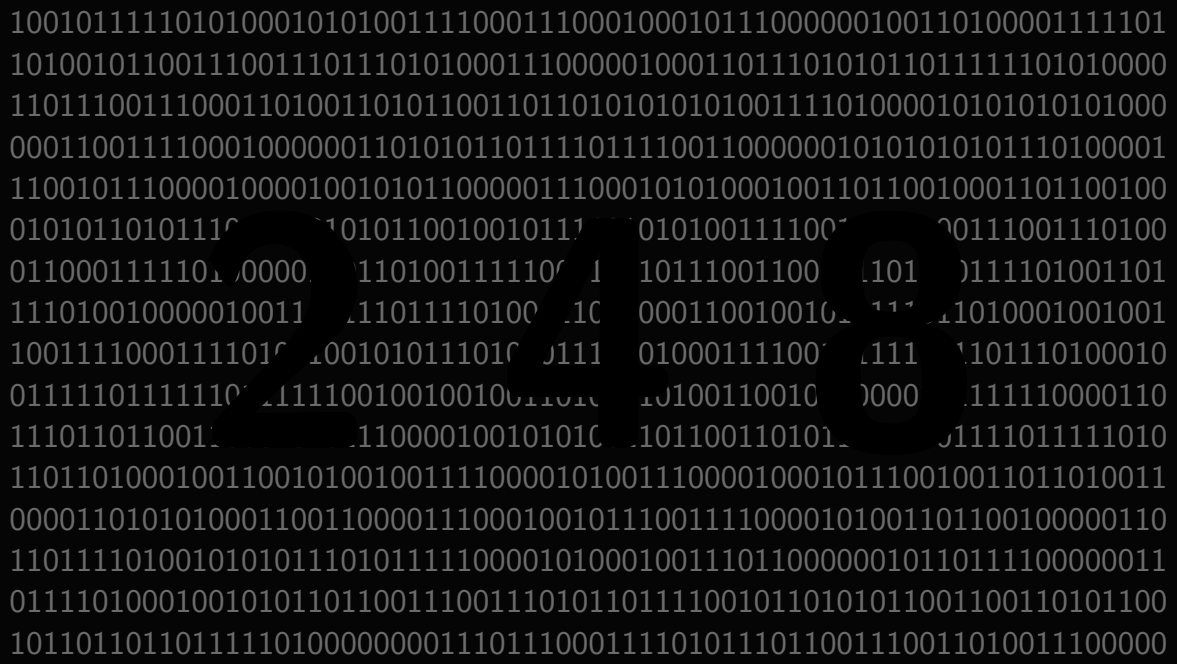
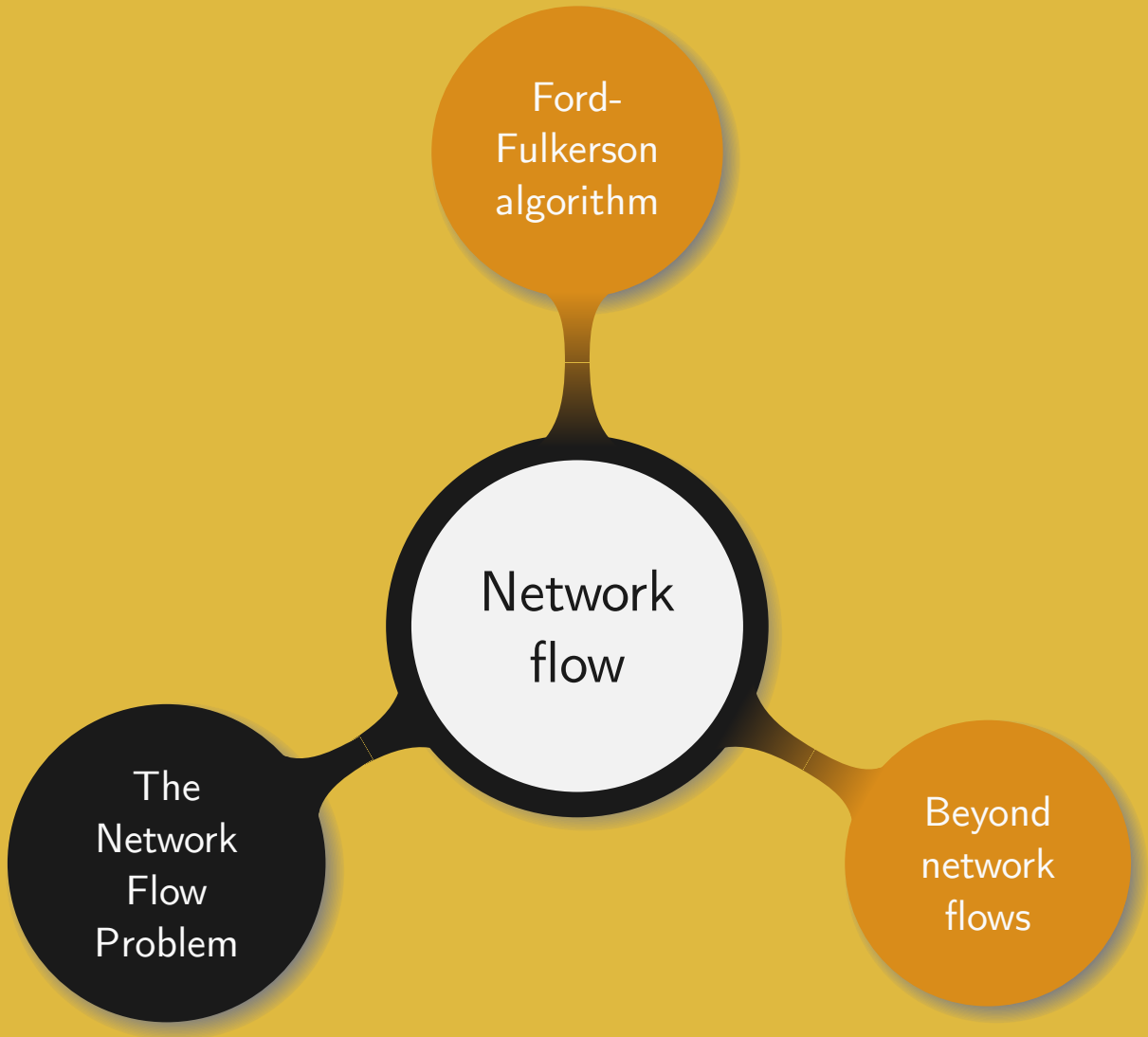


