

CS 445

Analysis of algorithms 2

Flows and cuts

Éric Schost

eschost@uwo.ca

Combinatorial optimization

Optimization: solving minimization or maximization problems, such as

- the maximum number of suitcases you can squeeze in the trunk of the car,
- the shortest route to connect a list of places on campus, ...

Combinatorial: the space where we do the search is finite (or maybe discrete).

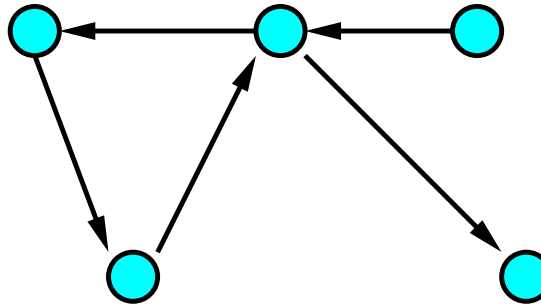
That **excludes** things such as $\min_{x \in [-1,1]} f(x)$, but it makes the task easier.

Today: How to send the maximal amount of **stuff** along a transportation network:
maximal flow problem.

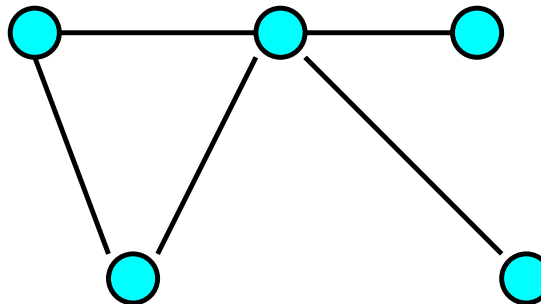
Graphs and flows

Graphs

Def. An **oriented** graph consists in a finite set of **nodes (vertices)** and a finite set of **oriented edges** that connect the nodes.



Def. A **symmetric (non-oriented)** graph consists in a finite set of **nodes (vertices)** and a finite set of **non-oriented edges** that connect the nodes.

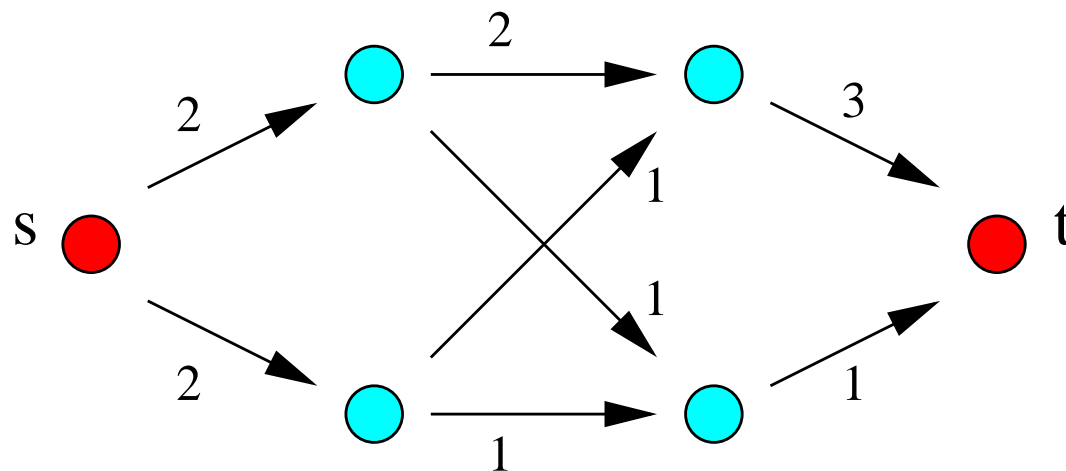


For the moment, we consider **oriented** graphs. Most of the time, they are **connected**.

Flows

Setup.

- Let G be an **oriented** graph, and let c be a **capacity** on the edges of G .
The graph is connected. The capacity of an edge is always an integer ≥ 0 .
- We isolate two vertices in G , which will be called the **source** s and the **sink** t .
There is no edge going to s or from t .
- We want to send as much “flow” as possible (water in pipes, material on transport networks, ...) while respecting certain rules.



Flows

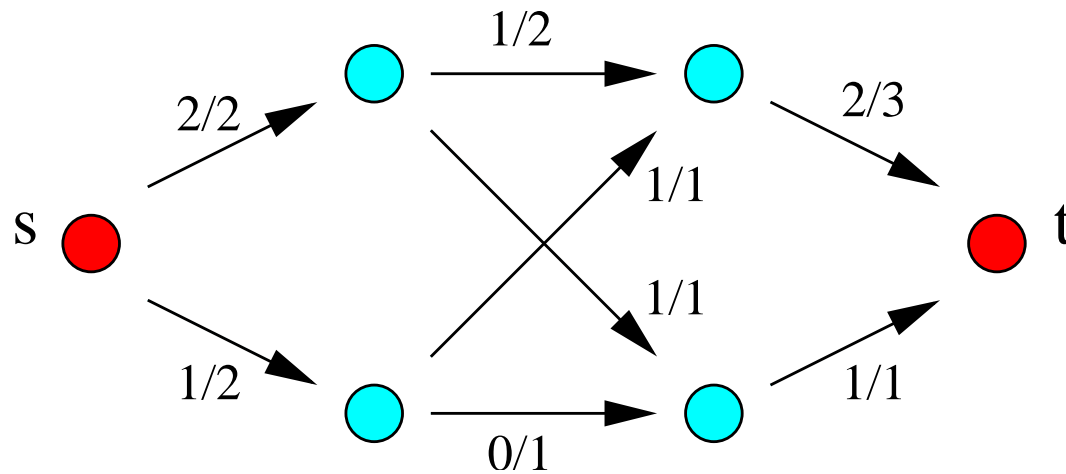
A **flow** is a function f of the edges.

- The amount of flow on an edge is ≥ 0 but cannot exceed its capacity.

For any edge e , we have $0 \leq f(e) \leq c(e)$.

- The amount of flow that enters a node equals the amount of flow that goes out of it.

You don't lose stuff at the nodes; like Kirchoff's laws in electricity.

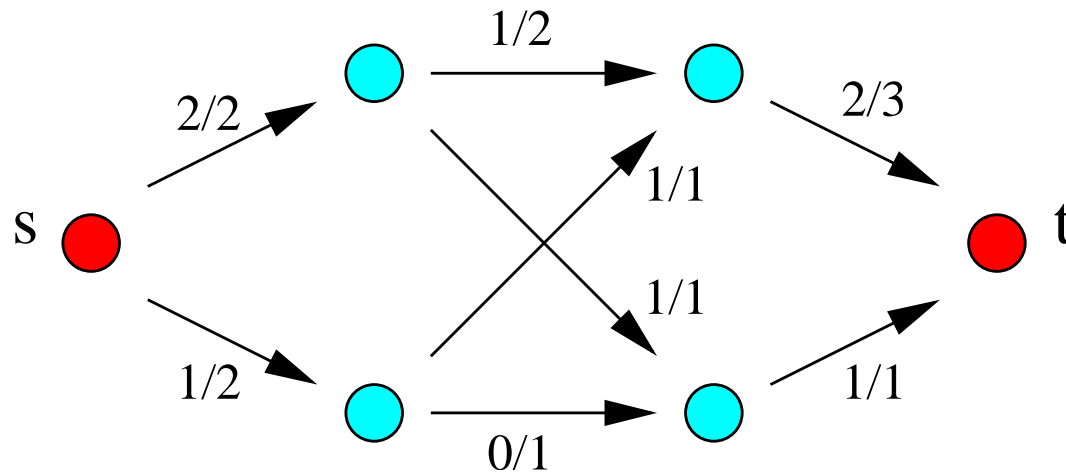


Flows

The **value** of a flow is the amount of flow that goes out of the source.

Formally,

$$\text{Val}(f) = \sum_{(s,v) \text{ edge}} f(e).$$



Here the value of the flow is 3.

MaxFlow problem: find a flow with a maximal value.

Producing bananas

Flow algorithms are useful because they can solve **a lot** of problems, even those that do **not** look like flow problems.

We have some banana factories F_1, F_2, F_3 , and some grocery stores S_1, S_2 .

- F_i can produce up to f_i tons of bananas,
- S_j demands s_j tons of bananas.

How to maximize the production?

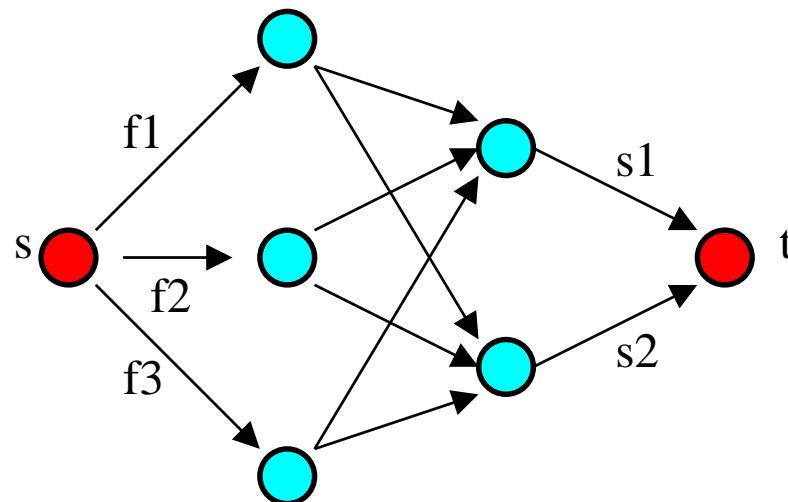
Producing bananas

Flow algorithms are useful because they can solve **a lot** of problems, even those that do **not** look like flow problems.

We have some banana factories F_1, F_2, F_3 , and some grocery stores S_1, S_2 .

- F_i can produce up to f_i tons of bananas,
- S_j demands s_j tons of bananas.

How to maximize the production? Compute the maximal flow in the following graph (the middle edges have large capacity, like $\max f_1, f_2, f_3$).

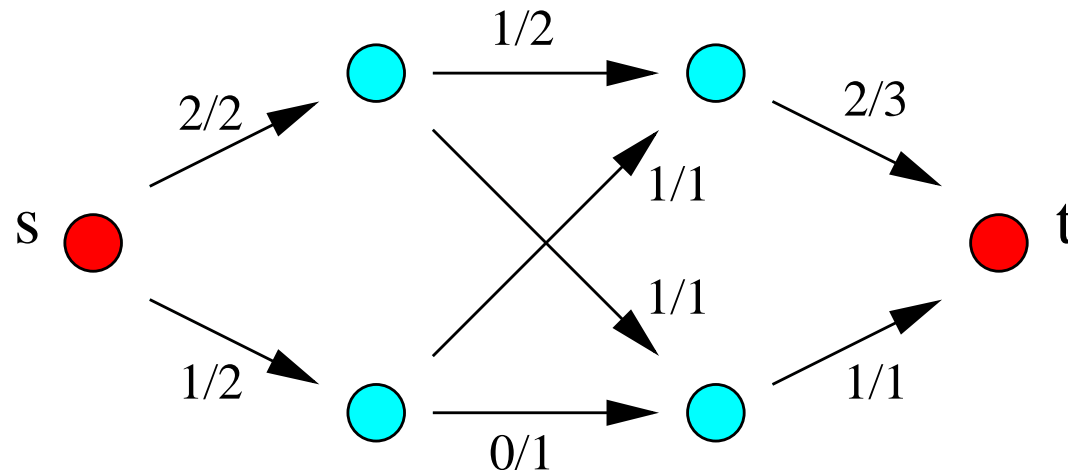


What if there is a limit $\ell_{i,j}$ on the quantity shippable from F_i to S_j ?

Ford-Fulkerson's algorithm

Improving the value

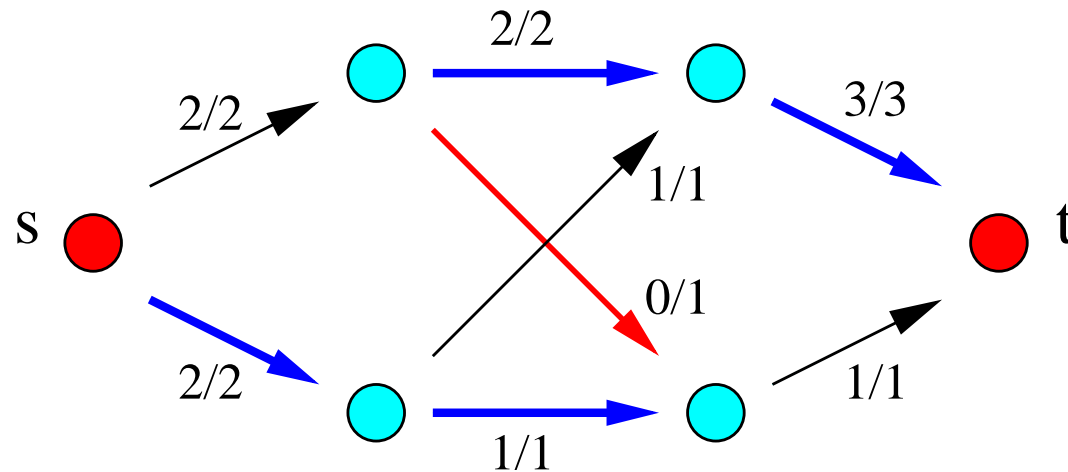
Improving the overall value may require that we **decrease** the flow through some edges.



