Data Structures and Algorithms – W11

Graph Theory part.2
Minimum Spanning Tree
Single Source Shortest Paths
All-pairs shortest paths
Maximum Flow

Contents

- Minimum spanning tree
  - Kruskala's alhorithm
- Single Source Shortest Paths
  - Dijkstra's algorithm
- Minimum spanning tree
  - Prim's algorithm
- All-Pairs Shortest paths
  - Floyda-Warshalla's algorithm
- Flow network
  - Maximum flow
  - Ford-Fulkerson method
  - Edmonds-Karp algorithm
 - Maximum bipartite matching

Minimum spanning tree (MST)

- Given a connected, undirected graph, a **spanning tree** of that graph is a **subgraph** which is a **tree** and **connects all** the vertices together.

- A **minimum spanning tree** or **minimum weight spanning tree** is then a spanning tree with weight less than or equal to the weight of every other spanning tree
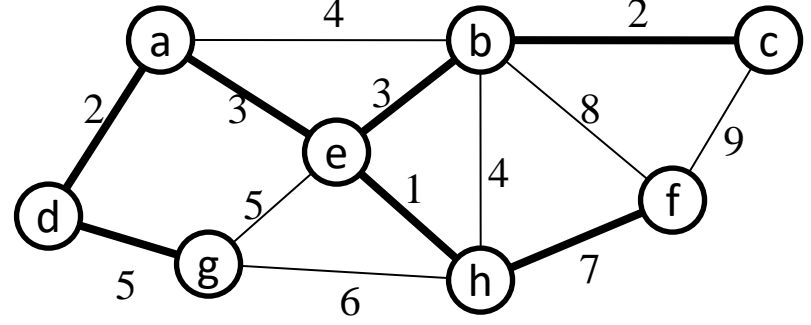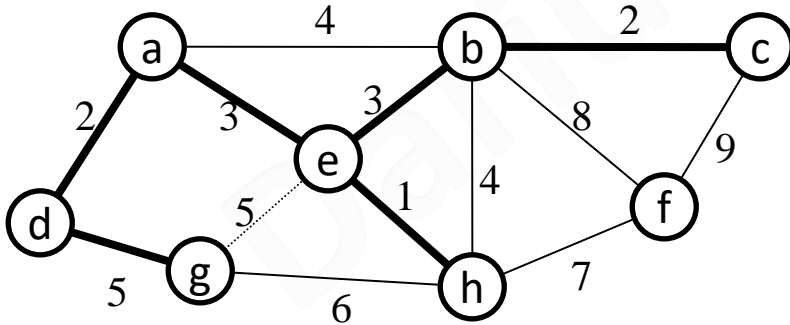

Kruskal's algorithm

1) create a forest F (a set of trees), where each vertex in the graph is a separate tree

2) create a set S containing all the edges in the graph

3) while S is nonempty

    3.1) remove an edge with minimum weight from S

    3.2) if that edge connects two different trees, then add it to the forest, combining two trees into a single tree

    3.3) otherwise discard that edge

Kruskal's algorithm – example 1/2
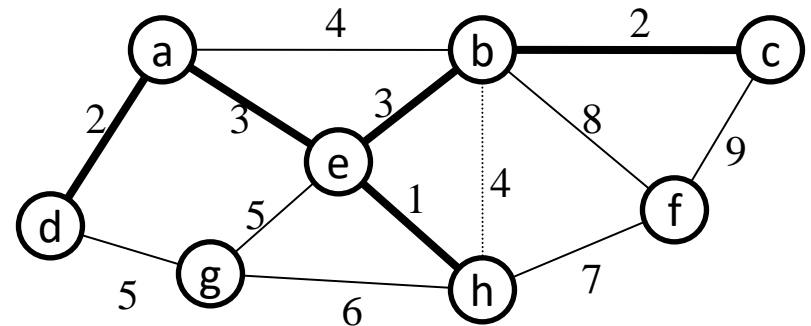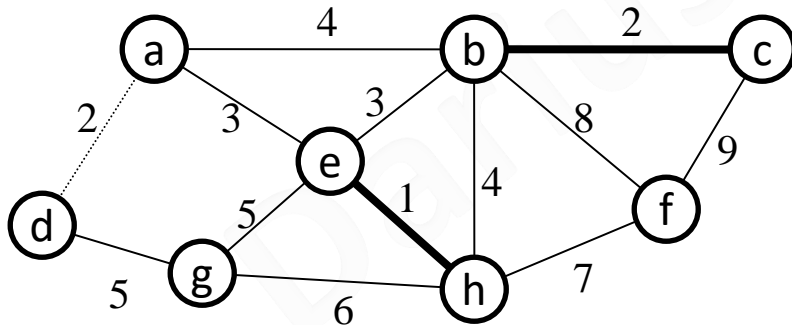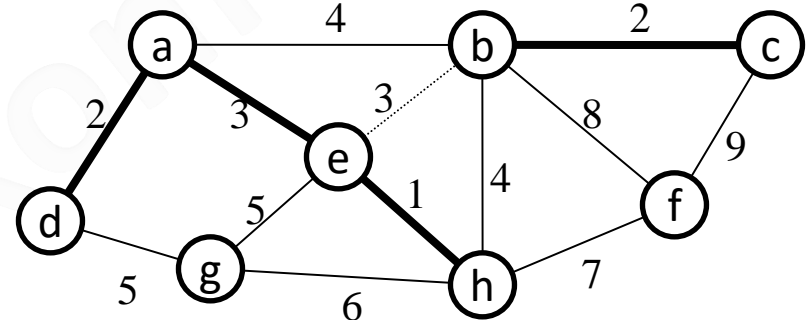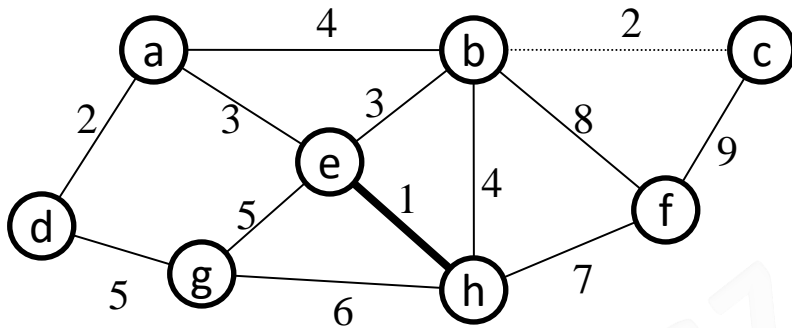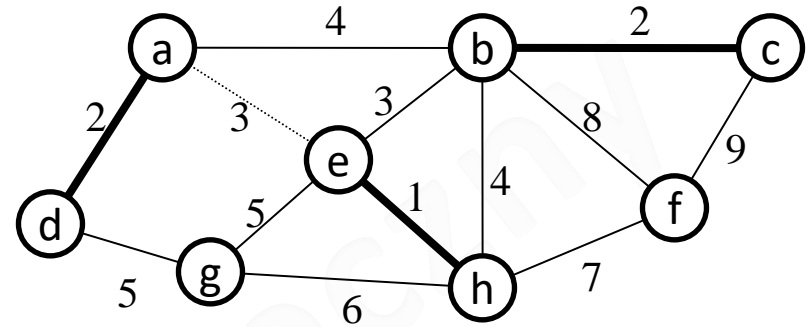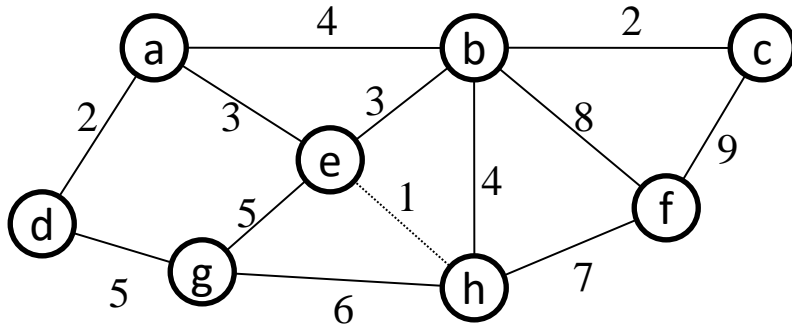
Kruskal's algorithm – example 2/2
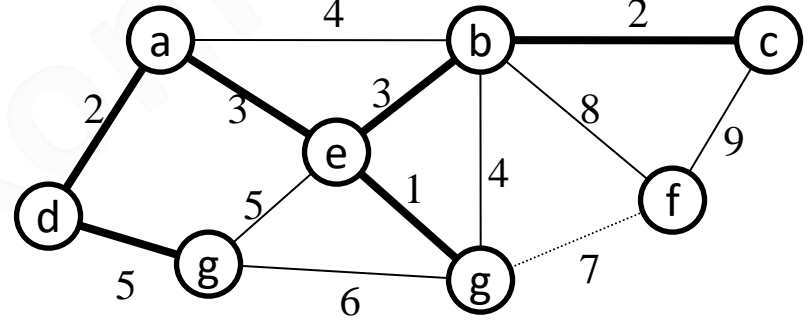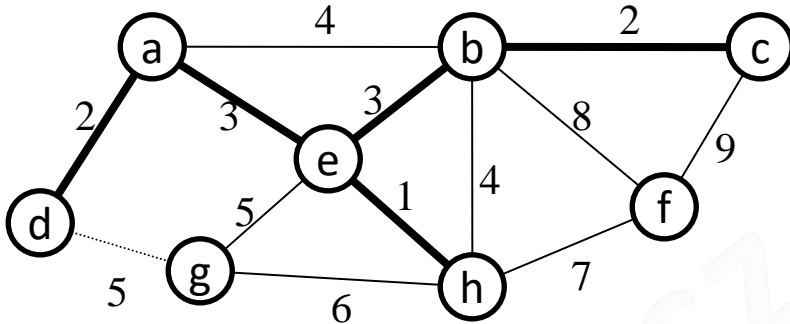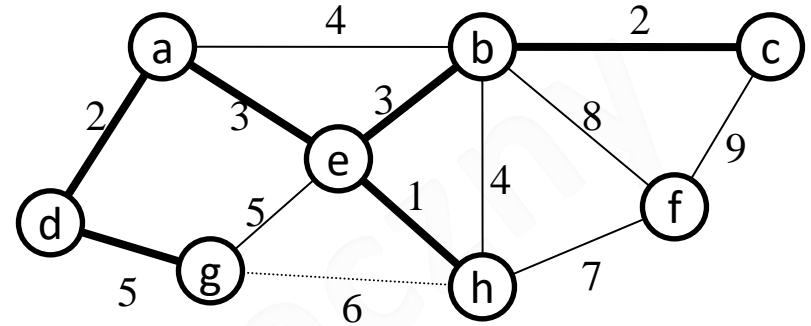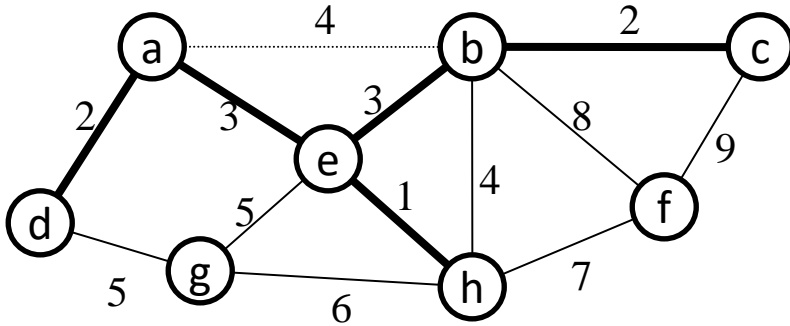
Kruskal's algorithm - code

```
MST(G,w)
{ 1}   A := ∅
{ 2}   for each vertex u∈V[G]
{ 3}       do MakeSet(v)
{ 4}   sort the edges of E into nondecreasing order by weight w
{ 5}   for each edge (u,v)∈E, taken in nondecreasing order by weight
{ 6}         do if FindSet(u) ≠ FindSet(v)
{ 7}             then A:=A ∪ {(u,v)}
{ 8}                     Union(u,v)
{ 9}   return A
```