Introduction to Algorithms

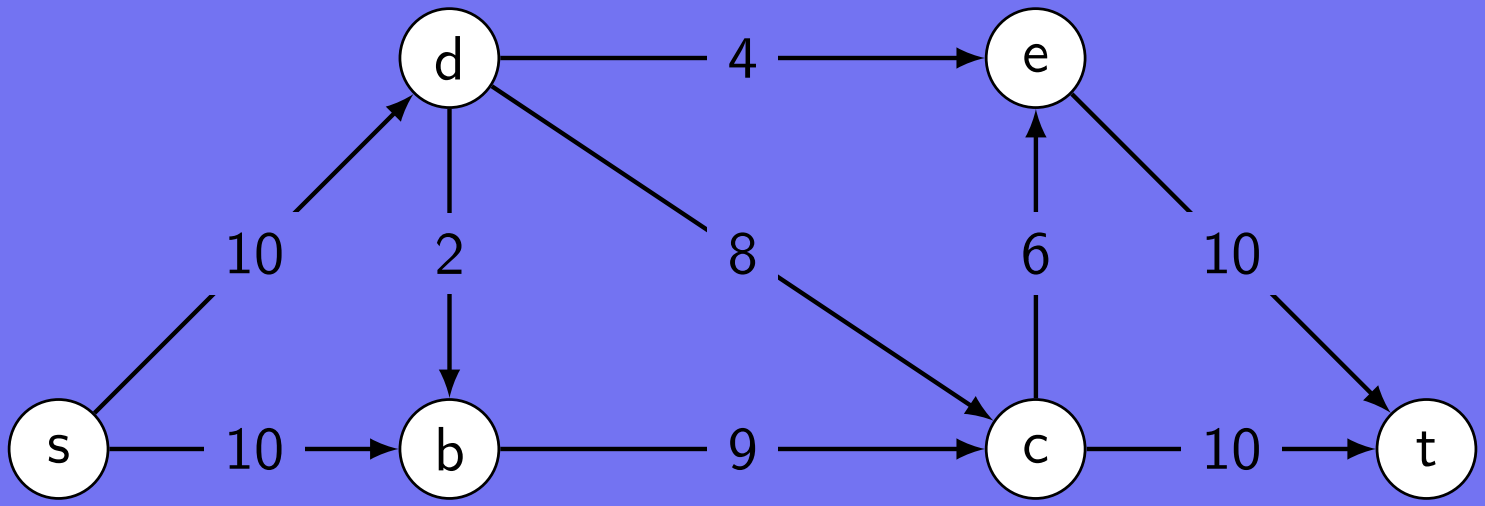
4. Network flow

Manuel – Fall 2021

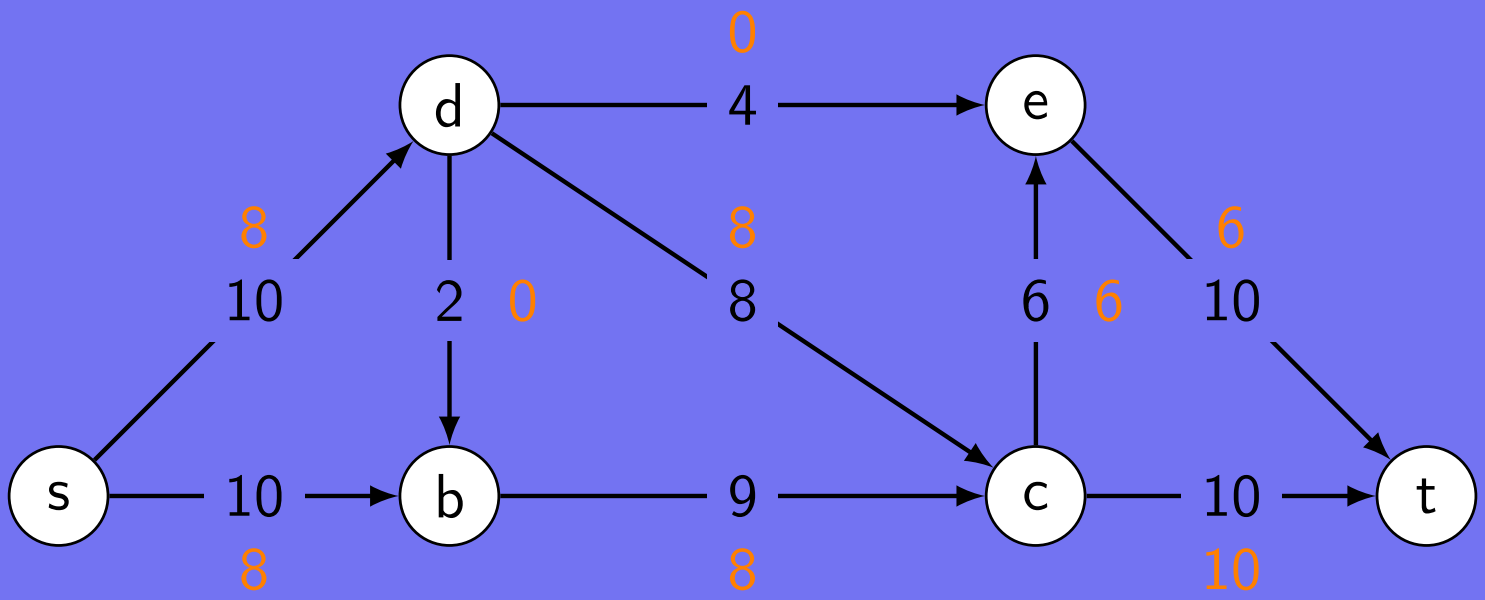
Chapter organisation



A toy example

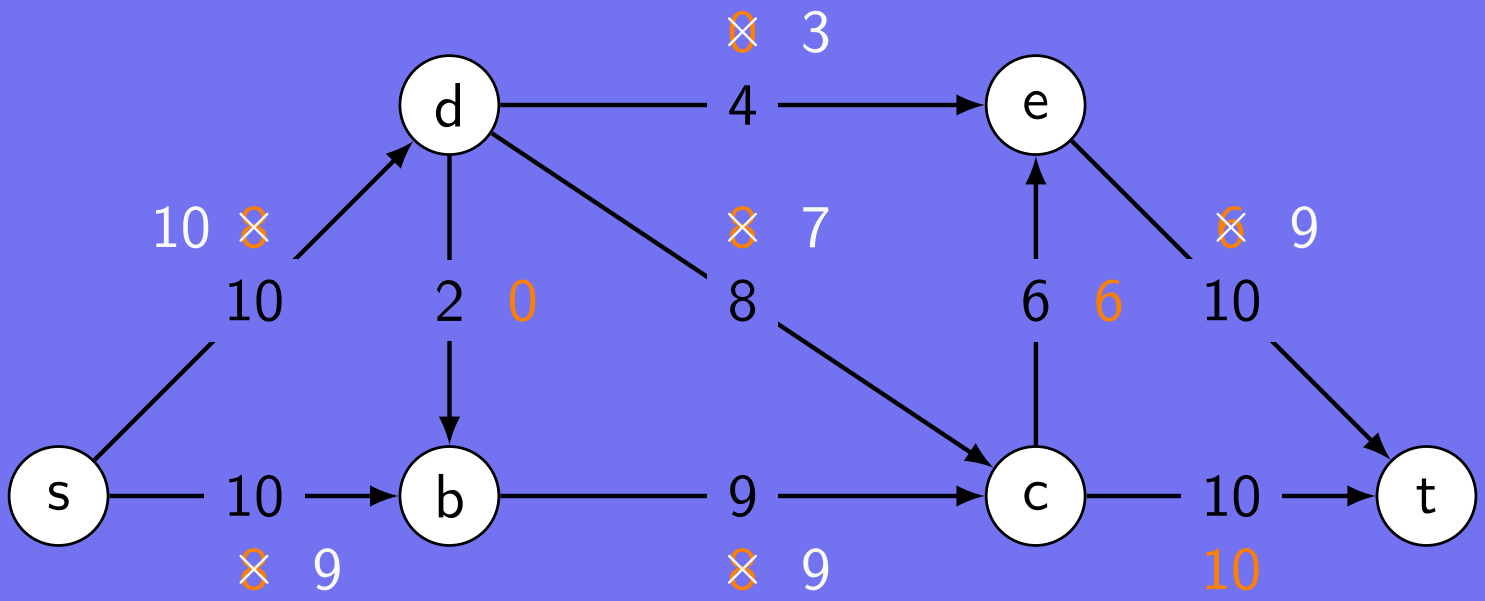


A toy example



Total flow: 16

A toy example



Total flow: 19

Informal approach

Let $G = \langle V, E \rangle$ be a directed graph. We consider each node of G as a switch and each edge as carrying some traffic. This is for instance the case in the context of a highway system: nodes are the interchanges (hubs) and the edges the highway itself. It could also be illustrated using a fluid network where the edges are the pipes and the nodes the junctures where the pipes are plugged together.

To each edge one associates a number called *capacity*, which represents how much traffic an edge can handle. In the maximum network flow problem the goal is to arrange the traffic such as optimizing the available capacity.

Before being able to solve this problem we need to formalize the idea of flow.

Definition

Let $G = \langle V, E \rangle$ be a weighted directed graph with a *source* node s and a *sink* node t; G is called a *flow network*.
Given a function $c : E \to \mathbb{R}^+$, called *capacity*, a *flow* is a function $f : E \to \mathbb{R}^+$, satisfying the following properties.

i. *Capacity constraint:* the flow of an edge can never exceed its capacity, i.e. $\forall (u, v) \in E,\ f(u, v) \leq c(u, v)$.

ii. *Flow conservation:* at each node the entering flow equals the exiting flow, i.e.

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

Definition

Let $G = \langle V, E \rangle$ be a weighted directed graph with a *source* node s and a *sink* node t; G is called a *flow network*.
Given a function $c : E \to \mathbb{R}^+$, called *capacity*, a *flow* is a function $f : E \to \mathbb{R}^+$, satisfying the following properties.

i. *Capacity constraint:* the flow of an edge can never exceed its capacity, i.e. $\forall (u, v) \in E,\ f(u, v) \leq c(u, v)$.

ii. *Flow conservation:* at each node the entering flow equals the exiting flow, i.e.

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

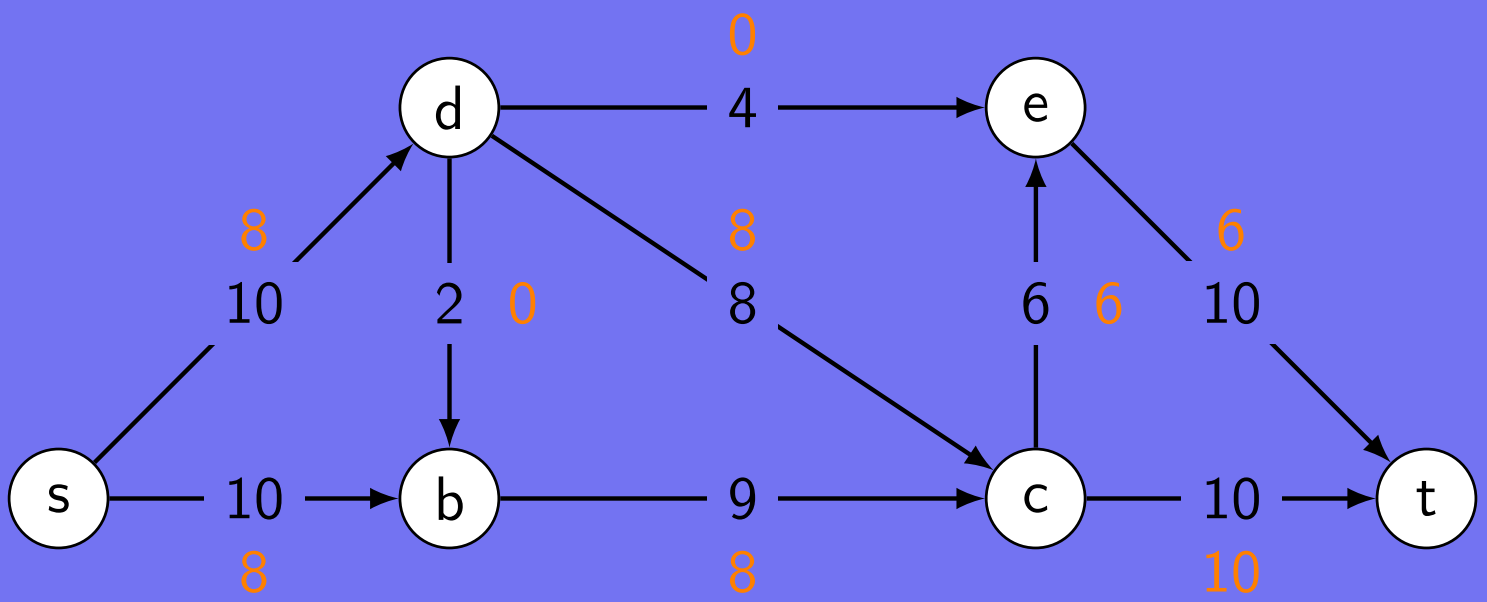
Remark. Unless specified otherwise, we restrict our attention to the case of integer or at least rational capacities and flows.

Problem (Maximum Network Flow)

Given a flow network arrange the flow such as to maximize the available capacity.

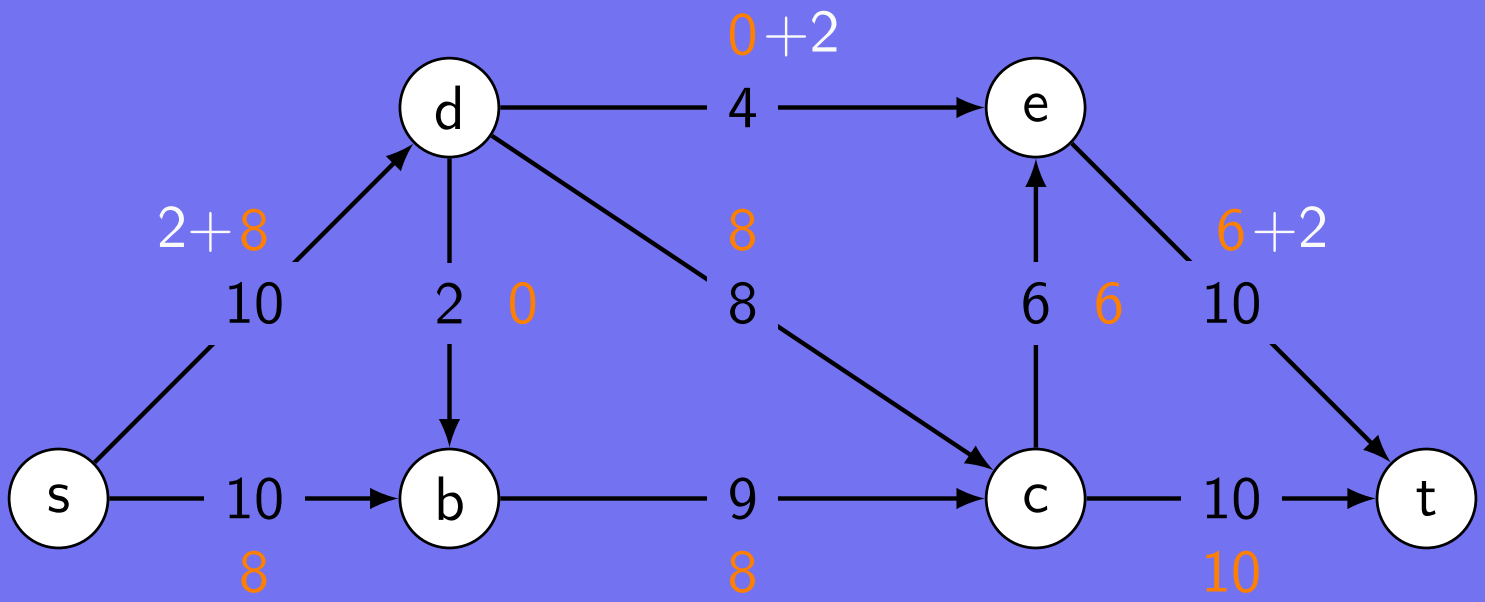
Remark. Assume a network flow is divided into two sets A and B of empty intersection and s is in A while t is in B. Intuitively any flow travelling from s to t must cross from A to B. This suggests that the capacity of the network flow can never exceed the capacity of the *cuts* A and B. The minimum capacity of any such division is called the *minimum cut*. We will prove that it is equal to the maximum flow value.

