Graph Algorithms

Max Flow - Min Cut

Michael Lampis

November 13, 2025

The story so far

- (Short) Reachability problems:
  - Is there a path from s to t? (Strongly) connected components? ...
  - What is the shortest path from s to t? From everyone to everyone?
  - What is the cheapest way to keep everyone connected?

The story so far

- (Short) Reachability problems:
  - Is there a path from s to t? (Strongly) connected components? ...
  - What is the shortest path from s to t? From everyone to everyone?
  - What is the cheapest way to keep everyone connected?
- Algorithms:
  - Unweighted Reachability: BFS/DFS $O(m+n)$ time
  - Single-Source Shortest Paths:
    - Unweighted: BFS $O(m+n)$ time
    - Positive weights: Dijkstra $O((m+n)\log n)$ time (with min-heaps)
    - General weights: Bellman-Ford $O(mn)$ time
  - Minimum Spanning Tree:
    - Prim's and Kruskal's algorithms: $O(m \log n)$ time

The story so far

- (Short) Reachability problems:
  - Is there a path from s to t? (Strongly) connected components? ...
  - What is the shortest path from s to t? From everyone to everyone?
  - What is the cheapest way to keep everyone connected?
- Algorithms:
  - Unweighted Reachability: BFS/DFS $O(m+n)$ time
  - Single-Source Shortest Paths:
    - Unweighted: BFS $O(m+n)$ time
    - Positive weights: Dijkstra $O((m+n)\log n)$ time (with min-heaps)
    - General weights: Bellman-Ford $O(mn)$ time
  - Minimum Spanning Tree:
    - Prim's and Kruskal's algorithms: $O(m \log n)$ time
- New problem: Maximum Flow - Minimum Cut
  - Find minimum weight set of edges that **destroys** connectedness.

Problem Definition

Definition

Definition

For an edge-weighted digraph $G = (V, A)$, a **Minimum Edge Cut** between vertices $s, t \in V$ is a set of arcs $A' \subseteq A$ such that

1. $G = (V, A \setminus A')$ has no directed path from s to t.
2. The weight of A' is minimum among sets satisfying (1).

Definition

Definition

For an edge-weighted digraph $G = (V, A)$, a **Minimum Edge Cut** between vertices $s, t \in V$ is a set of arcs $A' \subseteq A$ such that

1. $G = (V, A \setminus A')$ has no directed path from s to t.
2. The weight of A' is minimum among sets satisfying (1).

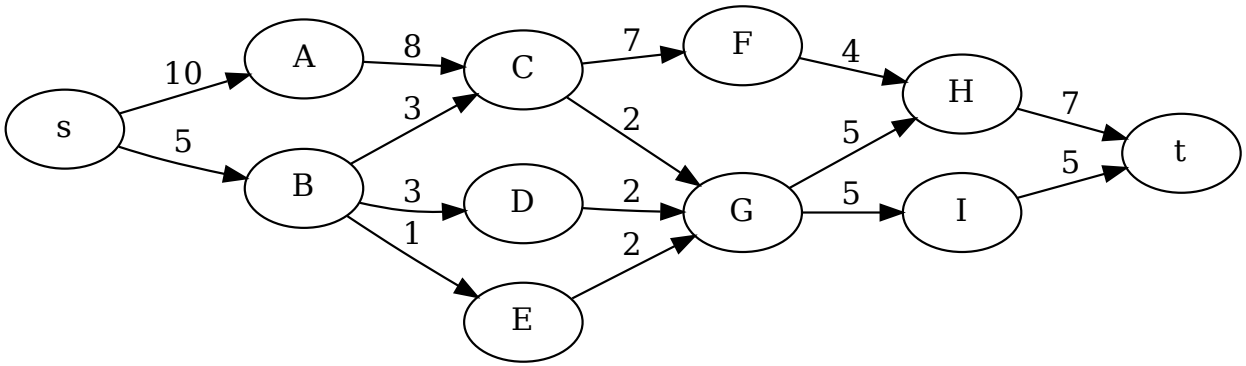
- Algorithmic problem: Given $G = (V, A)$ and $s, t \in V$, compute minimum $s - t$ cut.

Equivalent formulation:

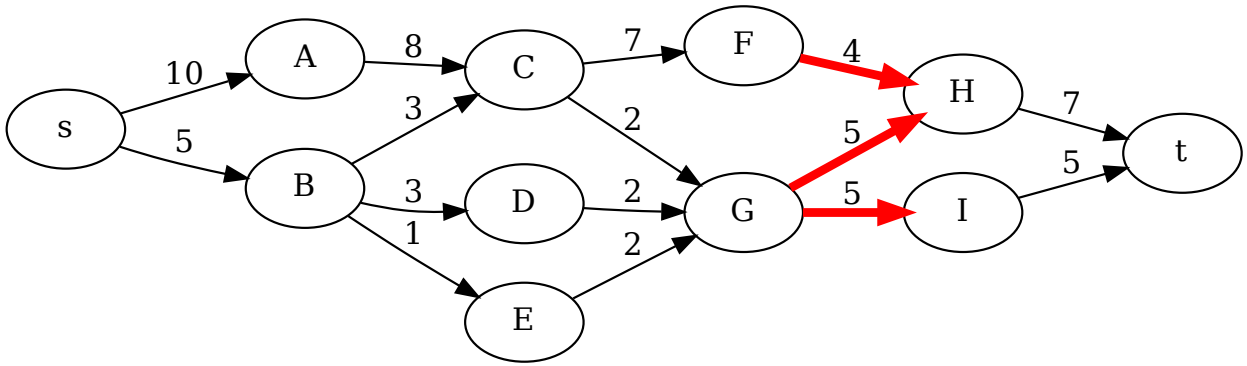
Definition

In the Min $s - t$ Cut problem, we are given edge-weighted digraph $G = (V, A)$, $s, t \in V$ and are asked to compute a set $S \subseteq V$ with $s \in S, t \notin S$ such that $\sum_{x \in S} \sum_{y \notin S} w(xy)$ is minimum.

Example

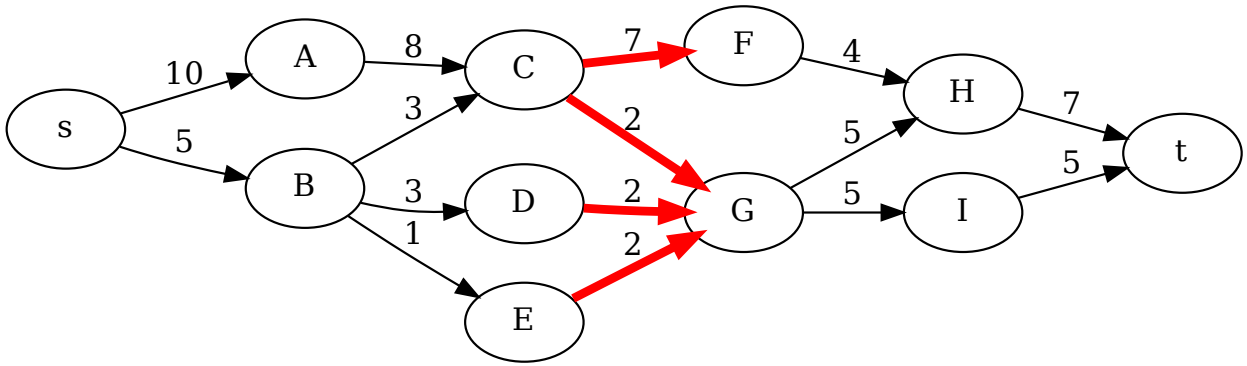


Example



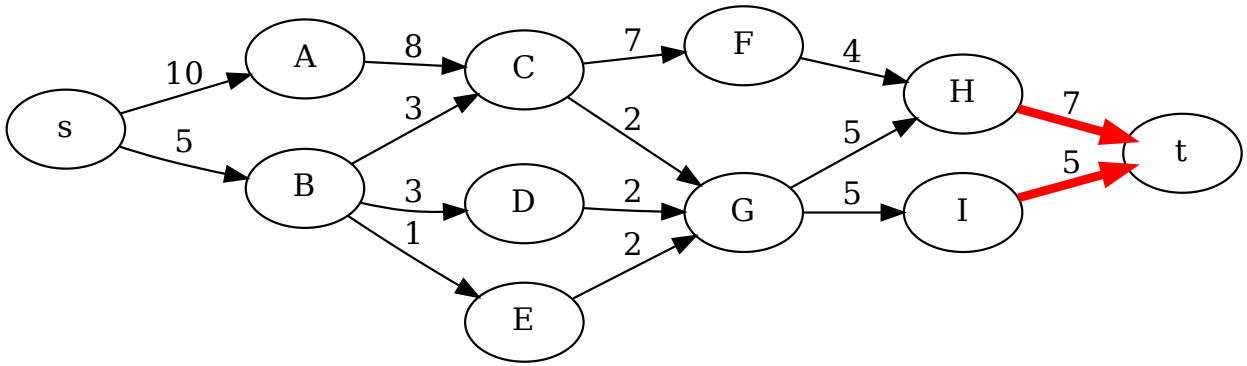
Cost: 14

Example



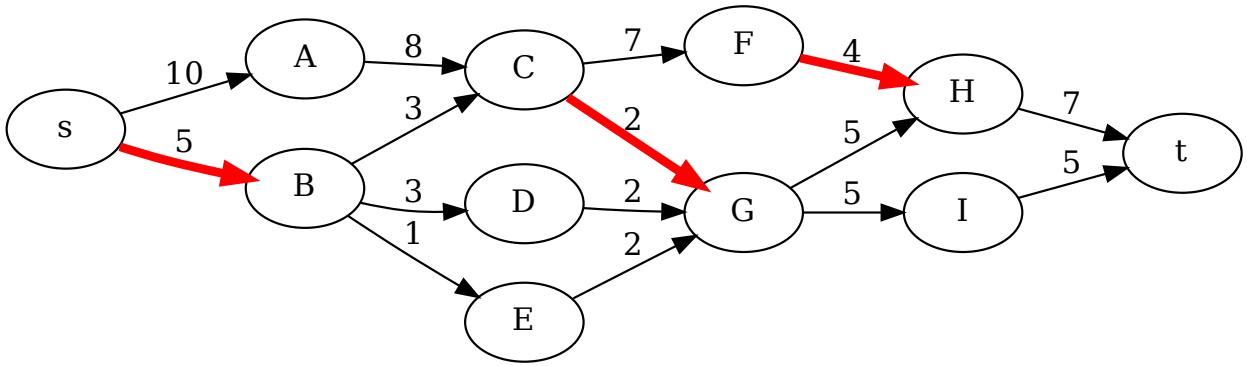
Cost: 13

Example



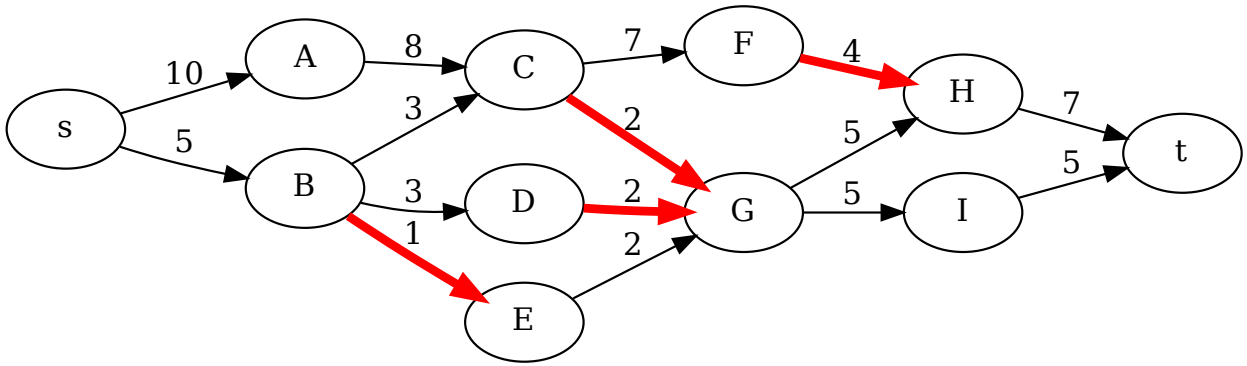
Cost: 12

Example



Cost: 11

Example



Cost: 9

Obvious (bad) algorithms

Lemma

The two formulations are equivalent.

Proof.

- Given cut-set A', define S as set of vertices reachable from s.
- Given set S, define A' as set of arcs from S to $V \setminus S$.

□

Obvious (bad) algorithms

Lemma

The two formulations are equivalent.

Proof.

- Given cut-set A', define S as set of vertices reachable from s.
- Given set S, define A' as set of arcs from S to $V \setminus S$.

□

- $O(2^m \cdot m)$ time: try all cut-sets

Obvious (bad) algorithms

Lemma

The two formulations are equivalent.

Proof.

- Given cut-set A', define S as set of vertices reachable from s.
- Given set S, define A' as set of arcs from S to $V \setminus S$.

□

- $O(2^m \cdot m)$ time: try all cut-sets
- $O(2^n \cdot m)$ time: try all sets S

Pretty terrible! Can we do polynomial time?

Flows

Definition

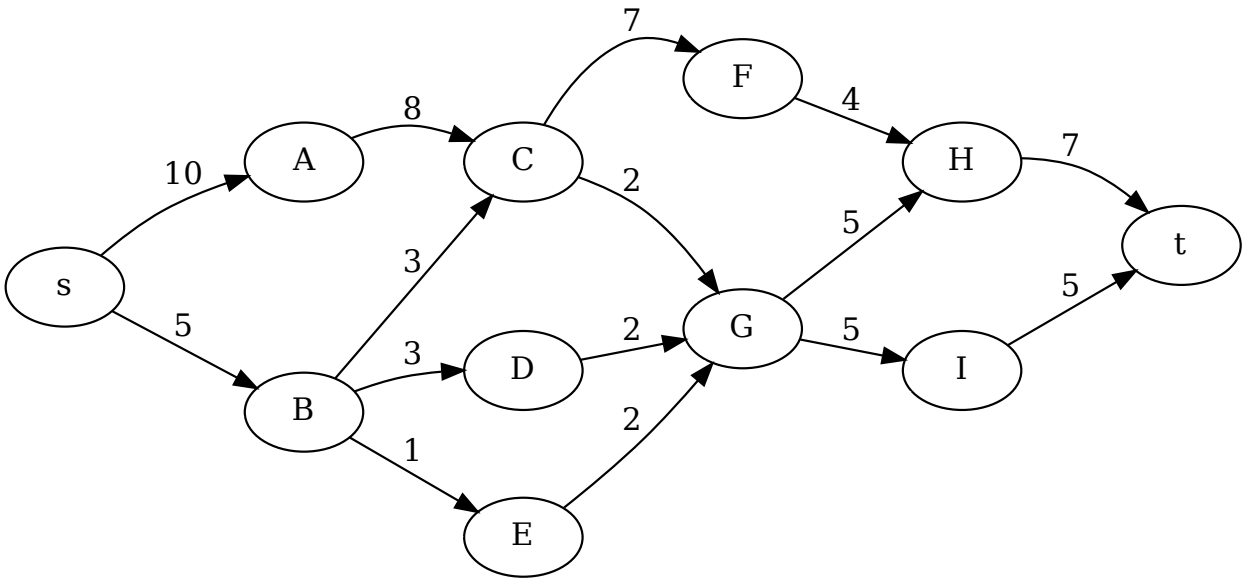
Given a digraph $G = (V, A)$, $s, t \in V$ and a capacity function $\mathbf{c} : A \to \mathbb{Q}^+$, a flow is a function $\mathbf{f} : A \to \mathbb{Q}^+$ satisfying the following:

1. Capacity-respecting: for all $a \in A$ we have $\mathbf{f}(a) \leq \mathbf{c}(a)$
2. Flow-conservation: for all $x \in V \setminus \{s, t\}$ we have $\sum_{z \in N^-(x)} \mathbf{f}(zx) = \sum_{y \in N^-(x)} \mathbf{f}(xy)$

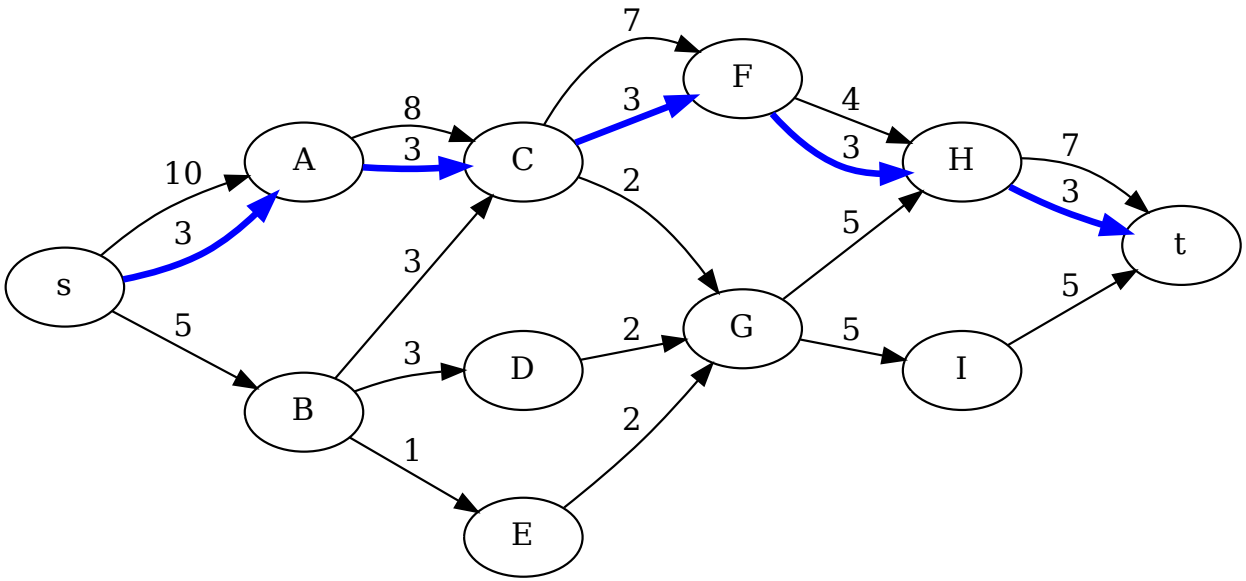
Intuition:

- s is sending something (information, energy, etc.) to t.
- Capacities of arcs must be respected.
- Flows can be re-arranged in vertices, but must be conserved and go from s to t
- **Value** of flow $\mathbf{f}$ is total flow coming out of s
- We seek a flow of **maximum value**