Complexity (adjacent lists): O($|E|$log$|E|$)

Single-source shortest paths

- The **shortest path problem** is the problem of finding a path between two vertices such that the sum of the weights of its constituent edges is minimized

- The **single-source shortest path problem** is a more general problem, in which we have to find shortest paths from a source vertex v to all other vertices in the graph

```
1.    procedure Dijkstra_Single_Source_SP(V,E,w,s)
2.    begin
3.        V_T := {s};
4.        for all v ∈ (V - V_T) do
5.            if edge(s, v) exists then l[v] := w(s, v);
6.            else set l[v] := ∞;
7.        while V_T ≠ V do
8.        begin
9.            find a vertex u such that l[u] = min{ l[v] | v ∈ (V - V_T)};
10.           V_T := V_T ∪ {u};
11.           for all v ∈ (V - V_T) do
12.               l[v] = min{ l[v], l[u] + w(u, v) };
13.       endwhile
14.   end Dijkstra_Single_Source_SP
```

greedy algorithm

SSSP – example 1/2
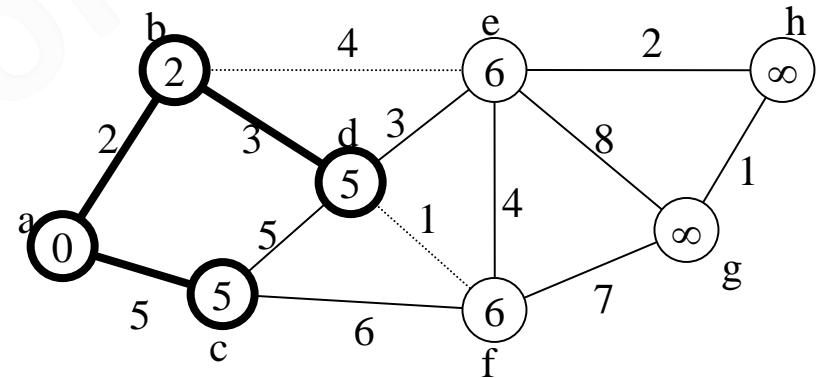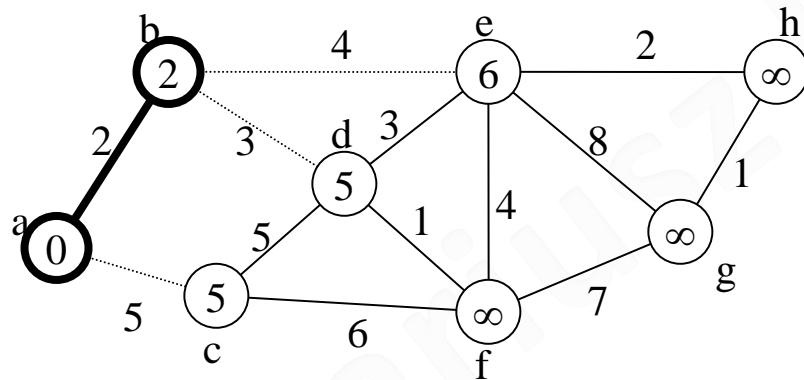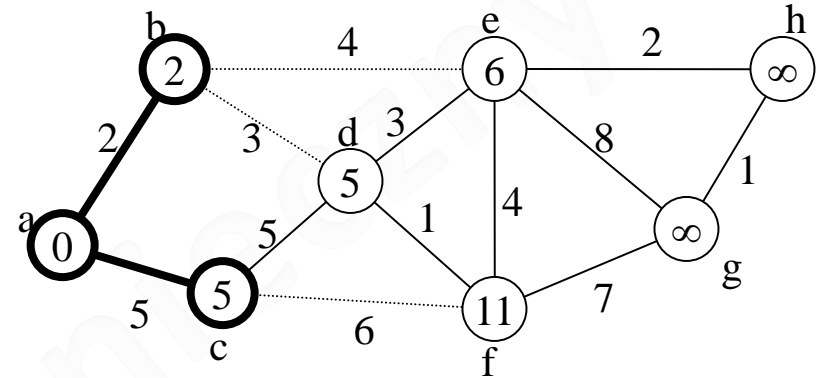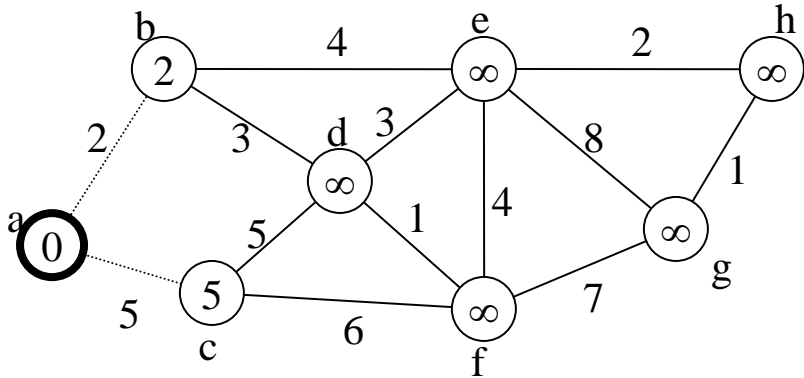
SSSP – example 2/2
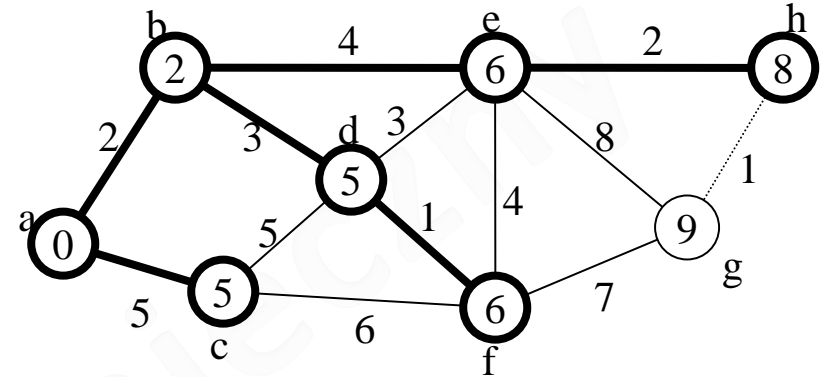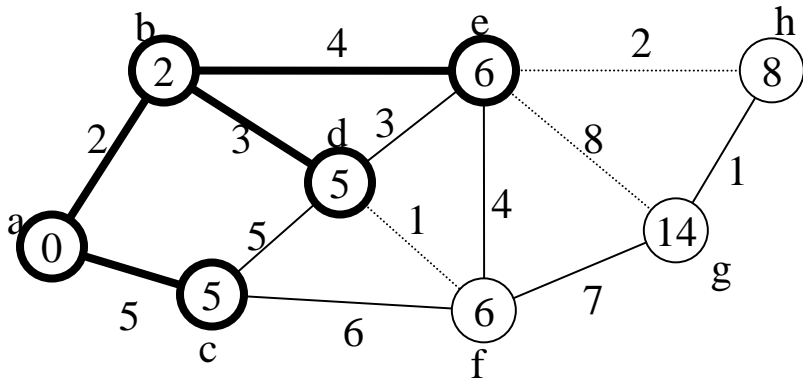
Complexity (binary heap): O(|V| log |V| + |E| log |V|)
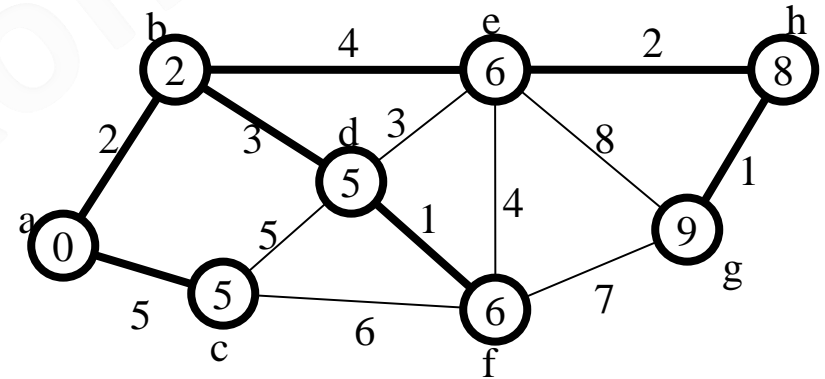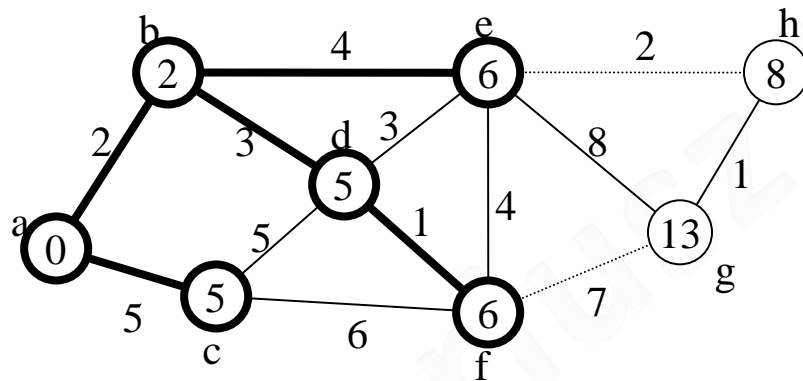
Complexity (Fibonacci heap): O(|V| log |V| + |E|)

MST – Prim's algorithm

- use the same idea as in Dijkstra_Single_Source_SP
- the difference is only in line 12, in which this algorithm take into consideration only the weight of a edge (but not a sum)

```
1.    procedure Prim_MST(V,E,w,s)
2.    begin
3.          V_T := {s};
4.          for all v ∈ (V - V_T) do
5.                if edge(s, v) exists then d[v] := w(s, v);
6.                else set d[v] := ∞;
7.          while V_T ≠ V do
8.          begin
9.                find a vertex u such that d[u] = min{ d[v] | v ∈ (V - V_T)};
10.               V_T := V_T ∪ {u};
11.               for all v ∈ (V - V_T) do
12.                     d[v] = min{ d[v], w(u, v) };
13.         endwhile
14.   end Prim_MST
```