Back to the toy example



Improving the original path:

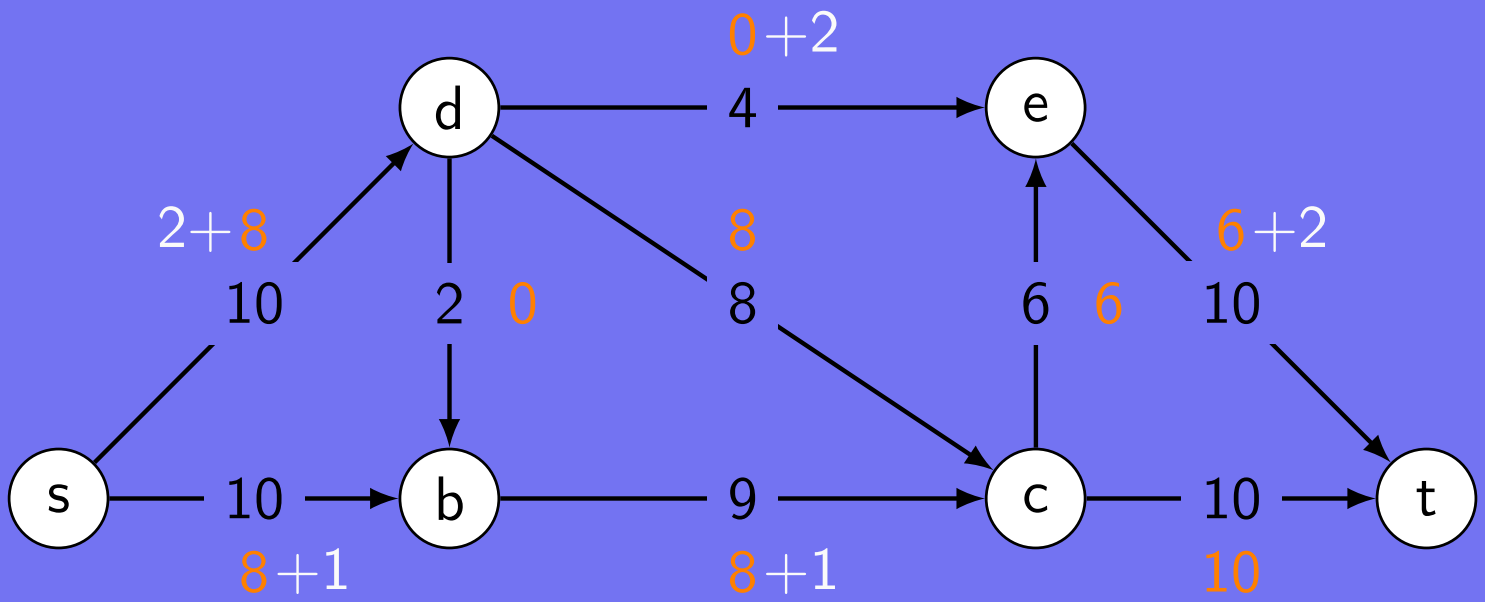
Back to the toy example



Improving the original path:

- Add $+2$ along $s \to d \to e \to t$; total flow: 18

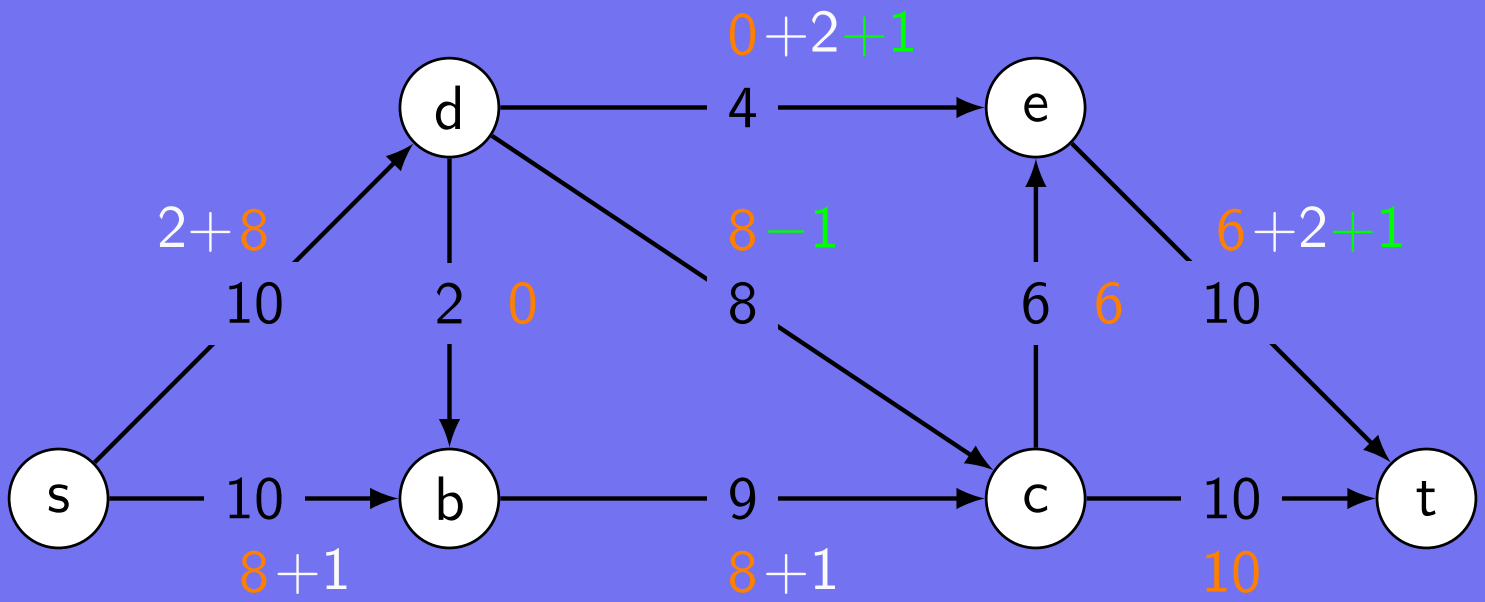
Back to the toy example



Improving the original path:

- Add $+2$ along $s \to d \to e \to t$; total flow: 18
- Add $+1$ along $s \to b \to c$: c is a bottleneck; total flow: 18

Back to the toy example



Improving the original path:

- Add $+2$ along $s \to d \to e \to t$; total flow: 18
- Add $+1$ along $s \to b \to c$: c is a bottleneck; total flow: 18
- Reallocate 1 from $d \to c \to t$ to $d \to e \to t$; total flow: 19

Informal strategy

Finding the maximum flow can be achieved as follows:

1. Start with a null flow
2. Find paths to increase the flow:
   - Path where the capacity has not been reached
   - Reallocate flow to a different path and increase the capacity of the current one
3. Stop when no more flow can be injected

Informal strategy

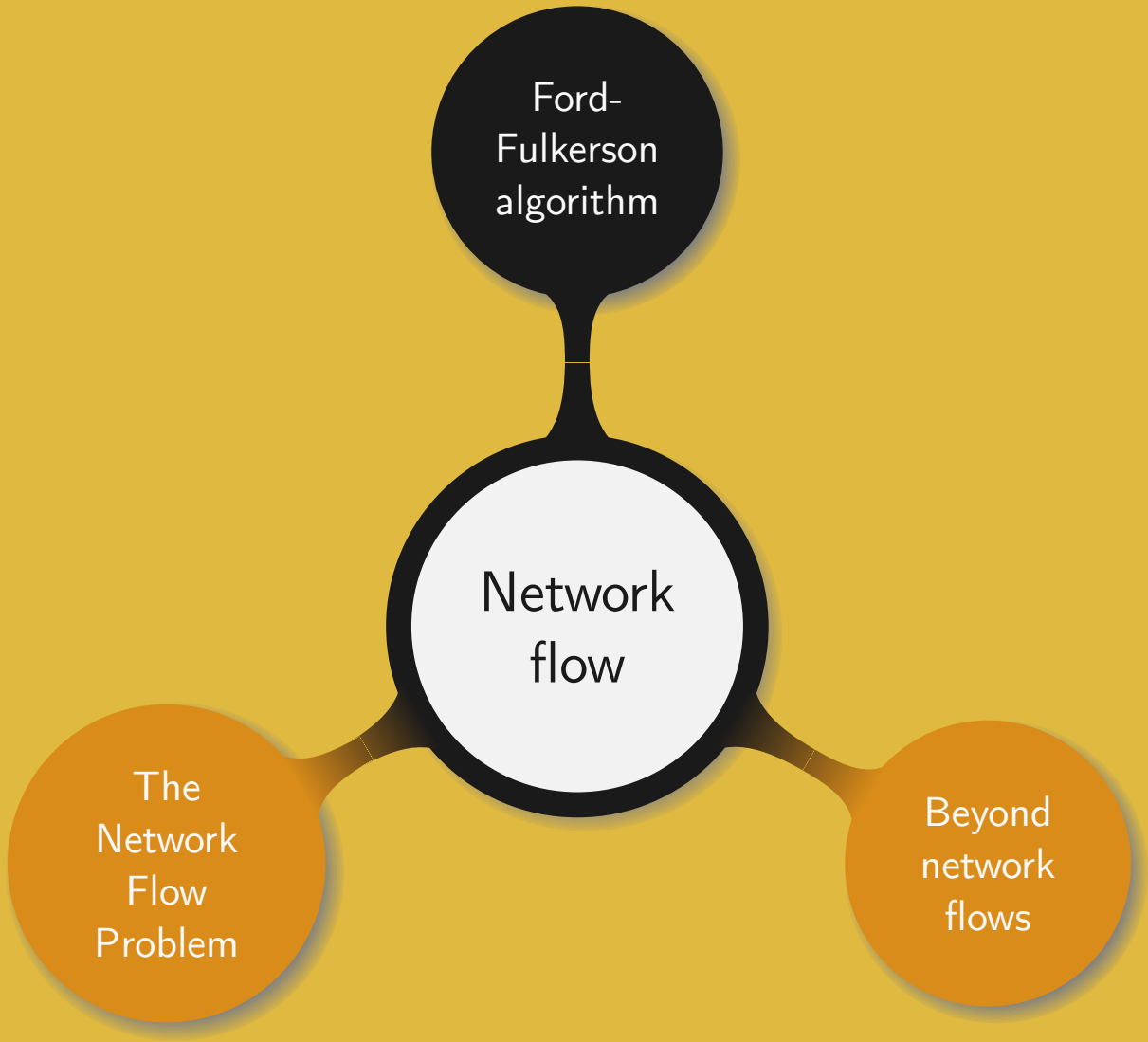
Finding the maximum flow can be achieved as follows:

1. Start with a null flow
2. Find paths to increase the flow:
   - Path where the capacity has not been reached
   - Reallocate flow to a different path and increase the capacity of the current one
3. Stop when no more flow can be injected

Goals:

- Determine all the paths that allow an increase of the flow
- Make sure the search for such paths stops at some stage

Chapter organisation



Residual graph

Definition (Residual graph)

Let G be a graph and f be a flow network. The *residual graph* G_f of G with respect to f is the graph whose vertices are the vertices of G; Regarding its edges:

- Each edge e of G with capacity $c(e)$ and flow $f(e) < c(e)$ is added to G_f with capacity $c(e) - f(e)$; it is a *forward edge*.
- For each edge $e = (u, v)$ of G with flow $f(e) > 0$, a *backward edge* $e' = (v, u)$ with capacity $c(e') = f(e)$ is added; it is a *backward edge*.

For a path P, the maximum amount by which the flow can be increased on each edge along P is called the *residual capacity*.

Residual graph

Definition (Residual graph)

Let G be a graph and f be a flow network. The *residual graph* G_f of G with respect to f is the graph whose vertices are the vertices of G; Regarding its edges:

- Each edge e of G with capacity $c(e)$ and flow $f(e) < c(e)$ is added to G_f with capacity $c(e) - f(e)$; it is a *forward edge*.
- For each edge $e = (u, v)$ of G with flow $f(e) > 0$, a *backward edge* $e' = (v, u)$ with capacity $c(e') = f(e)$ is added; it is a *backward edge*.

For a path P, the maximum amount by which the flow can be increased on each edge along P is called the *residual capacity*.

Remark. Given a graph G its residual graph G_f has at most twice the number of edges as G.

