(s,v)+\sum_{v\in V} f'(s,v)\\ &= |f|+|f'|\end{aligned}$$

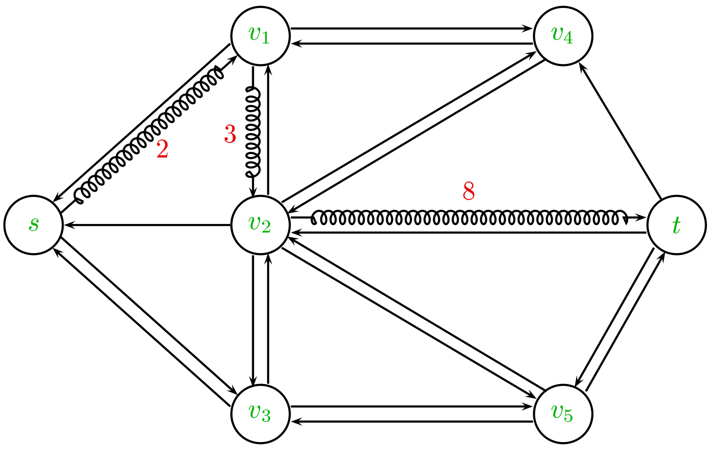
□

Recall that an augmenting path is a path such that we can push more flow through it. Given a flow network $G = (V, E)$ and a flow f, an augmenting path p is a simple path from s to t in the residual network G_f.

By finding a path from s to t in G_f, we know that we can push more flow through that path. We shall formally prove this fact later.

Let us consider our example once more and look for a path in G_f.

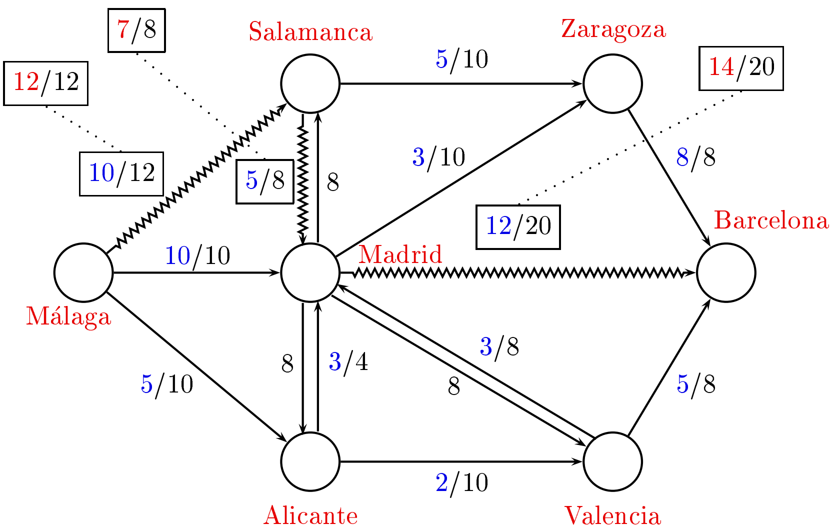
EXAMPLE:



One augmenting path p is the one that goes from s to v_1 to v_2 to t. □
The residual capacity of a path p is:

$$c_f(p) = \min\{c_f(u,v)|(u,v) \text{is on } p\}$$

It is the maximum amount of net flow that we can move along the edges of p. In our example, $c_f(p) = \min\{2, 3, 8\}$. So 2 more crates of olives could be shipped from Málaga to Barcelona via Salamanca then Madrid



The flow in G that results from augmenting along path p by its residual capacity $c_f(p) = 2$.

Now that we have a "new" flow f, we can repeat the process:

first: construct the new residual network

second: find an augmenting path p and compute its residual capacity

third: augment the path by its residual capacity

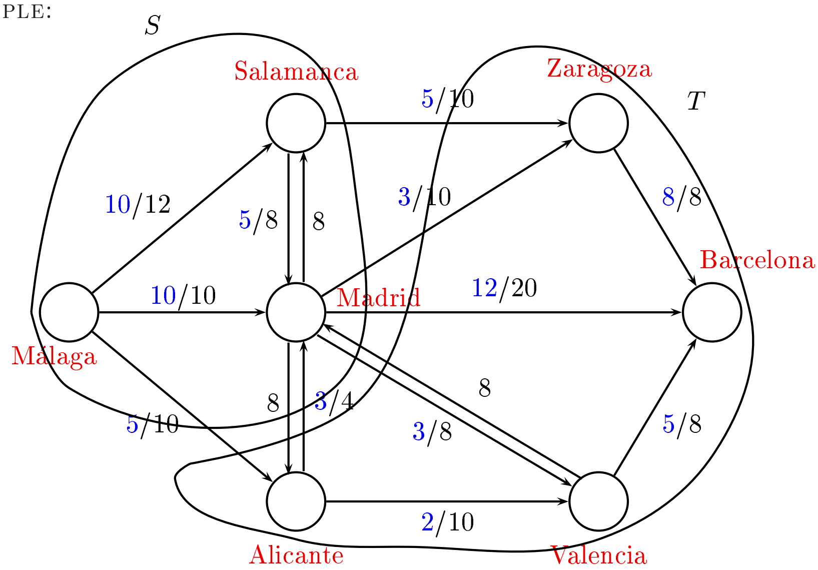
To prove that a flow is maximum if and only if its residual network contains no augmenting path, we need to study cuts first.

A cut (S,T) of a flow network $G=(V,E)$ is a partition of V into two disjoint sets $S \subseteq V$ and $T \subseteq V$ such that $S \cup T = V$, $s \in S$, and $t \in T$. The net flow across the cut (S,T) is $f(S,T)$ and the capacity of the cut (S,T) is $c(S,T)$.

A minimum cut of a network is a cut whose capacity is minimum over all cuts of the network.

Note that the net flow across a cut can have negative net flows between vertices. On the other hand, the capacity of a cut is made up of nonnegative-valued capacities.

EXAMPLE:



Consider the cut (S,T) where

$$S = \{s, v_1, v_2\} \text{ and } T = \{v_3, v_4, v_5, t\}$$

The net flow across the cut is:

$$\begin{aligned} f(S,T) &= f(v_1,v_4) + f(v_2,v_4) + f(v_2,t) + f(v_2,v_5) + f(v_2,v_3) + f(s,v_3) \\ &= 5+3+12+3+(-3)+5 = 25 \end{aligned}$$

And its capacity is:

$$\begin{aligned} c(S,T) &= c(v_1,v_4) + c(v_2,v_4) + c(v_2,t) + c(v_2,v_5) + c(v_2,v_3) + c(s,v_3) \\ &= 10+10+20+8+8+10 \\ &= 66 \end{aligned}$$

Remark: Not that $f(S,T) = 25$. But

$$|f| = \sum_{v \in V} f(s,v) = 10+10+5 = 25$$

That is no coincidence.
Result:

$$f(S,T) = |f|$$

Proof: Use Remark 1, 2, 3 seen earlier.

$$f(S,T) \quad = \quad f(S,V) - f(S,S)$$

$$
\begin{aligned}
&= f(S,V)\\
&= f(s \cup (S-s), V)\\
&= f(s,V) + f(S-s,V)\\
&= f(s,V)\\
&= |f|
\end{aligned}
$$

□

Remark: Note that because we have

$$|f| = f(S,T)$$

and

$$f(S,T) = \sum_{u \in V} \sum_{v \in T} f(u,v) \leq \sum_{u \in S} \sum_{v \in T} c(u,v)$$

and

$$\sum_{u \in S} \sum_{v \in T} c(u,v) = c(S,T) \ ,$$

we conclude that

$$|f| \leq c(S,T).$$

1.3 Max-Flow Min-Cut Theorem

Theorem 1.3.1 *If f is a flow in a flow network $G = (V, E)$ with source s and sink t, then the following statements are equivalent:*

1. *f is a maximum flow*
2. *the residual network G_f contains no augmenting paths*
3. *$|f| = c(S,T)$ for some cut (S,T) of G*

Proof:

$1 \Rightarrow 2$: Proof by Contradiction: Suppose that f is a maximum flow in G but G_f contains an augmenting path p. The proof is a direct result of the following lemma:

Lemma: *If we define a function $f_p : V \times V \to \mathbf{R}$ such that:*

$$f_p(u,v) = \begin{cases} c_f(p) & \textit{if } (u,v) \textit{ is on } p \\ -c_f(p) & \textit{if } (v,u) \textit{ is on } p \\ 0 & \textit{otherwise,} \end{cases}$$

where p is an augmenting path in G_f, then f_p is a flow in G_f and $|f_p| = c_p(p) > 0$

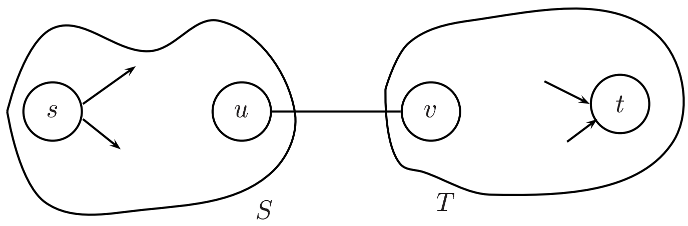
$2 \Rightarrow 3$: Suppose that G_f contains no path from s to t. Let

$$S = \{v \in V | \exists \text{ a path from } s \text{ to } v \text{ in } G_f\}$$

and let

$$T = V - S$$

(S,T) is a cut. $s \in S$ and since we have no path from s to t, $t \in T$.



For each pair (u,v) in the cut (with $u \in S$ and $v \in T$), we have $(u,v) \notin E_f$ (because we don't have a path from s to v). So for every such pair, we have $f(u,v) = c(u,v)$ (i.e. (u,v) has 0 residual capacity). We know that $f(S,T) = |f|$. But here $|f| = c(S,T)$, so $|f| = c(S,T)$.

$3 \Rightarrow 1$: We know by a previous result that $|f| \leq c(S,T)$, i.e. $c(S,T)$ is an upper bound for $|f|$. Since $|f| = c(S,T)$, we know that f is a maximum flow.

□

We are now ready to present the Ford-Fulkerson algorithm that simply implements Ford-Fulkerson's method.

1.4 The basic Ford-Fulkerson algorithm

The algorithm is labeled "basic" since after its appearance, many variations (and improvements) were designed.

Ford-Fulkerson algorithm starts by initializing the flow to zero. It then constructs the residual graph G_f. The main loop of the algorithm will be executed as long as we can find an augmenting path p in G_f.

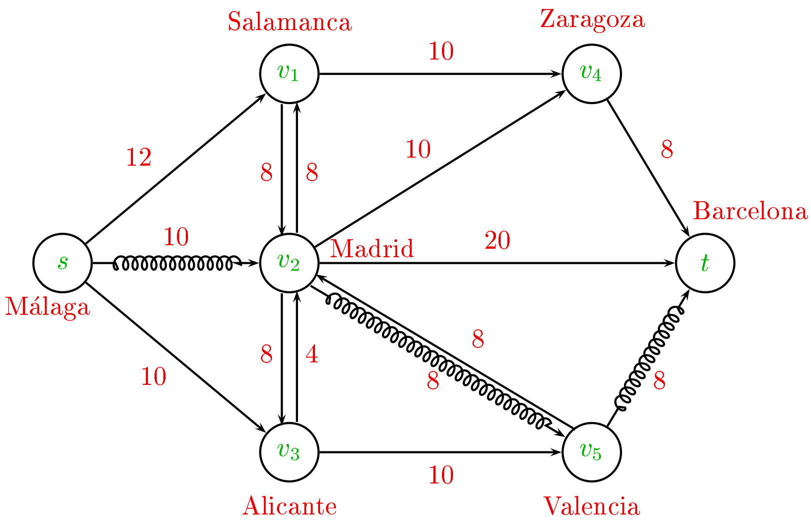
Recall that the residual capacity $c_f(u,v)$ is given by

$$c_f(u,v) = c(u,v) - f(u,v)$$

FORD-FULKERSON*(G)*

begin
  for each edge $(u,v) \in E$
    $f(u,v) \leftarrow 0$
    $f(v,u) \leftarrow 0$
  while there exists an augmenting path p in G_f
    $c_f(p) \leftarrow \min\{c_f(u,v) | (u,v) \text{ is in } p\}$ determine residual capacity
    for each edge (u,v) in p
      $f(u,v) \leftarrow f(u,v) + c_f(p)$
      $f(v,u) \leftarrow -f(u,v)$
    Find the new residual graph after having
    pushed $c_f(p)$ along the path p
    Construct the residual graph G_f
  return the function f
end

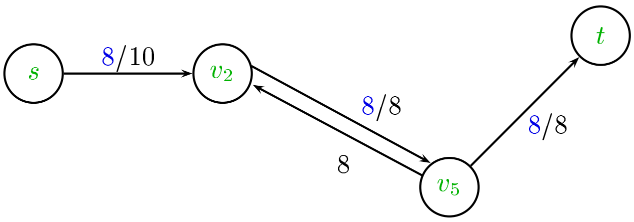
EXAMPLE: Consider our flow network once more with $f(u,v) = 0$ for all edges $(u,v) \in E$.



