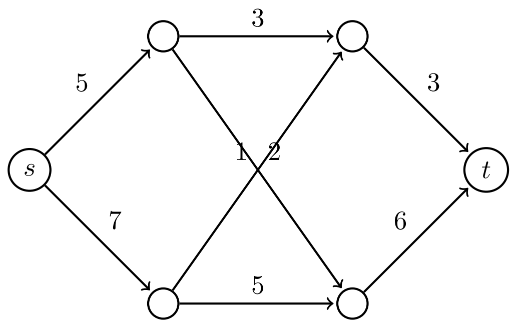
Chapter 3

Network Flows

Definition 3.1. A *network* is a weighted directed graph with 2 distinguished vertices s (called *source*) and t (called *sink*), where s has only outgoing edges and t has only incoming edges. For a network, the weight function is called *capacity function* (denoted by c).

Example 3.1. Here is an example of network, to each edge is attached a capacity.



3.1 Maximum Flow, Minimum Cut

Definition 3.2. A *flow* on a network $G = (V, E)$ is a map $f : E \to \mathbb{R}^+$ such that

(1) $0 \leqslant f(e) \leqslant c(e), \forall\, e \in E$ $(flow\ is\ feasible)$

(2) $\displaystyle\sum_{u \in I(v)} f(u,v) = \sum_{u \in O(v)} f(v,u), \forall\, v \in V \backslash \{s,t\}$ $(flow\ conservation)$

Here, $O(v)$ (resp. $I(v)$) is the set of vertices which have an edge coming from (resp. going into) v.

Sometimes, the term "feasible" refers to both conditions instead of just the first one.

Problem 4. Given a network $G = (V, E)$, find a maximum flow.

The sum of the values of a flow f on the edges leaving the source is called the *strength* of the flow (denoted by $|f|$), that is

$$|f| = \sum_{u \in O(s)} f(s, u).$$

It can be shown (see Exercise 25) that $|f| = \sum_{u \in I(t)} f(u, t)$, that is, the sum of the values of f leaving the source is also the sum of the values of f entering the sink.

Definition 3.3. A *cut* of a network $G = (V, E)$ with source s and sink t is a decomposition $V = S \sqcup T$ of V into disjoint subsets S, T such that $s \in S$, $t \in T$. Such a cut is denoted by (S, T). The *capacity* of (S, T) is defined as

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

A *minimum cut* is a cut of minimum capacity.

Lemma 3.1. *For any cut (S, T), $|f| \leqslant C(S, T)$.*

Proof. First, we prove by induction on $|S|$ that

$$|f| = \sum_{\substack{u \in S \\ v \in T}} f(u, v) - \sum_{\substack{u \in S \\ v \in T}} f(v, u). \tag{3.1}$$

See Example 3.2 for an illustration of this claim. If $|S| = 1$, then S contains only the source s. Then

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

Suppose true for $|S| > 1$. Now move one vertex w from T to S and we want to compute

$$\sum_{\substack{u \in S \cup \{w\} \\ v \in T \setminus \{w\}}} f(u, v) - \sum_{\substack{u \in S \cup \{w\} \\ v \in T \setminus \{w\}}} f(v, u).$$

Then we both have an increase by

$$\sum_{x \in O(w)} f(w, x)$$

(if $x \in T$ before, it adds up after, if $x \in S$ before it used to count negatively, so it also adds up after) and similarly a decrease by

$$\sum_{y \in I(w)} f(y, w),$$

that is:

$$\sum_{\substack{u\in S\cup\{w\}\\ v\in T\setminus\{w\}}} f(u,v) - \sum_{\substack{u\in S\cup\{w\}\\ v\in T\setminus\{w\}}} f(v,u) = \sum_{\substack{u\in S\\ v\in T}} f(u,v) - \sum_{\substack{u\in S\\ v\in T}} f(v,u) + \sum_{x\in O(w)} f(w,x) - \sum_{y\in I(w)} f(y,w).$$

But flow conservation must hold. So the total change

$$\sum_{x\in O(w)} f(w,x) - \sum_{y\in I(w)} f(y,w)$$

after moving w from T to S is 0 which concludes the proof by induction.

Now

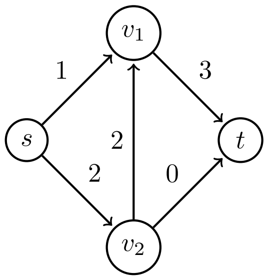
$$|f| = \sum_{\substack{u\in S\\ v\in T}} f(u,v) - \sum_{\substack{u\in S\\ v\in T}} f(v,u) \leqslant \sum_{\substack{u\in S\\ v\in T}} f(u,v) \leqslant \sum_{\substack{u\in S\\ v\in T}} c(u,v) = C(S,T)$$

□

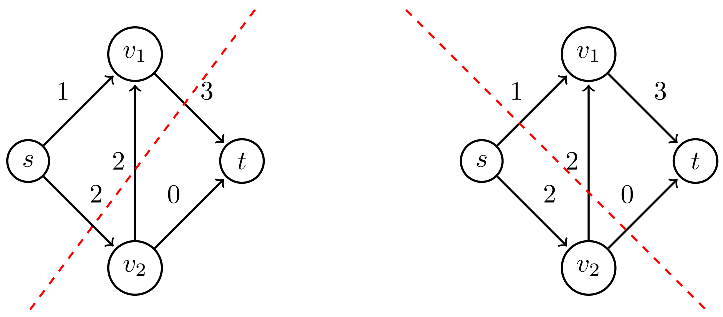
In particular, if we pick a minimum cut in the above lemma, then we get an upper bound on the maximum flow:

$$\max |f| \leq \min_{S,T} C(S,T).$$

Example 3.2. Different cuts of a network whose flow f is described on the network edges each gives the strength $|f| = 3$ according to (3.1).



Cut 1 : $|f| = 3 + 2 - 2 = 3$ Cut 2 : $|f| = 1 + 2 + 0 = 3$

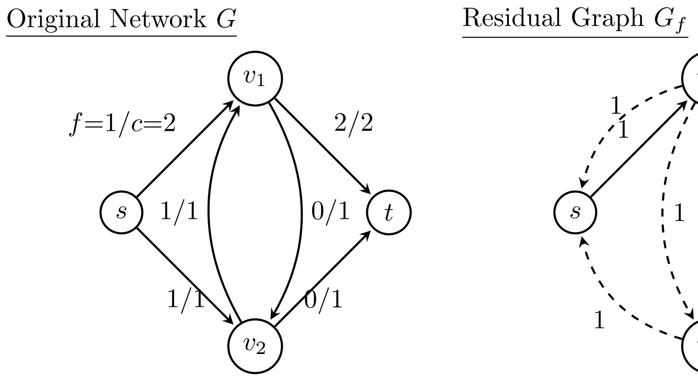


Definition 3.4. Given a flow f on a graph (network) $G = (V, E)$, the corresponding *residual graph* G_f has V for vertices, and

(1) edges have capacities $c(v, w) - f(v, w)$, only edges with $c_f := c(v, w) - f(v, w) > 0$ are shown, and they are shown in the same direction as (v, w),

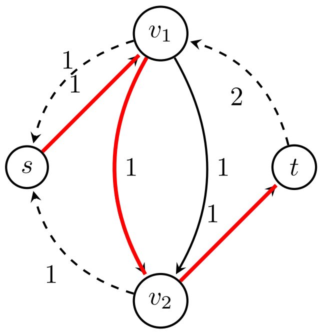
(2) if $f(v, w) > 0$, place an edge with capacity $c_f(v, w) = f(v, w)$ in the opposite direction of (v, w).

Example 3.3. We show a network G with its residual graph G_f, dashed lines are used to emphasize "backward" edges, that is edges created based on the condition (2) of Definition 3.4.



Definition 3.5. Let f be a flow in a network G. An *augmenting path* is a directed path from s to t in the residual network G_f. Alternatively, an augmenting path for f is an undirected path P from s to t such that $f(e) < c(e)$ for all "forward" edges e contained in P, and $f(e) > 0$ for all "backward" edges contained in P.

Example 3.4. An augmenting path is shown below, in the residual graph G_f computed above.

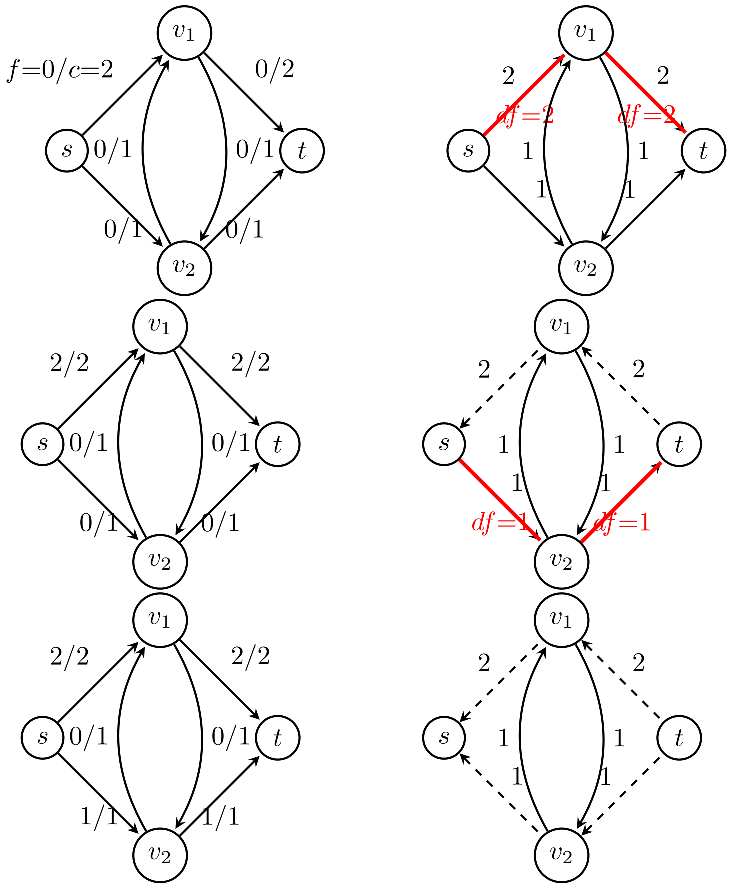


Residual networks and augmenting paths form the core of Ford-Fulkerson algorithm.

