Graph Foundations of Network Modelling

Abhishek Dixit

1

As was already indicated in the introductory chapter, a lot of problems encountered in the area of communication networks essentially come down to the solving of graph-theoretical problems. In this chapter, these graph foundations will be presented. Some fundamental techniques will be presented to solve elementary graph problems. Some applications will be highlighted during the chapter, where these techniques can be applicable. Of course, more communication network applications will be presented during the exercise and lab sessions as well.

Outline

0. Introduction
1. Definition and notations
   - 1.1 Graph
   - 1.2 Digraph
2. Minimum spanning tree problem
3. Shortest path problem
   - 3.1 in unweighted (di)graphs
   - 3.2 in weighted (di)graphs (positive weights)
   - 3.3 in weighted (di)graphs (general weights)
   - 3.4 all-pairs shortest path

On this slide and the next slide, an overview is given of the chapter on graph foundations of network modelling.
Section 1 will introduce some basic terminology. The two mathematical structures that will be used throughout this chapter - (undirected) graphs and (directed) digraphs - are described.
In section 2 the minimum spanning tree problem will be discussed. An algorithm will be presented to solve this problem: the algorithm of Kruskal.
Section 3 will deal with the shortest path problem in graphs. Four different situations will be considered, leading to four different solution approaches.

Outline

4. Maximum flow problem
   - 4.1 Network flow
   - 4.2 Problem definition
   - 4.3 Residual network concept
   - 4.4 Flow augmenting path algorithm
   - 4.5 Max-flow min-cut theorem
   - 4.6 Application: calculation of connectivity
5. Minimum cost flow problem
   - 5.1 Introduction
   - 5.2 Problem definition
   - 5.3 Residual network
   - 5.4 Busacker-Gowen algorithm
   - 5.5 Several supply/demand vertices
6. Extension: multi-commodity flow problems

Section 4 deals with the maximum flow problem. After this section, we will be able to solve problems like the following. Consider an oil distribution system. Large oil depots ('servers') and pumping stations ('clients') are connected via a network of pipelines. Each pipeline has a limited capacity: only a limited volume of oil can be pumped through the pipe in a unit of time. We want to calculate the maximum flow (in litres/second) that can be pumped through the network from oil depot s to pumping station t. This section also presents the max- flow min-cut theorem. From this theorem an algorithm will be deduced to calculate the connectivity of a graph.

In section 5 we will discuss the minimum cost flow problem. In this case the links will have a limited capacity and also a cost associated to each link, representing the cost to transport one flow unit over this link. The goal is to find the cheapest way to route a certain amount of traffic in the network, while respecting the capacity restrictions.

Finally, the concept of single and multiple commodities to be conveyed through one network is introduced in section 6 and the multi-commodity extensions of the flow problems in sections 1 to 5 are described.

Outline

0. **Introduction**
1. Definition and notations
2. Minimum spanning tree problem
3. Shortest path problem
4. Maximum flow problem
5. Minimum cost flow problem
6. Extension: multi-commodity flow problems

Why graph foundations in a network course?

efficient transfer of information across a communication network

network flow problem

flow algorithm

Examples :

- shortest path problem
- maximum flow problem
- minimum cost flow problem

Graph foundations of network modelling 5

In communication networks, we typically want to move some entity (e.g. a packet in an IP-network) from one point to another in the network in an efficient way: we want to provide good services to the users of the network and we want to use the underlying (and typically expensive) transmission facilities as efficient as possible. In this chapter, we will model the data of different network applications as mathematical models - better known as *network flow problems* - and study the various ways - better known as *algorithms* - to find the solution(s) for the resulting models.

Some basic network flow problems are the consequence of some questions which are ever-recurring in practice:

1. What is the best (cheapest, fastest, most reliable) way to traverse a network to get from one point to another? The network flow problem associated with this question is the *shortest path problem*, and different ways to solve it will be discussed in section 3.

2. The transmission facilities of a communication network have limited capacity (e.g. an STM-4 link is able to transfer 4 VC-4 network connections, but not more). In a network with limited arc capacities, we want to know how to send as much flow as possible from point s to point t in the network, while respecting the arc capacities. The network flow problem associated with this question is the *maximum flow problem*, which will be discussed in section 4.

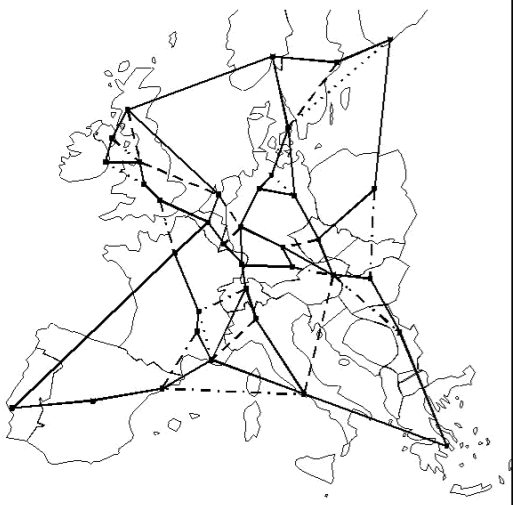
3. The most fundamental of all network flow problems is the *minimum cost flow problem.* This problem can be described as follows. We consider a network with limited arc capacities. In order to satisfy a given demand, we want to send units of an entity (e.g. VC-4 containers) through this capacitated network. How can we send these entities as cheap as possible through the network? This will be discussed in section 5.

Abstract modelling of communication networks

What must be modeled ?

all the relevant data of the real network:

- topology
- transfer of information
- different parameters
  - cost of equipment
  - capacity of equipment
  - ...



Graph foundations of network modelling 6

A good abstract model for a communication network must be able to represent the most important features of the real network:

1. the network *topology* (i.e. we need to know which switching elements are directly connected by a link)

2. the *transfer of information* through the network (e.g. the flow of VC-4 containers in an SDH-network or the flow of the packets in an IP-network)

3. the different *parameters* which are important for a specific problem (e.g. if we want to know how many VC-4 containers can be transported from one point in the network to another, our model must certainly contain a parameter denoting the capacity of the link, i.e. the sum of the capacities of the different STM-N line systems on that link)

In order to solve network problems efficiently, we must also ensure that our model is not too complicated (e.g. our mathematical model needs no details about the internal functionality of switching elements).

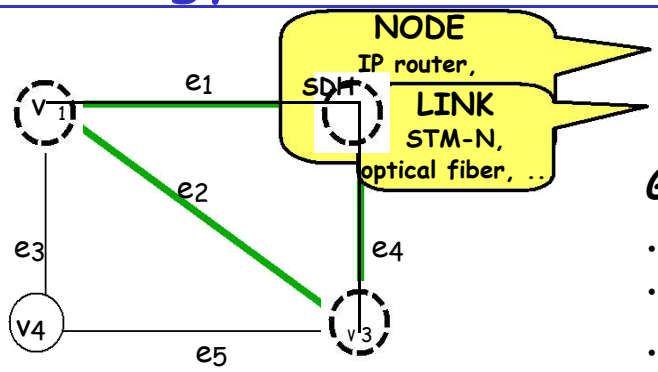
Note : Although we will concentrate in this chapter on the application of graph algorithms in communications applications, it is important to note that a lot of these algorithms can also be used in other areas than communication networks. A few examples are :

- navigation systems in cars : route planning (shortest route, fastest route, cheapest route, ...)
- planning of logistics : how to provision different sites of a company as efficient as possible
- design of other network types, such as road networks, electricity networks, oil distribution networks, etc.

Outline

0. Introduction
1. **Definition and notations**
   - 1.1 **Graph**
   - 1.2 **Digraph**
2. Minimum spanning tree problem
3. Shortest path problem
4. Maximum flow problem
5. Minimum cost flow problem
6. Extension: multi-commodity flow problems

Terminology and notations



$G = (V, E)$

- graph order p = #V = 4
- graph size q = #E = 5
- G is a (p, q) - graph
- e = uv

- incidence: I (v_1) = {e_1,e_2,e_3}
- adjacency: A(v_2) = {v_1,v_3}
- degree: δ(v_3) = 3

Subgraph G[W] : subset W of vertices of G + edges incident to vertices of subgraph W only

example : {v1, v2, v3}

==> {e1, e2, e4}

Graph foundations of network modelling 8

A *graph G = (V,E)* consists of a vertex set V and an edge set E. Each edge e = {u,v} is an unordered pair of vertices u,v(*). The order of a graph is the number of vertices and the size of a graph is the number of edges. A graph with order p and size q is called a (p,q)-graph.

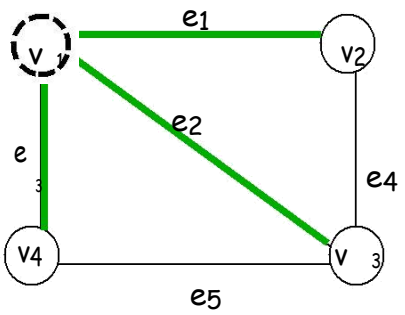
If e = {u,v} is an edge of the graph G, then e and u are incident (e and v are also incident). The incidence of a vertex is the set of edges incident to it. The degree of a vertex is the number of edges incident to it.

If e = {u,v} is an edge of the graph G, then u and v are adjacent vertices. The adjacency of a vertex is the set of vertices adjacent to it.

A *subgraph* of G = (V,E) is a graph whose vertex set is a subset of V and whose edge set is a subset of E. A special case of such a subgraph is a subgraph induced by a subset $W \subset V$ is a graph whose vertex set is W and whose edge set contains all edges of E which are incident to vertices in W only (i.e. an edge e = uv is element of the edge set of the subgraph if and only if $u \in W$ and $v \in W$). The notation we will use for the subgraph of G induced by W is G[W].

(*) Remark : In this chapter, it is always assumed that a graph contains no parallel edges (i.e. at most one edge can connect two vertices with each other). It is also assumed that no loop edges exist (i.e. an edge is never connecting a vertex with itself).

Representation



➡ incidence matrix of G

edges (q)

I_G =

1	1	1	0	0
1	0	0	1	0
0	1	0	1	1
0	0	1	0	

vertices (p)

- I_G (v,e)=1 if v and e are inciden
- I_G (v,e)= 0 otherwise

➡ incidence lists of G

L_G = ({1,2,3}, {1,4}, {2,4,5}, {3,5})

Graph foundations of network modelling 9

A graph can be represented in many ways. Two important representations are:

1. incidence matrix

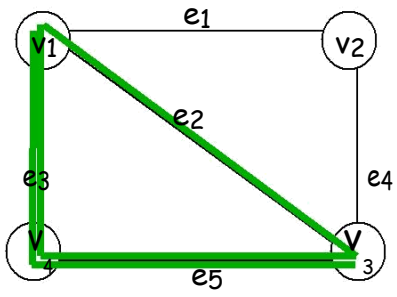
The incidence matrix I_G of a (p,q)-graph is defined as follows. I_G is a (p,q)-matrix with I_G(v,e) = 1 if v and e are incident and I_G(v,e) = 0 otherwise.

2. incidence lists

For each vertex v of G, the incidence of v is given.

Paths and Cycles

walk $w = (v_0, v_1, \ldots, v_{n-1}, v_n)$ such that for $i = 0, 1, \ldots, n-1$, $v_iv_{i+1} \in E$



Special types of walks : paths / cycles

(v1, v4, v3) is a path with length 2

Walk with all vertices distinct

(v1, v4, v3, v1) is a cycle with length 3

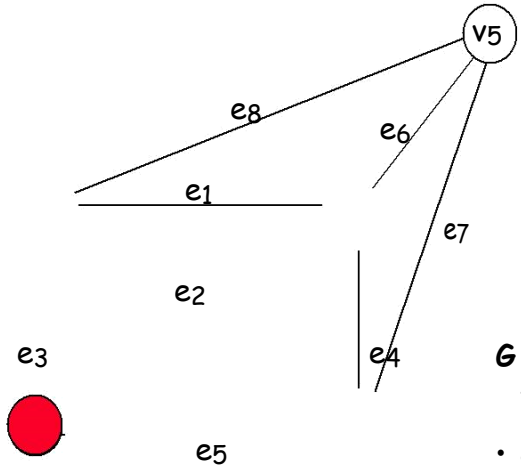
Walk with all vertices but the first and the last distinct

Graph foundations of network modelling 10

A *walk* in a graph G is a sequence $w = (v_0, v_1, \ldots, v_{n-1}, v_n)$ such that for $i = 0 \ldots n-1$, $v_iv_{i+1} \in E$. Note that the same vertex or edge may appear more than once in the walk.

A *path* is a walk in which all vertices are distinct. A *cycle* is a walk in which the first and the last vertex are the same, but all other vertices are distinct.

Cut and Cutset



G is connected : ∀ vertex pairs {u, v}, a path exists between u and v

- **cut, e.g. {e2, e3, e5}**
- **cutset (= minimal cut), e.g. {e3, e5}**
- **vertex-cut, e.g. {v1, v2, v3}**
- **vertex-cutset (= minimal vertex-cut), e.g. {v1, v3}**

Graph foundations of network modelling 11

Consider a graph G and two vertices u,v in G. We say that *u is connected* to v if there exists a u,v-path in G. We say that *G is connected* if for every pair of vertices u,v of G, u is connected to v. (Special case of this definition : the so-called trivial graph (i.e. a graph with one vertex and no edges) is connected.)

A *cut* (edge-cut) C in a graph G is a subset of E with the property that after removal from G of all edges in C, G becomes disconnected. For example, with every partition {X,Y} of V, a cut [X:Y] can be associated : [X:Y] = {e ∈ E | e = ij, i ∈ X, j ∈ Y}.