Example

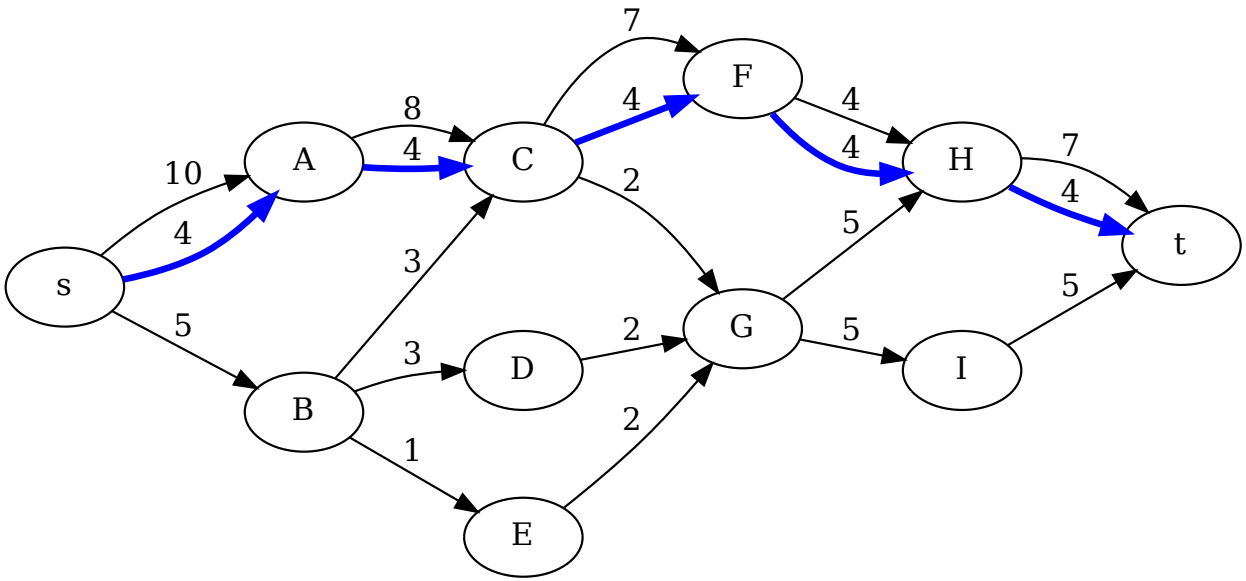


Example



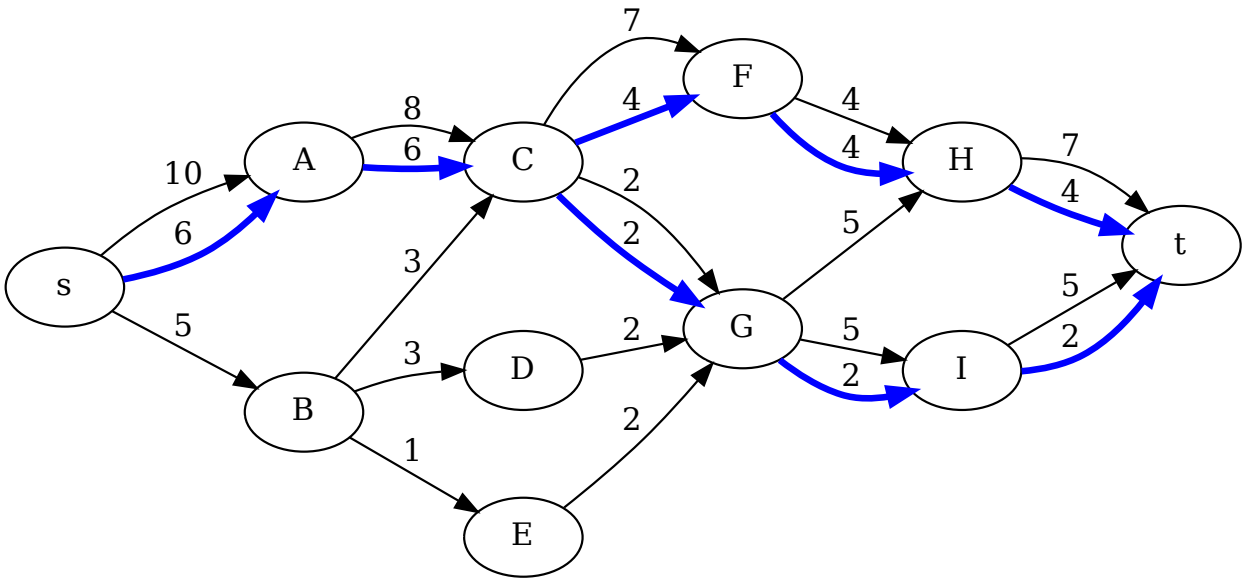
Flow: 3

Example



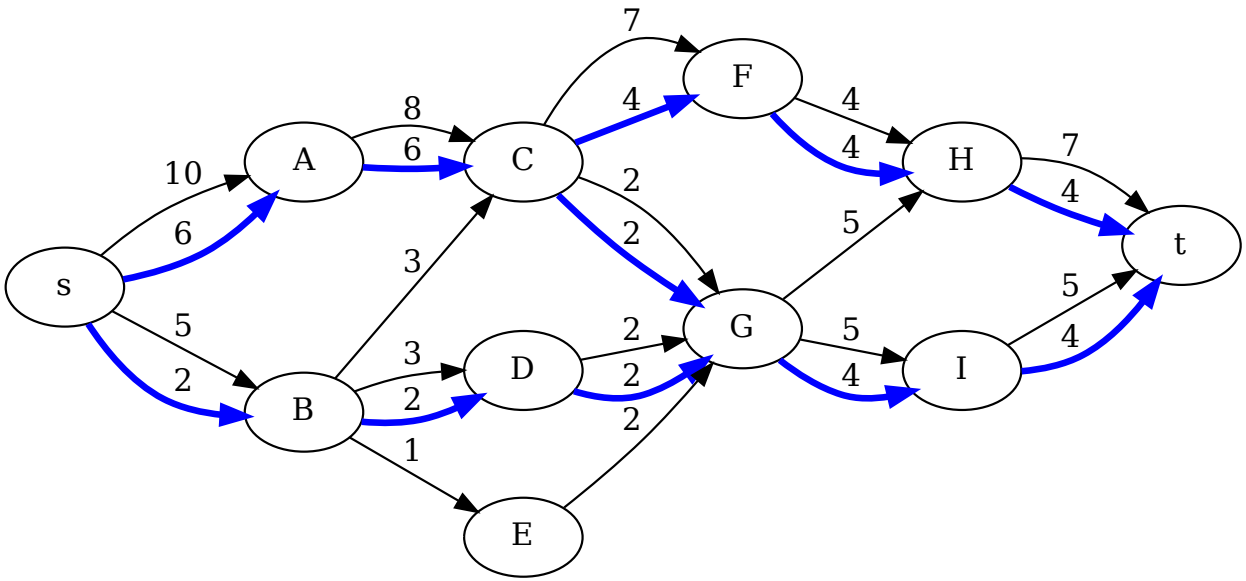
Flow: 4

Example



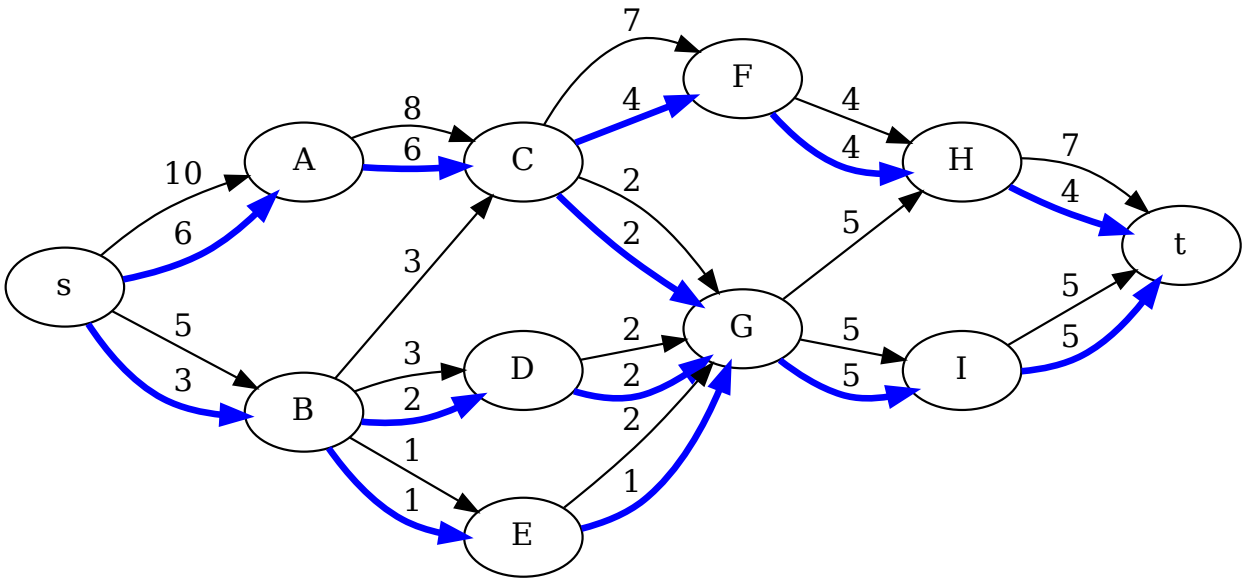
Flow: 6

Example



Flow: 8

Example



Flow: 9

Max-Flow Min-Cut: easy connection

Lemma

For all directed graphs $G = (V, A)$, s, t, and weight function $w : A \to \mathbb{Q}^+$ if f if a feasible flow value from s to t and c is a feasible cut from s to t, then $f \leq c$.

Corollary

If f^* is the max flow value and c^* is the minimum cut value, then $f^* \leq c^*$.

Max-Flow Min-Cut: easy connection

Lemma

For all directed graphs $G = (V, A)$, s, t, and weight function $w : A \to \mathbb{Q}^+$ if f if a feasible flow value from s to t and c is a feasible cut from s to t, then $f \leq c$.

Corollary

If f^* is the max flow value and c^* is the minimum cut value, then $f^* \leq c^*$.

Proves that for previous example $f^* = c^* = 9$.

Lemma

For all directed graphs $G = (V, A)$, s, t, and weight function $w : A \to \mathbb{Q}^+$ if f if a feasible flow value from s to t and c is a feasible cut from s to t, then $f \leq c$.

Lemma

For all directed graphs $G = (V, A)$, s, t, and weight function $w : A \to \mathbb{Q}^+$ if f if a feasible flow value from s to t and c is a feasible cut from s to t, then $f \leq c$.

Proof.

- Let A' be a feasible cut, S the set of vertices reachable from s.
- Let $\mathbf{f}$ be a flow function of value f.
- We have $f \leq \sum_{x \in S} \sum_{y \notin S} \mathbf{f}(xy)$.
- $\sum_{x \in S} \sum_{y \notin S} \mathbf{f}(xy) \leq \sum_{x \in S} \sum_{y \notin S} \mathbf{c}(xy) = c$.

(Weak) duality

- What we know: **any** flow has value $\leq$ than **any** cut

(Weak) duality

- What we know: **any** flow has value $\leq$ than **any** cut
- Common situation in combinatorial optimization:
  - We want to optimize (say, minimize) something
    - For example: Min Weight Spanning Tree (variants)
  - We find a dual objective (maximization) problem that lower bounds the solution
    - For example: the diameter (max distance of any two vertices) is a lower bound on MST (why?)

(Weak) duality

- What we know: **any** flow has value $\leq$ than **any** cut
- Common situation in combinatorial optimization:
  - We want to optimize (say, minimize) something
    - For example: Min Weight Spanning Tree (variants)
  - We find a dual objective (maximization) problem that lower bounds the solution
    - For example: the diameter (max distance of any two vertices) is a lower bound on MST (why?)
- In situations where the original problem is hard to solve (NP-hard), this trick at least allows us to get an idea for how far we are from the optimal solution.

(Weak) duality

- What we know: **any** flow has value $\leq$ than **any** cut
- Common situation in combinatorial optimization:
  - We want to optimize (say, minimize) something
    - For example: Min Weight Spanning Tree (variants)
  - We find a dual objective (maximization) problem that lower bounds the solution
    - For example: the diameter (max distance of any two vertices) is a lower bound on MST (why?)
- In situations where the original problem is hard to solve (NP-hard), this trick at least allows us to get an idea for how far we are from the optimal solution.
- In some **rare** situations we can find a dual problem where the optimal values of the two objectives **match**!!!
- This is the case for Max-Flow and Min-Cut

The Ford-Fulkerson Method

The Ford-Fulkerson Method

- Strategy: reduce Max-Flow computation to Reachability computation
  - Find a path from s to t
  - Saturate it!
  - Update the graph
  - Repeat

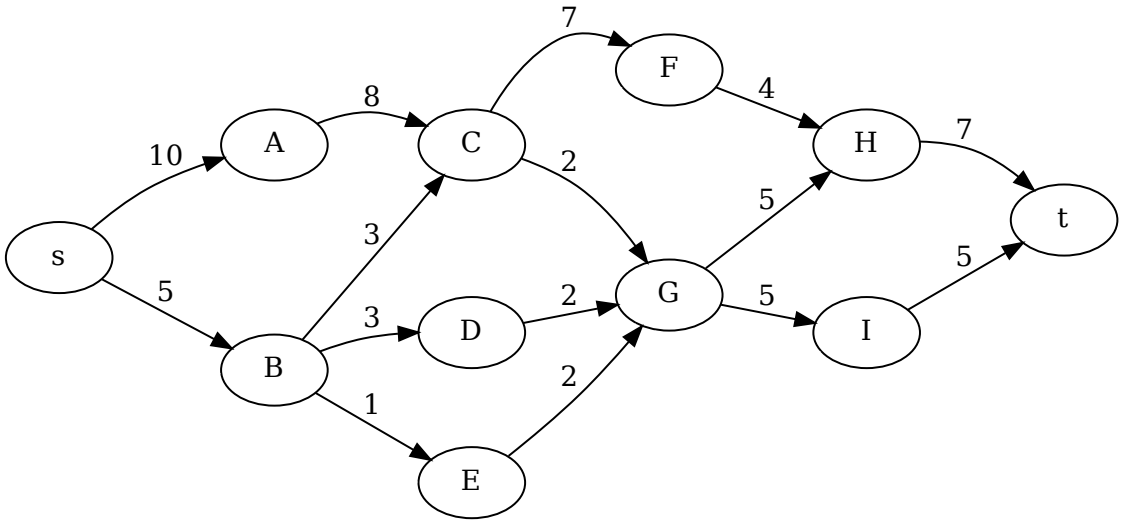
The Ford-Fulkerson Method

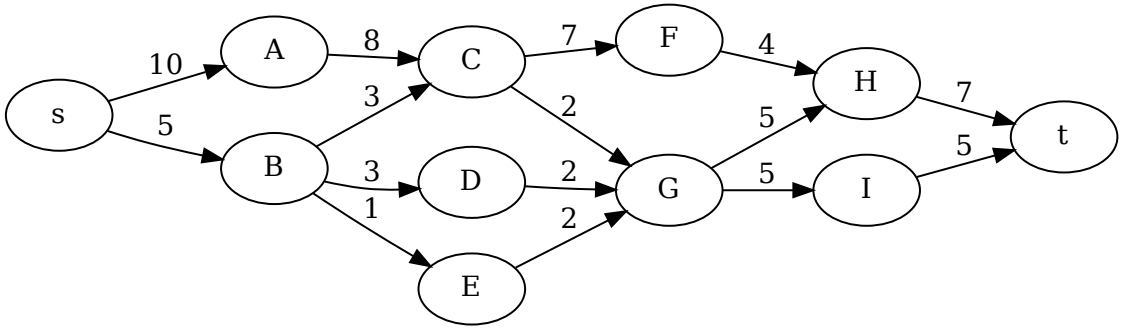
- Strategy: reduce Max-Flow computation to Reachability computation
  - Find a path from s to t
  - Saturate it!
  - Update the graph
  - Repeat
- Saturate: send as much capacity as allowed by the arc of minimum capacity in the path
  - $\Rightarrow$ this arc is then saturated (cannot take more flow)

The Ford-Fulkerson Method

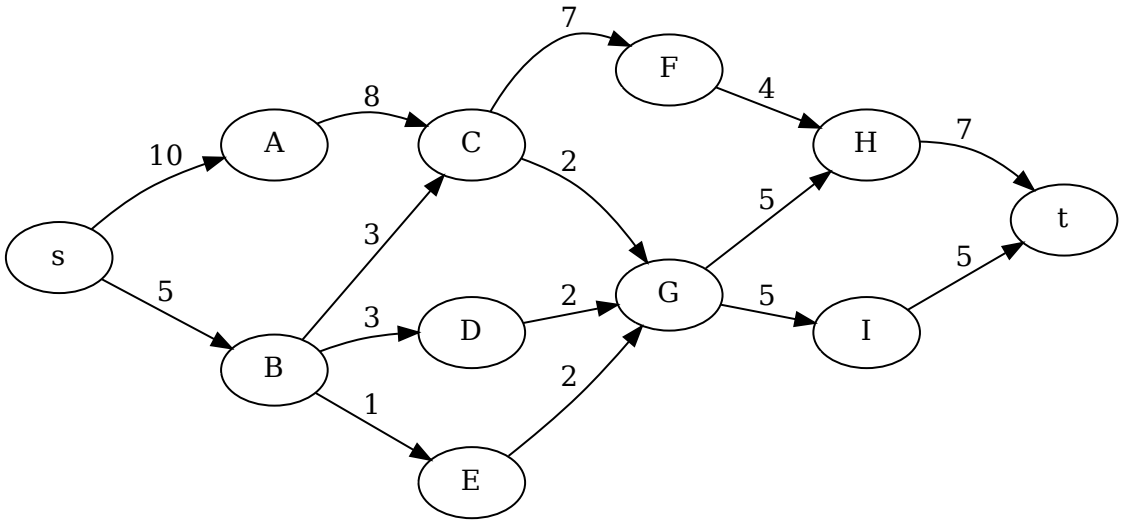
- Strategy: reduce Max-Flow computation to Reachability computation
  - Find a path from s to t
  - Saturate it!
  - Update the graph
  - Repeat
- Saturate: send as much capacity as allowed by the arc of minimum capacity in the path
  - $\Rightarrow$ this arc is then saturated (cannot take more flow)
- How to update the graph so that we respect future capacities?
  - Naïve method: subtract flow from capacities of used arcs (doesn't work!!)
  - Correct method: Construct **Residual network**
- Complexity:
  - Depends heavily on how paths are selected (more later!)

Example

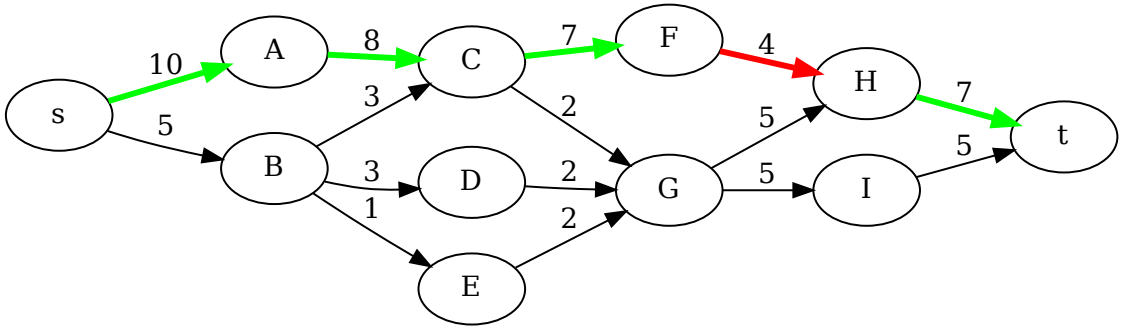

s
A
B
C
D
E
F
G
H
I
t
10
5
8
3
3
1
7
2
2
2
4
5
5
7
5


s
A
B
C
D
E
F
G
H
I
t
10
5
8
3
3
1
7
2
2
2
4
5
5
7
5

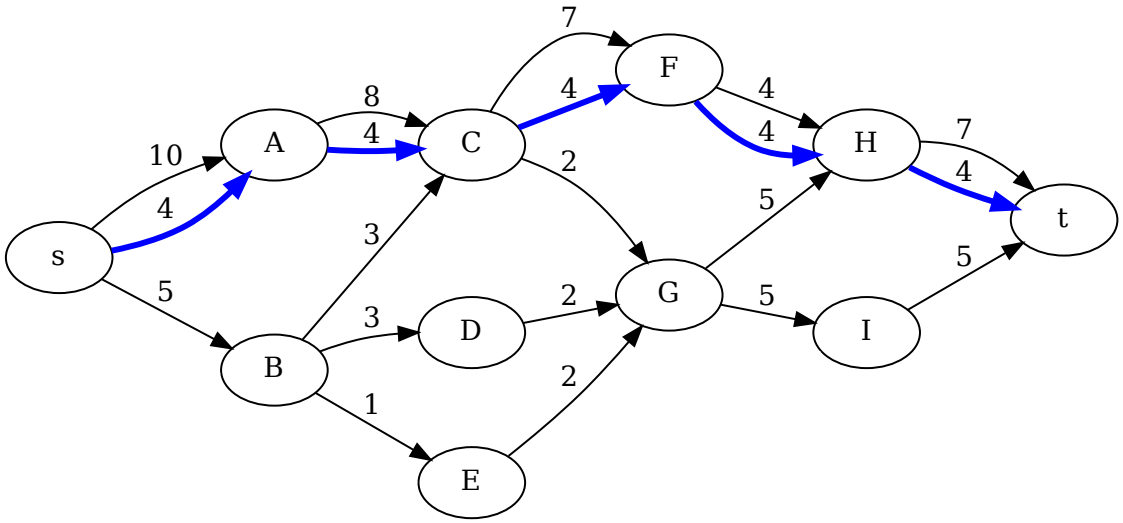
Example



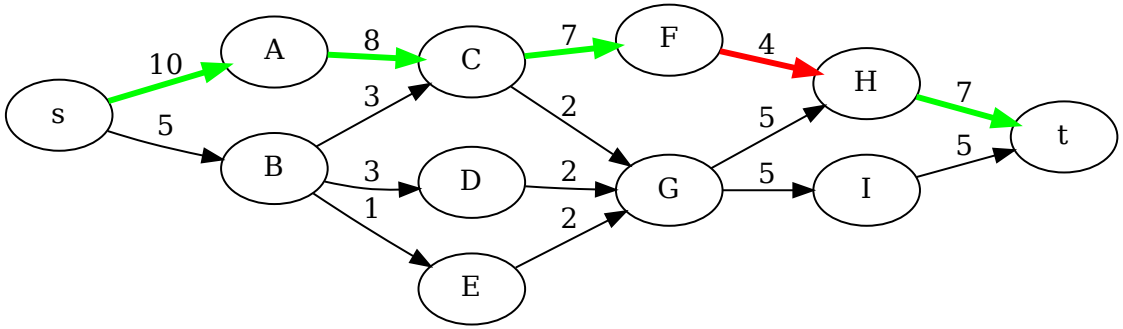
Find an $s \rightarrow t$ path.
Min capacity:4