MST Prim's – example 1/2
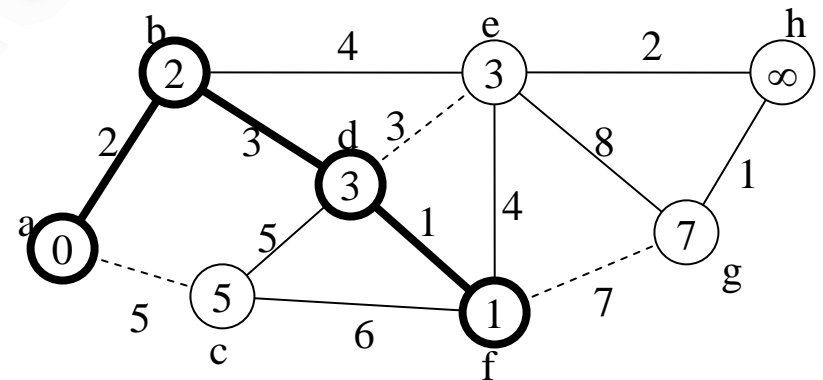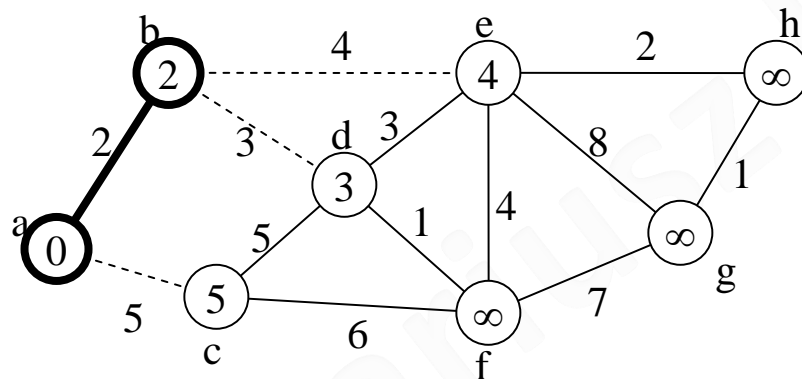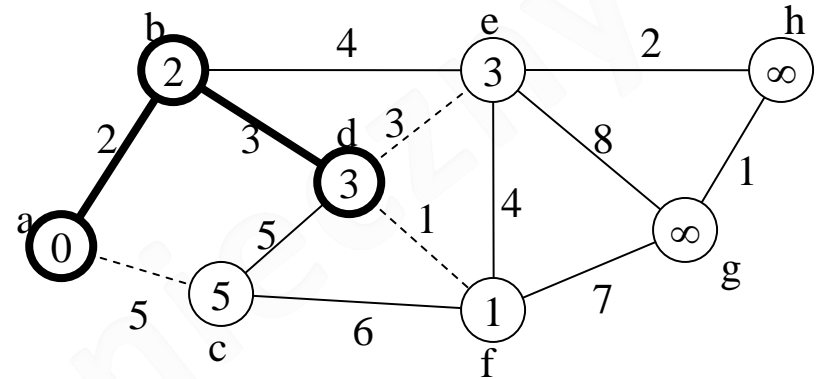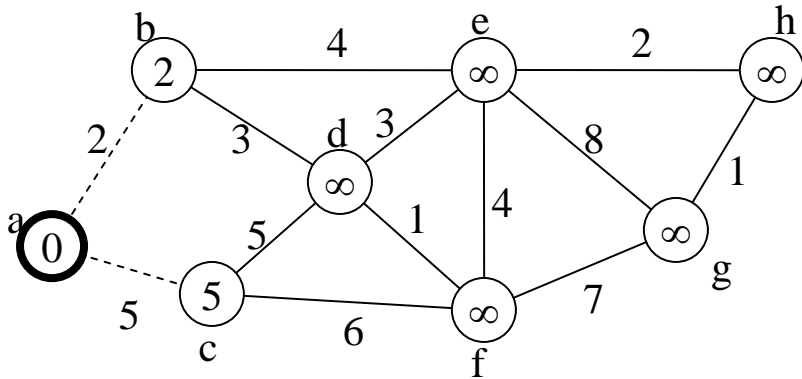
MST Prim's – example 2/2
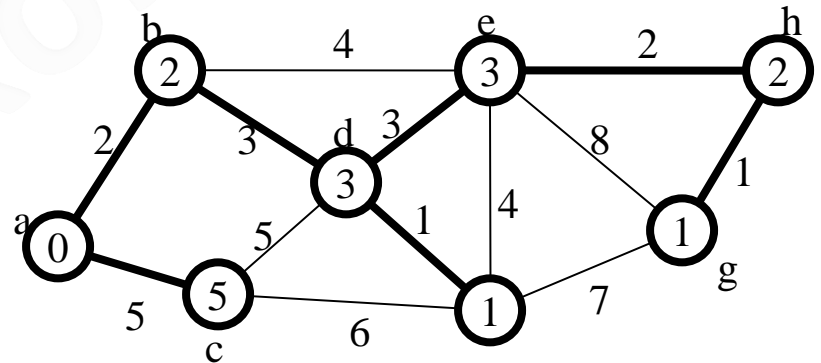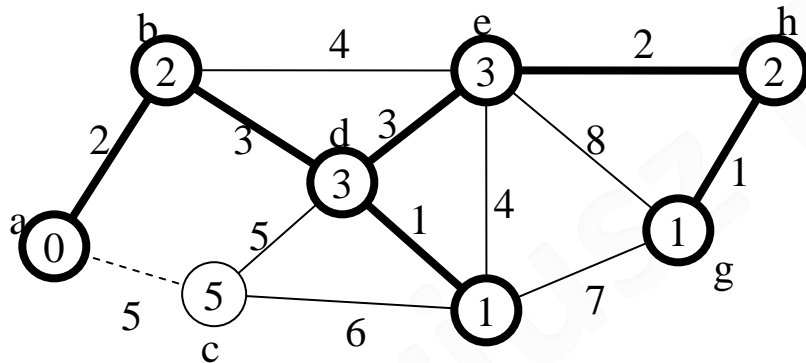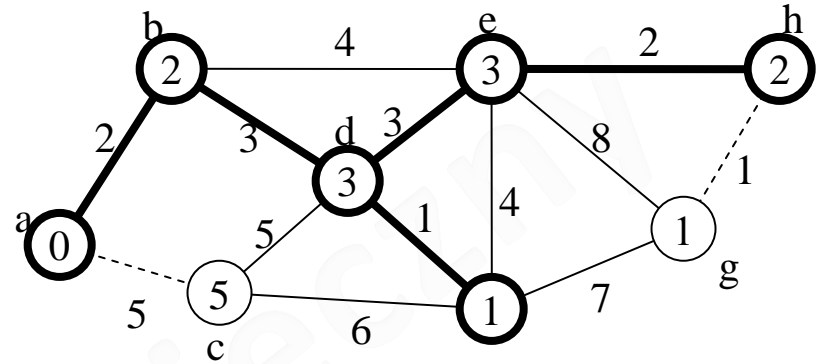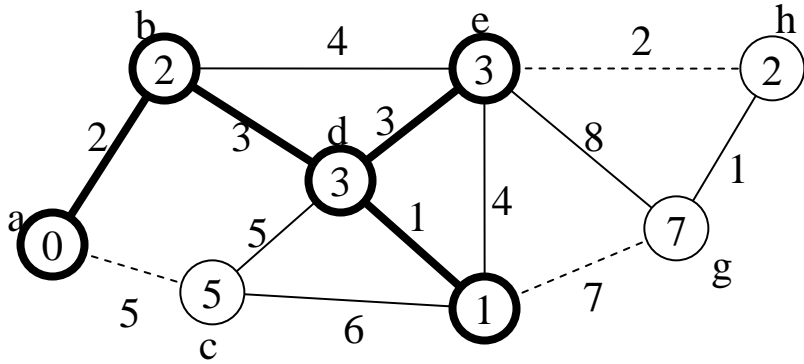
Complexity (binary heap): O(|V| log |V| + |E| log |V|)

Complexity (Fibonacci heap): O(|V| log |V| + |E|)

All-pairs shortest paths

- run single-source shortest paths algorithm for each vertex

  - complexity of Dijkstra's algorithm for single source shortest paths:O(V log V + E)

  - complexity for V vertices: $O(V^2 \log V + V*E)$

- if we have a graph with negative weights (but without negative cycles):

  - Bellman-Ford algorithm single source shortest paths can be used. Complexity: O(VE)

  - complexity for V vertices : $O(V^2 E)$

Floyd's algorithm – structure of the path

Let $p_{ij}^{(k)}$ means shortest path form vertex i to wertex j with intermediate vertices from a set $\{1, 2, ..., k\}$. Let $d_{ij}^{(k)}$ means weight of the path.

all intermediate vertices in $\{1, 2,.., k\text{-}1\}$

all intermediate vertices in $\{1, 2,.., k\text{-}1\}$

all intermediate vertices in $\{1, 2,.., k\text{-}1\}$
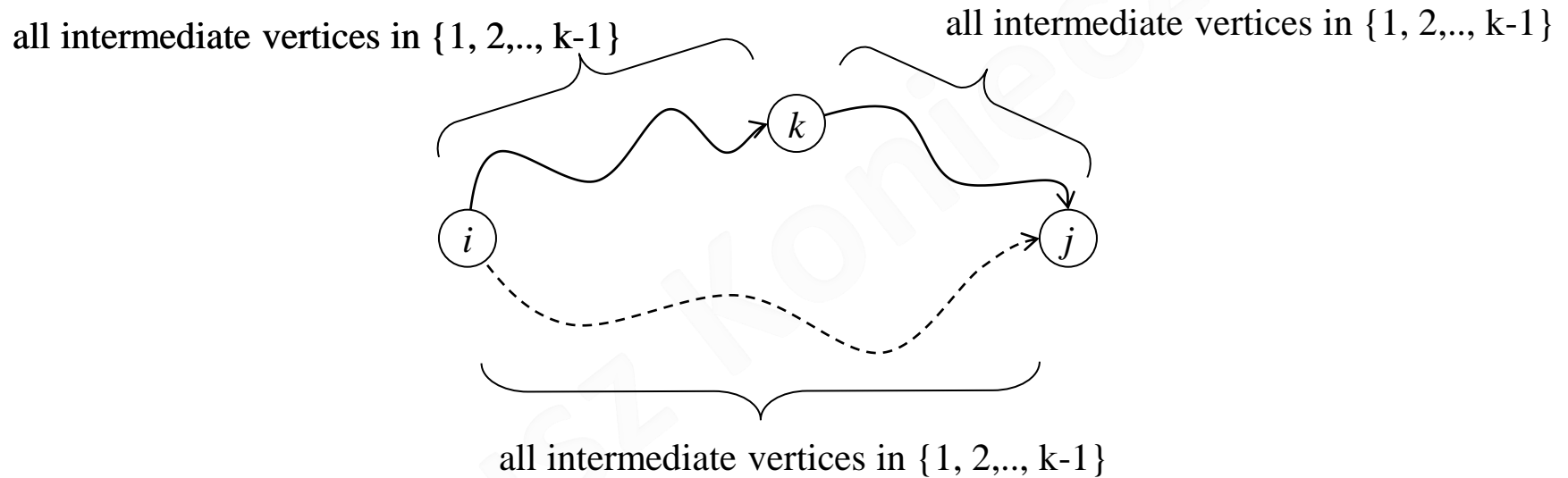
$$d_{ij}^{(k)} = \begin{cases} w_{ij} & if \quad k = 0 \\ \min\left(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\right) & if \quad k \geq 1 \end{cases}$$

dynamic programming

Floyd's algorithm – code, example 1/3

```
1.    procedure FLOYD_ALL_PAIRS_SP(A)
2.    begin
3.        D⁽⁰⁾ = A;
4.        for k = 1 to n do
5.            for i = 1 to n do
6.                for j = 1 to n do
7.                    d⁽ᵏ⁾ᵢ,ⱼ := min( d⁽ᵏ⁻¹⁾ᵢ,ⱼ, d⁽ᵏ⁻¹⁾ᵢ,ₖ + d⁽ᵏ⁻¹⁾ₖ,ⱼ );
8.    end FLOYD_ALL_PAIRS_SP
```