Augmenting paths

Algorithm. (*Augment*)

Input : a flow f and a simple path P
Output: a new flow

```
1 Function Augment(f, P):
2 |   b ← minimum residual capacity on P with respect to f;
3 |   foreach edge e ∈ P do
4 |   |   if e is a forward edge then f(e) ← f(e) + b;
5 |   |   else f(e) ← f(e) − b;
6 |   end foreach
7 |   return f
8 end
```

Lemma

The Augment algorithm (4.12) returns a flow in the original graph G.

Proof. We must ensure that the output of the algorithm matches definition 4.6, that is satisfies the capacity constraint and the flow conservation properties.

Consider the edges from the residual graph G_f whose capacity differ from the ones of G. In the case of e being such a forward edge with capacity $c(e) - f(e)$, we have

$$0 \leq f(e) \leq f(e) + b \leq f(e) + (c(e) - f(e)) = c(e).$$

If e is a backward edge then its residual capacity is $f(e)$ and

$$c(e) \geq f(e) \geq f(e) - b \geq f(e) - f(e) = 0.$$

Hence the capacity constraint holds whether e is a forward or backward edge.

Augmenting path

Intuitively the algorithm will return a flow since f itself is a flow. A more formal prove involves looking at the edges depending on whether they are backward or forward edges and compare the entering flow to the exiting one. □

Augmenting path

Intuitively the algorithm will return a flow since f itself is a flow. A more formal prove involves looking at the edges depending on whether they are backward or forward edges and compare the entering flow to the exiting one. □

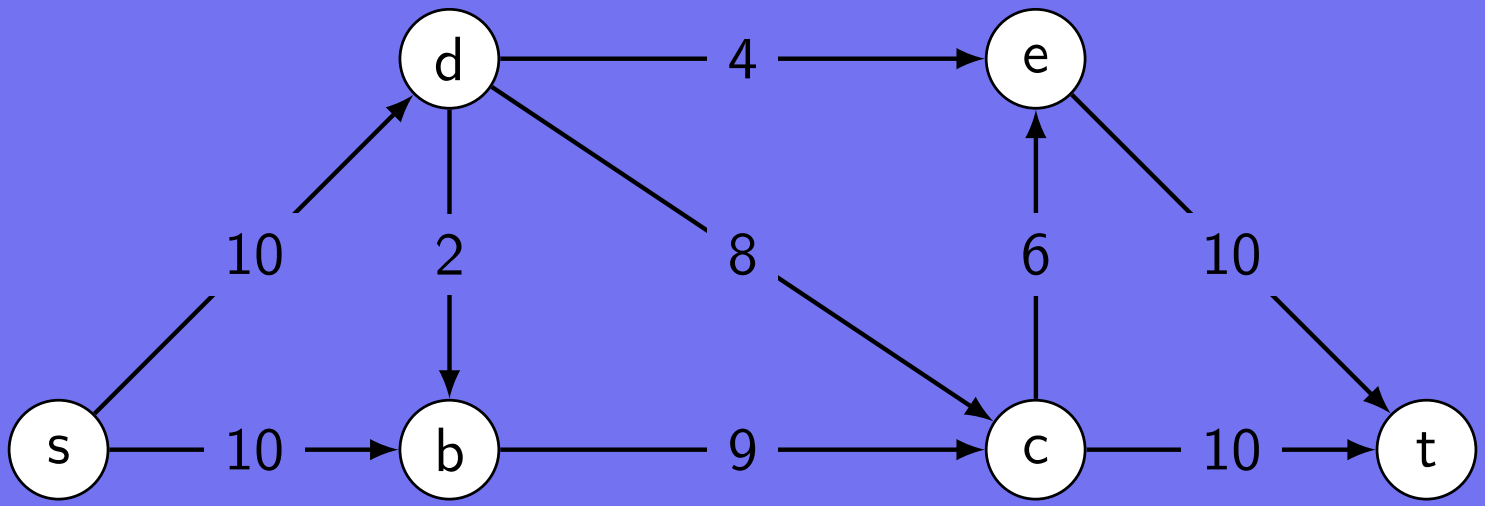
Algorithm. (*Ford-Fulkerson*)

Input : A graph G
Output : A maximum flow in G

```
1 foreach edge e in G do
2 |   f(e) ← 0;
3 end foreach
4 while there is a path P in the residual graph G_f do
5 |   f ← Augment(f, P);
6 |   update the residual graph G_f;
7 end while
8 return f
```

Exercise

Apply Ford-Fulkerson algorithm (4.14) to our initial toy example.



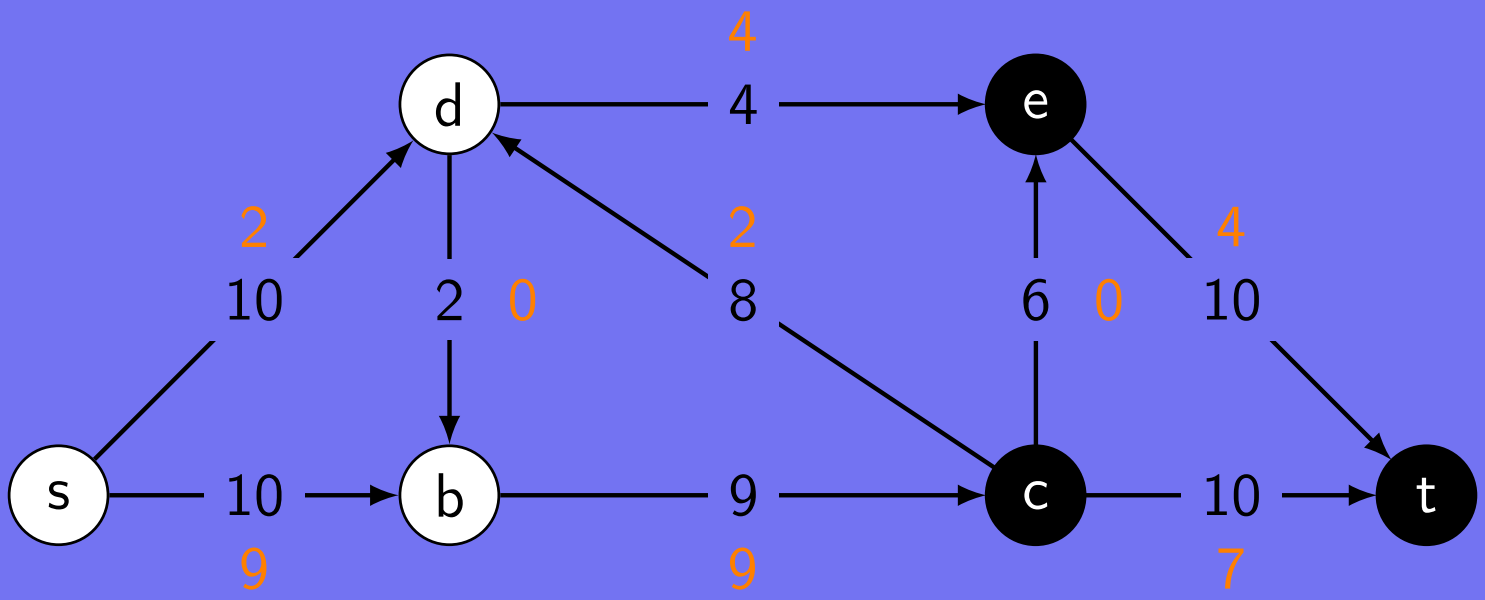
Cuts

Definitions

Let $G = \langle V, E \rangle$ be a flow network and s and t be the source and sink vertices, respectively.

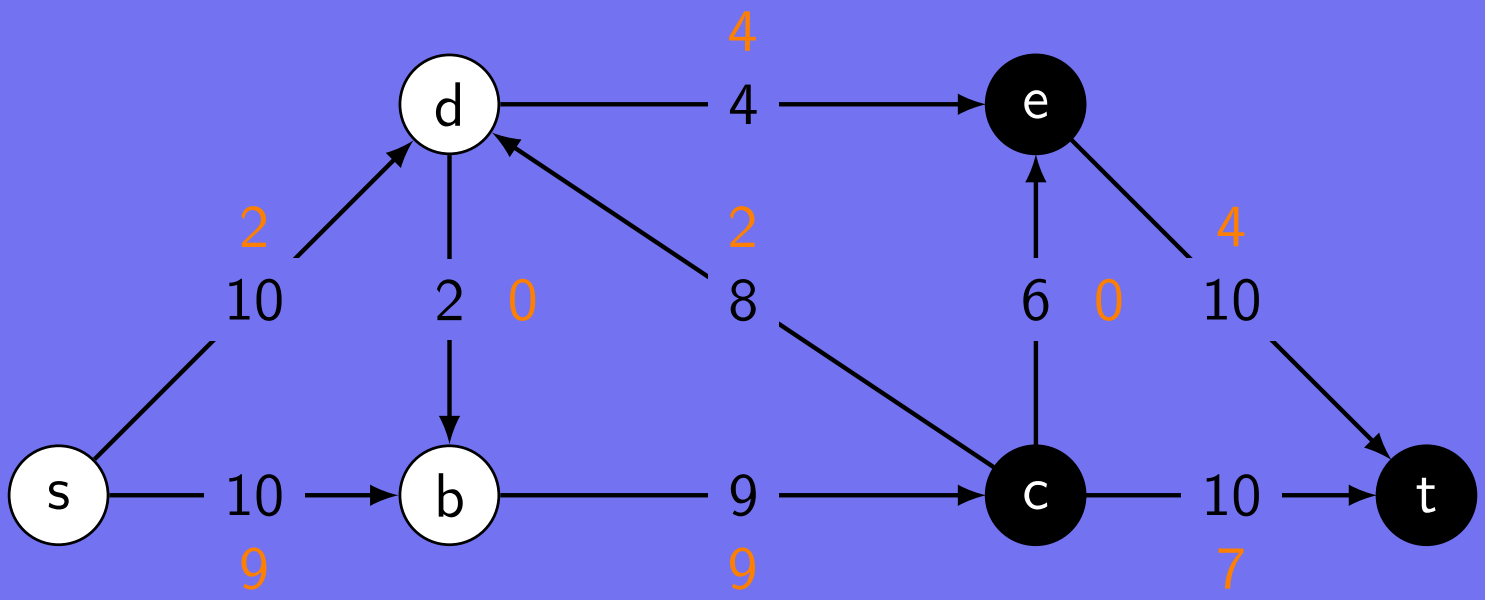
1. A *cut* is a partition of V into two connected subsets S and $T = V \setminus S$, with $s \in S$ and $t \in T$.
2. Given a flow f the *net flow* across the cut (S, T) is defined as
$$f(S, T) = \sum_{u \in S} \sum_{v \in T} f((u, v)) - \sum_{u \in S} \sum_{v \in T} f((v, u)).$$
3. The *capacity* of the cut (S, T) is defined as
$$c(S, T) = \sum_{u \in S} \sum_{v \in T} c((u, v)).$$
4. A *minimum cut* is a cut of minimum capacity over all the cuts of the network.