Algorithm 4 Ford-Fulkerson (1956)

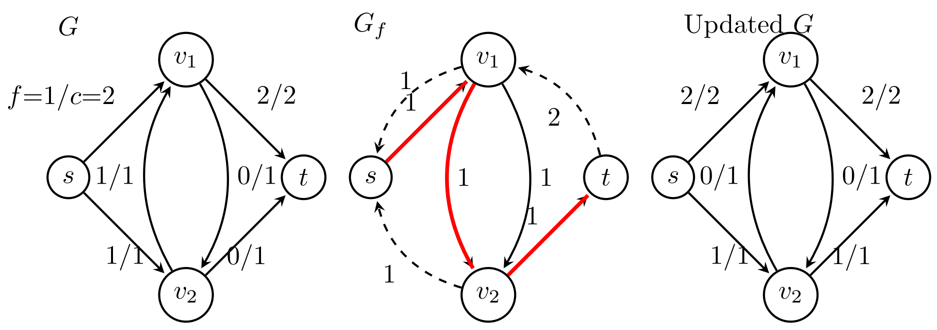
```
    Input: $G = (V, E)$ a network, with capacity $c$, source $s$ and sink $t$.
    Output: a maximal flow $f^*$.
1: Initialize $f(e) = 0 \; \forall \, e \in E$.
2: Compute $G_f$.
3: while (there is a path $P$ from $s$ to $t$ in $G_f$) do
4:     Set $df = \min_{e \in P} c_f(e)$ in $G_f$.
5:     Update $f$ in $G$: set $f(u,v) = f(u,v) + df$ for $(u,v) \in P$ if $(u,v) \in E$
       and $f(v,u) = f(v,u) - df$ for $(u,v) \in P$ if $(v,u) \in E$.
6:     Rebuild the residual network $G_f$.
```

Example 3.5. Ford-Fulkerson algorithm is illustrated, the left-hand side shows the network G while the right hand-side is the corresponding residual graph G_f.



Ford-Fulkerson algorithm returns the maximum flow $|f^*| = 3$ when no more path P is found in the residual graph.

Since the above example does not use any path in G_f which involves a "backward" edge, let us illustrate this case using Examples 3.3 and 3.4. Suppose that one runs the algorithm for some iterations, to get to the step constructed in Example 3.3, in which the augmenting path displayed in Example 3.4 is chosen. The example below show what would be the next updated graph G:



Theorem 3.2. If Ford-Fulkerson algorithm terminates, it outputs a maximum flow.

Proof. Suppose the algorithm terminates and outputs a flow f^*, this means there is no path from s to t in G_{f^*}. In other words, s and t are disconnected. Let S be the set of nodes reachable from s in G_{f^*}; i.e., $v \in S \iff \exists$ a path from s to v. Let $T = V \backslash S$, we claim that $|f^*| = C(S, T)$. Before proving the claim, recall that $|f^*| \leq C(A, B)$ for any cut (A, B), by Lemma 3.1. Thus when equality is reached, which is the case with $|f^*| = C(S, T)$, this means f^* is the maximum flow and $C(S, T)$ the minimum cut. We next prove the claim:

$$|f^*| = C(S, T).$$

Consider any edge e from S to T in the original network G. Edge e must not exist in G_{f^*}, or else its endpoint in T would be reachable from s, contradicting the definition of T. Thus it must be the case that $f^*(e) = c(e)$ for all such e (by construction of G_{f^*}, when $f^*(e) = c(e)$ in G, the edge is removed in its residual graph). Next consider e' from T to S in G. If $f^*(e') > 0$, there will be an edge in the opposite direction of e' in G_{f^*}; i.e., an edge from S to T, again contradicting the definition of T. We conclude that $f^*(e') = 0$ for all such e'. Now recall from (3.1) that

$$|f^*| = \sum_{\substack{u \in S \\ v \in T}} f^*(u, v) - \sum_{\substack{u \in S \\ v \in T}} f^*(v, u)$$

where for our particular cut $f^*(u, v) = c(u, v)$ and $f^*(v, u) = 0$. Thus

$$|f^*| = \sum_{\substack{u \in S \\ v \in T}} c(u, v) = C(S, T).$$

□

Corollary 3.3. (Integral Flow) *If all edge capacities are non-negative integers, then there exists an integral maximum flow.*

Proof. Since edge capacities are integral, the capacity of every edge in G_f is also integral. At each step, *df* is at least one. Thus the value of the flow f increases by at least one. Since $|f| < \infty$, $|f|$ cannot increases indefinitely, hence the algorithm stops after a finite number of steps. □

For an example of network where Ford-Fulkerson algorithm may not terminate, see Exercise 27. The above corollary tells us that this counter-example will use some edge with capacity that is not a non-negative integer.

Corollary 3.4. (Max Flow/Min Cut) *The minimum cut value in a network is the same as the maximum flow value.*

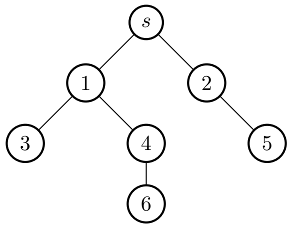
Proof. If Ford-Fulkerson algorithm terminates, as in Corollary 3.3, then we have a proof (we have a flow f^* for which $|f^*| = C(S,T)$, and equality means, as recalled in the proof of Theorem 3.2, that we have both a minimum cut and a maximum flow). Now it turns out that the algorithm always terminates, assuming a particular search order for the the augmenting paths, this is a version of the algorithm called Edmonds-Karp algorithm, which we will see below. □

Using the shortest augmenting path found by breadth-first search, instead of any augmenting path, guarantees that Ford-Fulkerson algorithm terminates.

Algorithm 5 Breadth-First Search (BFS)

```
    Input: a graph $G = (V, E)$, with start vertex $s$.
    Output: a function $d : V \to \mathbb{R}^+$, $d(v)$ is the distance from $v$ to $s$ in $G$.
    Data Structure: a queue $Q$ (first in first out)
1: $Q = \varnothing$ ; $d(s) = 0$ ; $d(v) = \infty \ \forall \ v \neq s$
2: Add $s$ to $Q$;
3: while ($Q \neq \varnothing$) do
4:     Remove first vertex $v$ from $Q$;
5:     for (all $w$ adjacent to $v$) do
6:         if ($d(w) = \infty$) do
7:             $d(w) = d(v) + 1$;
8:             add $w$ to $Q$;
```

Example 3.6. We illustrate the Breadth-First Search (BFS) algorithm.



$Q = \varnothing,\ d(s) = 0, d(1) = \ldots = d(6) = \infty$
$Q = \{s\}$
$Q = \{\}, d(1) = 1, d(2) = 1, Q = \{1, 2\}$
$Q = \{2\}, d(3) = 2, d(4) = 2, Q = \{2, 3, 4\}$
$Q = \{3, 4\}, d(5) = 2, Q = \{3, 4, 5\}$
$Q = \{4, 5\}$
$Q = \{5\}, d(6) = 3, Q = \{5, 6\}$
$Q = \{6\}$
$Q = \{\}$

The breadth first search algorithm is incorporated to Ford-Fulkerson algorithm to find an augmenting path with minimum number of edges.

Algorithm 6 Edmonds-Karp (1970)

```
    Input: $G = (V, E)$ a network, with capacity $c$, source $s$ and sink $t$
    Output: a maximal flow $f^*$.
1: Initialize $f(e) = 0 \; \forall \; e \in E$
2: Compute $G_f$.
3: while (there is a path from $s$ to $t$ in $G_f$) do
4:     Let $P$ be the shortest $s, t-$path in $G_f$ found by BFS.
5:     Set $df = \min_{e \in P} c_f(e)$ in $G_f$ along $P$.
6:     Update $f$ in $G$: set $f(u,v) = f(u,v) + df$ for $(u,v) \in P$ if $(u,v) \in E$
   and $f(v,u) = f(v,u) - df$ for $(u,v) \in P$ if $(v,u) \in E$.
7:     Rebuild the residual network $G_f$.
```

Theorem 3.5. The Edmonds-Karp algorithm terminates after at most $|E|(|V|-1)$ iterations.

Proof. The proof counts the number of while loop iterations before the algorithm stops.

At every iteration of the while loop, the algorithm uses a BFS to find the shortest augmenting path, so it builds a tree that starts at s, then the level 1 of the tree will contain a set V_1 of vertices that are at distance 1 from s, and more generally, the level i of the tree will contain a set V_i of vertices at distance i from s. A shortest path from s to t must use edges (u, v) with $u \in V_i$, $v \in V_{i+1}$ at each step, $i = 1, 2, \ldots$ and a path that uses an edge u, v with $u \in V_i$, $v \in V_j$ and $j \leqslant i$ cannot be a shortest path.

We will look at how the BFS tree starting at s in G_f changes from one iteration to another. Given G_f, we find an augmenting path P, we then update G accordingly:

- If P contains a forward edge e (that is e has the same direction in G_f than in G), then in G, e will get augmented by df, so in the new G_f, the edge e either has disappeared (if $f(e) = c(e)$) or is still there with a lower capacity, and a backward edge may be added to the new G_f if it was not there before.

- If P contains a backward edge e (that is e has the reverse direction in G_f than in G), then in G, e will be diminished by df, so in the new G_f, the edge e either has disappeared, or is still there, and a forward edge may be added to the new G_f if it was not there before.

Thus every new edge that is created in the residual graph from one iteration to another is the reverse of an edge that belongs to P. Since every edge of P goes from level V_i to level V_{i+1} in the BFS tree, every new edge that gets created must go from level V_{i+1} to V_i. This shows that:

- if at a certain iteration of the algorithm, the length of a shortest path from s to t in the residual network is l, then at every subsequent iteration it is $\geq l$.

In words, the length of P from s to t can never decrease as we repeat the while loop.

Furthermore, if the length of the path from s to t in the residual network remains l from one iteration T to the next one $T+1$, then the path P in G_f at iteration $T+1$ must be using only edges which were already present in the residual network at iteration T and which were "forward" edges (from V_i to V_{i+1}). Thus, in all subsequent iterations in which the distance from s to t remains l, it is so because there is a length l path made entirely of edges that were "forward" edges at iteration T. At each iteration however, at least one of those edges is saturated and is absent from G_f in subsequent stages. So there can be at most $|E|$ iterations during which the distance from s to t stays l.