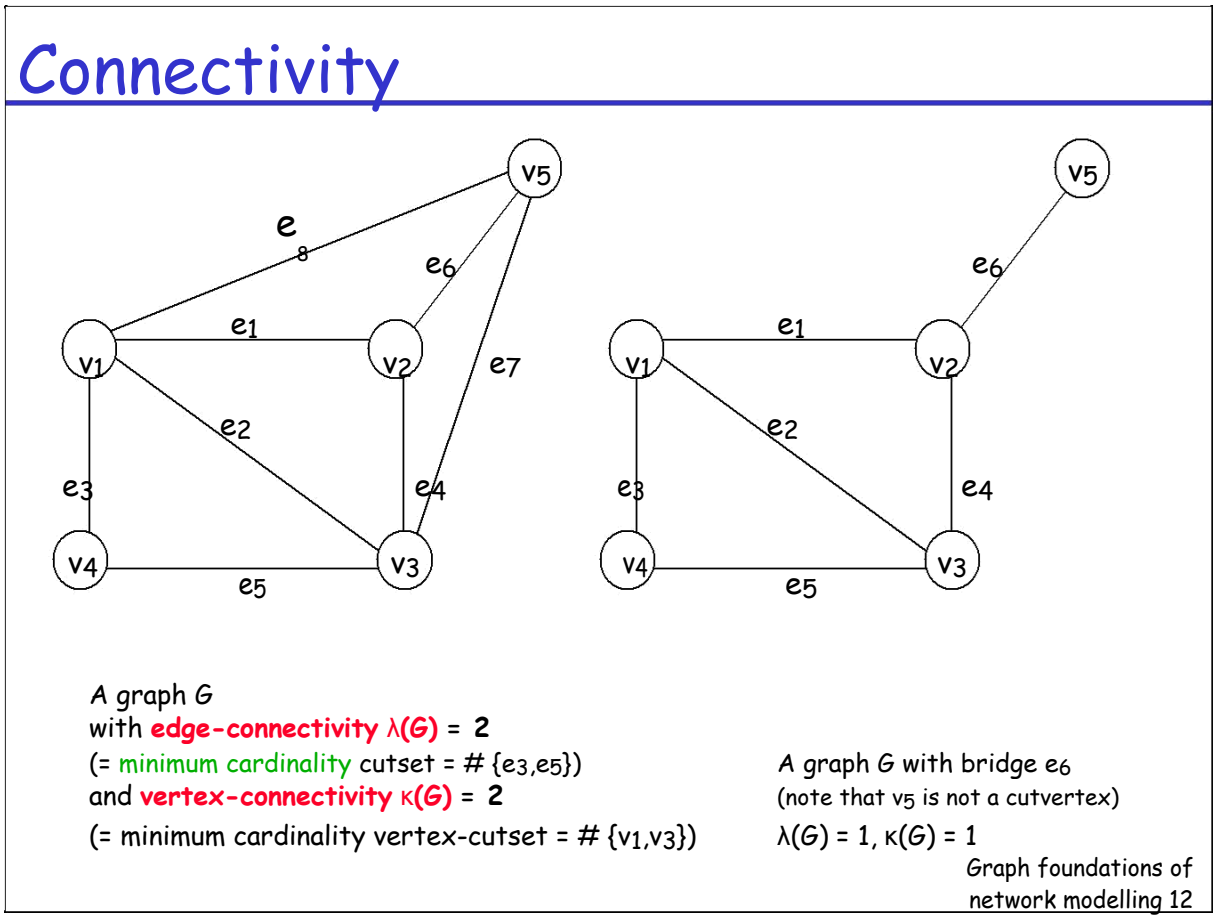
A *cutset* (edge-cutset) C in a graph G is a cut in G that is minimal with respect to this property. This means that a cutset does not contain a strict subset which is in itself a cut. A cut [X:Y] associated with a partition {X,Y}of V(G) is a cutset if and only if the subgraphs of G induced by X and Y are both connected. (Why?)

Analogous definitions hold for *vertex-cuts* and *vertex-cutsets* in a graph :

A *vertex-cut* C in a graph G is a subset of V with the property that after removal from G of every vertex v in C and every edge e incident from this vertex (e ∈ I(v)), G becomes disconnected or becomes the trivial graph.

A *vertex-cutset* C in a graph G is a vertex-cut in G that is minimal with respect to this property. This means that a vertex-cutset does not contain a strict subset which is in itself a vertex-cut.

Task : Try to find out in which cases the sentence “or becomes the trivial graph” in the definition of a vertex-cut will make a difference.



The *edge-connectivity λ (G)* of a graph G is the minimum cardinality of an edge-cutset of G.

A graph G has λ(G) = 0 if and only if G is disconnected or G is trivial. If λ(G) = 1, then G contains a so-called *bridge*. A bridge is an edge whose removal disconnects the graph.

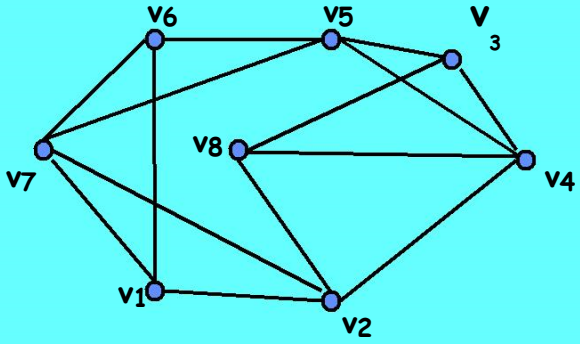
The *vertex -connectivity κ (G)* of a graph G is the minimum cardinality of a vertex-cutset of G. Stated otherwise, it is the minimum number of vertices whose deletion from G produces a disconnected graph or the trivial graph.

Based on these definitions, one can prove that the vertex-connectivity is always less or equal to the edge-connectivity : κ(G) ≤ λ(G).

Question : Would these properties still hold if "or the trivial graph" were not included in the definition of κ(G) ? Why (not) ?

Connectivity

EXERCISE



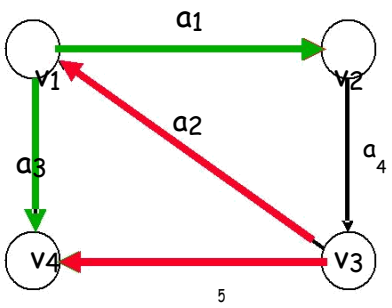
edge-connectivity $\lambda(G)$ = ?
vertex-connectivity $\kappa(G)$ = ?

Outline

0. Introduction
1. **Definition and notations**
   - 1.1 **Graph**
   - 1.2 **Digraph**
2. Minimum spanning tree problem
3. Shortest path problem
4. Maximum flow problem
5. Minimum cost flow problem
6. Extension: multi-commodity flow problems

Digraph

Directed Graph



- vertex set V
- arc set A

D = (V,A)

- digraph order = #V = 4
- digraph size = #A = 5

- from - incidence : I (v_1) = {a_1, a_3}
- to - incidence : I'(v_4) = {a_3, a_5}
- from - adjacency : A(v_3) = {v_1,v_4}
- to - adjacency : A'(v_2) = {v_1}
- out - degree : δ(v_3) = 2
- in - degree : δ'(v_1) = 1

Graph foundations of network modelling 15

A *digraph D = (V,A)* (or directed graph) consists of a vertex set V and an arc set A. Every arc a = (u,v) is an ordered pair of vertices u,v(*). The order of a digraph is the number of vertices and the size of a digraph is the number of arcs. A digraph D with order p and size q is called a (p,q)-digraph.

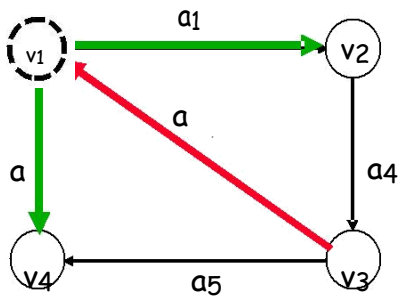
If a = (u,v) is an arc of the digraph D, then a is incident from u and incident to v, u is the tail of a, and v is its head. The from-incidence (to-incidence) of a vertex is the set of arcs incident from (to) it. The out-degree (in-degree) of a vertex is the number of arcs incident from (to) it.

If a = (u,v) is an arc of the digraph D, then u is adjacent to v and v is adjacent from u. The from-adjacency (to-adjacency) of a vertex is the set of vertices adjacent from (to) it.

(*) Remark : Again (as for undirected graphs), we will assume in this chapter that a directed graph contains no parallel arcs(**) and no loop arcs. Anti-parallel arcs (e.g. an arc (u,v) and an arc (v,u)) are allowed.

(**) The only exception to this is subsection 5.3, where parallel arcs will be needed.

Digraph



incidence matrix of D

arcs (q)

$$I_D = \begin{pmatrix} 1 & -1 & 1 & 0 & 0 \\ -1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & -1 & 0 \\ 0 & 0 & -1 & 0 & 1 \end{pmatrix}$$

vertices (p)

- $I_D(v, a) = 1$ if a is incident from v
- $I_D(v, a) = -1$ if a is incident to v
- $I_D(v, a) = 0$ otherwise

incidence lists of D

$L_D = (\,(\{1,3\}, \{2\}),\ (\{4\}, \{1\}),\ (\{2,5\}, \{4\}),(\varnothing, \{3,5\}))$

Graph foundations of network modelling 16

A digraph can be represented in many ways. Two important representations are:

1. incidence matrix

The incidence matrix I_D of a (p,q)-digraph D is defined as follows. I_D is a (p,q)-matrix with $I_D(v,a)=1$ if a is incident from v, $I_D(v,a)=-1$ if a is incident to v and $I_D(v,a)=0$ otherwise. This representation is often used for building linear programming models of network problems (cf. chapter 3).

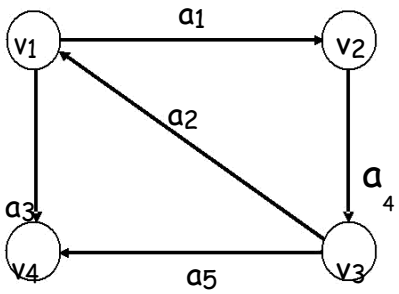
2. incidence lists

For each vertex v of D, the from-incidence and the to-incidence are given. This representation is often used for the software implementation of classical graph algorithms, such as the algorithm of Dijkstra (cf. section 3).

Paths and Cycles

walk $w = (v_0, v_1, \ldots, v_{n-1}, v_n)$ such that for $i = 0, 1, \ldots, n-1$, $(v_i, v_{i+1}) \in A$

semiwalk $= (v_0, v_1, \ldots, v_{n-1}, v_n)$ such that for $i = 0, 1, \ldots, n-1$, $(v_i, v_{i+1}) \in A \vee (v_{i+1}, v_i) \in A$



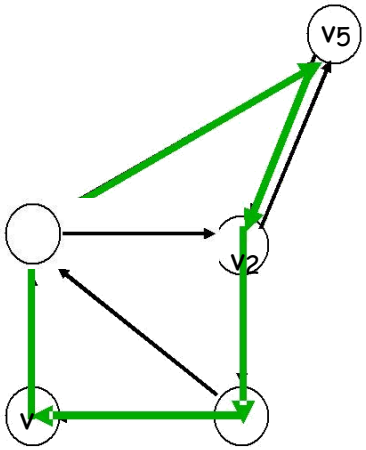
(v_1, v_2, v_3) is a path with length 2

(v_1, v_2, v_3, v_1) is a cycle with length 3

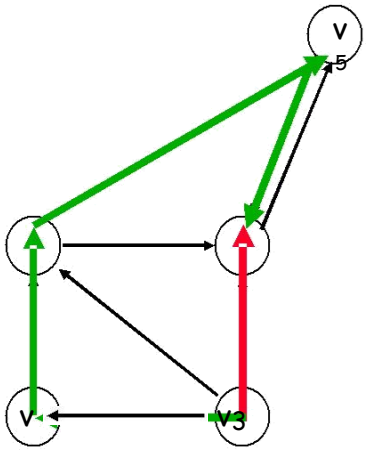
(v_1, v_4, v_3) is a semipath with length 2 (v_1, v_4, v_3, v_1) is a semicycle with length 3

The definition of *walks*, *paths* and *cycles* in a digraph is analogous to the definition of walks paths and cycles in a graph.
In addition, we can define semiwalks, semipaths and semicycles. A *semiwalk* in a digraph D=(V,A) is a sequence $w = (v_0, v_1, \ldots, v_{n-1}, v_n)$ such that for $i = 0, 1, \ldots n-1$, $(v_i, v_{i+1}) \in A$ or $(v_{i+1}, v_i) \in A$. *Semipaths* and *semicycles* are defined starting from semiwalks just like paths and cycles are defined starting from walks.

Strong and weak connectedness



this digraph is strongly connected
(u,v)- AND (v,u)-path
⇒ (directed) cycle through u and v
e.g.: (v_4, v_1, v_5, v_2, v_3, v_4)



this digraph is weakly connected
(u,v)- AND/OR (v,u)-path
there are no paths terminating in v_3 (no cycle through u and v)

Graph foundations of network modelling 18

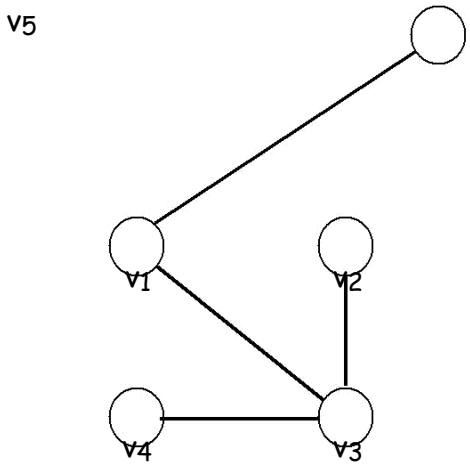
A digraph is *strongly connected* if for every pair of vertices u,v there exist both a (u,v)-path and a (v,u)-path.
A digraph is *weakly connected* if for every pair of vertices u,v there exists a (u,v)-path and/or (v,u)-path.

Question: If for every pair of vertices u and v of a digraph there exists a (u,v)-semipath, does that imply that the digraph is weakly connected ?

Outline

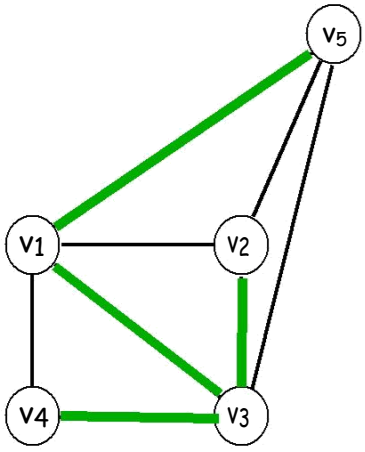
0. Introduction
1. Definition and notations
2. **Minimum spanning tree problem**
3. Shortest path problem
4. Maximum flow problem
5. Minimum cost flow problem
6. Extension: multi-commodity flow problems

Tree and spanning tree



a tree T=(V,F)
- T is connected
- T does not contain cycles

⇒ T contains #V-1 edges



a spanning tree T=(V,F) of G
- T is a tree
- T has the same vertex set as G

Graph foundations of network modelling 20

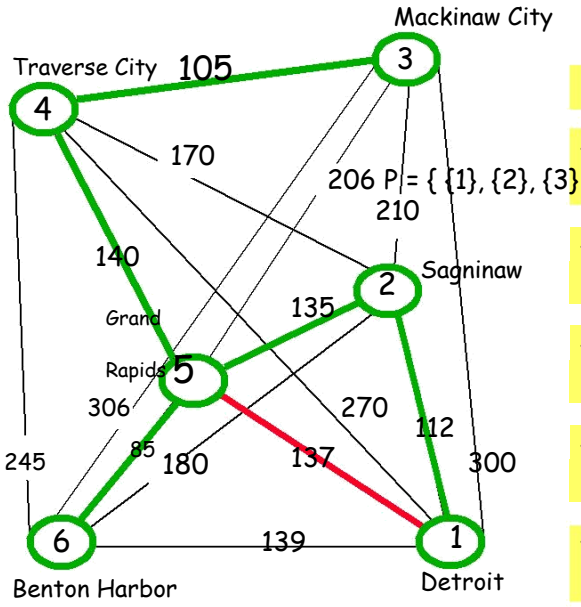
A *tree* is a connected (undirected) graph which contains no cycles. One can prove that for each tree T=(V,F): #F = #V-1.

A tree T=(V,F) has an interesting property: for every two vertices s, t of V, T contains exactly one (s,t)-path.

A subgraph T = (V,F) of a connected graph G=(V,E) which has the same vertex set as G and is in itself a tree is called a *spanning tree* of G.