Line 3: $D^{(0)} = A;$

Line 7: $d^{(k)}_{i,j} := \min(d^{(k-1)}_{i,j},\ d^{(k-1)}_{i,k} + d^{(k-1)}_{k,j});$

complexity:
$O(V^3)$

$$D^{(0)} = \begin{cases} 0 & 2 & 5 & \infty & \infty & \infty & \infty & \infty \\ 2 & 0 & \infty & 3 & 4 & \infty & \infty & \infty \\ 5 & \infty & 0 & 5 & \infty & 6 & \infty & \infty \\ \infty & 3 & 5 & 0 & 3 & 1 & \infty & \infty \\ \infty & 4 & \infty & 3 & 0 & 4 & 8 & 2 \\ \infty & \infty & 6 & 1 & 4 & 0 & 7 & \infty \\ \infty & \infty & \infty & \infty & 8 & 7 & 0 & 1 \\ \infty & \infty & \infty & \infty & 2 & \infty & 1 & 0 \end{cases}$$

Floyd's algorithm – example 2/3

$$D^{(1)} = \begin{cases} 0 & 2 & 5 & \infty & \infty & \infty & \infty & \infty \\ 2 & 0 & 7 & 3 & 4 & \infty & \infty & \infty \\ 5 & 7 & 0 & 5 & \infty & 6 & \infty & \infty \\ \infty & 3 & 5 & 0 & 3 & 1 & \infty & \infty \\ \infty & 4 & \infty & 3 & 0 & 4 & 8 & 2 \\ \infty & \infty & 6 & 1 & 4 & 0 & 7 & \infty \\ \infty & \infty & \infty & \infty & 8 & 7 & 0 & 1 \\ \infty & \infty & \infty & \infty & 2 & \infty & 1 & 0 \end{cases}$$

$$D^{(2)} = \begin{cases} 0 & 2 & 5 & 5 & 6 & \infty & \infty & \infty \\ 2 & 0 & 7 & 3 & 4 & \infty & \infty & \infty \\ 5 & 7 & 0 & 5 & 11 & 6 & \infty & \infty \\ 5 & 3 & 5 & 0 & 3 & 1 & \infty & \infty \\ 6 & 4 & 11 & 3 & 0 & 4 & 8 & 2 \\ \infty & \infty & 6 & 1 & 4 & 0 & 7 & \infty \\ \infty & \infty & \infty & \infty & 8 & 7 & 0 & 1 \\ \infty & \infty & \infty & \infty & 2 & \infty & 1 & 0 \end{cases}$$

$$D^{(3)} = \begin{cases} - & - & - & - & - & - & - & - \\ - & - & - & - & - & - & - & - \\ - & - & - & - & - & - & - & - \\ - & - & - & - & - & - & - & - \\ - & - & - & - & - & - & - & - \\ - & - & - & - & - & - & - & - \\ - & - & - & - & - & - & - & - \\ - & - & - & - & - & - & - & - \end{cases}$$

$$D^{(4)} = \begin{cases} - & - & - & - & - & - & - & - \\ - & - & - & - & - & - & - & - \\ - & - & - & - & - & - & - & - \\ - & - & - & - & - & - & - & - \\ - & - & - & - & - & - & - & - \\ - & - & - & - & - & - & - & - \\ - & - & - & - & - & - & - & - \\ - & - & - & - & - & - & - & - \end{cases}$$

Floyd's algorithm – example 3/3

$$D^{(5)} = \left\{ \begin{array}{ccccccc} - & - & - & - & - & - & - \\ - & - & - & - & - & - & - \\ - & - & - & - & - & - & - \\ - & - & - & - & - & - & - \\ - & - & - & - & - & - & - \\ - & - & - & - & - & - & - \\ - & - & - & - & - & - & - \end{array} \right.$$

$$D^{(6)} = \left\{ \begin{array}{ccccccc} - & - & - & - & - & - & - \\ - & - & - & - & - & - & - \\ - & - & - & - & - & - & - \\ - & - & - & - & - & - & - \\ - & - & - & - & - & - & - \\ - & - & - & - & - & - & - \\ - & - & - & - & - & - & - \end{array} \right.$$

$$D^{(7)} = \left\{ \begin{array}{ccccccc} - & - & - & - & - & - & - \\ - & - & - & - & - & - & - \\ - & - & - & - & - & - & - \\ - & - & - & - & - & - & - \\ - & - & - & - & - & - & - \\ - & - & - & - & - & - & - \\ - & - & - & - & - & - & - \end{array} \right.$$

$$D^{(8)} = \left\{ \begin{array}{ccccccc} - & - & - & - & - & - & - \\ - & - & - & - & - & - & - \\ - & - & - & - & - & - & - \\ - & - & - & - & - & - & - \\ - & - & - & - & - & - & - \\ - & - & - & - & - & - & - \\ - & - & - & - & - & - & - \end{array} \right.$$

Flow network

- Flow network G=(V,E) is a directed graph in which each edge $(u,v) \in E$ has a nonnegative capacity $c(u,v) \geq 0$. If $(u,v) \notin E$, we assume that $c(u,v)=0$. We distinguish two vertices in a flow network: a **source** s and a **sink** t. For convenience, we assume that every vertex lies on some path from the source to the sink. That is, for every vertex $v \in V$, there is a path $(s,...,v,...,t)$. The graph is therefore connected, and $|E| \geq |V| - 1$
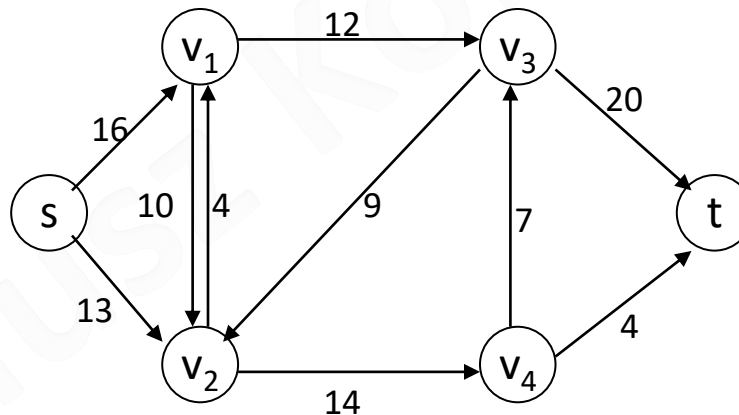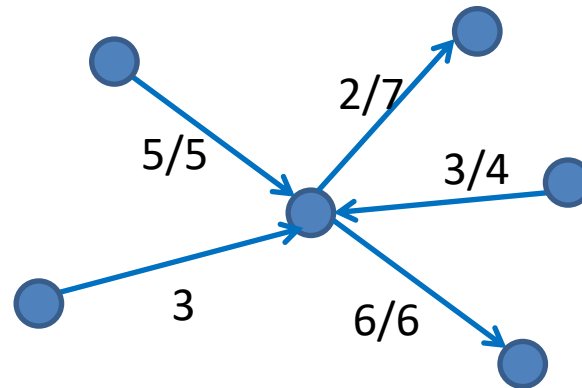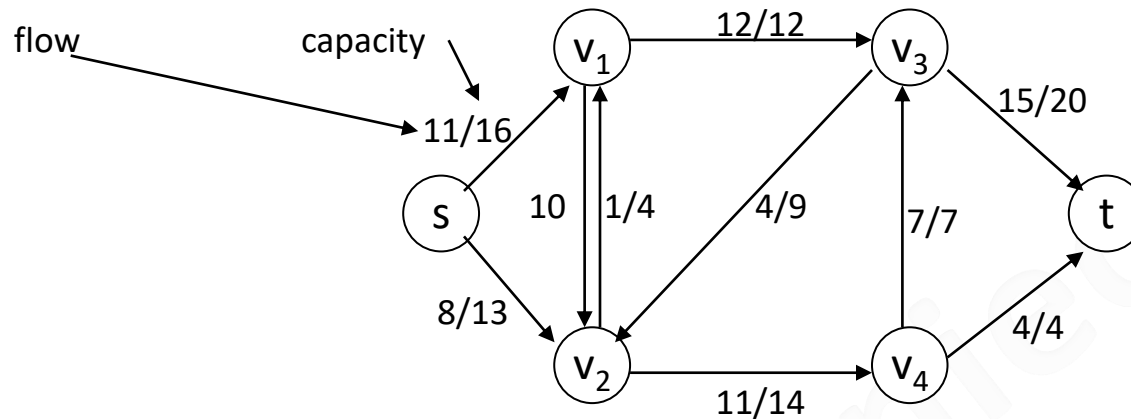
Flow

- Let $G = (V, E)$ be a flow network with a **capacity** function c. Let s be the source of the network, and let t be the sink. A ***flow*** in G is a real-valued function $f : V \times V \rightarrow$ **R** that satisfies the following three properties:

  - **Capacity constraint:** For all $u, v \in V$, we require $f(u, v) \leq c(u, v)$.
  - **Skew symmetry:** For all $u, v \in V$, we require $f(u, v) = -f(v, u)$.
  - **Flow conservation:** For all $u \in V - \{s, t\}$, we require

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

2/7

5/5

3/4

3

6/6

Flow - example

flow   capacity

v_1 12/12 v_3

11/16

15/20

s   10   1/4   4/9   7/7   t

8/13

v_2 4/4

11/14 v_4

The quantity $f(u, v)$, which can be positive, zero, or negative, is called the ***flow*** from vertex u to vertex v. The ***value*** of a flow f is defined as

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

that is, the total flow out of the source. (Here, the $|\cdot|$ notation denotes flow value, not absolute value or cardinality.)

