CS 225 – Data Structures

ZJUI – Spring 2022

Lecture 11: Graph Algorithms II

Klaus-Dieter Schewe

ZJU–UIUC Institute, Zhejiang University

International Campus, Haining, UIUC Building, B404

email: kd.schewe@intl.zju.edu.cn

11 Advanced Graph Algorithms

We continue exploring algorithm on graphs:

- **Maximum flow** algorithms are concerned with maximising the flow through a directed graph with a designated sourse and sink, where edge labels are to model the capacity of a connection

- **Graph colouring** algorithms are concerned with "colouring" the vertices of an undirected graph (i.e. labelling the edges with elements from a finite set C of "colours") using a minimum number of colours

- We will explore how to find **Euler** and **Hamilton** cycles, and generalise this to graphs, where such cycles do not exist

- As much as time allows, we may also look into algorithms to decide, whether two undirected graphs are **isomorphic**

- Another class of graph algorithms is concerning with **matching problems**, even when the graph is not bipartite—as there is not enough time, we refer to the literature

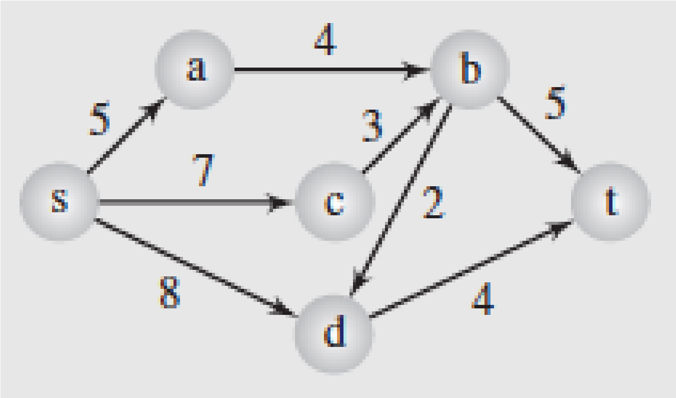
11.1 Maximum Flow Algorithms

A **network** is a directed graph $G = (V, E)$ with two designated vertices $s \in V$ without incoming edges and $t \in V$ without outgoing edges—we call s the **source** and t the **sink**

Assume that each vertex lies on a path from the source s to the sink t

Further assume a labelling of the edges by positive real numbers, i.e. $c : E \to \mathbb{R}_+$—we call $c(e) \in \mathbb{R}_+$ the **capacity** of the edge e

Example:



For instances, the vertices may represent pump stations and the edges may be water tubes or oil/gas pipelines

Maximum Flow Problem

A **flow** in a network (G, c) (as defined above) is a function $f : E \to \mathbb{R}$ satisfying the following two conditions:

(i) Capacity constraint. The flow through any edge cannot exceed the capacity of the edge, i.e.

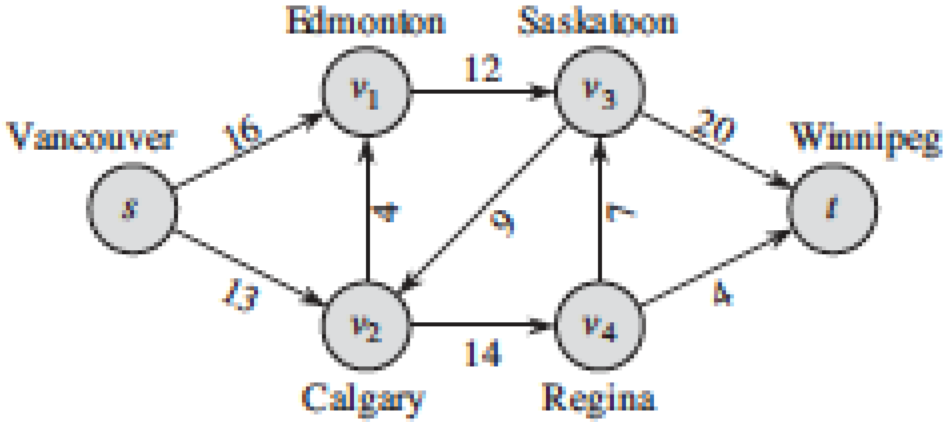
$$0 \le f(e) \le c(e) \text{ holds for all edges } e \in E$$

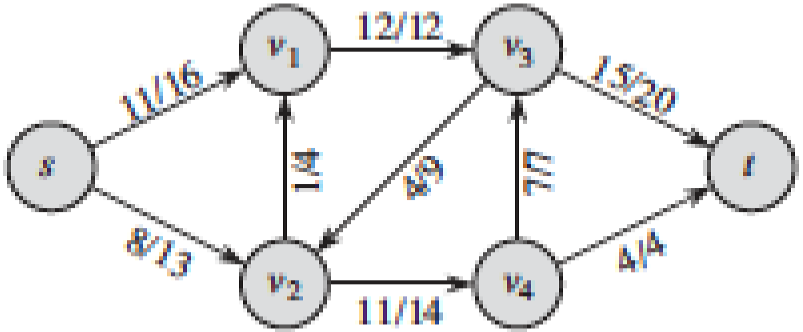
(ii) Flow conservation. For every vertex $v \in V$ except the source and the sink the incoming flow must be equal to the outgoing flow, i.e.

$$\sum_{(w,v)\in E} f(w,v) = \sum_{(v,w)\in E} f(v,w) \text{ holds for all vertices } v \in V - \{s,t\}$$

The **maximum flow problem** is to find for a given network (G, c) a flow f such that the flow value $|f| = \sum_{(v,t)\in E} f(v,t)$ to the sink is maximal

Example





The graph on the left shows a network with vertices corresponding to some Canadian cities

The graph on the right shows a flow f with flow value $|f| = 19$; each edge e is labelled by $f(e)/c(e)$

The maximum possible flow would have a flow value 23

Anti-Parallel Edges

In networks we may assume that there are neither loops (v, v) nor **anti-parallel** edges, i.e. edges (v, w) and (w, v) for $v \neq w$

That is, we request the condition $(v, w) \in E \Rightarrow (w, v) \notin E$ for all $v \in V$

This is no loss of generality: if there are anti-parallel edges $(v, w) \in E$ and $(w, v) \in E$, we can replace (w, v) be two edges (w, v') and (v', v) with a new vertex v'

In addition, extend the capacity function c to the new edges by

$$c(w, v') \ = \ c(w, v) \quad \text{and} \quad c(v', v) \ = \ c(w, v)$$

Multiple Sources and Sinks

The assumption that there is a unique source and a unique sink can also be relaxed

If there are multiple sources $s_1, \dots, s_n$, then simply add a **supersouce** s together with new edges (s, s_i) for all $1 \le i \le n$ to the network

Extend the capacity function c to the new edges by $c(s, s_i) = c_{\max}$ with $c_{\max} \ge \sum_{e \in E} c(e)$

In the same way, if there are multiple sinks $t_1, \dots, t_m$, then add a **supersink** t together with new edges (t_i, t) for all $1 \le i \le m$ to the network

Extend the capacity function c to the new edges by $c(t_i, t) = c_{\max}$

In this way the maximum flow problem in presence of multiple sources $s_1, \dots, s_n$ and multiple sinks $t_1, \dots, t_m$ is reduced to the maximum flow problem with the unique (super)source s and the unique (super)sink t

The Ford-Fulkerson Method

The Ford-Fulkerson method solves the maximum flow problem by initialising the flow to $f(e) = 0$ for all edges $e \in E$, thus $|f| = 0$, and iteratively increasing the flow

The key ingredients are **cuts**, **residual networks** and **augmenting paths**

In each iteration step the flow f is increased along some augmenting path p in the residual network G_f, as long as such a path p exists

While the flow value $|f|$ is increased in every step, the individual values $f(e)$ may also decrease, which corresponds to redirection part of a flow to a different path

The correctness of the method follows then from the **max-flow min-cut theorem**

Residual Capacity

Suppose $G = (V, E)$ is a network with capacity function $c : E \to \mathbb{R}_+$, and let $f : E \to \mathbb{R}_+$ be a flow in (G, c)

The **residual capacity** defined by G, c and f is a function $c_f : V \times V \to \mathbb{R}$ with

$$c_f(v, w) = \begin{cases} c(v, w) - f(v, w) & \text{if } (v, w) \in E \\ f(w, v) & \text{if } (w, v) \in E \\ 0 & \text{else} \end{cases}$$

As we request that $(v, w) \in E$ implies $(w, v) \notin E$, the residual capacity is well defined

The first case expresses how much capacity is still left that might be used to increase the flow value, the second case expresses how much of the flow could be redirected

The edges appearing in the two cases are there called **forward** and **backward** edges, respectively

Residual Networks

The **residual network** defined by the network G with capacity c and the flow f is the graph $G_f = (V, E_f)$ with $E_f = \{(v, w) \in V \times V \mid c_f(v, w) > 0\}$

Note that $|E_f| \leq 2|E|$, and that E_f may contain both forward and backward edges (v, w) and (w, v)

As residual networks can be easily turned into networks with capacity function c_f, we can use the notion of flow in a residual network

If f is a flow in the network (G, c) and f' is a flow in the residual network (G_f, c_f), then the **augmentation of f by f'** (notation: $f \uparrow f'$) is defined by

$$(f \uparrow f')(v, w) \;=\; f(v, w) + f'(v, w) - f'(w, v) \text{ for all } (v, w) \in E$$

Note that a flow $f'(v, w)$ through an edge (v, w) and a flow $f'(w, v)$ back through (w, v) may be seen as the same as a flow $f'(v, w) - f'(w, v)$ through the edge $(v, w) \in E$—this is called **cancellation**

Residual Networks – Augmentation Property

Lemma. If f is a flow in the network (G, c) and f' is a flow in the residual network (G_f, c_f), then the augmentation $f \uparrow f'$ is also a flow in the network (G, c) with flow value $|f \uparrow f'| = |f| + |f'|$.

Proof. We have to show the satisfaction of the **capacity constraint** for each $e \in E$ and the **flow conservation** property for each $v \in V - \{s, t\}$, and then calculate the **flow value**

Capacity constraint. First, for $(v, w) \in E$ we have $f'(w, v) \leq c_f(w, v) = f(v, w)$, so we get

$$\begin{aligned}(f \uparrow f')(v, w) &= f(v, w) + f'(v, w) - f'(w, v) \\ &\geq f(v, w) + f'(v, w) - f(v, w) = f'(v, w) \geq 0\end{aligned}$$

Furthermore,

$$\begin{aligned}(f \uparrow f')(v, w) &= f(v, w) + f'(v, w) - f'(w, v) \leq f(v, w) + f'(v, w) \\ &\leq f(v, w) + c_f(v, w) = f(v, w) + c(v, w) - f(v, w) = c(v, w)\end{aligned}$$

Proof / cont.