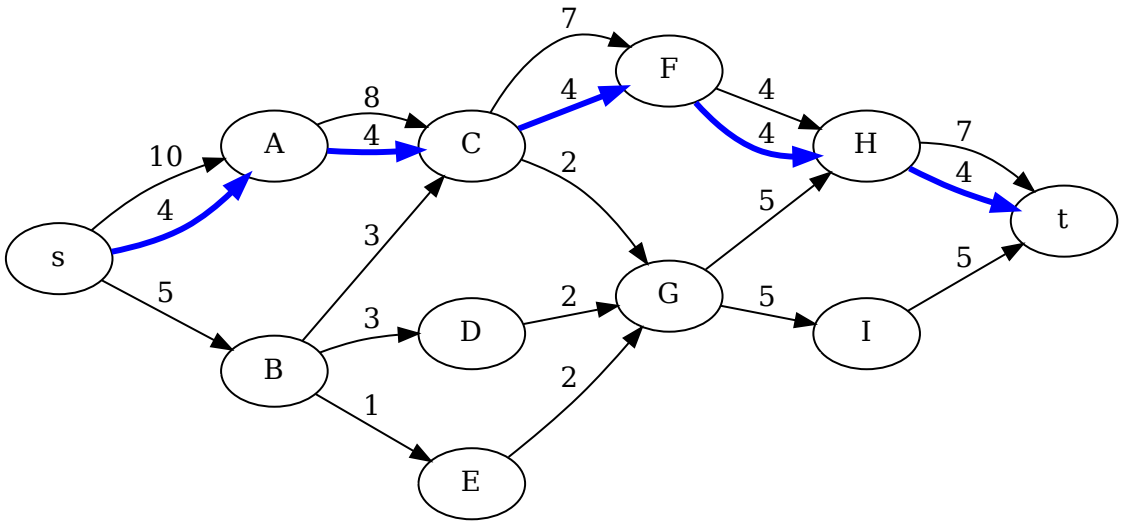
Example



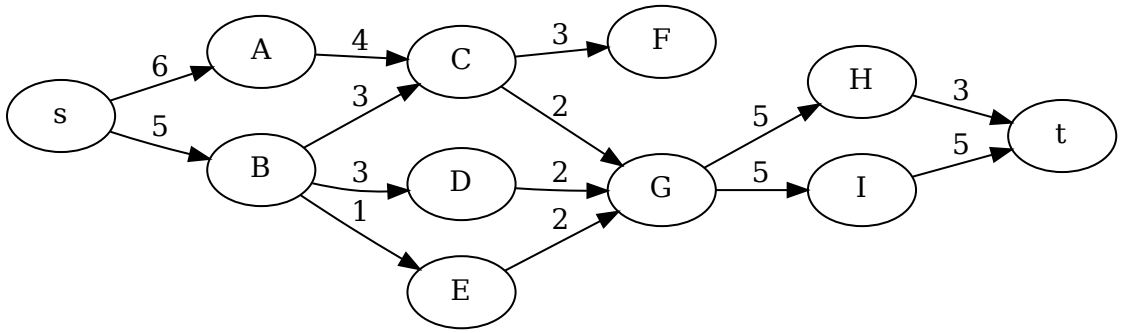
Saturate the path



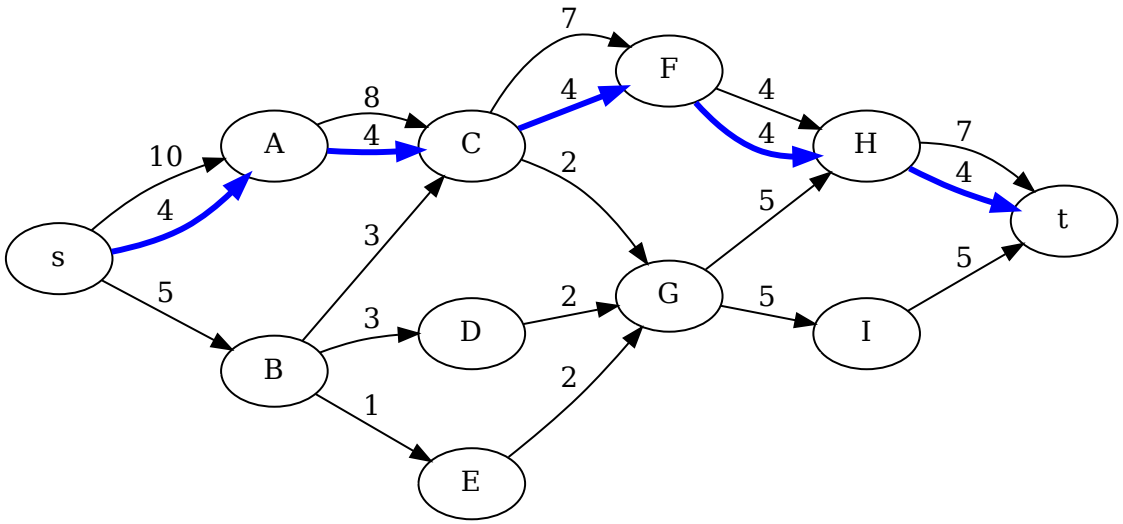
Example



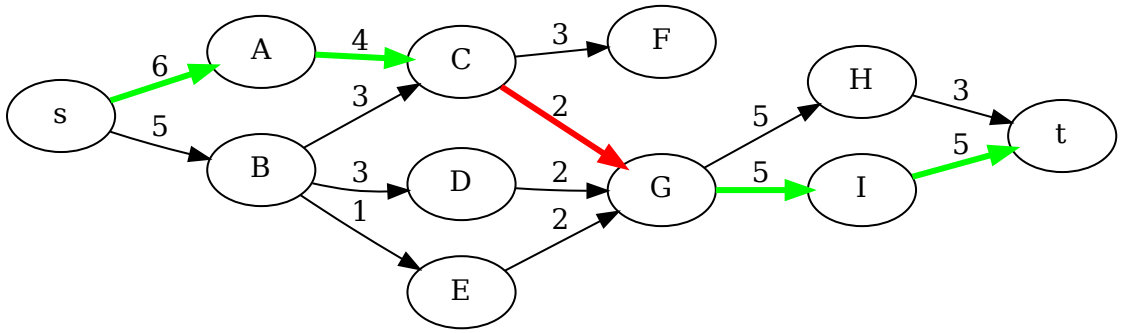
Update Capacities



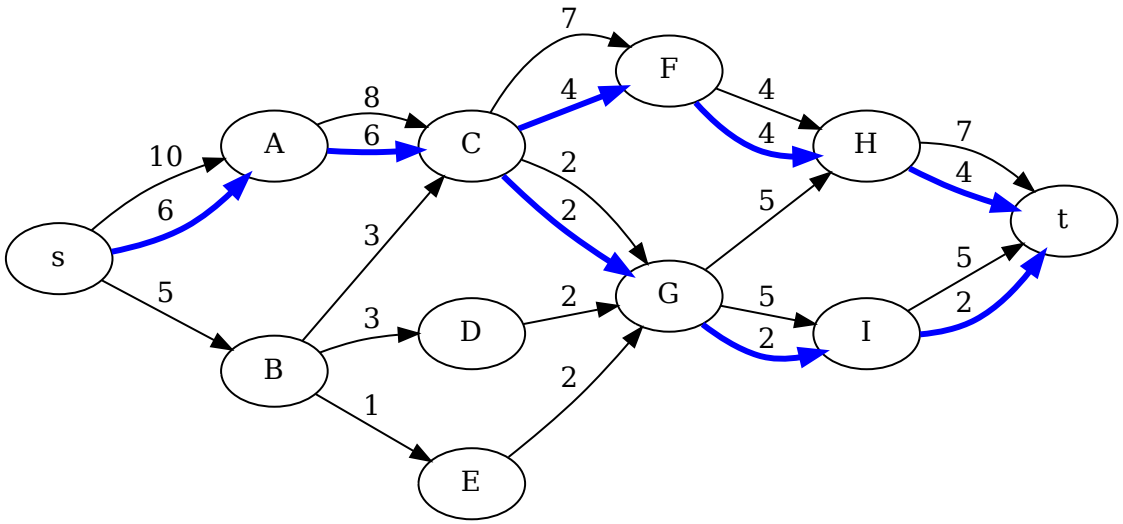
Example



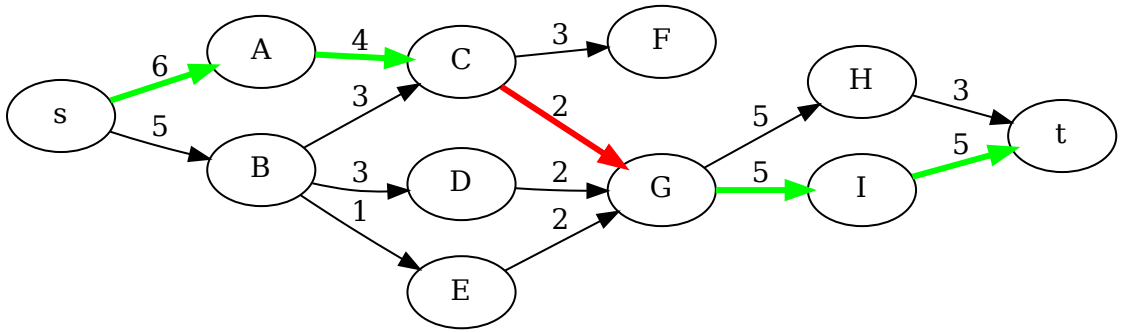
Find an $s \rightarrow t$ path.
Min capacity:2



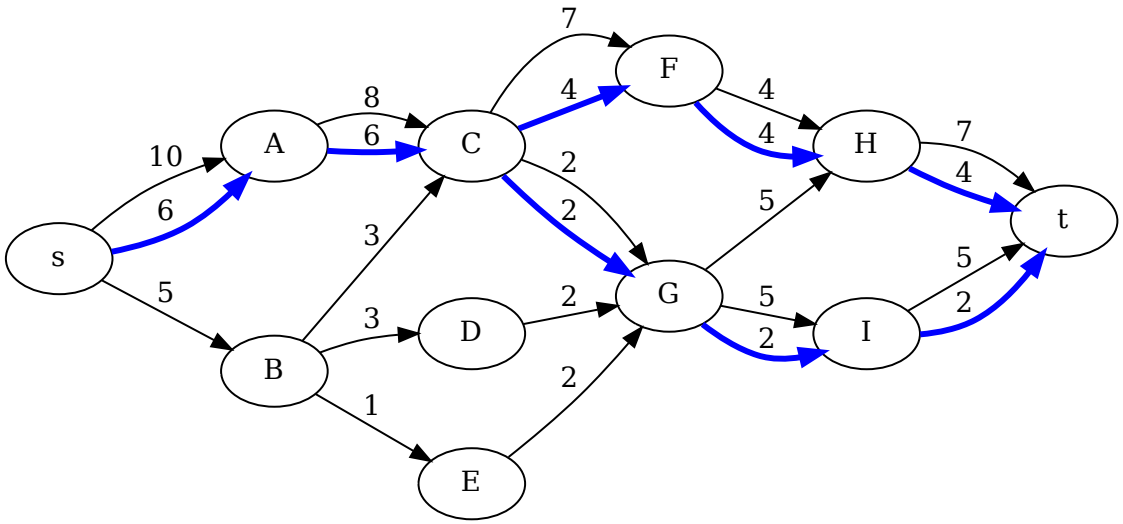
Example



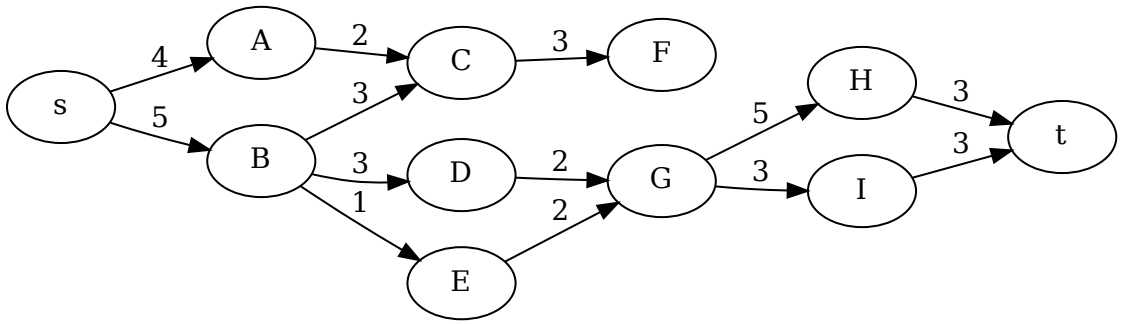
Saturate the path



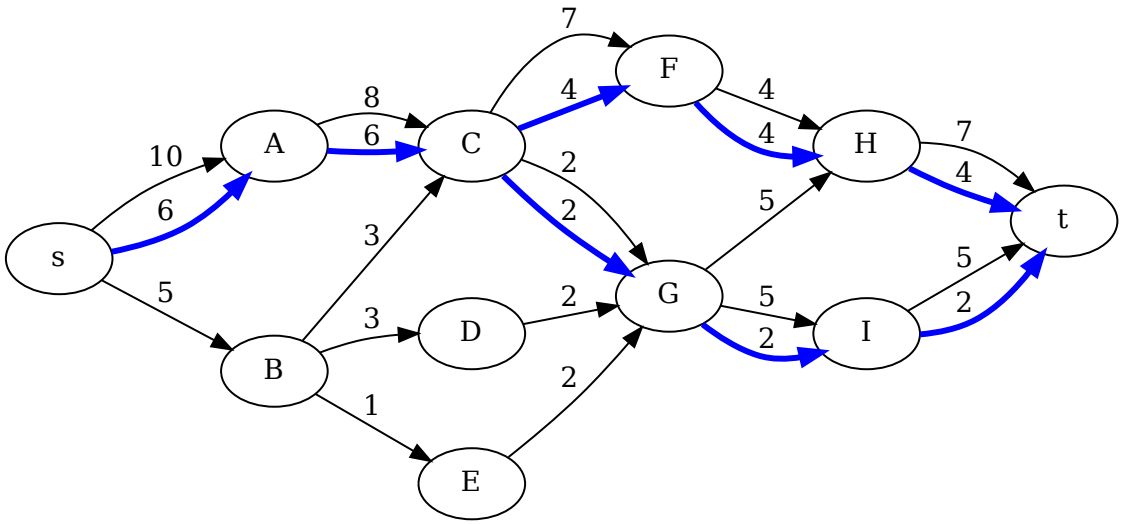
Example



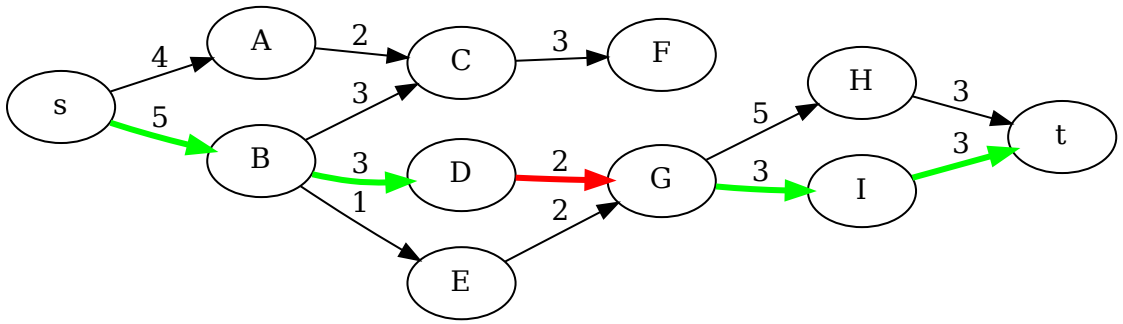
Update Capacities



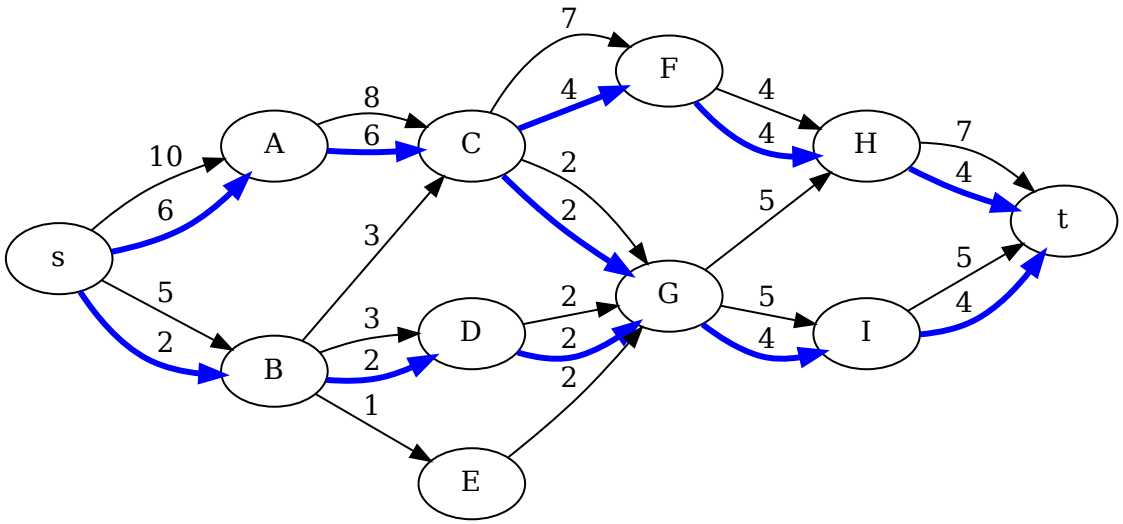
Example


s
A
B
C
D
E
F
G
H
I
t
10
6
5
8
6
3
3
1
7
4
2
2
2
2
4
4
5
5
2
7
4
5
2

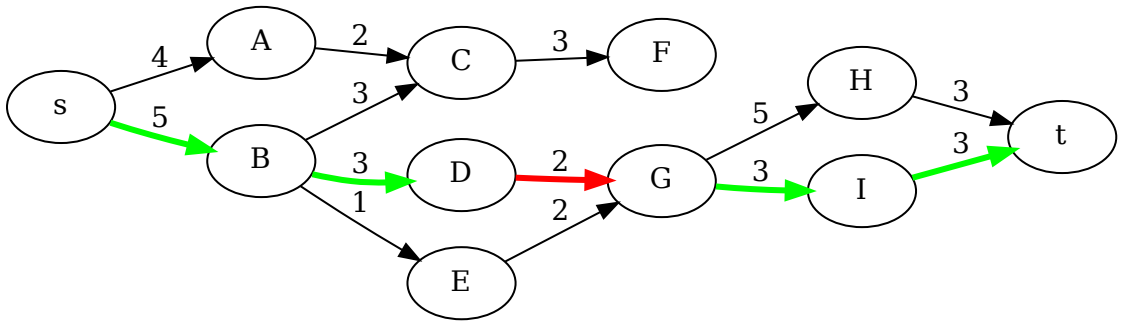
. . .


s
A
B
C
D
E
F
G
H
I
t
4
5
2
3
3
1
3
2
2
5
3
3
3

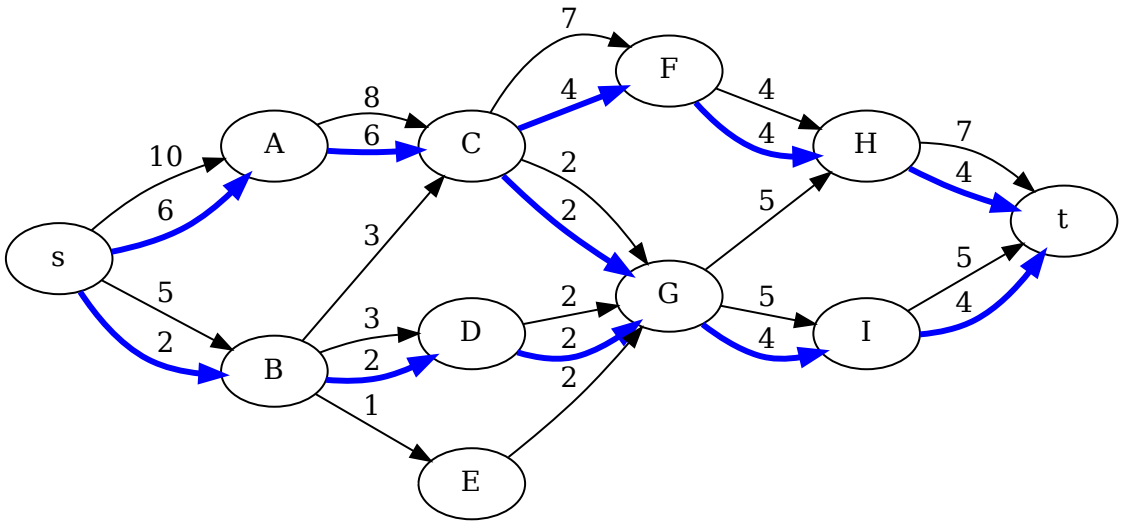
Example


s
A
B
C
D
E
F
G
H
I
t
10
6
5
2
8
6
3
3
2
1
7
4
2
2
2
2
2
4
4
4
5
5
4
7
4
5
4

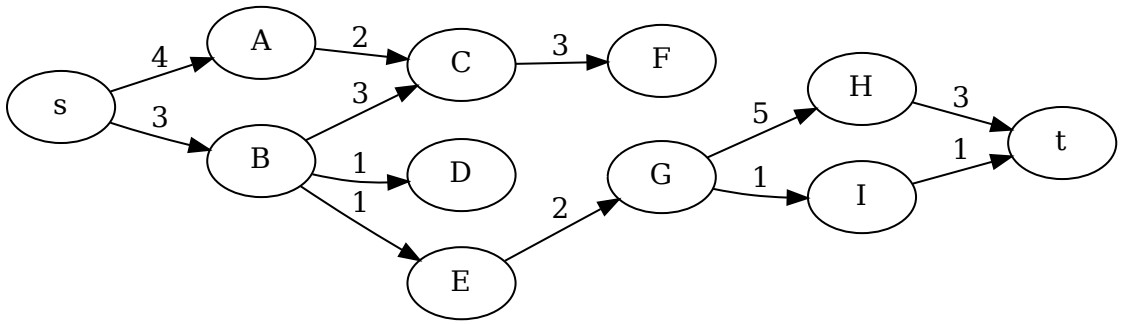
. . .