In the ***maximum-flow problem***, we are given a flow network G with source s and sink t, and we wish to **find** a **flow** of **maximum value**.

Networks with multiple sources and sinks

- We can reduce the problem of determining a maximum flow in a network with multiple sources and multiple sinks to an ordinary maximum-flow problem

- We add **super-source** s' and **super-sink** t' vertices.
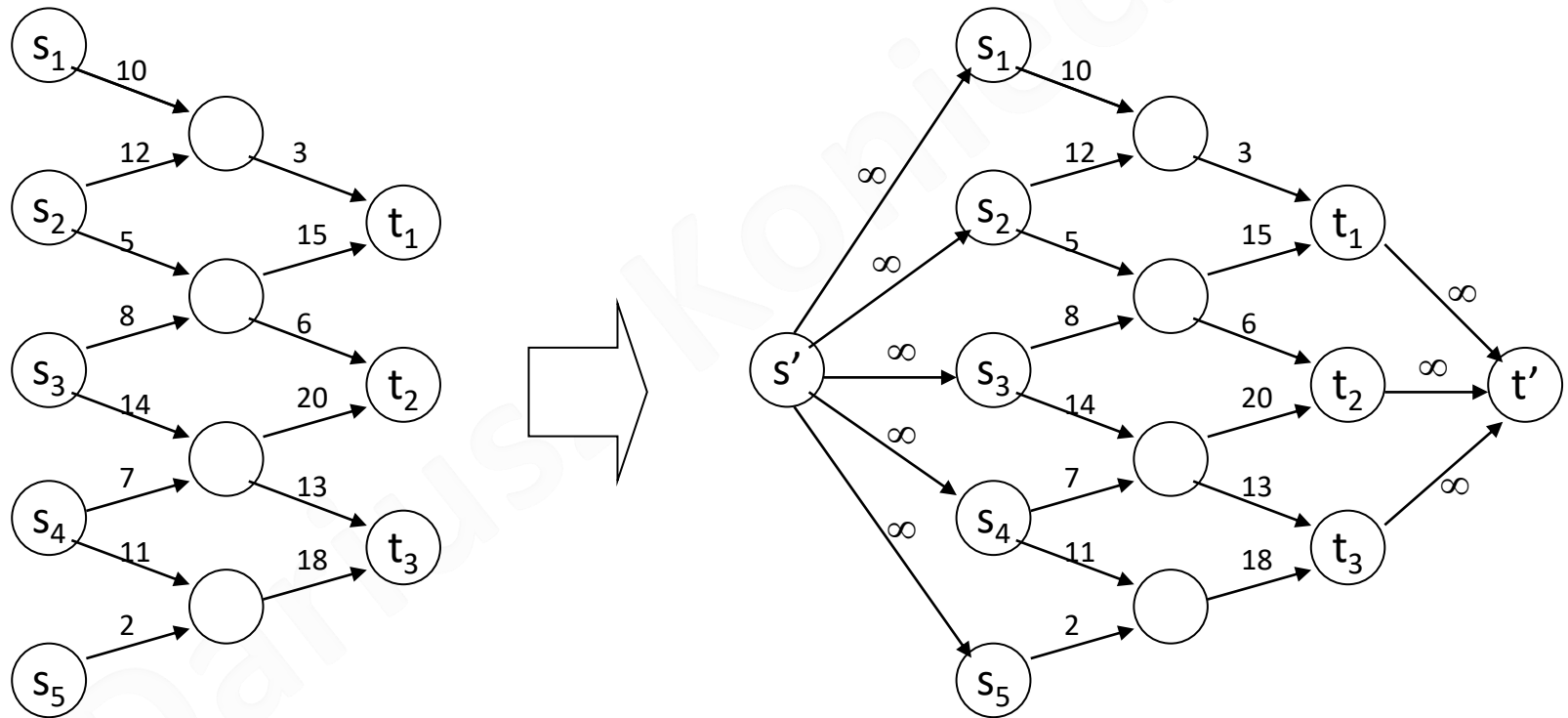
Cancelation
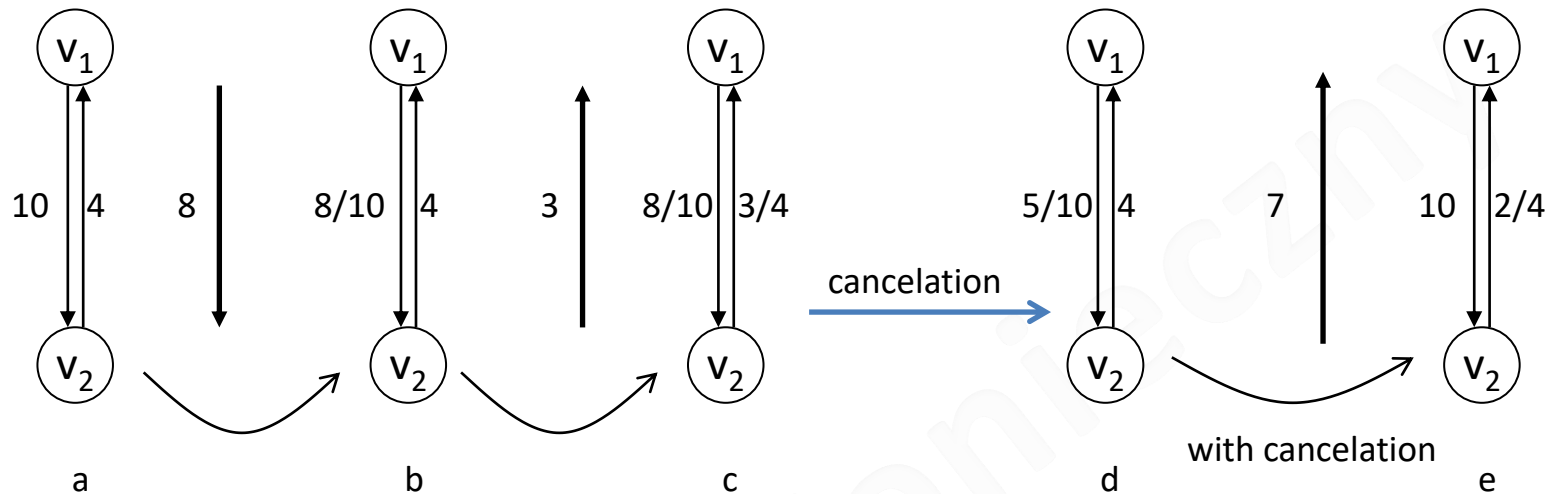
- a) Initial state with capacity
- b) First we send from v_1 to v_2 8 containers per day (c/d).
- c) A new order where we have to send from v_2 to v_1 3 c/d.
- d) Cancelation of flow in reverse directions of the size 3 c/d.
- e) In an another order we have to send 7 c/d from v_1 to v_2. The flow after cancelation.

Cancelation operation does not violate property of flow.

The Ford-Fulkerson method

- The Ford-Fulkerson method is iterative. We start with $f(u, v) = 0$ for all $u, v \in V$, giving an initial flow of value 0. At each iteration, we increase the flow value by finding an "augmenting path", which we can think of simply as a path from the source s to the sink t along which we can send more flow, and then augmenting the flow along this path. We repeat this process until no augmenting path can be found.

```
FORD-FULKERSON-METHOD(G, s, t)
{ 1}  initialize flow f to 0
{ 2}  while there exists an augmenting path p do
{ 3}        augment flow f along p
{ 4}  return f
```

Residual networks
Intuitively, given a flow network and a flow, the residual network consists of edges that can admit more flow. More formally, suppose that we have a flow network $G = (V, E)$ with source s and sink t. Let f be a flow in G, and consider a pair of vertices $u, v \in V$. The amount of *additional* flow we can push from u to v before exceeding the capacity $c(u, v)$ is the ***residual capacity*** of (u, v), given by

$$c_f(u,v) = c(u,v) - f(u,v)$$

Residual network

Given a flow network $G = (V, E)$ and a flow f, the ***residual network*** of G induced by f is $G_f = (V, E_f)$, where

$$E_f = \{(u, v) \in V \times V : c_f(u, v) > 0\}$$
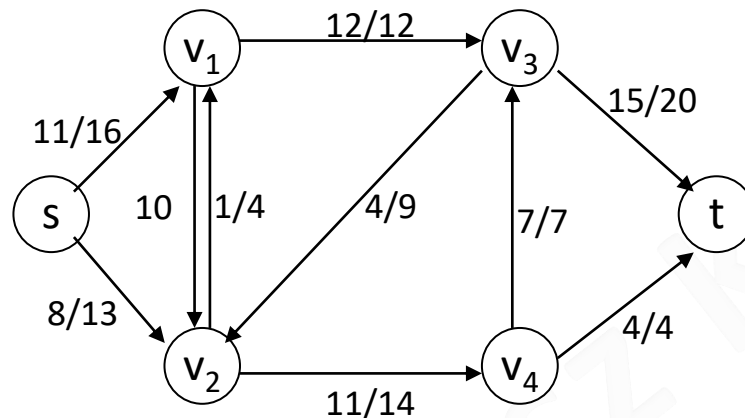
flow
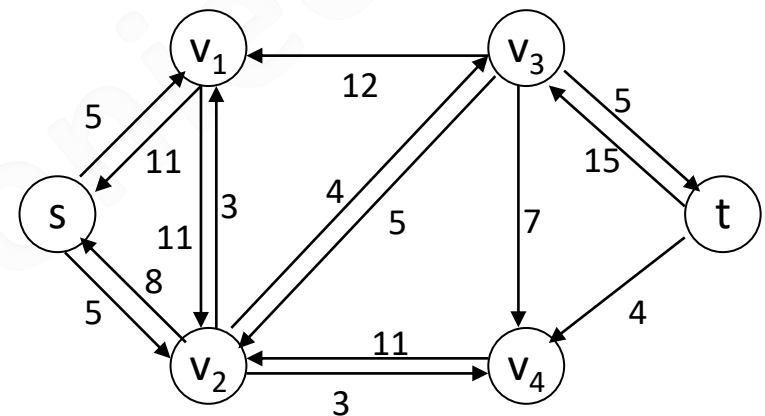
residual network

Let $G = (V, E)$ be a flow network with source s and sink t, and let f be a flow in G. Let G_f be the residual network of G induced by f, and let f' be a flow in G_f. Then the flow sum $f + f'$ defined by equation is a flow in G with value $|f + f'| = |f| + |f'|$.