Example



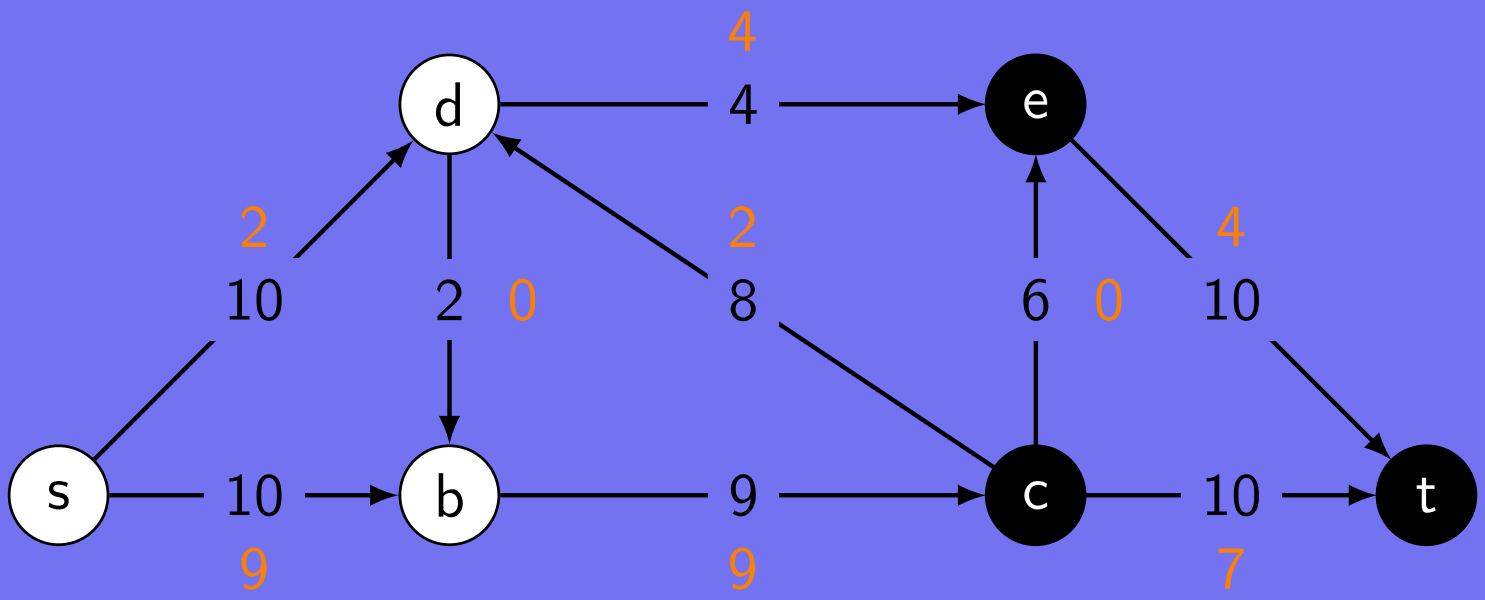
- Cut: $S = \{s, d, b\}$ and $T = \{c, e, t\}$
- Net flow:

Example



- Cut: $S = \{s, d, b\}$ and $T = \{c, e, t\}$
- Net flow: $4 + 9 - 2 = 11$
- Capacity of the cut:

Example



- Cut: $S = \{s, d, b\}$ and $T = \{c, e, t\}$
- Net flow: $4 + 9 - 2 = 11$
- Capacity of the cut: $9 + 4 = 13$

Net flow across cuts

Definition

Let $G = \langle V, E \rangle$ be a flow network, s be the source, and f be a flow on G. We denote the *value* of the flow f by

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

We will now prove that given a flow f the net flow across any cut is the same.

Lemma

Let f be a flow in a flow network $G = \langle V, E \rangle$ with source s and sink t. For any cut (S, T) of G, the net flow across (S, T) is $f(S, T) = |f|$.

Net flow across cuts

Proof. From the flow conservation (4.6) for any node $u \in V \setminus \{s, t\}$

$$\sum_{v \in V} f(u, v) - \sum_{v \in V} f(v, u) = 0.$$

Adding it to the value of the flow we get

$$|f| = \sum_{v \in V} f(s, v) - \sum_{v \in V} f(v, s) + \sum_{u \in S \setminus \{s\}} \left(\sum_{v \in V} f(u, v) - \sum_{v \in V} f(v, u) \right).$$

Rewriting the right-hand sum yields

$$|f| = \sum_{v \in V} \sum_{u \in S} f(u, v) - \sum_{v \in V} \sum_{u \in S} f(v, u).$$

Net flow across cuts

Noting that $S \cup T = V$ and $S \cap T = \emptyset$, the sum over the elements of V can be split over S and T to obtain

$$
\begin{aligned}
|f| &= \sum_{v \in S} \sum_{u \in S} f(u, v) + \sum_{v \in T} \sum_{u \in S} f(u, v) - \sum_{v \in S} \sum_{u \in S} f(v, u) - \sum_{v \in T} \sum_{u \in S} f(v, u) \\
&= \sum_{v \in T} \sum_{u \in S} f(u, v) - \sum_{v \in T} \sum_{u \in S} f(v, u) + \left(\sum_{v \in S} \sum_{u \in S} f(u, v) - \sum_{v \in S} \sum_{u \in S} f(v, u) \right) \\
&= \sum_{v \in T} \sum_{u \in S} f(u, v) - \sum_{v \in T} \sum_{u \in S} f(v, u) \\
&= f(S, T)
\end{aligned}
$$

Hence the net flow across the cut (S, T) is the value of the flow f. $\square$

Upper bound on a flow

Lemma

Given a flow network G and a flow f, f is upper bounded by the capacity of any cut in G.

Proof. Using lemma 4.18 and the capacity constraint (4.6) we have

$$\begin{aligned} |f| = f(S,T) &= \sum_{v \in T} \sum_{u \in S} f(u,v) - \sum_{v \in T} \sum_{u \in S} f(v,u) \\ &\leq \sum_{v \in T} \sum_{u \in S} f(u,v) \leq \sum_{v \in T} \sum_{u \in S} c(u,v) \\ &= c(S,T) \end{aligned}$$

□

Max-flow Min-cut Theorem

From the previous lemma (4.21) we can conclude that the maximum flow is upper bounded by the capacity of a minimum cut. We now prove that they are in fact equal.

Theorem (Max-flow Min-cut Theorem)

Let f be a flow in a flow network $G = \langle V, E \rangle$, with source s and sink t. Then following conditions are equivalent.

i. The flow f is a maximum flow in G.

ii. The residual network G_f contains no augmenting path.

iii. The value of the flow $|f|$ is equal to the capacity of some cut (S, T) of G.

Max-flow Min-cut Theorem

Proof. $(i) \Rightarrow (ii)$: we suppose that f is a maximum flow but G_f has an augmenting path. This generates a flow with value strictly larger than $|f|$, which contradicts lemma 4.21.↯

$(ii) \Rightarrow (iii)$: we suppose that G_f has no augmenting path, i.e. has no path from s to t. Let $S = \{v \in V \ / \text{ there is a path from } s \text{ to } v\}$, and $T = V \setminus S$.

Clearly (S, T) is a cut, since $s \in S$ and $t \notin S$ for otherwise there would be a path from s to t.

Let $u \in S$ and $v \in T$ be a pair of vertices.
If $e = (u, v)$ is in E then $f(e) = c(e)$, otherwise e would be in E_f and v would be in S.↯
If $e = (v, u)$ is in E then $f(e) = 0$, otherwise $c_f((u, v)) = f(e)$ would be positive, (u, v) would be in E_f, and v would be in S.↯

Max-flow Min-cut Theorem

For the last case, note that if neither (u, v) nor (v, u) belongs to E it means that $f((u, v)) = f((v, u)) = 0$.

Thus we have

$$\begin{aligned} f(S, T) &= \sum_{v \in T} \sum_{u \in S} f(u, v) - \sum_{v \in T} \sum_{u \in S} f(v, u) \\ &= \sum_{v \in T} \sum_{u \in S} c(u, v) - \sum_{v \in T} \sum_{u \in S} 0 \\ &= c(S, T) \end{aligned}$$

Therefore, by lemma 4.18, $|f| = c(S, T)$.

$(iii) \Rightarrow (i)$: combining lemma 4.21 with $|f| = c(S, T)$ yields (i). $\square$

Finding paths

While the Max-flow Min-cut theorem (4.22) provides a proof of accuracy for Ford-Fulkerson algorithm (4.14), the question of its efficiency remains unanswered.

In fact the algorithm still features an unclear request: "while there is a path P in the residual graph G_f". So the question boils down to knowing "how to determine all the paths and what is the cost".

A simple implementation of the algorithm would complete as the value of the flow keeps increasing by at least one unit while never exceeding the maximum flow f^*. Therefore in the worst case the complexity is $\mathcal{O}(|E||f^*|)$.

On the other hand if edges have a large capacity this strategy is not optimal and finding a "good path" becomes of a major importance. In practice this can be done using the breadth-first search algorithm which will allow us to determine the shortest path.

Algorithm. (*Breadth-First Search (BFS)*)

Input : a graph G and a starting vertex s
Output: none – only access all the vertices accessible from s

```
1  foreach vertex v in G do v.dist ← ∞; v.parent ← NULL;
2  Q ← ∅; s.dist ← 0; append s to Q;
3  while Q ≠ ∅ do
4  |   v ← first element in Q; remove v from Q;
5  |   foreach vertex u adjacent to v do
6  |   |   if u.dist = ∞ then
7  |   |   |   u.dist ← v.dist + 1; u.parent ← v; append u to Q;
8  |   |   end if
9  |   end foreach
10 end while
```

Theorem

Given a graph $G = \langle V, E \rangle$, the complexity of Breadth-First search is $\mathcal{O}(|E|)$.

Proof. The result is straight forward as the whole adjacency list is to be scanned.

□

We now prove that BFS correctly computes the shortest path distance between s and any vertex v.

Lemma

Let $G = \langle V, E \rangle$ be a graph and s be a source vertex in G. Upon running BFS on G, for any vertex v, the computed value $v.dist$ is larger than the shortest-path distance between s and v.

Proof. Let δ_v be the shortest distance between s and v. We will prove the result by induction on the number of "append" operations, the hypothesis being $v.dist \geq \delta_v$.

Base case: the first element to be appended to the list is s for which $\delta_s = s.dist = 0$; for all other vertices $\delta_v \leq v.dist = \infty$.

Shortest distance

Induction step: let v and u be two vertices where u is discovered from v. Noting that $u.dist = v.dist + 1$ we apply the induction hypothesis to v and get

$$\begin{aligned} u.dist &\geq \delta_v + 1 \\ &\geq \delta_u \end{aligned}$$

Noting that a vertex in never appended more than once, $u.dist$ will not be updated. Therefore the induction principle applies and for any vertex v in V, $v.dist$ is larger than the shortest-path distance between s and v. $\square$

We now want to prove that (i) nodes in Q are ordered with respect to their distance, and (ii) the first and last have a distance difference of at most one.

Lemma

Let $G = \langle V, E \rangle$ be a graph and s be a source vertex in G. Upon running BFS on G, Q contains the vertices $\{v_1, v_2, \cdots, v_r\}$ where v_1 and v_r are the head and tail, respectively. Then $v_r.dist \leq v_1.dist + 1$ and for $1 \leq i \leq r - 1$, $v_i.dist \leq v_{i+1}.dist$.

Proof. We prove the result by induction on the number of operations on Q, that is we take into account both "append" and "remove".

Base case: clearly the result holds when Q only contains s.

Induction step: two cases must be considered, when (i) removing and (ii) appending an element.

(i) When removing v_1 two cases can arise: either Q becomes empty and in this case the result holds vacuously, or the second element, v_2 becomes the head.

If v_2 is the new head then by the induction hypothesis $v_1.dist$ was less that $v_2.dist$. But as by induction $v_r.dist$ was less than $v_1.dist + 1$, we conclude that $v_r.dist \leq v_2.dist + 1$. All the remaining equalities being unaffected by the change of head, the result holds when removing an element.

(ii) When exploring the vertices adjacent to some vertex v, v has already been removed from Q. Let u be a neighboring vertex from v. Then u is appended to Q and can be renamed as v_{r+1}.
By the induction hypothesis $v.dist \leq v_1.dist$. Thus $v_{r+1}.dist$ is the same as $v.dist + 1$ which is less than $v_1.dist + 1$. Furthermore by the induction hypothesis $v_r.dist \leq v.dist + 1$, such that $v_r.dist$ is less than $v.dist + 1 = v_{r+1}.dist$. All the remaining equalities are unaffected.