s
A
B
C
D
E
F
G
H
I
t
4
5
2
3
3
1
3
2
2
5
3
3
3

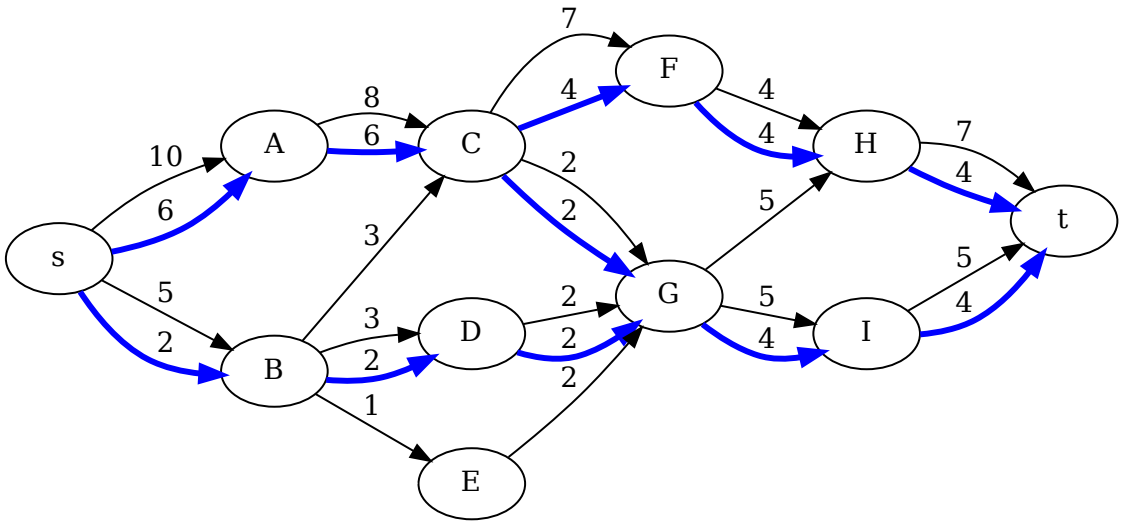
Example


s
A
B
C
D
E
F
G
H
I
t
10
6
5
2
8
6
3
3
2
1
7
4
2
2
2
2
2
4
4
4
5
5
4
7
4
5
4

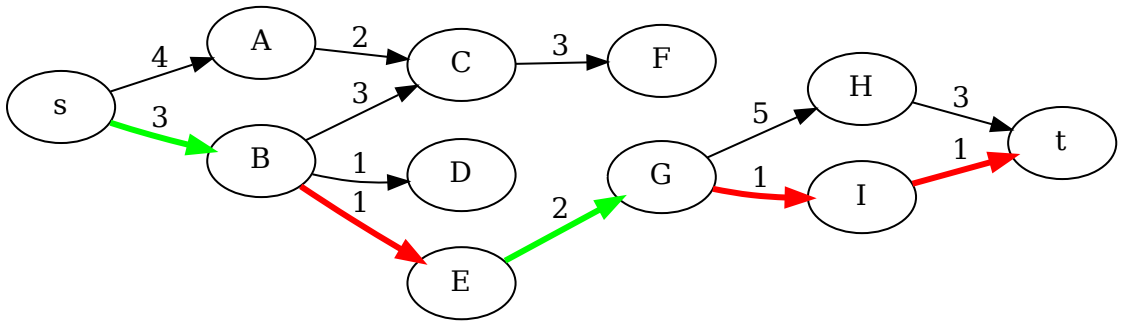
. . .


s
A
B
C
D
E
F
G
H
I
t
4
3
2
3
1
1
3
2
5
1
3
1

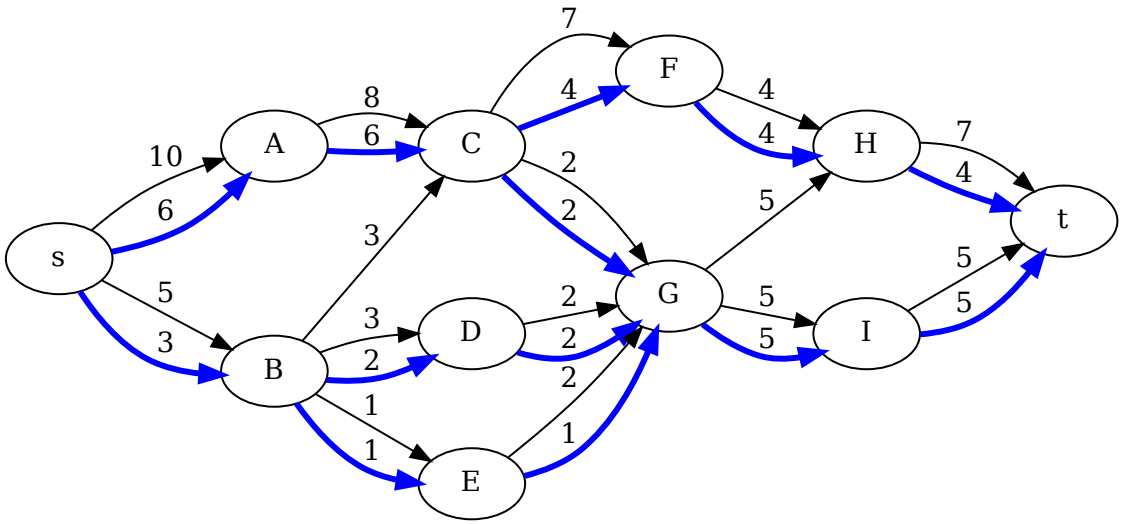
Example


s
A
B
C
D
E
F
G
H
I
t
10
6
5
2
8
6
3
3
2
1
7
4
2
2
2
2
2
4
4
4
5
5
4
7
4
5
4

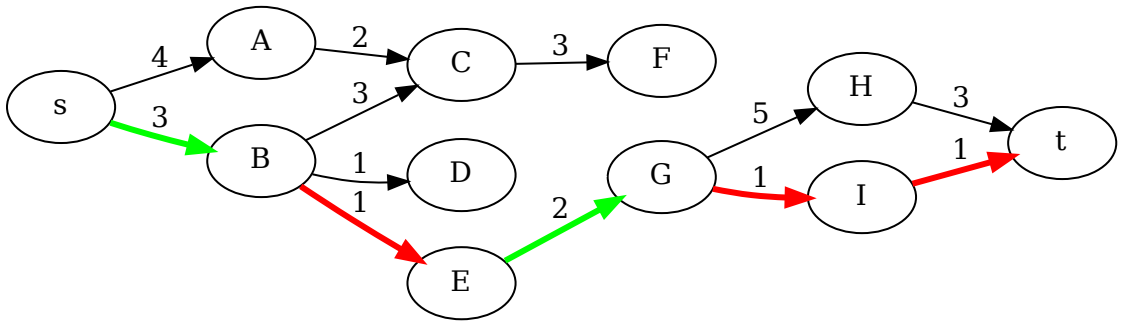
. . .


s
A
B
C
D
E
F
G
H
I
t
4
3
2
3
1
1
3
2
5
1
1
3
1

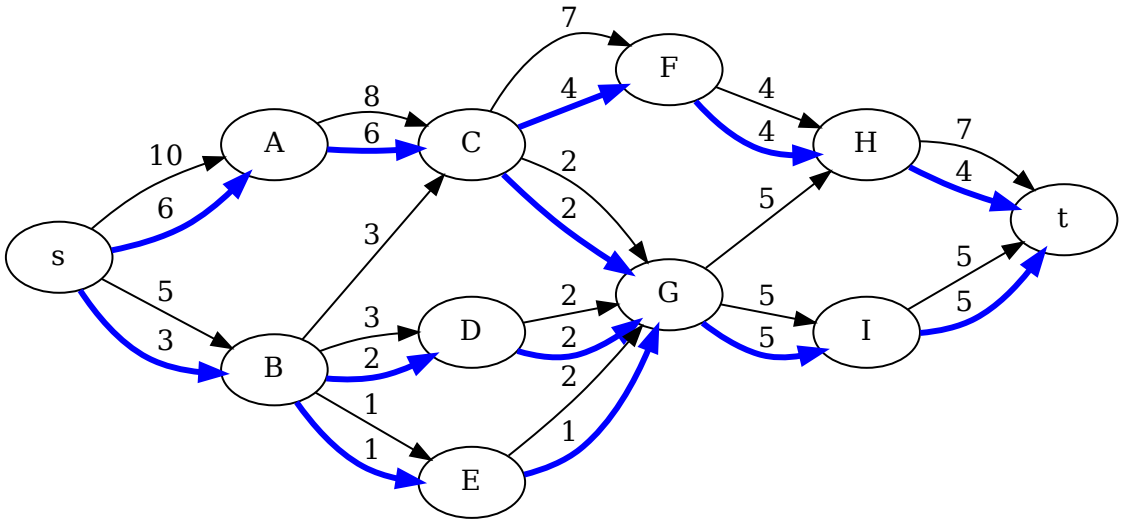
Example

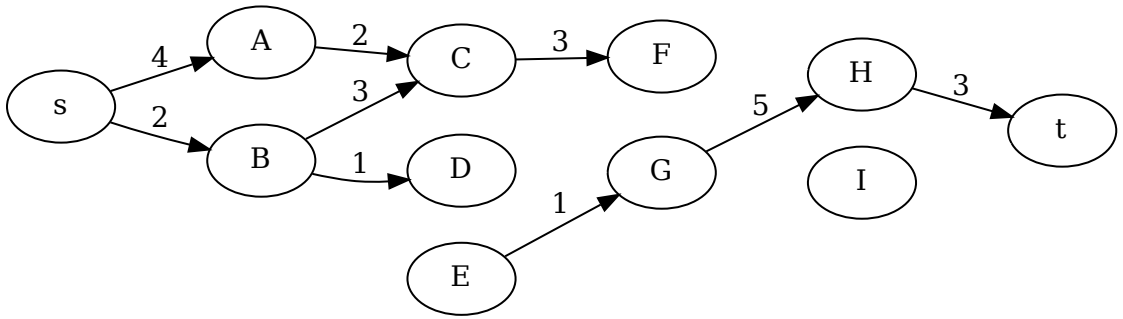

s
A
B
C
D
E
F
G
H
I
t
10
6
5
3
8
6
3
3
2
1
1
7
4
2
2
2
2
2
1
4
4
5
5
5
7
4
5
5

. . .


s
A
B
C
D
E
F
G
H
I
t
4
3
2
3
1
1
3
2
1
5
1
3
1

Example



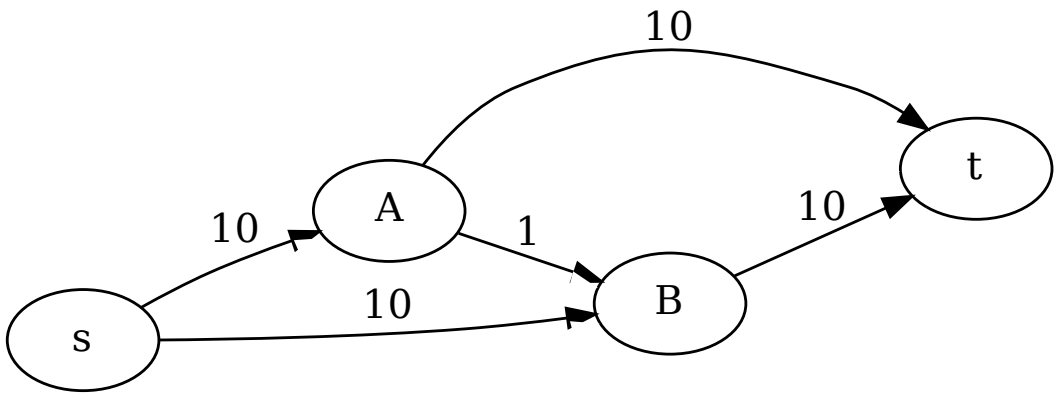


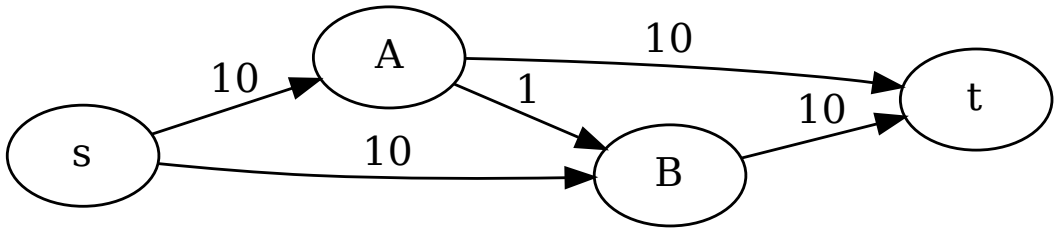
No path exists
$\Rightarrow$ current flow
is max

How to update capacities

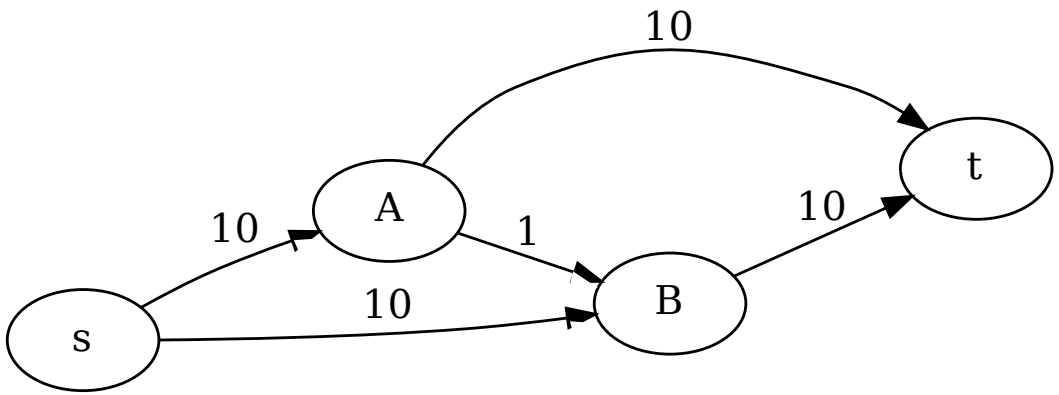
- Each time we find a path we subtract its flow from the capacities of used arcs (good!).
- However, the algorithm as currently formulated is **not** correct!
  - Intuitive reason: once we commit to a path, we never decrease the flow along its arcs.
  - However, we don't know beforehand if the optimal solution should saturate the path.
- To fix this, we need to update capacities in a way that allows us to later on decrease the flow along a used arc, as needed.

Counter-example

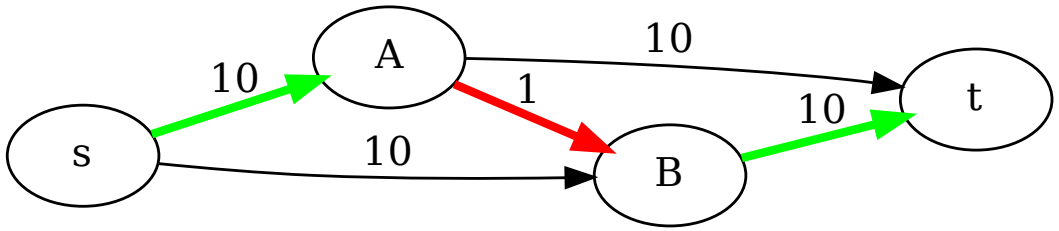




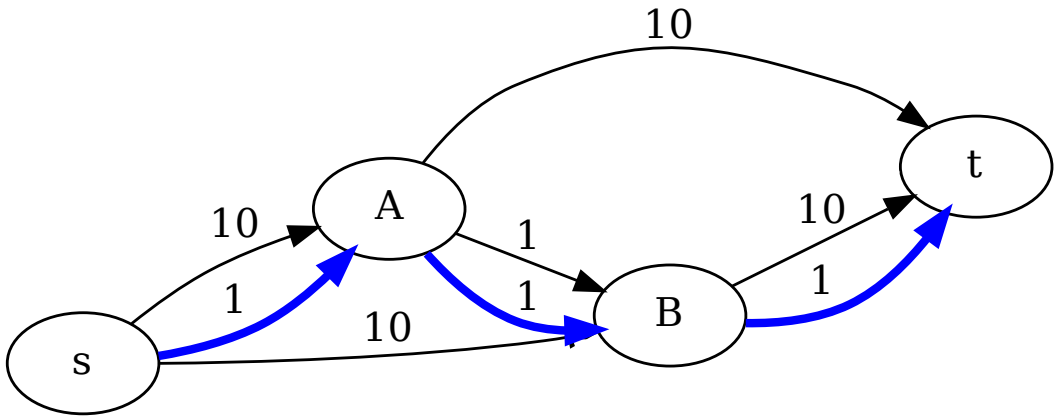
Counter-example



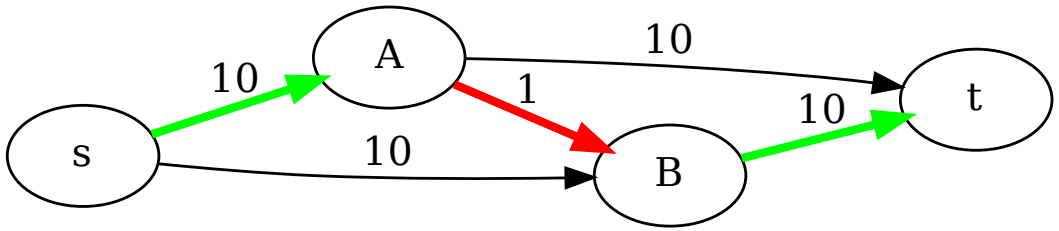
Find an $s \to t$ path.
Min capacity:1



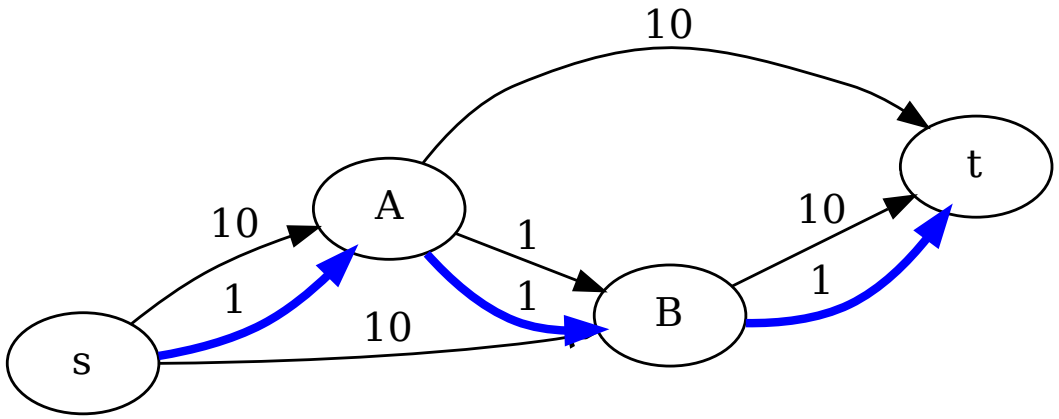
Counter-example

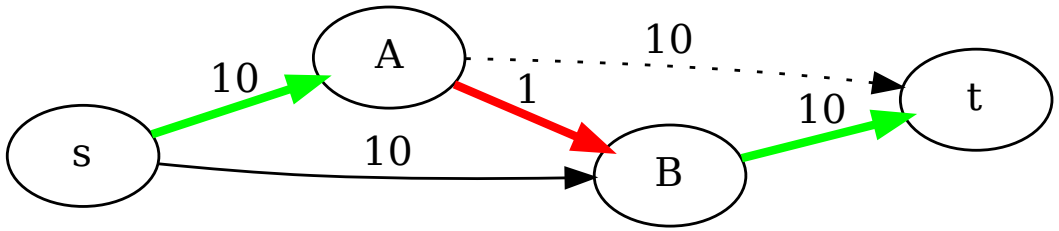


Saturate the path



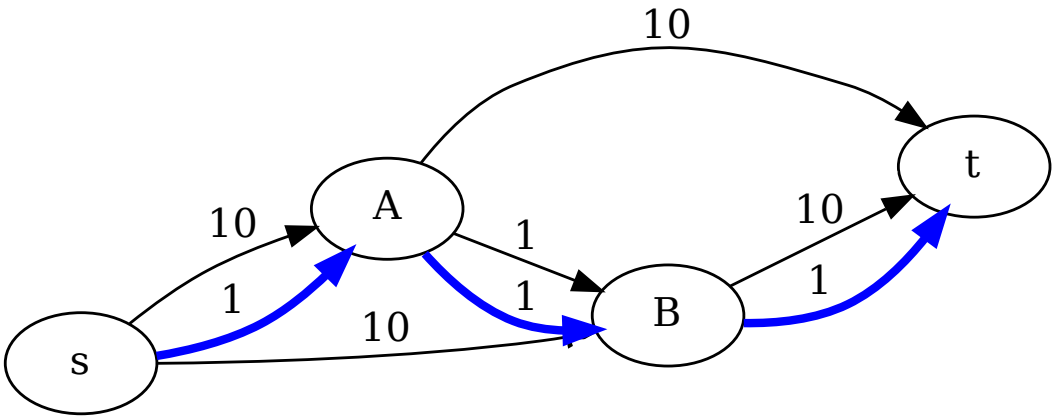
Counter-example



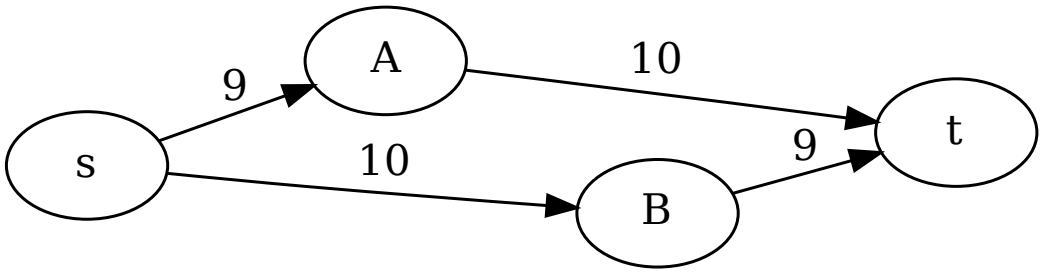


Note: this path looks stupid, but may in fact be the shortest path (e.g. if we subdivide At many times)

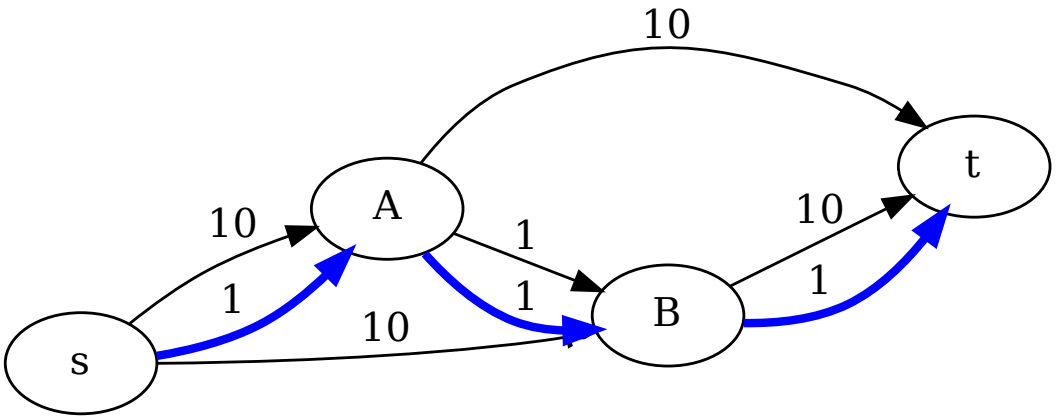
Counter-example

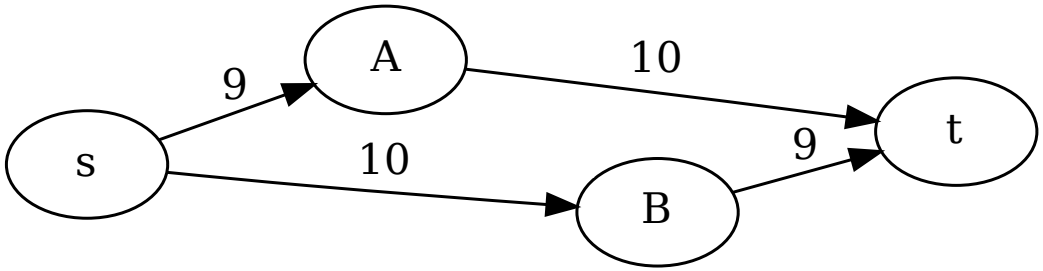


Update Capacities



Counter-example





Now we have a problem!
New graph has max-flow=18, so in total we get a flow of 19.
Max=20 (!!)