MST: algorithm of Kruskal

weighted graph
c(e) = weight of edge e
weight of weighted graph = sum of weights of all edges
minimum spanning tree = spanning tree with minimum weight (in a weighted graph)



P = { {1}, {2}, {3}, {4}, {5}, {6} }

{insertion of edge 56 with weight 85 into *F*}
P = { {1}, {2}, {3}, {4}, {5,6} }

{insertion of edge 34 with weight 105 into *F*}
P = { {1}, {2}, {3,4}, {5,6} }

{insertion of edge 12 with weight 112 into *F*}
P = { {1,2}, {3,4}, {5,6} }

{insertion of edge 25 with weight 135 into *F*}
P = { {1,2,5,6}, {3,4} }

{insertion of edge 45 with weight 140 into *F*}
P = { {1,2,3,4,5,6} }

We will call a graph G=(V,E) *weighted* if there exists a mapping c(·) from the edge set E to the real numbers. We call c(e) the weight of edge e.

The weight of a subgraph H of a weighted graph G is the sum of the weights of the edges of the subgraph. A *minimum spanning tree* in a weighted graph G is a spanning tree with minimum weight.

The *algorithm of Kruskal* is one of the simplest graph algorithms. It allows to determine a minimum spanning tree of a weighted graph. During the execution of the algorithm, a partial solution F is maintained. At each iteration, we add to F the lowest weight edge which does not lead to a cycle in T=(V,F). The algorithm terminates when T=(V,F) is a tree spanning all vertices of G.

The correctness of the algorithm is mainly based on the following observation:

• suppose W1 and W2 are disjunct subsets of V and both G[W1] and G[W2] are connected;

• suppose we have a minimum spanning tree T1 on G[W1] and a minimum spanning tree T2 on G[W2];

• suppose G[W1 ∪ W2] is connected;

• then a minimum spanning tree T on G[W1 ∪ W2] can be constructed by taking the edges of both T1 and T2, and adding the edge of [W1:W2] which has the lowest weight.

Minimum spanning tree: applications

Characteristics MST:

- network infrastructure connecting all 'entities'
- minimal cost ⇗ price is key
- no fault-tolerance ⇗ not important ?
- unique path ⇗ unequivocal routing, cycles avoided

Applications area:

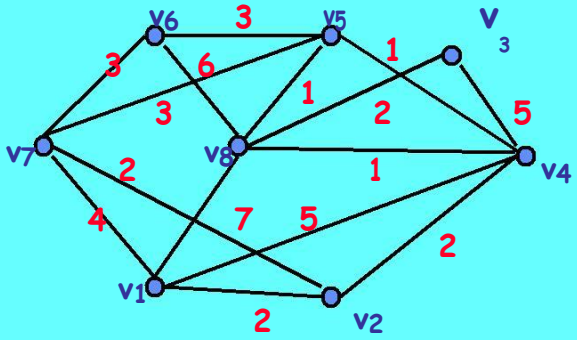
- communication networks:
  - minimal topology
  - virtual tree network
- other networking situations:
  - electricity network
  - sewer system network
  - ...

Graph foundations of network modelling 22

The main characteristics of a minimum spanning tree are summarised above. From these characteristics, it is clear what the main advantages and disadvantages of MST-alike solutions are and where they will be applied in realistic situations. For instance, it could be used in communication network applications to find the topology with minimal installation cost to connect all network nodes. Also within an existing network, the MST algorithm could be used to set up a virtual private tree network between all network nodes. Of course, also in other network applications, the MST algorithm can be useful, for instance to design minimal cost electricity or sewer system networks.

Minimum spanning tree

EXERCISE



Outline

0. Introduction
1. Definition and notations
2. Minimum spanning tree problem
3. **Shortest path problem**
   - **3.1 In unweighted (di)graphs**
   - **3.2 In weighted (di)graphs (positive weights)**
   - **3.3 In weighted (di)graphs (general weights)**
   - **3.4 All-pairs shortest path problem**
4. Maximum flow problem
5. Minimum cost flow problem
6. Extension: multi-commodity flow problems

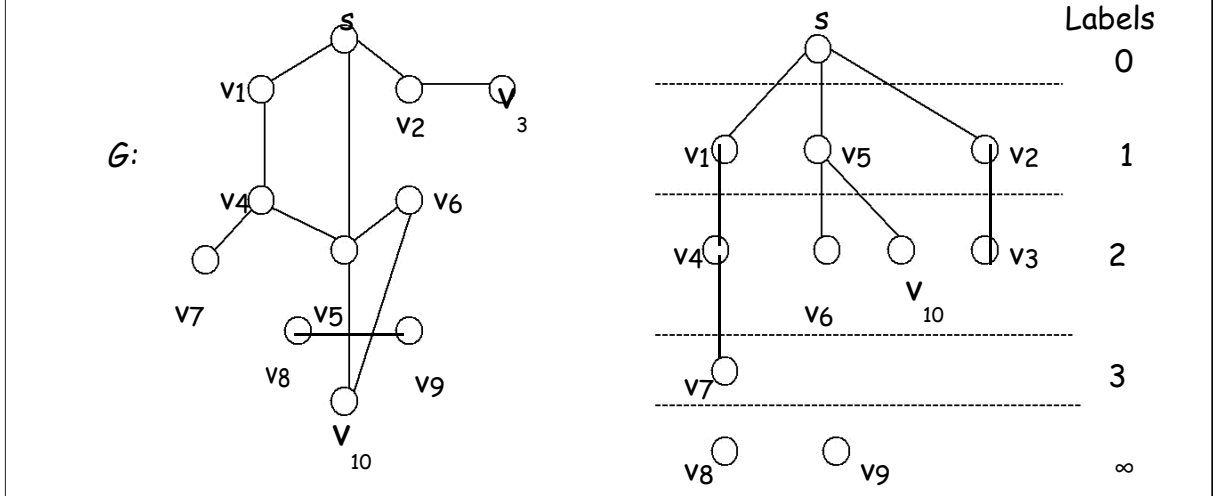
Distance in unweighted graph

The length of a path in an unweighted graph is
equal to the number of edges in the path (='hopcount').

The distance d(u,v) between two vertices u,v in a graph G = (V,E)
is equal to the length of a shortest u,v-path in G,
if such a path exists (if not: d(u,v) = ∞).

In an undirected graph without weights, the *length of a path* is equal to the number of edges in the path (so-called hopcount).
The *distance d(u,v)* between two vertices u,v in a graph G = (V,E) is equal to the length of a shortest u,v-path in G, if such a path exists (if no path exists between u and v, then d(u,v) = ∞).

Example : the Moore Algorithm



s	v1	v2	v3	v4	v5	v6	v7	v8	v9	v10
0	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞
0	1	1	∞	∞	1	∞	∞	∞	∞	∞
0	1	1	2	2	1	2	∞	∞	∞	2
0	1	1	2	2	1	2	3	∞	∞	2

Graph foundations of network modelling 26

Given an unweighted graph G = (V,E) and two vertices s,t ∈ V, the *Moore algorithm* can be used to calculate the distance between s and t. Essentially the Moore algorithm performs a breadth-first search of G. Initially s is labeled 0, the distance from s to itself. All other vertices are labeled ∞ (since no path from s to the other vertices is found yet). We then label every vertex that is adjacent to s with label 1. To every vertex labeled ∞ that is adjacent to a vertex labeled 1, we give label 2. We continue like this either until t has a finite label or until all vertices with finite label are adjacent only to vertices that already have a finite label (in which case s and t are not connected in G). When the algorithm terminates, every vertex v where d(s,v) ≤ d(s,t) carries a label corresponding to the correct distance d(s,v) .

Note: This algorithm is applicable on graphs as well as on digraphs.

Outline

0. Introduction
1. Definition and notations
2. Minimum spanning tree problem
3. **Shortest path problem**
   - **3.1 In unweighted (di)graphs**
   - **3.2 In weighted (di)graphs (positive weights)**
   - **3.3 In weighted (di)graphs (general weights)**
   - **3.4 All-pairs shortest path problem**
4. Maximum flow problem
5. Minimum cost flow problem
6. Extension: multi-commodity flow problems

Distance in weighted graph

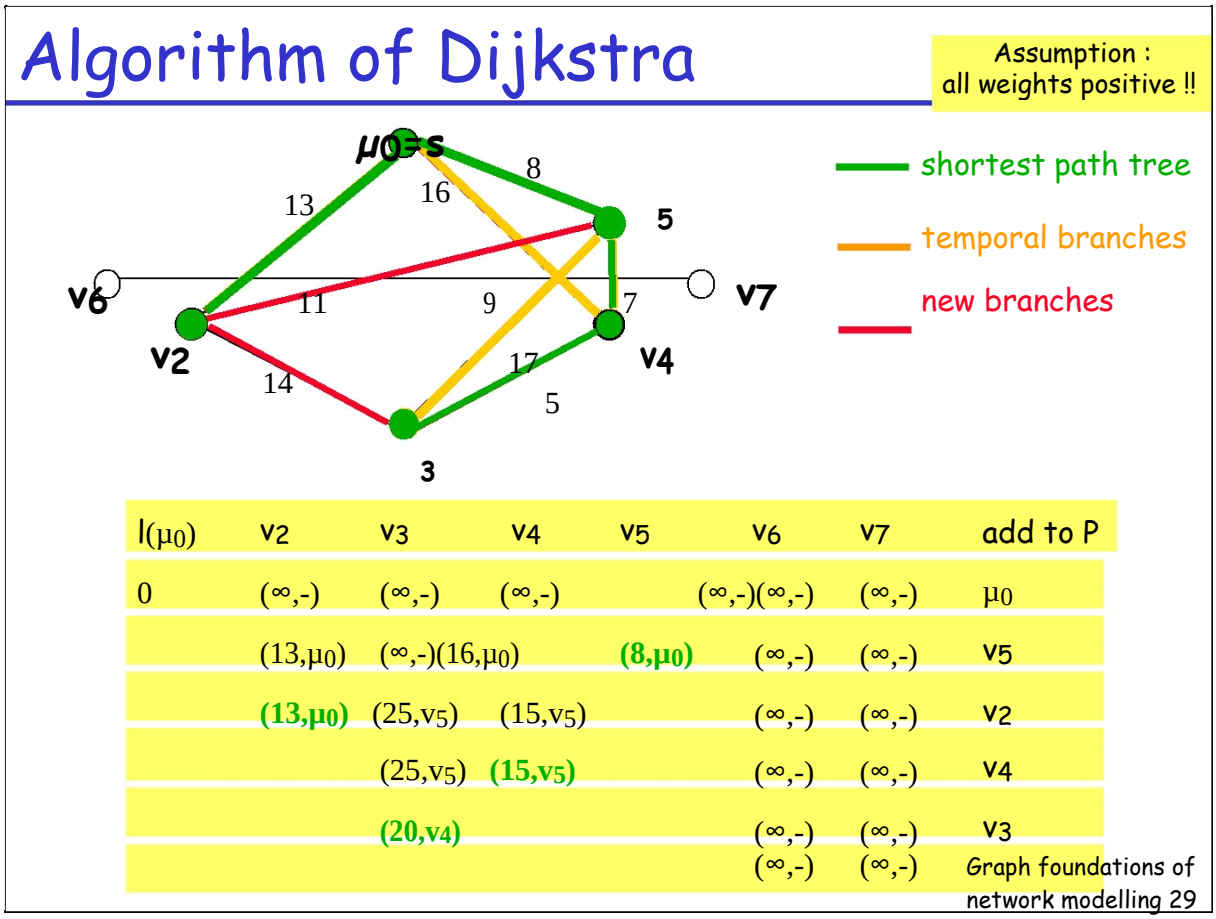
Weighted Graph : c : E ⇨ R : map every edge to a real number

Length of a path : sum of length of its edges

Distance between two vertices s,t : length of shortest s,t path

A graph G, together with a length (or weight, or cost) function c:E⇨R mapping every edge of G to a real number is called a weighted graph. The *length of a path* in a weighted graph is equal to the sum of the lengths of its edges.

The *distance* between two vertices u,v in a weighted (undirected) graph is equal to the length of the shortest u,v-path.

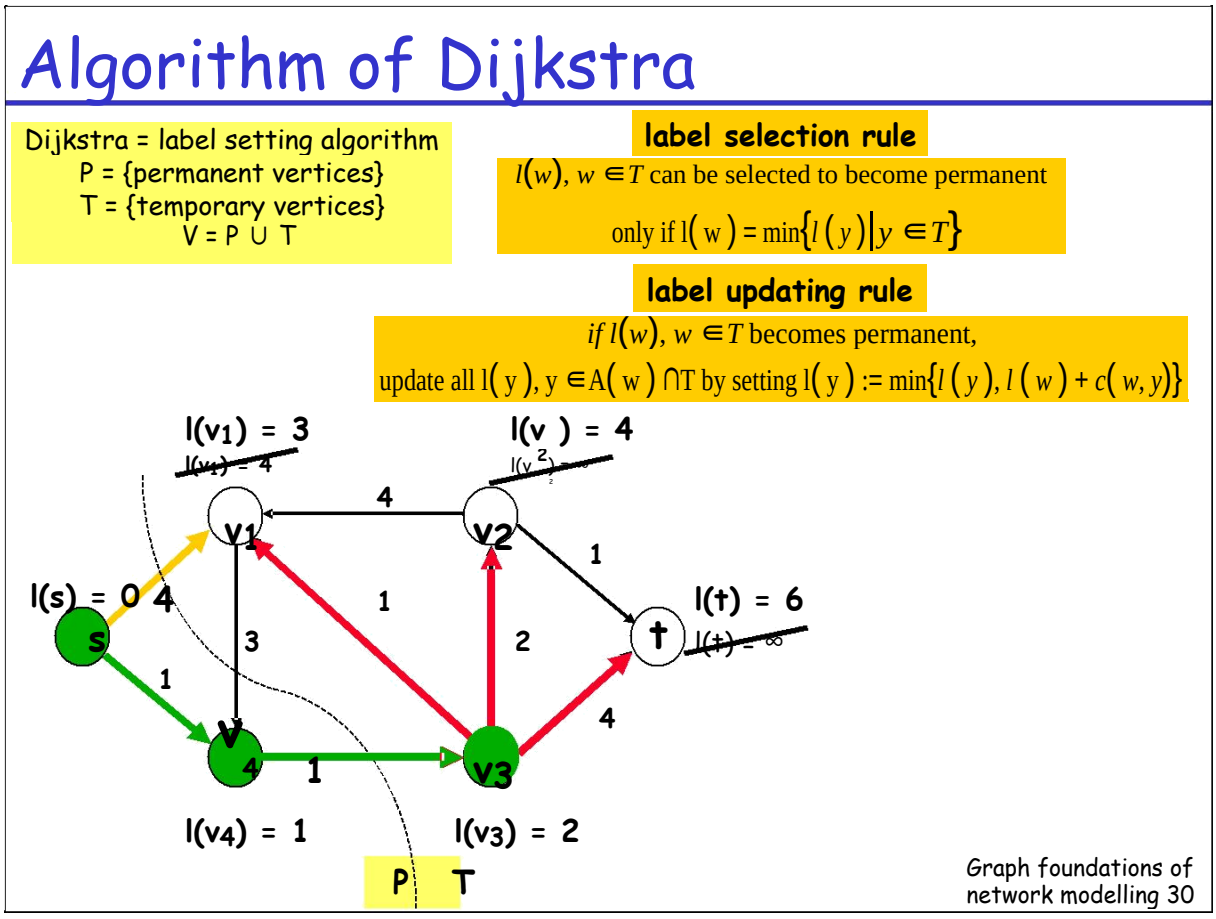


Given a weighted graph with all weights positive and two vertices s,t, the algorithm of Dijkstra can be used to calculate the distance between s and t.