Augmenting paths

- Given a flow network $G = (V, E)$ and a flow f, an **augmenting path** p is a simple path from s to t in the residual network G_f. By the definition of the residual network, each edge (u, v) on an augmenting path admits some additional positive flow from u to v without violating the capacity constraint on the edge.
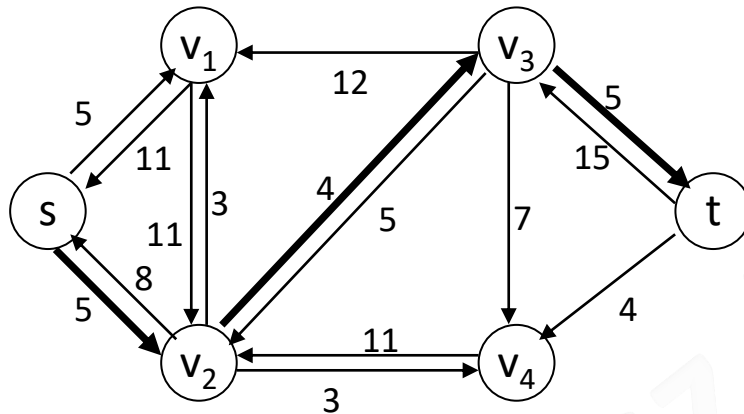
residual network
with augmenting path
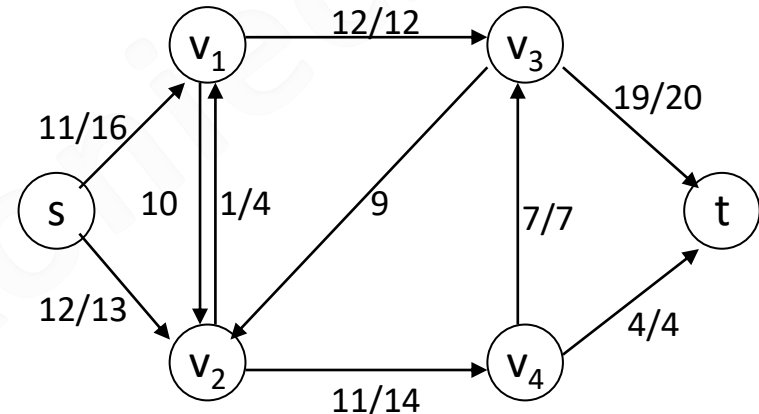
new flow that result from adding
residual capacity on augmenting path

We call the maximum amount by which we can increase the flow on each edge in an augmenting path p the **residual capacity** of p, given by

$$c_f(p) = \min\{c_f(u,v) : (u,v) \text{ is on } p\}$$

The basic Ford-Fulkerson algorithm

- In each iteration of the Ford-Fulkerson method, we find *some* augmenting path p and increase the flow f on each edge of p by the residual capacity $c_f(p)$. The presented implementation of the method computes the maximum flow in a graph $G = (V, E)$ by updating the flow $f[u, v]$ between each pair u, v of vertices that are connected by an edge. If u and v are not connected by an edge in either direction, we assume implicitly that $f[u, v] = 0$
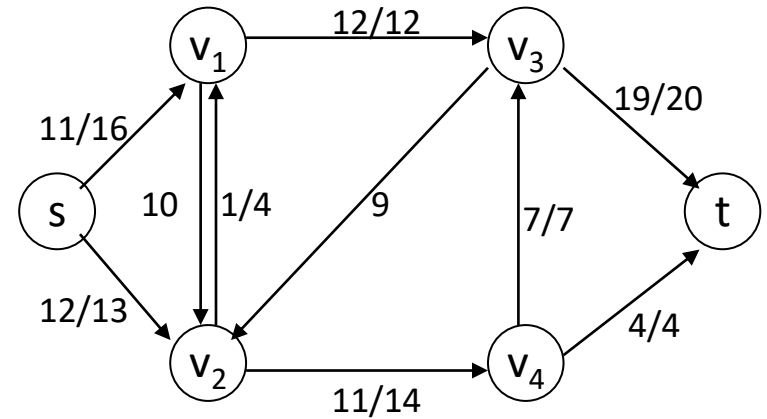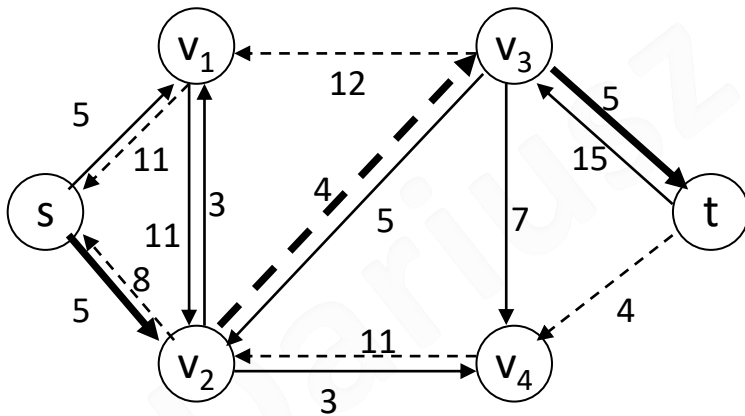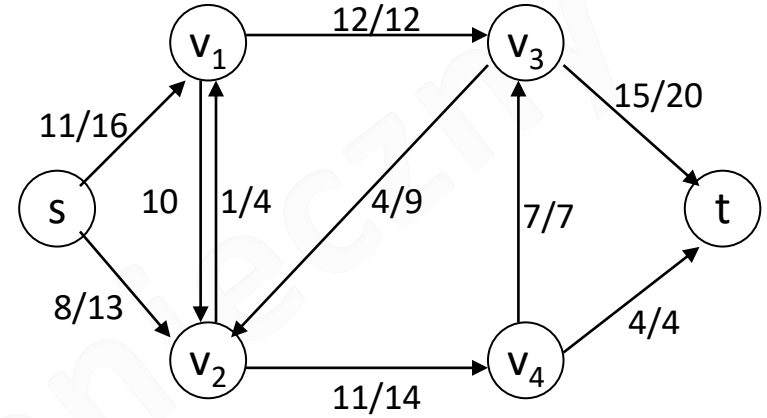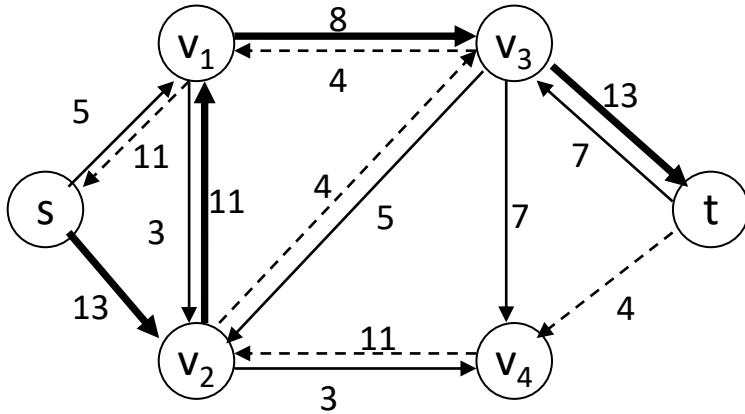
```
          FORD-FULKERSON(G, s, t)
{ 1}  for each edge (u, v)∈E[G] do
{ 2}          f[u, v]:=0
{ 3}          f[v, u]:=0
{ 4}  while there exists a path p from s to t in the residual network Gf do
{ 5}     cf(p):=min {cf(u, v) : (u, v) is in p}
{ 6}        for each edge (u, v) in p
{ 7}           do f[u, v]:=f[u, v] + cf(p)
{ 8}              f[v, u]:=-f[u, v]
```
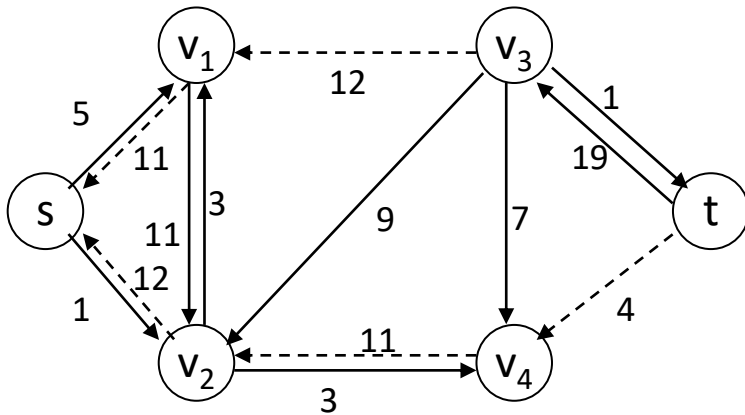
- We can use depth-first search to find an augmenting path.