Residual Networks

Definition

Let $G = (V, A), s, t \in V, \mathbf{c} : A \to \mathbb{Q}^+$ be a Max-Flow instance, $\mathbf{f} : A \to \mathbb{Q}^+$ a valid flow on G. The residual network $G = (V, A), \mathbf{c}' : A \to \mathbb{Q}^+$ is defined as follows:

- For each $uv \in A$ we set
  1. $\mathbf{c}'(uv) = \mathbf{c}(uv) - \mathbf{f}(uv)$
  2. $\mathbf{c}'(vu) = \mathbf{c}(uv) + \mathbf{f}(uv)$

Residual Networks

Definition

Let $G = (V, A), s, t \in V, \mathbf{c} : A \to \mathbb{Q}^+$ be a Max-Flow instance, $\mathbf{f} : A \to \mathbb{Q}^+$ a valid flow on G. The residual network $G = (V, A), \mathbf{c}' : A \to \mathbb{Q}^+$ is defined as follows:

- For each $uv \in A$ we set
  1. $\mathbf{c}'(uv) = \mathbf{c}(uv) - \mathbf{f}(uv)$
  2. $\mathbf{c}'(vu) = \mathbf{c}(uv) + \mathbf{f}(uv)$

(To ease notation, assume that when $ij \in A$, then $ji \in A$, possibly with capacity 0)

Residual Network: intuition

- We currently have a flow $\mathbf{f}$.
- If $\mathbf{f}(uv) > 0$ we decrease $\mathbf{c}'(uv)$ by $\mathbf{f}(uv)$
  - Makes sense, as we have used some of the capacity of uv.

Residual Network: intuition

- We currently have a flow $\mathbf{f}$.
- If $\mathbf{f}(uv) > 0$ we decrease $\mathbf{c}'(uv)$ by $\mathbf{f}(uv)$
  - Makes sense, as we have used some of the capacity of uv.
- **Furthermore**, we **increase** $\mathbf{c}'(vu)$ by $\mathbf{f}(uv)$
  - We do this, even if vu previously did not exist (had capacity 0).
  - Intuition, if we later send flow through vu, we can simulate this by decreasing the flow through uv.
  - Increasing $\mathbf{f}(vu)$ by 1 is the same as decreasing $\mathbf{uv}$ by 1 for the flow-conservation constraints.

Residual Network: intuition

- We currently have a flow $\mathbf{f}$.
- If $\mathbf{f}(uv) > 0$ we decrease $\mathbf{c}'(uv)$ by $\mathbf{f}(uv)$
  - Makes sense, as we have used some of the capacity of uv.
- **Furthermore**, we **increase** $\mathbf{c}'(vu)$ by $\mathbf{f}(uv)$
  - We do this, even if vu previously did not exist (had capacity 0).
  - Intuition, if we later send flow through vu, we can simulate this by decreasing the flow through uv.
  - Increasing $\mathbf{f}(vu)$ by 1 is the same as decreasing $\mathbf{uv}$ by 1 for the flow-conservation constraints.
- Thanks to this idea, we don't **commit** when we add a path to the solution (the flow can be reversed later).

Ford-Fulkerson method

1: Initialize flow $\mathbf{f} := 0$, residual graph $G' := G$.
2: **while** G' has an $s \to t$ path **do**
3: Find P, an $s \to t$ path in G' ▷ BFS/DFS
4: Find c_p, min capacity of P in G'
5: For $uv \in P$ set $\mathbf{f}(uv) := \mathbf{f}(uv) + c_p$ ▷ Eliminate cycles!
6: Update residual graph G'
7: **end while**
8: Output $\mathbf{f}$

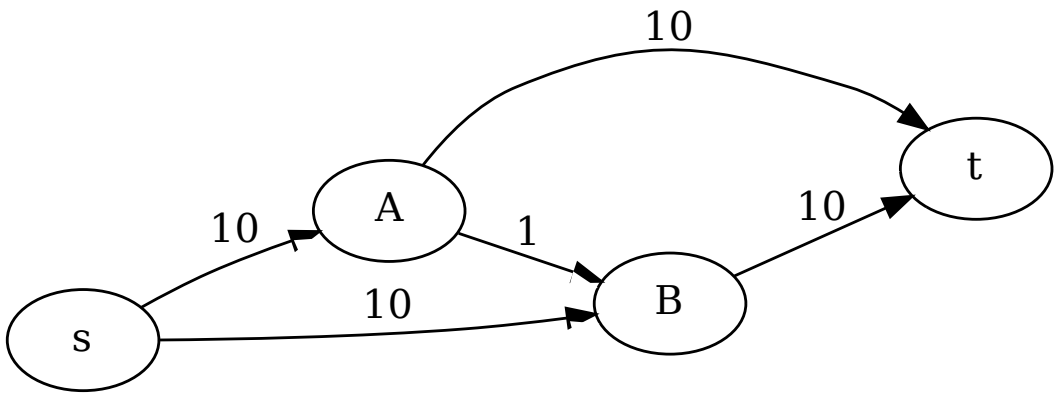
Ford-Fulkerson method

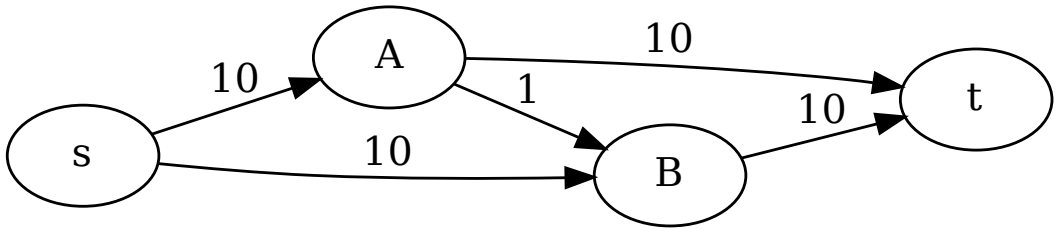
1: Initialize flow $\mathbf{f} := 0$, residual graph $G' := G$.
2: **while** G' has an $s \to t$ path **do**
3: Find P, an $s \to t$ path in G' ▷ BFS/DFS
4: Find c_p, min capacity of P in G'
5: For $uv \in P$ set $\mathbf{f}(uv) := \mathbf{f}(uv) + c_p$ ▷ Eliminate cycles!
6: Update residual graph G'
7: **end while**
8: Output $\mathbf{f}$

Eliminate cycles:

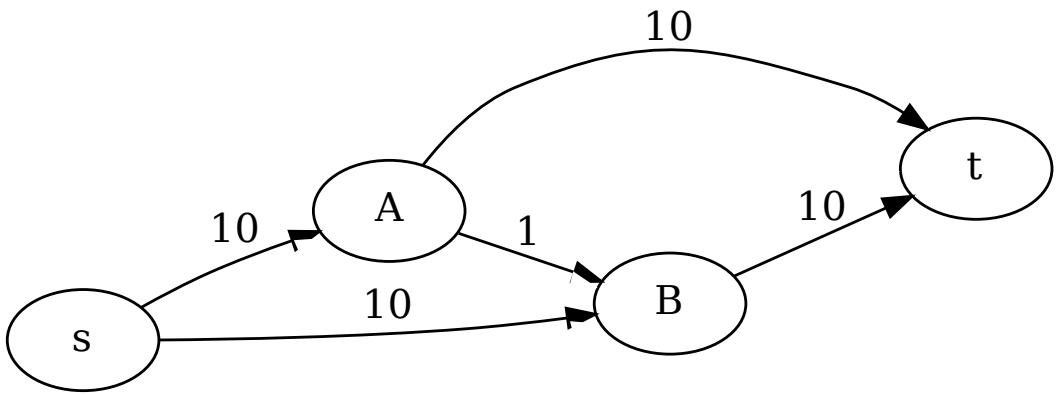
1: **if** $\mathbf{f}(uv) > 0$ and $\mathbf{f}(vu) > 0$ **then**
2: $\mathbf{f}(uv) -= \min\{\mathbf{f}(uv), \mathbf{f}(vu)\}$
3: $\mathbf{f}(vu) -= \min\{\mathbf{f}(uv), \mathbf{f}(vu)\}$
4: **end if**

Counter-example fixed

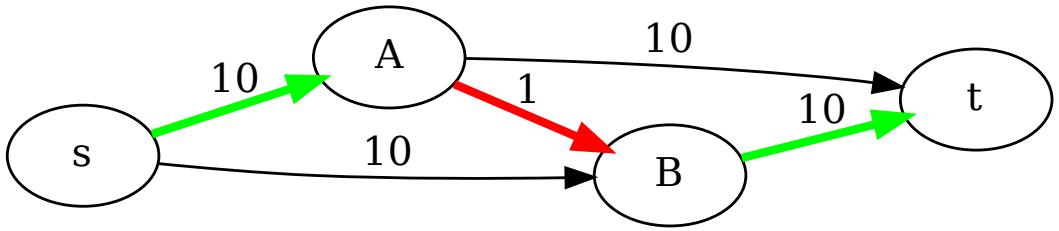




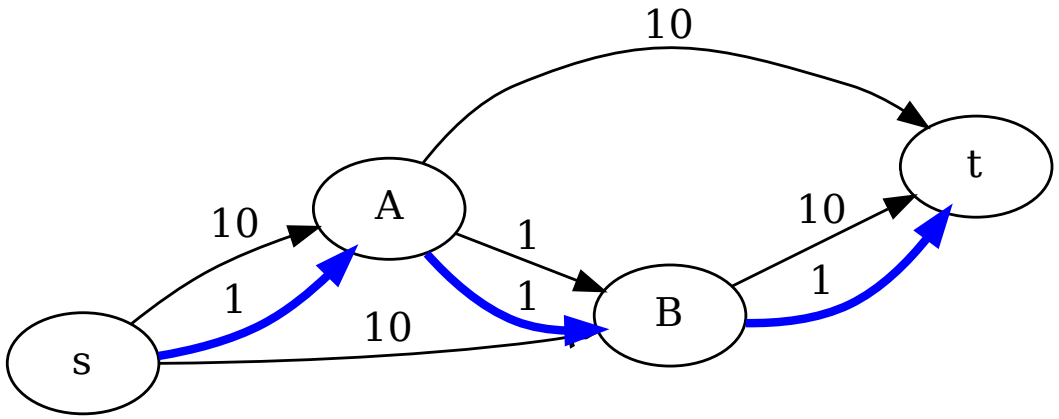
Counter-example fixed



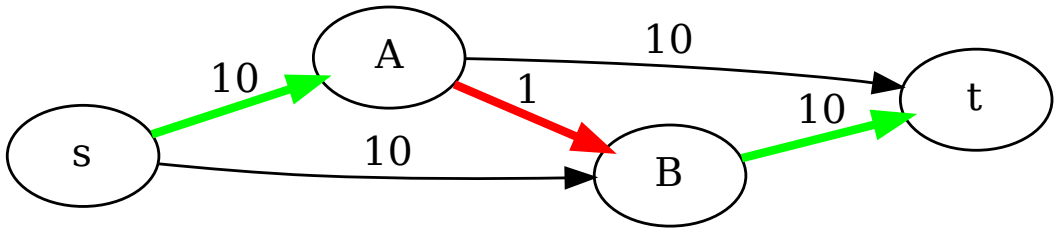
Find an $s \to t$ path.
Min capacity:1



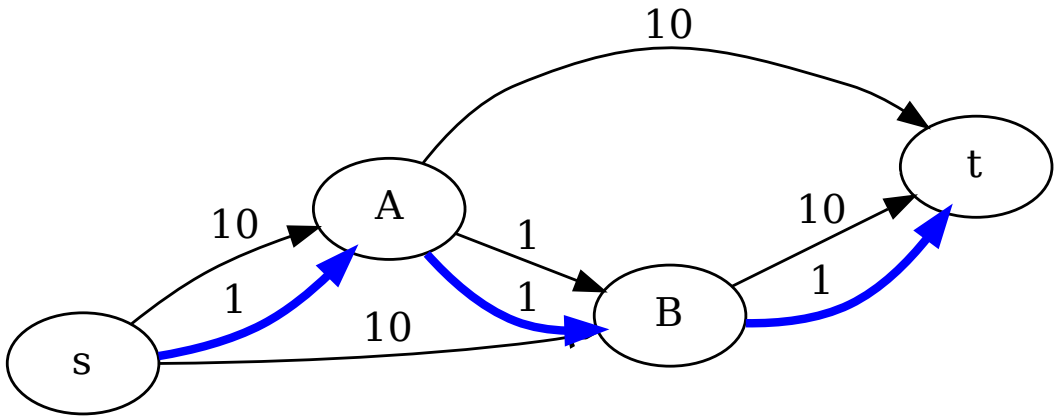
Counter-example fixed



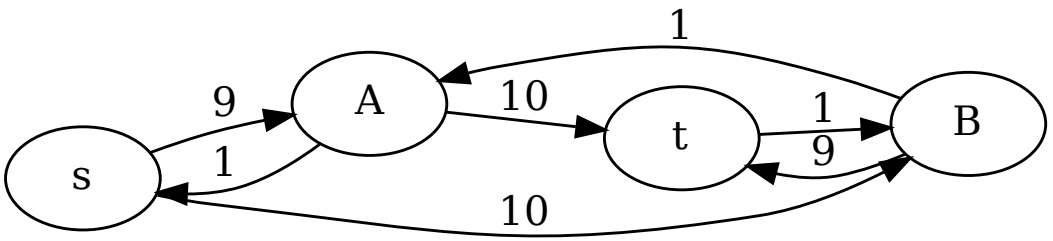
Saturate the path



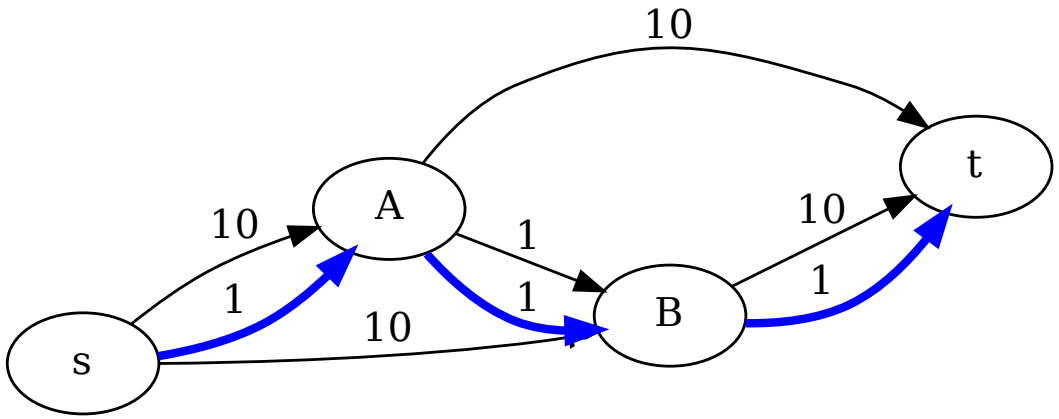
Counter-example fixed

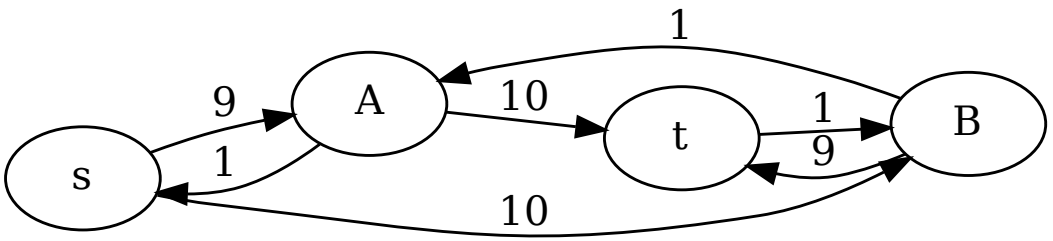


Update Capacities



Counter-example fixed





Now the residual network has a flow of value 19. Together with the path this gives a flow of value 20.

Correctness

Theorem

The Ford-Fulkerson algorithm calculates a maximum flow.

Correctness

Theorem

The Ford-Fulkerson algorithm calculates a maximum flow.

Lemma

The Ford-Fulkerson algorithm always terminates and outputs a valid flow.

Proof.

- Suppose we have a valid current flow **f**
- If the residual network G' has no $s \to t$ path, done.
- Otherwise, we compute a **larger** flow, by adding to **f** the minimum capacity of the computed path.
- New flow is feasible (why?).
- Flow cannot become ∞, so algorithm will eventually terminate.

Correctness

Theorem

The Ford-Fulkerson algorithm calculates a maximum flow.

Correctness

Theorem

The Ford-Fulkerson algorithm calculates a maximum flow.

Proof.

- Let $\mathbf{f}$ be the output valid flow of value f^*, G' the residual network where no $s \to t$ path exists.
- Let S be the set of vertices reachable from s in G'.
  - $\Rightarrow$ no arc comes out of S in G'.
- Claim: S gives a cut of weight f^*
  - Arcs coming into S have 0 flow (otherwise G' has outgoing arc from S)
  - Arcs coming out of S are removed in G'
  - $\Rightarrow$ for such arcs uv we have $\mathbf{f}(uv) = \mathbf{c}(uv)$
  - $f^* = \sum_{u \in S} \sum_{v \notin S} \mathbf{f}(uv) = \sum_{u \in S} \sum_{v \notin S} \mathbf{c}(uv)$
- $\Rightarrow$ no flow can have value higher than f^*, so current flow is maximum.

Consequences

Theorem

In any weighted digraph $G = (V, A)$, for any $s, t \in V$, the max-flow value from s to t is **equal** *to the min-cut value from s to t.*

Consequences

Theorem

In any weighted digraph $G = (V, A)$, for any $s, t \in V$, the max-flow value from s to t is **equal** *to the min-cut value from s to t.*

Proof.

- Max-Flow$\leq$Min-Cut is easy (as we saw).
- Ford-Fulkerson algorithm produces a flow **equal** to some cut, so the flow is maximum, the cut is minimum.

Consequences

Theorem

In any weighted digraph $G = (V, A)$, for any $s, t \in V$, the max-flow value from s to t is **equal** *to the min-cut value from s to t.*

Proof.

- Max-Flow$\leq$Min-Cut is easy (as we saw).
- Ford-Fulkerson algorithm produces a flow **equal** to some cut, so the flow is maximum, the cut is minimum.

□

- **Rare** duality theorem, considered one of the deepest in OR.
- For most optimization problems, no such duality is known.
  - Classes: NP vs coNP, P

Consequences

Corollary

If all capacities are integral, there exists an integral maximum flow.

Consequences

Corollary

If all capacities are integral, there exists an integral maximum flow.