So we further established that:

- after at most $|E|$ iterations, the distance $\geq l+1$.

Since the distance from s to t is at most $|V|-1$, and it takes at most $|E|$ iterations to increase the length of the path by 1, after $|E|(|V|-1)$ iterations, the length of the shortest path becomes the total number of vertices in the network, thus it cannot increase anymore, and the algorithm terminates. □

3.2 Some Applications of Max Flow-Min Cut

Results around the Max Flow-Min Cut of a network were motivated by computing a maximum flow in a network. However they can also be used to prove other results in graph theory. We give as example below Hall's Theorem. Menger's Theorem is given in Exercise 28.

Definition 3.6. Given a graph $G=(V,E)$, a *perfect matching* is a subset of E which covers all vertices but each vertex only once.

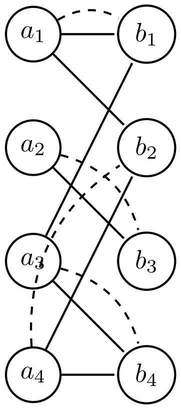
Theorem 3.6. ***[Hall's theorem]*** *Let $G=(V,E)$ be a bipartite graph with $V=A\sqcup B$ and $|A|=|B|$. For $J\subseteq A$, let $\Gamma(J)$ be the set of vertices adjacent to some vertex in J:*

$$\Gamma(J)=\{v\in B,\exists\ w\in J,\{v,w\}\in E\}.$$

Then G has a perfect matching if and only if

$$|\Gamma(J)|\geqslant|J|\ \forall\ J\subseteq A.$$

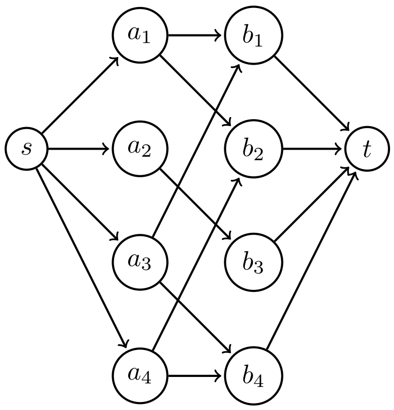
Example 3.7. In the bipartite graph below, with $A = \{a_1, a_2, a_3, a_4\}$, $B = \{b_1, b_{2,3}, b_4\}$, a perfect matching is shown using dashed lines.



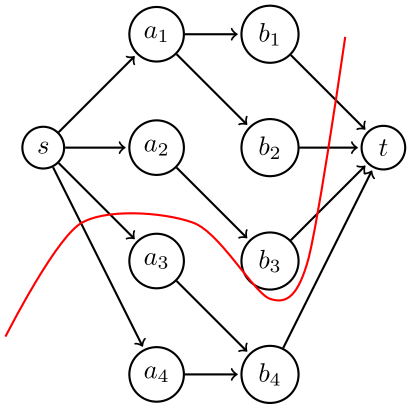
We can illustrate the condition $|\Gamma(J)| \geq |J| \forall J \subset A$. For $|J| = 1$, that is J is $\{a_i\}$ for some i, each vertex in A has at least 1 adjacent vertex in B. For $|J| = 2$, that is $J \in \{\{a_1, a_2\}, \{a_1, a_3\}, \{a_1, a_4\}, \{a_2, a_3\}, \{a_2, a_4\}, \{a_3, a_4\}\}$, then $|\Gamma(J)| = 3$. The same computations can be checked for $|J| = 3, 4$.

Proof. ($\Longrightarrow$) Suppose we have a perfect matching. The edges of a perfect matching provide a unique neighbour for each node in J. There maybe other edges and thus other adjacent nodes in B, but there are at least those provided by the matching, which makes sure that $|J| \leqslant |\Gamma(J)|$.
($\Longleftarrow$) Suppose by contradiction that G has no perfect matching but $|J| \leqslant |\Gamma(J)|$. We frame the matching problem in terms of network flow by turning G into a network N as follows:

- Direct all edges in G from A to B and give them capacity ∞.
- Add a node s and an edge (s, a) $\forall$ $a \in A$ with capacity 1.
- Add a node t and an edge (b, t) $\forall$ $b \in B$ with capacity 1.



Now G has a perfect matching $\iff$ the max flow in N is n $\iff$ the min cut in N is n. Our contradiction assumption thus implies that the min-cut is $< n$. Consider a min-cut (S, T) and let $J = S \cap A$. Then all of the edges from s to $A \backslash J$ cross the cut. These edges have total capacity $|A \backslash J|$ (since each edge has capacity 1).



The edges $(a_3, b_1), (a_4, b_2)$ were removed from the previous example to remove the perfect matching

Now all neighbours of J in G must also lie in S, that is $\Gamma(J) \subset S$, or else an edge of capacity ∞ would cross the cut. Then all of the edges from the nodes $\Gamma(J)$ to t cross the cut. These edges have total capacity $|\Gamma(J)|$. We then have:

$$\begin{cases} |J| + |A \backslash J| = |A| = n \\ |A \backslash J| + |\Gamma(J)| \leqslant C(S, T) < n \end{cases}$$

$$\implies n - |J| + |\Gamma(J)| < n \iff |\Gamma(J)| < |J|$$

which is a contradiction. □

3.3 The Min Cost Flow Problem and some Applications

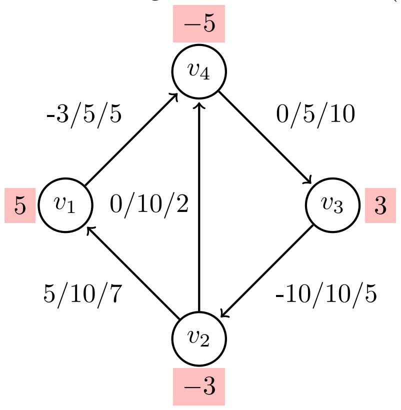
Definition 3.7. Let $G = (V, E)$ be a directed graph, and let

$b : E \to \mathbb{R}$	(lower capacity)
$c : E \to \mathbb{R}$	(upper capacity, or just capacity)
$\gamma : E \to \mathbb{R}$	(cost function)
$d : V \to \mathbb{R}$	(demand function)

be functions (note that they take value in the whole of $\mathbb{R}$). Furthermore, the demand function is such that $\sum_{v \in V} d(v) = 0$. Then G together with the functions b, c, γ, d is called a *minimum cost flow network*.

When $d(v) > 0$, we refer to v as a *supply* node, when $d(v) < 0$, we refer to v as a *demand* node, and when $d(v) = 0$, we say that v is a transit node.

Example 3.8. Here is an example of minimum cost flow network. The demands of the nodes are written next to the nodes. We notice that $\sum_{v\in V} d(v) = 0$. The labels on the edges are of the form $b(e)/c(e)/\gamma(e)$.



Definition 3.8. A *flow* in such a network is a map $f : E \to \mathbb{R}$ with

1. $b(e) \leq f(e) \leq c(e)$ for all $e \in E$,
2. $d(v) = \sum_{u\in O(v)} f(v,u) - \sum_{u\in I(v)} f(u,v)$ for all $v \in V$[1].

The *cost* of such a flow is $\gamma(f) = \sum_{e\in E} \gamma(e)f(e)$.

Problem 5. The *min-cost-flow problem* consists of finding a flow in a minimum cost flow network of minimum cost.

We will show first that the maximum flow problem is a special case of the min-cost-flow problem. Indeed, let $G = (V,E)$ be a network with source s and sink t, and edge capacities given by a function c. Construct the min-cost flow network $G' = (V,E')$ where $E' = E \cup \{(t,s)\}$. We specify the 4 functions of a min-cost flow network G':

- $b'(e) = 0$ for all $e \in E'$,
- $c'(e) = c(e)$ for all $e \in E$ and $c'(t,s) = \sum_{u\in O(s)} c(s,u)$,
- $\gamma'(e) = 0$ for all $e \in E$ and $\gamma'(t,s) = -1$,
- $d'(v) = 0$ for all $v \in V$.

The functions are clearly defined on $G = (V,E)$: the lower bound is 0 for every edge in E in a max flow problem, so is the demand at every vertex but for the source and the sink. Then each edge is assigned its capacity in G. The only things to discuss are the addition of (t,s), the choice of capacity and cost for this edge, as well as the choice of demand for s and t.

[1]You may find $d(v) = -\sum_{u\in O(v)} f(v,u) + \sum_{u\in I(v)} f(u,v)$, both definitions are found.

Intuitively, if we want to maximize the flow between s and t using a min-cost formulation, this means that decreasing the cost must increase the flow. By introducing a new edge (t, s) with cost $\gamma'(t, s) = -1$, we push the flow to use this new edge as much as possible, since going through it decreases the cost (all the other costs are 0: $\gamma'(e) = 0$ for all $e \in E$). But now, the demand at the source s is 0, so the edge (t, s), which is the only incoming edge of s, must carry as much flow as the source can output, and similarly at the sink t whose demand is also 0, the edge (t, s) being its only outgoing edge, it must take in as much flow as possible.

Lemma 3.7. *With G and G' as defined above, let f be a flow in G and f' be defined by*

$$f'(e) = f(e)\ \forall e \in E,\ f'(t, s) = \sum_{u \in O(s)} f(s, u).$$

Then f is a maximum flow in $G \iff f'$ is a minimum cost flow in G'.

Proof. First we check that f' is indeed a flow in G'.