Here, we are stuck if we only allow to increase all edges' flow.

Improving the value

Improving the overall value may require that we **decrease** the flow through some edges.



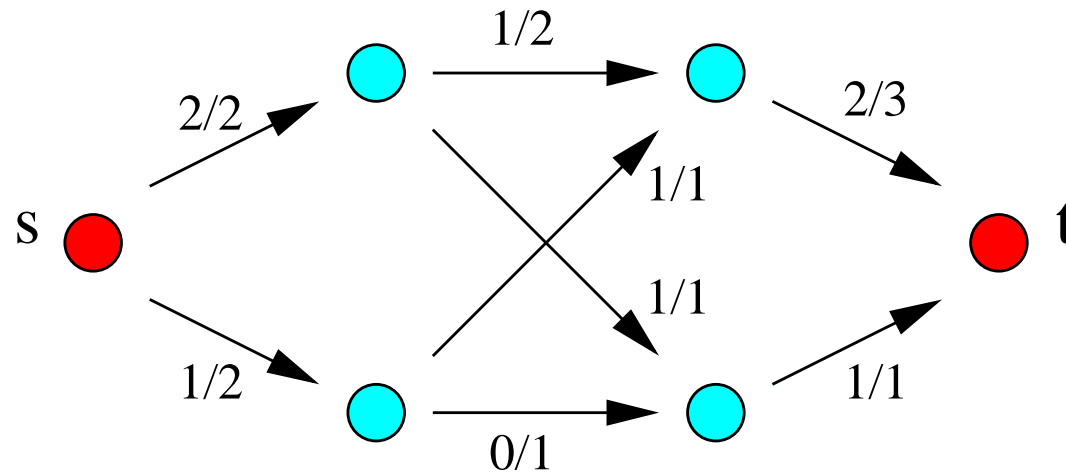
We improve the value to 4 by redirecting some flow that was going through the red edge.

This amounts to send one extra flow unit all along the colored path, taking the red edge “backward”.

The residual graph

The residual graph G_f shows all the ways to increase the value of the flow.

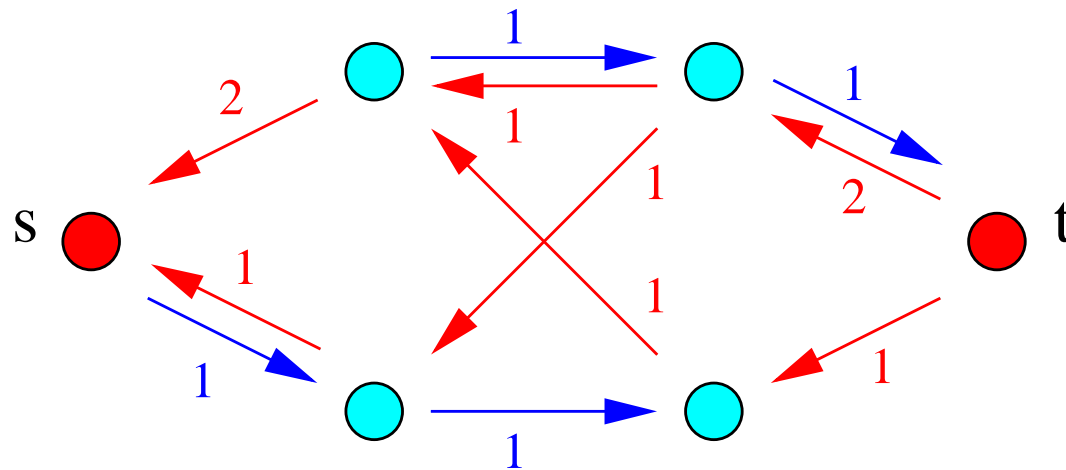
- The vertices of G_f are those of G .
- The edges of G_f show what modifications are possible. For e in $\text{edge}(G)$...
 - if $f(e) < c(e)$, put e in $\text{edge}(G_f)$ with the capacity $c(e) - f(e)$.
 - if $f(e) > 0$, put $\text{reverse}(e)$ in $\text{edge}(G_f)$ with the capacity $f(e)$.



The residual graph

The residual graph G_f shows all the ways to increase the value of the flow.

- The vertices of G_f are those of G .
- The edges of G_f show what modifications are possible. For e in $\text{edge}(G)$...
 - if $f(e) < c(e)$, put e in $\text{edge}(G_f)$ with the capacity $c(e) - f(e)$.
 - if $f(e) > 0$, put $\text{reverse}(e)$ in $\text{edge}(G_f)$ with the capacity $f(e)$.



Using the residual graph

A path from s to t in G_f indicates a way to increase the flow's value in G .

- a **blue edge** of capacity c means you can **increase** the flow by up to c on that edge in G
- a **red edge** of capacity c means you can **decrease** the flow by up to c on that edge in G .

Improvement step:

- compute the residual graph,
- find a path P , without loop, from s to t in G_f (if one exists),
- let x be the **minimal** value of all capacities on P in G_f .
- update the flow on G accordingly (increase the blue edges by x , decrease the red edges by x).

Correctness of the improvement step

Prop. After the improvement step, we still have a flow on G and its value has increased by x .

- We still have a flow:
 - all flow values on the edges are ≥ 0 and not exceeding the capacities;
Case discussion for red / blue edges.
 - at any vertex v , the amount of incoming flow still equals the amount of outgoing flow.
If v is not on the path, nothing changes there.
Else, case discussion on the color of the incoming / outgoing edges of v .
- The value increases:
 - the path must have a single edge containing s , and this edge is blue.

Finding a path in graph

Let H be a directed graph (here, it will be the residual graph) and let x and y be two vertices in H (here, they will be s and t).

How to find a path from x to y , if one exists?

Finding a path in graph

Let H be a directed graph (here, it will be the residual graph) and let x and y be two vertices in H (here, they will be s and t).

How to find a path from x to y , if one exists?

Input representation:

- vertices are numbered $1, \dots, N$, such that x is 1.
- edges are given by an array of N linked lists.

Finding a path in graph

Let H be a directed graph (here, it will be the residual graph) and let x and y be two vertices in H (here, they will be s and t).

How to find a path from x to y , if one exists?

Input representation:

- vertices are numbered $1, \dots, N$, such that x is 1.
- edges are given by an array of N linked lists.

Data structures:

- A boolean array `AlreadySeen` of size N , initially `false`
- An integer array `Parent` of size N , initially 0
- A set `ToDo` of edges (pairs of integers), initially empty.

Finding a path in graph

- set $\text{Parent}(1) = 1$
- put all the edges going out of 1 in `ToDo`
- while `ToDo` is not empty, do
 1. remove one edge (a, b) from `ToDo`
 2. if $\text{AlreadySeen}(b)$ is false
 - set $\text{Parent}(b) = a$
 - set $\text{AlreadySeen}(b) = \text{true}$
 - put all edges going out of b in `ToDo`

At the end, if $\text{AlreadySeen}(y) = \text{true}$, we can find its parent, then the parent of its parent, \dots , until we reach $x = 1$.

Each edge is only visited once, so the cost is $O(|\text{edges}(H)|)$.

Ford and Fulkerson's algorithm

Max Flow algorithm

1. Initialize the flow with all values at 0;
2. While possible, do the improvement step.

Theorem.

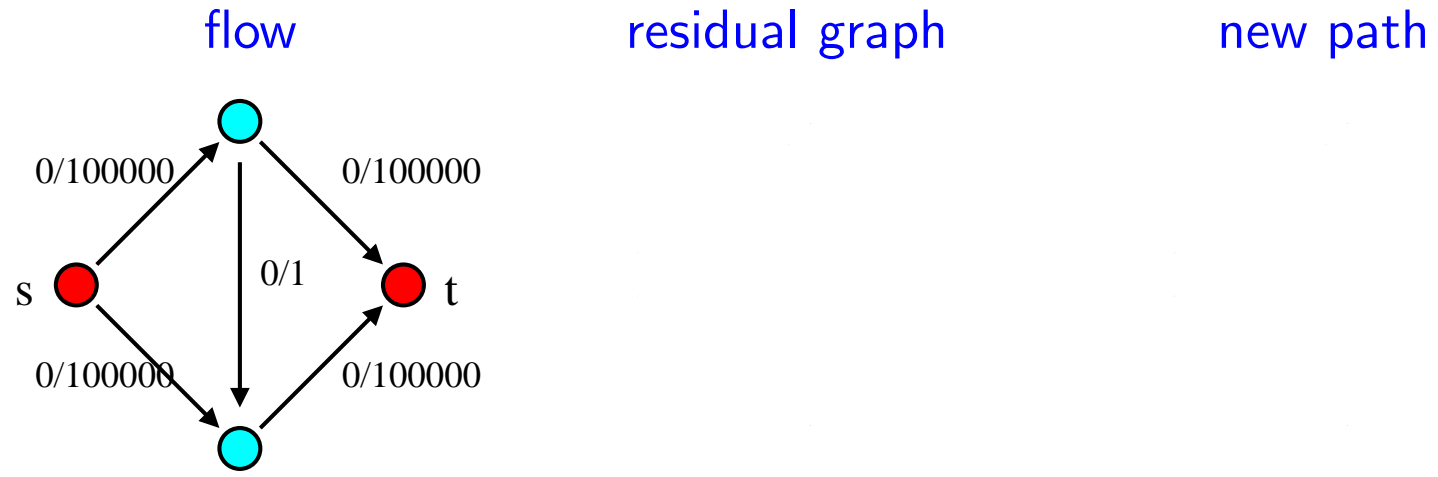
- The algorithm computes a maximal flow.

The proof is hard!

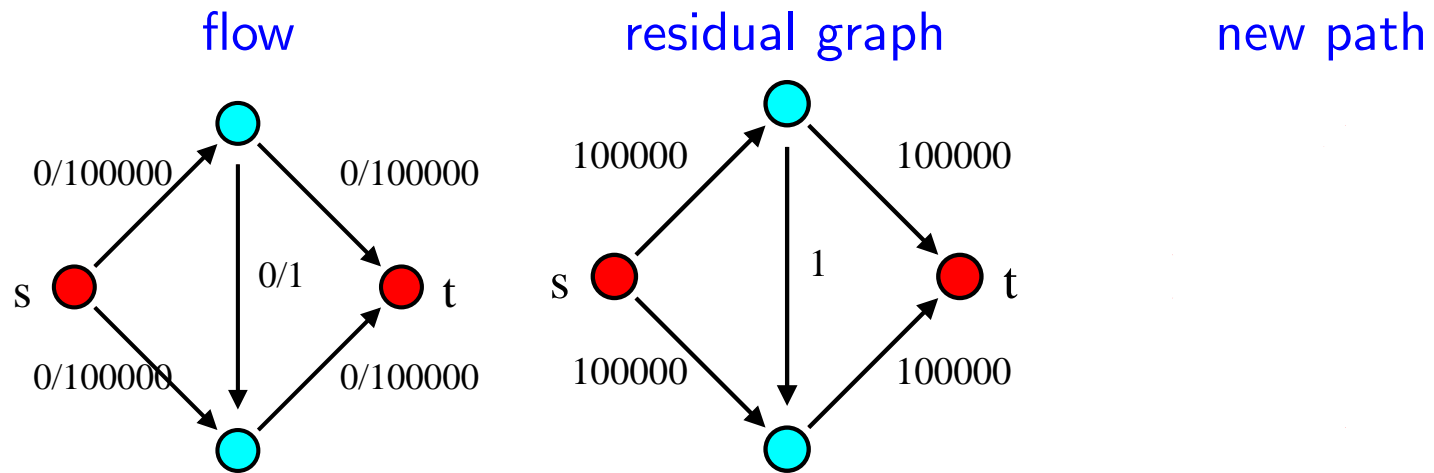
- The complexity is $O(|\text{edges}(G)| M)$, where M is the maximal value of the flow.

The proof is easy: each improvement step has cost $O(|\text{edges}(G)|)$ and increases the value by at least 1, so we can do at most M improvement steps.