- Statement is obvious for min-cuts.
- Non-obvious for flows, as we can partition flows into fractional pieces (this is allowed by the problem definition).

Complexity Analysis

1: Initialize flow $\mathbf{f} := 0$, residual graph $G' := G$.
2: **while** G' has an $s \to t$ path **do** ▷ How many iterations?
3: Find P, an $s \to t$ path in G' ▷ BFS/DFS: $O(m)$ time
4: Find c_p, min capacity of P in G' ▷ $O(n)$ time
5: For $uv \in P$ set $\mathbf{f}(uv) := \mathbf{f}(uv) + c_p$ ▷ $O(m)$ time
6: Update residual graph G'
7: **end while**
8: Output $\mathbf{f}$

- Each iteration takes $O(m)$ time, but how many times do we need to execute the main loop?
- Easy argument: each execution increases flow value, so if capacities are integer, complexity is $O(m \cdot f^*)$.

Complexity Analysis

Theorem

Ford-Fulkerson method runs in time $O(m \cdot f^*)$ and there exist instances on which it takes $\Omega(m \cdot f^*)$ time.

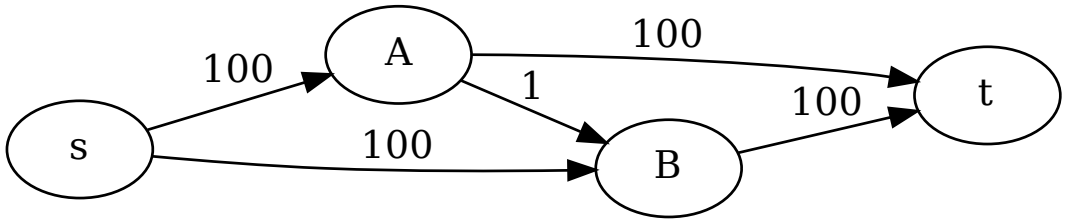
Complexity Analysis

Theorem

Ford-Fulkerson method runs in time $O(m \cdot f^*)$ and there exist instances on which it takes $\Omega(m \cdot f^*)$ time.

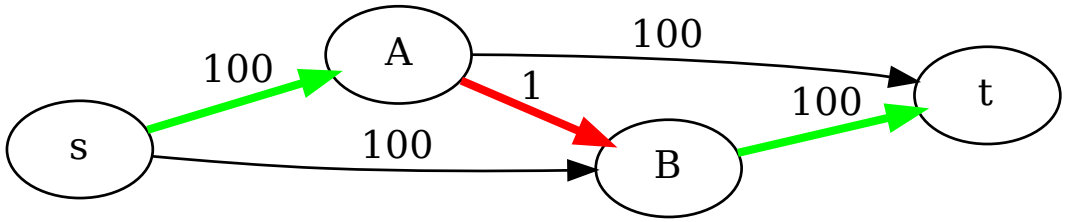
- **NB**: this means that FF method is an **exponential**-time algorithm!
  - More precisely, a **pseudopolynomial**-time algorithm.
  - **Pseudopolynomial**: polynomial-time if input numbers have $O(\log n)$ bits.
- Problem: we haven't specified **which** $s \to t$ path to pick in the residual graph.
- If bad paths are repeatedly picked, FF method takes a very long time...
- Generic method can still be useful if we know that f^* is small.

Bad Example



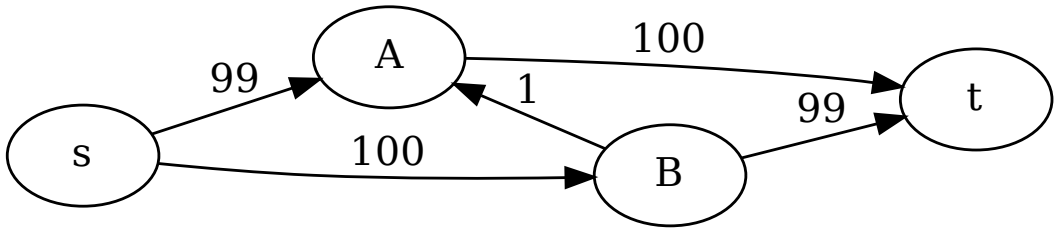
$f = 0$

Bad Example



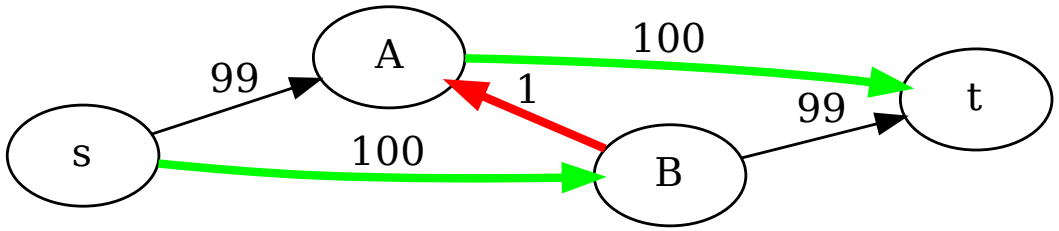
$f = 0$

Bad Example



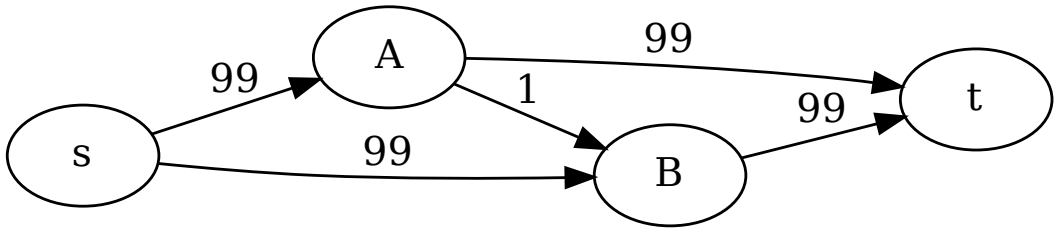
$f = 1$

Bad Example



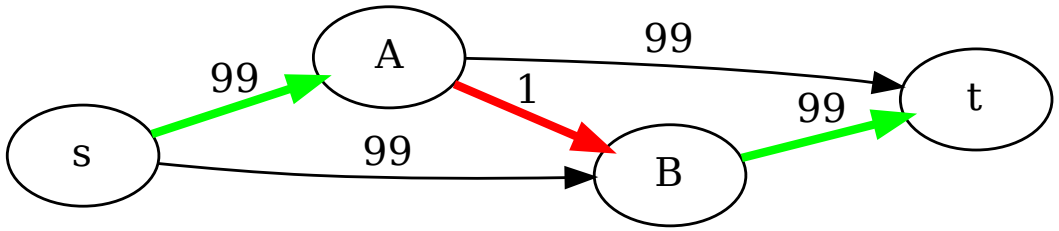
$f = 1$

Bad Example



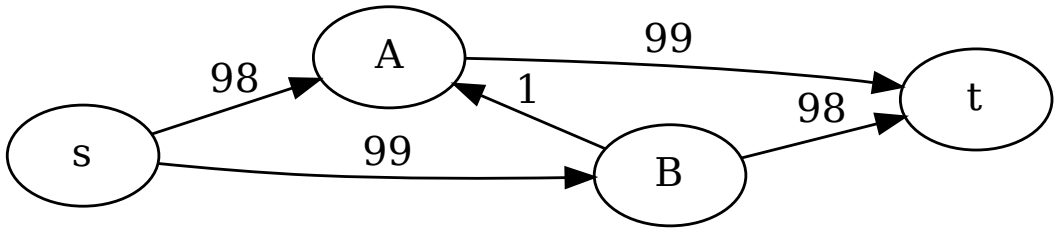
$f = 2$

Bad Example



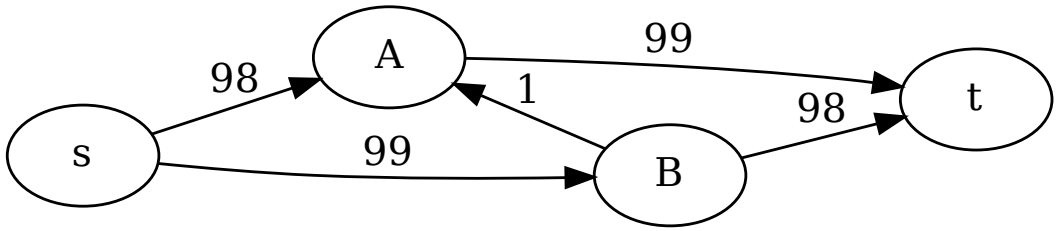
$f = 2$

Bad Example



$f = 3\ldots$

Bad Example



$f = 3\ldots$

Would you like to repeat this another 197 times?

Edmonds-Karp algorithm

Outline

- Take Ford-Fulkerson method but specify **which** path to select in each iteration.
- Natural choice: take the **shortest** path
  - **NB:** shortest in the sense of minimum number of arcs, disregarding weights.
- Key idea: shortest-path distances from s (in unweighted sense) can only increase for each iteration
- $\Rightarrow$ we can bound the number of iterations as a function of n, independently of f^*.

Analysis

Theorem

Edmonds-Karp algorithm runs in time $O(nm^2)$.

Analysis

Theorem

Edmonds-Karp algorithm runs in time $O(nm^2)$.

Lemma

For all v, distance from s to v never decreases in the residual network.

Lemma

Every arc $ij \in A$ is saturated at most $\frac{n}{2}$ times.

Lemma

For all v, distance from s to v never decreases in the residual network.

Lemma

For all v, distance from s to v never decreases in the residual network.

Proof.

- Let v be the first and closest to s vertex for which distance decreased between two iterations.
- Let G_1, G_2 be the two residual networks and $\mathrm{dist}_{G_2}(s, v) < \mathrm{dist}_{G_1}(s, v)$.
- Let u be the last vertex of a shortest $s \to v$ path in G_2.
- Claim: $uv \notin G_1$

Lemma

For all v, distance from s to v never decreases in the residual network.

Proof.

- Let v be the first and closest to s vertex for which distance decreased between two iterations.
- Let G_1, G_2 be the two residual networks and $\mathrm{dist}_{G_2}(s, v) < \mathrm{dist}_{G_1}(s, v)$.
- Let u be the last vertex of a shortest $s \to v$ path in G_2.
- Claim: $uv \notin G_1$
  - Suppose for contradiction $uv \in G_1$:
  - $\mathrm{dist}_{G_1}(s, v) \leq \mathrm{dist}_{G_1}(s, u) + 1 \leq \mathrm{dist}_{G_2}(s, u) + 1 = \mathrm{dist}_{G_2}(s, v)$
  - So, distance to v actually did not decrease...

Lemma

For all v, distance from s to v never decreases in the residual network.

Proof.

(Continued...)

- $uv \notin G_1$ and $uv \in G_2$
- $\Rightarrow s \to t$ path in G_1 uses vu
- $\mathrm{dist}_{G_1}(s, v) = \mathrm{dist}_{G_1}(s, u) - 1 \leq \mathrm{dist}_{G_2}(s, u) - 1 = \mathrm{dist}_{G_2}(s, v) - 2$
- So, again distance to v increased, contradiction!

Lemma

Every arc $ij \in A$ is saturated at most $\frac{n}{2}$ times.

Lemma

Every arc $ij \in A$ is saturated at most $\frac{n}{2}$ times.

Proof.

- Suppose $\mathrm{dist}_{G_1}(s, i) = d$ when ij is saturated.
- Later, ji is used in some residual network G_2.
- We claim, next time ij is saturated, $\mathrm{dist}_{G_3}(s, i) \geq d + 2$.

Lemma

Every arc $ij \in A$ is saturated at most $\frac{n}{2}$ times.

Proof.

- Suppose $\text{dist}_{G_1}(s, i) = d$ when ij is saturated.
- Later, ji is used in some residual network G_2.
- We claim, next time ij is saturated, $\text{dist}_{G_3}(s, i) \geq d + 2$.
- When ij first saturated, $\text{dist}_{G_1}(s, j) = \text{dist}_{G_1}(s, i) + 1 = d + 1$.
- Later, ji is used, so $\text{dist}_{G_2}(s, i) = \text{dist}_{G_2}(s, j) + 1 \geq d + 2$.
- $\text{dist}_{G_3}(s, i) \geq \text{dist}_{G_2}(s, i)$.

Theorem

Edmonds-Karp algorithm runs in time $O(nm^2)$.

Theorem

Edmonds-Karp algorithm runs in time $O(nm^2)$.

Proof.

- Each iteration takes $O(m)$ time and saturates an edge of the residual network.
- There are m edges, each saturated at most $O(n)$ times.

□

Theorem

Edmonds-Karp algorithm runs in time $O(nm^2)$.

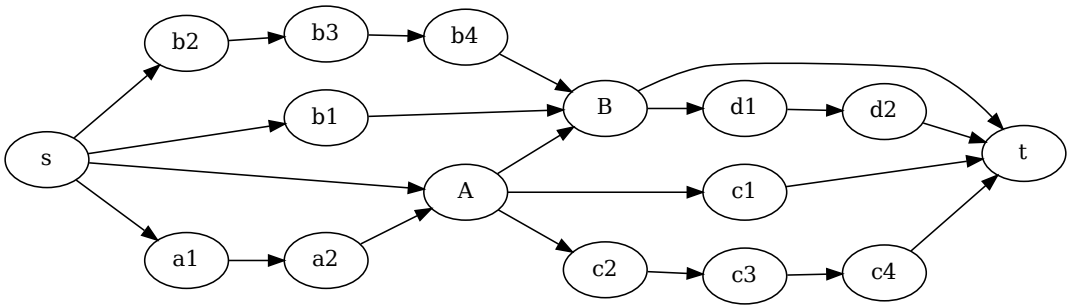
Proof.

- Each iteration takes $O(m)$ time and saturates an edge of the residual network.
- There are m edges, each saturated at most $O(n)$ times.

□

- Running time can be improved with better analysis.
- However, bad examples can force the algorithm to flip-flop an arc many times (see next slide).
- Best known algorithm: $m^{1+o(1)}$ time (almost-linear), with completely different techniques.

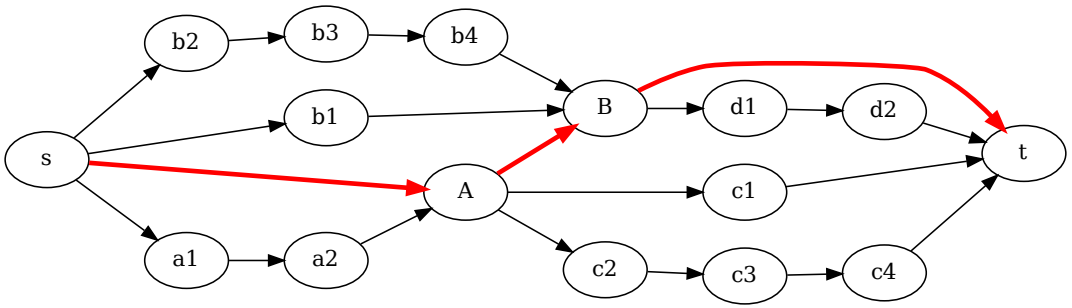
Bad Example



- Capacity 1 everywhere
- $s \to A$ paths: 1,3
- $s \to B$ paths: 2,4
- $A \to t$ paths: 2,4
- $B \to t$ paths: 1,3

Goal: make AB arc flip many times for Edmonds-Karp.

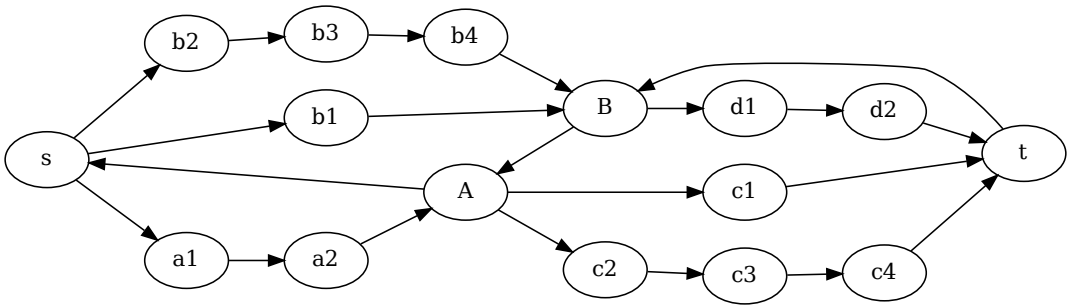
Bad Example



- Capacity 1 everywhere
- $s \to A$ paths: 1,3
- $s \to B$ paths: 2,4
- $A \to t$ paths: 2,4
- $B \to t$ paths: 1,3

Goal: make AB arc flip many times for Edmonds-Karp.

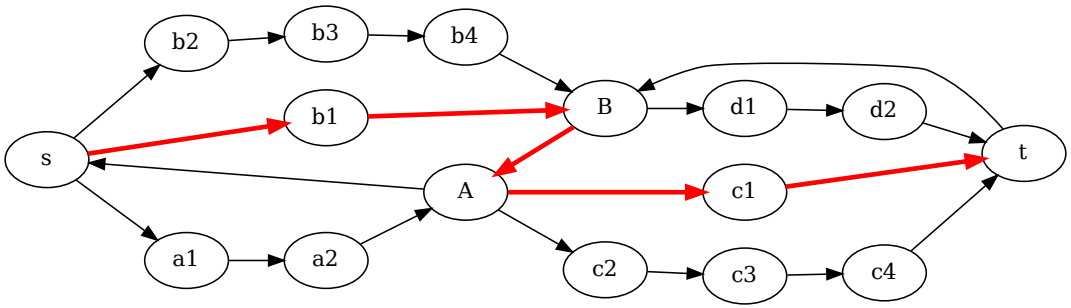
Bad Example



- Capacity 1 everywhere
- $s \to A$ paths: 1,3
- $s \to B$ paths: 2,4
- $A \to t$ paths: 2,4
- $B \to t$ paths: 1,3

Goal: make AB arc flip many times for Edmonds-Karp.

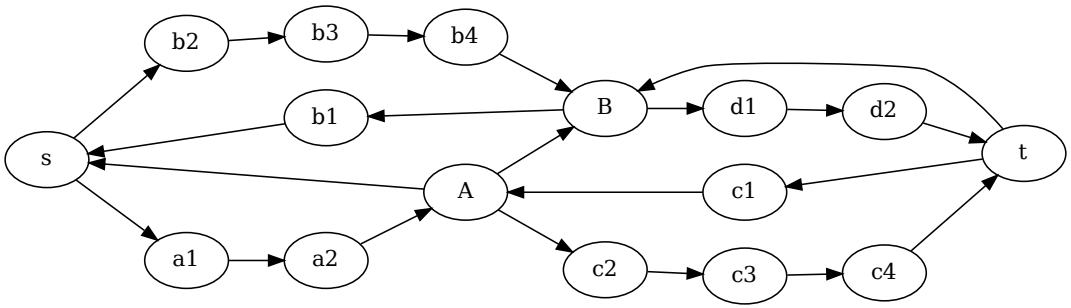
Bad Example



- Capacity 1 everywhere
- $s \to A$ paths: 1,3
- $s \to B$ paths: 2,4
- $A \to t$ paths: 2,4
- $B \to t$ paths: 1,3

Goal: make AB arc flip many times for Edmonds-Karp.

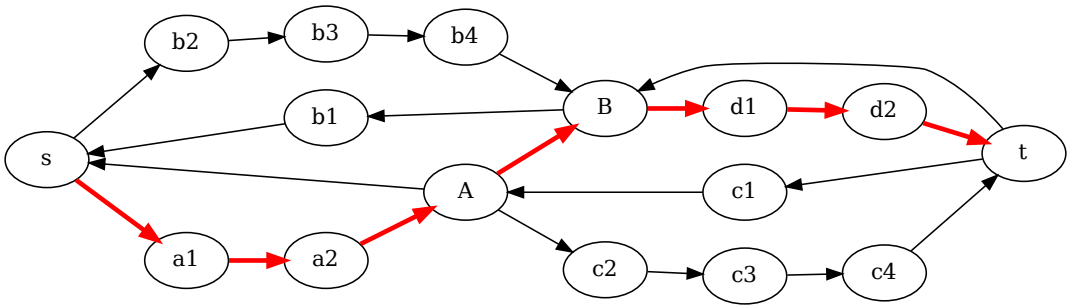
Bad Example



- Capacity 1 everywhere
- $s \to A$ paths: 1,3
- $s \to B$ paths: 2,4
- $A \to t$ paths: 2,4
- $B \to t$ paths: 1,3

Goal: make AB arc flip many times for Edmonds-Karp.

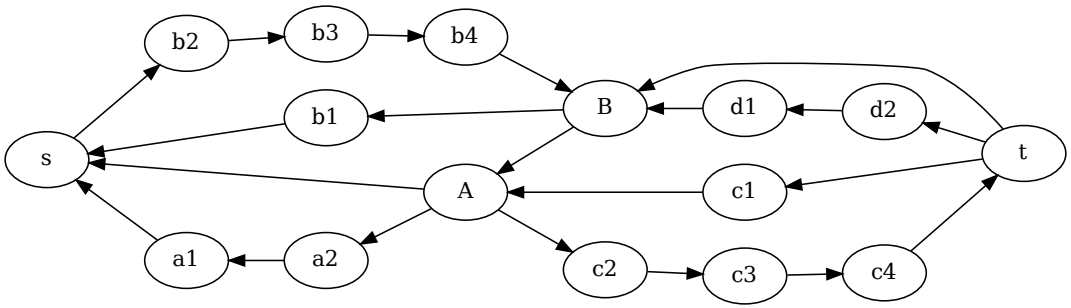
Bad Example



- Capacity 1 everywhere
- $s \to A$ paths: 1,3
- $s \to B$ paths: 2,4
- $A \to t$ paths: 2,4
- $B \to t$ paths: 1,3

Goal: make AB arc flip many times for Edmonds-Karp.