- We have that f is a flow in G, thus $0 \le f(e) \le c(e)$ for all $e \in E$, and f' satisfies $0 \le f'(e) = f(e) \le c(e) = c'(e)$ for all $e \in E$. Then $f'(t, s) = \sum_{u \in O(s)} f(s, u) \le c'(t, s) = \sum_{u \in O(s)} c(s, u)$ so f' is feasible.
- Then for every vertex v in V which is neither the source nor the sink, by flow conservation of f, and since $f(e) = f'(e)$ for all $e \in E$:

  $$\sum_{u \in O(v)} f'(v, u) - \sum_{u \in I(v)} f'(u, v) = 0 = d(s).$$

  For s, since (t, s) is its only incoming edge, and by definition of $f'(t, s)$:

  $$\sum_{u \in O(s)} f(s, u) - f'(t, s) = 0 = d(v).$$

  For t, since (t, s) is its only outgoing edge, by definition of $f'(t, s)$ and recalling that the flow that goes out of the source must reach the sink (see Exercise 25) :

  $$f'(t, s) - \sum_{u \in I(t)} f(u, t) = f'(t, s) - \sum_{u \in O(s)} f(s, u) = 0 = d(t).$$

Now the cost of the flow f' is

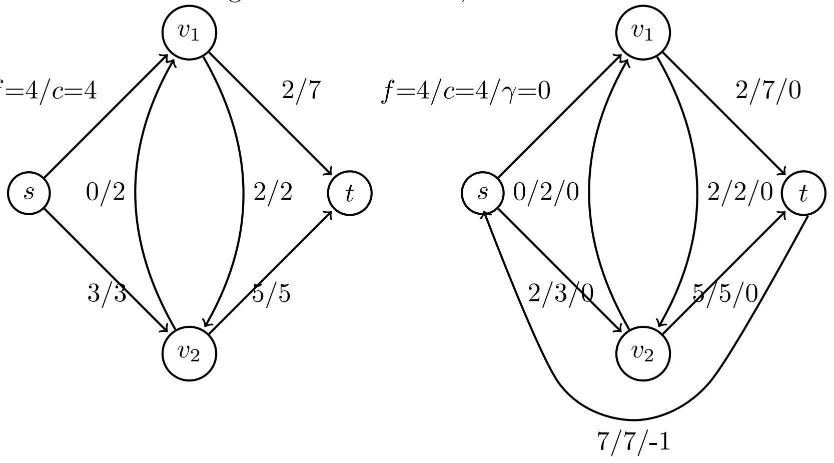
$$\gamma(f') = \sum_{e \in E'} \gamma(e) f'(e) = \gamma(t, s) f'(t, s) = -f'(t, s),$$

that is

$$\gamma(f') = -\sum_{u \in O(s)} f(s, u)$$

and minimizing $\gamma(f')$ maximizes $\sum_{u \in O(s)} f(s, u)$. □

Example 3.9. On the left hand-side, a graph G is shown with a maximal flow. On the right hand-side, its corresponding min-cost flow network is shown. The capacity of (t, s) is $\sum_{u \in O(s)} c(s, u)$. To minimize the cost, the edge (t, s) should be used as much as possible, constrained by its capacity which is the maximal amount of flow that can go out of the source, and come in the sink.



Next we will see that finding a shortest path is a special case of the min-cost flow too.

Let $G = (V, E)$ be a directed weighted graph with weight function $w : E \to \mathbb{R}_{>0}$, and consider two vertices s, t in V. A path $P \subseteq E$ from s to t with minimum weight $\sum_{e \in P} w(e)$ is called a *shortest path* from s to t.

We define a corresponding min-cost flow network $G' = (V, E)$ by using the same vertices and edges as that of G, and by adding the following functions:

- $b(e) = 0$ for all $e \in E$,
- $c(e) = 1$ for all $e \in E$,
- $\gamma(e) = w(e)$ for all $e \in E$,
- $d(v) = 0$ for all $v \in V \backslash \{s, t\}$, $d(s) = 1$, $d(t) = -1$.

Lemma 3.8. *Let f be a minimum cost flow in G' as defined above, such that all flow values are integral (in fact they are only 0 or 1). Then the edges with $f(e) = 1$ form a shortest path from s to t in G.*

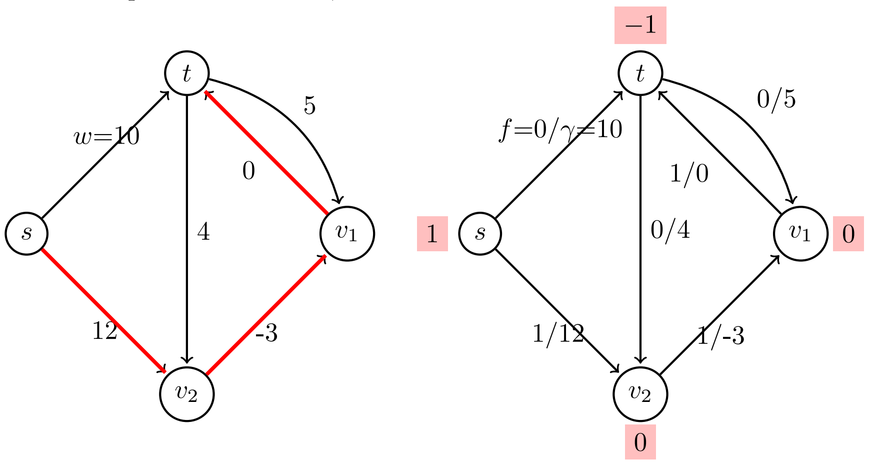
Proof. We first note that all nodes have a demand of 0, but for the start s and end t of the path, whose demands are $d(s) = 1$ and $d(t) = -1$. Thus there is a flow of 1 that has to go from s to t. Then the capacity of every edge is 1, which means that every edge is used either once or none. Also since the flow is integral, every visited vertex has at most one incoming edge, and one outgoing edge (if the flow was not integral, you could have 0.5 going of a node on two different edges). Now to minimize the cost

$$\sum_{e \in E} \gamma(e) f(e) = \sum_{e \in E} w(e) f(e),$$

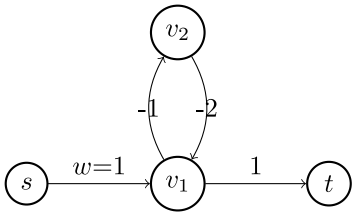
this flow of 1 will use edges with the lowest cost (the flow will be either 1 if the edge is used, or 0 else), thus finding the shortest path. Note that the output is really a path in that none of the vertices are repeated. This is because repeating a vertex means there is a cycle, and since the weights are positive, a cycle would increase the cost. □

In the above proof, we use the fact that the weights are positive. If we have negative edges, it is possible to find a shortest path as a min-cost flow, assuming that there is no cycle whose sum of weights is negative or zero. In that case, the same argument holds: adding a cycle would add a larger cost and thus the algorithm will avoid the cycle.

Example 3.10. This example shows that even with negative weights (this graph has no cycle whose sum of costs is negative), the shortest path problem can be solved using the min-cost flow problem. On the left-hand side, a graph with a shortest path from s to t is given. On the right hand-side, the equivalent min-cost flow problem is shown, whose cost is 9.



Example 3.11. The network below contains the cycle v_1, v_2, v_1 which has weight -3. The shortest path is given by s, v_1, t. Indeed, using the cycle v_1, v_2, v_1 means that v_1 is used twice, so we do not get a path. However if we were asked for a flow that minimizes the weights, we would enter the cycle, this would not create a path since v_1 would be repeated, and the iterations may not even terminate (if we do not specify a capacity).



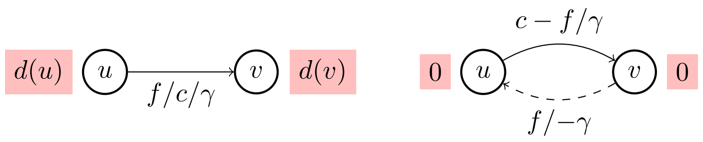
3.4 The Cycle Cancelling Algorithm

We next discuss an algorithm that actually solves the min-cost flow network problem. Similarly to Ford-Fulkerson algorithm, it relies on the notion of residual graph, only we need to define residual graph in our new context: we know how to define a residual graph that takes into account the flow and the capacity of the edges (see Definition 3.4), but we need to add what happens to the cost. The differences between the two definitions are highlighted.

Definition 3.9. Given a flow f on a graph (min-cost flow network) $G = (V, E)$, its *residual graph* G_f has V for vertices whose demands are set to 0, and

(1) edges have capacities $c(v, w) - f(v, w)$, only edges with $c_f := c(v, w) - f(v, w) > 0$ are shown, they have a cost of $\gamma_f(v, w) = \gamma(v, w)$, and they are shown in the same direction as (v, w),

(2) if $f(v, w) > 0$, place an edge with capacity $c_f(v, w) = f(v, w)$ in the opposite direction of (v, w), with a cost $\gamma_f(v, w) = -\gamma(v, w)$.



While augmenting paths were the key ingredient to compute max-flows, for min-cost flows, we look at negative cost cycles.

Definition 3.10. In a minimum cost flow network, a *negative cost cycle* is a cycle C whose edges have a cost γ such that $\sum_{e \in C} \gamma(e) < 0$.

We first give the algorithm, we will prove next why it works. For that, we will make the following assumptions:

- The lower capacity b is zero (and not written anymore): it is always possible to replace a min-cost-flow network where b is arbitrary by an equivalent min-cost-flow network where $b(e) = 0$ for all $e \in E$ (see Exercise 31).
- All upper capacities $c(e)$ are finite: if $c(e)$ is infinite for some edge e (which we can useful to describe a problem formulation), one can prove that there exists an optimal flow which is upper bounded by a quantity that only depends on the demands and the finite capacities of the graph, and this upper bound can be used as a finite capacity to replace $c(e)$.
- All costs are non-negative: it can be shown that it is possible to replace a min-cost flow network where γ can be negative (e.g. in G' in Lemma 3.7) by an equivalent min-cost-flow network where $\gamma(e) \geq 0$ for all e.
- All upper capacities c, costs γ and demands d are integral: arguments saying demands and/or flows are decreasing at each iteration and thus will reach 0 (as used later) do not hold if we deal with real values.

- Networks have no directed cycle of negative cost and infinite capacity: this scenario is ill-defined (see Example 3.11), no matter how one can transform such a network.