Example 1/2
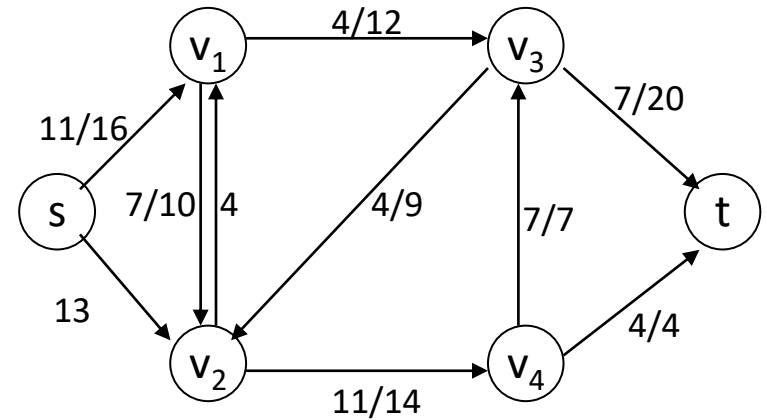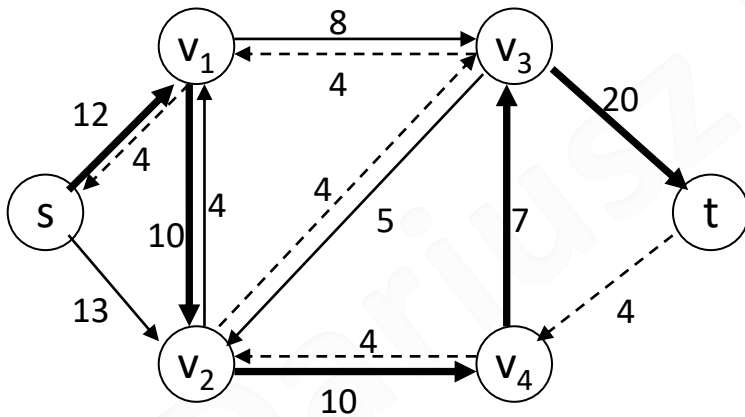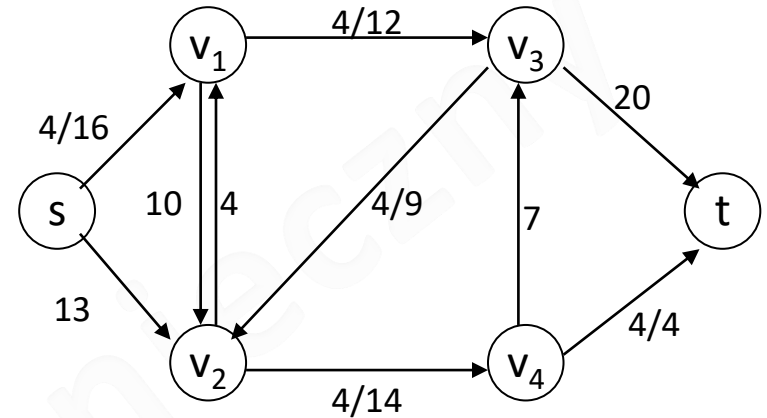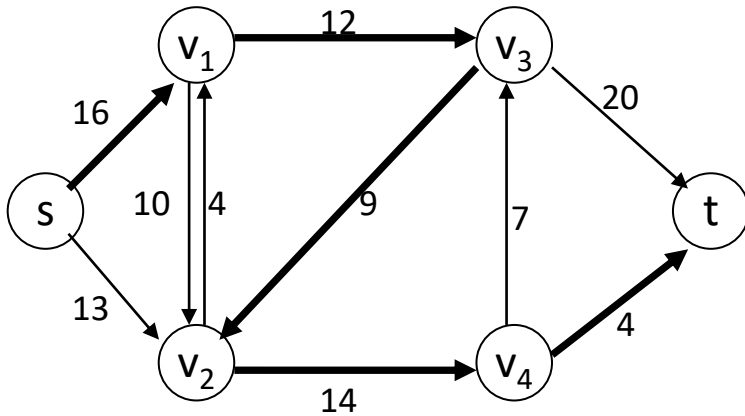
Example 2/2
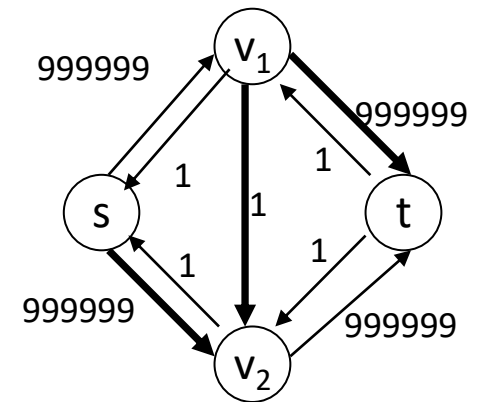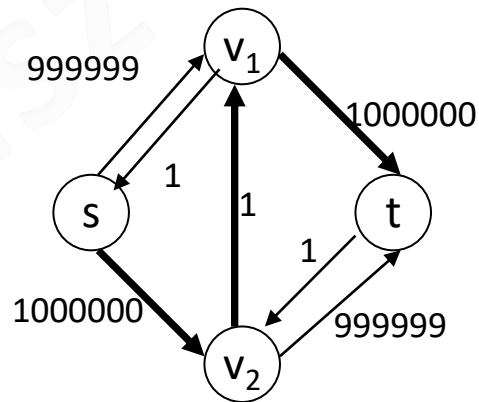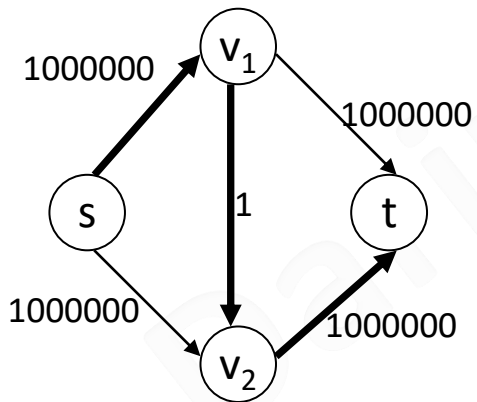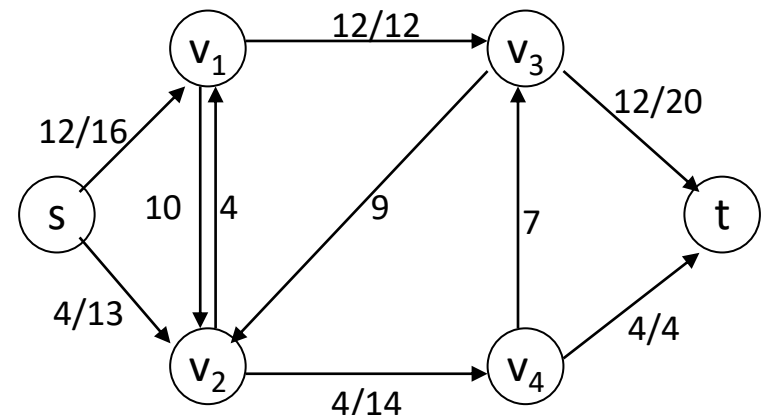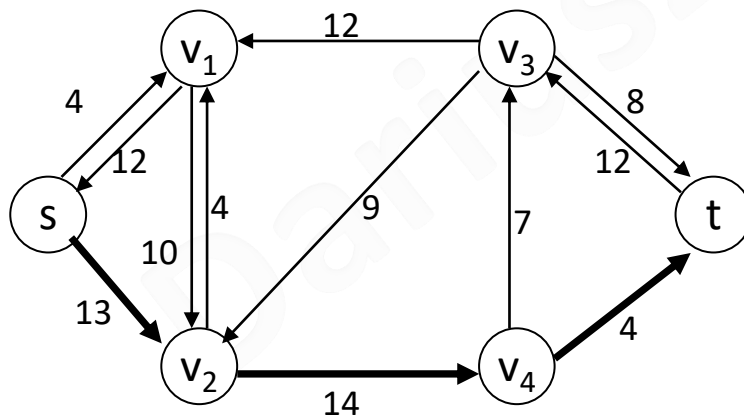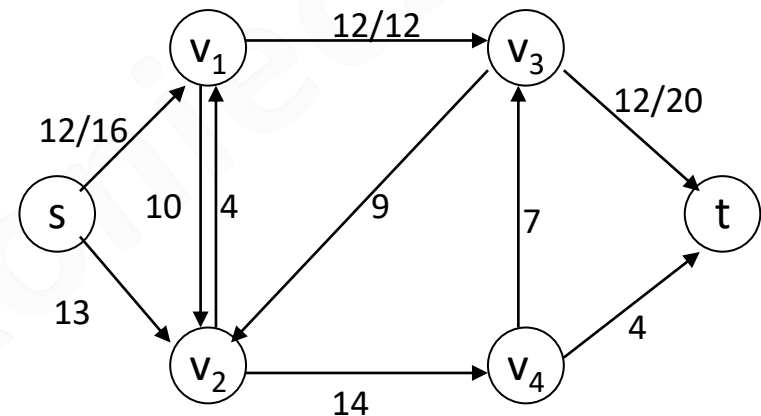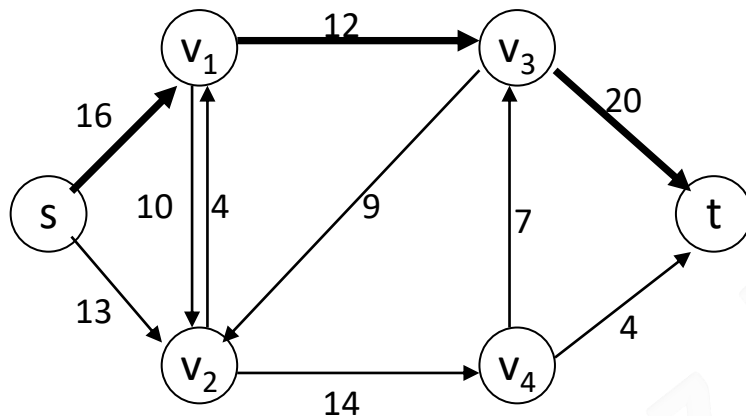
Analysis

- complexity:
 - $O(|f|V)$

flow value

Edmonds-Karp algorithm, example 1/2

- The bound on FORD-FULKERSON can be improved if we implement the computation of the augmenting path p in line 4 with a breadth-first search, that is, if the augmenting path is a *shortest* path from s to t in the residual network, where each edge has unit distance (weight). We call the Ford-Fulkerson method so implemented the ***Edmonds-Karp algorithm***.

Example 2/2, analysis
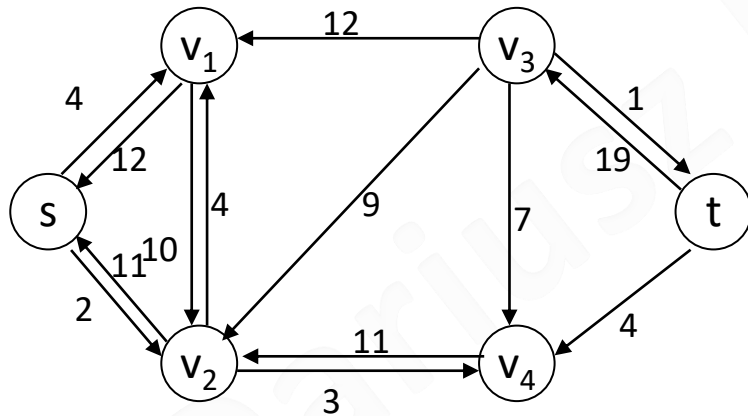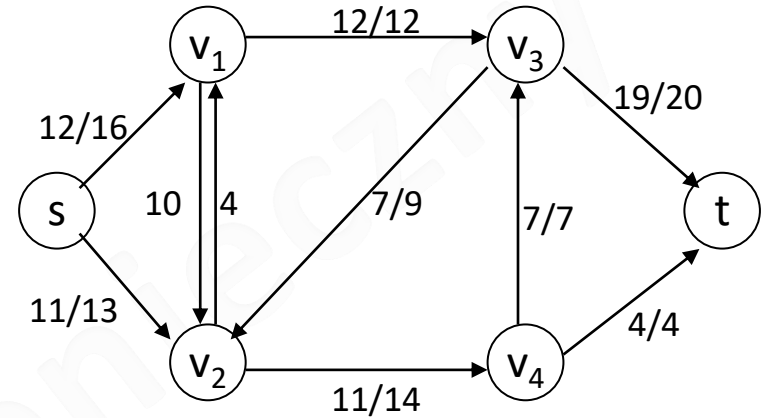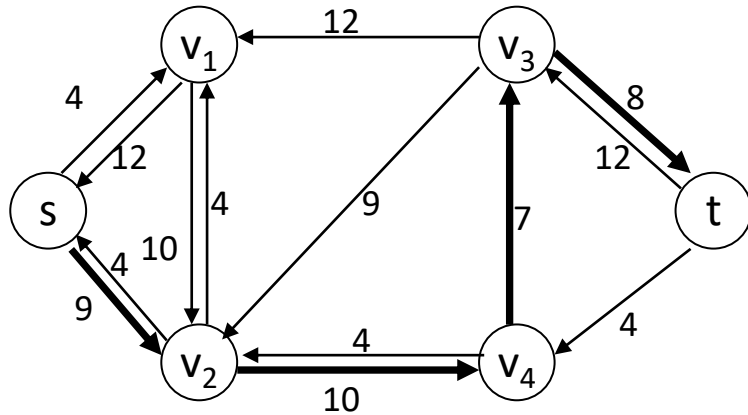
- complexity:
 - $O(VE^2)$