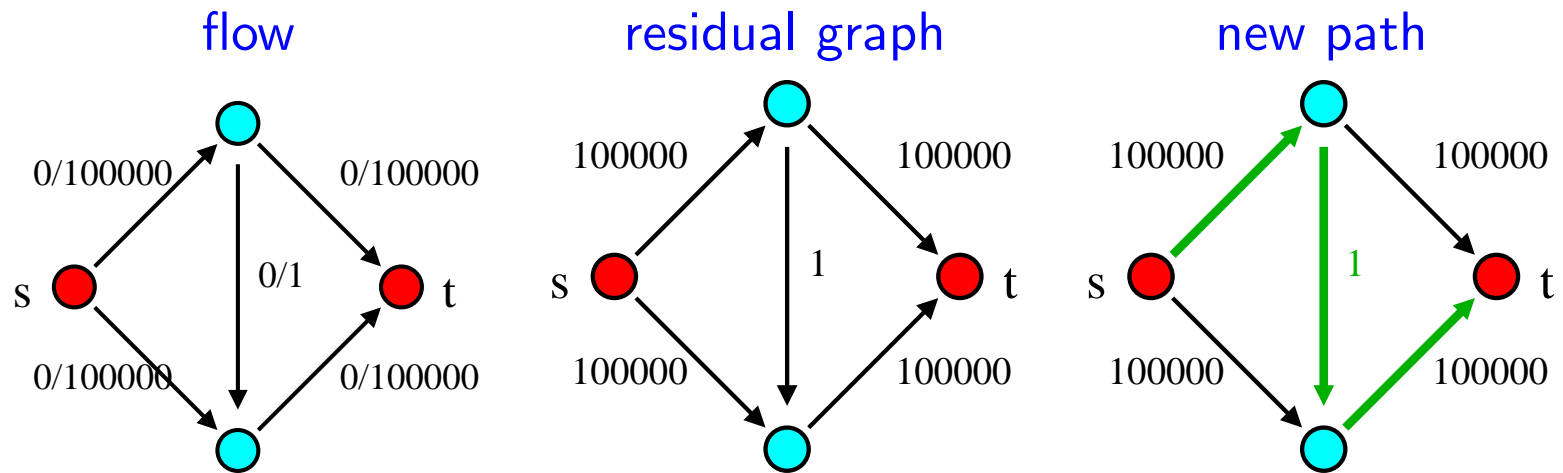
An example of a slow calculation



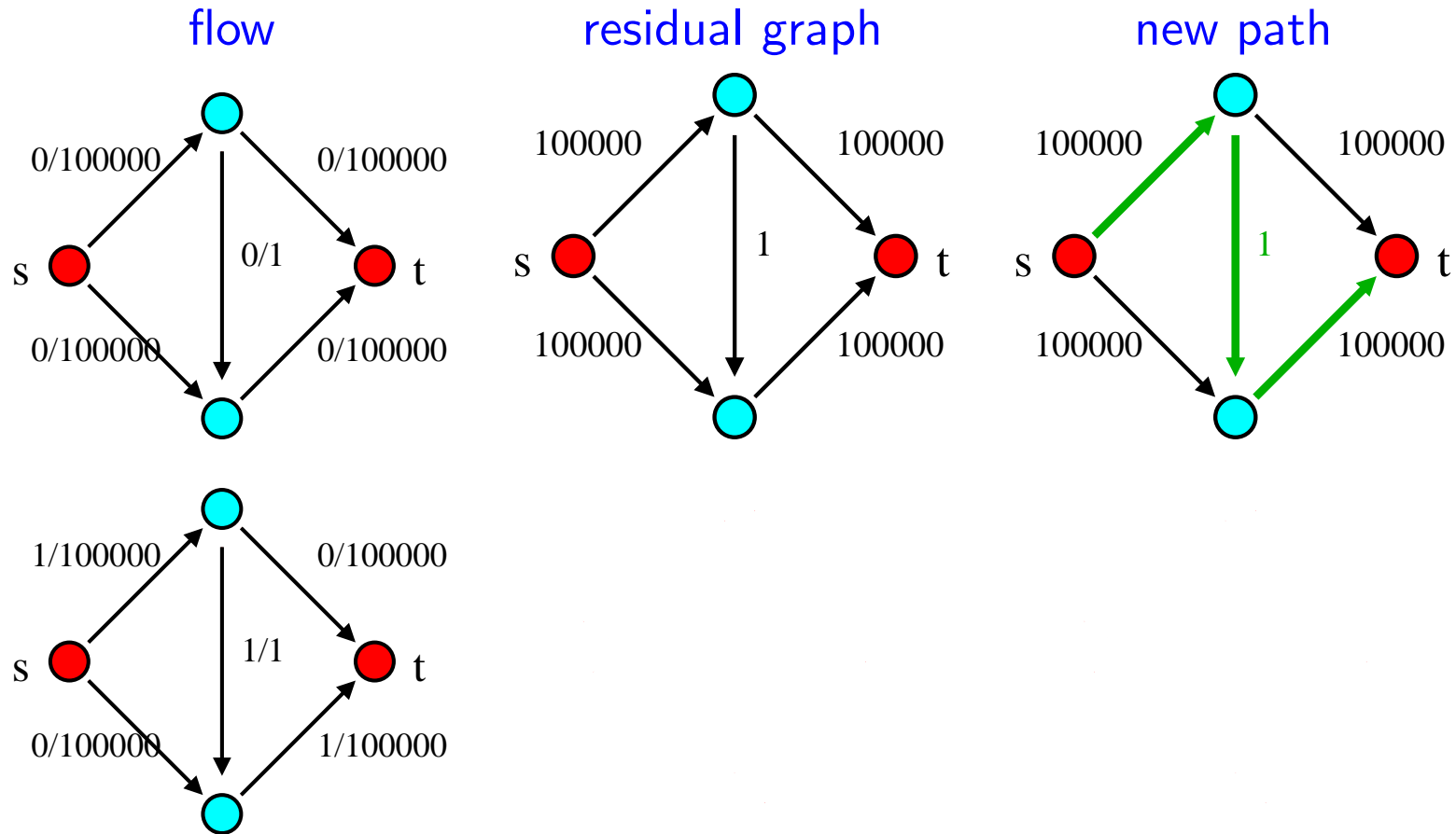
An example of a slow calculation



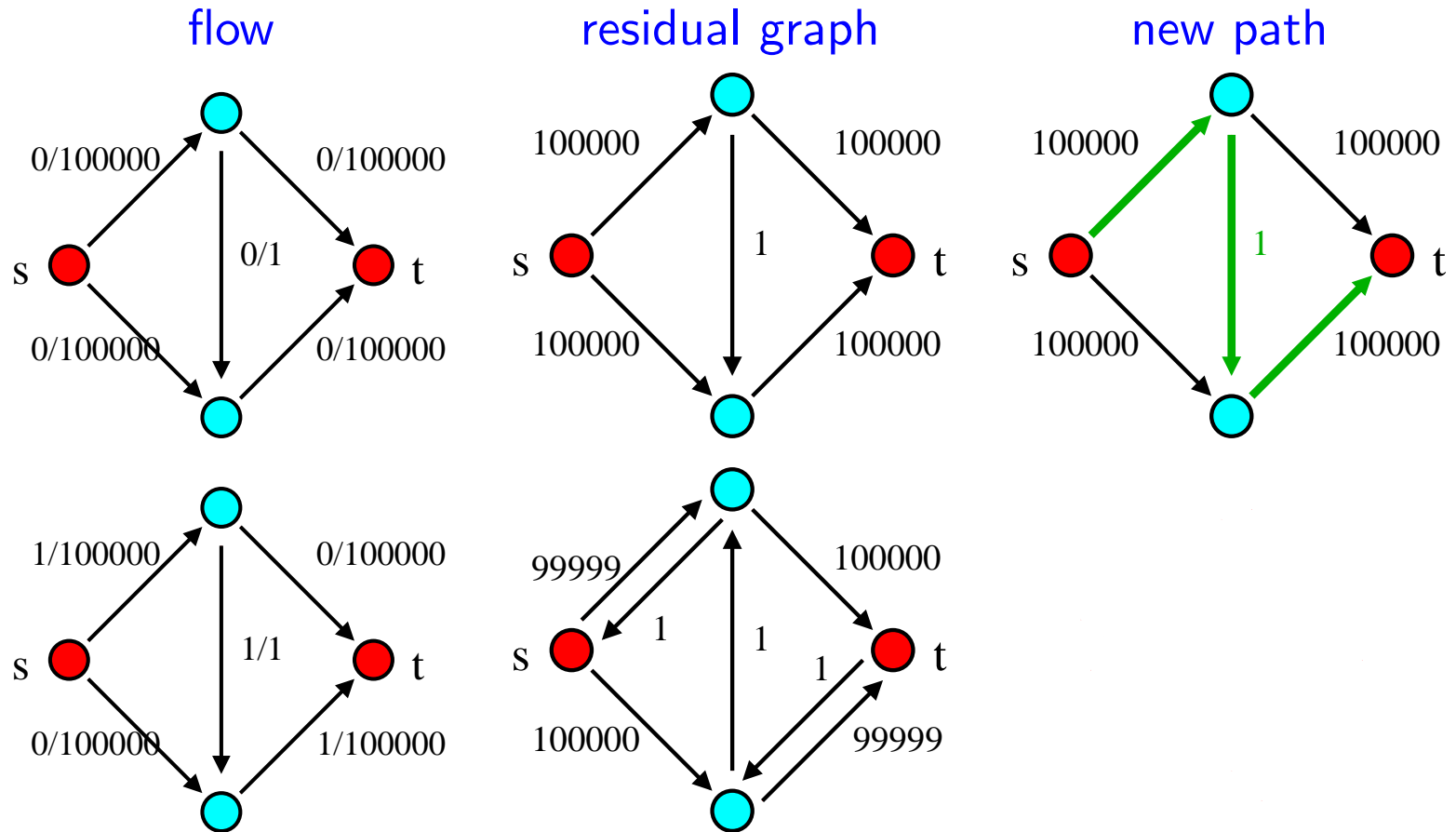
An example of a slow calculation



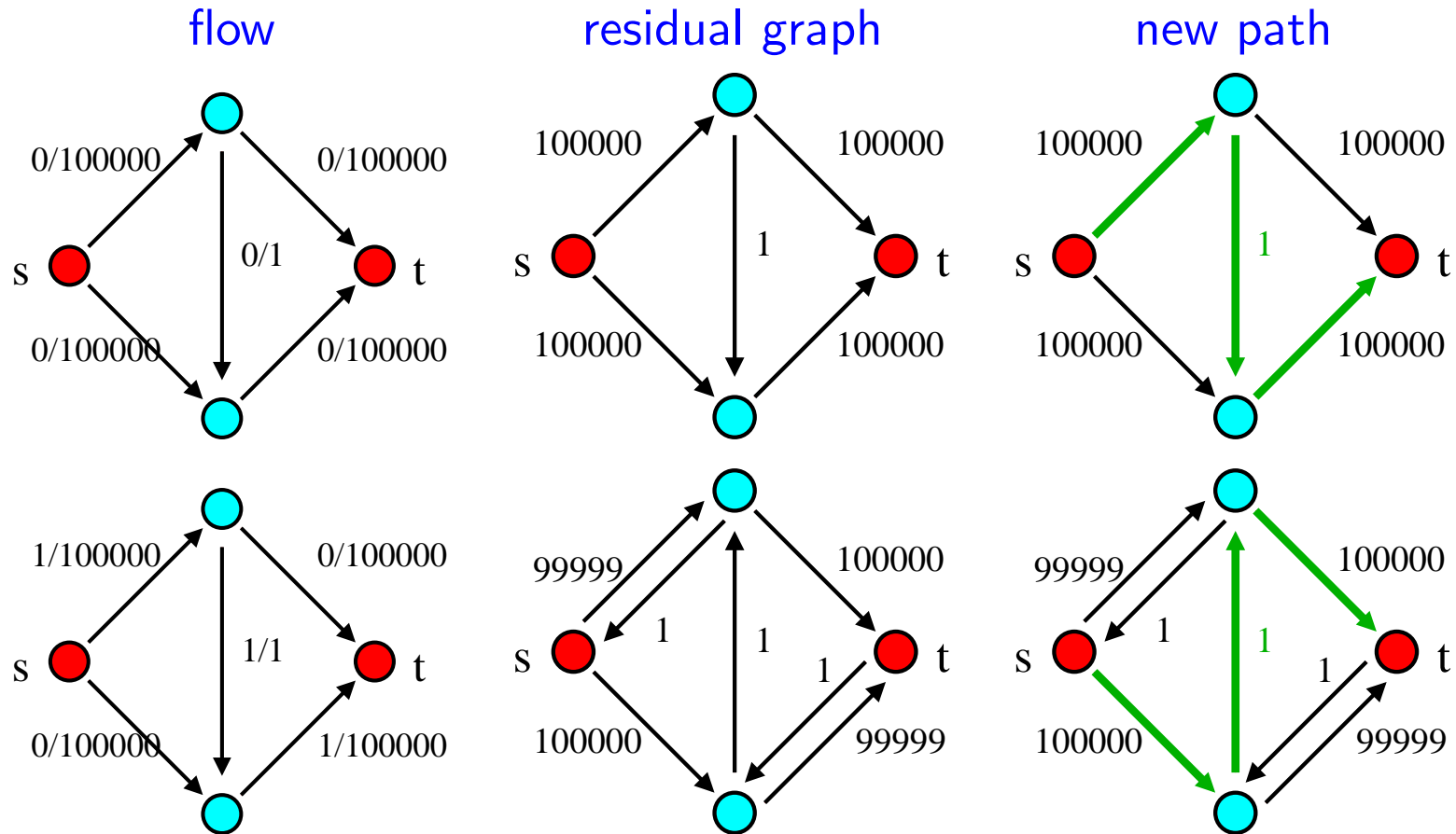
An example of a slow calculation



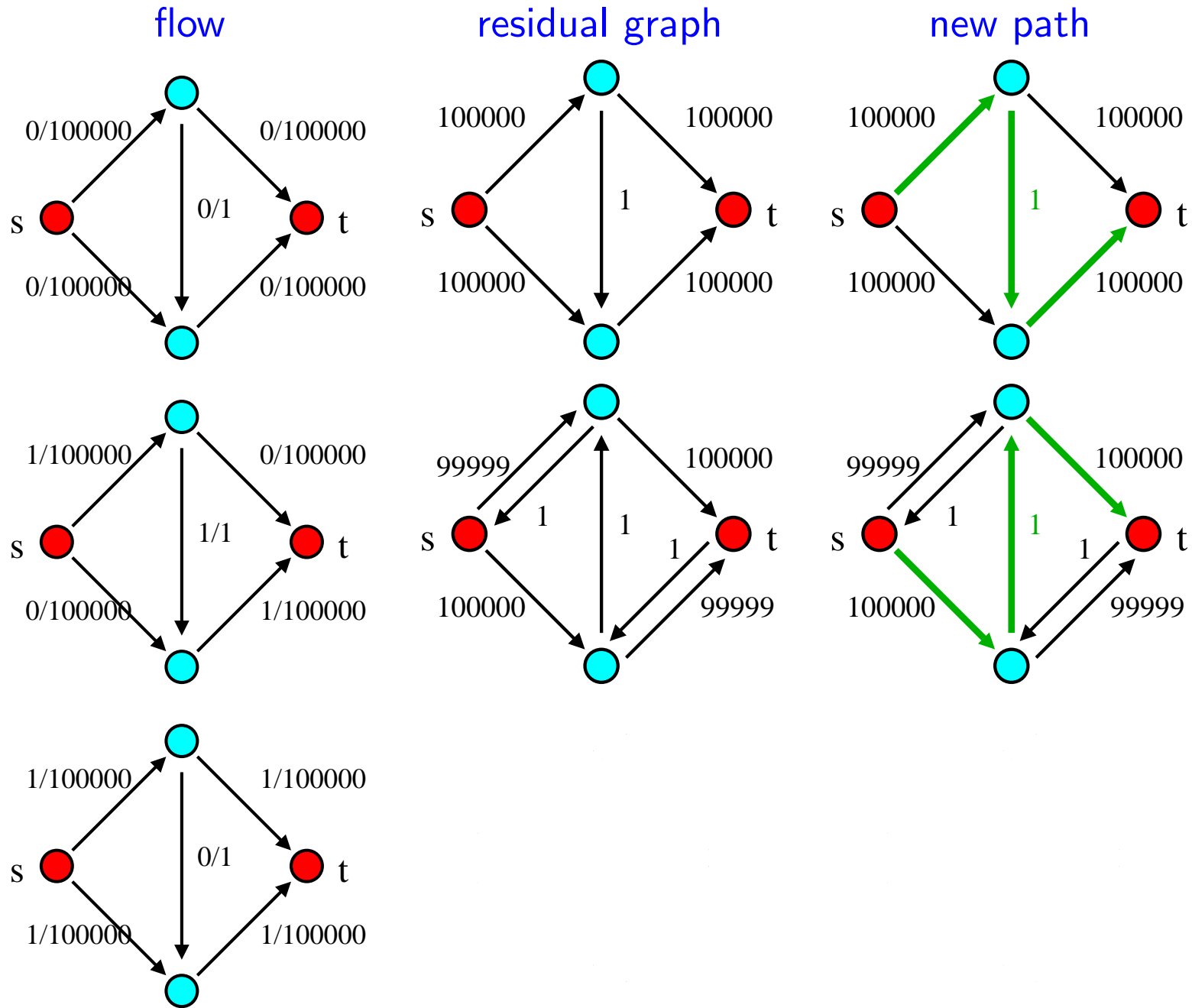
An example of a slow calculation



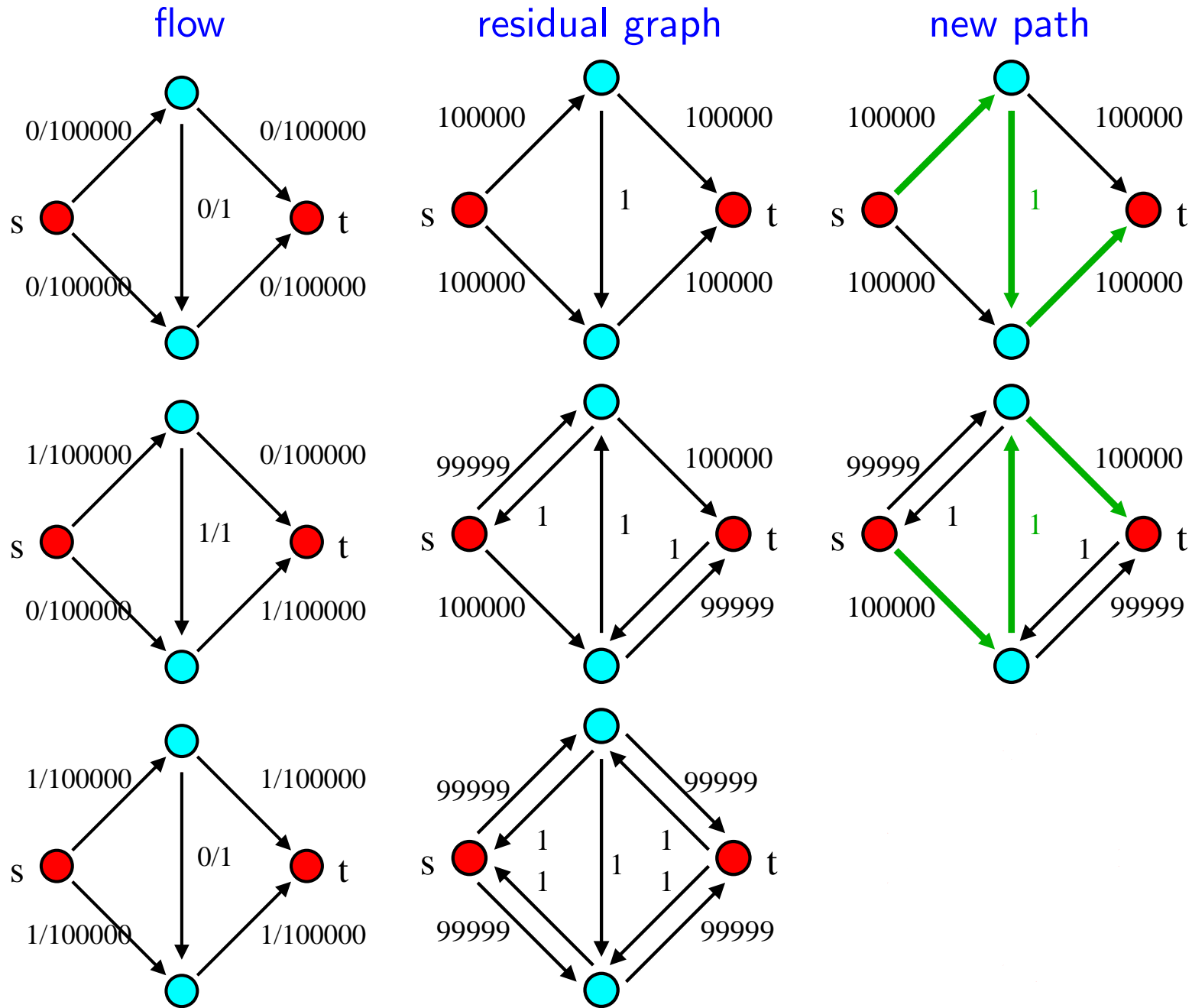
An example of a slow calculation



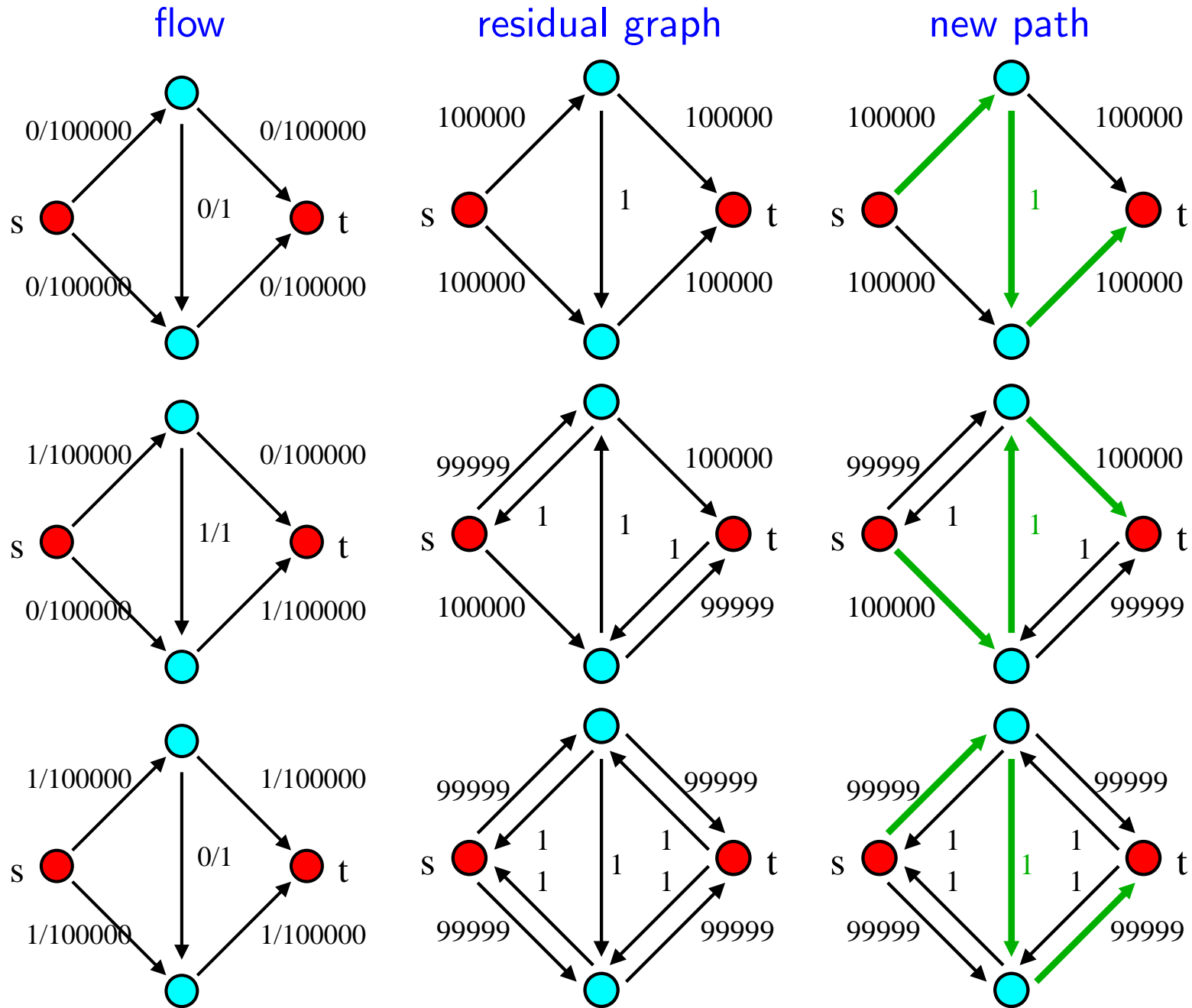
An example of a slow calculation



An example of a slow calculation



An example of a slow calculation



Cuts

Cuts

Definition

- A **cut** in the graph G is a partition of the vertices into 2 sets A and B , with $s \in A$ and $t \in B$.

Cuts

Definition

- A **cut** in the graph G is a partition of the vertices into 2 sets A and B , with $s \in A$ and $t \in B$.
- The **capacity** of the cut is

$$c(A, B) = \sum_{e \text{ going out of } A} c(e).$$

Remark that this does not depend on a flow, only on the graph and its capacities.