Algorithm 7 Cycle Cancelling

```
Input: $G = (V, E)$ a network, with capacity $c : E \to \mathbb{Z}_{\geq 0}$, $|c| < \infty$, demand $d : V \to \mathbb{Z}$, and cost $\gamma : E \to \mathbb{Z}_{\geq 0}$
Output: a minimum cost flow $f^*$, or $\varnothing$.
1: $f \leftarrow$ FeasibleFlow$(G, d, c)$
2: if $(f = \varnothing)$ do
3:     return $\varnothing$.
4: Compute the residual graph $G_f$ with cost $\gamma_f$ and capacity $c_f$.
5: Find a negative cost cycle $W$ in $G_f$, set $W = \varnothing$ if none exists.
6: while $(W \neq \varnothing)$ do
7:     Set $df = \min_{e \in W} c_f(e)$ in $G_f$ along $W$.
8:     Update $f$ in $G$: set $f(u, v) = f(u, v) + df$ for $(u, v) \in W$ if $(u, v) \in E$ and $f(v, u) = f(v, u) - df$ for $(u, v) \in W$ if $(v, u) \in E$.
9:     Rebuild the residual network $G_f$, with cost $\gamma_f$ and capacity $c_f$.
10:    Find a negative cost cycle $W$ in $G_f$, set $W = \varnothing$ if none exists.
```

The idea is to start with a feasible flow in G (a flow that satisfies Definition 3.8)[2], then try to find a negative cost cycle in G_f. When no such a cycle exists, the claim is that we have an optimal flow. Otherwise, we augment the flow along the cycle, decrease the cost of the flow, and repeat. To find a negative cycle, one may use Floyd-Warshall algorithm (see Algorithm 9).

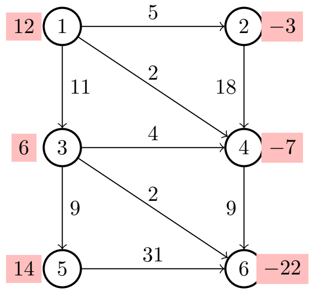
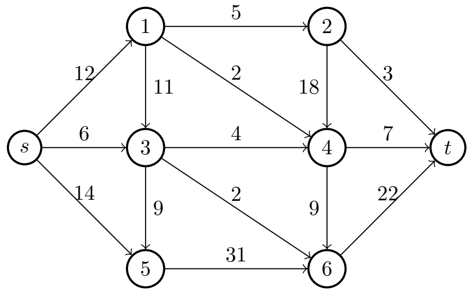
We still need to answer two questions: how do we find a feasible flow, and why does the cycle cancelling algorithm work.

Algorithm 8 FeasibleFlow

```
Input: $G = (V, E)$ a network, with capacity $c : E \to \mathbb{Z}_{\geq 0}$, $|c| < \infty$, demand $d : V \to \mathbb{Z}$, and cost $\gamma : E \to \mathbb{Z}_{\geq 0}$
Output: a feasible flow $f$, or $\varnothing$.
1: $V' \leftarrow V \cup \{s, t\}$.
2: $E' \leftarrow E \cup \{(s, v),\ d(v) > 0\}$ and set $c'(s, v) = d(v)$.
3: $E' \leftarrow E' \cup \{(v, t),\ d(v) < 0\}$ and set $c'(v, t) = -d(v)$.
4: Create a graph $G' = (V', E')$, with capacity $c'$ and $c'(e) = c(e)$ for all $e \in E$.
5: Find $f^*$ that solves the max flow problem from $s$ to $t$ in $G'$.
6: if ($f^*$ saturates the source edges) do
7:     Return $f^*|_E$.
8: else:
9:     Return $\varnothing$.
```

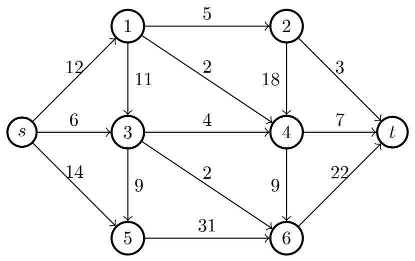
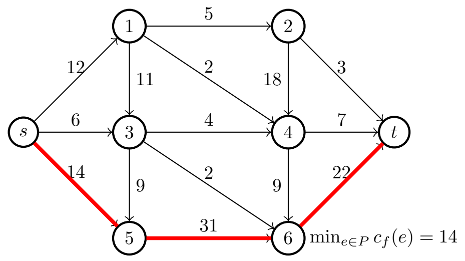
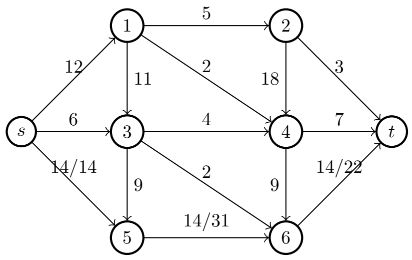
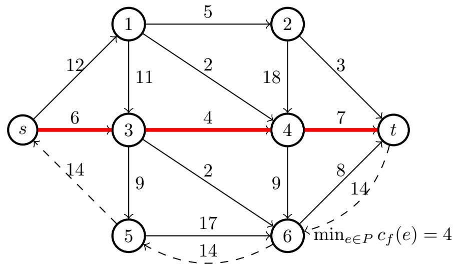
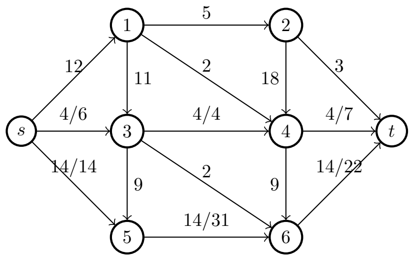
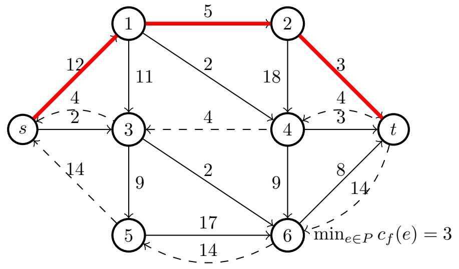
[2]For max flow problems, we use "feasible" for the condition $0 \leq f(e) \leq c(e)$, for min cost flow problems, we use "feasible" for both properties.

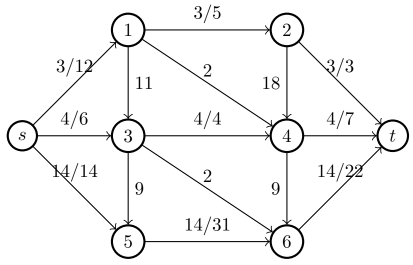
Example 3.12. Let us try to find a feasible flow for the network on the left below.

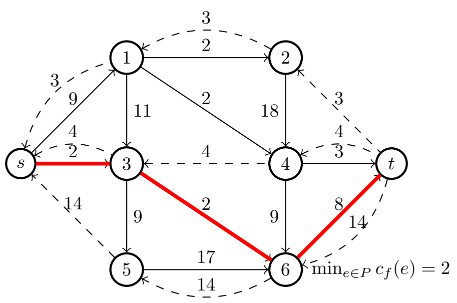



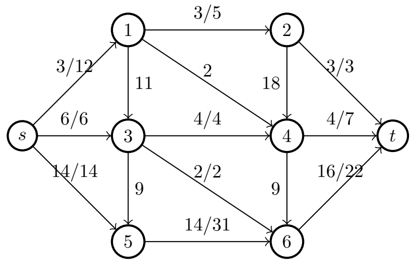
To start with, we create a graph $G' = (V', E')$ where $V' = V \cup \{s,t\}$, $E' = E \cup \{(s,v),\ d(v) > 0\} \cup \{(v,t),\ d(v) < 0\}$ and $V' = V \cup \{s,t\}$. Then the capacities are $c'(e) = c(e)$ for all $e \in E$, $c'(s,v) = d(v)$ and set $c'(v,t) = -d(v)$.

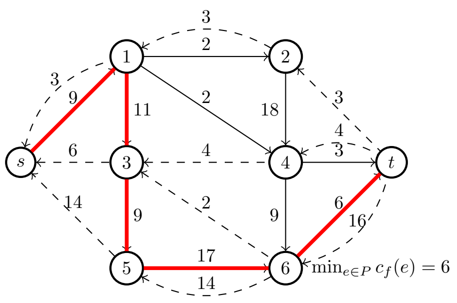
Then to solve the max-flow problem in the network G', we apply Ford-Fulkerson algorithm. We show the graph and its updates on the left, and the corresponding residual graphs with paths from s to t on the right.

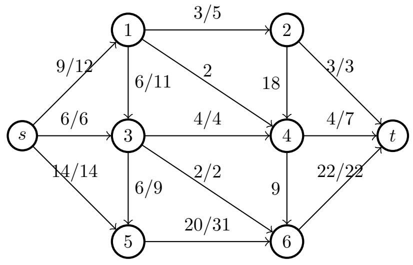


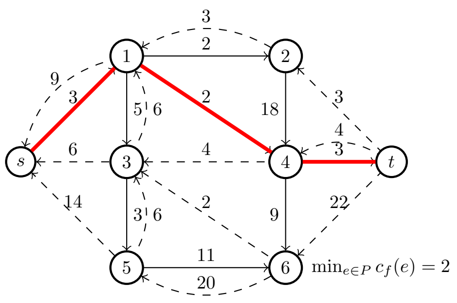


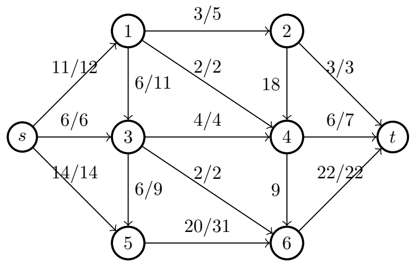


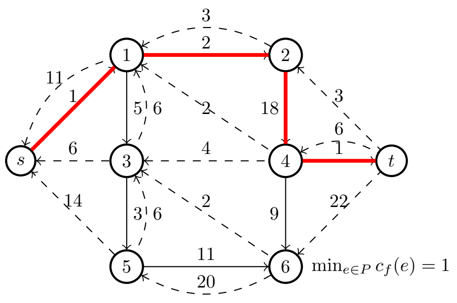


At this point, we observe that s has only one outgoing edge which is not saturated, namely the one from s to 1.

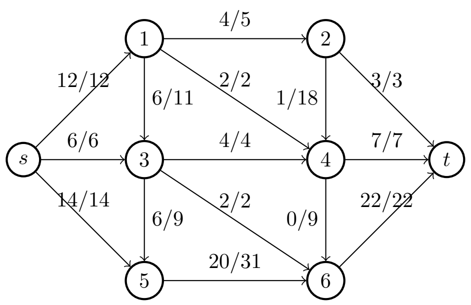




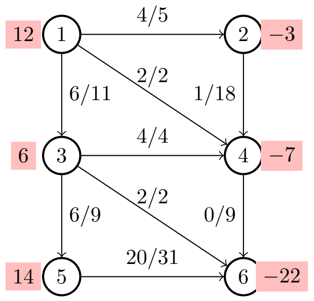




This gives us a last update:



We see that the flow obtained saturates the outgoing edges of s. We now remove s and t and the edges connected them. We can check that the flow obtained is feasible.