Matching in graph

- Given an undirected graph $G = (V, E)$, a ***matching*** is a subset of edges $M \subseteq E$ such that for all vertices $v \in V$, at most one edge of M is incident on v. We say that a vertex $v \in V$ is ***matched*** by matching M if some edge in M is incident on v; otherwise, v is ***unmatched***. A ***maximum matching*** is a matching of maximum cardinality, that is, a matching M such that for any matching M', we have $|M| \geq |M'|$
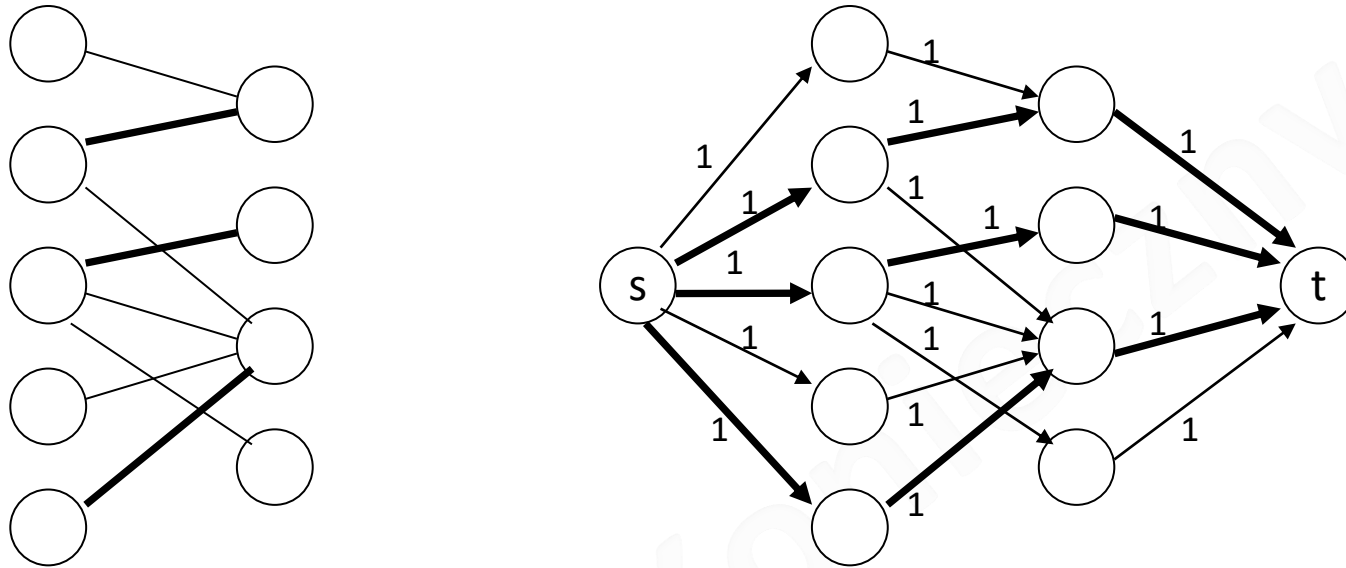
matching
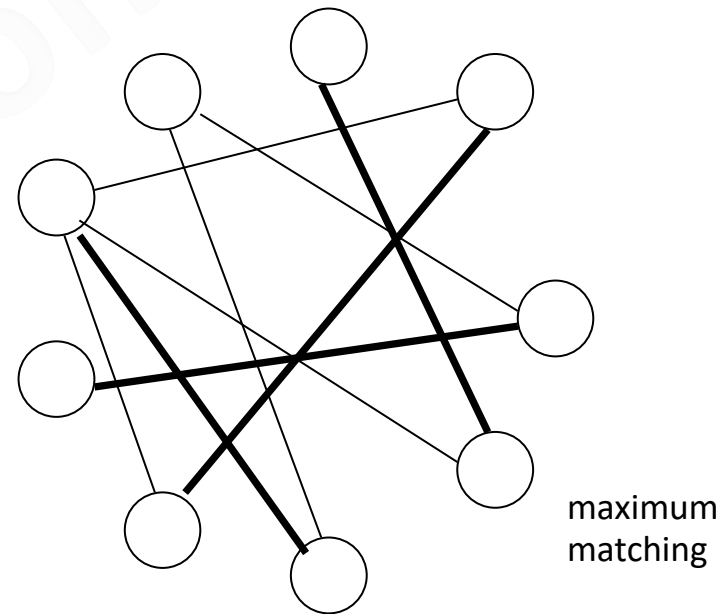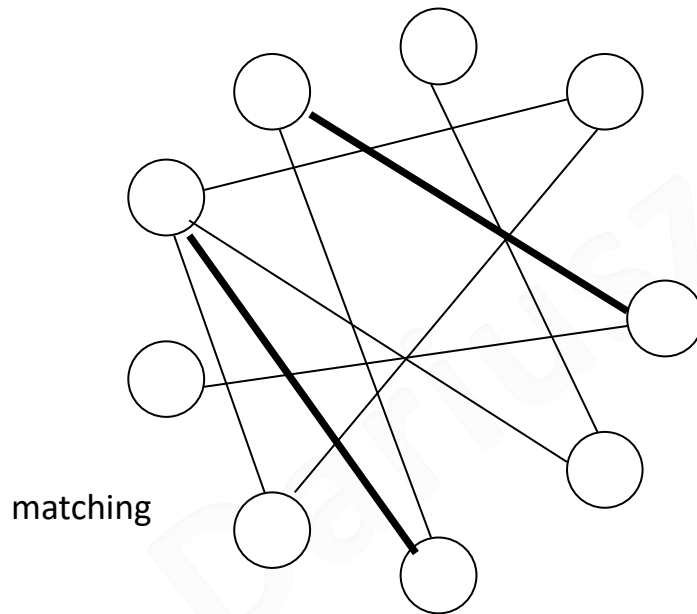
maximum matching

Maximum bipartite matching

- We assume that the vertex set can be partitioned into $V = L \cup R$, where L and R are disjoint and all edges in E go between L and R. We further assume that every vertex in V has at least one incident edge.


L

R

maximum matching
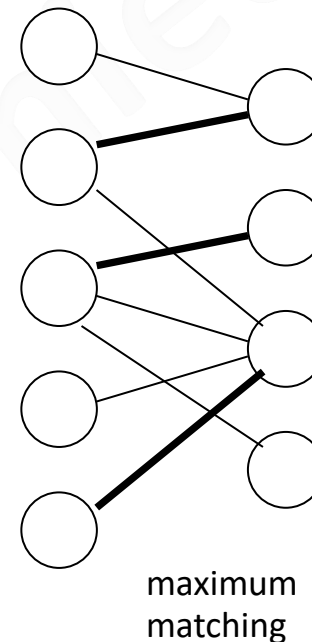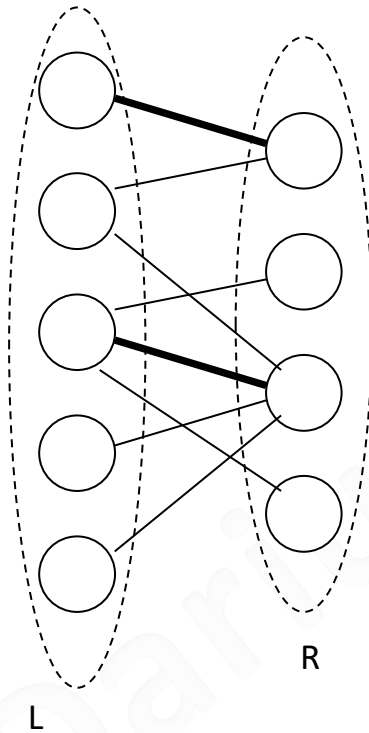
Corresponding flow network

We define the ***corresponding flow network*** G' = (V', E') for the bipartite graph G as follows. We let the source s and sink t be new vertices not in V, and we let V' = V ∪ {s, t}. If the vertex partition of G is V = L ∪ R, the directed edges of G' are the edges of E, directed from L to R, along with V new edges:

$$E' = \{(s,u): u \in L\} \cup \{(u,v): u \in L \wedge v \in R \wedge (u,v) \in E\} \cup \{(v,t): v \in R\}$$

complexity:O(VE)

Cuts of flow networks

- A **cut** (S, T) of flow network $G = (V, E)$ is a partition of V into S and $T = V - S$ such that $s \in S$ and $t \in T$. (If f is a flow, then the **net flow** across the cut (S, T) is defined to be $f(S, T)$. The **capacity** of the cut (S, T) is $c(S, T)$. A **minimum cut** of a network is a cut whose capacity is minimum over all cuts of the network.
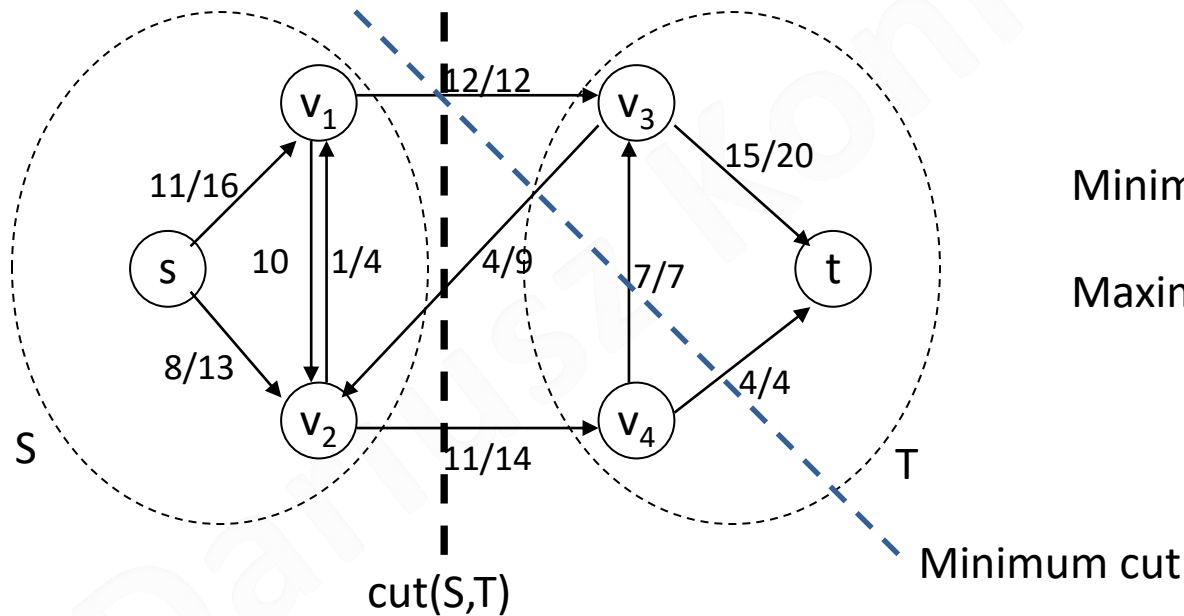
Minimum cut problem
||
Maximum flow problem

cut(S,T)

Minimum cut

$f(S,T)=f(\{s,v_1,v_2\},\{v_3,v_4,t\})=12-4+11=19$ $c(S,T)=12+14=26$