Flow conservation. Let $v \in V$ with $s \neq v \neq t$

We know that f and f' satisfy the flow conservation property, so we get

$$\begin{aligned}
\sum_{w \in V} (f \uparrow f')(v,w) &= \sum_{w \in V} (f(v,w) + f'(v,w) - f'(w,v)) \\
&= \sum_{w \in V} f(v,w) + \sum_{w \in V} f'(v,w) - \sum_{w \in V} f'(w,v) \\
&\quad \text{(now use the flow conservation properties)} \\
&= \sum_{u \in V} f(u,v) + \sum_{u \in V} f'(u,v) - \sum_{u \in V} f'(v,u) \\
&= \sum_{u \in V} (f(u,v) + f'(u,v) - f'(v,u)) \\
&= \sum_{u \in V} (f \uparrow f')(u,v)
\end{aligned}$$

Proof / cont.

Flow value. As f' is defined on a network with anti-parallel edges, we need the following modified definition:

$$|f'| = \sum_{e \in V_{in}} f'(e) - \sum_{e \in V_{out}} f'(e) \; ,$$

where $V_{in} = \{e \in E_f \mid e = (v,t) \text{ for some } v \in V\}$ and
$V_{out} = \{e \in E_f \mid e = (t,v) \text{ for some } v \in V\}$

With this we get

$$\begin{aligned} |f \uparrow f'| &= \sum_{e \in V_{in}} (f \uparrow f')(e) = \sum_{v \in V, (v,t) \in E} (f(v,t) + f'(v,t) - f'(t,v)) \\ &= \sum_{v \in V, (v,t) \in E} f(v,t) + \sum_{v \in V, (v,t) \in E} f'(v,t) - \sum_{v \in V, (v,t) \in E} f'(t,v) \\ &= |f| + \sum_{e \in V_{in}} f'(e) - \sum_{e \in V_{out}} f'(e) = |f| + |f'| \end{aligned}$$

Augmenting Paths

Given a network $G = (V, E)$ with capacity function $c : E \to \mathbb{R}$, and a flow f in (G, c), an **augmenting path** is a simple path p from s to t in the residual network G_f

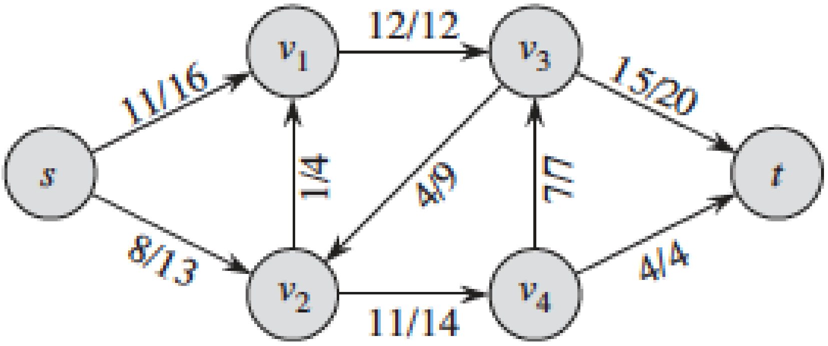
The **residual capacity** of an augmenting path $p = e_1, \dots, e_k$ is
$c_f(p) = \min\{c_f(e_i) \mid 1 \le i \le k\}$

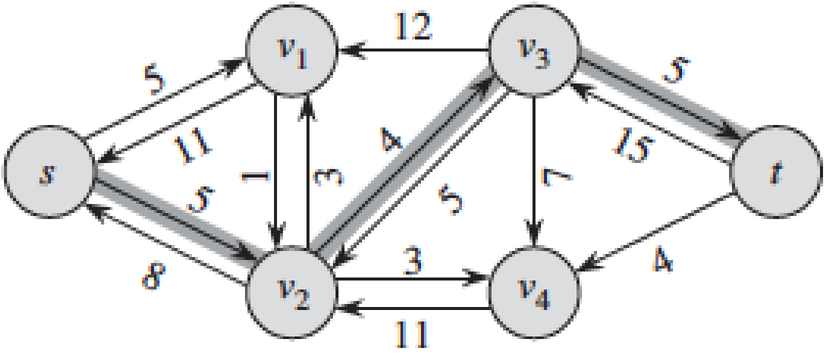
Obviously, if we define $f_p(e_i) = c_f(e_i)$ for the edges e_i in an augmenting path p and extend it by $f_p(e) = 0$ for all other edges in E_f, we obtain a flow in the residual network (G_f, c_f)

The flow value of the flow f_p is $|f_p| = c_f(p) > 0$

Combining these facts about augmenting paths with our previous lemma on the augmentation property shows that $f \uparrow f_p$ is a flow in the network (G, c) with flow value $|f \uparrow f_p| = |f| + c_f(p) > |f|$

Example



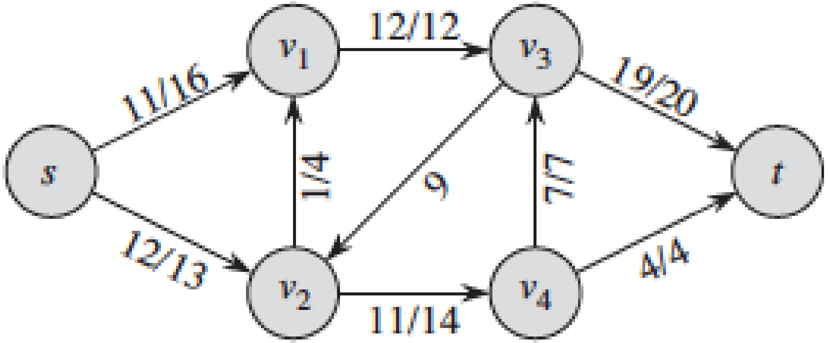


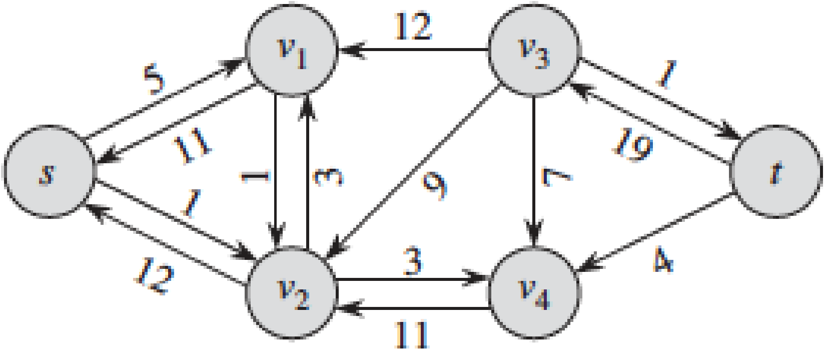
The graph on the left shows our "Canadian network" (G, c) together with a flow f

Edges are labelled by $f(e)/c(e)$

The graph on the right shows the residual network (G_f, c_f) with a shaded augmenting path $p = (s, v_2), (v_2, v_3), (v_3, t)$ with residual capacity $c_p = 4$

Example / cont.





The graph on the left shows the same network with the flow $f' = f \uparrow f_p$

Instead of $0/c(e)$ we simply use the label $c(e)$

The graph on the right shows again the residual network $(G_{f'}, c_{f'})$

Minimum Cuts

A **cut** of a network (G, c) is a set of edges $E' \subseteq E$ such that there is a partition $V = X \cup \bar{X}$ of the set of vertices with $s \in X$, $t \in \bar{X}$ and $E = E(X) \cup E' \cup E(\bar{X})$, where $E(V')$ for any $V' \subseteq V$ is the set of edges with endpoints in V'

That is, a cut is a set of edges E' that separates the source from the sink: removing these edges gives a graph, where the sink t cannot be reached from the sourse s

The **capacity of a cut** E' is $c(E') = \sum_{e \in E'_+} c(e)$, where E'_+ contains the edges from E' with source in X and destination in $\bar{X}$

Minimum cuts, i.e. cuts for which the capacity is minimal, are essential for solving the maximum flow problem

Example. In our "Canadian example" the edge set

$$E' = \{(v_1, v_3), (v_4, v_3), (v_4, t), (v_3, v_2)\}$$

defines a minimum cut with $X = \{s, v_1, v_2, v_4\}$, $\bar{X} = \{v_3, t\}$ and capacity $c(E') = 23$

Net Flow

Let us write $E'(X, \bar{X})$ for a cut E' with corresponding partition $V = X \cup \bar{X}$

Furthermore, in addition to the set E'_+ let E'_- be the edges in E' with source in $\bar{X}$ and destination in X

Then the **net flow** of a flow f in (G, c) through a cut E' is

$$f_{\text{net}}(E') = \sum_{e \in E'_+} f(e) - \sum_{e \in E'_-} f(e)$$

Note the assymetry between the notions of capacity—only edges in E'_+ are considered—and net flow—also edges in E'_- are considered

Example. The flow f in the "Canadian network" and the cut

$$E' = \{(v_1, v_3), (v_4, v_3), (v_4, t), (v_3, v_2)\}$$

give rise to the net flow

$$f_{\text{net}}(E') = 12 + 7 + 4 - 4 = 19$$

Net Flow and Flow Value

Lemma. Let E' be a cut of a network (G, c) and let f be a flow in (G, c). Then the net flow of f through the cut E' is the flow value of f, i.e. $f_{\text{net}}(E') = |f|$.

Proof. We simply calculate

$$
\begin{aligned}
|f| &= \sum_{e \in E, e=(v,t)} f(e) = \sum_{v \in V} f(v,t) + \sum_{w \in \bar{X}, w \neq t} \left(\sum_{v \in V} f(v,w) - \sum_{v \in V} f(w,v) \right) \\
&= \sum_{v \in V} \left(f(v,t) + \sum_{w \in \bar{X}, w \neq t} f(v,w) \right) - \sum_{v \in V} \left(f(t,v) + \sum_{w \in \bar{X}, w \neq t} f(w,v) \right) \\
&= \sum_{v \in V} \sum_{w \in \bar{X}} f(v,w) - \sum_{v \in V} \sum_{w \in \bar{X}} f(w,v) \\
&= \sum_{v \in X} \sum_{w \in \bar{X}} f(v,w) + \sum_{v \in \bar{X}} \sum_{w \in \bar{X}} f(v,w) - \sum_{v \in X} \sum_{w \in \bar{X}} f(w,v) - \sum_{v \in \bar{X}} \sum_{w \in \bar{X}} f(w,v) \\
&= \sum_{e \in E'_+} f(e) - \sum_{e \in E'_-} f(e) = f_{\text{net}}(E')
\end{aligned}
$$

Minimum Cut Bound

The following is a simple consequence of the net-flow lemma

Corollary. Let E' be a cut of a network (G, c) and let f be a flow in (G, c). Then the flow value of f is bounded from above by the capacity of any minimum cut E', i.e. $|f| \leq c(E')$.

Proof. For an arbitrary cut E' we have

$$\begin{aligned} |f| \; &= \; f_{\text{net}}(E') \; = \; \sum_{e \in E'_+} f(e) - \sum_{e \in E'_-} f(e) \\ &\leq \; \sum_{e \in E'_+} f(e) \; \leq \; \sum_{e \in E'_+} c(e) \; = \; c(E') \end{aligned}$$

Max-Flow Min-Cut Theorem

Theorem. Let f be a flow in a network (G, c). Then the following conditions are equivalent:

(i) f is a maximum flow;

(ii) the residual network (G_f, c_f) contains no augmenting path;

(iii) $|f| = c(E')$ holds for some minimum cut E'.