Therefore by the induction principle lemma 4.30 holds. □

Theorem

Let $G = \langle V, E \rangle$ be a graph and s be a source vertex in G. Then BFS discovers all the vertices v in V reachable from s and upon termination for all of them $v.dist$ is the shortest distance δ_v.

Proof. Let $v \in V$ be a vertex such that $v.dist \neq \delta_v$. Then by lemma 4.28 $v.dist > \delta_v$. Clearly v must be reachable from s otherwise δ_v would be $\infty \geq v.dist$.
Let u be the vertex immediately preceding v on the shortest path to v. Then $\delta_v = \delta_u + 1$ and $u.dist = \delta_u$. Hence

$$v.dist > \delta_v = \delta_u + 1 = u.dist + 1 \tag{4.1}$$

The vertex v can be in three states: (i) neither in Q nor visited, (ii) in Q, and (iii) not in Q but visited.

Shortest distance

(i) When v is visited $v.dist$ is set to $u.dist + 1$, which contradicts equation (4.1).↯

(ii) If v is in Q then it means it was added during the exploration of the neighbors of a vertex w. Either $u = w$ or w was removed from Q earlier than u. So $v.dist = w.dist + 1$ and $w.dist \leq u.dist$. Therefore $v.dist = w.dist + 1 \leq u.dist + 1$, contradicting (4.1).↯

(iii) If v has already been removed from Q then clearly $v.dist < u.dist$, which once again contradicts (4.1).↯

Hence for all v in V, $v.dist = \delta_v$. Moreover all the vertices from V must be discovered otherwise if such a vertex v existed, then δ_v would be smaller than $v.dist = \infty$. □

The Edmonds-Karp algorithm

In order to evaluate the complexity of the Ford-Fulkerson algorithm (4.14), the question of how to find an augmenting path had to be answered.

We now prove that a shortest path in the residual network can be discovered using BFS and then used as an augmenting path. The resulting algorithm is called the Edmonds-Karp algorithm.

Lemma

Let $G = \langle V, E \rangle$ be a flow network with source s and sink t. If the Edmonds-Karp algorithm is run on G then for any vertex v in $V \setminus \{s, t\}$, the shortest-path distance $\delta_{f,v}$ in the residual network G_f increases monotonically with each flow augmentation.

Shortest path as augmenting path

Proof. Suppose that there exists a vertex $v \in V \setminus \{s, t\}$ such that a flow augmentation causes the shortest-path distance between s and v to decrease.
Let f and f' be the flows just before and after the augmentation that decreases the shortest-path distance, respectively. Then for v we have $\delta_{f',v} < \delta_{f,v}$. If u is the vertex visited just before v in $G_{f'}$ then $\delta_{f',u} = \delta_{f',v} - 1$.

From the choice of v, the shortest distance between s and u did not decrease on the flow augmentation. Therefore $\delta_{f,u} \leq \delta_{f',u}$ and (u, v) cannot belong to E_f. In fact if this was the case then we would have

$$\begin{aligned} \delta_{f,v} &\leq \delta_{f,u} + 1 \\ &\leq \delta_{f',u} + 1 \\ &= \delta_{f',v}. \end{aligned}$$

This contradicts the assumption $\delta_{f',v} < \delta_{f,v}$. ↯

Shortest path as augmenting path

Since (u, v) does not belong to E_f but then belongs to $E_{f'}$ it means that the flow has been increased from v to u. However as the Edmonds-Karps applies BFS to augment the flow along the shortest path we conclude that the shortest path from s to u has (v, u) as its last edge. Therefore,

$$\begin{aligned} \delta_{f,v} &= \delta_{f,u} - 1 \\ &\leq \delta_{f',u} - 1 \\ &= \delta_{f',v} - 2. \end{aligned}$$

Again this contradicts the assumption $\delta_{f',v} < \delta_{f,v}$.↯

Hence there exists no vertex $v \in V \setminus \{s, t\}$ such that a flow augmentation causes the shortest-path distance between s and v to decrease. □

Edmonds-Karp complexity

Theorem

Let $G = \langle V, E \rangle$ be a flow network with source s and sink t. If the Edmonds-Karp algorithm is run on G then it returns a maximum flow in time $\mathcal{O}(|V||E|^2)$.

Proof. Given a path P in the residual network G_f, we call an edge e such that $c_f(P) = c_f(e)$ a *critical edge*. We immediately notice that (i) such an edge disappears from the residual network as soon as an augmentation is performed along P, and (ii) at least one edge on any augmenting path is critical.

In order to determine the complexity of the Edmonds-Karp algorithm we must figure out how many flow augmentation are performed.

Edmonds-Karp complexity

Let u and v be two vertices and $e = (u, v)$ an edge in E. Since augmenting paths are shortest paths we have $\delta_{f,v} = \delta_{f,u} + 1$.

If e is a critical edge then it will disappear form the residual network as soon as the flow is augmented. It can however reappear later after the flow from u to v is decreased, that is if (v, u) is part of an augmenting path. The question is then to know how many times it can disappear and reappear.

Let f' be the flow in G when (v, u) appears in an augmenting path. Then $\delta_{f',u} = \delta_{f',v} + 1$. Moreover as $\delta_{f,v} \leq \delta_{f',v}$ (4.34) we get

$$\begin{aligned} \delta_{f',u} &= \delta_{f',v} + 1 \\ &\geq \delta_{f,v} + 1 \\ &= \delta_{f,u} + 2 \end{aligned}$$

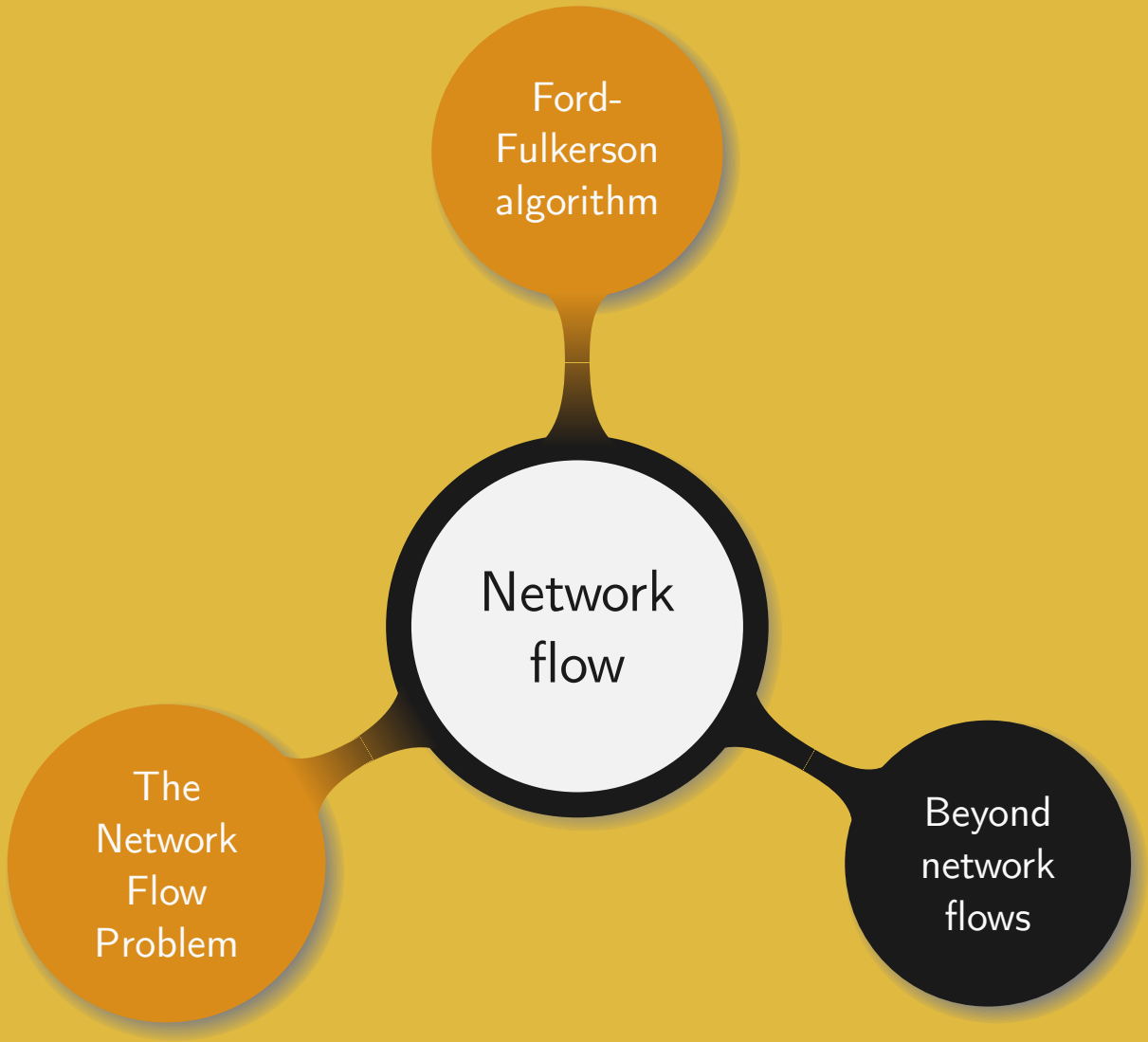
Thus, if e is a critical edge then the next time it becomes critical the distance between the s and u has increased by at least 2, while it originally was at least 1.

Then observing that the path from s to u cannot contain u, or t, we conclude that the distance will be at most $|V| - 2$.

Finally by combining the two bounds we conclude that an edge is critical no more than $|V|/2$ times. And as the number of edges in the residual network is $\mathcal{O}(|E|)$, the total number of critical edges during the execution of the algorithm is $\mathcal{O}(|V||E|)$.

Recalling that an augmenting path has at least one critical edge, we loop $\mathcal{O}(|V||E|)$ times in the Edmonds-Karp algorithm. For each of them, BFS with complexity $\mathcal{O}(|E|)$ (theorem 4.27) is run, leading to a total cost of $\mathcal{O}(|V||E|^2)$. □

Chapter organisation



General view on algorithms

When studying computability theory the importance of finding similarities between problems was highlighted (2.66). The idea behind this approach is to solve new problems by deriving appropriate algorithms from known ones.

The difficulty is therefore to view a problem from a different perspective. In practice this is similar to determining polynomial reductions in computability theory (2.49): one wants to efficiently "rephrase" a given problem into an new one for which a solution is known.

We now study such an example as we solve the maximum bipartite matching problem using network flows.

Maximum Bipartite Matching

Definition

Let $G = \langle V, E \rangle$ be a graph.

1. If $V = L \cup R$, with L and R two disjoint sets, every vertex in V has at least one incident edge, and for any edge (u, v) either $u \in L$ and $v \in R$ or $u \in R$ and $v \in L$, then G is called *bipartite graph*.
2. A *matching* is a subset M of E such that for any vertex $v \in V$ at most one edge in M is incident to v.
3. A *maximum matching* is a matching of maximum cardinality.