Important note: The algorithm of Dijkstra is only applicable if all weights are positive: $c(e) \geq 0$!!!

The figure above shows how the algorithm of Dijkstra proceeds to calculate the distances of all vertices of a graph from a source vertex $s = \mu_0$.

Note: This algorithm is applicable on graphs as well as on digraphs.



The algorithm of Dijkstra is a *label setting algorithm* and works as follows. During the execution of the algorithm a partition P ∪ T = V of the vertex set V is maintained. P is the set of vertices whose label is permanent, T is the set of vertices whose label is temporary.

Initially, all labels are set to ∞, except the label of s, which is set to 0, and all labels are temporary. At every iteration, one of the temporary labels becomes permanent (is set), and adjacent temporary labels are updated. The corresponding vertex moves from T to P. The following *label selection rule* is used to determine the label which becomes permanent

l(w), w ∈ T, can be selected to become
permanent only if l(w) = min { l(y) y ∈ T }

(R1)

where ties can be broken by making a random choice. Temporary labels of vertices adjacent to the vertex whose label becomes permanent are always updated using the following *label updating rule*

if l(w), w ∈ T, becomes permanent,
update all l(y), y ∈ A(w) ∩ T by setting l(y) := min { l(y), l(w) + c(w,y) }

(R2)

One can prove that by using this label selection rule and this label updating rule, the following *property* holds :

∀ x ∈ P : l(x) = d(s,x)

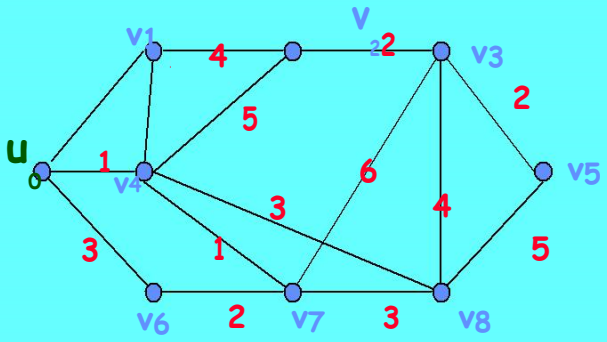
(P)

This is a very useful property: as soon as t is permanently labeled, we can stop because we know that l(t) = d(s,t) as soon as t ∈ P.

The figure above shows a snapshot during the execution of the algorithm of Dijkstra.

Dijkstra

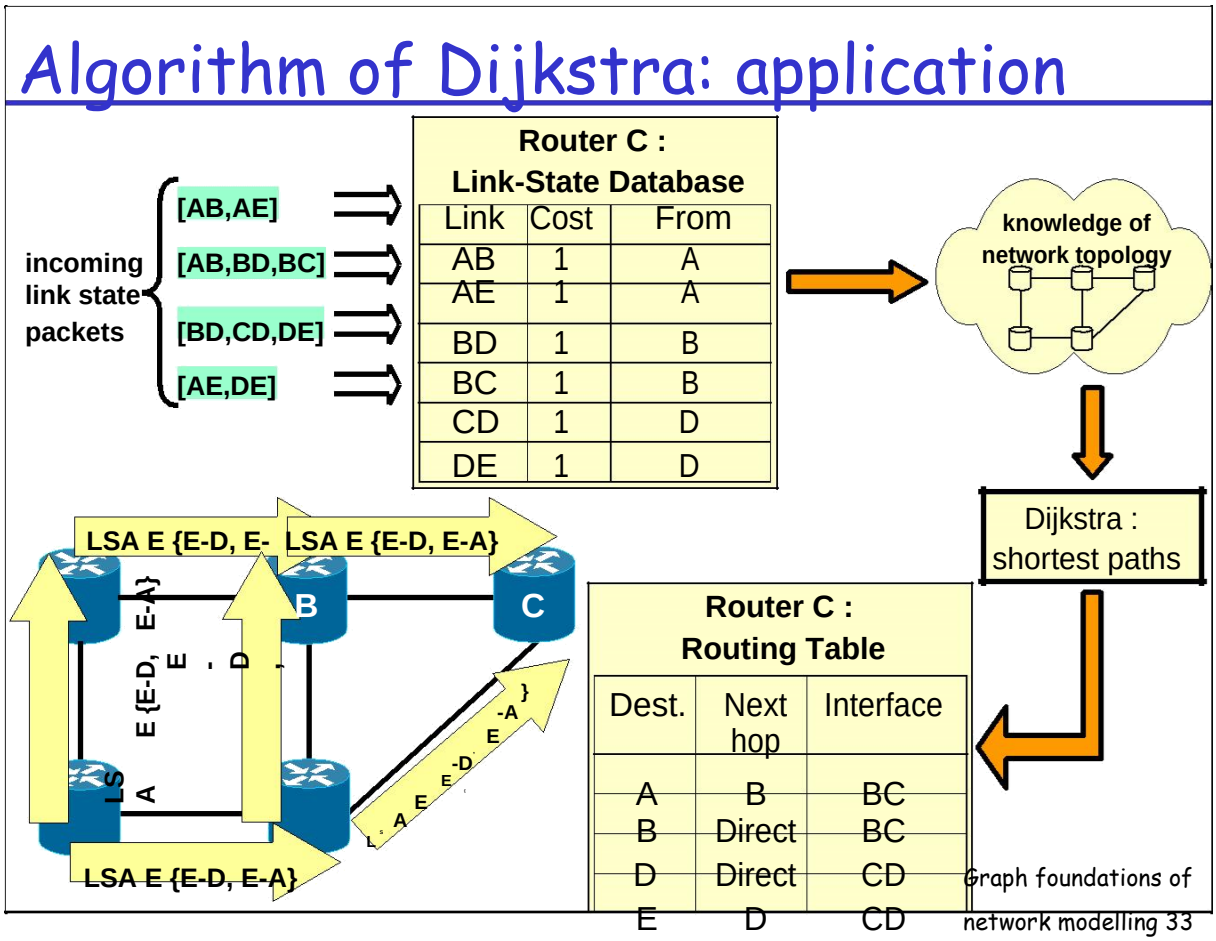
EXERCISE



d(u0,v5) ?
1. op zicht
2. m.b.v. Dijkstra

Dijkstra

EXERCISE



Dijkstra's algorithm is one of the most frequently applied graph algorithms, in many fields. One of the main applications of the algorithm is found in the Internet, where the Open Shortest Path First (OSPF) protocol is based on it.

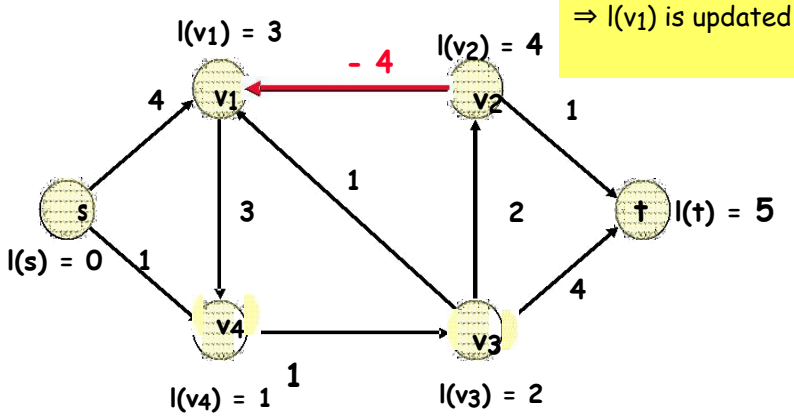
Through Link State Advertisements, every router (e.g. router C in the slide above) can collect the needed information to know the network topology (within its OSPF domain). To create the routing table in this router, indicating to which outgoing interface an incoming IP packet must be forwarded to depending on the IP destination address, the shortest paths must be calculated from router C to all other routers (destinations) in the OSPF domain. This essentially comes down to calculating the shortest path tree with router C as the source. As all link weights are positive, the algorithm of Dijkstra is very suited for this job.

Outline

0. Introduction
1. Definition and notations
2. Minimum spanning tree problem
3. **Shortest path problem**
   - **3.1 In unweighted (di)graphs**
   - **3.2 In weighted (di)graphs (positive weights)**
   - **3.3 In weighted (di)graphs (general weights)**
   - **3.4 All-pairs shortest path problem**
4. Maximum flow problem
5. Minimum cost flow problem
6. Extension: multi-commodity flow problems

The algorithm of Ford, Bellman and Moore

Result with Dijkstra:



$l(v_1) \le l(v_2) + c(v_2v_1)$ is violated
$\Rightarrow l(v_1)$ is updated and set to $l(v_1) = l(v_2) + c(v_2v_1) = 0$

Principle : label correcting algorithm

Let $l(i)$, $i \in V$ be a set of labels. These labels represent the shortest path distances from the source vertex s if they satisfy the following conditions:

(i) $l(s) = 0$;
(ii) $l(i)$ is the length of some (s,i)-path;
(iii) $l(j) \le l(i) + c(ij)$, $\forall (i,j) \in A$

Graph foundations of network modelling 35

If we want to calculate a shortest path in a (di)graph containing negative weights, the algorithm of Dijkstra can not be used anymore. In this case, the algorithm of Ford, Bellman and Moore can be applied.

The algorithm of Ford-Bellman-Moore is a *label correcting algorithm*. Label correcting algorithms, as the name implies, maintain tentative distance labels for vertices and correct the labels at every iteration. Unlike label setting algorithms, these algorithms maintain all distance labels as temporary until the end, when they all become permanent simultaneously. The label correcting algorithms are conceptually more general than the label setting algorithms and are applicable to more general situations (e.g. to (di)graphs containing negative length arcs). To produce shortest paths, these algorithms typically require that the network does not contain any negative (un)directed cycle (why?).

Label correcting algorithms can be viewed as a procedure for solving the following recursive equations (*Bellman's equations*):

$$l(s) = 0$$
$$l(j) = \min \{ l(i) + c(ij) | (i,j) \in A \}, \quad \forall j \in V \setminus \{s\}$$

These conditions represent necessary and sufficient conditions for optimality of the shortest path problem (why ?).

The following *alternate version of these conditions* is more suitable for explaining the Ford-Bellman-Moore algorithm.

Let $l(i)$, $i \in V$ be a set of labels. These labels represent the shortest path distances from the source vertex s if they satisfy the following conditions:

(i) $l(s) = 0$;
(ii) $l(i)$ is the length of some (s,i)-path;
(iii) $l(j) \le l(i) + c(ij)$, $\forall (i,j) \in A$

The algorithm of Ford, Bellman and Moore

(1) l(s) := 0 ; ∀v∈V\{s} do l(v) := ∞ end;

l(s)=0, l(v1)=l(v2)=…=l(t)=∞

(2) insert s into List;

List={s}

(3) while List is not empty do

(4) select the first element v from List and remove it from List;

v=v1,s,ListList=={ {v4} }

(5) ∀w∈A(v) do

A(v)={v4}, A(v)={v1,v4}

(6) if l(w) > l(v) + c(vw)

1)=4, l(v4)=1

l(v4)=1

(7) l(w) := l(v) + c(vw);

List = {v1,v4 }

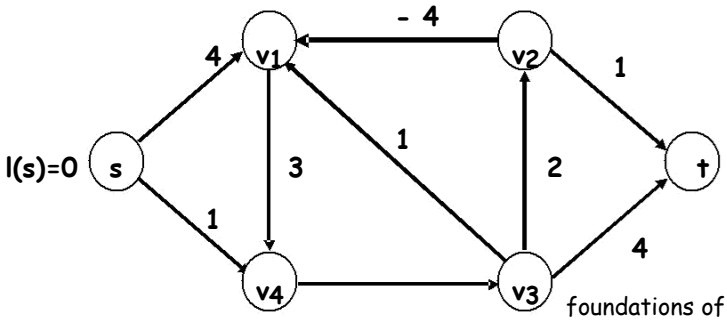
(8) if w is not yet an element of List, insert w at the back of List;

List = {v4 }

(9) end;

(10) end;

A(v)=from-adjacency in digraph

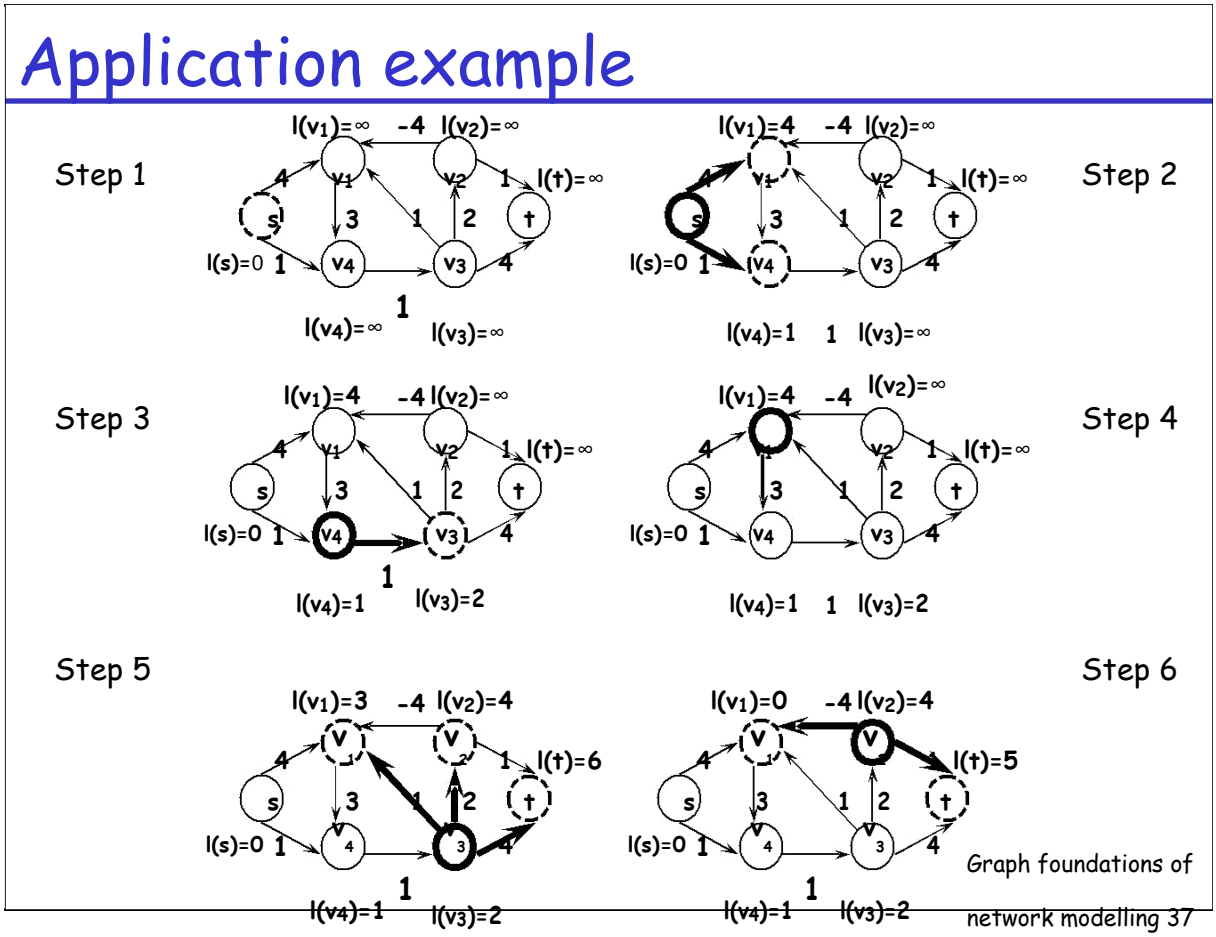


Extension : negative cycles ?