Cuts

Definition

- A **cut** in the graph G is a partition of the vertices into 2 sets A and B , with $s \in A$ and $t \in B$.
- The **capacity** of the cut is

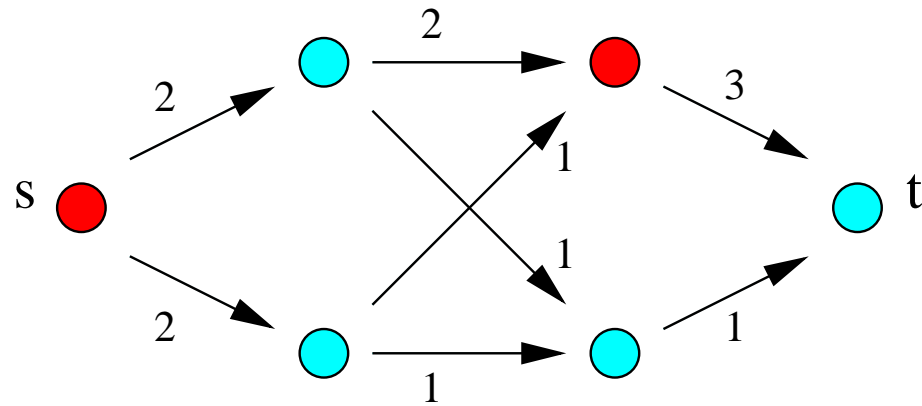
$$c(A, B) = \sum_{e \text{ going out of } A} c(e).$$

Remark that this does not depend on a flow, only on the graph and its capacities.

- If f is a flow on G , the **out-going** and **in-going** flows of the cut are

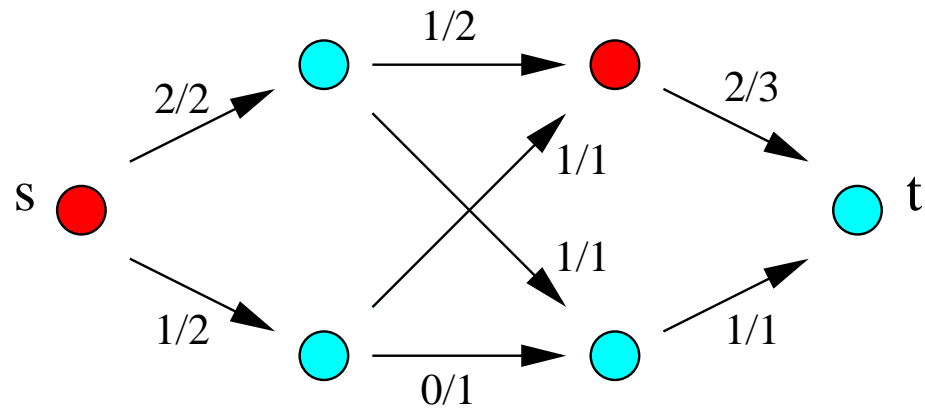
$$v_{\text{out}}(f, A, B) = \sum_{e \text{ going from } A \text{ to } B} f(e), \quad v_{\text{in}}(f, A, B) = \sum_{e \text{ going from } B \text{ to } A} f(e).$$

Examples



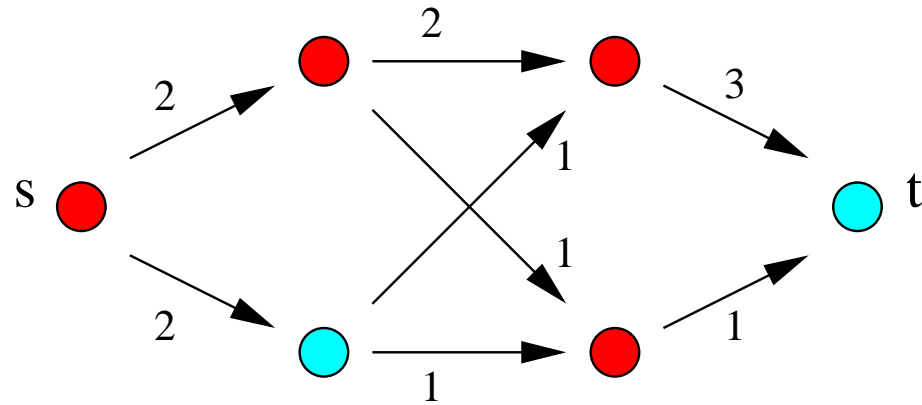
- A is in red and B in light blue,
- the capacity is $2 + 2 + 3 = 7$,

Examples



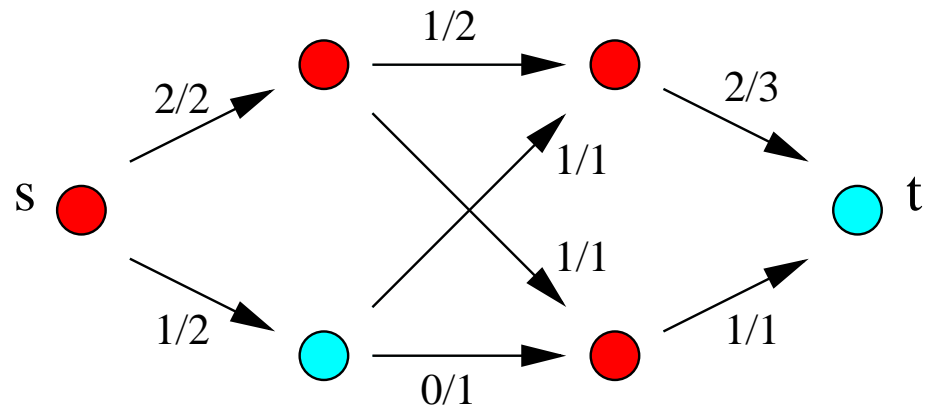
- A is in red and B in light blue,
- the capacity is $2 + 2 + 3 = 7$,
- the out-going flow is $2 + 1 + 2 = 5$,
- the in-going flow is $1 + 1 = 2$.

Examples



- A is in red and B in light blue,
- the capacity is $2 + 3 + 1 = 6$,

Examples



- A is in red and B in light blue,
- the capacity is $2 + 3 + 1 = 6$,
- the out-going flow is $1 + 2 + 1 = 4$,
- the in-going flow is 1.

Flows and cuts

Theorem

- For any flow f and any cut (A, B) , we have

$$\text{Val}(f) = v_{\text{out}}(f, A, B) - v_{\text{in}}(f, A, B)$$

Remark: this shows that what comes out of s equals what comes into t .

Flows and cuts

Theorem

- For any flow f and any cut (A, B) , we have

$$\text{Val}(f) = v_{\text{out}}(f, A, B) - v_{\text{in}}(f, A, B)$$

Remark: this shows that what comes out of s equals what comes into t .

Proof. We do an induction on A .

- This is true when $A = \{s\}$, by definition.
- Supposing this is true for a cut A, B , we show this is true for the cut $A' = A \cup \{v\}, B' = B - \{v\}$, for any vertex $v \in B$ (with $v \neq t$).

What we need to do:

- relate $v_{\text{out}}(f, A, B)$ to $v_{\text{out}}(f, A', B')$,
- relate $v_{\text{in}}(f, A, B)$ to $v_{\text{in}}(f, A', B')$.

Details of the proof, step 1

We can rewrite

$$\begin{aligned} v_{\text{out}}(f, A, B) &= \sum_{e \text{ going from } A \text{ to } B} f(e) \\ &= \sum_{e \text{ going from } A \text{ to } v} f(e) + \sum_{e \text{ going from } A \text{ to } B'} f(e) \end{aligned}$$

Details of the proof, step 1

We can rewrite

$$\begin{aligned} v_{\text{out}}(f, A, B) &= \sum_{e \text{ going from } A \text{ to } B} f(e) \\ &= \sum_{e \text{ going from } A \text{ to } v} f(e) + \sum_{e \text{ going from } A \text{ to } B'} f(e) \end{aligned}$$

and

$$\begin{aligned} v_{\text{out}}(f, A', B') &= \sum_{e \text{ going from } A' \text{ to } B'} f(e) \\ &= \sum_{e \text{ going from } A \text{ to } B'} f(e) + \sum_{e \text{ going from } v \text{ to } B'} f(e). \end{aligned}$$

Details of the proof, step 1

We can rewrite

$$\begin{aligned}v_{\text{out}}(f, A, B) &= \sum_{e \text{ going from } A \text{ to } B} f(e) \\ &= \sum_{e \text{ going from } A \text{ to } v} f(e) + \sum_{e \text{ going from } A \text{ to } B'} f(e)\end{aligned}$$

and

$$\begin{aligned}v_{\text{out}}(f, A', B') &= \sum_{e \text{ going from } A' \text{ to } B'} f(e) \\ &= \sum_{e \text{ going from } A \text{ to } B'} f(e) + \sum_{e \text{ going from } v \text{ to } B'} f(e).\end{aligned}$$

So

$$v_{\text{out}}(f, A, B) - v_{\text{out}}(f, A', B') = \sum_{e \text{ going from } A \text{ to } v} f(e) - \sum_{e \text{ going from } v \text{ to } B'} f(e)$$

Details of the proof, step 2

We can rewrite

$$\begin{aligned} v_{\text{in}}(f, A, B) &= \sum_{e \text{ going from } B \text{ to } A} f(e) \\ &= \sum_{e \text{ going from } v \text{ to } A} f(e) + \sum_{e \text{ going from } B' \text{ to } A} f(e) \end{aligned}$$

Details of the proof, step 2

We can rewrite

$$\begin{aligned} v_{\text{in}}(f, A, B) &= \sum_{e \text{ going from } B \text{ to } A} f(e) \\ &= \sum_{e \text{ going from } v \text{ to } A} f(e) + \sum_{e \text{ going from } B' \text{ to } A} f(e) \end{aligned}$$

and

$$\begin{aligned} v_{\text{in}}(f, A', B') &= \sum_{e \text{ going from } B' \text{ to } A'} f(e) \\ &= \sum_{e \text{ going from } B' \text{ to } A} f(e) + \sum_{e \text{ going from } B' \text{ to } v} f(e). \end{aligned}$$

Details of the proof, step 2

We can rewrite

$$\begin{aligned} v_{\text{in}}(f, A, B) &= \sum_{e \text{ going from } B \text{ to } A} f(e) \\ &= \sum_{e \text{ going from } v \text{ to } A} f(e) + \sum_{e \text{ going from } B' \text{ to } A} f(e) \end{aligned}$$

and

$$\begin{aligned} v_{\text{in}}(f, A', B') &= \sum_{e \text{ going from } B' \text{ to } A'} f(e) \\ &= \sum_{e \text{ going from } B' \text{ to } A} f(e) + \sum_{e \text{ going from } B' \text{ to } v} f(e). \end{aligned}$$

So

$$v_{\text{in}}(f, A, B) - v_{\text{in}}(f, A', B') = \sum_{e \text{ going from } v \text{ to } A} f(e) - \sum_{e \text{ going from } B' \text{ to } v} f(e)$$

Details of the proof, step 3

Because f is a flow, we have

$$\begin{aligned} \sum_{e \text{ going from } v \text{ to } A} f(e) + \sum_{e \text{ going from } v \text{ to } B'} f(e) \\ = \sum_{e \text{ going from } B' \text{ to } v} f(e) + \sum_{e \text{ going from } A \text{ to } v} f(e) \end{aligned}$$

Details of the proof, step 3

Because f is a flow, we have

$$\begin{aligned} \sum_{e \text{ going from } v \text{ to } A} f(e) + \sum_{e \text{ going from } v \text{ to } B'} f(e) \\ = \sum_{e \text{ going from } B' \text{ to } v} f(e) + \sum_{e \text{ going from } A \text{ to } v} f(e) \end{aligned}$$

Plugging this into the equalities of step 1 and step 2, we get

$$v_{\text{in}}(f, A, B) - v_{\text{in}}(f, A', B') = v_{\text{out}}(f, A, B) - v_{\text{out}}(f, A', B')$$

Details of the proof, step 3

Because f is a flow, we have

$$\begin{aligned} \sum_{e \text{ going from } v \text{ to } A} f(e) + \sum_{e \text{ going from } v \text{ to } B'} f(e) \\ = \sum_{e \text{ going from } B' \text{ to } v} f(e) + \sum_{e \text{ going from } A \text{ to } v} f(e) \end{aligned}$$

Plugging this into the equalities of step 1 and step 2, we get

$$v_{\text{in}}(f, A, B) - v_{\text{in}}(f, A', B') = v_{\text{out}}(f, A, B) - v_{\text{out}}(f, A', B')$$

This gives

$$\begin{aligned} v_{\text{out}}(f, A', B') - v_{\text{in}}(f, A', B') &= v_{\text{in}}(f, A, B) - v_{\text{out}}(f, A, B) \\ &= \text{Val}(f). \end{aligned}$$

Maximum flow and minimal cut

Consequence

- For any flow f and any cut (A, B) , we have

$$\text{Val}(f) \leq c(A, B).$$

Maximum flow and minimal cut

Consequence

- For any flow f and any cut (A, B) , we have

$$\text{Val}(f) \leq c(A, B).$$

Proof.

$$\begin{aligned}\text{Val}(f) &= v_{\text{out}}(f, A, B) - v_{\text{in}}(f, A, B) \\ &\leq v_{\text{out}}(f, A, B) \\ &\leq c(A, B)\end{aligned}$$

Consequence: the maximal value of a flow \leq minimal capacity of a cut.