Proof. "(i) $\Rightarrow$ (ii)": Let f be a maximum flow, and assume that the residual network (G_f, c_f) contains an augmenting path p

We have already shown that p defines a flow f_p in (G_f, c_f) with $|f \uparrow f_p| > |f|$

This is a contradiction to the assumption that f is already maximal

Proof (cont.)

"(ii) $\Rightarrow$ (iii)": Assume that there is no augmenting path, i.e. in G_f there is no path from s to t

Define $X = \{v \in V \mid \text{there is a path from } s \text{ to } v \text{ in } G_f\}$ and $\bar{X} = V - X$

Then the set E' of edges e with one endpoint in X and another one in $\bar{X}$ defines a cut

Now let $v \in X$ and $w \in \bar{X}$

For $(v, w) \in E$ we must have $c_f(v, w) = 0$—otherwise there would be a path in G_f from s to w using the edge $(v, w) \in E_f$—so we get $f(v, w) = c(v, w)$

For $(w, v) \in E$ we must have $f(w, v) = 0$—otherwise $c_f(v, w) = f(w, v) > 0$ implies $(v, w) \in E_f$ and hence there is a path from s to w using the edge $(v, w) \in E_f$

Proof (cont.)

With this we get

$$f_{\text{net}}(E') = \sum_{e \in E'_+} f(e) - \sum_{e \in E'_-} f(e) = \sum_{e \in E'_+} c(e) = c(E')$$

This implies $|f| = f_{\text{net}}(E') = c(E')$, i.e. E' must be a minimum cut

"(iii) $\Rightarrow$ (i)": We know that $|f| \leq c(E')$ holds for any minimum cut E'

The fact that we have $|f| = c(E')$ holds implies that $|f|$ is maximal, so f is a maximum flow

The Ford-Fulkerson Algorithm

The basic idea of the Ford-Fulkerson algorithm is to

- start with a flow f on the network (G, c) that is 0 on all edges $e \in E$
- find an augmenting path p in the residual network (G_f, c_f)
- update f to $f \uparrow f_p$

Naturally, the algorithm terminates—in each step the flow value $|f|$ will be increased—when there is no more augmenting path

The correctness is a consequence of the max-flow min-cut theorem

The Ford-Fulkerson Algorithm / Initialisation

For the input network we use a unary function symbol Vertex, a binary function symbol Edge and a unary function symbol Cap for the vertices, edges and the capacity of edges

We write $v \in \mathit{Vertex}$ instead of $\mathit{Vertex}(v) = \mathit{true}$ and $(v, w) \in \mathit{Edge}$ instead of $\mathit{Edge}(v, w) = \mathit{true}$

Use a binary function symbol Flow to capture the flow in (G, c)—initially we have $\mathit{Flow}(v, w) = 0$ for $(v, w) \in \mathit{Edge}$; otherwise it is undefined

For the residual capacity we use a derived binary function symbol ResCap with

$$\mathit{ResCap}(v, w) = \begin{cases} \mathit{Cap}(v, w) - \mathit{Flow}(v, w) & \text{if } (v, w) \in \mathit{Edge} \\ \mathit{Flow}(w, v) & \text{if } (w, v) \in \mathit{Edge} \\ 0 & \text{else} \end{cases}$$

The Ford-Fulkerson Algorithm / Initialisation / cont.

We use a unary function symbol *Label* to mark vertices—initially, we only have $\mathit{Label}(s) = (\perp, \infty)$, otherwise it is undefined

We use another unary function symbol *Path* to collect the vertices on an augmenting path from s to a vertex v—initially, we only have $\mathit{Path}(s) = s$

Finally, use a nullary function symbol *Labelled* to keep track of vertices that have already been explored when searching for an augmenting path

Usually, the basic Ford-Fulkerson algorithm uses a stack to represent the set *Labelled*; then the set difference becomes a *pop* operation, and the set union becomes a concatenation

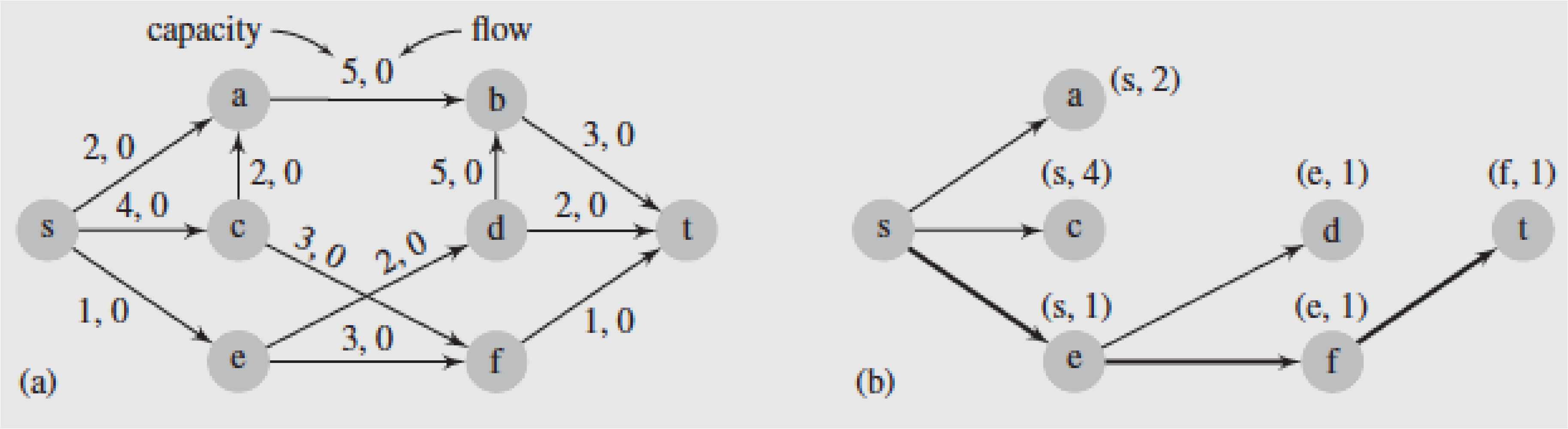
For the path lists we use $+$ to denote concatenation

The Ford-Fulkerson Algorithm / Rule

Then the following rule is iterated:

```
IF $Labelled \neq \emptyset$
THEN CHOOSE $v \in Labelled$ DO
        $Labelled := Labelled - \{v\}$
        FORALL $w \in Vertex - Labelled$ DO
            IF $((v, w) \in Edge \vee (w, v) \in Edge) \wedge ResCap(v, w) > 0$
            THEN $Label(w) := (v, \min(\pi_2(Label(v)), ResCap(v, w)))$
                 $Path(w) := Path(v) + [w]$
            IF $w \neq t$
            THEN $Labelled := Labelled \cup \{w\}$
            ELSE $Labelled := \{s\}$
                FORALL $u_1, u_2 \in Vertex$ WITH
                        $\exists p_1, p_2 . Path(w) = p_1 + [u_1, u_2] + p_2$ DO
                    LET $p, val, v'$ WITH
                        $Path(w) = p + [v', t] \wedge Flow(v', t) = val$ IN
                        IF $(u_1, u_2) \in Edge$ THEN $Flow(u_1, u_2) : +val$
                        IF $(u_2, u_1) \in Edge$ THEN $Flow(u_1, u_2) : -val$
```

Example / 1

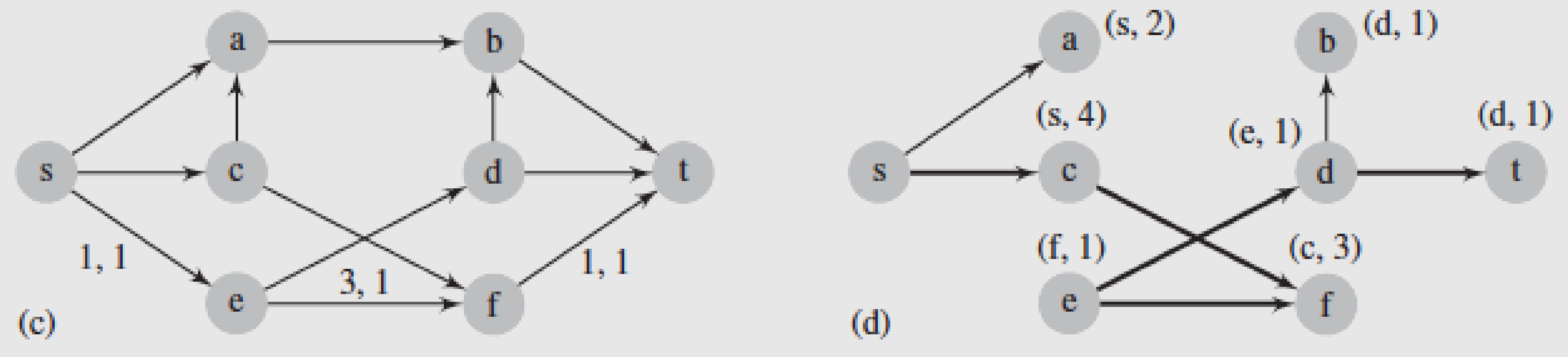


The graph on the left shows the input network and the initial flow; the label of vertex s has been omitted

The vertices are processed in the order s, e, f each time labelling all neighbours and continuing with the last explored vertex that is added to *Labelled*

The graph on the right shows the result, when *Labelled* is reset to $\{s\}$ with labels assigned to the vertices, and the augmented path from s to t highlighted

Example / 2



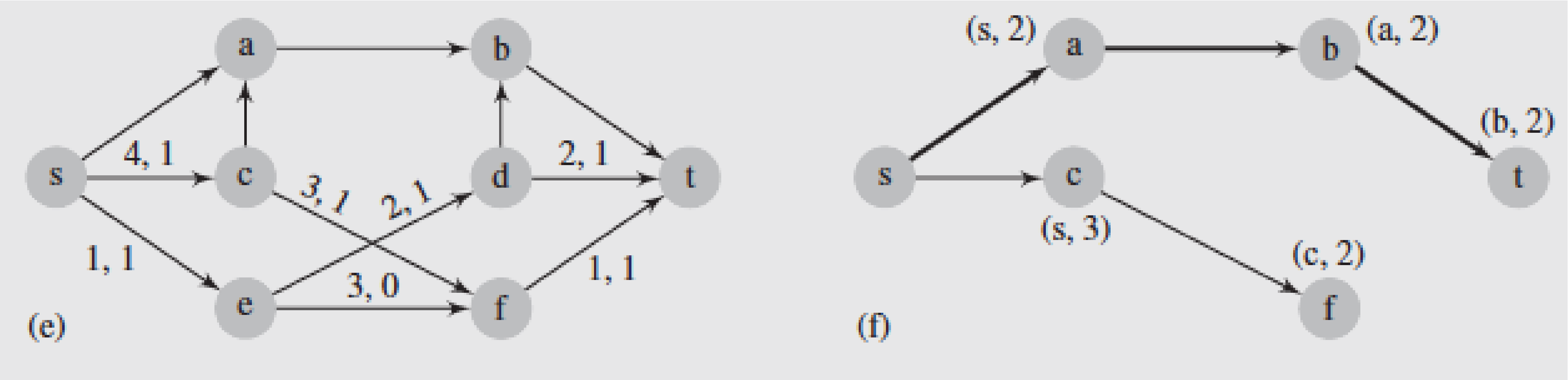
The graph on the left shows the augmented flow $f \uparrow f_p$ with the discovered augmenting path p