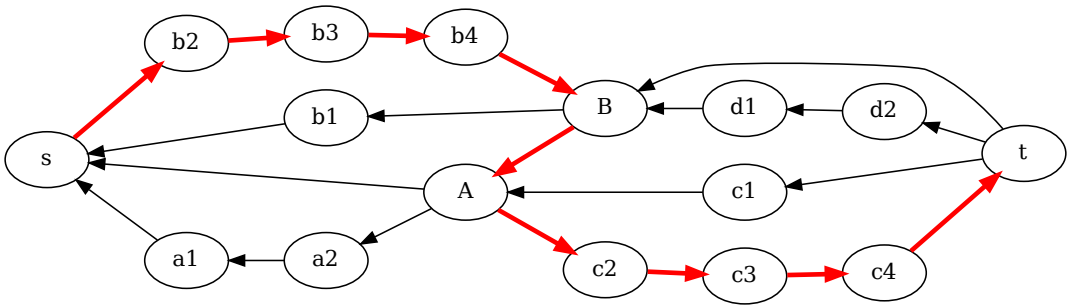
Bad Example



- Capacity 1 everywhere
- $s \to A$ paths: 1,3
- $s \to B$ paths: 2,4
- $A \to t$ paths: 2,4
- $B \to t$ paths: 1,3

Goal: make AB arc flip many times for Edmonds-Karp.

Bad Example



- Capacity 1 everywhere
- $s \to A$ paths: 1,3
- $s \to B$ paths: 2,4
- $A \to t$ paths: 2,4
- $B \to t$ paths: 1,3

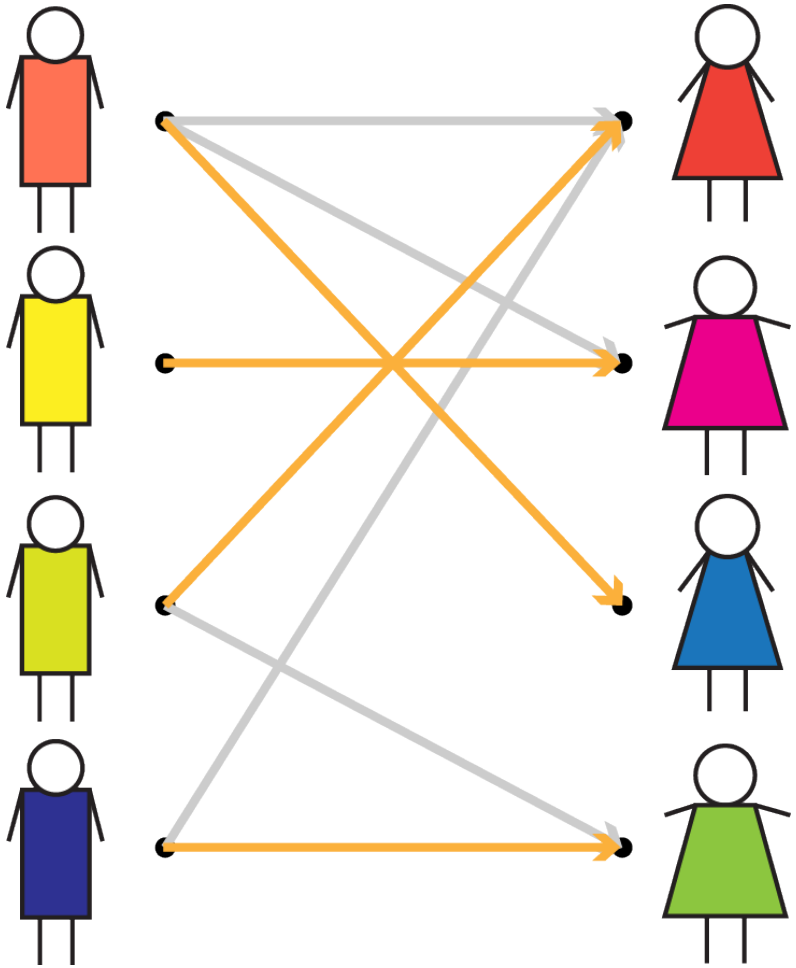
Goal: make AB arc flip many times for Edmonds-Karp.

Bipartite Matching

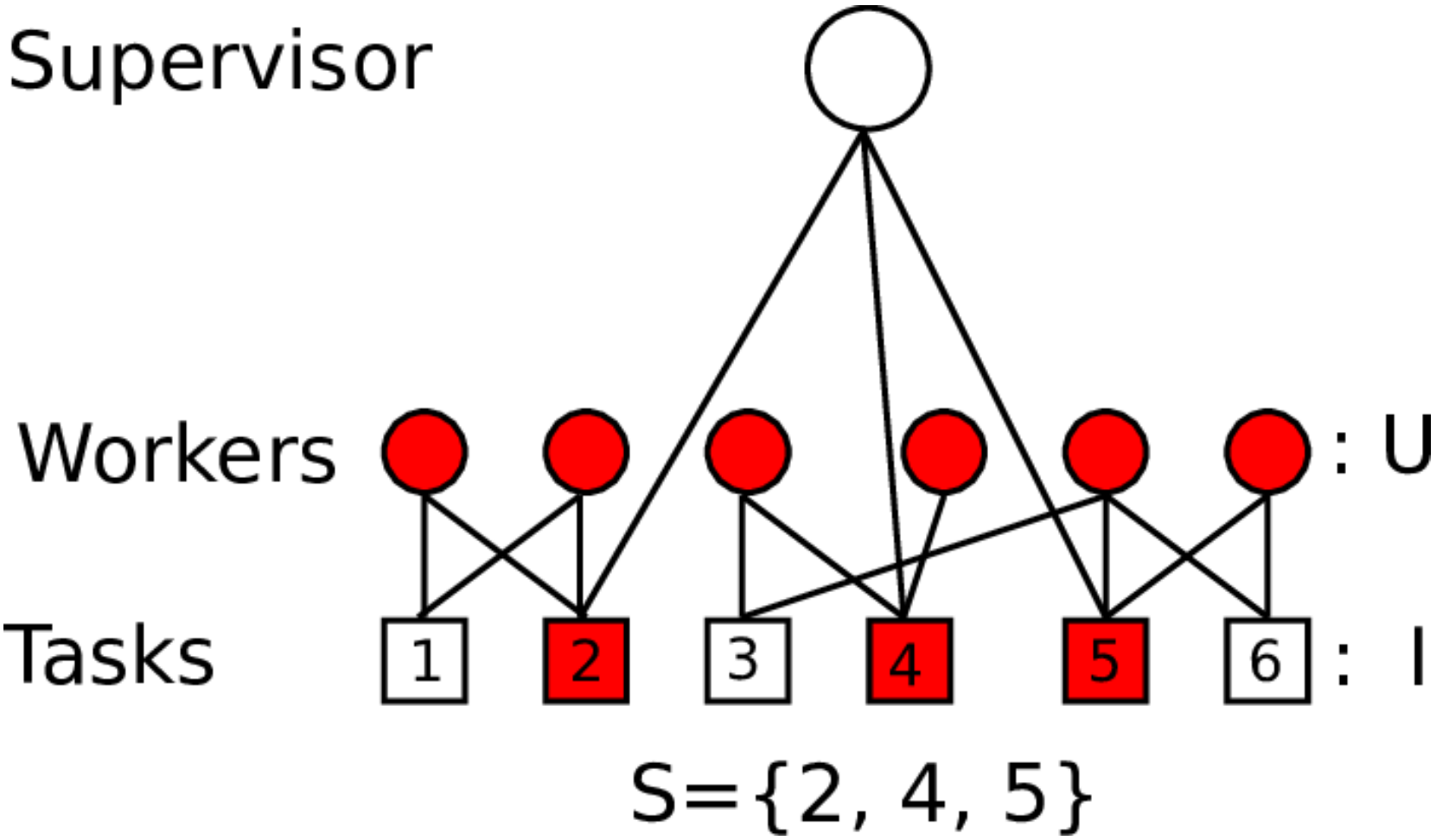
Bipartite Matching

- We are given a graph $G = (V, E)$ with vertices partitioned into two parts A, B.
- **Goal:** We want to match elements of A to elements of B using edges of E.
- Each vertex should be matched to exactly one other vertex.

Motivation



Motivation



Matchings

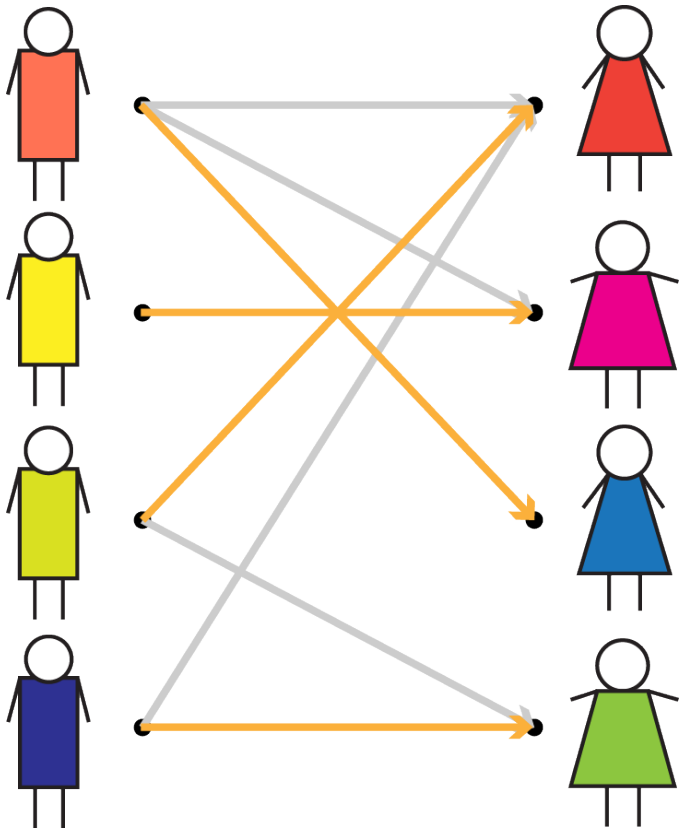
Definition

A **matching** in a graph $G = (V, E)$ is a set $M \subseteq E$ such that no two elements of M share a vertex.

Matchings

Definition

A **matching** in a graph $G = (V, E)$ is a set $M \subseteq E$ such that no two elements of M share a vertex.



Matchings

Definition

A **matching** in a graph $G = (V, E)$ is a set $M \subseteq E$ such that no two elements of M share a vertex.

Definition

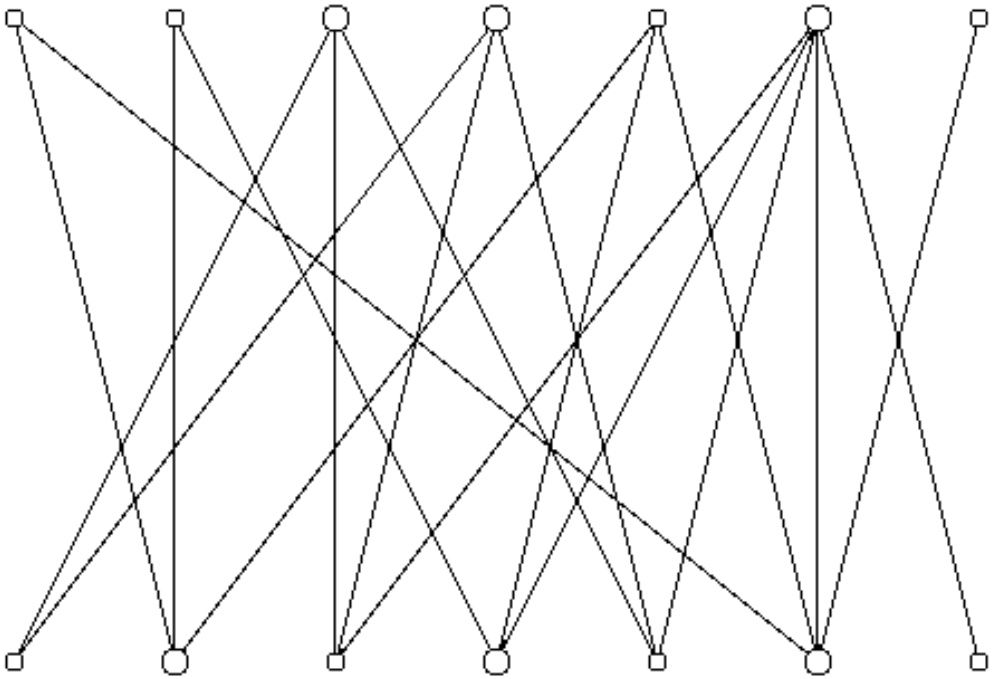
A matching M is **perfect** if all vertices are incident to an edge of M.

Definition

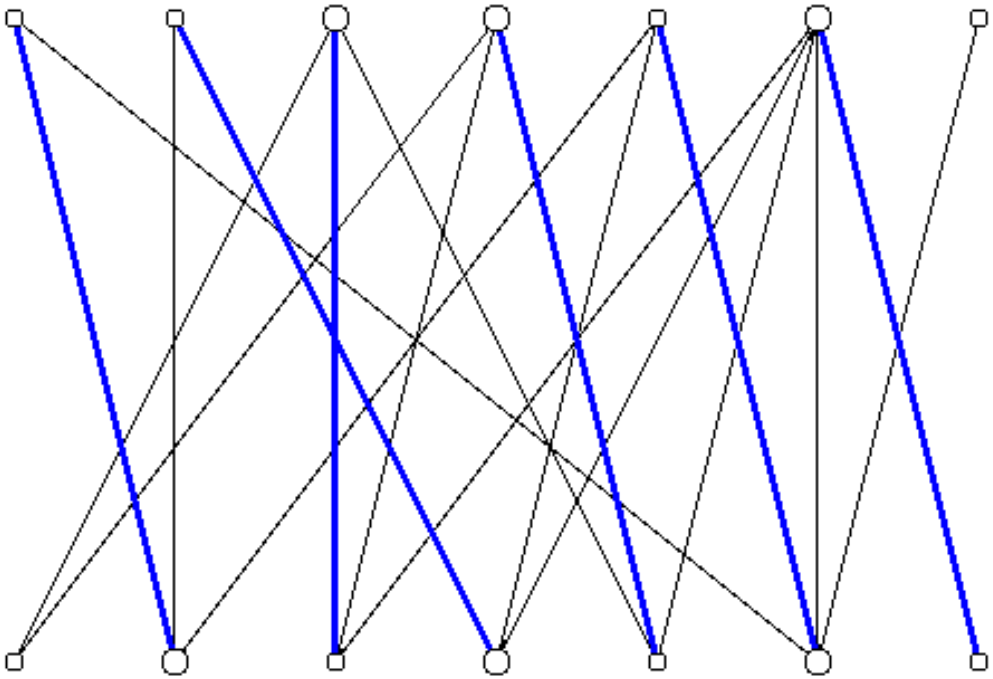
A matching M is **maximum** if all sets of edges of size $|M| + 1$ or more contain two edges incident on the same vertex.

Algorithmic problem: Given a graph, find the maximum matching.

Example



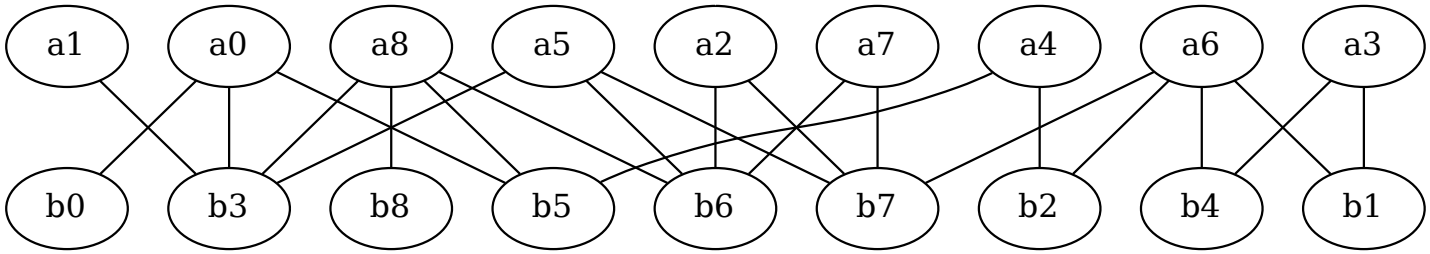
Example



Matchings to Flows

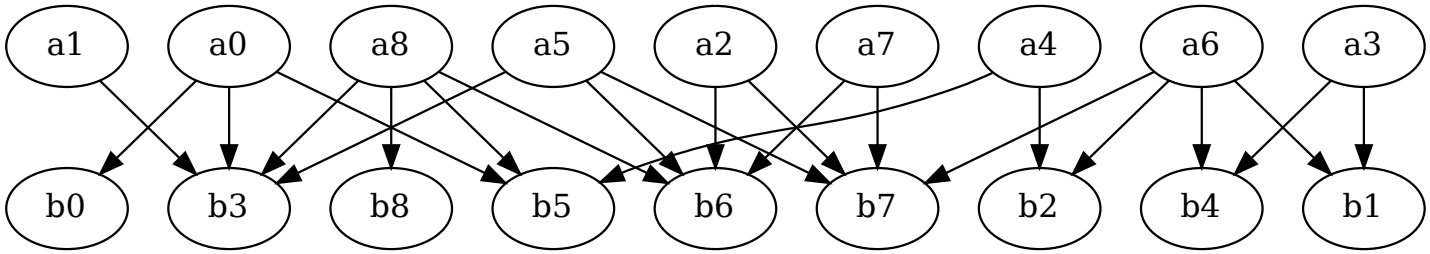
- Finding a maximum matching is a non-trivial problem
- Can be reduced to computing a maximum flow as follows:
  - Orient edges from A to B
  - Add a vertex s with arcs to A
  - Add a vertex t with arcs from B
  - Compute maximum $s \to t$ flow