Proof of correctness

Theorem

- Ford-Fulkerson's algorithm computes the maximal flow.

Proof of correctness

Theorem

- Ford-Fulkerson's algorithm computes the maximal flow.

Proof. Let f be the flow obtained by Ford-Fulkerson's algorithm.

To prove correctness, we will find a cut (A, B) such that

$$\text{Val}(f) = c(A, B).$$

Why is this enough?

- For any other flow f' , $\text{Val}(f') \leq c(A, B) = \text{Val}(f)$.
- So f is the maximal flow.

Proof of correctness

Theorem

- Ford-Fulkerson's algorithm computes the maximal flow.

Proof. Let f be the flow obtained by Ford-Fulkerson's algorithm.

To prove correctness, we will find a cut (A, B) such that

$$\text{Val}(f) = c(A, B).$$

Why is this enough?

- For any other flow f' , $\text{Val}(f') \leq c(A, B) = \text{Val}(f)$.
- So f is the maximal flow.

Remark: this also shows that **maximal value** of a flow = **minimal capacity** of a cut.

So Ford and Fulkerson's algorithm can be used to solve **MinCut** (minimal cut) problems.

Finding a cut

Let f be the flow obtained by Ford-Fulkerson's algorithm. We define the cut (A, B) by

- A is the sets of all vertices connected by a path from s in the residual graph.
- B is the set of all other vertices.

Finding a cut

Let f be the flow obtained by Ford-Fulkerson's algorithm. We define the cut (A, B) by

- A is the sets of all vertices connected by a path from s in the residual graph.
- B is the set of all other vertices.

Is it **really** a cut?

- This is a partition of the vertices.
- Of course, s is in A .
- When the algorithm finishes, there is no path $s \rightarrow t$ in the residual graph.
So t is in B .

So (A, B) is a cut.

What's left to prove: $\text{Val}(f) = c(A, B)$.

Checking the edges between A and B

Let e be an edge from $a \in A$ to $b \in B$ in G .

- If $f(e) < c(e)$, then e would be in the residual graph.
- But there is **no** edge $a \rightarrow b$ in the residual graph.
- So $f(e) = c(e)$.

Summing over all these e 's, we see that

$$v_{\text{out}}(f, A, B) = \sum_{e \text{ going from } A \text{ to } B} f(e) = \sum_{e \text{ going from } A \text{ to } B} c(e) = c(A, B).$$

Checking the edges between A and B

Let e be an edge from $b \in B$ to $a \in A$ in G .

- If $f(e) \neq 0$, then $\text{reverse}(e)$ would be in the residual graph.
- $\text{reverse}(e)$ goes from a to b
- But there is **no** edge $a \rightarrow b$ in the residual graph.
- So $f(e) = 0$.

Summing over all these e 's, we see that

$$v_{\text{in}}(f, A, B) = \sum_{e \text{ going from } B \text{ to } A} f(e) = 0.$$

Finally,

$$\text{Val}(f) = v_{\text{out}}(f, A, B) - v_{\text{in}}(f, A, B) = c(A, B).$$

Applications

Bipartite matchings

A situation with students and professors:

- each professor offers **one** internship, but not all students may be eligible;
- each student wants **one** internship, and is ready to go for any of them.

Example: 3 students and 3 professors.

P_1 will only consider S_1 and S_2

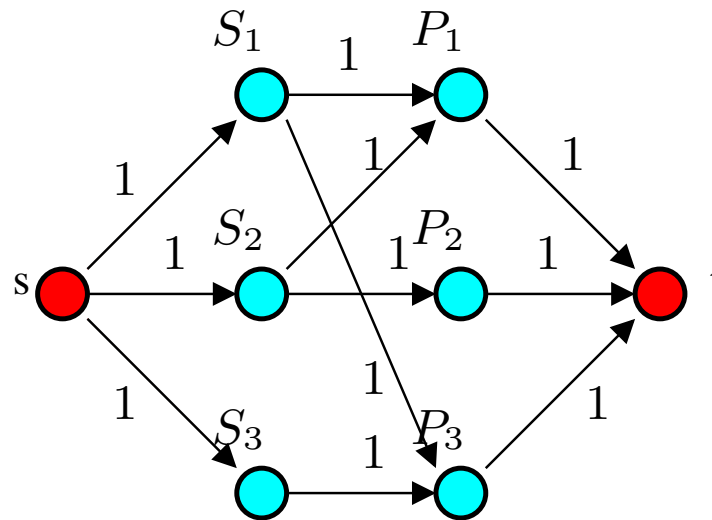
P_2 will only consider S_2

P_3 will only consider S_1 and S_3

How to find the best matching?

Bipartite matchings

Consider the **bipartite graph**



Prop.

- Matchings correspond to flow with values 0's and 1's.
- A maximal flow in this graph has values 0's and 1's.

So Ford and Fulkerson's algorithm solves the problem.

In general

Bipartite graph:

- a symmetric graph whose vertices are split into two groups S_i and P_j , with no edge between the S_i 's or the P_j 's.

Bipartite matching:

- a set of edges (between the S_i and the P_j) with no common vertex, or
- vertices S_{m_1}, \dots, S_{m_r} and $P_{\ell_1}, \dots, P_{\ell_r}$ such that S_{m_i} is connected to P_{ℓ_i} only.

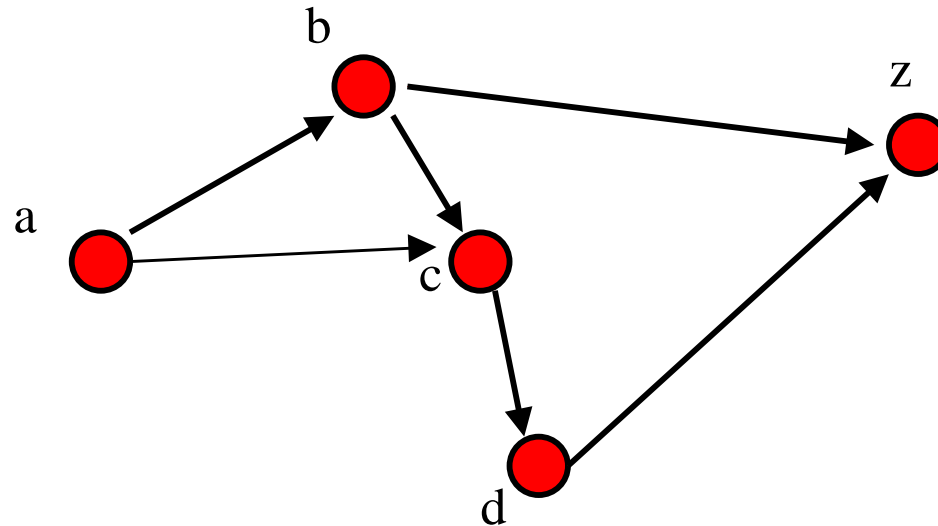
How to find one: set up a flow problem as before:

- A matching of size r gives a 0/1 flow of value r .
- A 0/1 flow of value r gives a matching of size r .
- In a graph with capacities 0 and 1, Ford and Fulkerson's algorithm returns a 0/1 flow.

(a 0/1 flow is a flow which only takes values 0 and 1)

A shipping problem

Some cities a, b, c, d, \dots want to send supplies to a city z , with a network of roads like this:

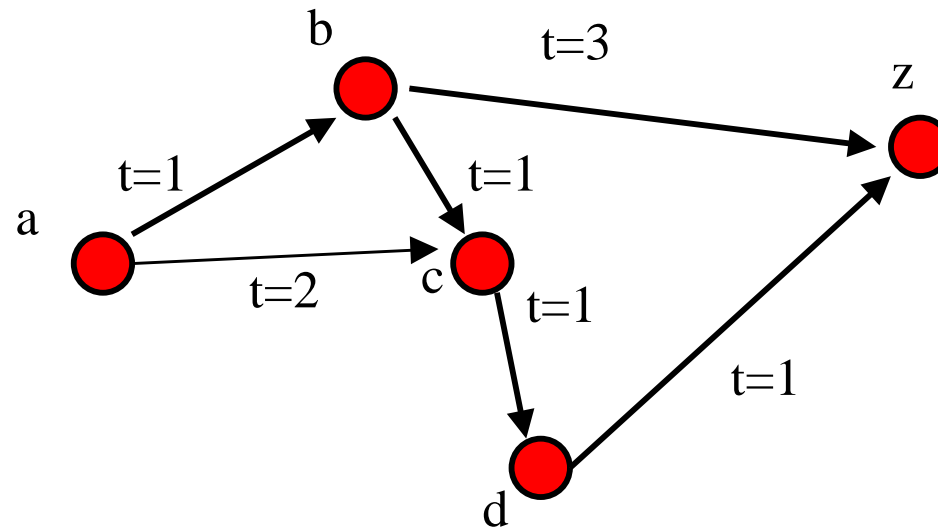


Initially, there are s_a, \dots units of stuff in a, \dots . We take **shipping time** into account: a maximum of stuff should arrive before $t = T$.

- the amount of supply that can go from a to b **per time unit** is $c_{a,b}$, and the same for $c_{a,c}, \dots$
- the **traversal time** from a to b is $t_{a,b}$, and the same for $t_{a,c}, \dots$

A shipping problem

Some cities a, b, c, d, \dots want to send supplies to a city z , with a network of roads like this:

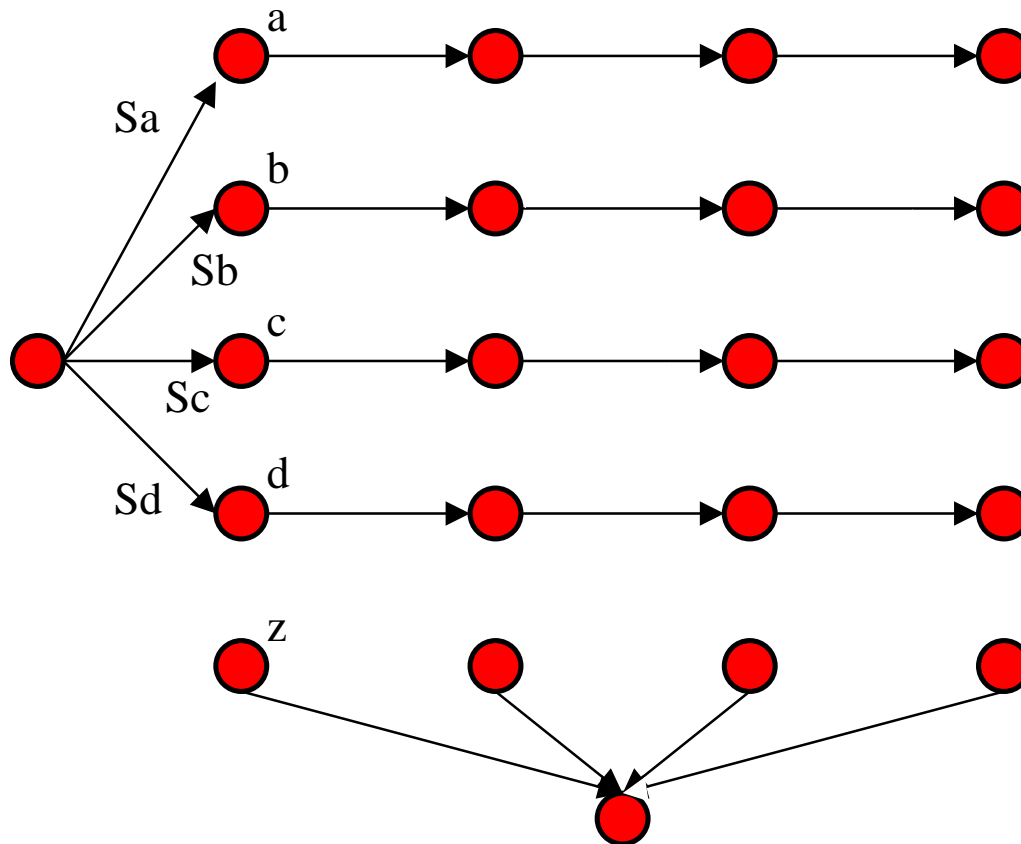


Initially, there are s_a, \dots units of stuff in a, \dots . We take **shipping time** into account: a maximum of stuff should arrive before $t = T$.