In the slide above, a formal description of the algorithm of Ford, Bellman and Moore is presented.

At any point in the execution of the algorithm, the label l(i) will either be ∞, indicating that we still have to discover a path from s to i, or it is the length of some path from s to i. The algorithm is based on the simple observation that whenever l(j) > l(i) + c(ij), the current path from the source s to vertex i, of length l(i), together with the arc(i,j) is a shorter path to j than the current path of length l(j).

Note : A variant of the algorithm of Ford, Bellman and Moore can be used to detect negative cycles in a graph. How would you do that ?

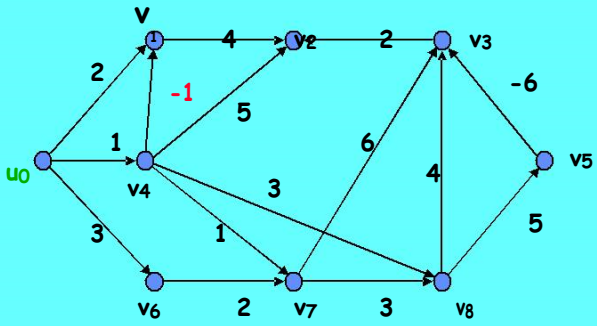


The operation of the algorithm of Ford, Bellman and Moore is illustrated above on an example. Vertices removed from List are shown bold, vertices entering List appear dashed bold. There are still two more steps to go after step 6 (v_1 and t are still in List) but no more label corrections will occur so these steps are not shown.

We remark that labels can enter List more than once: v_1 enters the List both when s is evaluated (step 2) and when v_3 is evaluated (step 5).

Ford-Bellman-Moore

EXERCISE



Outline

0. Introduction
1. Definition and notations
2. Minimum spanning tree problem
3. **Shortest path problem**
   - **3.1 In unweighted (di)graphs**
   - **3.2 In weighted (di)graphs (positive weights)**
   - **3.3 In weighted (di)graphs (general weights)**
   - **3.4 All-pairs shortest path problem**
4. Maximum flow problem
5. Minimum cost flow problem
6. Extension: multi-commodity flow problems

All-pairs shortest path problem

Until now:

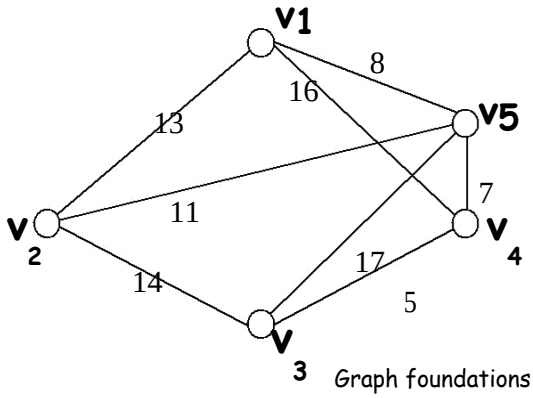
- Shortest path from s to t
- Side effect: shortest path tree from s to all other nodes

Current subsection:

Shortest path between ALL PAIRS OF NODES

Possible approaches:

- p times Dijkstra
- Dedicated algorithm: Floyd



Graph foundations of network modelling 40

In the previous subsections 3.1 – 3.3, we did only consider the calculation of shortest paths from one particular vertex (the so-called source vertex s) towards all the other vertices in a graph. If we would like to know the shortest path between every pair of vertices in a graph, we could of course use the above mentioned algorithms repeatedly. However, dedicated algorithms can be devised for this problem, that calculate all shortest path in a graph simultaneously. For instance the algorithm of Floyd, described in the next slides, is such a dedicated all-pairs shortest path algorithm.

Algorithm of Floyd

Assumption : all weights positive !!

Step 0: direct paths distances

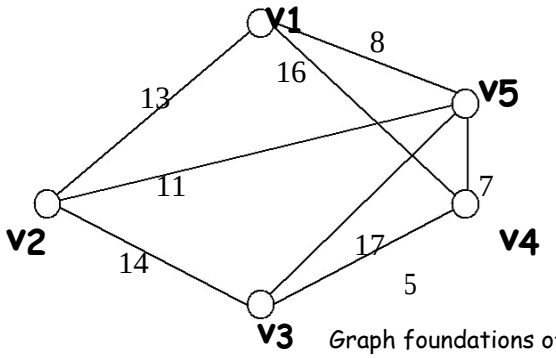
	v_1	v_2	v_3	v_4	v_5
v_1	0		13	∞	16	8
v_2	13	0	14	∞		11
v_3	∞	14	0		5	17
v_4	16	∞	5		0	7
v_5	8	11	17	7		0

Step 1: x – v_1 – y better ?

	v_1	v_2	v_3	v_4	v_5
v_1	0		13	∞	16	8
v_2	13	0	14	29		11
v_3	∞	14	0		5	17
v_4	16	29	5		0	7
v_5	8		11	17	7	0

Step 2: x – v_2 – y better ?

	v_1	v_2	v_3	v_4	v_5
v_1	0	13	27	16	8
v_2	13	0	14	29	11
v_3	27	14	0	5	17
v_4	16	29	5	0	7
v_5	8	11	17	7	0



Graph foundations of network modelling 41

As it was the case for the Dijkstra algorithm as well, Floyd's algorithm assumes that all weights of the graph are positive.

In the initialisation phase (step 0 in the slide above), only direct paths are taken into account. Basically, one can distinguish three different situations:

- from each node to itself: distance = 0
- between two nodes that are not directly connected by an edge: temporal distance = ∞
- between two nodes that are directly connected by an edge: temporal distance = weight of that edge

This is illustrated in the slide above for the shown weighted graph.

The iteration phase of Floyd's algorithm (steps 1 to 5 in the slide above and the next slide) are considering every node v of the graph one by one and checking whether considering this node v as an intermediate node between nodes x and y could yield a shorter path from x to y. If so, the matrix of temporal distances is updated accordingly.

For instance, in step 1, it is checked whether the temporal distance v_2-v_1 plus the temporal distance v_1-v_3 is not smaller than the temporal distance v_2-v_3.

Remark: It is important that all calculations in the iteration phase are done on the matrix, not on the graph picture. Try to repeat the same procedure by looking at the graph picture to understand this.

Algorithm of Floyd

Step 3: x – v_3 – y better ?

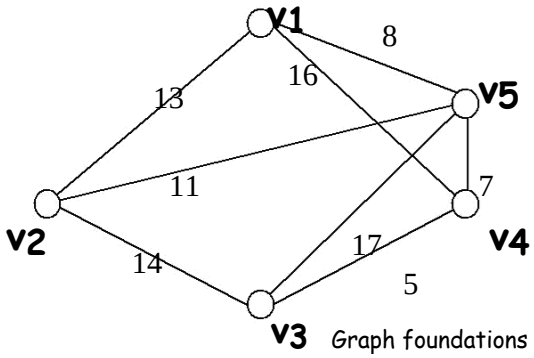
	v_1	v_2	v_3	v_4	v_5
v_1	0	13	27	16	8
v_2	13	0	14	19	11
v_3	27	14	0	5	17
v_4	16	19	5	0	7
v_5	8	11	17	7	0

Step 4: x – v_4 – y better ?

	v_1	v_2	v_3	v_4	v_5
v_1	0	13	21	16	8
v_2	13	0	14	19	11
v_3	21	14	0	5	12
v_4	16	19	5	0	7
v_5	8	11	12	7	0

Step 5: x – v_5 – y better ?

	v_1	v_2	v_3	v_4	v_5
v_1	0	13	20	15	8
v_2	13	0	14	18	11
v_3	20	14	0	5	12
v_4	15	18	5	0	7
v_5	8	11	12	7	0



Graph foundations of network modelling 42

As soon as all nodes have been considered as potential intermediate nodes in the iteration phase, the algorithm is terminated. One can prove that the elements in the final matrix represent the correct distances between every two nodes in the graph.

Remarks:

•It is important that all calculations in the iteration phase are done on the matrix, not on the graph picture. Try to repeat the same procedure by looking at the graph picture to understand this.

•The algorithm is applicable to both undirected and directed graphs. Try to apply the algorithm to a simple directed graph to understand this.

Question: What happens if non-positive weights appear in the graph? Is the result by applying Floyd's algorithm still correct in this case?

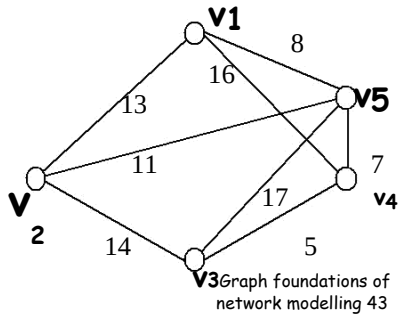
Algorithm of Floyd

(1) $D^{(0)} = A$;
(2) for *step* := 1 to p do
(3) for v_i := v_1 to v_p do
(4) for v_j := v_1 to v_p do
(5) $d_{i,j}^{(step)} := \min \{ d_{i,j}^{(step-1)} , d_{i,vstep}^{(step-1)} + d_{vstep,j}^{(step-1)} \}$
(6) end;
(7) end;
(8) end.

Implementation effort: small

Time complexity: $O(p^3)$

↔ p times Dijkstra: p x $O(p^2)$



In the slide above, a formal description of Floyd's algorithm is presented. At a certain step, counted by the variable '*step*' above, the node v_{step} is considered as the intermediate node between all pairs of nodes i and j.

It is clear from this formal description the algorithm of Floyd allows a very easy computer implementation. The time complexity is easy to deduct, as we have three nested for-loops enumerating all network nodes: $O(p^3)$. With respect to time complexity, this is equal to the performance of using Dijkstra's algorithm for every source node individually.

Outline

0. Introduction
1. Definition and notations
2. Minimum spanning tree problem
3. Shortest path problem
4. **Maximum flow problem**
   - **4.1 Network flow**
   - **4.2 Problem definition**
   - **4.3 Residual network concept**
   - **4.4 Flow augmenting path algorithm**
   - **4.5 Max-flow min-cut theorem**
   - **4.6 Application: calculation of connectivity**
5. Minimum cost flow problem
6. Extension: multi-commodity flow problems

Network flow models

abstract modelling of a communication network

- transfer of information ⇨ **flow**
- topology ⇨ directed graph (digraph) $D = (V,A)$
- every arc $a \in A$ has associated numerical values
  - **cost** $c(a)$ ⇨ weighted digraph
  - **capacity** $u(a)$ ⇨ capacitated digraph

network N = weighted, capacitated digraph (V,A)

Graph foundations of network modelling 45

The topology of a communications network is usually modeled mathematically by a *digraph*. This digraph gives enough information about the interconnectedness of network elements: every network node corresponds to a vertex in this digraph D, and every link between two network elements corresponds to two arcs (one in each direction) in the digraph. The directed nature of a digraph will give us the opportunity to model the direction of the transferred information (e.g. from a to b or from b to a).

In addition, we have to associate numerical values to each arc:

1. It is obvious that we have to model some costs for using network facilities. This means that every arc a in the digraph has an associated *cost c(a)* (e.g. this weight may be the length of the link, or a cost coefficient for using one unit of flow on the transmission facilities of the physical link associated to the arc). See also previous section. The cost for transferring a flow f(a) over arc a is then defined as c(a).f(a).
2. We also have to model the fixed capacities of the facilities of a communication network (e.g. a cross-connect in an SDH-network has a finite numbers of ports, limiting the number of VC-4s that can be routed through the cross-connect). This implies that every arc also has a second associated numerical value, the *capacity u(a)*.

This results in a communication network that is modeled by a ***weighted, capacitated digraph***.

To simplify our terminology, we will use from now on the simple term *network N* to denote such a weighted, capacitated digraph.

We also have to model the amount of information transferred via the links of the network. Furthermore, we want to model this as simple as possible, to concentrate in our calculations on the most essential features of the information transfer. This transport is modeled by the abstract concept ***flow***.

Network flow-associated constraints

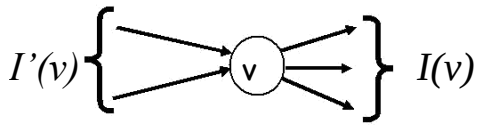
flow → function $f(.) : A \rightarrow Z^+$ or R^+
subject to the following constraints:

$$\sum_{a \in I(v)} f(a) - \sum_{a \in I'(v)} f(a) = b(v), \forall v \in V \qquad (1)$$

$$0 \le f(a) \le u(a \qquad a \in A \qquad (2)$$

flow conservation

capacity constraints



u(a) = capacity of link

- supply vertex : $b(v) > 0$ (source)
- demand vertex : $b(v) < 0$ (sink)

Graph foundations of network modelling 46

The flow is a function $f(.)$: A→ Z^+ or R^+ on the arcs of the network.

The flow in a communications network is always subject to two ever-recurring constraints:

- the flow conservation constraints (1)
- capacity constraints (2)

We refer to the constraints (1) as the *flow conservation constraints* or the mass balance constraints. The first term in this constraint for a vertex v represents the total outflow of the vertex v (i.e. the flow leaving v) and the second term represents the total inflow of the vertex (i.e. the flow entering v). The flow conservation constraints state that the outflow minus the inflow must equal the *supply/demand* $b(v)$ of the vertex v. In SDH networks for example, a VC-4 arriving at a cross-connect cannot just disappear, but is either routed to another cross-connect or fulfills a local demand. Notice the resemblance between (1) and 'Kirchhoff's node law' for electrical circuits.

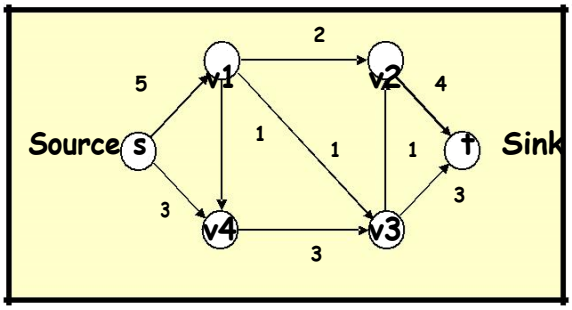
If the vertex is a *supply vertex* ($b(v) > 0$), its outflow exceeds its inflow. If the vertex is a *demand vertex* ($b(v) < 0$), its inflow exceeds its outflow. In network flow problems with only one supply vertex and one demand vertex, these vertices are often denoted as *source* and *sink*.