Let us prove that the FeasibleFlow algorithm actually works.

Lemma 3.9. *The original network G has a feasible flow if and only if every maximal flow of G' saturates all the source edges. Furthermore, if f^* is a maximal flow of G', then the restriction $f^*|_E$ to the edges of G is a feasible flow.*

Proof. Recall that a feasible flow for the min-cost-flow problem must satisfy (1) $b(e) \leq f(e) \leq c(e)$, and (2) $\sum_u f(v,u) - \sum_u f(u,v) = d(v)$. We may assume $b(e) = 0$ for all e.

($\Leftarrow$) Consider an arbitrary maximum flow f^* of G' which saturates all source edges. We will show that G has a feasible flow, given by $f^*|_E$ (this also proves the second part of the lemma statements). Clearly $f^*|_E$ satisfies (1) by definition of flow in G'. Then consider vertices v such that $d(v) > 0$ in the original network G. They are connected to s in G' with capacity $d(v)$, so since every such an edge saturates, this means that the flow on (s,v) is $d(v)$, thus $f(s,v)+\sum_{u\neq s} f(u,v) = \sum_u f(v,u)$ by definition of flow in G', and $f^*|_E$ satisfies (2) for these vertices. Next consider vertices such that $d(v) < 0$. Since f saturates the edges out of the source, by definition of demand, it also saturates the edges entering the sink. Then $\sum_u f(u,v) = \sum_{u\neq t} f(v,u) + f(v,t)$ by definition of flow in G', that is $\sum_{u\neq t} f(v,u) - \sum_u f(u,v) = -f(v,t) = d(v)$ and $f^*|_E$ satisfies (2) for these vertices.

($\Rightarrow$) Suppose we have a feasible flow for G, and a maximal flow for G'. When the manipulation that transforms the original network into a max flow problem is done, the demands have been put as edge capacities between the source and nodes of positive demand (and between nodes of negative demands and the sink with a change of sign), so a maximal flow can at most saturate the source edges, and it will do so since the flow is feasible in G. $\square$

This lemma also tells us that if we cannot find a feasible flow, then we cannot solve the min-cost-flow problem.

Next, we prove a first result that shows why it is interesting to work with a residual graph. For an edge $e = (u, v) \in E$, we write $\tilde{e} = (v, u)$, that is $\tilde{e}$ is the backward edge obtained by changing the direction of e.

Theorem 3.10. *Let f^o be a feasible flow of a network G, and let G_{f^o} be its residual graph. Then a flow f is feasible in G if and only if the flow f' defined by*

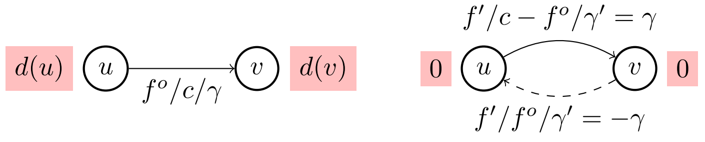
$$\begin{cases} f'(e) = f(e) - f^o(e),\ f'(\tilde{e}) = 0 & \text{if } f(e) \geq f^o(e) \\ f'(\tilde{e}) = -f(e) + f^o(e),\ f'(e) = 0 & \text{if } f(e) < f^o(e) \end{cases}$$

is a feasible flow in G_{f^o}. Furthermore

$$\sum_{e \in E} \gamma(e) f(e) = \sum_{e \in E(G_{f^o})} \gamma'(e) f'(e) + \sum_{e \in E} \gamma(e) f^o(e)$$

where γ is the cost in G and γ' is the cost in G_{f^o}.

This theorem is looking at one update of G with a feasible flow f^o, from which a residual graph G_{f^o} is built. It tells when a feasible flow f' in G_{f^o} will correspond to a feasible flow f in G, and how the cost of the flow f' in G_{f^o} will update the cost of the flow f in G with respect to the cost given by the flow f^o.



Proof. ($\Rightarrow$) Assume that f is a feasible flow, that is (1) $0 \leq f(e) \leq c(e)$ and (2) $\sum_u f(v, u) - \sum_u f(u, v) = d(v)$. We need to check that f' is feasible. We start with (1). Clearly $f'(\tilde{e}) = 0$ and $f'(e) = 0$ respectively satisfy (1).

- If $f(e) \geq f^o(e)$, then $f'(e) = f(e) - f^o(e) \geq 0$ and $f'(e) = f(e) - f^o(e) \leq c(e) - f^o(e) = c_{f^o}(e)$ since f is feasible and e is forward.
- If $f(e) < f^o(e)$, then $f'(\tilde{e}) = -f(e) + f^o(e) > 0$ and $f'(\tilde{e}) \leq f^o(e)$, the residual capacity of the backward edge $\tilde{e}$. Since f is feasible, $f(e) \geq 0$ and $f^o(e) > 0$, thus the backward edge $\tilde{e}$ appears in the residual graph.

Then we check (2) for f' in G_{f^o}, recalling that $d(v) = 0$ in G_{f^o}. For every v, we have, recalling that an edge e in G_{f^o} is either a forward edge in G, or a backward version of an edge in G:

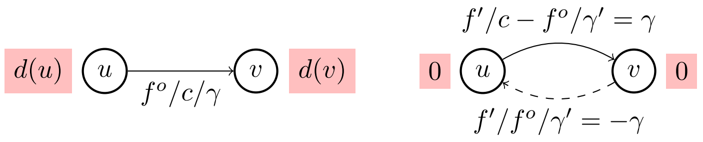
$$\begin{aligned}
&\sum_{u,(v,u)\in E(G_{f^o})} f'(v,u) - \sum_{u,(u,v)\in E(G_{f^o})} f'(u,v) &(3.2)\\
=&\sum_{u,(v,u)\in E(G)} f'(v,u) + \sum_{u,(u,v)\in E(G)} f'(v,u) - \sum_{u,(u,v)\in E(G)} f'(u,v) - \sum_{u,(v,u)\in E(G)} f'(u,v)\\
=&\sum_{u,(v,u)\in E(G)} (f'(v,u)-f'(u,v)) - \sum_{u,(u,v)\in E(G)} (f'(u,v)-f'(v,u)) &(3.3)\\
=&\sum_{u,(v,u)\in E(G)} (f(v,u)-f^o(v,u)) - \sum_{u,(u,v)\in E(G)} (f(u,v)-f^o(u,v))
\end{aligned}$$

because in the first sum, either $f'(v,u) = f(v,u) - f^o(v,u)$ and $f'(u,v) = 0$, or $f'(v,u) = 0$, and $-f'(u,v) = f(v,u) - f^o(v,u)$. Similarly, in the second sum, either $f'(u,v) = f(u,v) - f^o(u,v)$ and $f'(v,u) = 0$, or $f'(u,v) = 0$, and $-f'(v,u) = f(u,v) - f^o(u,v)$. But now, since f is feasible

$$\sum_{u,(v,u)\in E(G)} f(v,u) - \sum_{u,(u,v)\in E(G)} f(u,v) = d(v).$$

Since f^o is also feasible, (2) is proven by noting that

$$-\sum_{u,(v,u)\in E(G)} f^o(v,u) + \sum_{u,(u,v)\in E(G)} f^o(u,v) = -d(v).$$



($\Leftarrow$) For the converse, assume that f' is a feasible flow.

- If $f'(\tilde{e}) = 0$, then $f(e) = f'(e) + f^o(e)$, and $f(e) \geq 0$. Also since f' is a feasible flow in G_{f^o}, $f(e) \leq c_{f^o}(e) + f^o(e) = (c(e) - f^o(e)) + f^o(e) = c(e)$.
- If $f'(e) = 0$, then $f(e) = f^o(e) - f'(\tilde{e})$, and since f' is feasible in G_{f^o}, it must be less than the residual capacity which for a backward edge is $f^o(e)$ and $f(e) \geq 0$. Also, $f(e) \leq f^o(e) \leq c(e)$ since f^o is a feasible in G.

We then need to check that $\sum_u f(v,u) - \sum_u f(u,v) = d(v)$. We write $f(e) = f'(e) - f'(\tilde{e}) + f^o(e)$ since either $f'(e) = 0$ or $f'(\tilde{e}) = 0$, so that

$$\begin{aligned}
&\sum_{u,(v,u)\in E(G)} f(v,u) - \sum_{u,(u,v)\in E(G)} f(u,v)\\
=&\sum_{u,(v,u)\in E(G)} (f'(v,u) - f'(u,v) + f^o(v,u)) - \sum_{u,(u,v)\in E(G)} (f'(u,v) - f'(v,u) + f^o(u,v))
\end{aligned}$$

Since f^o is feasible:

$$\sum_{u,(v,u)\in E(G)} f^o(v,u) - \sum_{u,(u,v)\in E(G)} f^o(u,v) = d(v).$$

This proves what we wanted since

$$\sum_{u,(v,u)\in E(G)} (f'(v,u) - f'(u,v)) - \sum_{u,(u,v)\in E(G)} (f'(u,v) - f'(v,u)) = 0$$

using (3.3), which is equal to (3.2), which is 0 once we know f' is feasible.

Finally, we compute the cost of the flow f as a function of the cost of the flows f' and f^o. The cost γ' of f' is computed in G_{f^o}, and $\gamma' = \pm\gamma$ depending on whether the edge is forward or backward. Thus

$$\gamma'(e)f'(e) + \gamma'(\tilde{e})f'(\tilde{e}) = \gamma(e)(f'(e) - f'(\tilde{e})) = \gamma(e)(f(e) - f^o(e))$$

since $f(e) = f'(e) - f'(\tilde{e}) + f^o(e)$. Thus $\gamma(e)f(e) = \gamma'(e)f'(e) + \gamma'(\tilde{e})f'(\tilde{e}) + \gamma(e)f^o(e)$. We conclude the proof by summing over the edges $e \in E$.

$$\begin{aligned}\sum_{e\in E}\gamma(e)f(e) &= \sum_{e\in E}\gamma'(e)f'(e) + \gamma'(\tilde{e})f'(\tilde{e}) + \gamma(e)f^o(e)\\ &= \sum_{e\in E}\gamma'(e)(f'(e) - f'(\tilde{e})) + \sum_{e\in E}\gamma(e)f^o(e)\end{aligned}$$

and $\sum_{e\in E}\gamma'(e)(f'(e) - f'(\tilde{e})) = \sum_{e\in E(G_{f^o})}\gamma'(e)f'(e)$ since edges in G_{f^o} come as both forward and backward edges of G. □

The above result states how the cost of a flow changes in G based on the cost of a flow in its corresponding residual graph. The next result describes the role of cycles in a flow of G.