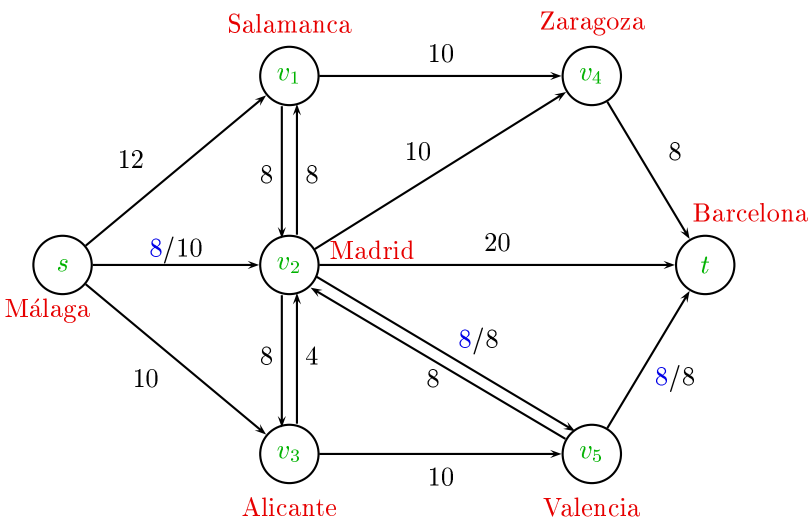
Assume the algorithm picks $s \to v_2 \to v_5 \to t$ to be the augmenting path p.

$$c_f(p) = \min\{10, 8\} \Rightarrow c_f(p) = 8$$

The flow network will be exactly the same as above except for the augmenting path, which will be:



The following is the flow network obtained after adding $c_f(p) = 8$ to the augmenting path p.

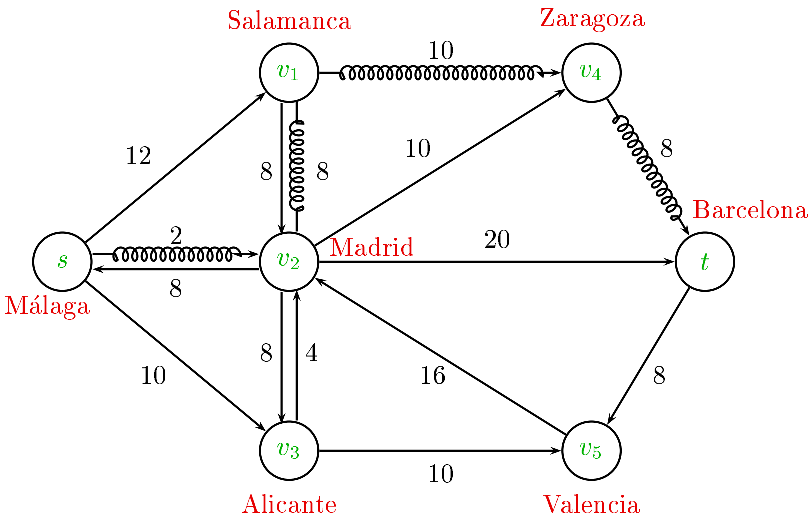


We now construct the residual network G_f of the above flow network

$$\text{edge}(s, v_2) : c_f(s, v_2) \quad = \quad c(s, v_2) - f(s, v_2)$$

$$
\begin{aligned}
& = && 10 - 8 = 2 \\
\text{edge}(v_2, s) : c_f(v_2, s) & = && c(v_2, s) - f(v_2, s) \\
& = && 0 - (-8) = 8 \\
\text{edge}(v_2, v_5) : c_f(v_2, v_5) & = && 8 - 8 = 0 \\
\text{edge}(v_5, v_2) : c_f(v_5, v_2) & = && 8 - (-8) = 16 \\
\text{edge}(v_5, t) : c_f(s, v_2) & = && 8 - 8 = 0 \\
\text{edge}(t, v_5) : c_f(s, v_2) & = && 0 - (-8) = 8
\end{aligned}
$$

The new residual network G_f induced by the flow f in the previous figure.



One possible choice for an augmenting path p is:

$$s \rightarrow v_2 \rightarrow v_1 \rightarrow v_4 \rightarrow t$$

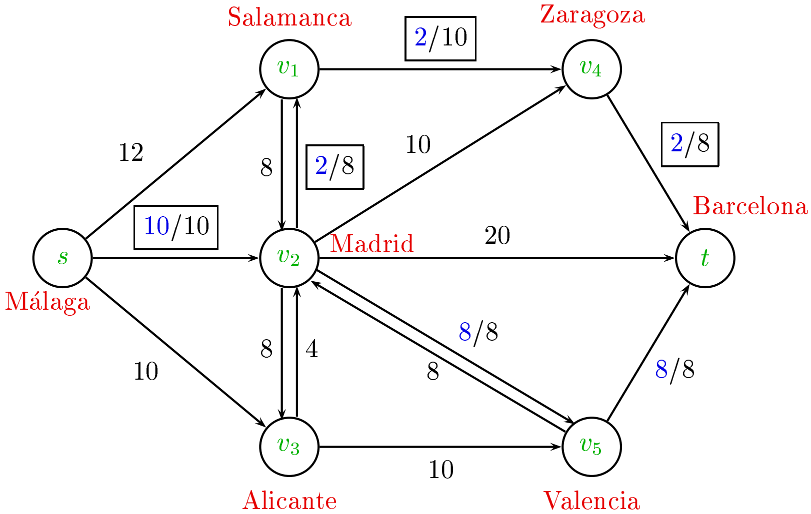
$$c_f(p) = \min\{2, 8, 10\} = 2$$

¿From the last flow network:

$$f(u, v) \leftarrow f(u, v) + c_f(p) \text{ and } f(v, u) \leftarrow -f(u, v)$$

In other words

$$
\begin{array}{ll}
f(s, v_2) = 8 + 2 = 10 & f(v_2, s) = -10 \\
f(v_2, v_1) = 0 + 2 = 2 & f(v_1, v_2) = -2 \\
f(v_1, v_4) = 0 + 2 = 2 & f(v_1, v_4) = -2 \\
f(v_4, t) = 0 + 2 = 2 & f(t, v_4) = -2
\end{array}
$$



The flow network after pushing two more crates along path p:

$$s \to v_2 \to v_1 \to v_4 \to t$$

Note that we don't show $f(u, v)$ when its value is nonpositive (as is the convention with flow networks). Nonpositive $c_f(u, v)$ are shown in residual networks only. The value/capacity that have changed, since the last iteration, are in the red rectangles.

The next step involves constructing the residual network G_f, then choosing an augmenting path and so on. The algorithm stops when no such path can be found in the residual network.

1.5 Analysis of the Ford-Fulkerson algorithm

Only one algorithm for the maximum flow-problem appeared before Ford–Fulkerson's algorithm. The 1951 network simplex method of Dantzig for the transportation problem solved the maximum flow problem as a natural special case.

Since the development of the augmenting path method by Ford and Fulkerson, many more efficient algorithms were (and are being) developed by several computer scientists.

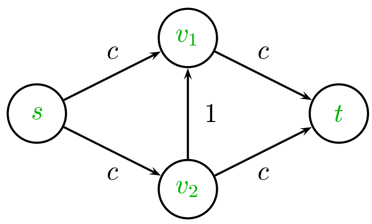
The following table from "Beyond the flow decomposition barrier" by A. Goldberg and Rao gives a history of maximum flow bounds.

year	discoverer(s)	exact bounds
1951	Dantzig	$O(n^2mU)$
1955	Ford & Fulkerson	$O(nmU)$
1970	Dinitz Edmonds & Karp	$O(nm^2)$
1970	Dinitz	$O(n^2m)$
1972	Edmonds & Karp Dinitz	$O(m^2 \log U)$
1973	Dinitz Gabow	$O(nm \log U)$
1974	Karzanov	$O(n^3)$
1977	Cherkassky	$O(n^2\sqrt{m})$
1980	Galil & Naamad	$O(nm \log^2 n)$
1983	Sleator & Tarjan	$O(nm \log n)$
1986	Goldberg & Tarjan	$O(nm \log(n^2/m))$
1987	Ahuja & Orlin	$O(nm + n^2 \log U)$
1987	Ahuja et al.	$O(nm \log(n\sqrt{\log U}/(m+2)))$
1989	Cheriyan & Hagerup	$O(nm + n^2 \log^2 n)$
1990	Cheriyan et al.	$O(n^3/\log n)$
1990	Alon	$O(nm + n^{8/3} \log n)$
1992	King et al.	$O(nm + n^{2+\epsilon})$
1993	Phillips & Westbrook	$O(nm(\log_{m/n} n + \log^{2+\epsilon} n))$
1994	King et al.	$O(nm \log_{m/(n \log n)} n)$
1997	Goldberg & Rao	$O(m^{3/2} \log(n^2/m) \log U)$ $O(n^{2/3} m \log(n^2/m) \log U)$

In the table, n denotes the number of vertices, m the number of edges and the capacities of the edges are integers in the range $[1 \dots U]$.

Back to Ford-Fulkerson's algorithm. Its running time heavily depends on the way the augmenting path p in the head of the **while**-loop is chosen. The result can be pretty bad with poor choices of the path. Let us assume that all capacities have integral values. Let us also assume that the augmenting path is arbitrarily chosen. Under these assumptions, the Ford-Fulkerson algorithm runs in time $O(|E||f^*|)$, where $|f^*|$ is the maximum flow found by the algorithm. The computation is straightforward: It takes $\Theta(|E|)$ to initialize the edges of G and at most $|f^*|$ iterations of the **while**-loop are performed.

When the capacities are integral and the optimal flow value $|f^*|$ is small, we obtain a good running time. But if we have a network as shown in the figure below, the algorithm will take exponential time (even though the capacities have integer values).



1.6 Variations of the Ford-Fulkerson algorithm

1.6.1 Edmonds-Karp algorithm

A natural variant of the Ford-Fulkerson algorithm is to choose the shortest augmenting path in the residual network G_f. This algorithm can be implemented by using breadth first search and can be shown to run in polynomial time. It was independently developed by Dinitz and by Edmonds and Karp in the early 1970's.

Note that by shortest augmenting path we mean the path from s to t that contains as few edges as possible. As before, Edmonds-Karp algorithm repeats this step until no further paths can be found. A crucial observation is that the length of the shortest path increases with each iteration of the main loop in the algorithm. This fact is not obvious since new edges can be added to the residual graph as we augment a path. Note that the added edges are along the augmenting path and are directed in the opposite direction of the flow. It can be shown that the Edmonds-Karp algorithm runs in $O(|V||E|^2)$ time.

1.6.2 Preflow-push algorithms

The preflow-push algorithms are the fastest maximum-flow algorithms to date. Preflow-push algorithms work in a more localized fashion than the Ford-Fulkerson algorithm. Rather than examining the entire residual network to find an adequate augmenting path, preflow-push algorithms concentrate on one vertex at a time, looking only at the vertex's immediate neighbors in the residual network.