As for the edge (s, e) the flow coincides with the capacity, i.e. $c_f(s, e) = 0$ the edge is no longer in E_f

The next iteration explores the vertices s, c, f, e, d, again labelling neighbours

In this case the augmenting path contains a backward edge (f, e), which leads to a drecrementation of the flow on the edge (e, f)

Example / 3

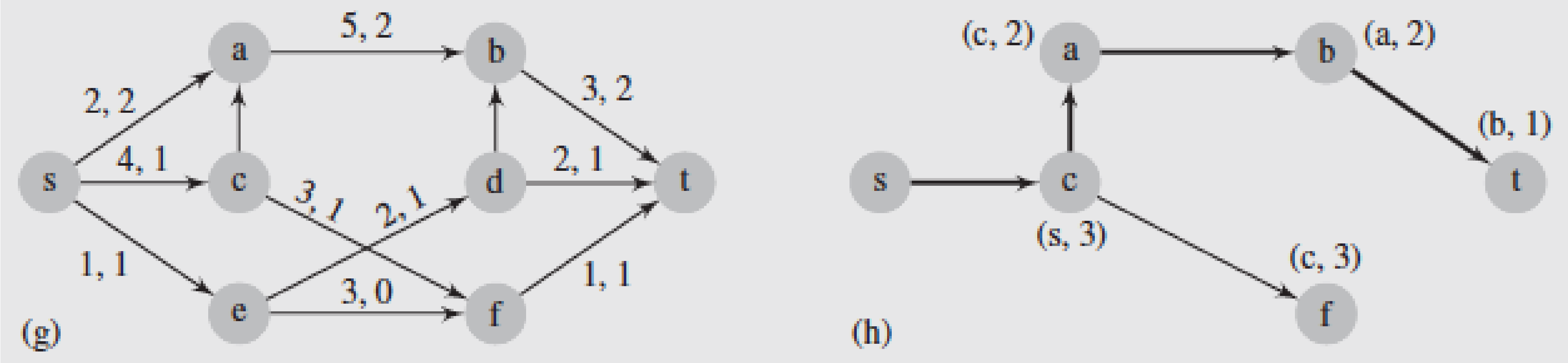


The graph on the left shows the augmented flow $f \uparrow f_p$—$f(e, f)$ is indeed decremented, i.e. the flow 1 is redirected via (e, d) and (d, t)

With this flow we get $(f, e) \notin E_f$, as the flow has become 0

The algorithm now processes the vertices s, c, f, a, b (note the backtracking for f) computing the labels in the right graph, which lead to the highlighted augmenting path $s - a - b - t$

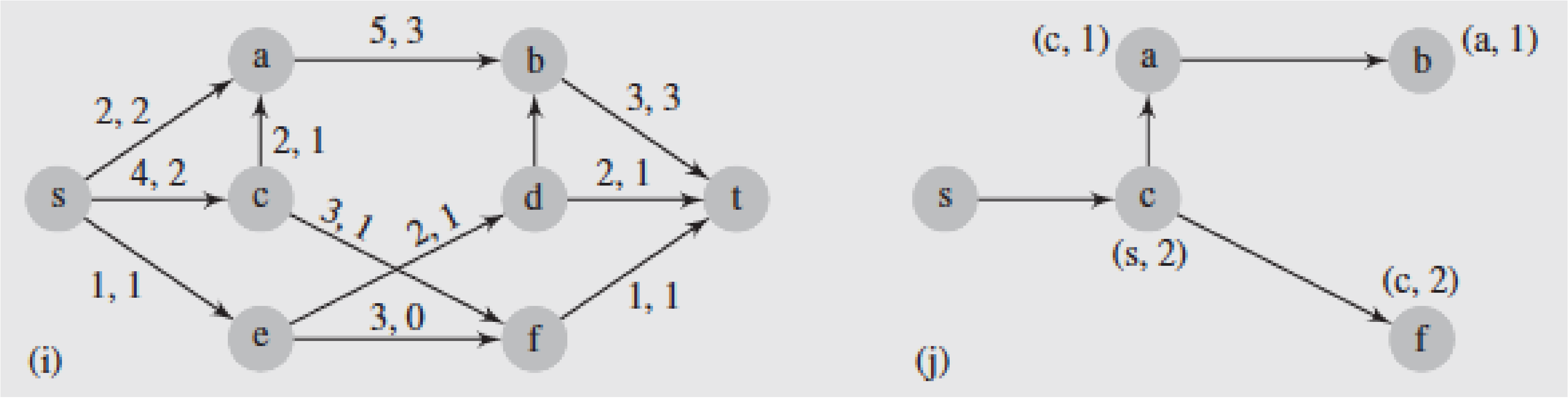
Example / 4



Again, the left graph shows the augmented flow $f \uparrow f_p$— now the edge (s, a) is removed from E_f

Vertices are processed in the order s, c, f, a, b, which gives rise to the computed labels and the highlighted augmenting path $s - c - a - b - t$

Example / 5



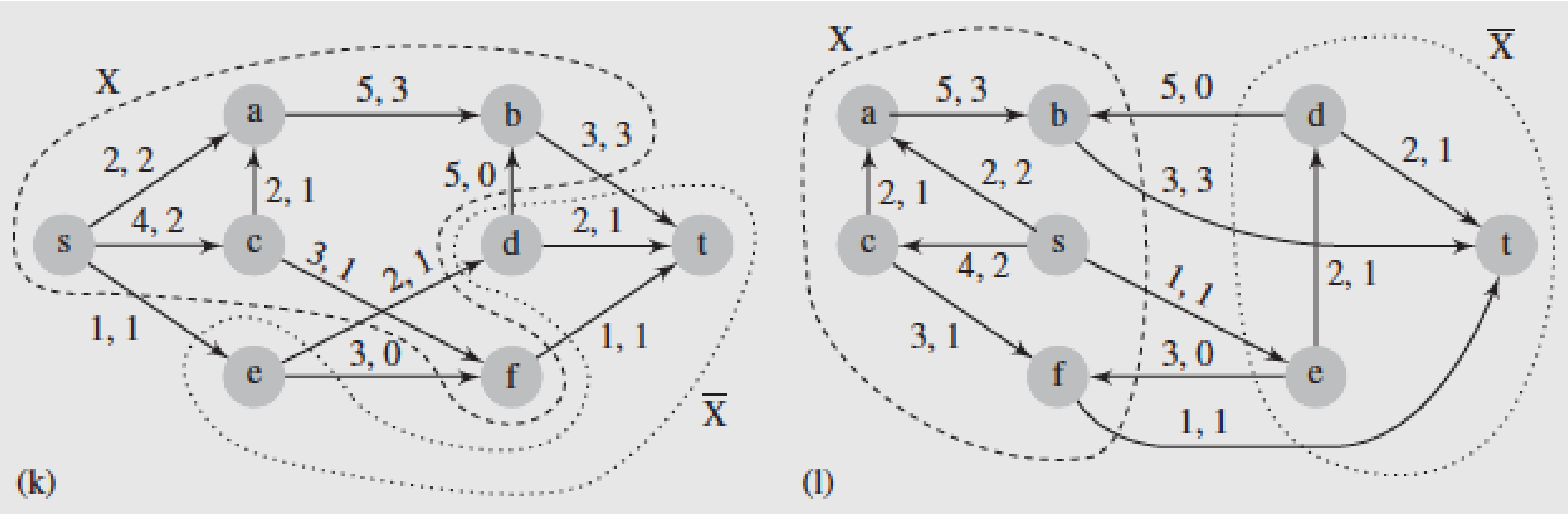
Again, the left graph shows the augmented flow $f \uparrow f_p$— now the edge (b, t) is removed from E_f

Vertices are processed again in the order s, c, f, a, b

However, in this case it is impossible to obtain an augmenting path, i.e. there is no possibility to further augment the flow

The result is a maximum flow f with flow value $|f| = 5$

Example / 6



The graph on the left shows again the resulting maximum flow

The vertices labelled in the last step of the algorithm are the vertices in the set X of a minimum cut E' with the edges $(b, t), (s, e), (f, t), (d, b), (e, f)$—the capacity of E' is 5

The right graph is just a rearrangement of the result and the minimum cut to better illustrate the cut property

Complexity Analysis

In order to determine an augnmenting path, the Ford-Fulkerson algorithm explores all edges $e \in E_f$, i.e. at most $2|E|$ edges

As the exploration for any edge $e \in E_f$ requires constant time, the time complexity for finding an augmenting path is in $O(|E|)$

An augmenting path can have a length between 1 and $|V| - 1$, so the number of different augmenting paths is in $O(|V| \cdot |E|)$

So the number of iterations of the Ford-Fulkerson algorithm is also bounded by $O(|V| \cdot |E|)$

In summary, the time complexity of the Ford Fulkerson algorithm in the worst case is in $O(|V| \cdot |E|^2)$

The Edmonds-Karp Algorithm

The disadvantage of the Ford-Fulkerson algorithm is that it greedily considers the first augmenting path p, but the residual capacity of the chosen graph may be very small

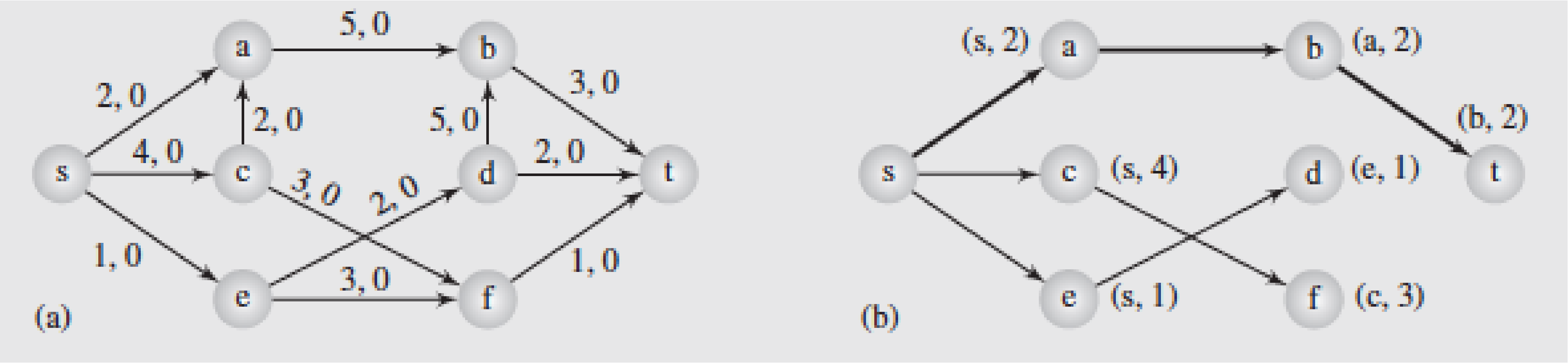
So it will take many iterations each time searching for a new augmenting path to obtain a maximum flow