Theorem 3.11. [Flow Decomposition Theorem.] *Consider a graph $G = (V, E)$ with $|V| = n$ vertices and $|E| = m$ edges. Every flow f can be decomposed into cycles and paths, such that:*

1. *Every directed path with nonzero flow connects a supply node (with positive demand) to a demand node (with negative demand).*
2. *At most $n + m$ paths and cycles have nonzero flow, and out of these, at most m cycles have nonzero flow.*

Proof. Suppose v_0 is a supply vertex, that is $d(v_0) > 0$. Then there is some edge $e = (v_0, v_1)$ with nonzero flow (otherwise the flow is not feasible). If v_1 is a demand vertex, then we stop, we found a path $P = (v_0, v_1)$. Otherwise, v_1 is either another supply vertex, or transit vertex, and the property $\sum_u f(v_1, u) - \sum_u f(u, v_1) = d(v_1)$ ensures that there is another edge (v_1, v_2) with nonzero flow. We repeat this argument until either a demand node is reached, or we encounter a previously visited node. Surely, one of these two cases must occur within n steps, since n is the number of vertices. Thus either we obtain a directed path P from a supply node to a demand node, or we obtain a directed cycle W. In both cases, the path or the cycle only consists of nonzero flow edges.

If we obtain a directed path P from v_0 to v_k, set

$$\delta(P) = \min\{d(v_0), -d(v_k), \min_{e\in P} f(e)\}$$

and update $d(v_0) \leftarrow d(v_0)-\delta(P)$, $d(v_k) \leftarrow d(v_k)+\delta(P)$, and $f(e) \leftarrow f(e)-\delta(P)$, for every $e \in P$.

If instead we obtain a directed cycle W, then set

$$\delta(W) = \min_{e\in W} f(e)$$

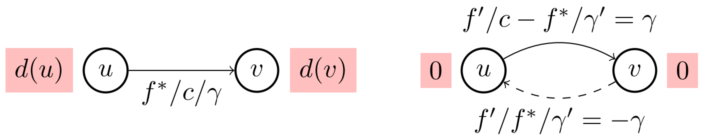
and update $f(e) \leftarrow f(e) - \delta(W)$, for every $e \in W$.

We repeat the above process with the newly obtained problem, until there are no more supply nodes, i.e., $d(v) = 0$ for all vertices v (if supply nodes are gone, by the definition of demand, the demand nodes must be gone too). This will happen because whenever a path is found, the supply demands are decreased, so they will eventually reach zero. Then we select any node with one outgoing edge with nonzero flow as a starting point, and repeat the procedure. Since $d(v) = 0$, we will find a directed cycle. The process stops when $f = 0$.

The original flow is the sum of flows on the paths and cycles we found. Then each time we find a directed path, the update $d(v_0) \leftarrow d(v_0) - \delta(P)$, $d(v_k) \leftarrow d(v_k)+\delta(P)$, and $f(e) \leftarrow f(e)-\delta(P)$, will send either a supply node, a demand node, or an edge to 0. Similarly, each time we find a directed path, the flow on some edge will be updated to 0. Therefore the process terminates after identifying at most $n+m$ cycles and paths, and identify at most m cycles. □

We finally have the result that justifies the cycle cancelling algorithm.

Theorem 3.12. *A feasible flow f^* is an optimal solution of the minimum cost flow problem if and only if G_{f^*} contains no negative cost directed cycle.*



Proof.
(⇒) Assume that f^* is an optimal flow, but that G_{f^*} contains a negative cost directed cycle W of cost $\gamma'(W) = \sum_{e\in W}\gamma'(e) < 0$. We will get a contradiction to the optimality of f^*, intuitively because we can augment the flow f' along W, which will decrease its cost value.

Formally, consider the flow f' along W in G_{f^*} given by: $f'(e) = 0$ for $e \notin W$, and for $e \in W$, use the maximal flow obtained by setting $f'(e)$ for all $e \in W$ to be the minimum residual capacity of the cycle W (this flow is surely feasible). Then by Theorem 3.10, we get a feasible flow f in G such that:

$$\sum_{e\in E}\gamma(e)f(e) - \sum_{e\in E}\gamma(e)f^*(e) = \sum_{e\in E(G_{f^*})}\gamma'(e)f'(e) = \sum_{e\in W}\gamma'(e)f'(e) < 0$$

since $f'(e) = 0$ for $e \notin W$. Also $f'(e)$ is constant, it was set to be the minimum residual capacity for every $e \in W$, thus it can be taken out of the sum, and we can then use that the cycle has a negative cost: $\gamma'(W) < 0$. Then

$$\sum_{e \in E} \gamma(e) f(e) < \sum_{e \in E} \gamma(e) f^*(e),$$

a contradiction to the optimality of f^*.

($\Leftarrow$) Now assume that f^* is a feasible flow and that G_{f^*} contains no negative cost directed cycle. We want to show that f^* is optimal. Let f^o be any feasible flow. We will show that f^* has a lesser cost than f^o.

By Theorem 3.10, with f^* a feasible flow with residual graph G_{f^*}, since f^o is also feasible in G, we can find a feasible flow f' in G_{f^*}. Now applying Theorem 3.11 to f' in G_{f^*}, we decompose f' into paths and cycles. But in G_{f^*}, all nodes have demands 0, thus the flow f' is only composed of cycles. But since G_{f^*} contains no negative cost directed cycle, the cost of any cycle must be nonnegative, and thus the cost of the flow over these cycles must be positive, and in fact, equal to

$$\sum_{e \in E(G_{f^*})} \gamma'(e) f'(e) \geq 0.$$

By Theorem 3.10, the cost in G is

$$\sum_{e \in E} \gamma(e) f^o(e) - \sum_{e \in E} \gamma(e) f^*(e) = \sum_{e \in E(G_{f^*})} \gamma'(e) f'(e) \geq 0,$$

that is

$$\sum_{e \in E} \gamma(e) f^o(e) \geq \sum_{e \in E} \gamma(e) f^*(e)$$

thus f^* is an optimal flow (that is, with minimal cost). $\square$

We conclude this chapter by discussing Floyd-Warshall algorithm. The goal of this algorithm, proposed in 1962, is to compute all shortest paths of a weighted directed graph G. It is an example of so-called *dynamic programming.* Given a path $P = (1, \dots, l)$, any vertex in P different from $1, l$ is called an *intermediate vertex.* We denote by w_{ij} the weight $w(i,j)$ of the directed edge (i,j). The weight 0 is given to w_{ii} and the weight ∞ is a convention if there is no edge between i and j. We form an $n \times n$ matrix W whose coefficients are w_{ij} for $n = |V|$. We then denote by $d_{ij}^{(k)}$ the weight of a shortest path from i to j for which all intermediate vertices are in the set $\{1, \dots, k\}$. We similarly store $d_{ij}^{(k)}$ in a matrix $D^{(k)}$. Now to compute $d_{ij}^{(k)}$ knowing $d_{ij}^{(k-1)}$, observe that there are two ways for a shortest path to go from i to j:

- not using k: then only intermediate vertices in $\{1, \dots, k-1\}$ are visited, and the weight of a shortest path is $d_{ij}^{(k-1)}$.

- using k: observe that a shortest path does not pass through the same vertex twice, so k is visited exactly once. To ensure that the algorithm actually does that, we need the assumption that there is no directed cycle whose sum of weights is negative. Since this means we go from i to k, and then from k to j, we are then using a shortest path in both cases, so the length with be $d_{ik}^{(k-1)} + d_{kj}^{(k-1)}$.

We then recursively define

$$d_{ij}^{(k)} = \begin{cases} w_{ij},\ k = 0 \\ \min(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}),\ k \geq 1. \end{cases}$$

To keep track of paths, denote by $\pi_{ij}^{(k)}$ the predecessor of j on a shortest path from i, with intermediate vertices in $\{1, \ldots, k\}$:

$$\pi_{ij}^{(k)} = \begin{cases} \pi_{ij}^{(k-1)},\ d_{ij}^{(k-1)} \leq d_{ik}^{(k-1)} + d_{kj}^{(k-1)} \\ \pi_{kj}^{(k-1)},\ d_{ij}^{(k-1)} > d_{ik}^{(k-1)} + d_{kj}^{(k-1)} \end{cases}$$

with $\pi_{ij}^{(0)} = i$ for $i \neq j$ and $w_{ij} < \infty$ and *NIL* if $i = j$ or $w_{ij} = \infty$. Indeed, the first condition means that k was not used, and we store the predecessor of j from i, while for the second condition, k was used and the path went from k to j, so we store the predecessor of j from k.

Algorithm 9 Floyd-Warshall algorithm

Input: $G = (V, E)$ a weighted directed graph with weight w and no directed cycle whose sum of weights is negative.

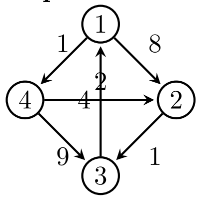
Output: $D^{(n)}$

```
1: $D^{(0)} \leftarrow W$.
2: for ($k = 1, \ldots, n$) do
3:     Initialize $D^{(k)}$.
4:     for ($i = 1, \ldots, n$) do
5:         for ($j = 1, \ldots, n$) do
6:             $d_{ij}^{(k)} = \min(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)})$
7:     Return $D^{(n)}$.
```

We notice that this involves storing the n matrices $D^{(k)}$, $k = 1, \ldots, n$. We can instead use a single matrix D, initialized to be W as above. A matrix to contain the predecessors is initialized at the same time. Then replace $d_{ij}^{(k)} = \min(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)})$ by **if** $d_{ij} > d_{ik} + d_{kj}$, **then** $d_{ij} \leftarrow d_{ik} + d_{kj}$, $\pi_{ij} = \pi_{kj}$.