The flow-conservation property is not maintained throughout the execution. Instead, the algorithm maintains a preflow which is a function $f : V \times V \to \mathbf{R}$ that satisfies:

i) skew symmetry

ii) capacity constraint

ii) $f(V, u) \geq 0$ for all $u \in V \backslash \{s\}$

The third property of preflow is a relaxation of flow conservation that postulates that the net flow into each vertex other than the source is nonnegative.

The excess flow into u, given by $e(u) = f(V, u)$ is the net flow into u. A vertex $u \in V \backslash \{s, t\}$ is overflowing if $e(u) > 0$.

The algorithm can be understood, intuitively, by thinking of the flow network as being a system of interconnected pipes of given capacities. The Ford-Fulkerson method iteratively adds more streams of flow until no more can be added. On the other hand, vertices for preflow-push algorithms are pipe junctions with two properties: First, each vertex has an outflow pipe leading to an arbitrarily large reservoir, to accommodate excess flow. Second, each vertex, its reservoir, and all its pipe connections are on a platform whose height increases as the algorithm progresses. Naturally, flow is pushed from a high vertex to a lower vertex.

The height of the source is $|V|$ and of the sink is 0. Other vertex heights start at 0 and increase with time. The increase of height of a vertex is performed through the operation of lifting. The height is increased to one unit more than the height of the lowest of its neighbors to which it has an unsaturated pipe. So the lifting of the vertex will result in the pushing of flow to at least that unsaturated pipe. We now formally present the preflow-push algorithm. As usual, assume that $G = (V, E)$ is a flow network with source s and sink t and f a preflow function in G. A height function $h : V \to \mathbf{N}$ is a function such that:

$$h(s) = |V|,\ h(t) = 0 \text{ and}$$
$$h(u) \leq h(v) + 1 \text{ for every residual edge } (u, v)$$

Note that the definition of h implies that if $h(u) > h(v) + 1$ then (u, v) is not an edge in the residual graph G_f.

Operation Push

PUSH(u, v) is an operation that is activated if u is an overflowing vertex (i.e., $e(u) > 0$) and is such that $c_f(u, v) = 0$ and $h(u) = h(v) + 1$. PUSH(u, v) updates the preflow f in a flow network $G = (V, E)$. The amount of flow that can be pushed from u to v is stored in $d_f(u, v)$. The excess flow stored at vertex u is maintained as $e(u)$ and its current height is $h(u)$.

Note that at any given time, we do not want to push a quantity of flow that will cause $e(u)$ to become negative, or that will exceed the capacity of the edge. So, $d_f(u, v) = \min\{e(u), c_f(u, v)\}$ units of flow will be allowed to be pushed.

```
PUSH(u,v)
begin
//applicability: u is overflowing, (i.e. e(u) > 0)
//c_f(u,v) > 0 and h(u) = h(v) + 1
//action: PUSH d_f(u,v) = min{e(u), c_f(u,v)}
//units of flow from u to v
   d_f(u,v) ← min{e(u), c_f(u,v)}            amount to push
//move the flow from u to v
//by updating f
   f(u,v) ← f(u,v) + d_f(u,v)
   f(v,u) ← −f(u,v)
//and by updating e
   e(u) ← e(u) − d_f(u,v)
   e(v) ← e(v) + d_f(u,v)
end //of PUSH(u,v)
```

Note that PUSH(u, v) is activated only when $h(u) = h(v) + 1$. If after a PUSH(u, v), edge (u, v) becomes a saturated edge, i.e. $c_f(u, v) = 0$ after the push, then the push is called a saturating push, otherwise it is called a nonsaturating push. Note that saturated edges do not show up in the residual network.

Operation Lift

We lift an overflowing vertex u for every vertex v for which there is residual capacity from u to v. Flow cannot be pushed from u to v because v is not downhill from u. In other words, LIFT(u) is activated if u is overflowing, and if

$$c_f(u, v) > 0 \Rightarrow h(u) \leq h(v) \text{ for all vertices } v$$

```
LIFT(u)
begin
//applicability: u is overflowing (e(u) > 0)
//and for all v ∈ V, we have
//(u,v) edge in G_f ⇒ h(u) ≤ h(v)
//action: increase the height of u
   h(u) ← 1 + min{h(v)|(u,v) is an edge in G_f}
end //of PUSH(u)
```

After the execution of LIFT(u), we say that u is lifted. Recall that neither s nor t can be lifted. We note that $\{h(v)|(u, v) \text{ is an edge in } G_f\}$ is not an empty set since u is an overflowing vertex (i.e. $c_f(u, v) > 0$).

1.6.3 The preflow-push procedure

The preflow-push procedure basically consists of two parts: initialization and a push or lift loop.

Initially, each edge leaving s is filled to capacity, while all other edges carry no flow at all. In other words:

$$f(u,v) = \begin{cases} c(u,v) & \text{if } u = s \\ -c(u,v) & \text{if } v = s \\ 0 & \text{otherwise} \end{cases}$$

As for the height function, it initially assigns the following values:

$$h(u) = \begin{cases} |V| & \text{if } u = s \\ 0 & \text{otherwise} \end{cases}$$

And for each vertex v adjacent to s we have $e(v) = c(s,v)$, otherwise $e(u) = 0$ for all other vertices.

Recall the definition of a height function, $h : V \to \mathbf{N}$ such that

$$\begin{aligned} h(s) &= |V| \\ h(t) &= 0 \\ h(u) &\leq h(v) + 1 \text{ for every residual edge } (u,v) \end{aligned}$$

In our initialization, we have $h(s) = |V|$, $h(t) = 0$ and the edges (u,v) that violate $h(u) \leq h(v)+1$ (that is for which $h(u) > h(v)+1$), are the one for which $u = s$ and those are saturated ($f(u,v) = c(u,v)$) and so are not present in the residual graph G_f.

In other words, these edges are not residual edges, and so the initial function is indeed a height function.

After the initialization, the procedure repeatedly performs the Push or Lift procedures seen in the previous sections. Note that as long as an overflowing vertex exists, at least one of the two operations applies. This is true since for any residual edge (u,v), we have $h(u) \leq h(v)+1$ (since h is a height function). Now if Push(u,v) is not applicable, then we must have $h(u) < h(v) + 1$ for all residual edges (u,v). In other words, $h(u) \leq h(v)$ and thus a Lift operation can be applied to u.

```
PREFLOW-PUSH(G)
begin
//Initialization phase
  for each vertex u ∈ V
    h(u) ← 0
    e(u) ← 0
  for each edge (u, v) ∈ E
    f(u, v) ← 0
    f(v, u) ← 0
  h(s) ← |V|                        height of s is the only nonzero
                                    valued height in the beginning
//saturate edges leaving the source
  for each vertex u ∈ Adj[s]
    f(s, u) ← c(s, u)
    f(u, s) ← −c(s, u)
    e(u) ← c(s, u)
//PUSH or LIFT phase
  while there exists an applicable PUSH
      or LIFT operation
    do select an applicable PUSH
      or LIFT operation and perform it
  return f //f is the maximum flow of G
end
```

Remarks:

1. Only when the algorithm terminates does the preflow become a flow, and then it is a maximum flow. For that reason it is labeled preflow. Karzanov was the first to use the idea of preflow.

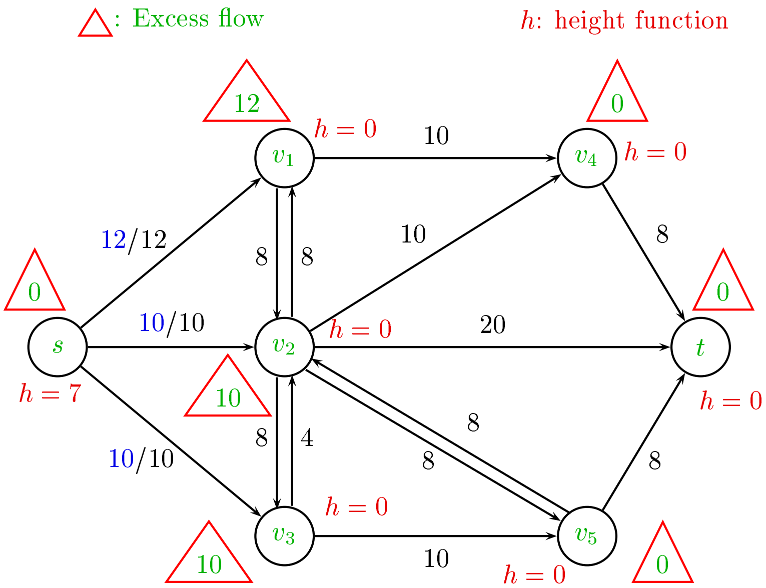
2. Note that the condition of "overflowing" is present for both operations: push and lift. Thus, for the implementation of the preflow-push algorithm, we could maintain a list (or set) of active vertices, where a vertex u is active if it is overflowing (i.e. $e(u) > 0$).

The **while**-loop of the preflow-push repeatedly performs the following steps, in any order, until there are no active vertices:

push: Select any active vertex u. Select any residual edge (u, v) with $h(u) = h(v) + 1$. Send $d_f(u, v) = \min\{e(u), c_f(u, v)\}$ units of flow from u to v. The push has the effect of increasing $f(u, v)$ and $e(v)$ by $d_f(u, v)$ and decreasing $f(v, u)$ and $e(u)$ by $d_f(u, v)$. The push is saturating if $d_f(u, v) = c_f(u, v)$ and nonsaturating otherwise. After performing push, if $e(u)$ is still positive, then of course u remains an active vertex.

lift: Select any active vertex u for which $(u, v) \in E_f \Rightarrow h(u) \leq h(v)$. Replace $h(u)$ by $1 + \min\{h(v) | (u, v) \in E_f\}$.

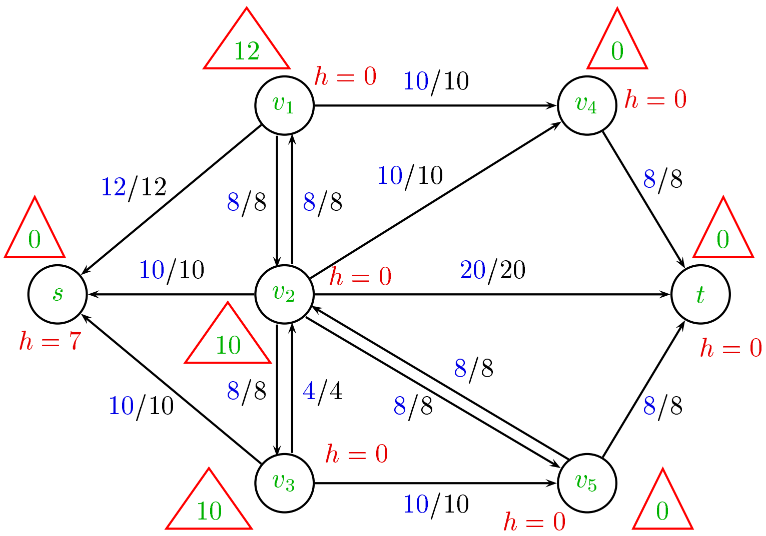
EXAMPLE: Consider the AAC example once more.



Network after the initialization phase

The active nodes are: v_1, v_2, v_3. At this point we have only one possible operation: lift. To be able to push we should have $h(u) = h(v) + 1$, where u is an active vertex, and $(u, v) \in E_f$. But $h = 0\ \forall u \in V \backslash \{s\}$. So no pushes are possible.