The Edmonds-Karp algorithm avoids this problem by taking a breadth-first approach: instead of a stack representing the set *Labelled* it uses a FIFO queue—further details are omitted

However, while tiny increments of the flow value can be avoided, the Edmonds-Karp algorithm (same as the Ford-Fulkerson algorithm) does not avoid multiple recomputations of labels; the worst case complexity remains in $O(|V| \cdot |E|^2)$

The algorithm by Dinic overcomes this problem as well using labelled layered networks

Example / 1

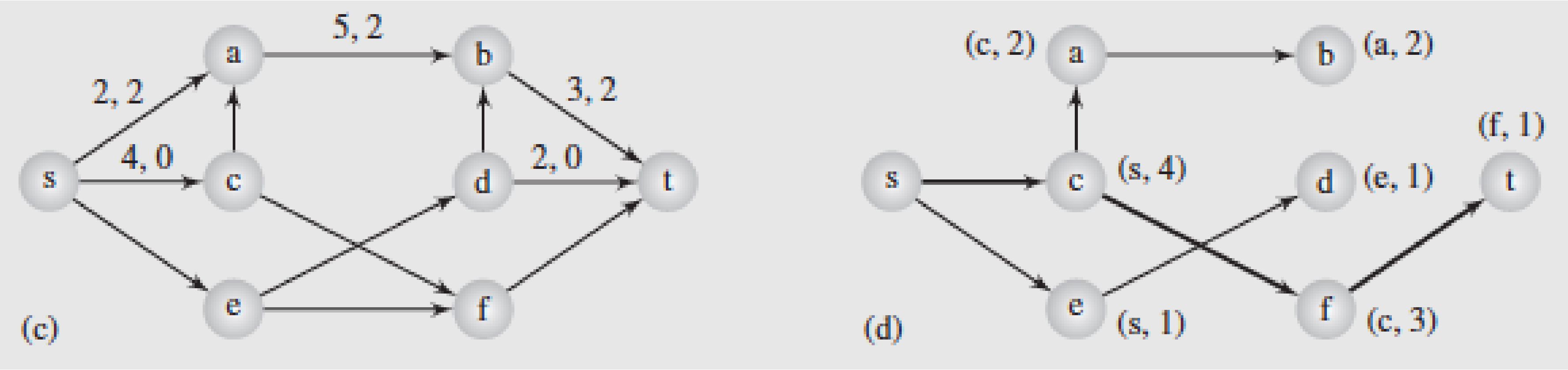


The graph on the left shows the input network and the initial flow; the label of vertex s has been omitted

The vertices are processed in the order s, a, c, e, b each time labelling all neighbours and adding them to the queue*Labelled*

The graph on the right shows the result, when *Labelled* is reset to $\{s\}$ with labels assigned to the vertices, and the augmented path from s to t highlighted

Example / 2



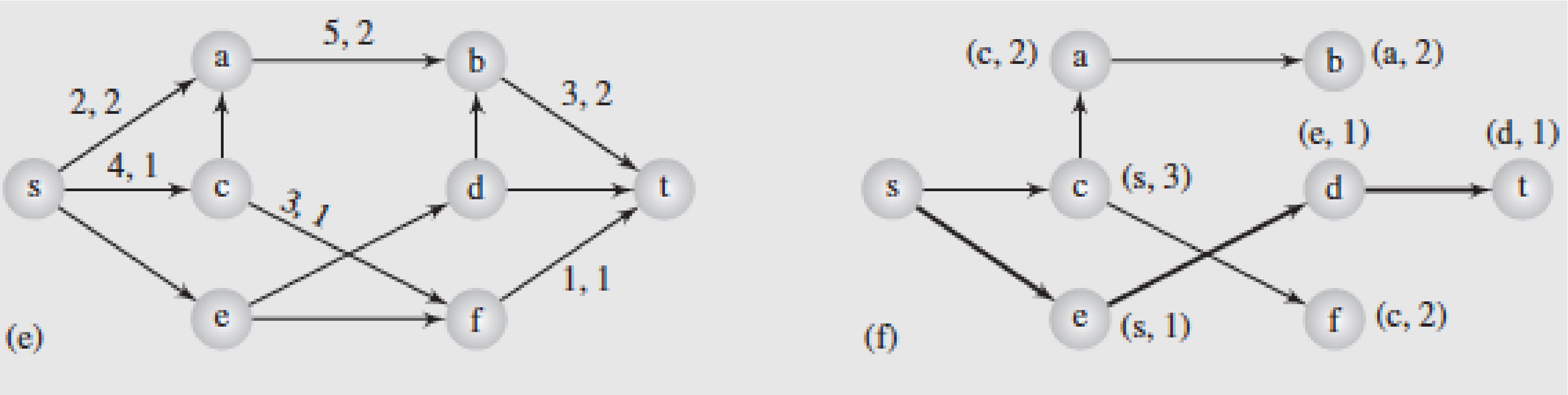
The graph on the left shows the augmented flow $f \uparrow f_p$ with the discovered augmenting path p

As for the edge (s, a) the flow coincides with the capacity, i.e. $c_f(s, e) = 0$ the edge is no longer in E_f

The next iteration explores the vertices s, c, e, a, f, again labelling neighbours

The graph on the right shows the result, when *Labelled* is reset to $\{s\}$ with labels assigned to the vertices, and the augmented path from s to t highlighted

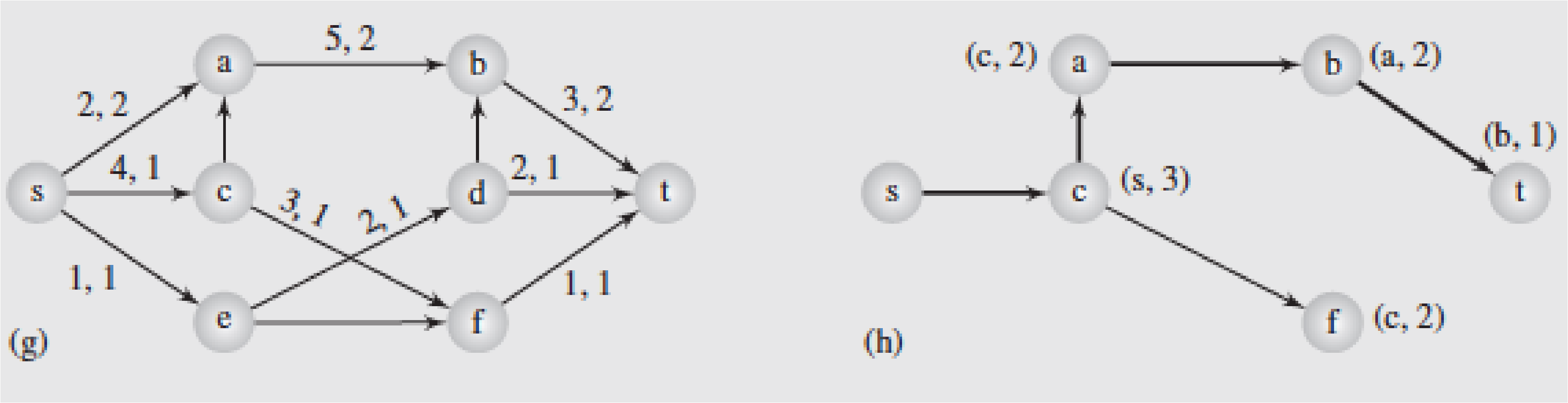
Example / 3



The graph on the left shows the augmented flow $f \uparrow f_p$—with this flow we get $(f, t) \notin E_f$, as the flow is equal to the capacity

The algorithm now processes the vertices s, c, e, a, f, d computing the labels in the right graph, which lead to the highlighted augmenting path $s - c - d - t$

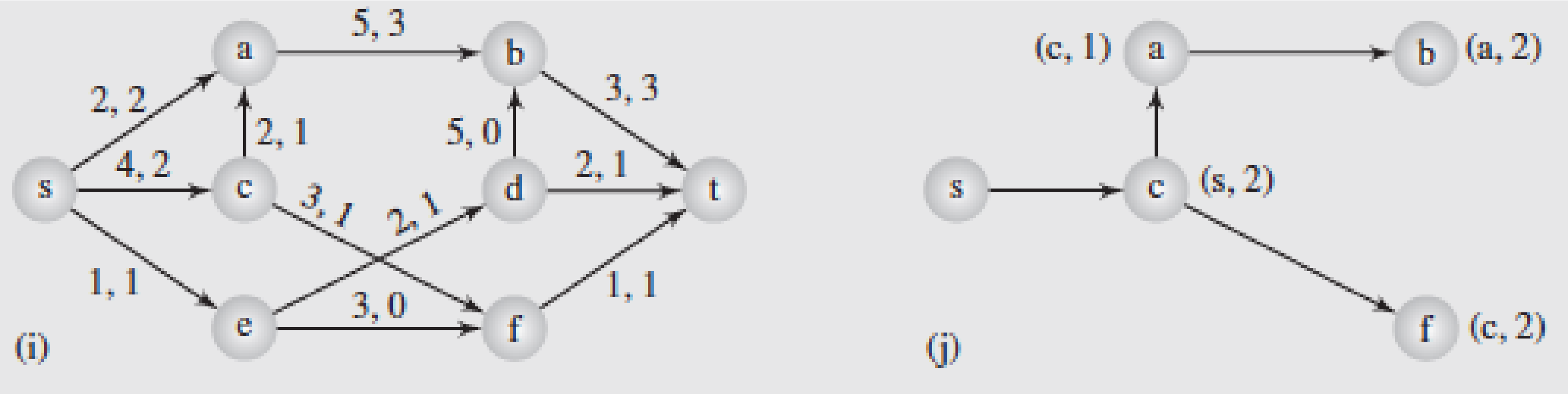
Example / 4



Again, the left graph shows the augmented flow $f \uparrow f_p$— now the edge (s, a) is removed from E_f

Vertices are processed in the order s, c, a, fb, which gives rise to the computed labels and the highlighted augmenting path $s - c - a - b - t$

Example / 5



Again, the left graph shows the augmented flow $f \uparrow f_p$— now the edge (b, t) is removed from E_f

Vertices are processed again in the order s, c, a, f, b

However, in this case it is impossible to obtain an augmenting path, i.e. there is no possibility to further augment the flow

The result is again a maximum flow f with flow value $|f| = 5$

Dinic's Algorithm

The refinement by Dinic follows the idea not to compute just one augmenting path at a time, but to compute all augmenting paths of a given length