The path can be recovered as follows. To know the path between two nodes i and j: if π_{ij} is NIL, output i, j. Otherwise, repeat the procedure using both i and π_{ij}, this gives the path from i to the predecessor of j from i, and π_{ij} and j, this gives the rest of the path.

Example 3.13. Consider the following weighted graph.



Then the matrix D at $k = 0$ is given by

$$D = \begin{bmatrix} 0 & 8 & \infty & 1 \\ \infty & 0 & 1 & \infty \\ 4 & \infty & 0 & \infty \\ \infty & 2 & 9 & 0 \end{bmatrix}, \ \Pi = \begin{bmatrix} NIL & 1 & NIL & 1 \\ NIL & NIL & 2 & NIL \\ 3 & NIL & NIL & NIL \\ NIL & 4 & 4 & NIL \end{bmatrix}.$$

At the first iteration, $k = 1$, we look at the condition $d_{ij} > d_{i1} + d_{1j}$, so for $i = 1$, $d_{1j} > d_{11} + d_{1j} = d_{1j}$ which is never true, for $i = 2$, $d_{2j} > d_{21} + d_{1j} = \infty$ which is never true either, the same holds for $i = 4$, so we just need to consider $i = 3$, that is $d_{3j} > d_{31} + d_{1j} = 4 + d_{1j}$. This gives

$$D = \begin{bmatrix} 0 & 8 & \infty & 1 \\ \infty & 0 & 1 & \infty \\ 4 & 12 & 0 & 5 \\ \infty & 2 & 9 & 0 \end{bmatrix}, \ \Pi = \begin{bmatrix} NIL & 1 & NIL & 1 \\ NIL & NIL & 2 & NIL \\ 3 & \pi_{12} = 1 & NIL & \pi_{14} = 1 \\ NIL & 4 & 4 & NIL \end{bmatrix}.$$

At the second iteration, $k = 2$, we look at the condition $d_{ij} > d_{i2} + d_{2j}$, so for $i = 1$, $d_{1j} > d_{12} + d_{2j} = 8 + d_{2j}$ and for $i = 4$, $d_{4j} > d_{42} + d_{2j} = 2 + d_{2j}$, the other cases are never true. This gives

$$D = \begin{bmatrix} 0 & 8 & 9 & 1 \\ \infty & 0 & 1 & \infty \\ 4 & 12 & 0 & 5 \\ \infty & 2 & 3 & 0 \end{bmatrix}, \ \Pi = \begin{bmatrix} NIL & 1 & \pi_{23} = 2 & 1 \\ NIL & NIL & 2 & NIL \\ 3 & 1 & NIL & 1 \\ NIL & 4 & \pi_{23} = 2 & NIL \end{bmatrix}.$$

When $k = 3$, we have $d_{ij} > d_{i3} + d_{3j}$ which we consider for $i = 1, 2, 4$:

$$D = \begin{bmatrix} 0 & 8 & 9 & 1 \\ 5 & 0 & 1 & 6 \\ 4 & 12 & 0 & 5 \\ 7 & 2 & 3 & 0 \end{bmatrix}, \ \Pi = \begin{bmatrix} NIL & 1 & 2 & 1 \\ 3 & NIL & 2 & 1 \\ 3 & 1 & NIL & 1 \\ 3 & 4 & 4 & NIL \end{bmatrix}.$$

Finally for $k = 4$, we have $d_{ij} > d_{i4} + d_{4j}$ which we consider for $i = 1, 2, 3$:

$$D = \begin{bmatrix} 0 & 3 & 4 & 1 \\ 5 & 0 & 1 & 6 \\ 4 & 7 & 0 & 5 \\ 7 & 2 & 3 & 0 \end{bmatrix}, \ \Pi = \begin{bmatrix} NIL & 4 & 2 & 1 \\ 3 & NIL & 2 & 1 \\ 3 & 4 & NIL & 1 \\ 3 & 4 & 4 & NIL \end{bmatrix}.$$

We know from D that the shortest path from 1 to 3 is of weight 4, to know the path itself, we look at the path from 1 to $\pi_{1,3} = 2$, which gives 4, so we know that the path starts with 1,4,2, and then we look at the path from 2 to 3 which is 2, so the total path is 1,4,2,3. We can check that this path has indeed a weight of 4.

This algorithm works assuming there is no cycle whose sum of weights is negative. However, if the graph has such cycles, we can use the algorithm to actually detect them. The length of a path from i to itself is set to 0 when the algorithm starts. Now a path from i to itself can only improve if the length is less than zero, but that would mean a negative cycle. Thus once the algorithm terminates, if there is a diagonal coefficient in the matrix D which is negative, that means this node is involved in a negative cycle. Thus the presence of at least one negative diagonal coefficient reveals the presence of at least one negative cycle. As an example, one can replace the weight $w(3,1)$ in the above example to be -10, this creates a negative cycle (see Exercise 34).

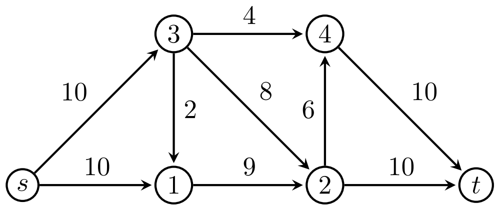
3.5 Exercises

Exercise 25. Show that

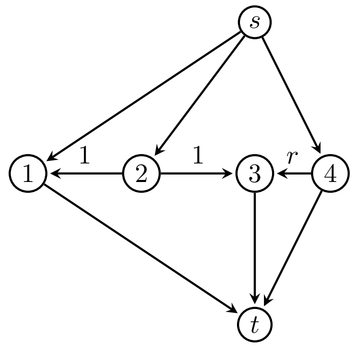
$$|f| = \sum_{u \in I(t)} f(u,t),$$

that is, the strength of the flow is the sum of the values of f on edges entering the sink.

Exercise 26. Use Ford-Fulkerson algorithm to find a maximum flow in the following network:



Exercise 27. Here is a famous example of network (found on wikipedia and in many other places) where Ford-Fulkerson may not terminate. The edges capacities are 1 for $(2,1)$, $r = (\sqrt{5}-1)/2$ for $(4,3)$, 1 for $(2,3)$, and M for all other edges, where $M \geq 2$ is any integer.



1. Explain why Ford-Fulkerson algorithm may not be able to terminate.

2. Apply Edmonds-Karp algorithm to find a maximum flow in this network.

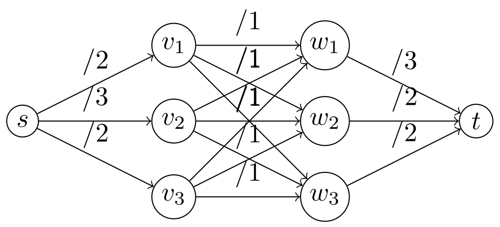
Exercise 28. Menger's Theorem states the following. Let G be a directed graph, let u, v be distinct vertices in G. Then the maximum number of pairwise edge-disjoint paths from u to v equals the minimum number of edges whose removal from G destroys all directed paths from u to v. Prove Menger's Theorem using the Max Flow- Min Cut Theorem.

Exercise 29. Let $G = (V, E)$ be an undirected graph that remains connected after removing any $k-1$ edges. Let s, t be any two nodes in V.

1. Construct a network G' with the same vertices as G, but for each edge $\{u, v\}$ in G, create in G' two directed edges (u, v), (v, u) both with capacity 1. Take s for the source and t for the sink of the network G'. Show that if there are k edge-disjoint directed paths from s to t in G', then there are k edge-disjoint paths from s to t in G.

2. Use the max-flow min-cut theorem to show that there are k edge-disjoint paths from s to t in G'.

Exercise 30. 1. Compute a maximal flow in the following network, where each edge (v_i, w_j) has a capacity of 1, for $i, j = 1, 2, 3$:



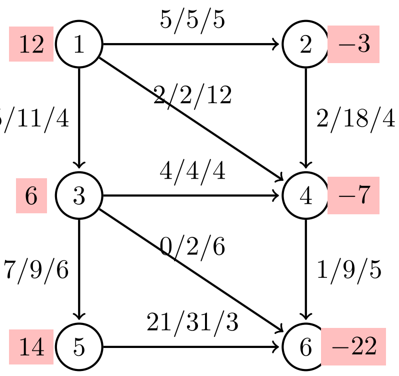
2. Interpret the above as a placement of a number of balls of different colours into bins of different capacities, such that no two balls with the same colour belong to the same bin. More generally, describe in terms of flow over a

network the problem of placing b_i balls of a given colour, for $i = 1, \ldots, m$ colours, into n bins of different capacities c_j, $j = 1, \ldots, n$, such that no two balls with the same colour belong to the same bin (b_i, c_j are all positive integers).

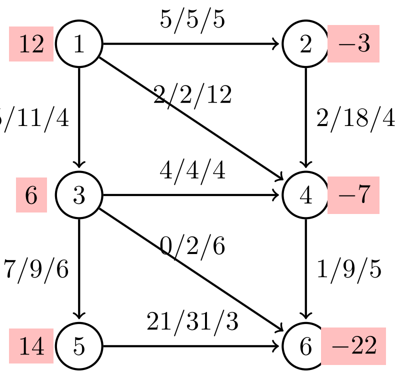
3. Give a necessary and sufficient condition on the max flow of the above graph to ensure that the ball placement problem has a solution (that is a condition (C) such that (C) holds if and only if the ball placement problem has a solution).

Exercise 31. Show that one can always consider min-cost-flow networks were lower capacities are zero, that is, if a network has lower capacities which are not zero, the network can be replaced by an equivalent network were all lower capacities are zero.

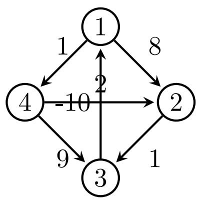
Exercise 32. Compute the residual graph G_f of the following graph G with respect to the flow given. Each edge e is labeled with $(f(e), c(e), \gamma(e))$.



Exercise 33. Consider the following graph G. Each edge e is labeled with $(f(e), c(e), \gamma(e))$. Solve the min-cost-low problem for this graph, using the flow given as initial feasible flow.



Exercise 34. Use Floyd-Warshall algorithm to detect the presence of at least one negative cycle in the graph below.


1
2
3
4
1
8
2
-10
9
1