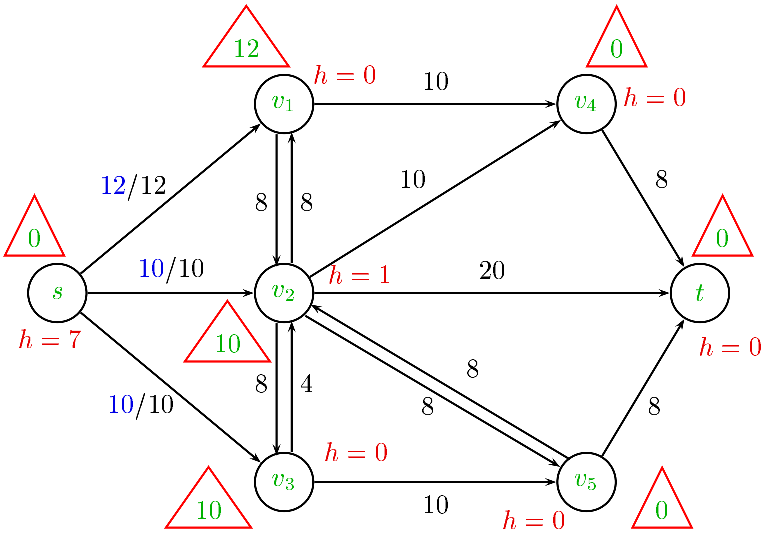
The residual network G_f induced by the initial preflow in G.



Again, notice that the nodes that have an excess flow: v_1, v_2, and v_3 are such that $h(u) = h(v) = 0$ for $u \in \{v_1, v_2, v_3\}$ and $(u, v) \in E_f$ and $v \neq s$.

Suppose that preflow-push chooses v_2 to lift. $\textsc{Lift}(v_2)$ will increase the height of v_2 by $1 + \min\{h(v) | (v_2, v) \in E_f\} = 1 + 0 = 1$. So the new height of v_2 will be 1.

The network after performing $\textsc{Lift}(v_2)$ now has $h(v_2) = 1$ (instead of $h(v_2) = 0$).



G after the lifting of v_2

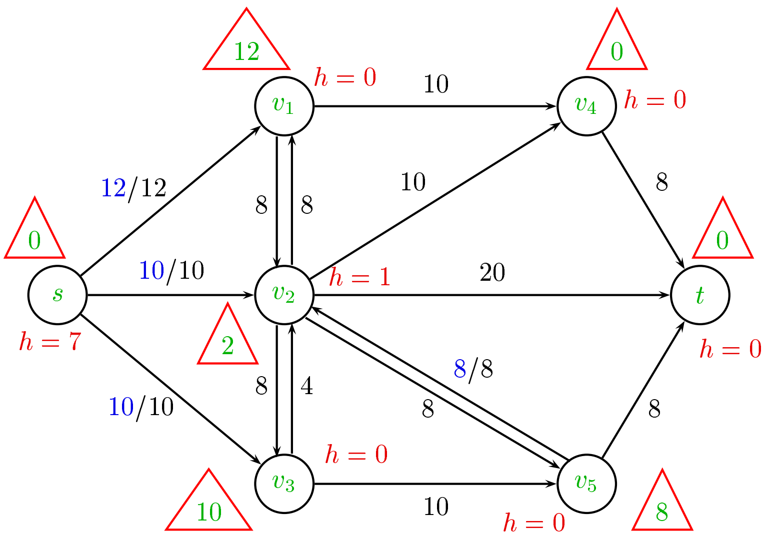
At this point we still have the same three active nodes: v_1, v_2 and v_3 since $e(u) > 0$ for $u \in \{v_1, v_2, v_3\}$. Preflow-push can choose between three options:

a) LIFT(v_1)

b) LIFT(v_3) or

c) push the excess flow found in vertex v_2.

Assume that preflow-push chooses option c), i.e. push the excess flow of v_2.



Network after performing PUSH(v_2, v_5)

Moreover, suppose that the excess flow is pushed to v_5. Note that v_1, v_3, v_4 and t would have been other possible recipients of the excess flow of v_2 (since $h(v_2) = h(v) + 1$ and $c_f(v_2, v) > 0$ for $v \in \{v_1, v_3, v_4, t\}$. PUSH($v_2, v_5$) will send $c_f(v_2, v_5) = \min\{e(v_2), c_f(v_2, v_5)\} = 8$ units of flow to v_5. So $f(v_2, v_5) = 8$, $f(v_5, v_2) = -8$ and $e(v_2) = 10 - 8 = 2$, $e(v_5) = 0 + 8 = 8$.

Note that v_2 remains an active vertex even after PUSH(v_2, v_5) since $e(v_2) = 2$. So now the set of active vertices comprises v_1, v_2, v_3 and v_5.

At this point preflow-push can choose between pushing the excess flow of v_2 or choosing a vertex from $\{v_1, v_3, v_5\}$ to perform a lift operation.

And so preflow-push goes on pushing or lifting until there are no active vertices (i.e., until no vertex is overflowing).

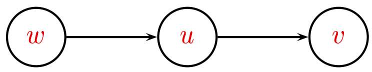
1.6.4 Correctness of the preflow-push algorithm

We want to show that the preflow-push algorithm terminates and that when it does, the flow it produces is the maximum possible flow. We start by showing that vertex heights never decrease. Heights are modified only by the lift operation which "raises" the value of the height of the vertex. For vertex u to be lifted, it must be such that $h(u) \leq h(v)$ for all $(u,v) \in E_f$. In other words $h(u) < 1 + \min\{h(v)|(u,v) \in E_f\}$ before the lift. And after the lift of u we have $h(u) = 1 + \min\{h(v)|(u,v) \in E_f\}$. So vertex heights

1. never decrease
2. increase by at least 1

Another important observation about the height function is to realize that the height function will remain a height function (recall the conditions $h(s) = |V|, h(t) = 0$, and $h(u) \leq h(v) + 1$ for every $(u,v) \in E_f$) after the execution of the preflow-push algorithm. We already saw that the initialization phase does indeed define a height function on the vertices of the network. Let us show that the operations of lift and push do indeed leave h a height function.

Operation lift: For a residual edge (u,v), we shall have $h(u) \leq h(v) + 1$ after the execution of $\text{Lift}(u)$.



As for residual edge (w,u), we have $h(w) \leq h(u)+1$ before and $h(w) < h(u)+1$ after the execution of $\text{Lift}(u)$. In other words, after the execution of $\text{Lift}(u)$, we'll have $h(u) \leq h(v) + 1$ for every $(u,v) \in E_f$ and so h is still a height function.

Operation push: Consider $\text{Push}(u,v)$. We have 2 possibilities:

a) It may add (v,u) to E_f (if in our example, we perform a $\text{Push}(v_2,t)$ instead of $\text{Push}(v_2,v_5)$, then we add (v_2,t) to E_f with $f(v_2,t) = 10$). In this case we have $h(v) = h(u) - 1$ and so h remains a height function (in our example, we have: $h(v_2) = 1$ and $h(t) = 0$, i.e. $h(t) = h(v_2) - 1$).

b) It may remove (u,v) from the residual network. (That is indeed the case after the execution of $\text{Push}(v_2,v_5)$ in our example. Now (v_2,v_5) is no longer in the residual graph.) But the removal of (u,v) from G_f also removes the corresponding constraint, and so h will still be a height function after the execution of $\text{Push}(u,v)$. □

We prove one last property before showing the correctness of the preflow-push algorithm.

Theorem 1.6.1 *$G = (V,E)$ is a flow network with source s and sink t, f is a preflow in G and h is a height function on V. There is no path from the source s to the sink t in the residual network G_f.*

Proof: The proof is by contradiction.

Suppose there is a path $p = < v_0, v_1, \ldots, v_k >$ from $s = v_0$ to $t = v_k$ in the residual graph G_f. W.l.o.g. (without loss of generality), p is a simple path (does not include cycles) so that $k < |V|$. So for $i = 0, 1, \ldots, k-1$, we have edge (v_i, v_{i+1}) in the residual graph G_f. h is a height function, so we must have $h(v_i) \leq h(v_{i+1}) + 1$ for $i = 0, 1, 2, \ldots, k-1$. Thus we have:

$$\begin{aligned}
h(s) &\leq h(v_1) + 1 \\
&\leq (h(v_2) + 1) + 1 = h(v_2) + 2 \\
&\leq h(v_3) + 1 + 2 \\
&\vdots \\
&\leq h(t) + k
\end{aligned}$$

But $h(t) = 0$ since h is a height function. So, $h(s) \leq k$ and $k < |V|$ i.e. $h(s) < |V|$ which contradicts the fact that $h(s) = |V|$ (since h is a height function). So no such path exists from s to t in G_f. $\square$

Theorem 1.6.2 *When the preflow-push algorithm terminates when run on a flow network $G = (V, E)$ with source s and sink t, then the preflow f it computes is indeed a maximum flow for G.*

Proof: When the preflow-push algorithm terminates, then the condition in the **while**-loop is not satisfied and thus there are no active vertices. In other words, no vertex in $V \backslash \{s, t\}$ has an excess flow (i.e., there are no overflowing vertices). Therefore, f is a flow for the flow network G. It is a maximum flow because we know that f is a maximum flow if and only if the residual network G_f contains no path from s to t (max-flow min-cut theorem) and from the previous theorem we know that since h is a height function, there are no paths from s to t in the residual graph.

So when the preflow-push algorithm terminates, it delivers a maximum flow from s to t. $\square$

1.6.5 Complexity of the preflow-push algorithm

We shall need the following theorem to find the running time function of the preflow-push algorithm.

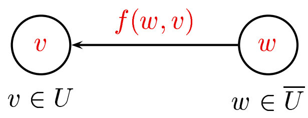
Theorem 1.6.3 *$G = (V, E)$ is a flow network with source s and sink t, and f is a preflow in G. For any overflowing vertex u in G, there exists a simple path from u to s in the residual network G_f.*

Proof: Suppose that $u \in V$ is overflowing (i.e. $e(u) > 0$). Let

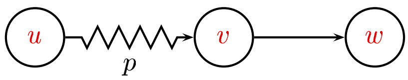
$$U = \{v | \ \exists \text{ a simple path from } u \text{ to } v \text{ in } G_f\}\ .$$

If we can show that $s \in U$, then we are done. The proof is by contradiction. Suppose that $s \notin U$. Let $\overline{U} = V - U$.

Claim: $f(w, v) \leq 0$ for any $w \in \overline{U}$ and any $v \in U$.



Proof of Claim: *(by contradiction)* If $f(w,v) > 0$, then $f(v,w) < 0$ but $c_f(v,w) = c(v,w) - f(v,w)$ (by definition), and so if $f(v,w) < 0$, we will have $c_f(v,w) > 0$, which means that there exists an edge (v,w) in the residual graph G_f.



And so in the residual graph G_f we have:

- a path p from u to v (since $v \in \overline{U}$)
- an edge from v to w

And therefore we have a path from u to w which contradicts the fact that $w \in \overline{U}$. So $f(w,v) \leq 0$.

Back to the main theorem. Since $f(w,v) \leq 0$ for every $w \in \overline{U}$ and every $v \in U$, we have $f(\overline{U}, U) \leq 0$.

$$\begin{aligned} e(U) &= f(V,U) \quad \text{by definition of excess flow} \\ &= f(\overline{U} \cup U, U) \\ &= f(\overline{U}, U) + f(U,U) \quad \text{by a property already studied} \end{aligned}$$

But $f(U,U) = 0$ and $f(\overline{U}, U) \leq 0$, so $e(U) \leq 0$.

By definition, an excess flow for any vertex in $V \backslash \{s\}$ is nonnegative, $U \subseteq V \backslash \{s\}$ and therefore we have $e(v) = 0$ for all vertices $v \in U$ (since $e(U) \leq 0$). In particular, $e(U) = 0$ which contradicts the fact that u is an overflowing vertex. So for any overflowing u in G, there exists a simple path from u to s in the residual network G_f. $\square$

We next find an upper bound on the heights of vertices.

Theorem 1.6.4 *At any time during the execution of the preflow-push algorithm, we have $h(u) \leq 2|V| - 1$ for all $u \in V$.*

Proof: Vertices s and t are never overflowing (by definition) and so the values of their heights never change: $h(s) = |V|$ and $h(t) = 0$. Vertices have their heights altered (increased) only when they are overflowing. For any overflowing vertex $u \in V \backslash \{s,t\}$, we have a simple path $p = < v_0, v_1, \ldots, v_k >$ with $u = v_0$, $s = v_k$ in the residual network G_f (by the previous theorem).