- the amount of supply that can go from a to b **per time unit** is $c_{a,b}$, and the same for $c_{a,c}, \dots$
- the **traversal time** from a to b is $t_{a,b}$, and the same for $t_{a,c}, \dots$

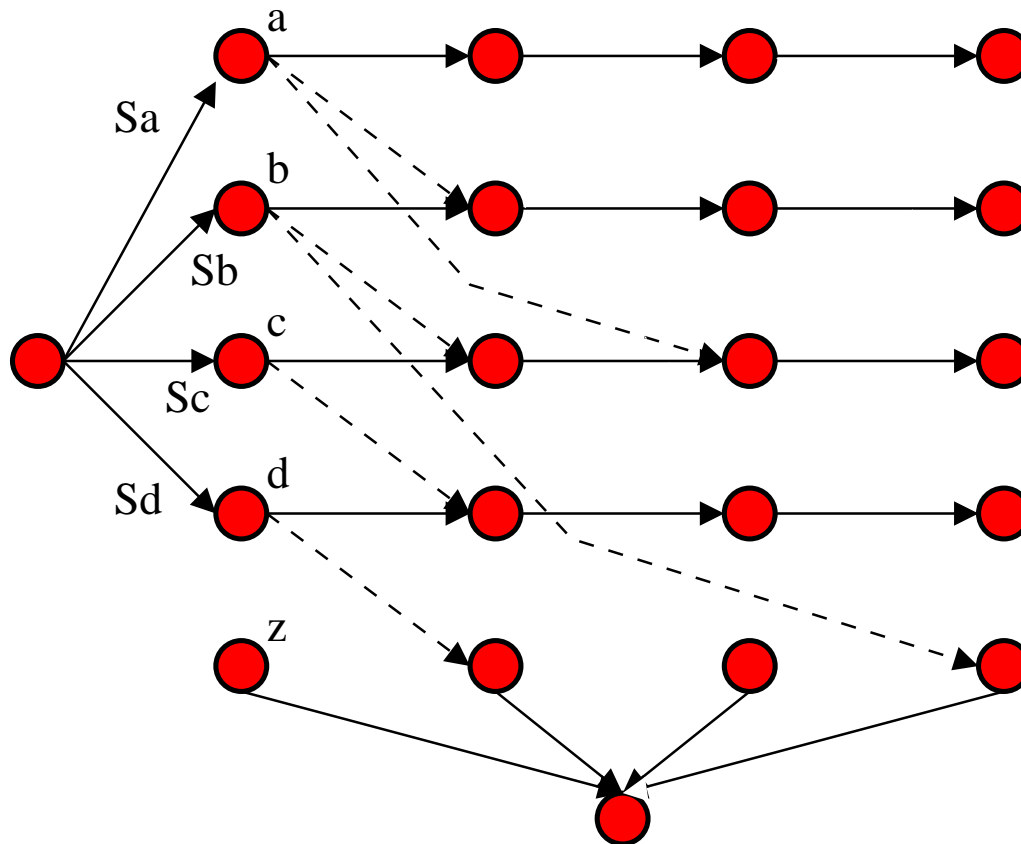
A shipping problem

- The previous graph is copied $T + 1$ times: one copy for each time step.
- Edges are arranged to match the time constraints.
- There is a super-source and a super-sink.



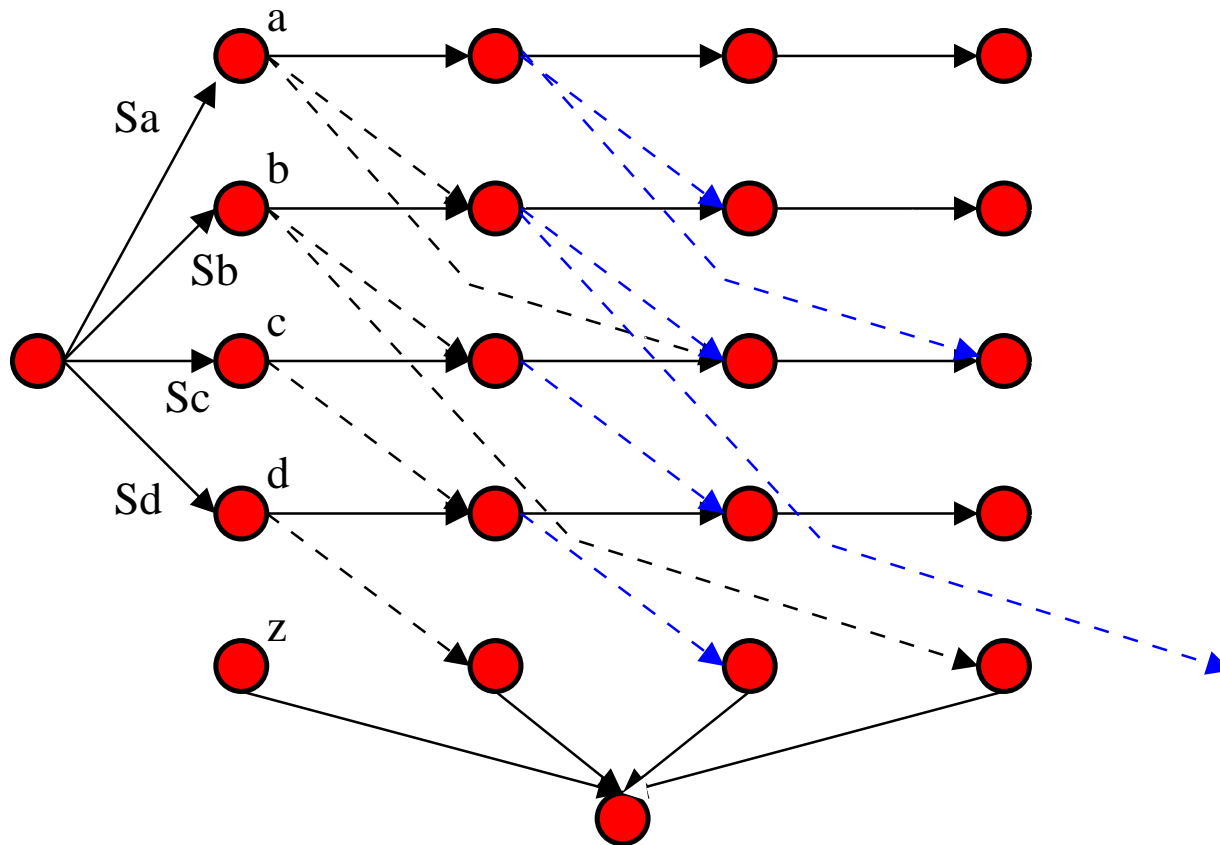
A shipping problem

- The previous graph is copied $T + 1$ times: one copy for each time step.
- Edges are arranged to match the time constraints.
- There is a super-source and a super-sink.



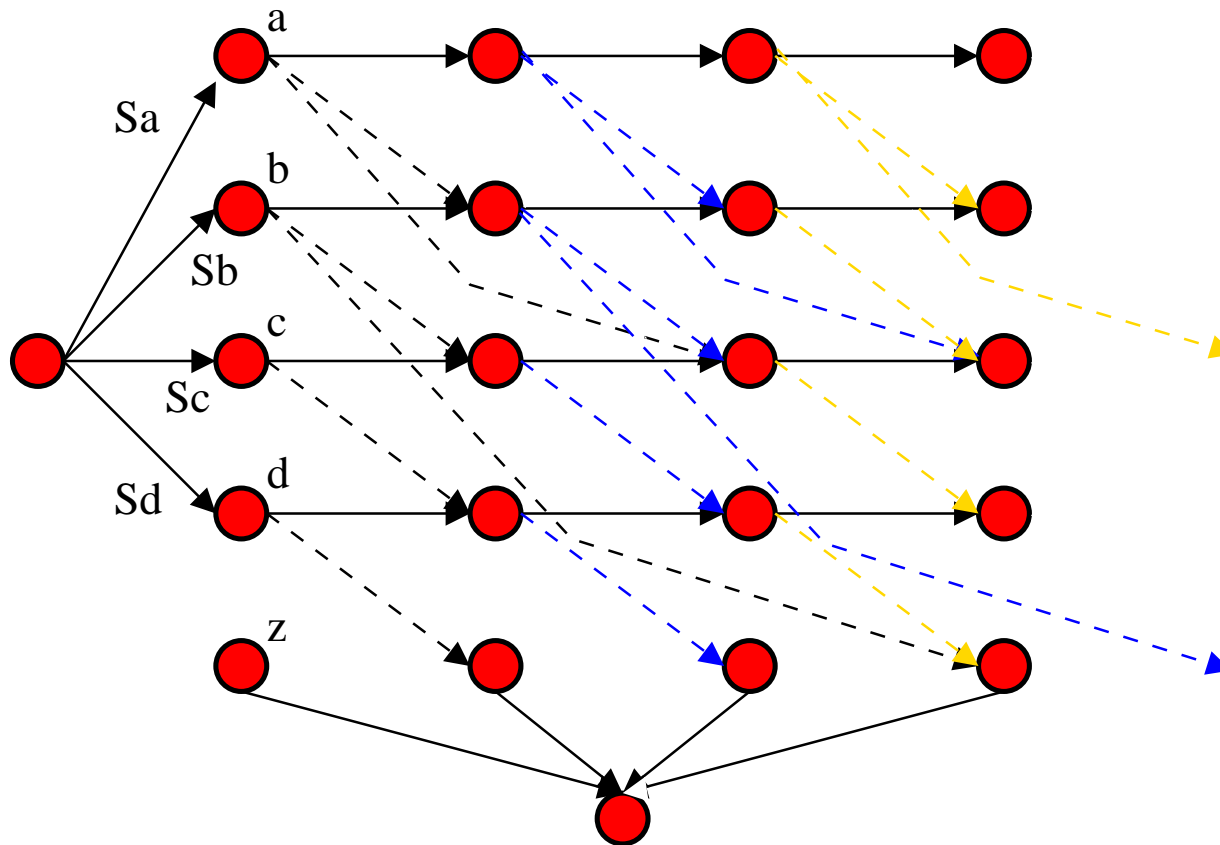
A shipping problem

- The previous graph is copied $T + 1$ times: one copy for each time step.
- Edges are arranged to match the time constraints.
- There is a super-source and a super-sink.



A shipping problem

- The previous graph is copied $T + 1$ times: one copy for each time step.
- Edges are arranged to match the time constraints.
- There is a super-source and a super-sink.



A shipping problem

- The previous graph is copied $T + 1$ times: one copy for each time step.
- Edges are arranged to match the time constraints.
- There is a super-source and a super-sink.

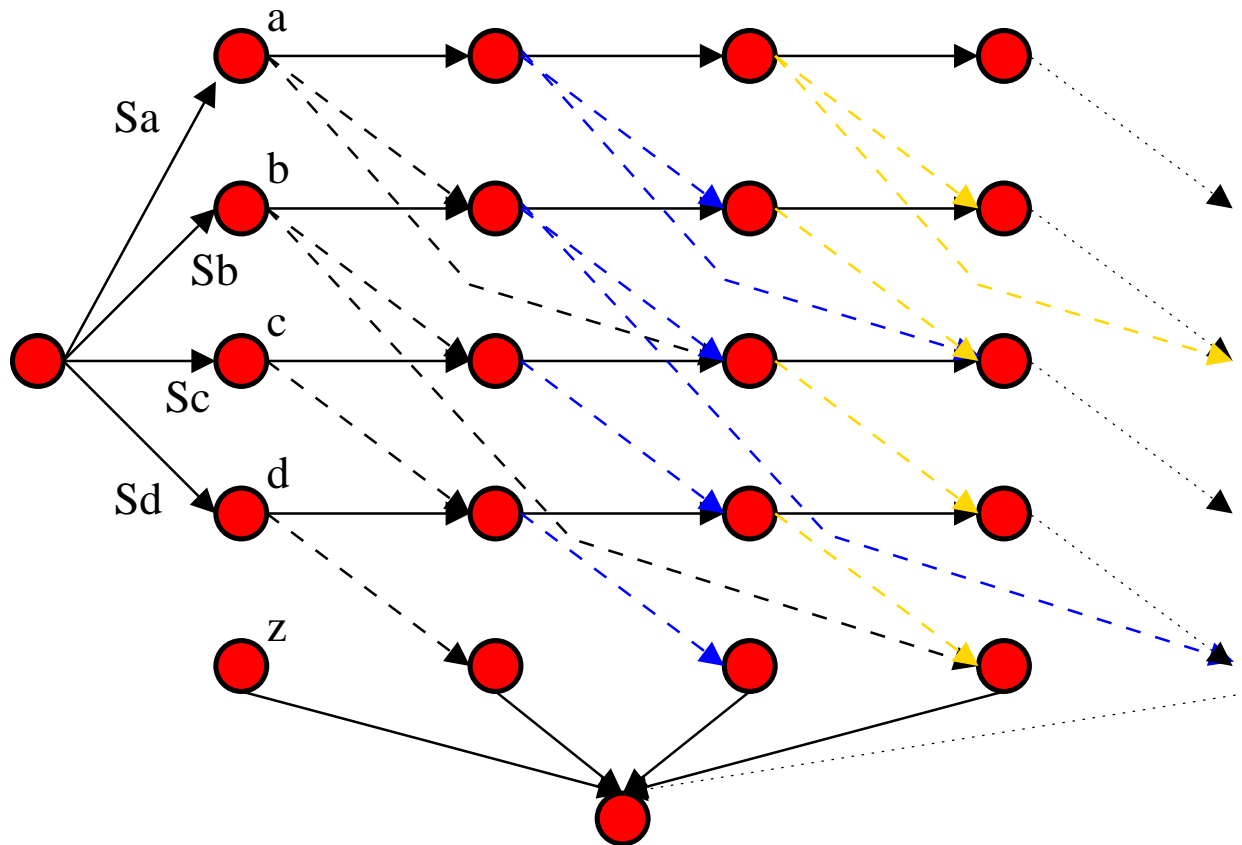


Image segmentation

Segmenting an image means separating it into **two** components: **background** and **foreground**.