Maximum Bipartite Matching

Definition

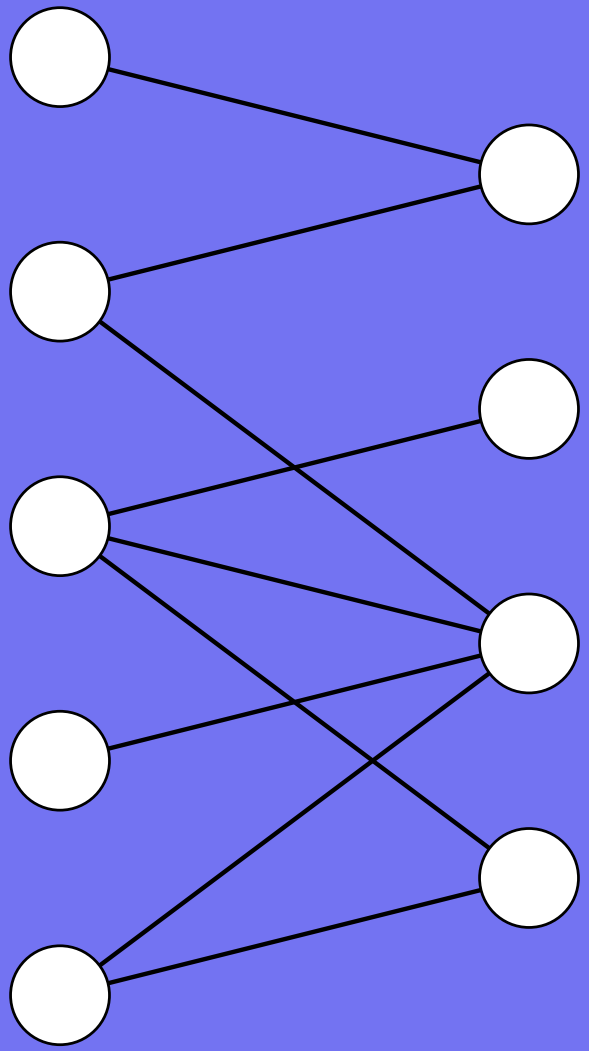
Let $G = \langle V, E \rangle$ be a graph.

1. If $V = L \cup R$, with L and R two disjoint sets, every vertex in V has at least one incident edge, and for any edge (u, v) either $u \in L$ and $v \in R$ or $u \in R$ and $v \in L$, then G is called *bipartite graph*.
2. A *matching* is a subset M of E such that for any vertex $v \in V$ at most one edge in M is incident to v.
3. A *maximum matching* is a matching of maximum cardinality.

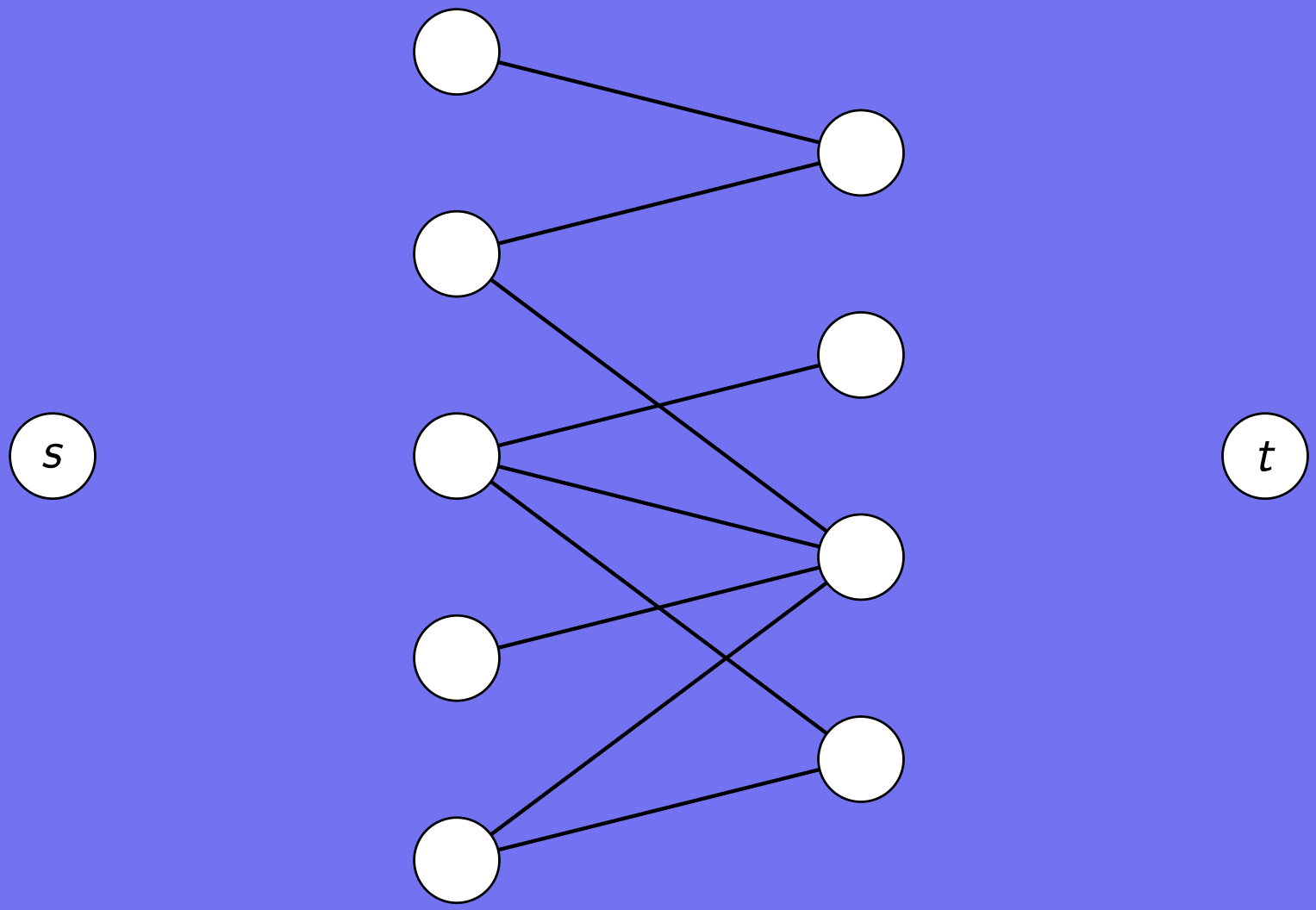
Problem (Maximum Bipartite Matching Problem)

Given a bipartite graph, determine a maximum matching.