Note that $k \leq |V| - 1$ since p is simple. Moreover,

$$(v_i, v_{i+1}) \in E_f \Rightarrow h(v_i) \leq h(v_{i+1}) + 1$$

So

$$
\begin{aligned}
h(u) &\leq h(v_1) + 1 \\
&\leq (h(v_2) + 1) + 1 = h(v_2) + 2 \\
&\vdots \\
&\leq h(s) + k
\end{aligned}
$$

$h(u) \leq h(s) + k$, $h(s) = |V|$ and $k \leq |V| - 1$, so

$$h(u) \leq |V| + (|V| - 1) = 2|V| - 1$$

□

As an immediate result of the previous theorem, we can show that the number of lift operations is at most $2|V|-1$ per vertex and at most $(2|V|-1)(|V|-2) < 2|V|^2$ overall. Note that any vertex in $V\backslash\{s,t\}$ can be lifted, and when it is, its value grows from 0 (initially $h(u) = 0$) to at most $2|V|-1$ (by the previous theorem). So each vertex is lifted at most $2|V|-1$ times, and the total number of lift operations is at most $(2|V|-1)(|V|-2) < 2|V|^2$.

That was the bound on lift operations. To bound the push operations, we consider the saturating pushes and the nonsaturating pushes separately.

Theorem 1.6.5 *During the execution of the preflow-push algorithm, the number of saturating pushes is at most* $2|V||E|$.

Proof: Recall that $\textsc{Push}(u,v)$ is a saturating push if edge (u,v) becomes saturated ($c_f(u,v) = 0$) after the push, and that saturated edges do not appear in the residual network. When (u,v) is saturated, $h(u) = h(v) + 1$, and $(u,v) \notin E_f$.

For (u,v) to reappear in the residual network, we must send flow in (v,u) and thus have $h(v)$ increase by at least 2. Likewise, $h(u)$ must increase by at least 2 between saturating pushes from v to u.

Consider the sequence A of integers given by $h(u) + h(v)$ for each saturating push that occurs between vertices u and v.

We want to upper bound the length of sequence A. Note that after the first push (i.e. $\textsc{Push}(u,v)$ or $\textsc{Push}(v,u)$), we will have $h(u) + h(v) \geq 1$. In other words, the first integer in sequence A is at least 1. When the last push occurs, we will have:

$$
\begin{aligned}
h(u) + h(v) &\leq (2|V| - 1) + (2|V| - 2) \text{ recall } h(u) \leq 2|V| - 1 \\
&= 4|V| - 3
\end{aligned}
$$

So the last integer in sequence A is at most $4|V| - 3$. Since heights must increase by at least 2 between saturating pushes, the number of integers in sequence A is at most

$$\frac{(4|V| - 3) - 1}{2} + 1 = 2|V| - 1.$$

Note that 1 is added on the left hand side of the equality to make sure that the first and last numbers of sequence A are taken into consideration. The total number of saturating pushes between vertices u and v is at most $2|V| - 1$.

So the total number of saturating pushes is at most $(2|V| - 1)|E| < 2|V||E|$. □

We now would like to bound the nonsaturating pushes. We use a potential function to achieve the goal.

Theorem 1.6.6 *During the execution of the preflow-push algorithm, the number of nonsaturating pushes is at most* $4|V|^2(|V| + |E|)$.

Proof: Define a potential function $\Phi = \sum_{v \in X} h(v)$ where $X \subseteq V$ is the set of overflowing vertices. Initially, $\Phi = 0$, since we have no overflowing vertices. Recall that at each iteration of the **while**-loop, we have to perform one of the following operations:

a) a lift

b) a saturating push

c) a nonsaturating push

Let $\Phi(k)$ denote the potential value at step k of the algorithm. If at the k^{th} step, the algorithm picks $u \in X$ and performs a:

a) lift, then $\Phi(k) - \Phi(k-1) \leq 2|V|$, since only $u \in X$ is lifted (heights of other elements in X do not change) and u cannot be lifted by more than its maximum possible height.

b) saturating push, $\textsc{Push}(u, v)$, then $\Phi(k) - \Phi(k-1) \leq 2|V|$, since again no heights change and only vertex v, whose height is at most $2|V|$, can possibly have a positive excess flow.

c) nonsaturating push, $\textsc{Push}(u, v)$, then $\Phi(k) - \Phi(k-1) \leq -1$, since u is not an overflowing vertex after the push, v now joins X (even it wasn't an overflowing vertex at step $k-1$), and $h(v) - h(u) = -1$

If K denotes the last step of the algorithm, then

$$\begin{aligned}
\Phi(K) - \Phi(0) &= \sum_{k=1}^{K} \Phi(k) - \Phi(k-1) \\
&\leq (\text{max number of lifts})(2|V|) \\
&\quad +(\text{max number of saturating pushes})(2|V|) \\
&= (2|V|^2)(2|V|) + (2|V||E|)(2|V|) \text{ (from previous theorems)} \\
&= 4|V|^2(|V| + |E|)
\end{aligned}$$

So $\Phi(K) - \Phi(0) \leq 4|V|^2(|V| + |E|)$. Since $\Phi > 0$, the total decrease (due to $\Phi(k) - \Phi(k-1) \leq -1$ when a nonsaturating push is performed), and therefore the total number of nonsaturating pushes, is at most $4|V|^2(|V| + |E|)$. □

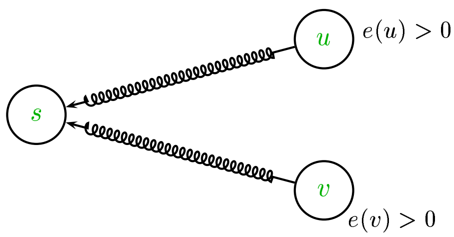
Since $|E| > |V|$, we have really shown the following theorem:

Theorem 1.6.7 *During the execution of the preflow-push algorithm on any flow network $G = (V, E)$, the number of basic operations is $O(|V|^2|E|)$.* □

Complexity of preflow-push algorithm – Summary

1. Existence of path from overflowing node to s in G_s.

   If u is overflowing $\Rightarrow$ $\exists$ a simple path from u to s in G_f.

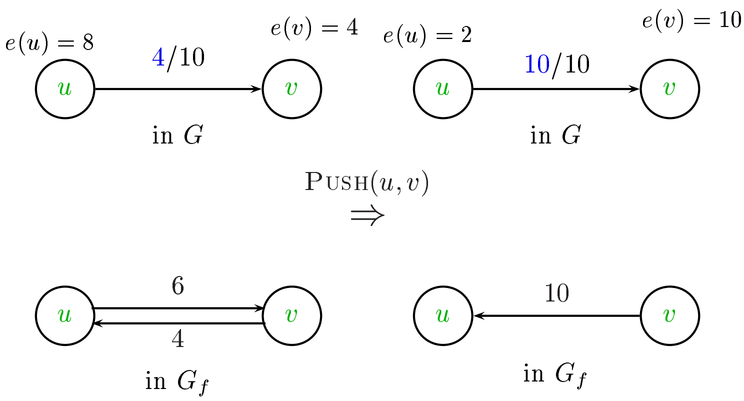


2. Upper bound on height of an vertex: $h(u) \le 2|V| - 1$ for all $u \in V$.
3. Number of lift operations:

$$\begin{gathered} < 2|V| - 1 \text{ per vertex} \\ < 2|V|^2 \text{ overall} \end{gathered}$$

4. Number of saturating pushes $< 2|V||E|$

   EXAMPLE:



$$A = \{a_1, a_2, \ldots, a_q\} \text{ find a bound on } q$$

a_i is the i^{th} saturated push. Because after the 1^{st} push $h(u) + h(v) \ge 1$, we know that $a_1 \ge 1$. We also know that $a_q \le 4|V| - 3$. Furthermore, $a_{i+1} - a_i \ge 2$ holds.

EXAMPLE: Consider the sequence

$$1, 3, 5, 7, 9, 11, 13, 15, 17, 19$$

Count the number of integers in this sequence:

$$\frac{\text{last} \ - \ \text{first}}{2} + 1 \ \Rightarrow \ \frac{19 - 1}{2} + 1 = 10$$

5. Number of nonsaturating pushes $< 4|V|^2(|V| + |E|)$. Use $\Phi = \sum_{v \in X} h(v)$, where X is the set of overflowing vertices. We denote by $\Phi(k)$ the potential value of the algorithm at step k.

$$\Phi(k) - \Phi(k-1) \le 2|V|$$

   and thus $\Phi(K) - \Phi(0) \le 4|V|^2(|V| + |E|)$ where K denotes the last step of the algorithm.

1.6.6 Specific implementations of the preflow-push algorithm

The preflow-push algorithm is flexible. By specifying different rules for selecting active nodes for the push and lift operations one can derive many different algorithms, each with different worst-case complexity than the preflow-push algorithm we have just studied.

The bottleneck operation in the preflow-push algorithm is the number of nonsaturating pushes. Different rules for examining active nodes can produce meaningful reductions in the number of nonsaturating pushes.

1.6.7 FiFo preflow-push algorithm

As the title suggests, the algorithm examines the active nodes in the first-in, first-out (FIFO) order [AMB93]. It can be shown, that this algorithm runs in $O(|V|^3)$ time. At any given iteration, we saw that the preflow-push algorithm selects an active node u and performs

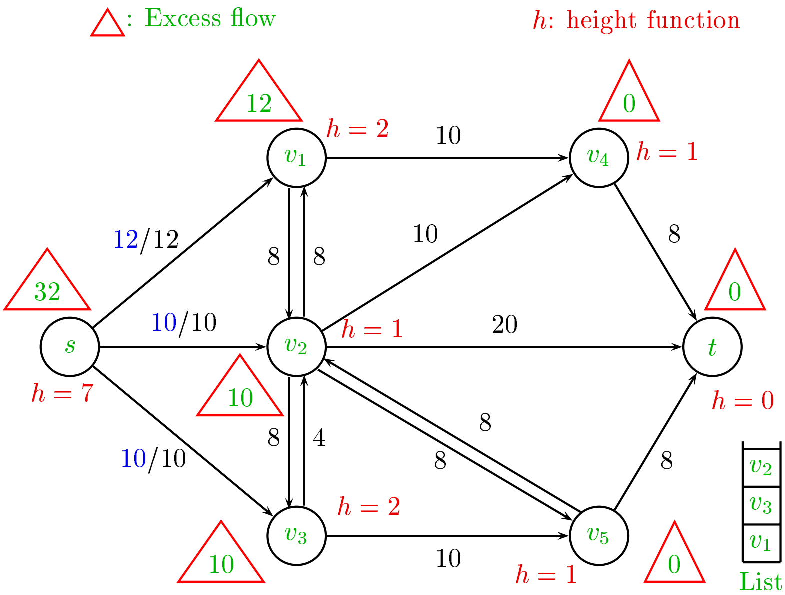
- a saturating push, PUSH(u, v), or
- a nonsaturating push, PUSH(u, w), or
- a lift operation, LIFT(u).

If the algorithm performs a saturating push, it is possible that u still remains active. The algorithm might select u again or might select another active node instead. In our FIFO preflow-push algorithm we want to force the selection of u again and keep pushing flow from u util either $e(u) = 0$ or the algorithm performs a LIFT(u) operation. Consequently, the algorithm might perform several saturating pushes followed either by a nonsaturating push or a lift operation. This sequence of operations is known as a node examination or we say that an overflowing vertex u is discharged by pushing all of its excess flow through admissible edges (u, v) (an edge (u, v) where $c_f(u, v) > 0$ and $h(u) = h(v) + 1$) to neighboring vertices, lifting u as necessary to cause edges leaving u to belong admissible.

The FIFO preflow-push algorithm examines active nodes in the FIFO order, by maintaining a set list as a queue. It selects a node u from the front of the list, performs pushes from this node, and adds newly active nodes to the rear of the list. The algorithm repeatedly considers u (i.e. examines u) until either it becomes inactive or it is lifted (in which case it is added to the end of the queue). As before, the algorithm terminates when the queue of active nodes is empty.