So there will be up to $|V| - 1$ iterations

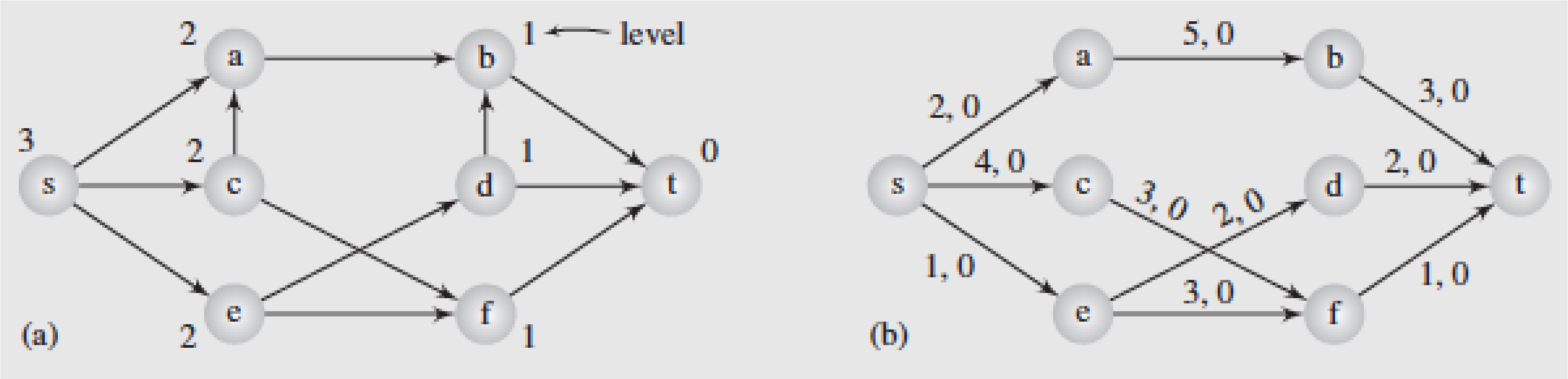
A **layering** of network $G = (V, E)$ with $p+1$ layers is a partition $V = V_0 \cup \cdots \cup V_p$ such that every path $s = v_p v_{p-1} \dots v_1 v_0 = t$ of length p from source to sink satisfies $v_i \in V_i$ for $0 \leq i \leq p$

In Dinic's algorithm for $1 \leq p \leq |V| - 1$ in every step for a fixed p a layering of G with $p + 1$ layers is determined: use BFS starting from t and assign layer values to all not yet labelled neighbours in G_f

Then use the Ford-fulkerson method to determine augmenting paths of length p and update the flow accordingly

This continues until no more layering is possible, in the worst case until $p = |V| - 1$

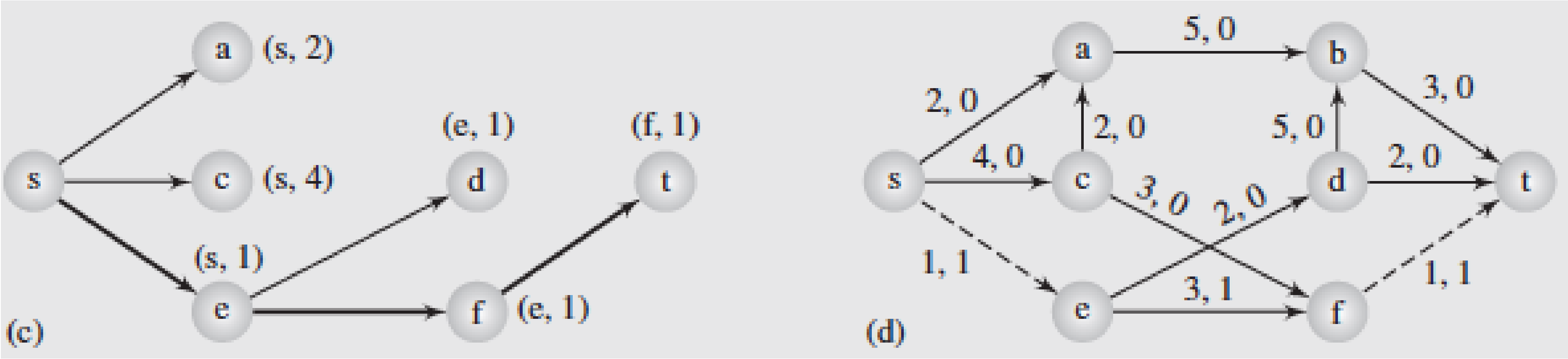
Example / 1



Starting with the minimum path length $p = 3$ we obtain the layering indicated in the left graph

The graph on the right shows the network and the initial flow containing only edges between neighbouring layers—other edges can not appear in paths of length 3

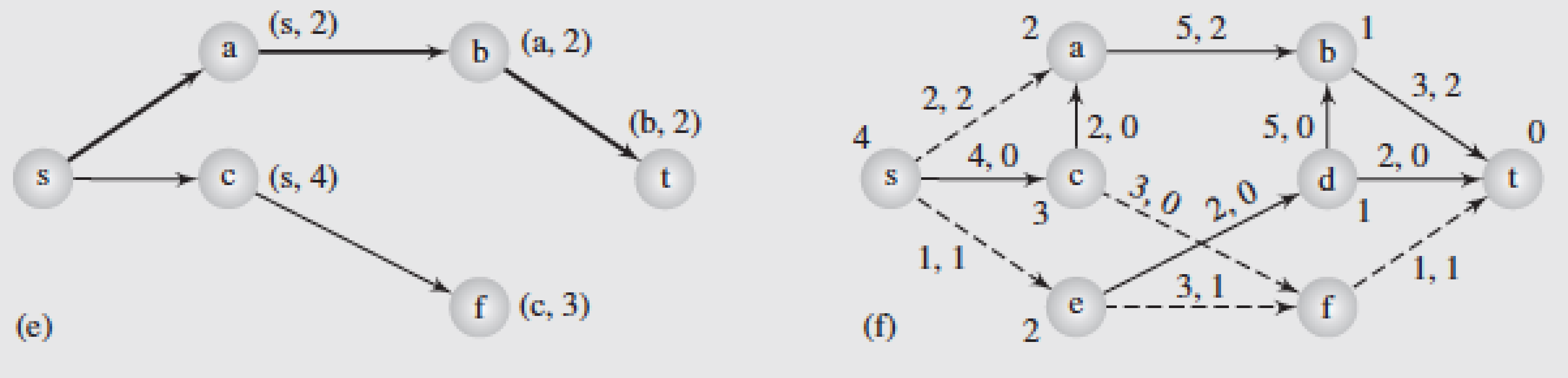
Example / 2



In this layered network the Ford-Fulkerson method finds an augmented path $s - e - f - t$ with residual capacity 1 as indicated on the left

The network on the right shows the augmented flow—the dashed edges are those for which the flow exhausts the capacity, i.e. the edge will be removed from the residual network G_f

Example / 3



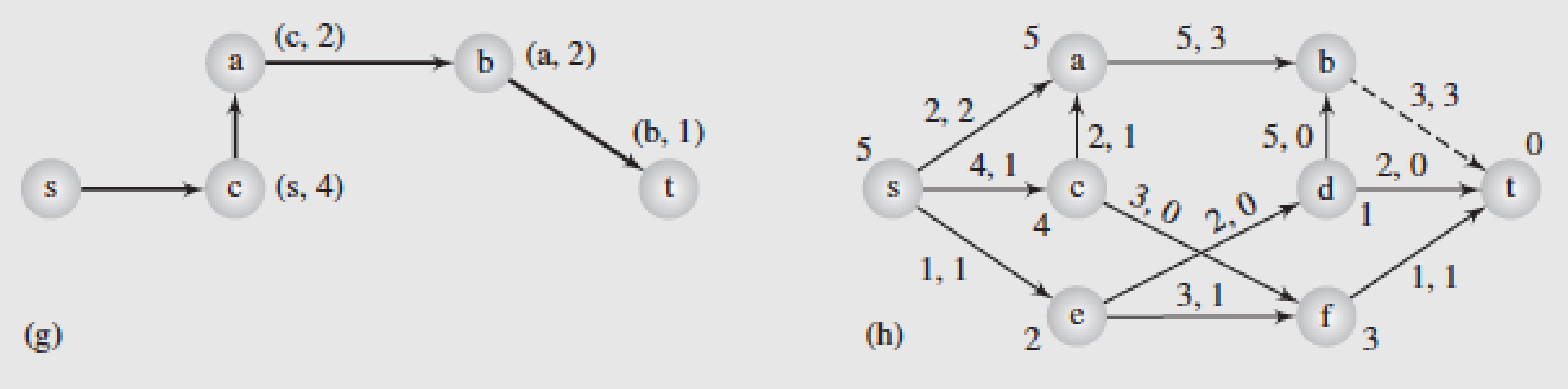
Then the Ford-Fulkerson method finds another augmenting path $s - a - b - t$ with residual capacity 2

The network on the right shows the augmented flow—the edge (s, a) is dashed, because its flow exhausts its capacity

As there are no more augmenting paths of length 3, a new layering with $p = 4$ is computed on G_f—this is also shown on the right

Here the edges (c, f) and (e, f) are dashed, because they cannot be part of a path of length 4 from s to t

Example / 4

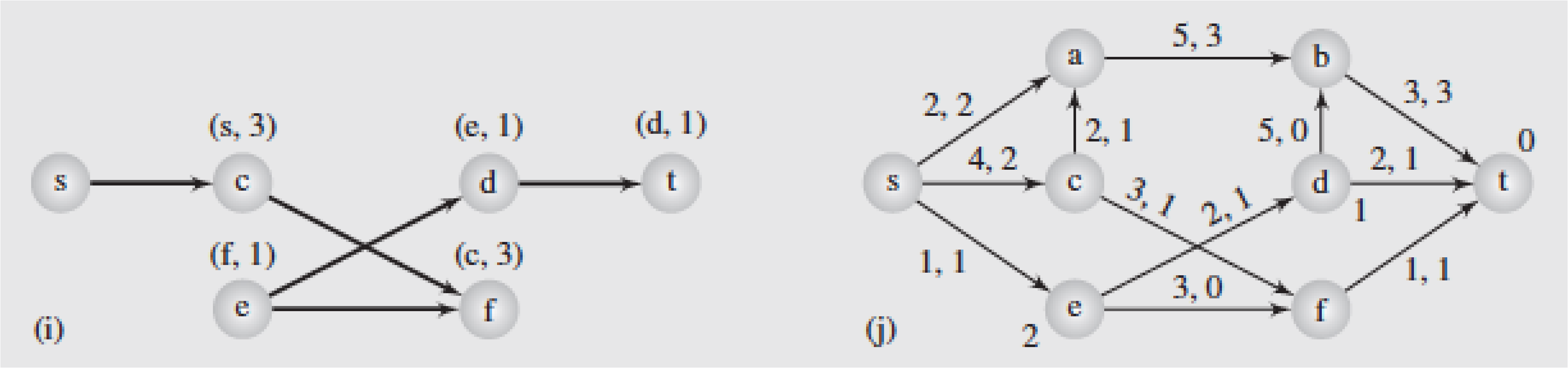


In this layered network the Ford-Fulkerson method finds the augmenting path $s - c - a - b - t$ with residual capacity 1 as shown on the left