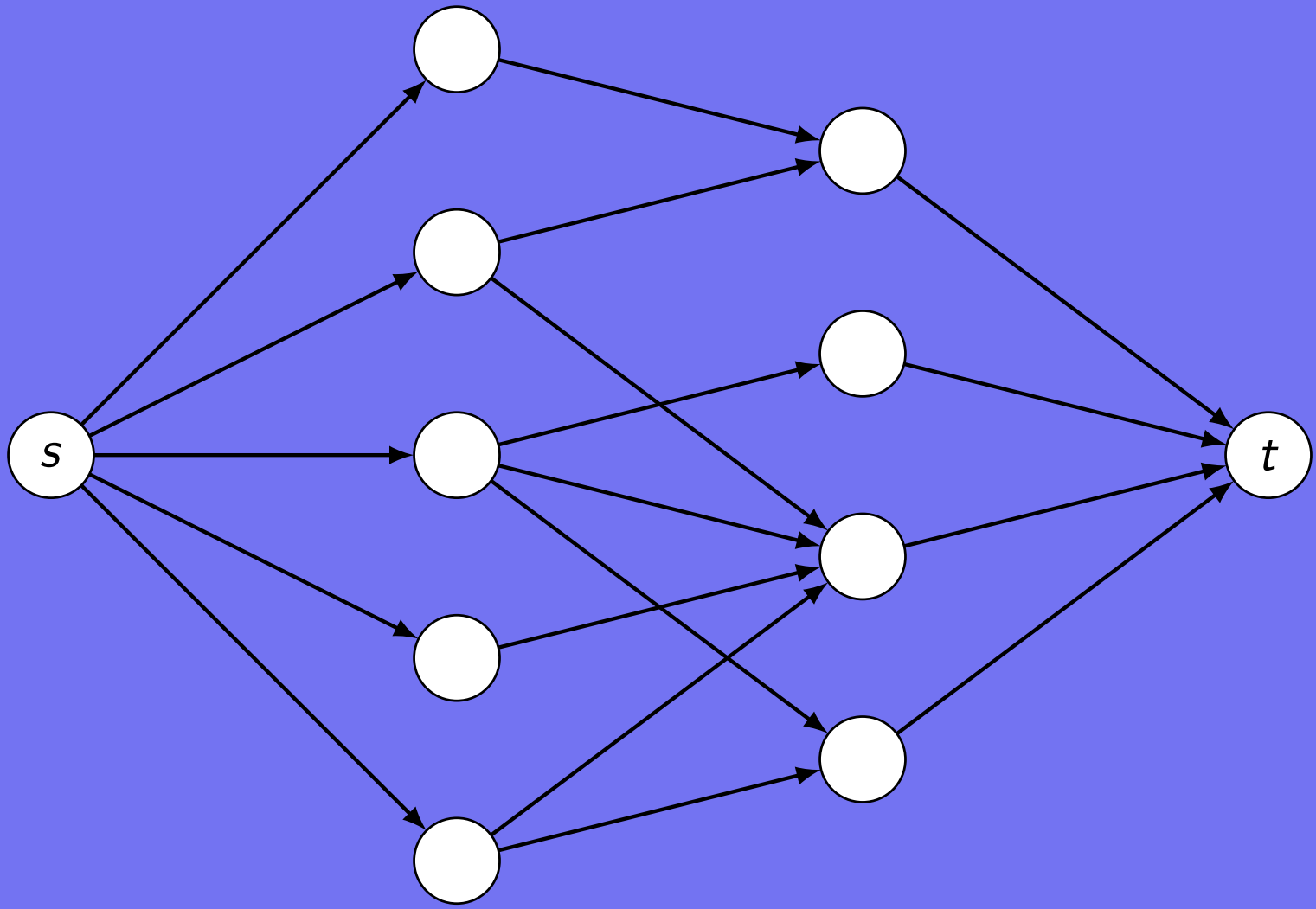
Example



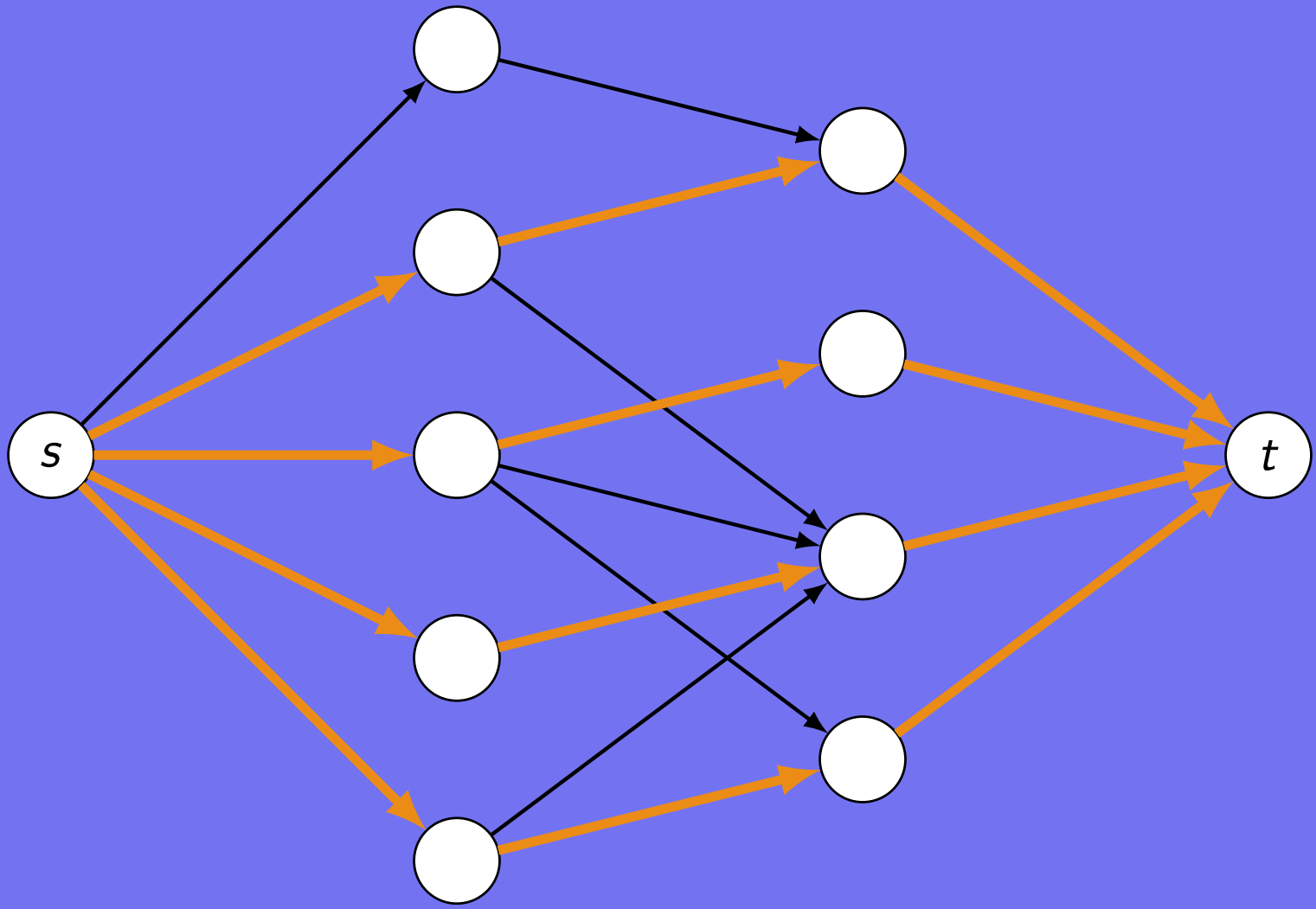
Example



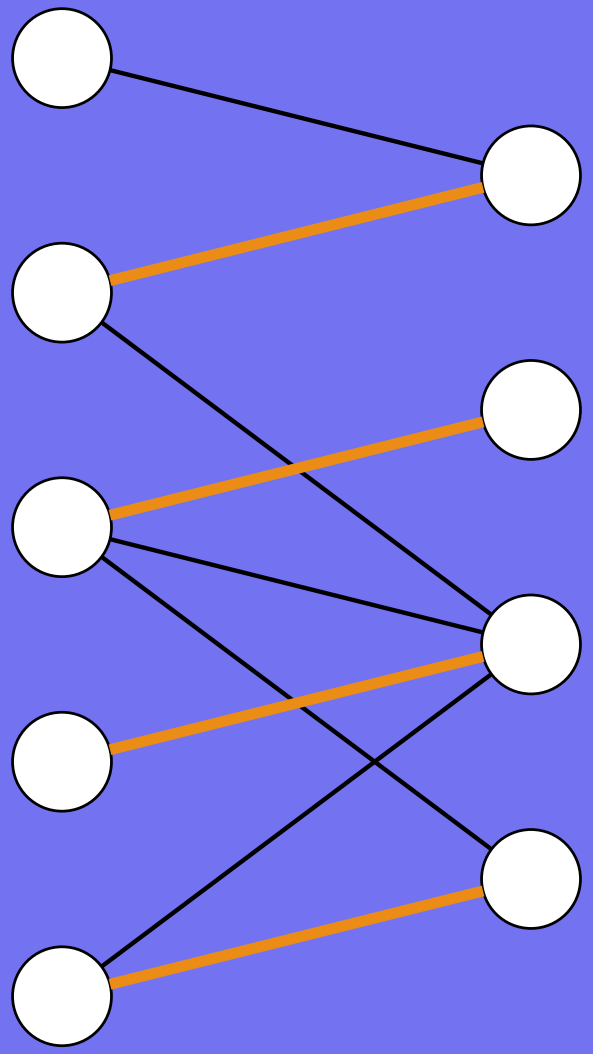
Example



Example



Example



Formalizing the transformation

To be able to apply Ford-Fulkerson method we formalize the transformation of a bipartite graph $G = \langle V, E \rangle$ into a network flow.

First define $G' = \langle V', E' \rangle$, with $V' = V \cup \{s, t\}$, s and t being the source and the sink, respectively. Since G is a bipartite graph V can be partitioned into two sets L and R, and

$$E' = \{(s, u) \ / \ u \in L\} \cup \{(u, v) \ / \ (u, v) \in V\} \cup \{(v, t) \ / \ v \in R\}.$$

It then suffices to assign a unit capacity to each edge in E'.

Formalizing the transformation

To be able to apply Ford-Fulkerson method we formalize the transformation of a bipartite graph $G = \langle V, E \rangle$ into a network flow.

First define $G' = \langle V', E' \rangle$, with $V' = V \cup \{s, t\}$, s and t being the source and the sink, respectively. Since G is a bipartite graph V can be partitioned into two sets L and R, and

$$E' = \{(s, u) \ / \ u \in L\} \cup \{(u, v) \ / \ (u, v) \in V\} \cup \{(v, t) \ / \ v \in R\}.$$

It then suffices to assign a unit capacity to each edge in E'.

At this stage an important step, that should not be omitted, is to check the cost of the transformation.
Since each vertex in V has at least one incident edge, $|E|$ is greater or equal to $|V|/2$. Thus the number of edges in G is smaller than the one in G' and $|E'| = |E| + |V| \leq 3|E|$. Hence $|E'| = \Theta(|E|)$, meaning that the transformation can be efficiently performed.

Lemma

Let $G = \langle V, E \rangle$ be a bipartite graph with vertex partition $V = L \cup R$, and $G' = \langle V', E' \rangle$ be its corresponding flow network. There is a matching M in G if and only if there exists a flow f in G'. In particular the value of the flow $|f|$ is equal to the cardinality of the matching $|M|$.

Proof. Let M be a matching in G. For an edge $e = (u, v)$ in E', define a flow f as $f(s, u) = f(u, v) = f(v, t) = 1$ if e is in M and 0 otherwise. Clearly f satisfies both the capacity constraint and the flow conservation properties (4.6).

Moreover as G is a bipartite graph a simple cut can be defined as $(S, T) = (L \cup \{s\}, R \cup \{t\})$. Observing that the net flow across the cut (S, T) is equal to $|M|$, we apply lemma 4.18 and get $|f| = |M|$.

To prove the converse define a flow f on G' and M to be

$$M = \left\{(u, v) \;/\; u \in L,\ v \in R, \text{ and } f(u, v) > 0\right\}.$$

By construction each vertex u in L has a single entering edge (s, u) with capacity at most 1. Therefore by the flow conservation property no more than 1 unit can leave on at most one edge. Thus a unit can enter u if and only if there is at most one v in R such that (u, v) is in M. Hence M is a matching.

Consequently for any matched vertex u in L, $f(s, u) = 1$, while $f(u, v) = 0$ for any edge in $E \setminus M$. This means that the net flow across cut the $(L \cup \{s\}, R \cup \{t\})$ is equal to $|M|$, which is exactly $|f|$, by lemma 4.18.

□

Remark. Over this whole chapter the net flow was implicitly assumed to only take integral values. However in the previous proof (4.45) it was explicitly mentioned that a unit flow cannot be split, which is true only in the case of integer valued flows.

In particular if the capacity function only takes integral values then the Ford-Fulkerson method returns an integer valued flow.

Remark. Over this whole chapter the net flow was implicitly assumed to only take integral values. However in the previous proof (4.45) it was explicitly mentioned that a unit flow cannot be split, which is true only in the case of integer valued flows.

In particular if the capacity function only takes integral values then the Ford-Fulkerson method returns an integer valued flow.

Theorem

Let $G = \langle V, E \rangle$ be a bipartite graph. The cardinality of a maximum matching M is equal to the value of the maximum flow in the flow network G' corresponding to G. This maximum matching is determined in time $\mathcal{O}(|V||E|)$.

Proof. Suppose M is a maximum matching in G while the corresponding flow in G' is not maximum. Then we can find a flow f' such that $\left|f'\right| > |f|$. But as noted in remark 4.47, both f and f' take integral values and there exists a matching M' corresponding to f'. Therefore we get

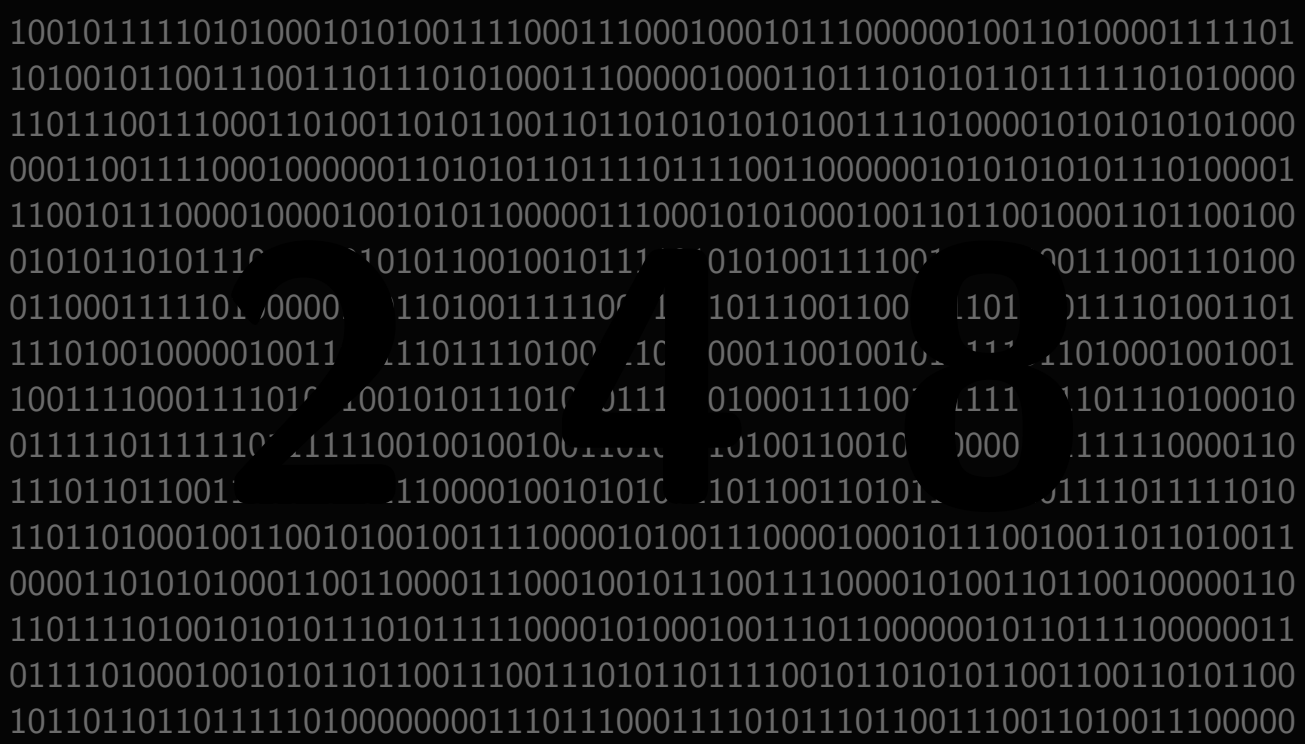
$$|M| = |f| < \left|f'\right| = \left|M'\right| .$$

Similarly if f is a maximum flow in G', then M is a maximum matching in G.

As any matching in a bipartite graph has cardinality at most $\min(|L|, |R|) = \mathcal{O}(|V|)$ the value of the maximum flow in G' is $\mathcal{O}(|V|)$. Therefore by 4.25 and 4.44 a maximum matching can be found in time $\mathcal{O}(|V|\left|E'\right|) = \mathcal{O}(|V||E|)$. □

Key points

- Explain the maximum flow problem
- How fast is the Ford-Fulkerson algorithm?
- How is breadth first search working?
- How to solve the maximum matching problem in a bipartite graph?



Thank you!