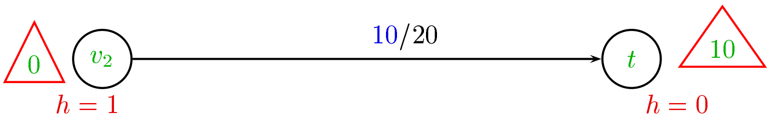
To cut down on the number of lifts that the algorithm has to perform in the early stages of the run, we do not initialize the height values to zero. Instead, we perform a backward breadth-first search starting at the sink t. So the sink will have a height of 0, vertices that share an edge with t will be of height one, vertices two hops from t have 2 as initial height, etc. As for the source, we keep the initial height value to be $|V|$. But the source's excess flow is initialized to $e(s) = \sum_{v \in V} c(s, v)$. We illustrate the FIFO preflow-push algorithm using the AAC example.
EXAMPLE: The initialization phase creates a network with height $h(t) = 0$, $h(v_2) = h(v_4) = h(v_5) = 1$, $h(v_1) = h(v_3) = 2$ and $h(s) = |V| = 7$.

As is the case with the preflow-push algorithm of the previous section, the algorithm then saturates edges leaving the source. Thus we'll have an excess flow of 12 crates at v_1 and an excess of 10 crates of olives at each of the vertices v_2 and v_3. Suppose that the queue of active nodes at this stage is LIST $= \{v_2, v_3, v_1\}$. The algorithm next removes vertex v_2 from the queue and "examines it.

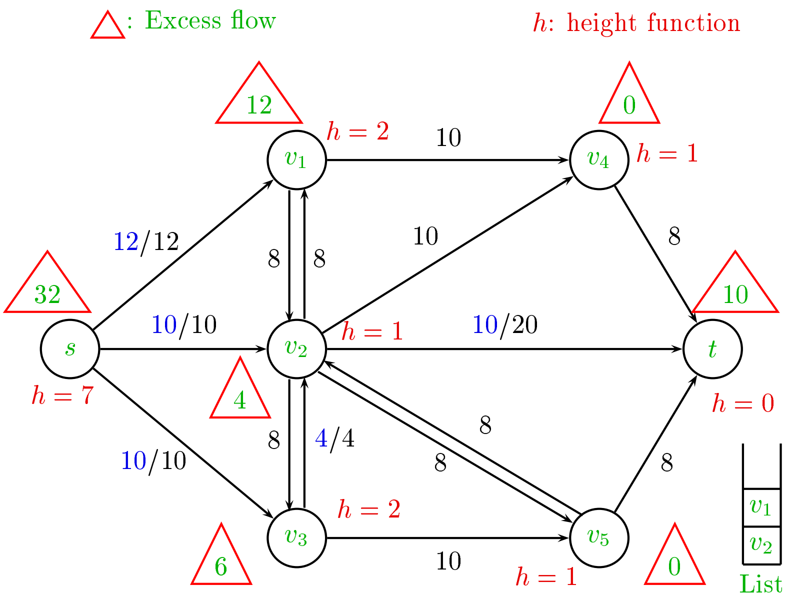


Snapshot of network after saturating the edges leaving s

At this stage, the algorithm performs a nonsaturating push, PUSH(v_2, t), and v_2 gets rid of its excess overflow.



Note that v_2 is not an active node after performing PUSH(v_2, t). So the algorithm next examines v_3 (the current node in LIST). At this stage, the algorithm could perform a saturating push, PUSH(v_3, v_2) of 4 units or a saturating push, PUSH(v_3, v_5) of 10 units. Suppose it does PUSH(v_3, v_2).



So now v_2 has an excess flow of 4 units and is therefore put in the queue. As mentioned earlier, the algorithm stays with v_3 since $e(v_3) = 6 > 0$, and performs $\textsc{Push}(v_3, v_5)$ [a nonsaturating push of 6 units]. So now v_5 is active and enters the queue. v_3 is not active anymore and the algorithm then considers vertex v_1. And so on.

Theorem 1.6.8 *The* FiFo *preflow-push algorithm runs in* $O(|V|^3)$ *time.*

Proof: To analyze the worst-case complexity of the FiFo preflow-push algorithm, we partition the total number of node examinations into several phases.

Phase one: Node examination of the nodes that become active because of the saturating of the edges that leave s (i.e. the last step in the algorithm before the **while**-loop). In our example, phase one consists in examining nodes v_1, v_2, v_3.

Phase two: Node examination of all the nodes that are in the queue after the algorithm has examined the nodes of phase one. In our example, phase two consists in examining nodes v_2, v_5, v_4.

Similarly, *phase three* consists of the node examination of all the nodes that are in the queue after the algorithm has examined the nodes in the second phase. And so on.

Notice that the algorithm examined a node at most once during a phase.

We shall show that the FiFo preflow-push algorithm performs at most $2|V|^2+|V|$ phases. Each phase examines any node at most once and each node examination performs at most **one** nonsaturating push. So if we can proof that we have at most $2|V|^2 + |V|$ phases, then we'll have at most $|V|(2|V|^2 + |V|)$ nonsaturating pushes. But recall that the number of nonsaturating pushes is the bottleneck of the preflow-push algorithm (and is also the bottleneck of the FiFo preflow-push algorithm). So this would imply that the FiFo preflow-push algorithm runs in $O(|V|^3)$.

Claim: FiFo preflow-push performs at most $2|V|^2 + |V|$ phases
Proof of Claim: We consider the total change in the potential function $\Phi = \max\{h(u)|u \text{ is active }\}$ over an entire phase.

Once again, we would like to compute the difference between the initial and final values of the potential function during a phase. Note that the potential function here is slightly different than the one we use in the previous section. Back to the proof of the claim. We consider two cases:

Case one: FiFo preflow-push performs at least one lift operation during a phase. We already saw in the previous section that $h(u) \leq 2|V| - 1$ for any $v \in V$, and that the total number of lift operations is at most $2|V|^2$. Since Φ cannot increase more than the maximum increase produced by a lift operation, the total increase in Φ over all the phases is at most $2|V|^2$.

Case two: FiFo preflow-push performs no lift operation during a phase. As was seen in the previous section, Φ decreases by at least 1 unit, since the excess of every node that was active at the beginning of phase moves to nodes with smaller weights.

So by combining cases one and two, and by realizing that the initial value of Φ could be at most $|V|$, we have that the total number of phases is at most $2|V|^2 + |V|$.

So this proves the claim that FiFo preflow-push performs at most $2|V|^2 + |V|$ phases. By the remarks preceding the claim, we have proven that the FiFo preflow-push algorithm runs in $O(|V|^3)$ time. □

Other possible implementations of the preflow-push algorithm include the highest-label preflow algorithm that always pushes flow from an active node with the highest distance value, and the excess scaling algorithm. The highest-label preflow push algorithm runs in $O(|V|^2\sqrt{|E|})$ and is considered to be the most efficient maximum flow algorithm in practice.

1.7 Applications of the maximum flow problem

1.7.1 Scheduling on uniform parallel machines

We are interested in scheduling a set of J jobs on M uniform parallel machines [AMB93]. Each job $j \in J$ has:

p_j: number of machine days required to complete the job

r_j: release date, i.e. the beginning of the day when job j becomes available for processing

d_j: due date, that is the beginning of the day by which the job must be completed

Note that $r_j + pj \le d_j$.

We make the following assumptions:

1. A machine can work on only one job at a time
2. Each job can be processed by at most one machine at a time
3. Preemption is allowed. In other words, we can interrupt a job and process it on different machines on different days.

The scheduling problem consists in determining a feasible schedule that computes all jobs before their due dates or to show that no such schedule exists.

Note that scheduling problems of this kind arise in batch processing systems involving batches with a large number of units.

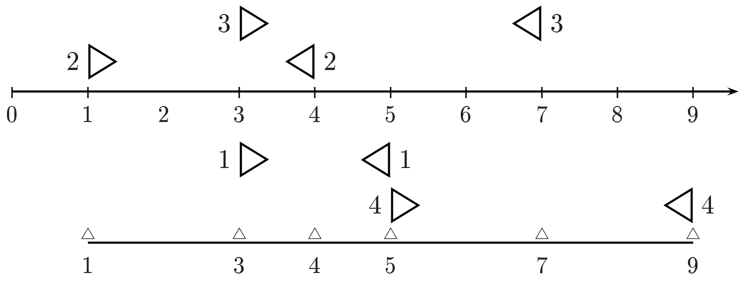
The feasible scheduling problem as defined here can be used by more general scheduling problems, such as

- the maximum lateness (tardy) problem
- the (weighted) minimum completion time problem
- the (weighted) maximum utilization problem

The feasible scheduling problem could be formulated as a maximum flow problem. We show the transformation of this problem into a maximum flow problem by considering the following example.
Example: Suppose we have $M = 3$ machines and 4 Jobs as given in the table:

Job	j	1	2	3	4
Processing time	p_j	1.5	1.25	2.1	3.6
Release time	r_j	3	1	3	5
due time	d_j	5	4	7	9

Let us rank all the release and due date, r_j and d_j for all j in ascending order and determine mutually disjoint intervals of dates between consecutive "milestones".



By ordering r_j and d_j in ascending order, we get:

$$1, 3, 4, 5, 7, 9$$

and so we get 5 mutually disjoint intervals of dates (or we can allow the intervals to share end points):

$$[1, 3], [3, 4], [4, 5], [5, 7], [7, 9]$$

Note that within each interval, the set of jobs that are available (i.e., jobs that have been released but not due yet) does not change.

Another way of seeing the 5 intervals is by considering the graph above, where j denotes the release time of j and j denotes the due date of job j.

Note that the maximum number of disjoint intervals we can have is $2|J| - 1$ (we have 2 entities for each job: due and release dates) where $|J|$ is the number of jobs.

Let $T_{k,l}$ denote the interval that starts at the beginning of day k and ends at the beginning of day $l + 1$. In our example, the five intervals are:

$$T_{1,2} \quad T_{3,3} \quad T_{4,4} \quad T_{5,6} \quad \text{and} \quad T_{7,8}$$

Notice that within each interval $T_{k,l}$ we can process all jobs with

a) $r_j \leq k$ and

b) $d_j \geq l + 1$

in the interval. For instance, in the interval 3 to 4, i.e. $T_{3,3}$ we can process the following jobs:

- 2 since $r_2 \leq 3$ and $d_2 = 4 \geq 3 + 1$
- 1 since $r_1 \leq 3$ and $d_1 = 5 \geq 3 + 1$ and
- 3 since $r_3 \leq 3$ and $d_3 = 7 \geq 3 + 1$

To cast this problem as a maximum flow problem, we create 4 levels of vertices:

Level one: The source node s.

Level two: Nodes corresponding to the $|J|$ jobs (one vertex per job).

Level three: Interval vertices; one vertex per interval.

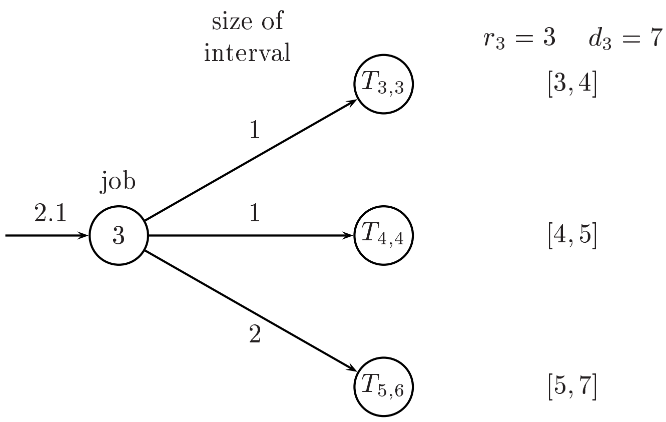
Level four: The sink node t.