The network on the right shows the augmented flow

As (b, t) becomes dashed, there is no other augmenting flow of length 4, so the new layering (on the right) with $p = 5$ is computed

Example / 5



In this layered network the Ford-Fulkerson method finds the augmenting path $s - c - f - e - d - t$ with residual capacity 1 as shown on the left—here (f, e) is a backward edge

The network on the right shows the augmented flow

As there is no other augmenting flow, the algorithms attempts to find a layering with $p = 6$, which is not possible

So a maximum flow will flow value 5 has been found

Final Remarks

The complexity improvement by Dinic's algorithm results from the fact that the number of steps to find an augmenting path is reduced from $O(|E|)$ to $O(|V|)$ for the price of additional $O(|V| \cdot |E|)$ steps for layering the network

In total, the worst case complexity of Dinic's algorithm is in $O(|V|^2 \cdot |E|)$

More sophisticated **push-relabel** algorithms have been developed to further improve the complexity for determing maximum flows—for details see the literature

Maximum flow algorithms can be combined with shortest path algorithms to obtain **maximum flows at minimal costs**

For time reasons we cannot dig deeper into this subject—see also the literature

11.2 Graph Colouring Algorithms

Let $G = (V, E)$ be an undirected graph, and let C be a (large enough) set of colours

A **graph colouring** is a function $c : V \to C$ such that $c(v) \neq c(w)$ holds for all $\{v, w\} \in E$

We are interested in a minimal colouring, i.e. $|c(V)|$ should be minimal—the mimum number of colours needed to colour a graph G is called the **chromatic number** of G

All known algorithms that provably compute a minimal colouring have exponential time complexity, so we are interested in algorithms that can obtain a colouring, which is close to an optimal colouring

Applications. Consider a time table problem, where different courses (vertices) cannot be assigned the same time slot (colour), if both need to be taken by at least one student (edges)

In general, an edge represents a dependency of any kind between the vertices, and only independent vertices can be grouped together by assigning the same colour to them

Greedy Graph Colouring

There is an obvious greedy strategy for graph colouring using the vertices $v_1, \dots, v_n$ in fixed order

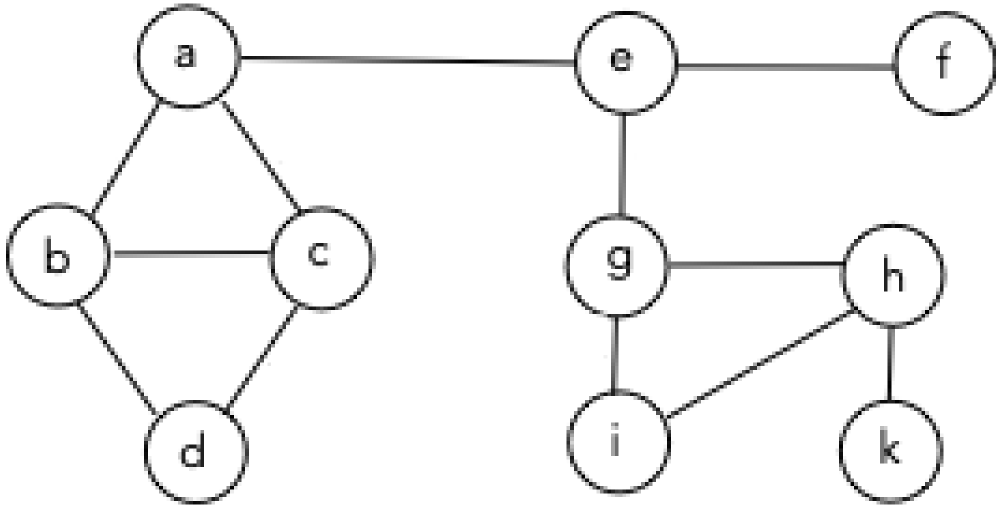
Let $V' \subseteq V$ be the set of already coloured vertices, so initially we have $V' = \emptyset$, and let $C' \subseteq C$ be the set of already used colours, so initially also $C' = \emptyset$

Then iterate until all vertices are coloured, i.e. $V' = V$, in each step using a new colour $c \in C - C'$ and processing all uncoloured vertices $v \in V - V'$ in the given order

If there is no edge between v and a vertex in V', then set $c(v) = c$ and add v to V'; otherwise ignore v

Clearly, it is possible that the greedy strategy finds a minimal colouring: just use a minimal colouring $c : V \to C$ and order V according to the colours in this colouring

Example



For the vertices in the order $a, b, c, d, e, f, g, h, i, k$ we get a colouring

$$c(a) = c(d) = c(f) = c(g) = c(k) = \text{red} \quad c(b) = c(e) = c(h) = \text{green}$$
$$\text{and} \quad c(c) = c(i) = \text{blue}$$

with three colours, which is minimal

For the vertices in the order $b, f, h, c, g, k, a, d, i, e$ we get a colouring with four colours

Bad Colourings

However, the greedy strategy may also result in an arbitrarily bad colouring

Let $chr(G)$ be the chromatic number of any graph G. Then for every $\alpha > 0$ we can find a graph $G = (V, E)$ and an ordering of V such that the number g of colours needed by the greedy colouring algorithm satisfies $g > \alpha \cdot chr(G)$.

Take a graph $G = (V, E)$ with $V = \{v_i, w_i \mid 1 \leq i \leq n\}$ and edges $\{v_i, w_j\}$ iff $i \neq j$ such that $n > 2 \cdot \alpha$ holds

The graph is bipartite with $V_1 = \{v_i \mid 1 \leq i \leq n\}$ and $V_2 = \{w_i \mid 1 \leq i \leq n\}$

So we get $chr(G) = 2$ by colouring V_1 with red and V_2 with blue

However, if we choose the order $v_1, w_1, \dots, v_n, w_n$, then our greedy strategy will require $g = n$ colours

For i running from 1 to n we can colour v_i and w_i with the same colour c_i, but then all following v_j and w_j must have different colours

Colouring Heuristics

It is obvious that the greedy strategy will produce a colouring with at most $d + 1$ colours, where d is the maximum degree of all vertices: in the worst case all d neighbours of a vertex v have been coloured before v

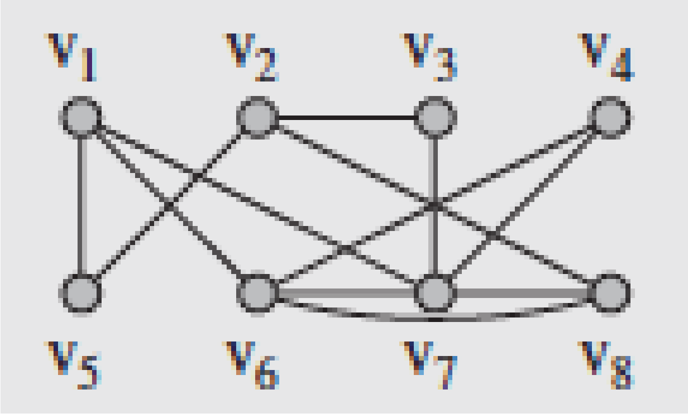
It is also obvious that in the worst case for the i'th vertex in the used ordering the i'th colour must be used

Hence $\max_i \min(i, d + 1)$ is a bound for the number of colours used by the greedy colouring strategy

This suggest to use an ordering, in which vertices with high degree are smaller than vertices with large degree—call this the **largest first heuristics**

Note that for our bad colouring example this is of no help, as all vertices have the same degree $n - 1$

Example



Ordering the vertices by decreasing degree gives the sequence

$$v_7, v_6, v_1, v_2, v_8, v_3, v_4, v_5$$

The greedy colouring algorithms results in the colouring

$$c(v_7) = c(v_2) = \text{red} \qquad c(v_6) = c(v_3) = c(v_5) = \text{green}$$
$$c(v_1) = c(v_8) = c(v_4) = \text{blue} \ ,$$

which is minimal, whereas $\max_i \min(i, d + 1) = 4$

Brelaz's Colouring Heuristics

The largest first heuristics uses only a single criterion for the ordering of vertices; it leaves a lot of latitude in case of vertices of equal degree

Instead, the heuristics by Brelaz takes first the **saturation degree**, i.e. the number of colours already used for neighbours, and second (in case of a tie) the number of not yet coloured neighbours

In our previous example the greedy colouring algorithms results in the colouring

$$c(v_7) = c(v_2) = \text{red} \qquad c(v_6) = c(v_5) = c(v_3) = \text{green}$$
$$\text{and} \quad c(v_1) = c(v_8) = c(v_4) = \text{blue}$$

11.3 Euler and Hamilton Graphs

Let $G = (V, E)$ be a connected undirected graph

Recall that an **Euler cycle** is a cycle in G that contains every edge exactly once

An **Euler graph** is a graph that contains an Euler cycle

A **Hamilton cycle** is a cycle in G that contains every vertex exactly once

An **Hamilton graph** is a graph that contains a Hamilton cycle

The Chinese Postman Problem

Recall that a graph is an Euler graph iff every vertex has even degree (see Math 213)

Therefore, if this condition is satisfied, then using DFS with an arbitrary start node and continuing, as long as there are unexplored edges, will produce an Euler cycle

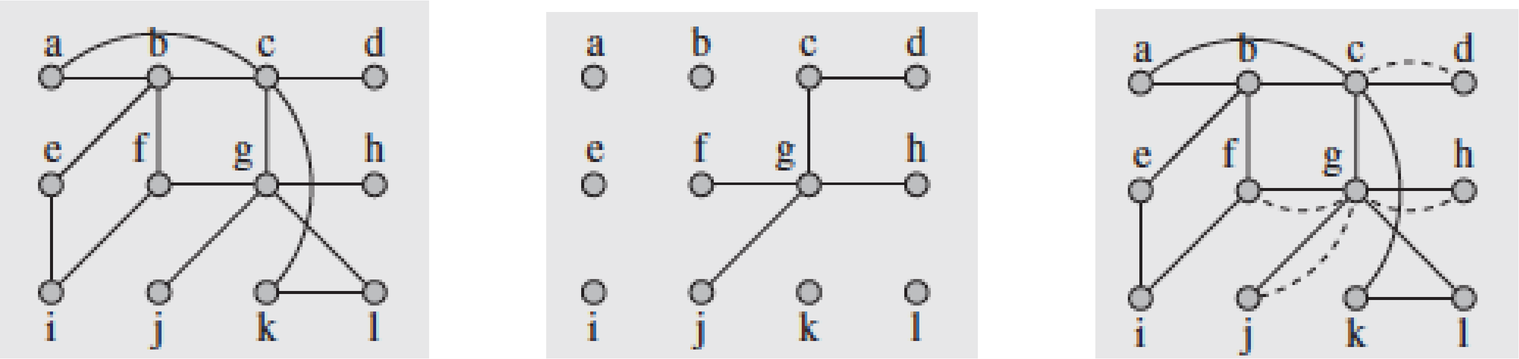
The **Chinese postman problem** is to find a cycle of minimum length in an arbitrary undirected connected graph $G = (V, E)$, in which every edge $e \in E$ appears a least once.