Example



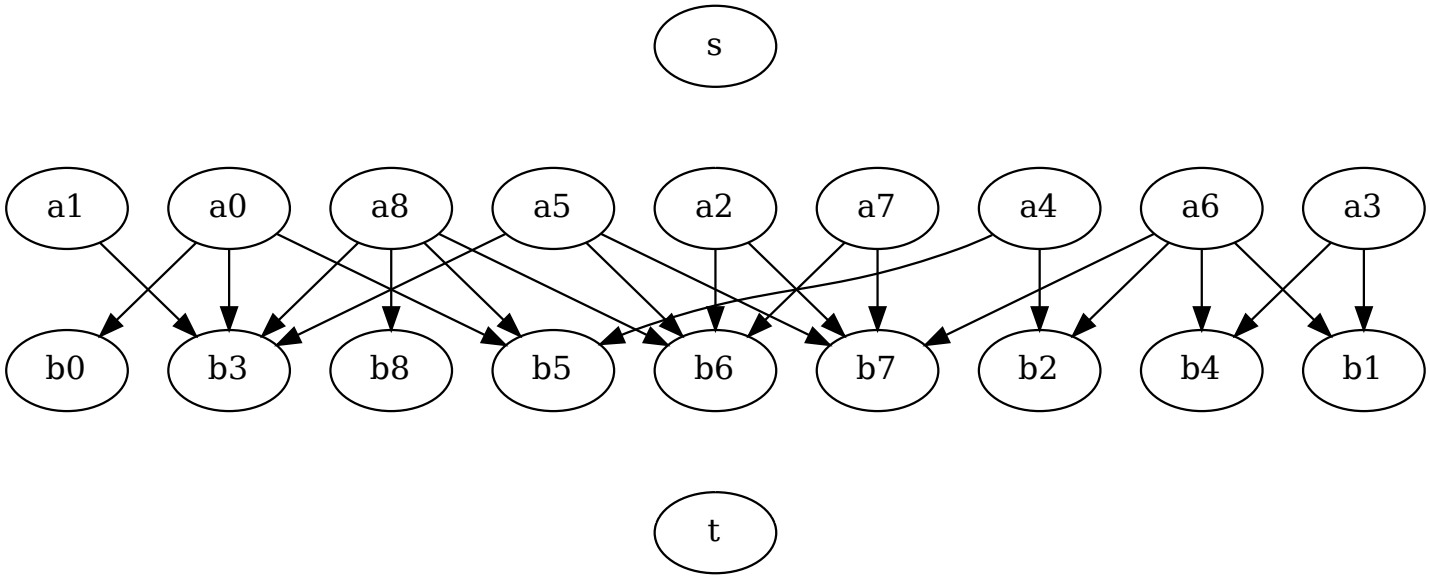
We want to calculate a maximum matching

Example



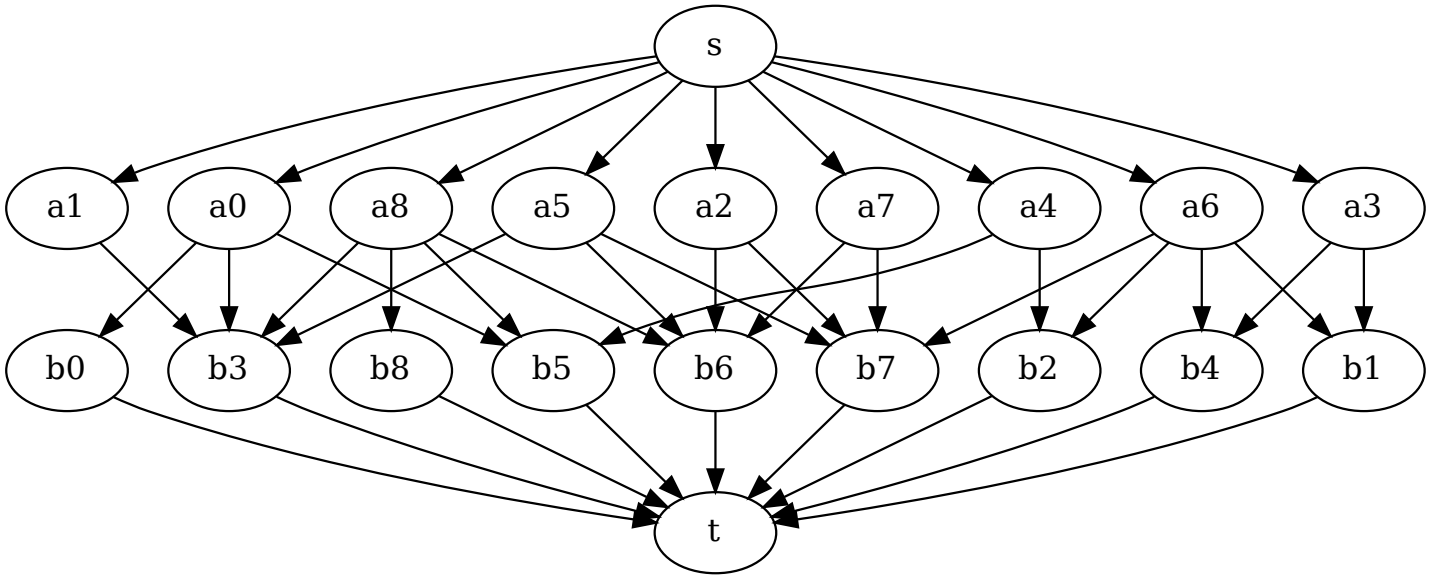
Orient graph, add s, t

Example



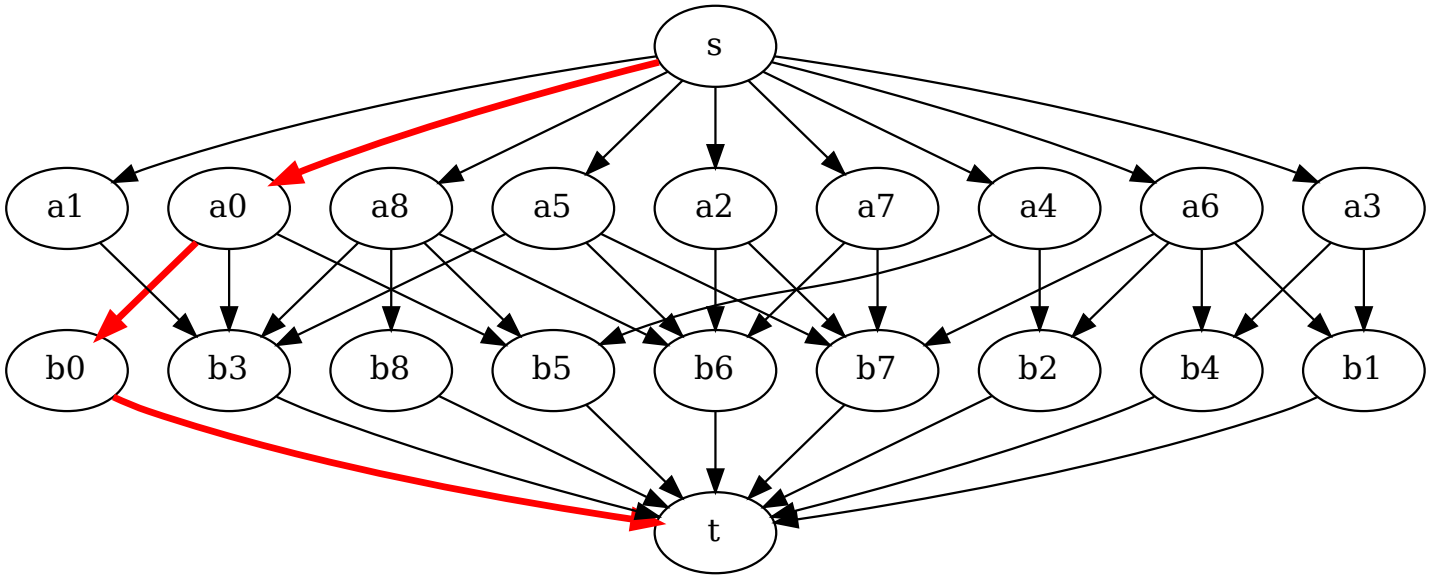
Orient graph, add s, t

Example



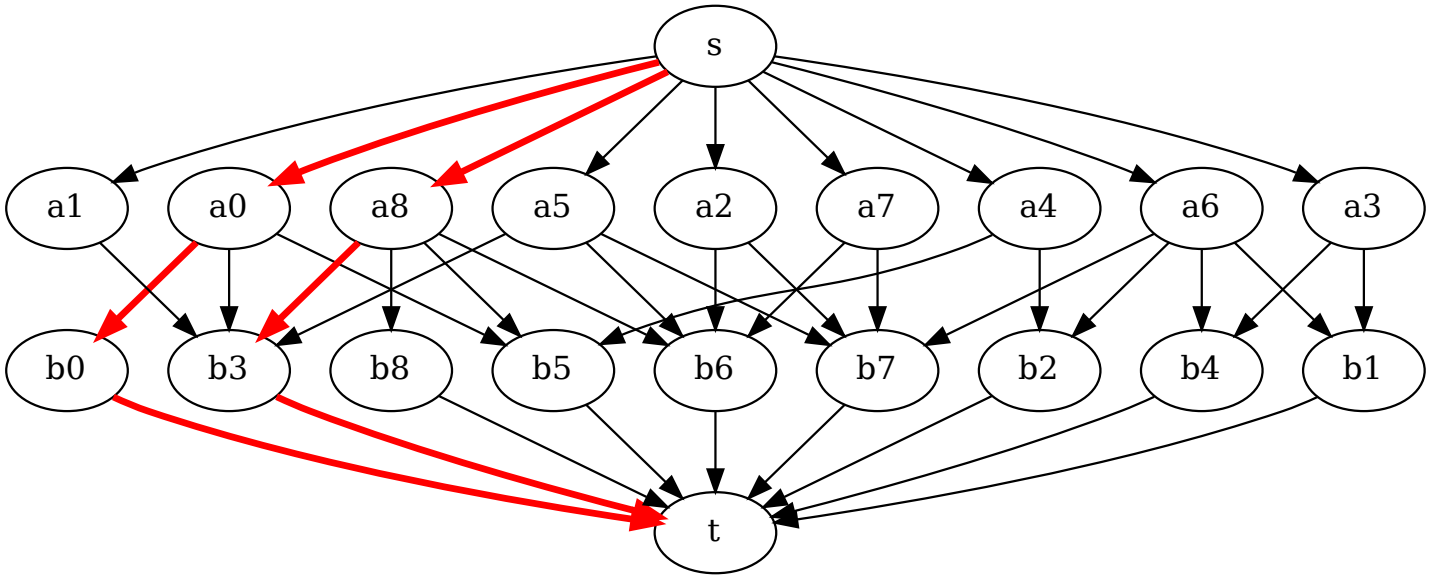
Orient graph, add s, t

Example



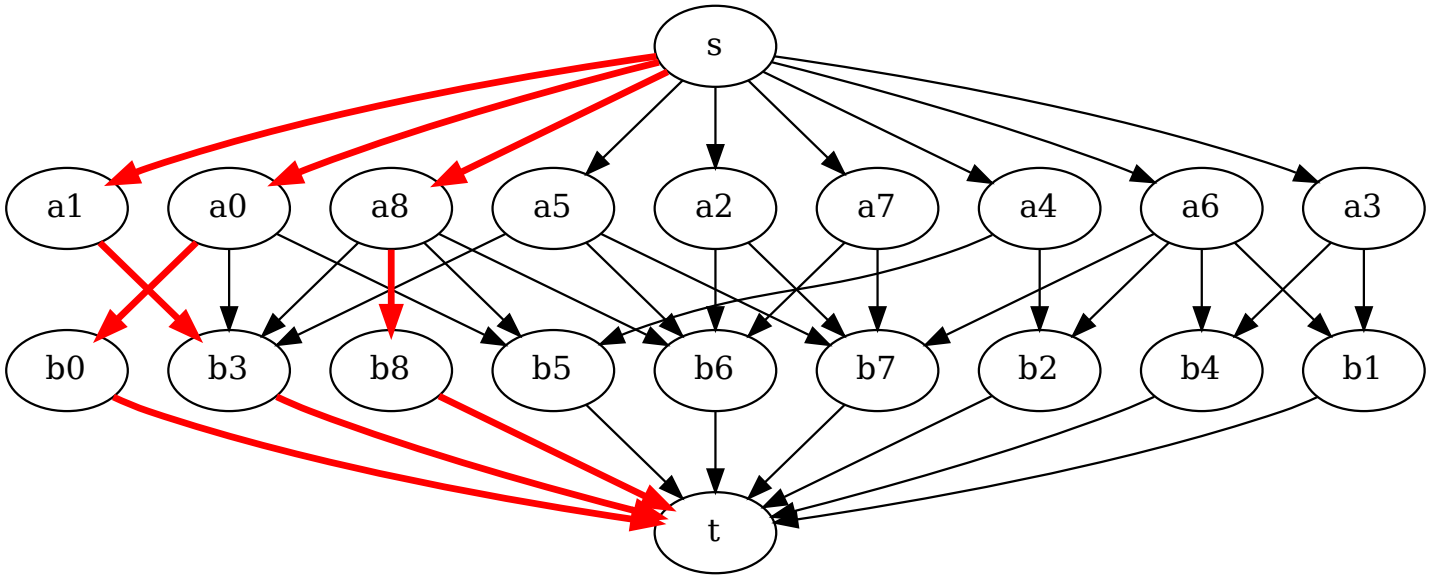
Run some max flow algorithm (some details skipped)

Example



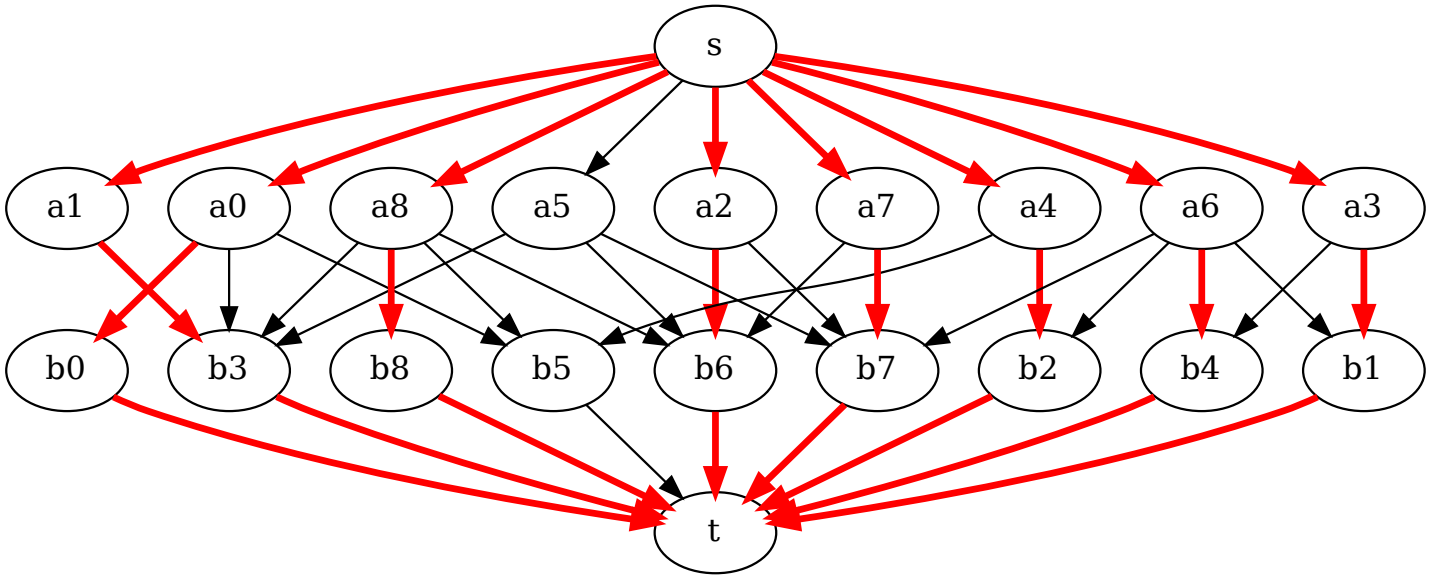
Run some max flow algorithm (some details skipped)

Example



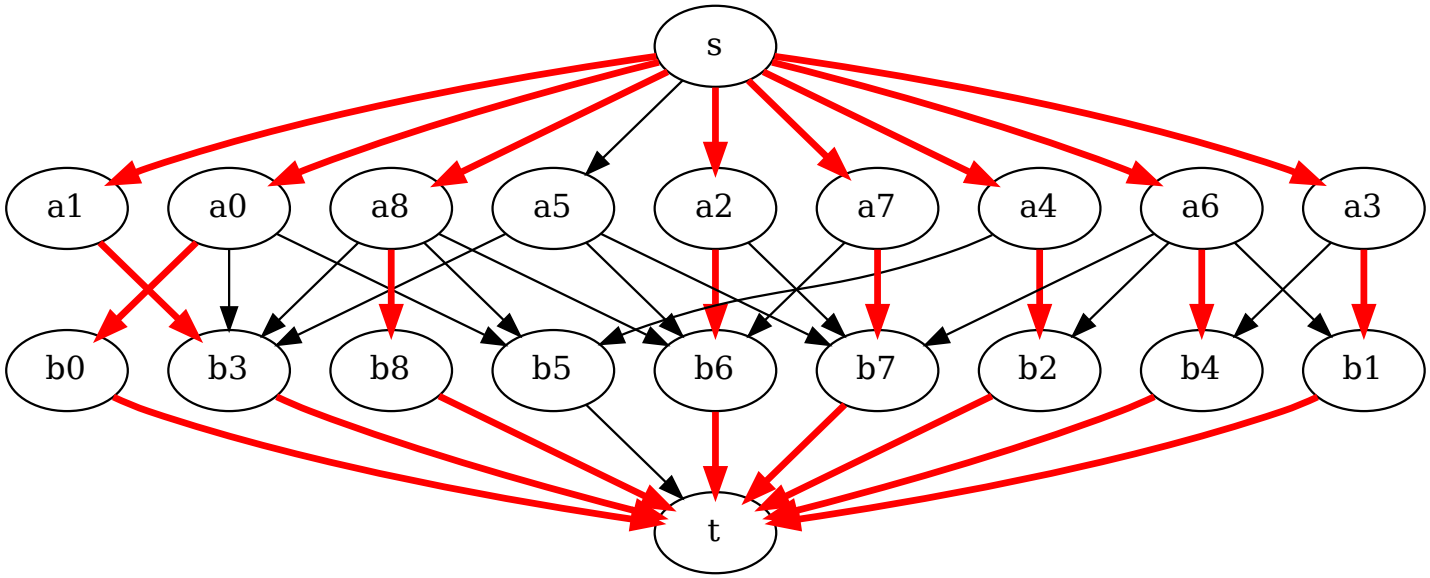
Run some max flow algorithm (some details skipped)

Example



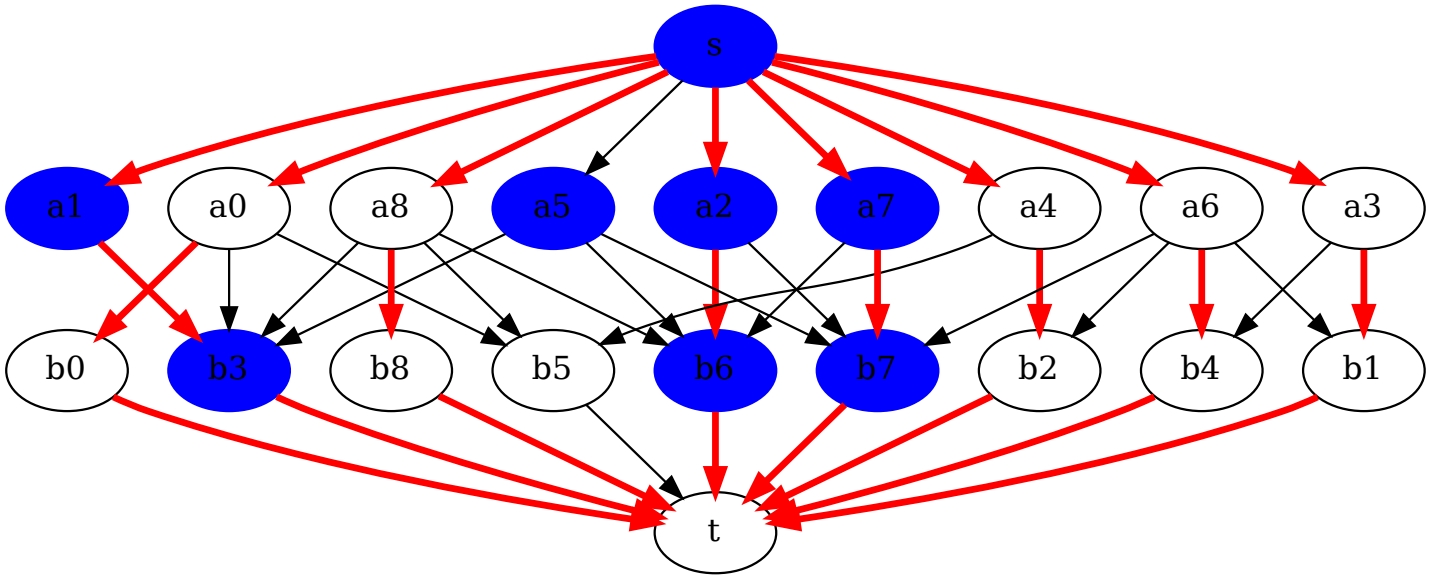
Arcs used in flow $\leftrightarrow$ Matching edges

Example



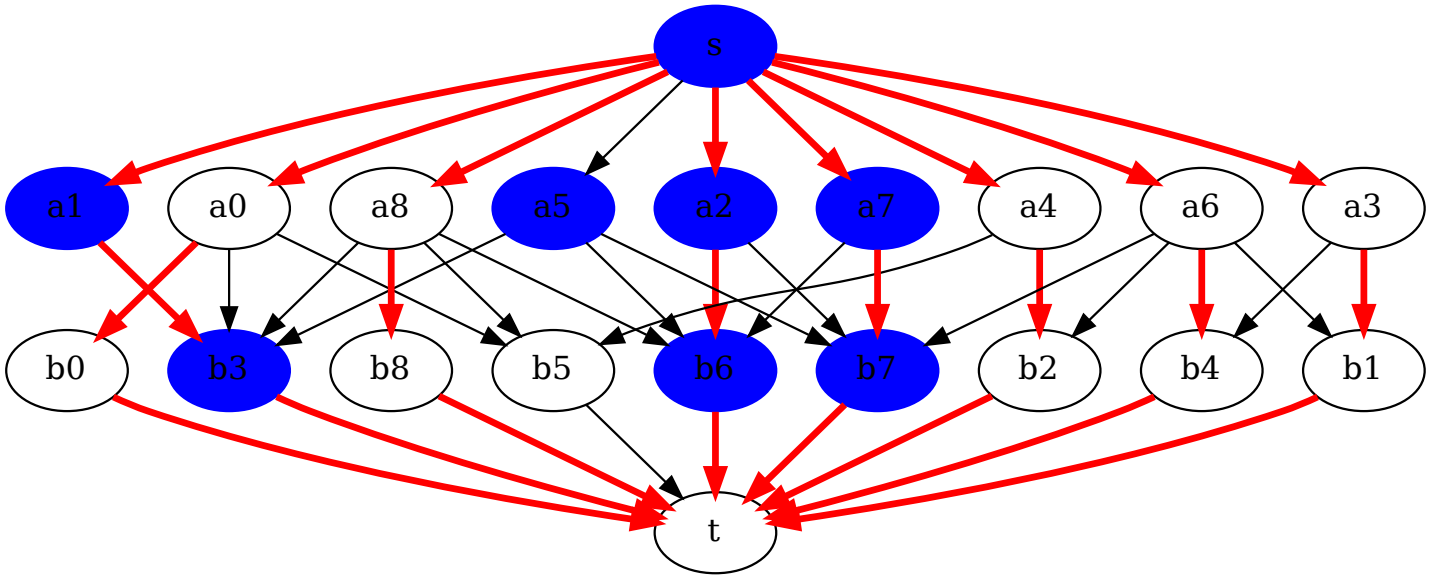
Flow is maximum $\leftrightarrow$ Matching is maximum

Example



Flow is maximum as blue set has cut size 8 (8 arcs coming out)

Example



$\Rightarrow$ maximum matching has size 8

Analysis

- Given $G = (V, E)$ with $V = A \cup B$ and $|E| = m$ we run Ford-Fulkerson
- Observe that $f^* \leq n$
- Running time: $O(mn)$
- (As mentioned, a much more complicated almost-linear time algorithm exists.)