As for the edges and capacities:

- Connect the source node to every job vertex in level two with an edge with capacity p_j. This means that we need to assign p_j days of machine time to job j.
- Connect each job vertex j to every interval vertex $T_{k,l}$ if

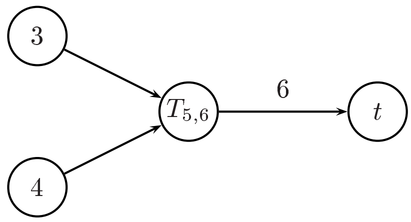
$$r_j \leq k \text{ and } d_j \geq l + 1$$

by an edge with capacity $l - k + 1$ which represents the maximum number of machine days we can allot to job j on the days from k to l. For instance:



Job 3 is ready for processing at $r_3 = 3$ and has a due date of $d_3 = 7$. Its $p_3 = 2.1$ of processing time could be scheduled during intervals $T_{3,3}$, $T_{4,4}$ and/or $T_{5,6}$.

- Connect each interval node $T_{k,l}$ to t by an edge with capacity $(l - k + 1)M$, representing the total number of machine days available on the days from k to l. For instance:

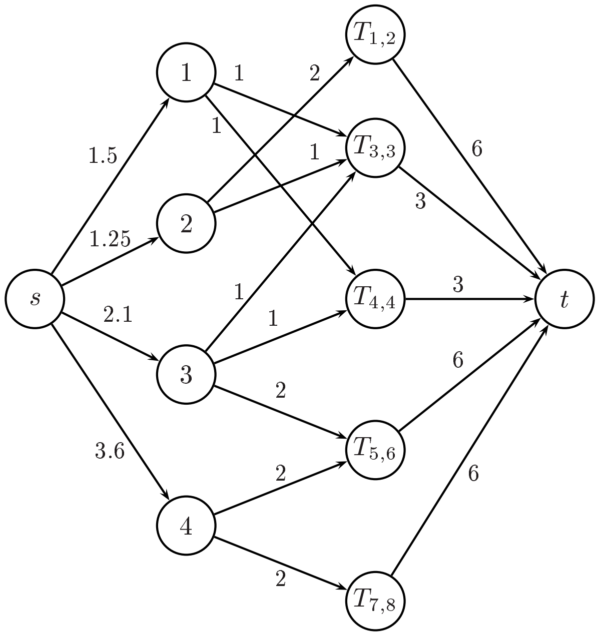


The size of the interval is $6 - 5 + 1 = 2$. But this represents the number of days per machine. So we have 6 machine days (3×2) to service the processing demands of jobs 3 and 4 that arrive at node $T_{5,6}$. □

So now we solve the problem and find a maximum flow.

$$\left.\begin{array}{c}\text{The scheduling problem}\\ \text{has a feasible solution}\end{array}\right\} \Leftrightarrow \begin{array}{c}\text{The maximum flow}\\ \text{is equal to } \sum\limits_{j \in J} p_j\end{array}$$

□



□

1.7.2 Distributed computing on a two-processor computer

A computer system has two (not necessarily identical) processors. We wish to execute a very large program that consists of several modules (subroutines) that will need to interact with each other during the execution of the program [AMB93]. The cost of executing a module on a processor is known in advance. Let

α_i: cost of computation of module i on processor 1

β_i: cost of computation of module i on processor 2

In general $\alpha_i \neq \beta_i$ because the memory, control, speed and arithmetic capabilities of the two processors are not the same. Let

c_{ij}: interprocess communication cost when modules i and j are assigned to different processors

Note that $c : ij$ is in general quite high and that if instead, modules i and j are assigned to the same processor, we do not have any cost.

We wish to allocate modules of the program on the two processors so that we minimize the total cost of processing and interprocess communication. This problem can be formulated as a minimum cut problem on undirected network. We show the formulation by considering an example.

Example: Suppose that a program can be broken into 4 modules and that the costs of computation of the different modules on processors one and two are as given as in table 1.1.

Moreover, the interprocess communication costs, c_{ij}, of assigning module i and j on different processors are given in the symmetric table 1.2 below. So for instance,

$$c_{3,2} = 6$$

To cast this problem as a minimum cut problem on an undirected network, we create three levels of vertices:

i	1	2	3	4	Modules
α_i	6	5	10	4	Processor one
β_i	4	10	3	8	Processor two

Table 1.1: "In-house" costs

	1	2	3	4
1	0	5	0	0
2	5	0	6	2
3	0	6	0	1
4	0	2	1	0

Table 1.2: Interprocess costs

Level one: source vertex s which represents processor one

Level two: vertices corresponding to the program modules: one vertex per module

Level three: sink vertex t which represents processor two

As for the edges and capacities:

a) connect the source node to every module vertex i in level two with an edge with capacity β_i

b) connect two module vertices i and j with an edge (i, j) of capacity c_{ij} if modules i and j interact during the execution of the program

c) connect every module vertex i with t to form arc and give a capacity value of α_i

Notice that there is a one-to-one correspondence between s–t cuts in the network and assignments of modules to the two processors.

Consider the extreme case of scheduling all four modules on processor two. Here, there is no interprocess communication happening and the cost is just

$$\sum_{i=1}^{4} \beta_i = 4 + 10 + 3 + 8 = 25$$

Now consider the cut $c(\{s\}, V - \{s\})$ of the network. Note that its cost is $4 + 10 + 3 + 8 = 25$.

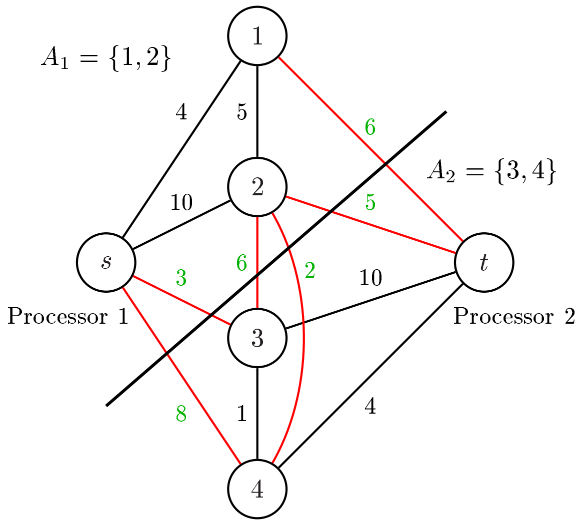
Indeed we shall always have the capacity of a cut equal to the cost of the corresponding assignment. In other words, if A_1 is the set of modules assigned to processor one and A_2 is the set of modules assigned to processor two, then the cost of this assignment is

$$\sum_{i \in A_1} \alpha_i + \sum_{i \in A_2} \beta_i + \sum_{(i,j) \in A_1 \times A_2} c_{ij}$$

The s–t cut correspondence to this assignment is

$$(\{s\} \cup A_1, \{t\} \cup A_2)$$

Example: With $A_1 = \{1, 2\}$ and $A_2 = \{3, 4\}$ we get the following flow network:



As shown in above figure,

$$
\begin{aligned}
c(\{s\} \cup A_1, \{t\} \cup A_2) &= 8 + 3 + 6 + 2 + 5 + 6 \\
&= 30
\end{aligned}
$$

In our example of $A_1 = \{1, 2\}$ and $A_2 = \{3, 4\}$ we saw that

$$c(\{s\} \cup A_1, \{t\} \cup A_2) = 30 \ .$$

Note that this cut contains:

1. edge $(1, t)$ of capacity α_1 and edge $(2, t)$ of capacity α_2, which implies that modules 1 and 2 are assigned to process 1

2. edge $(s, 3)$ of capacity β_3 and edge $(s, 4)$ of capacity β_4 which represents the fact that modules 3 and 4 are assigned to process 2

3. edge $(2, 3)$ of capacity c_{23} and edge $('', 4)$ of capacity c_{24} which represents the interprocess cost for the assignment of modules 1 and 2 to processor one and modules 3 and 4 to processor two.

   We showed that the cost of the assignment A_1 and A_2 = capacity of the cut $(\{s\} \cup A_1, \{t\} \cup A_2)$. This is true for any assignment A_1, A_2. Consequently, the minimum s–t cut in the network gives the minimum cost assignment of the modules to the two processors. □

1.7.3 The baseball elimination problem

This application has nothing to do with baseball (especially the way it is played now).

There are n teams: team 1, team 2, ..., team n. Team n is our favorite team. We are in mid-season and we are interested in knowing if team n still has a chance of ending the season in first place [AMB93] [Wes96]. We assume that no ties are possible.

Suppose that team i has won $w(i)$ of the games it has already played, and that $g(i, j)$ is the number of games that team i and team j have yet to play with each other.

We say that team t is eliminated if it has no chance of finishing first; in other words, if for every possible outcome of the unplayed games, at least one team will have more wins than team t.

Team n currently has $w(n)$ wins, and there are

$$\sum_{j=1}^{n-1} g(j,n)$$

games remaining on its schedule. The best scenario for team n is if it wins all of its remaining games. Let

$$W = w(n) + \sum_{j=1}^{n-1} g(j,n)$$

Team n cannot be eliminated if the rest of the games between the other teams is such that each other team has at most W wins at the end of the season. Recall there are $g(i,j)$ games remaining between team i and team j.

Let $x(i,j)$ denote the number of games that team i wins over j (from the $g(i,j)$ games they still have to play). Note that

$$\begin{aligned} x(i,j) + x(j,i) &= g(i,j) \\ &= g(j,i) \\ \text{for each } j = 1,2,\ldots,n & \\ \text{and } i = 1,2,\ldots,n & \end{aligned}$$

Let $W(i) = w(i) + \sum_{j=1}^{n} x(i,j)$ denote the number of wins that team i has at the end of the season, for $i = 1,2,\ldots,n-1$. Team n finishes the season in first place if there is a way of completing the season such that

$$W \geq W(i) \text{ for } i = 1,2,\ldots,n-1$$

In other words, team n is not eliminated if there is some assignment of integer values $x(i,j) \geq 0$, $i = 1,2,\ldots,n$ such that

$$\begin{aligned} x(i,j) + x(j,i) &= g(i,j) \\ &= g(j,i) \\ \text{for } i = 1,2,\ldots,n-1 & \\ \text{and } j = 1,2,\ldots,n-1 & \end{aligned}$$

and

$$W = w(n) + \sum_{j=1}^{n-1} g(j,n) \geq w(i) + \sum_{j=1}^{n-1} x(i,j) \text{ for each } i = 1,2,\ldots,n-1$$

Our problem is to decide whether team n is eliminated or not.

While in some problem instances it is easy to answer this question, in others it is a challenging question.

Example: There are 4 teams, and in the entire season, each team plays each of the other 3 teams for 3 games each. In other words, each team plays a 9–game season.

Suppose that team 1 has already played and beaten team 3 twice and team 4 twice. In other words

$$g(1,3) = 3 - 2$$

and

$$g(1,4) = 1$$

and $w(1) = 4$. Similarly, suppose that

$$g(2,3) = 1 g(2,4) = 1 \text{ and } w(2) = 4$$

and these are the only games played so far. So $w(3) = 0$ and $w(4) = 0$ (see table below).

Team i	wins $w(i)$
1	4
2	4
3	0
4	0

Table 1.3: Wins so far in the season

And there are quite a few remaining games, as the following table shows.

$i \backslash j$	1	2	3	4
1	—	3	1	1
2	3	—	1	1
3	1	1	—	3
4	1	1	3	—

Table 1.4: The remaining games

Is team 4 eliminated ?

$$\begin{aligned} w &= w(4) + \sum_{j=1}^{3} g(j,4) \\ &= 0 + 5 \\ &= 5 \end{aligned}$$

None of the teams has more than 4 wins so far in the season. But team 1 and team 2 still have to play 3 games. At the end of these games, either

$$x(1,2) \geq 2$$

(i.e. team 1 wins at least two of the 3 games) or

$$x(2,1) \geq 2$$

(that is, team 2 wins at least two of the 3 games). Thus either

$$W(1) = w(1) + \sum_{j=2}^{4} x(1,j) \geq 4 + 2 = 6$$

or

$$W(2) = w(2) + x(2,1) + x(2,3) + x(2,4) \geq 4 + 2 = 6$$

So either $W(1) > W$ or $W(2) > W$ and therefore our favorite team (team 4) is eliminated. $\square$

The following are 2 equivalent ways of thinking of the problem: Can the games between all of the teams $1, 2, \ldots, n-1$ be played such that:

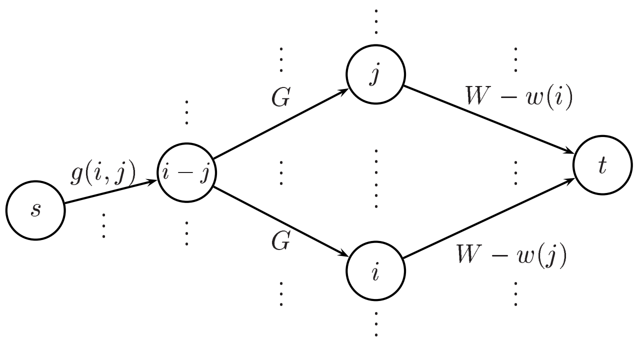
1. ate the end of the season, each team's total does not exceed W ?