The problem was originally formulated by Mei Ko Kwan in 1960 as finding a shortest tour for a postman who picks up mail from the post office, delivers post to houses in certain streets (so every street has to be traversed at least once), and finally returns to the post office

The simple idea to solve this problem is to extend the graph to an Euler graph $G^* = (V, E^*)$ by duplicating edges, i.e. $E \subseteq E^*$

It is always possible to get such an extension E^*; we need to find a minimal one

Example



The graph on the left is not an Euler graph; the graph in the middle is the subgraph spanned by the vertices with odd degree

The graph on the right shows duplicated edges (dashed) giving rise to an optimal postman tour

Note that this is still the case, if we consider edge length $\neq 0$, e.g. if we consider a length 1.4 for the edge $\{g, j\}$ (and 1 for the other edges), the solution remains optimal

Solution

Now assume a connected undirected graph $G = (V, E)$ with an edge cost function $c : E \to \mathbb{R}_+$

Take the subgraph $G_{\text{odd}} = (V_{\text{odd}}, E_{\text{odd}})$, where $V_{\text{odd}} \subseteq V$ contains the vertices of odd degree, and E_{odd} contains the edges with both endpoints in V_{odd}

For $v, w \in V_{\text{odd}}$ let $c(v, w)$ be the shortest distance between v and w

Create a bipartite graph $(V_1 \cup V_2, E')$ with $V_1 = \{u_i \mid v_i \in V_{\text{odd}}\}$ and $V_2 = \{w_i \mid v_i \in V_{\text{odd}}\}$, i.e. both vertex sets V_i are disjoint copies of V_{odd}, and

$$E' = \{\{u_i, w_j\} \mid i \neq j \wedge \{v_i, v_j\} \in E_{\text{odd}}\} \text{ with } c(u_i, w_j) = c(v_i, v_j)$$

Let $E'' \subseteq E'$ be a perfect matching with minimal costs $\sum_{e \in E''} c(e)$

Then duplicate all edges e on the shortest paths from v_i to v_j for all $\{u_i, w_j\} \in E''$

Hamilton Graphs

We know that a complete graph G is always a Hamilton graph

Furthermore, we can exploit the following theorem

Theorem. If $G' = (V, E \cup \{\{v, w\}\})$ is a Hamilton graph with $\{v, w\} \notin E$ and $deg(v) + deg(w) \geq |V|$ holds, then also $G = (V, E)$ is a Hamilton graph.

Proof. If there exists a Hamilton cycle in G' containing the edge $\{v, w\}$, then there exists a Hamilton path $v = v_1, v_2, \dots, v_{|V|-1}, v_{|V|} = w$ in G

As $deg(v) + deg(w) \geq |V|$ holds, we find v_i and v_{i+1} with $\{v, v_{i+1}\}, \{v_i, w\} \in E$

Then $v_1, v_{i+1}, \dots, v_{|V|}, v_i, v_{i-1}, \dots, v_1$ is a Hamilton cycle

Hamilton Cycle Algorithm

A pair of edges $e = \{v_1, v_{i+1}\}$ and $e' = \{v_i, v_{|V|}\}$ as used in the proof is called a pair of **crossover edges**

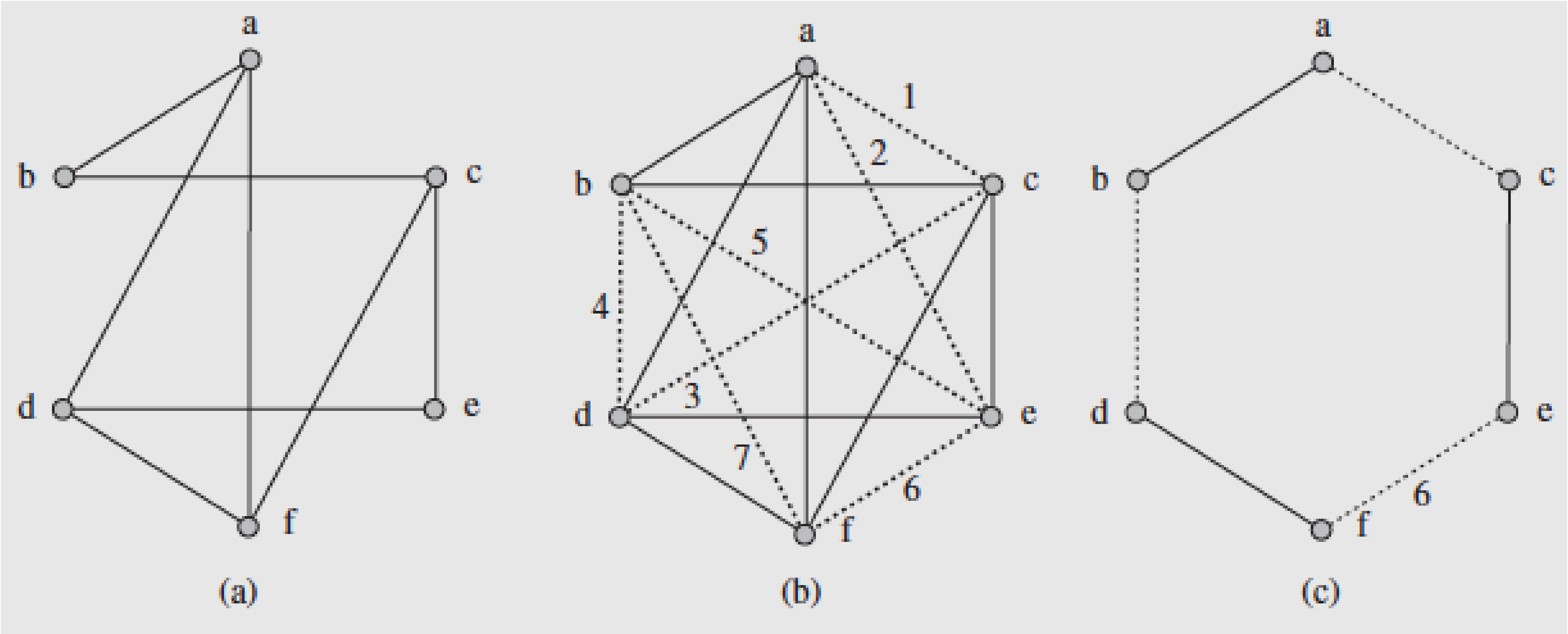
The theorem suggests an easy strategy for checking, whether a graph is Hamiltonian or not

First extend the given graph G by adding edges $\{v, w\}$, whenever $deg(v) + deg(w) \geq |V|$ holds

As this produces a graph with presumably many edges (it may even be a complete graph), it is easy to find a Hamilton cycle in this extended graph (if it exists)

Then successively use pairs of crossover edges (as in the proof above) and use the same replacement as in the proof above to obtain a Hamilton cycle without the additional edges

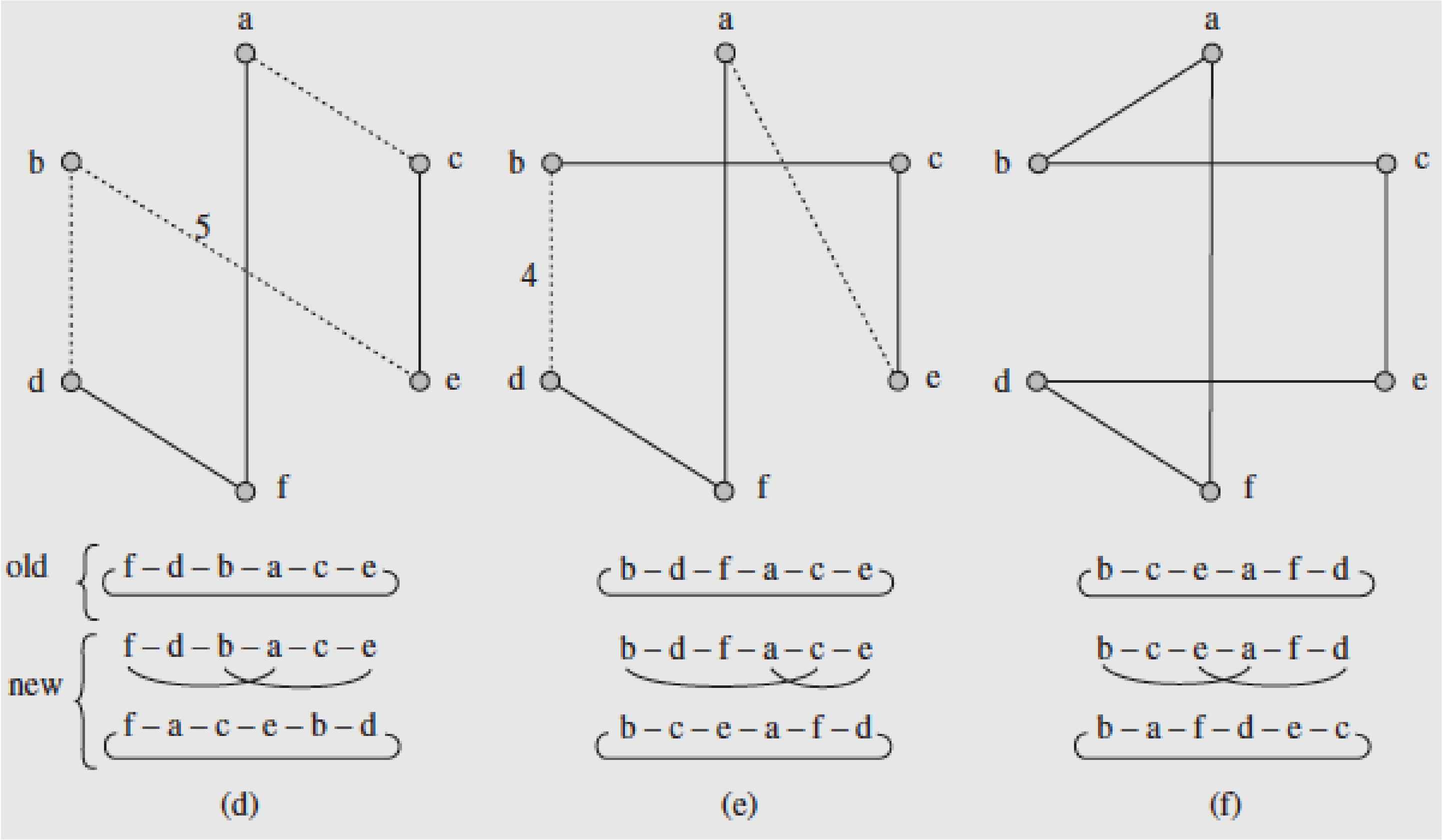
Example / 1



The graph (a) is the input graph, and the graph (b) is the one resulting from the extension of the set of edges

On the right in (c) we have a Hamilton cycle in the graph in (b)

Example / 2



Three steps using pairs of crossover edges to turn the Hamilton cycle in (b) step-by-step into the Hamilton cycle in (f)