Two kinds of constraints:

- **single-pixel** constraint: each pixel v in the image comes with integers f_v and b_v
 - a large f_v indicates that v should be in the **foreground**,
 - a large b_v indicates that v should be in the **background**.
- **connectivity** constraints: for each pair of adjacent pixels v, w , there is a penalty $p_{v,w}$ to pay if they are **not** in the same component.

Finding $f_v, b_v, p_{v,w}$ takes some work. Assuming we know them, we want to find F (**foreground**) and B (**background**) that maximizes

$$W(F, B) = \sum_{v \in F} f_v + \sum_{v \in B} b_v - \sum_{v \in F, w \in B, (v,w) \text{ connected}} p_{v,w}.$$

Making it a MinCut problem

Step 1: turning a max into a min.

Let $K = \sum_v f_v + \sum_v b_v$ (independent of the choice of F and B).

Remark that for any choice of F and B ,

$$K = \sum_{v \in F} f_v + \sum_{v \in B} f_v + \sum_{v \in F} b_v + \sum_{v \in B} b_v.$$

So

$$W(F, B) = K - \sum_{v \in B} f_v - \sum_{v \in F} b_v - \sum_{v \in F, w \in B, (v,w) \text{ connected}} p_{v,w}.$$

Since K does not depend on F and B , **maximizing** W is the same thing as **minimizing**

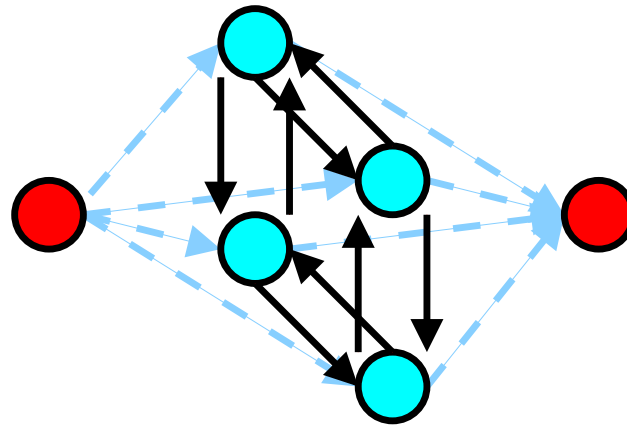
$$\sum_{v \in B} f_v + \sum_{v \in F} b_v + \sum_{v \in F, w \in B, (v,w) \text{ connected}} p_{v,w}.$$

Making it a MinCut problem

Step 2: setting the graph.

We let G be the following graph:

- the vertices of G are **all pixels**, plus a source s and a sink t ;
- the edges of G are:
 - for all v , an edge (s, v) , with capacity b_v ;
 - for all v , an edge (v, t) , with capacity f_v ;
 - for all neighbors v, w , edges (v, w) and (w, v) , with capacities $p_{v,w}$.



Making it a MinCut problem

Step 3: understanding the cuts.

A **cut** (U, V) in the graph G gives us a **partition** of the pixels:

- the background B is $U - \{s\}$,
- the foreground F is $V - \{t\}$.

What is its capacity? The edges $U \rightarrow V$ come into 3 categories:

- edges (v, t) , for v in B , **contributes f_v to the capacity**,
- edges (s, v) , for v in F , **contributes b_v to the capacity**,
- edges (v, w) , for v in B and w in F , **contributes $p_{v,w}$ to the capacity**.

The capacity of the cut is

$$\sum_{v \in B} f_v + \sum_{v \in F} b_v + \sum_{v \in B, w \in F, (v,w) \text{ connected}} p_{v,w},$$

so solving MinCut solves the problem.

Edmonds-Karp's algorithm

Edmonds-Karp vs Ford-Fulkerson

The strategy in Edmonds-Karp's algorithm refines that of Ford-Fulkerson:

- instead of choosing **any** path from s to t in the residual graph,
- choose a **shortest** path.

Remark: in the path-searching, this is done by a **breadth-first** search:

- the main data structures are the same,
- the set `ToDo` is implemented as a FIFO (**first in, first out**)

Disappearing edges

Setup. f is a flow and f' is obtained by an improvement step.

Prop. If (u, v) disappears in $G_{f'}$, then (u, v) is on the augmenting path.

Proof, 1. Suppose (u, v) was in **blue** in G_f . Then:

- (u, v) is in G because (u, v) is blue in G_f
- $f(u, v) < c(u, v)$ because (u, v) is blue in G_f
- $f'(u, v) = c(u, v)$ because (u, v) is not in $G_{f'}$

So the flow on (u, v) increased: (u, v) was on the augmenting path.

Proof, 2. Suppose (u, v) was in **red** in G_f . Then:

- (v, u) is in G because (u, v) is red in G_f
- $f(v, u) > 0$ because (u, v) is red in G_f
- $f(v, u) = 0$ because (u, v) is not in $G_{f'}$

So the flow on (v, u) decreased: (u, v) was on the augmenting path.

Appearing edges

Setup. f is a flow and f' is obtained by an improvement step.

Prop. If (u, v) appears in $G_{f'}$, then (v, u) is on the augmenting path.

Proof, 1. Suppose (u, v) appears in **blue** in $G_{f'}$. Then:

- (u, v) is in G because (u, v) is blue in $G_{f'}$
- $f'(u, v) < c(u, v)$ because (u, v) is blue in $G_{f'}$
- $f(u, v) = c(u, v)$ because (u, v) is not in G_f

So the flow on (u, v) decreased: (v, u) was on the augmenting path (in **red**).

Proof, 2. Suppose (u, v) appears in **red** in $G_{f'}$. Then:

- (v, u) is in G because (u, v) is red in $G_{f'}$
- $f'(v, u) > 0$ because (u, v) is red in $G_{f'}$
- $f(v, u) = 0$ because (u, v) is not in G_f

So the flow on (v, u) increased: (v, u) was on the augmenting path (in **blue**).

Distances

If H is a directed graph and x, y are in H , the **distance** $d(x, y)$ is the length of the shortest path from x to y .

Prop. For any x, y, z , if there is an edge $y \rightarrow z$, we have the **triangle inequality**

$$d(x, z) \leq d(x, y) + 1.$$

Proof.

- Let P be the shortest path $x \rightarrow y$; so, P has length $d(x, y)$.
- The path $P + (y, z)$ is a path $x \rightarrow z$ of length $d(x, y) + 1$.
- Hence, the shortest path $x \rightarrow z$ has length at most $d(x, y) + 1$.

Here, we consider path in the residual graphs G_f , where f is a flow. We write the distance of two vertices (x, y) in G_f by

$$d_f(x, y).$$

Distances increase

Setup. f is a flow and f' is obtained by a **shortest path** improvement step.

Prop. For any vertex v in G ,

$$d_f(s, v) \leq d_{f'}(s, v).$$

Distances to the source do not decrease after shortest path improvements.

Proof.

- We sort the vertices by **increasing distance** to the source in $G_{f'}$

$$d_{f'}(s, v_1) \leq d_{f'}(s, v_2) \leq \cdots \leq d_{f'}(s, v_n),$$

where v_1 is the source.

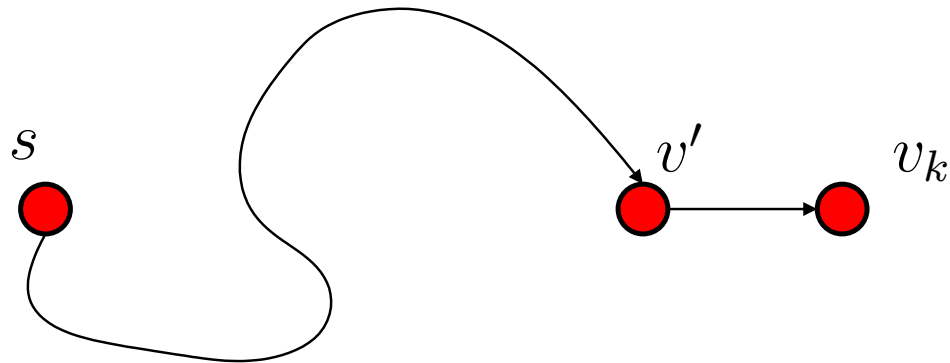
- Proof by induction.
 - We always have $d_f(s, s) = d_{f'}(s, s) = 0$, so the claim is true for $v_1 = s$.
 - We assume that the claim is true for v_1, \dots, v_{k-1} , and we prove it for v_k .

Induction step

Let P be a shortest path $s \rightarrow v_k$ in $G_{f'}$ and let v' be the vertex before v_k on P .

Because P is a shortest path, we have

$$d_{f'}(s, v_k) = d_{f'}(s, v') + 1. \quad (1)$$



In particular, $d_{f'}(s, v') < d_{f'}(s, v_k)$, so v' is one of v_1, \dots, v_{k-1} .

Let's say $v' = v_\ell$, for some $\ell < k$.

Induction assumption: $d_f(s, v') \leq d_{f'}(s, v')$.

Case discussion

Case 1. The edge (v', v_k) was already in G_f .

Then, by the triangle inequality, we have

$$d_f(s, v_k) \leq d_f(s, v') + 1.$$

By the induction assumption for v' , we get

$$d_f(s, v_k) \leq d_{f'}(s, v') + 1.$$

By equation (1), we get

$$d_f(s, v_k) \leq d_{f'}(s, v_k).$$

We are done!

Case discussion

Case 2. The edge (v', v_k) was not in G_f .

This means that (v', v_k) appeared when we changed f into f' .

We saw that this implies that (v_k, v') is on the augmenting path.

Since the augmenting path is a shortest path, we have

$$d_f(s, v') = d_f(s, v_k) + 1. \quad (2)$$

[like (1), but reversed]

Then, we get

$$d_f(s, v_k) = d_f(s, v') - 1 \leq d_{f'}(s, v') - 1 = d_{f'}(s, v_k) - 2 < d_{f'}(s, v_k),$$

and we are done!

Appearing and disappearing edges

Let n be the number of vertices in G_f .

Prop. In Edmonds-Karp's algorithm, each edge of G can **disappear** in G_f at most $n/2$ times.

Proof. Suppose that (u, v) disappears when switching from f_1 to f'_1 , and again (later) when switching from f_3 to f'_3 . In the meantime, it appeared, say when switching from f_2 to f'_2 .

- Since (u, v) disappeared during the transition $f_1 \rightarrow f'_1$, (u, v) was on the augmenting path. The augmenting path is a **shortest** path, so
$$d_{f_1}(s, v) = d_{f_1}(s, u) + 1.$$
- Since (u, v) appeared during the transition $f_2 \rightarrow f'_2$, (v, u) was on the augmenting path. The augmenting path is a **shortest** path, so
$$d_{f_2}(s, u) = d_{f_2}(s, v) + 1.$$

Appearing and disappearing edges

Putting these equalities together:

$$d_{f_3}(s, u) \geq d_{f_2}(s, u) = d_{f_2}(s, v) + 1 \geq d_{f_1}(s, v) + 1 = d_{f_1}(s, u) + 2.$$

all inequalities come from the fact that the distances to s are increasing

So between two disappearances, the distances increases at least by 2 .

But the distances are at most n .

So an edge can only disappear $n/2$ times.

Consequence. The **total** number of iterations in Edmonds-Karp's algorithm is $O(mn)$, where m is the number of edges.

Proof. At each iteration, an edge disappears in the residual graph.

Appearing and disappearing edges

Putting these equalities together:

$$d_{f_3}(s, u) \geq d_{f_2}(s, u) = d_{f_2}(s, v) + 1 \geq d_{f_1}(s, v) + 1 = d_{f_1}(s, u) + 2.$$

all inequalities come from the fact that the distances to s are increasing

So between two disappearances, the distances increases at least by 2 .

But the distances are at most n .

So an edge can only disappear $n/2$ times.

Consequence. The **total** number of iterations in Edmonds-Karp's algorithm is $O(mn)$, where m is the number of edges.

Proof. At each iteration, an edge disappears in the residual graph.

Consequence. The **cost** of Edmonds-Karp's algorithm is $O(m^2n)$.