When the links have associated capacities, the flow on the associated arcs must also satisfy the upper bound (2). The flow bounds typically model physical capacities or restrictions imposed on the flow's operating range. In communication networks, it is obvious that only positive flows are meaningful. However, in other applications, more general bounds might exist (e.g. also a lower bound: $l(a) \le f(a) \le u(a)$).

Outline

0. Introduction
1. Definition and notations
2. Minimum spanning tree problem
3. Shortest path problem
4. **Maximum flow problem**
   - **4.1 Network flow**
   - **4.2 Problem definition**
   - **4.3 Residual network concept**
   - **4.4 Flow augmenting path algorithm**
   - **4.5 Max-flow min-cut theorem**
   - **4.6 Application: calculation of connectivity**
5. Minimum cost flow problem
6. Extension: multi-commodity flow problems

Problem definition : maximum flow



maximize F

subject to

$$\sum_{a \in I(v)} f(a) - \sum_{a \in I'(v)} f(a) = \begin{cases} F, & \text{if } v = s \\ 0, & \forall v \in V \setminus \{s,t\} \\ -F, & \text{if } v = t \end{cases}$$

$$0 \leq f(a) \leq u(a), \forall a \in A$$

Graph foundations of network modelling 48

Consider an IP-network with a server at one location (i.e. a vertex) and a client located at another point of the network. Suppose that we want to know the highest data rate that we can achieve, assuming that only the capacities of the transmission facilities (i.e. the capacities of the links) limit the bitrate. If we model the IP-network as a capacitated digraph, as described in subsection 4.1., we can formulate our question mathematically. The resulting network flow problem is called the ***maximum flow problem*** which is formulated as follows:

We consider a capacitated digraph $D = (V,A)$ with a nonnegative capacity u(a) associated with each arc a $\in$ A. To define the maximum flow problem, we distinguish two special vertices in D: a source vertex s and a sink vertex t. We wish to find the maximum flow from the source vertex s to the sink vertex t, that satisfies the arc capacities and flow conservation constraints. In general, there can exist several optimal flows, all having the same maximal value.

In a sense, the maximum flow problem is complementary to the shortest path problem. The shortest path problem models situations in which flow incurs a cost, but is not restricted by any capacities. In contrast, flow in the maximum flow problem incurs no costs, but is restricted by capacities.

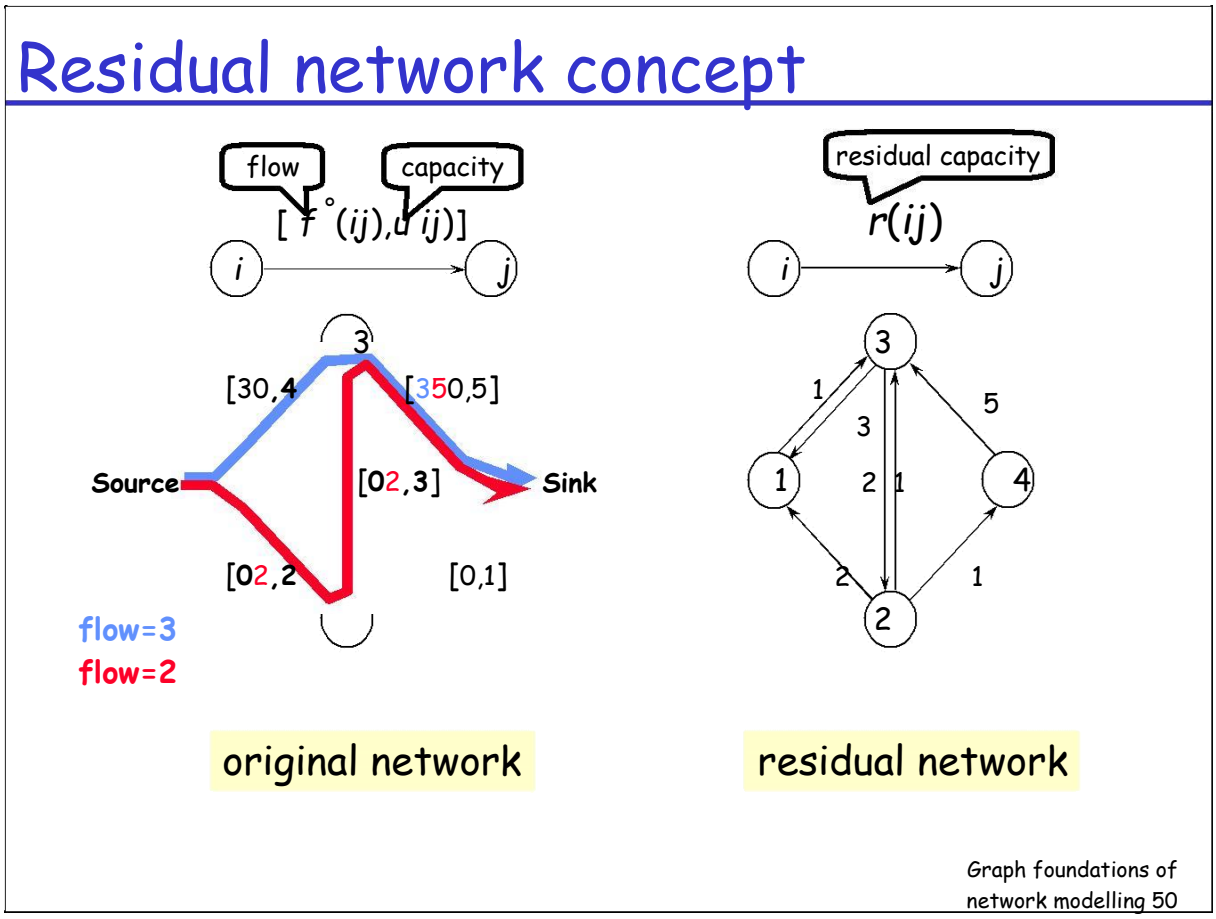
Other applications of the maximum flow problem are for example:

• computation of the maximum number of VC-4 paths that can be restored when a link has broken, by using the spare capacity of the network;

• computation of link and vertex connectivity (see subsection 4.6).

Outline

0. Introduction
1. Definition and notations
2. Minimum spanning tree problem
3. Shortest path problem
4. **Maximum flow problem**
   - **4.1 Network flow**
   - **4.2 Problem definition**
   - **4.3 Residual network concept**
   - **4.4 Flow augmenting path algorithm**
   - **4.5 Max-flow min-cut theorem**
   - **4.6 Application: calculation of connectivity**
5. Minimum cost flow problem
6. Extension: multi-commodity flow problems



In designing, developing and implementing network flow algorithms, it is often convenient to measure flow not in absolute terms, but rather in terms of incremental flow with respect to some given feasible solution - typically the solution at some intermediate point in an algorithm. This leads us to the definition of a new, auxiliary network, known as the *residual network* that represents the temporal network situation.

The concept of residual networks is based on the following intuitive idea. Suppose that during some algorithm calculations, the arc (i,j) carries $f°(ij)$ units of flow. The main question is now: to what extent can this temporal flow value be changed? Because we have to respect the arc capacities, we can only send $u(ij) - f°(ij)$ units of additional flow from vertex i to vertex j along arc (i,j). Also notice that we can send up to $f°(ji)$ units of flow from vertex j to vertex i over the arc (i,j), which comes in fact down to canceling the existing flow on the arc (i,j).

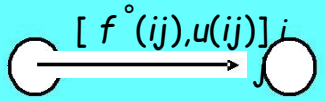
Using these ideas, we define the residual network with respect to a given flow f as follows. We replace each arc (i,j) in the original network by two arcs (i,j) and (j,i): the arc (i,j) has a residual capacity $r(ij) = u(ij) - f°(ij)$, and the arc (j,i) has a residual capacity $r(ji) = f°(ij)$. By convention, we only include arcs with a strictly positive residual capacity into the residual network.

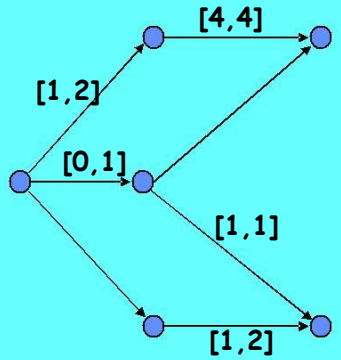
We use the notation $N(f°)$ to represent the residual network corresponding to a certain flow $f°$.

Remark: Notice that if a network N contains both the arcs (i,j) and (j,i), the residual network may contain two (parallel) arcs from vertex i to vertex j, and/or two (parallel) arcs from vertex j to vertex i. When solving the maximum flow problem, we can merge both of the parallel arcs into a single arc and set its residual capacity equal to the sum of the residual capacities of the two arcs. (However, when considering the minimum cost flow algorithm in the next section, this merging of parallel arcs will not be possible anymore.)

Residual Network

EXERCISE





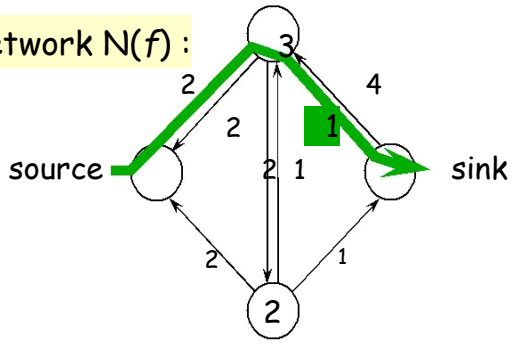
Outline

0. Introduction
1. Definition and notations
2. Minimum spanning tree problem
3. Shortest path problem
4. **Maximum flow problem**
   - **4.1 Network flow**
   - **4.2 Problem definition**
   - **4.3 Residual network concept**
   - **4.4 Flow augmenting path algorithm**
   - **4.5 Max-flow min-cut theorem**
   - **4.6 Application: calculation of connectivity**
5. Minimum cost flow problem
6. Extension: multi-commodity flow problems

Flow Augmenting Path Algorithm

determine the maximum flow between two vertices *s* and *t* in a network *N*.

residual network N(*f*) :



augmenting path ?

d = residual capacity of augmenting path

```
(1)   f := 0;
(2)   while ( N (f ) contains at least one augmenting path from s to t ) do
 (3)          identify one augmenting path P from vertex s to vertex t;
 (4)          d := min{r(ij) : ij ∈ P};
 (5)          augment the flow f with d units along P;
 (6)          update N (f );
(7)   od
```

Graph foundations of network modelling 53

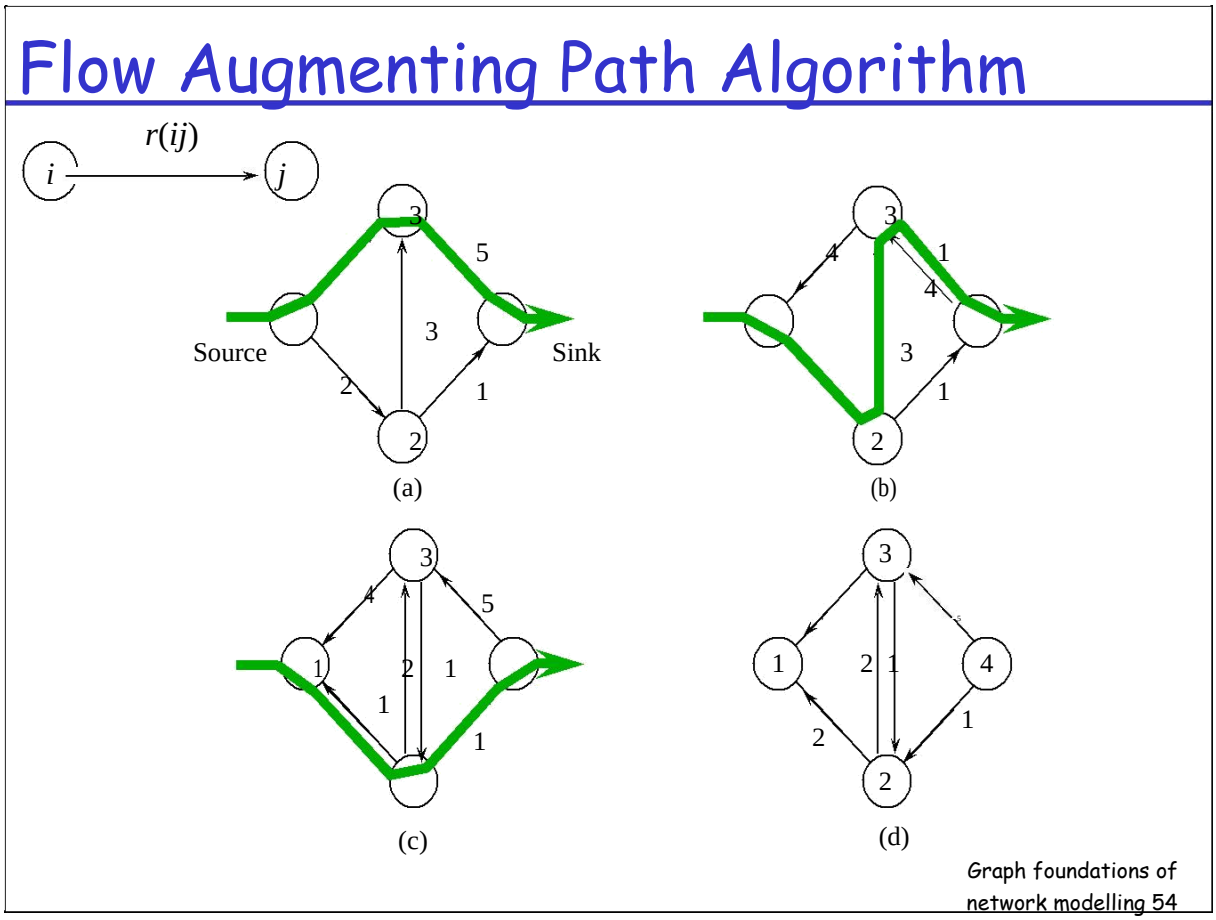
The flow augmenting path algorithm is one of the simplest and most intuitive algorithms for solving the maximum flow problem.

We refer to a directed path from the source to the sink in the residual network as an augmenting path P. We define the residual capacity d of an augmenting path as the minimum residual capacity (r(ij)) of any arc along the path. The residual capacity d of an augmenting path is always positive. Consequently, whenever the network contains an augmenting path, we can send additional flow from the source to the sink.

The generic augmenting path algorithm is essentially based on this simple observation. We start from an initial situation (1) with no flows in the network. The algorithm proceeds by identifying augmenting paths, augmenting flows on these paths and adjusting the residual network accordingly. This goes on until the residual network contains no such augmenting path anymore.