2. by the end of the season, team i wins at most $W - w(i)$ of its remaining games, for $i = 1, 2, \ldots, n-1$?

The formulation of this problem as a maximum flow problem is based on the second interpretation where the commodity that is being shipped from the source s to the sink t is "remaining games between team i and team j for $i = 1, 2, \ldots, n-1$ and $j = 1, 2, \ldots, n-1$. We transform the baseball elimination problem into a maximum flow problem.

Create a source s, pair nodes (a.k.a. game nodes), one for each pair of teams, team nodes, one for each team $i = 1, 2, \ldots, n-1$ and a sink t. Note that there are $\frac{(n-1)(n-2)}{2}$ pair nodes.

We create an edge leaving s to each $< i - j >$ game node and give it capacity $g(i,j)$.

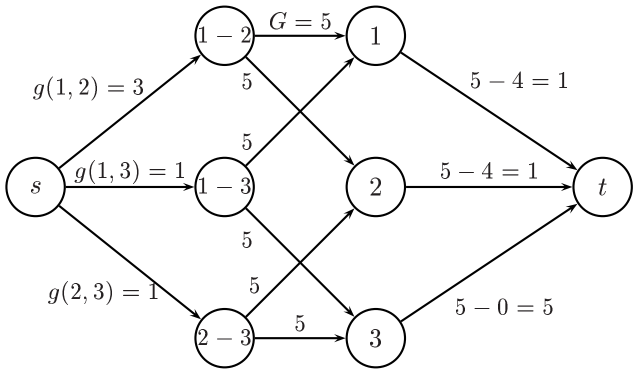


Create an edge leaving game node $< i - j >$ to team node i and another to team node j. The capacity of each of these edges is G, where G is the total number of games remaining among teams $\{1, 2, \ldots, n-1\}$. In other words

$$\begin{aligned} G &= \sum_{i=1}^{n-1} \sum_{j=1}^{i-1} g(i,j) \\ &= \sum_{< i - j > \text{ is a pair node}} g(i,j) \end{aligned}$$

Create an edge from each team node team i to t with capacity $W - w(i)$. Note that we are assuming that $W \geq w(i)$. If that is not the case, then the current number of wins of team i $w(i)$ is greater than W and so team n is eliminated (and no need to construct the network).

EXAMPLE: Let us go back to our example.



Recall that the flow here is "remaining game". A unit of flow from s to game node $< i - j >$ represents a game between team i and team j. The unit of flow then either goes to team i or team j, depending on whether team i or team j won. The unit of flow will go from the winning node, team i lets say to sink t.

The capacity of the edge from the source to game node $< i - j >$ is $g(i, j)$ and so will ensure that we cannot have more than $g(i, j)$ games between team i and team j.

Similarly, the capacity of $W - w(i)$ on the edge between team i and t will ensure that at most $W - w(i)$ games are won by team i (among its remaining games). Obviously, if that is not the case, then team n is indeed eliminated.

Also note that the cut $(\{s\}, V - \{s\})$, where V is the set of all vertices, is such that for its capacity holds

$$\begin{aligned} c(\{s\}, V - \{s\}) &= \sum_{< i - j > \text{ is a pair node}} g(i, j) \\ &= G \end{aligned}$$

Next we show that:

$$\text{Team } n \text{ is noteliminated} \quad \Leftrightarrow \quad \text{Maximum flow } = G$$

Proof:

($\Rightarrow$): We want to show that:

$$\text{Team } n \text{ is not eliminated} \quad \Rightarrow \quad \text{Maximum flow } = G$$

Team n is not eliminated, hence there is some assignment of integer values $x(i, j) \geq 0$, $i = 1, 2, \ldots, n$ and $j = 1, 2, \ldots, n$ such that

a) $x(i, j) + x(j, i) = g(i, j) = g(j, i)$ for $i = 1, 2, \ldots, n-1$ and $j = 1, 2, \ldots, n-1$

and

b) $W = w(n) + \sum_{j=1}^{n-1} g(j, n) \geq w(i) + \sum_{j=1}^{n-1} x(i, j)$ for each $i = 1, 2, \ldots, n-1$.

We define a flow function f in the following way: For each edge leaving s, we have

$$f(s, < i - j >) = g(i, j)$$

This assignment satisfies the capacity constraint of the edges $(s, < i - j >)$. For each edge $(< i - j >, i)$, let

$$f(< i - j >, i) = x(i, j)$$

and

$$f(< i - j >, j) = x(j, i)$$

for edge $(< i - j >, j)$.

Since $g(i, j) = x(i, j) + x(j, i)$ (by a)) the node flow conservation constraint at each game vertex $< i - j >$ is satisfied. In other words:

$$\begin{array}{ccc} \text{flow in vertex} & & \text{flow out of} \\ < i - j >\text{:} \quad g(i,j) & = & \text{vertex } < i - j >\text{:} \\ & & f(< i - j >, i) + f(< i - j >, j) \end{array}$$

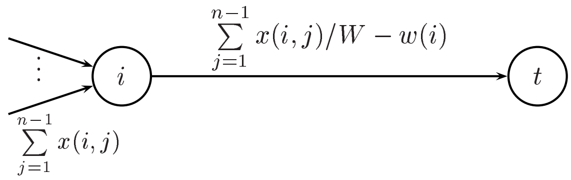
The total flow assigned to enter team i is

$$\sum_{j=1}^{n-1} x(i, j)$$

So we assign the following flow to each edge (i, t):

$$f(i, t) = \sum_{j=1}^{n-1} x(i, j)$$

Here too, the node flow conservation constraint is satisfied at each team i node.



Note that capacity constraint is also satisfied: Capacity of edge $(i, t) = W - w(i)$. Assigned flow is: $f(i, t) = \sum_{j=1}^{n-1} x(i, j)$. But by $b)$ we know that

$$W \geq w(i) + \sum_{j=1}^{n-1} x(i, j) \text{ for each } i = 1, 2, \ldots, n-1$$

So $W - w(i) \geq \sum\limits_{j=1}^{n-1} x(i, j)$. In other words, $f(i, t) \leq W - w(i)$. Let us compute the value of f:

$$\begin{aligned} |f| &= \sum_{< i - j > \text{ pair node}} f(s, < i - j >) \text{ (by definition of flow value)} \\ &= \sum_{< i - j > \text{ pair node}} g(i, j) \text{ (by our flow assignment)} \\ &= G \end{aligned}$$

But there exists a cut of capacity G. Recall that cut $(\{s\}, V - \{s\})$ has capacity

$$c(\{s\}, V - \{s\}) = G$$

Therefore f as defined here is a maximum flow and $|f| = G$ (by the Max Flow-Min Cut theorem).

($\Leftarrow$): We want to show:

$$\text{Maximum flow } = G \quad \Rightarrow \quad \begin{array}{c}\text{Team } n \text{ is not} \\ \text{eliminated}\end{array}$$

Suppose that we have a maximum flow function f and $|f| = G$. Assume that f assigns integer values to each edge. It is a reasonable assumption since all capacities are integer values.

To show that team n is not eliminated, we have to show that there is an assignment of integer values $x(i,j) \geq 0$ for $i = 1, 2, \ldots, n$ and $j = 1, 2, \ldots, n$ such that:

a) $x(i,j)+x(j,i) = g(i,j) = g(j,i)$ for $i = 1, 2, \ldots, n-1$ and $j = 1, 2, \ldots, n-1$

and

b) $W = w(n) + \sum_{j=1}^{n-1} g(j,n) \geq w(i) + \sum_{j=1}^{n-1} x(i,j)$ for each $i = 1, 2, \ldots, n-1$.

As seen previously, $c(\{s\}, V - \{s\}) = G$ (and $|f| = G$), so:

$$f(s, < i-j >) = g(i,j) \text{ for each game node } < i-j >$$

Set $\overline{x}(i,j) = f(< i-j >, i)$ and $\overline{x}(j,i) = f(< i-j >, j)$. Since $g(i,j)$ units of flow f go into game node $< i-j >$ and there are only two edges leaving that node, we know by the conservation constraint that

$$\begin{aligned} g(i,j) &= f(s, < i-j >) \text{ in flow} \\ &= f(< i-j >, i) + f(< i-j >, j) \text{ out flow} \\ &= \overline{x}(i,j) + \overline{x}(j,i) \end{aligned}$$

So property $a)$ is satisfied by our assignment of $\overline{x}$. Let us show that property $b)$ is also met. Note that $\sum_{j=1}^{n-1} \overline{x}(i,j)$ units of flow f enter team i vertex.

Since there is only one edge leaving team i (going to the sink), we have

$$f(i,t) = \sum_{j=1}^{n-1} \overline{x}(i,j) \text{ for each } i = 1, 2, \ldots, n-1$$

Since f is a flow, it satisfies the capacity constraints. So we have

$$f(i,t) \leq W - w(i) \text{ for } i = 1, 2, \ldots, n-1 \ .$$

But since $f(i,t) = \sum_{j=1}^{n-1} \overline{x}(i,j)$ for $i = 1, 2, \ldots, n-1$, we have

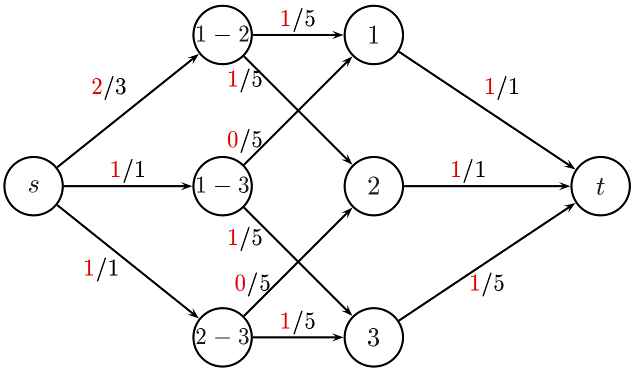
$$\begin{aligned} \sum_{j=1}^{n-1} \overline{x}(i,j) &\leq W - w(i) \text{ for } i = 1, 2, \ldots, n-1 \\ \Rightarrow W &\geq w(i) + \sum_{j=1}^{n-1} \overline{x}(i,j) \ . \end{aligned}$$

Therefore $\overline{x}$ satisfies property $b)$. Also $\overline{x}(i,j)$ are all non-negative integer values (since we assumed that flow f takes only integer values - apply the integrality property). So team n is not eliminated and so we have shown that

$$\begin{array}{c}\text{Team } n\text{is not} \\ \text{eliminated}\end{array} \quad \Rightarrow \quad \text{Maximum flow} = G$$

$\square$

Example:



By computing a maximum flow for our problem, we get the flow in the above figure with values in red. So

$$|f| = 4$$

According to the theorem, team 4 is eliminated, since the total number G of games remaining among teams $1, 2, 3$ is $5 \geq 4$. $\square$

Bibliography

[AMB93] Ravindra K. Ahuja, Thomas L. Magnanti, and Orlin James B. *Network Flows.* Prentice-Hall, Inc., 1993.

[CLR90] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms.* MIT Press, Cambridge, 1990.

[Wes96] Douglas B. West. *Introducation to Graph Theory.* Prentice Hall, Inc., 1996.