The validity of the algorithm relies on the following theorem:

Theorem (Augmenting Path Theorem). *A flow f* is a maximal flow if and only if the residual network N(f*) contains no augmenting path.*



The algorithm is illustrated on the simple problem shown above: we want to find the maximum flow between vertices 1 and 4 in the network given in the figure (a):

Suppose that the algorithm selects the path 1-3-4 for augmentation. The residual capacity of this path is d = min{r(13), r(34)} = min{4,5} = 4. This augmentation reduces the residual capacity of arc (1,3) to zero (thus we delete it from the residual network) and increases the residual capacity of arc (3,1) to 4 (so we add this arc to the residual network). The augmentation also decreases the residual capacity of arc (3,4) from 5 to 1 and increases the residual capacity of arc (4,3) from 0 to 4.

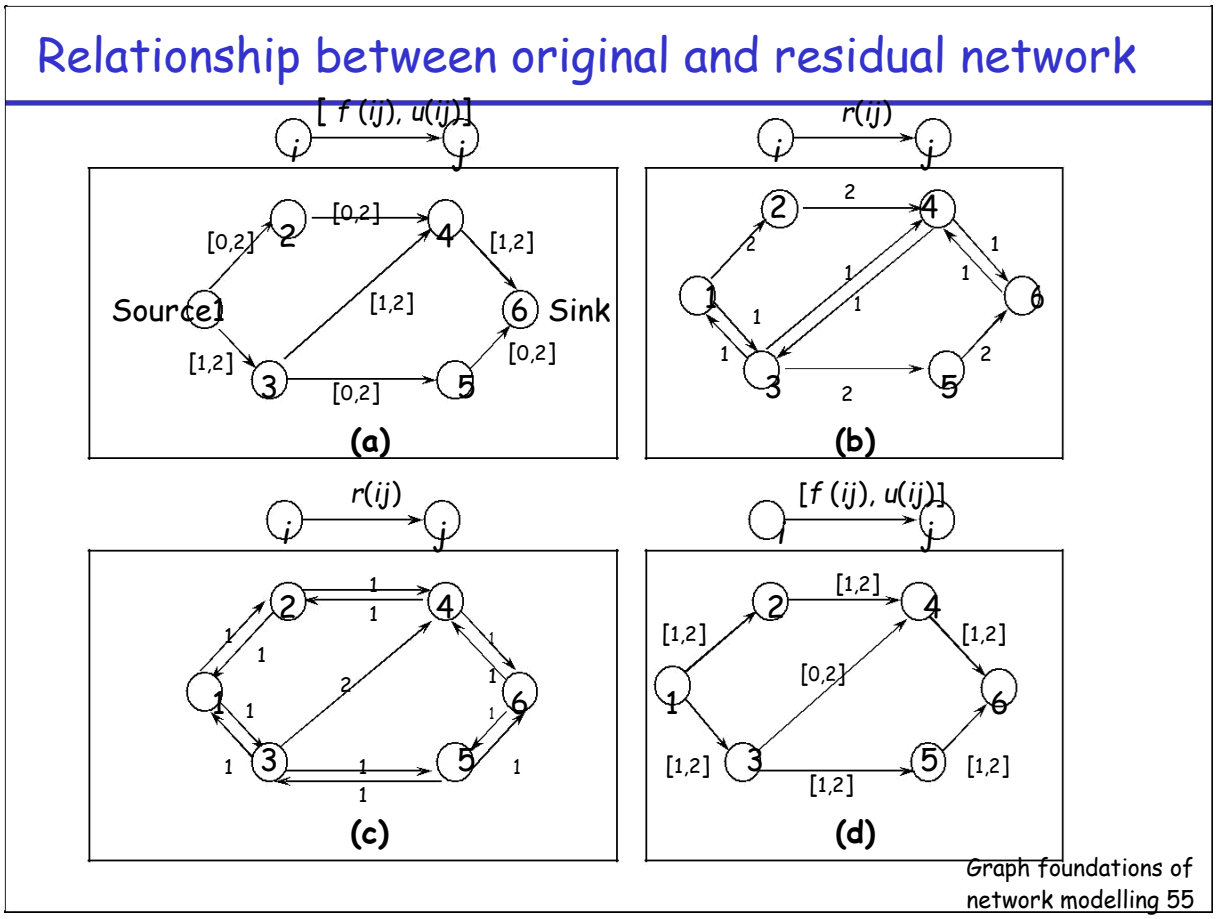
In the second step, suppose that the algorithm selects the path 1-2-3-4. The residual capacity of this path is d = min{r(12), r(23), r(34)} = min{2,3,1} = 1. Augmenting 1 unit of flow along this path yields the residual network shown in (c).

In the third step, the algorithm augments 1 unit of flow along the path 1-2-4. Figure (d) shows the corresponding residual network. Now the residual network contains no augmenting path, so the algorithm terminates.

The figure illustrates the following phases in the algorithm:

(a) residual network for the zero flow; (b) residual network after augmenting four units along the path 1-3-4; (c) residual network after augmenting one unit along the path 1-2-3-4; (d) residual network after augmenting one unit along the path 1-2-4.

After the last augmentation, $N(f)$ no longer contains any directed path from 1 to 4, so the maximum 1, 4-flow is equal to 6.



To understand the relationship between residual and original network, it is useful to consider the concept of an augmenting path in the original network as well.

Suppose that the residual capacities $r(ij)$ are updated at some point in the algorithm. What is the effect on the arc flows $f(ij)$ in the original network? The definition of the residual capacity (i.e. $r(ij) = u(ij) - f(ij) + f(ji)$) implies that an additional flow of d units on arc (i,j) in the residual network corresponds to (1) an increase in $f(ij)$ by d units in the original network, or (2) a decrease in $f(ji)$ by d units in the original network, or (3) a convex combination of (1) and (2).

We use the example given in figure (a) and the corresponding residual network in figure (b) to illustrate these possibilities. Augmenting 1 unit of flow on the path 1-2-4-3-5-6 in the network produces the residual network in figure (c) with the corresponding arc flows shown in figure (d). Comparing the solution in figure (d) with that in figure (a), we find that the flow augmentation increases the flow on arcs (1,2), (2,4), (3,5), (5,6) and decreases the flow on arc (3,4).

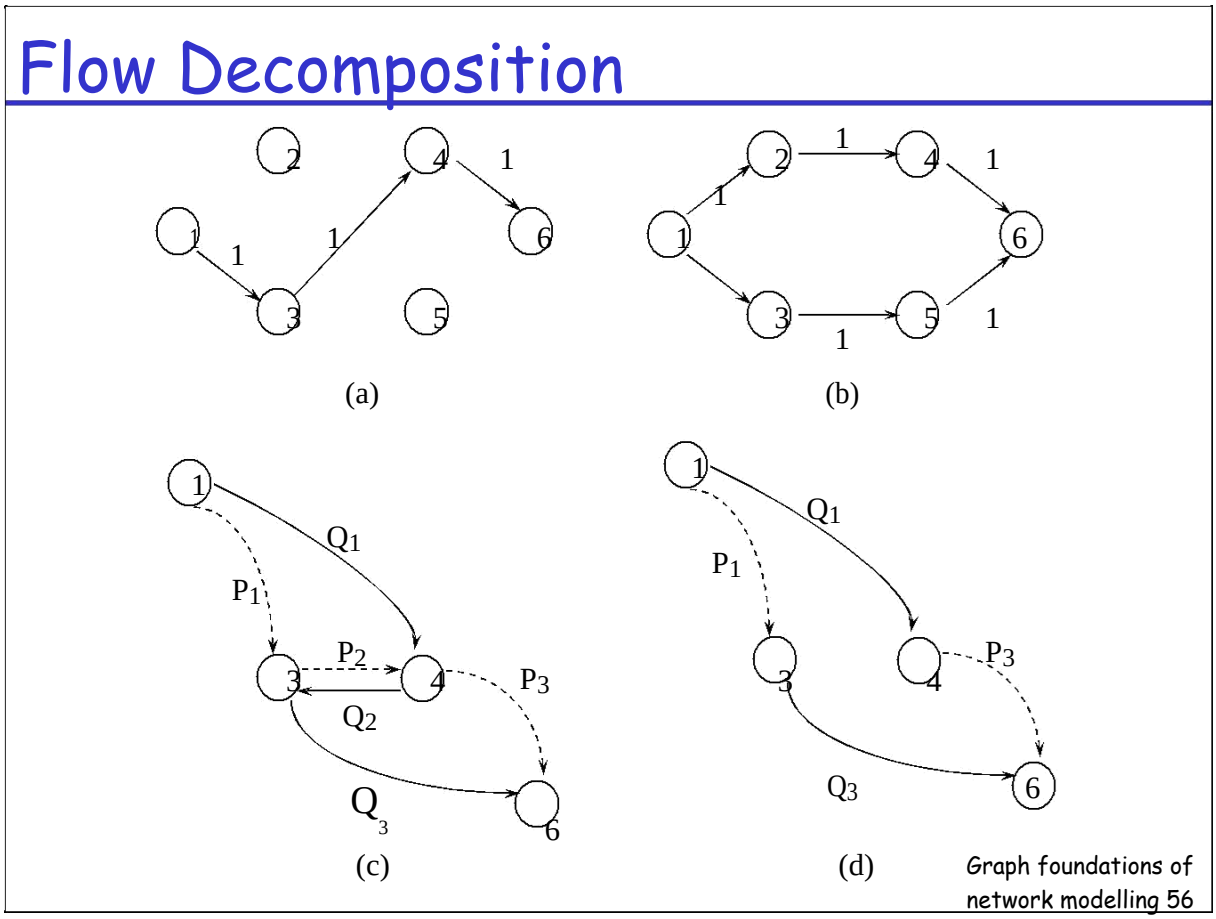
Finally, suppose that the values for the residual capacities $r(ij)$ and of course also $u(ij)$ are given. How should we determine the flows $f(ij)$? Observe that since $r(ij) = u(ij) - f(ij) + f(ji)$, many combinations of $f(ij)$ and $f(ji)$ correspond to the same value of $r(ij)$.

We can determine one such choice as follows:

1. Let us rewrite $r(ij) = u(ij) - f(ij) + f(ji)$ as $f(ij) - f(ji) = u(ij) - r(ij)$.

2.1. If $u(ij) \geq r(ij)$, we set $f(ij) = u(ij) - r(ij)$ and $f(ji) = 0$;

2.2. Otherwise, we set $f(ij) = 0$ and $f(ji) = r(ij) - u(ij)$.



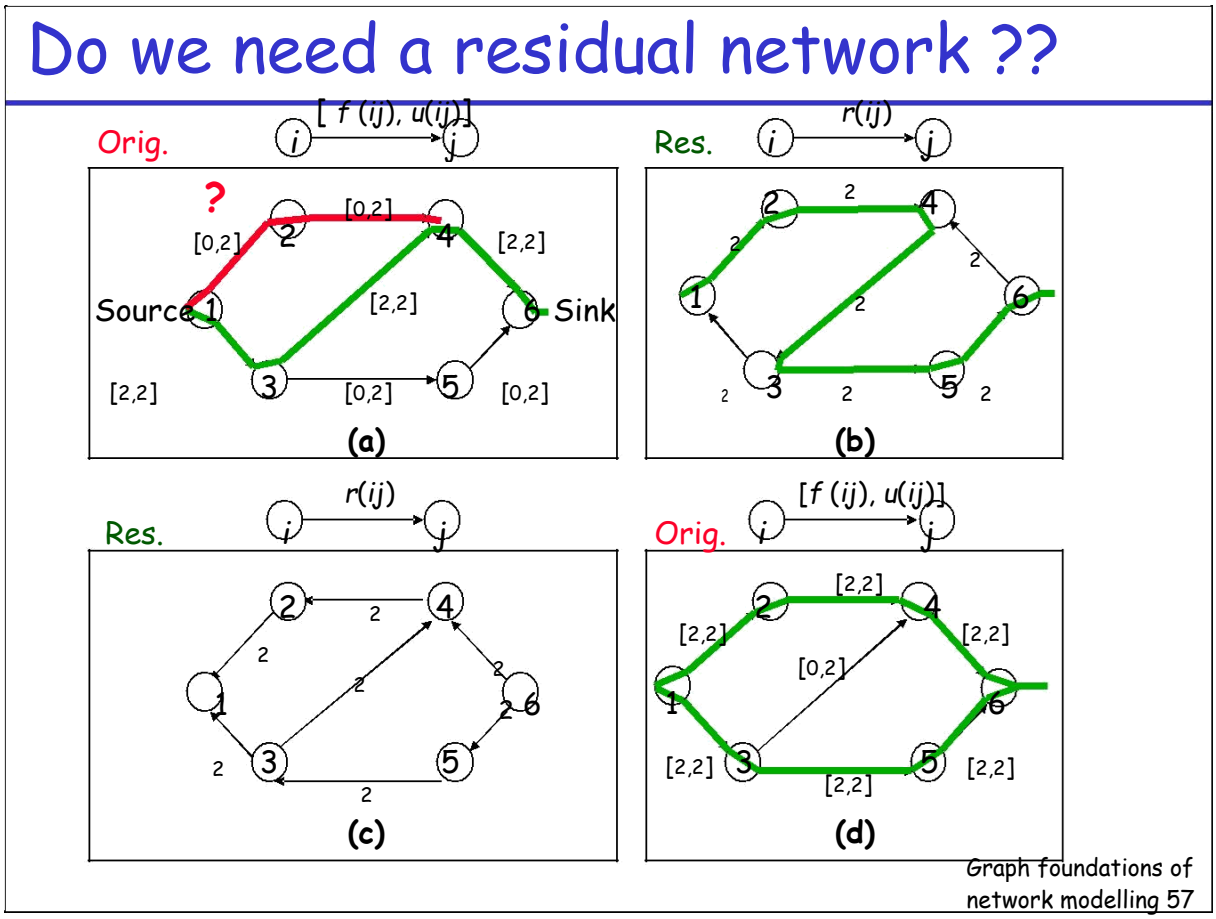
To obtain better insight concerning the augmenting path algorithm, we illustrate the effect of an augmentation on the flow decomposition. We use the preceding example.

Figure (a) gives the decomposition of the initial flow and figure (b) gives the decomposition of the flow after we have augmented 1 unit of flow on the path 1-2-4-3-5-6. Remark that although we augmented 1 unit of flow along the path 1-2-4-3-5-6, the flow decomposition contains no such path.

The path 1-3-4-6 defining the flow in figure (a) can be divided in three segments: the path up to vertex 3 (P_1), arc 3-4 as a forward arc (P_2), and the path up to vertex 6 (P_3). We can view this path as an augmentation on the zero flow. Similarly, the path 1-2-4-3-5-6 contains three segments: the path up to vertex 4 (Q_1), arc 3-4 as a backward arc (Q_2), and the path up to vertex 6 (Q_3).

We can view the augmentation on the path 1-2-4-3-5-6 as linking the initial segment of the path 1-3-4-6 with the last segment of the augmentation, linking the last segment of the path 1-3-4-6 with the initial segment of the augmentation, and canceling the flow on arc (34), which then drops from both path 1-3-4-6 and the augmentation (see figure (c) : two augmentations P_1-P_2-P_3 and Q_1-Q_2-Q_3 ; figure (d) : net effect of these augmentations).

In general, we can view each augmentation as 'pasting together' segments of the current flow decomposition to obtain a new flow decomposition.



This example clearly shows the advantage of the residual network:

(a) we set up a path of 2 flow units between 1 and 6 over vertices 3 and 4. If we look to the original network we can still try to set up a connection from 1 to 2 to 4 but then it blocks.

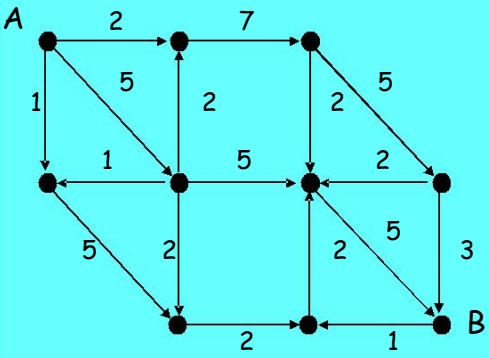
(b) If we look at the residual network however we observe that there still exists a flow augmenting path 1-2-4-3-5-6 with a capacity of 2.

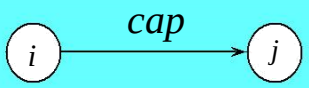
(c) This results in a residual network where no further flow increase is possible.

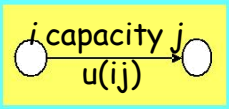
(d) Going back to the original network learns us that we can set up a total flow of 4 units by using the paths 1-2-4-6 and 1-3-5-6.

Maximum Flow Algorithm

EXERCISE







Maximum Flow Algorithm: applications

Characteristics max. flow solution:

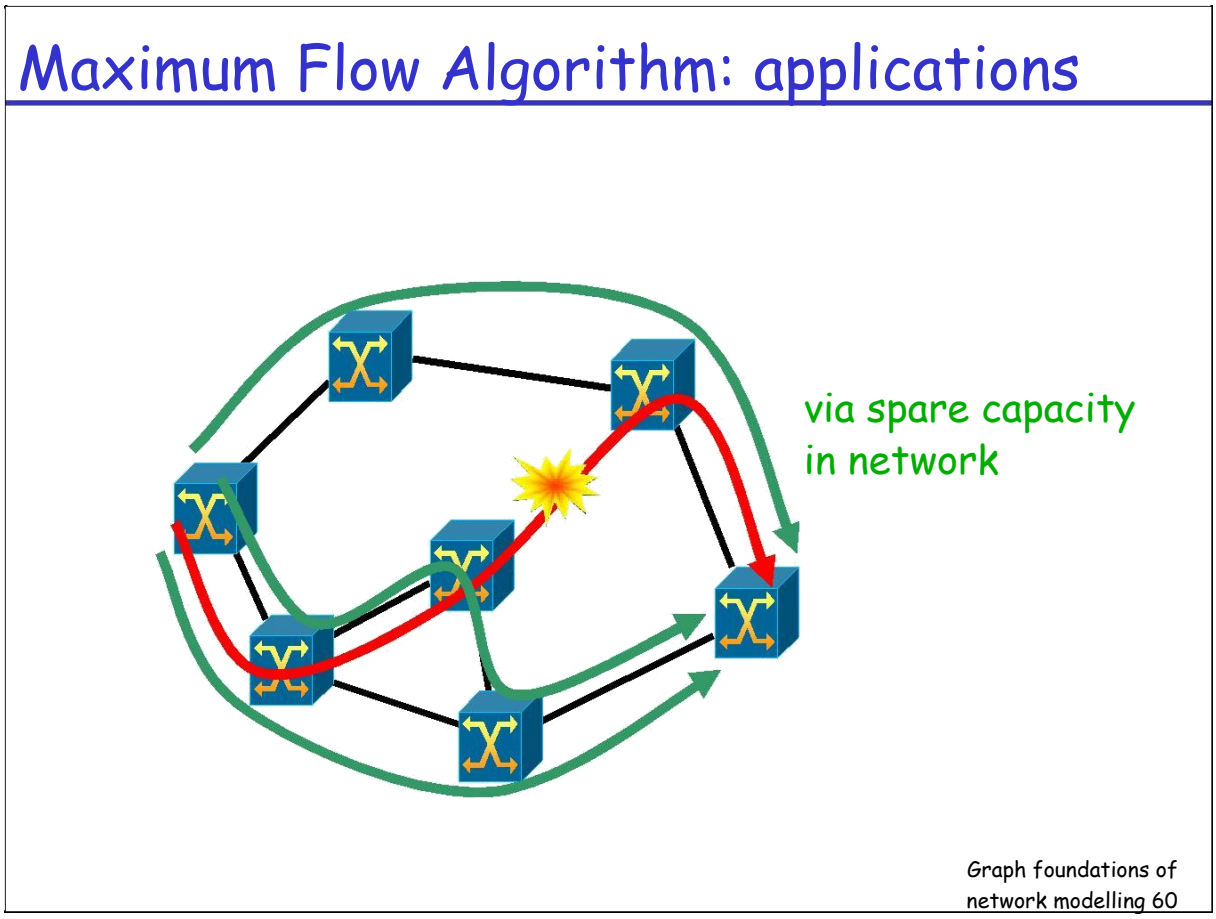
- from one source to one destination
- maximal use of network BW
  - ⇒ bifurcation of flow
  - ⇒ compliant with application ? (different delays, reordering, …)
- cost of paths not taken into account

Some applications:

- maximum BW from server ⇗ client
- indication on available BW source ⇗ destination
- restoration mechanism: available spare capacity ?

The main characteristics of the maximum flow problem are highlighted in the slide above. First of all, the problem concentrates on a single source node and a single destination (situations with multiple sources and/or multiple destinations will be considered further on in this course). Secondly, the main idea behind the maximum flow problem is to exploit the bandwidth available throughout the network to send as much information as possible from source to destination. This means that in many cases, the result will be a flow pattern consisting of several paths that bifurcate at some node(s) and merge again at other node(s). It depends on the considered situation whether this bifurcation, leading to several paths with different conditions, is allowed. For instance, a bifurcated packet flow with different delays along the paths may result in an out-of-order arrival of these packets at the destination node. If this is a problem, a reordering mechanism will be needed at the destination to restore the correct packet order. Finally, note that the maximum flow algorithm does not take any costs into account: the augmenting path is chosen randomly, regardless the cost of this path. These cost aspects will be added to the problem in section 5.

From these characteristics, some typical applications of the maximum flow problem in communications networks become apparent. First of all, the algorithm can be used to determine the flow pattern realising the maximum bandwidth flow from a server to a user (for instance to transfer a large file as quick as possible over the network). The maximum flow algorithm can also be used to get an indication on the bandwidth that is in fact available in the network from a particular source to a particular destination (indicating how 'good' the network performs for this source-destination pair). The algorithm is also applied frequently in network recovery mechanisms, for instance to restore the traffic affected by a failure exploiting the spare capacity in the network. This is illustrated on the next slide.



In the figure, an optical transport network is shown, consisting of seven optical cross-connects interconnected with optical fibers. A connection (red) is set up in this network from OXC A to OXC B. Suddenly, one of the links fails (see figure). To restore this network connection, a restoration algorithm could be implemented, looking for the maximum flow (green) that can be sent from A to B through the network, using only the spare capacity in the network and of course without using the failed link.

Outline

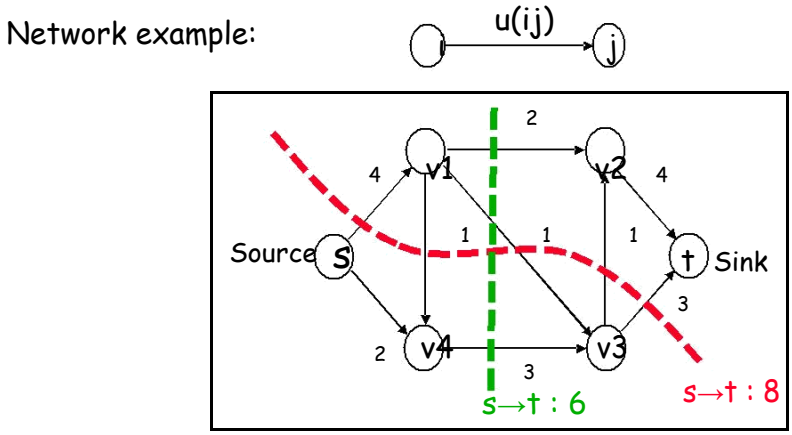
0. Introduction
1. Definition and notations
2. Minimum spanning tree problem
3. Shortest path problem
4. **Maximum flow problem**
   - **4.1 Network flow**
   - **4.2 Problem definition**
   - **4.3 Residual network concept**
   - **4.4 Flow augmenting path algorithm**
   - **4.5 Max-flow min-cut theorem**
   - **4.6 Application: calculation of connectivity**
5. Minimum cost flow problem
6. Extension: multi-commodity flow problems

The Maximum-Flow Minimum-Cut Theorem

correspondence between flows and cuts in a network

minimum cut problem :
"From among all cuts in the network that separate the source and the sink, find the cut with minimum capacity".

Network example:



Graph foundations of network modelling 62

The maximum flow - minimum cut theorem establishes an important correspondence between flows and cuts in networks. Indeed, as we will see, by solving a maximum flow problem, we also solve a complementary *minimum cut problem* : "From among all cuts in the network that separate the source and the sink, find the cut with minimum capacity". Recall from section 1 that a cut is a set of arcs whose deletion disconnects the network in two parts. The *capacity of a cut* is the sum of the capacities of the arcs that are part of it.

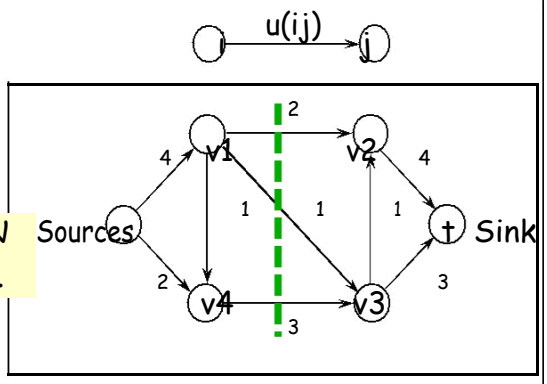
The relationship between maximum flows and minimum cuts is important for several reasons. First, it embodies a fundamental duality result that arises in many mathematical optimisation problems. It has also some practical implications: the theorem implies that algorithms and theory developed for the maximum flow problem are also applicable to many practical problems that are naturally cast as minimum cut problems.

It also allows to prove several important results in combinatorics that appear difficult to prove using other means. e.g. in the following subsection, we will use the following implications of the max flow - min cut theorem:

The maximum number of edge-disjoint (or vertex-disjoint) paths connecting two vertices *s* and *t* in a graph equals the minimum number of edges (or vertices) whose removal from the graph leaves no path from *s* to *t*.

Before stating the maximum flow - minimum cut theorem itself, we recall some definitions and mention some useful properties of flows and cuts.

The Maximum-Flow Minimum-Cut Theorem



Consider a partition {X,Y} of the vertices of N (i.e. $X \cup Y = V$ and $X \cap Y = \varnothing$) where $s \in X$ and $t \in Y$.

$$[X:Y] = \{a \in A | a = ij, i \in X, j \in Y\}$$

$$capacity([X:Y]) = \sum_{a \in [X:Y]} u(a)$$

e.g. we choose the s,t-cut [X:Y]
with X={s,v1,v4} and Y={v2,v3,t}
⇒ Then we have capacity ([X:Y]) = 6

Graph foundations of network modelling 63

Consider a network $N = (V,A)$, a source vertex s and a destination vertex t in N (i.e. $s,t \in V$). As we have seen in the previous paragraph, a flow f in N from s to t with value F is a function $f(.)$ satisfying the following conditions:

$$\forall a \in A : 0 \le f(a) \le u(a) \qquad (1)$$

$$\forall v \in V : \sum_{a \in I(v)} f(a) - \sum_{a \in I'(v)} f(a) = \begin{cases} F, & v = s \\ 0, & v \in V \setminus \{s, t\} \\ -F, & v = t \end{cases} \qquad (2)$$

Consider now a partition $\{X,Y\}$ of the vertices of N (i.e. $X \cup Y = V$ and $X \cap Y = \varnothing$) where $s \in X$ and $t \in Y$. Then $[X:Y]$ is an s,t-cut set in N with capacity given by

$$[X:Y] = \{a \in A | a = (ij), i \in X, j \in Y\}$$
$$capacity([X:Y]) = \sum_{\in [X:Y]} u(a) \qquad (3)$$

Property. *Let $N = (V,A)$ be a network, $s,t \in V$ and $f(.)$ a flow from s to t in N, with value F. If the partition $\{X,Y\}$ of V induces an s,t-cut $[X:Y]$ in N, then*

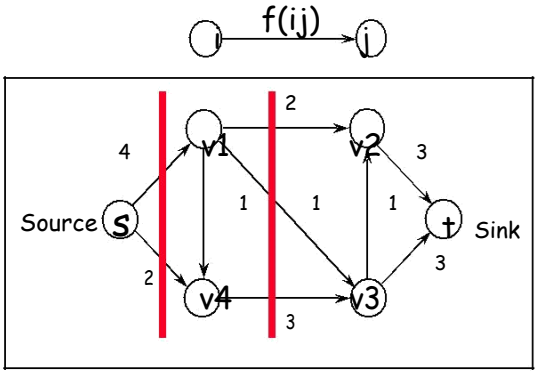
$$F = \sum_{a \in [X:Y]} f(a) - \sum_{a \in [Y:X]} f(a) \qquad (4)$$

A proof of this theorem follows easily from the conditions (2) which $f(.)$ must satisfy.

The Maximum-Flow Minimum-Cut Theorem

Theorem: The maximum value of a flow between s and t
=
capacity of a minimum s,t-cut

Example: maximum s,t-flow F = 6



minimum-cut

Combination of (1), (3) and (4) also leads to the following result:

Property. *Let N be a network and f an s,t-flow in N. Then the value of the flow f cannot exceed the capacity of any s,t-cut in N.*

This property is quite intuitive: Any flow from vertex s to vertex t must pass through every s-t cut in the network (because any cut divides the network into two disjoint components), and therefore the value of the flow can never exceed the capacity of the cut.

This property implies that if we discover a flow f between s and t whose value equals the capacity of some s,t-cut, then f is a maximum flow and the cut is a minimum cut.

This property is in fact stating that maximum flow ≤ minimum cut. On the other hand, one has proved that the minimum cut will certainly be reached by a maximum flow: maximum flow ≥ minimum cut (without proof, try to understand this intuitively by thinking on the flow augmenting path algorithm). These two properties lead to the maximum flow-minimum cut theorem:

Theorem. (Max-Flow Min-Cut Theorem) *The maximum value of the flow from a vertex s to a vertex t in a capacitated digraph equals the minimum capacity among all s-t cuts.*

The max-flow min-cut theorem suggests a method for determining the s,t-cut with minimal capacity in a graph. Using the flow augmentation method of the maximum flow algorithm, an s,t-flow with maximum value can be determined. The max-flow min-cut theorem then learns us that the value of this maximal s,t-flow is equal to the capacity of the minimal s,t-cut.

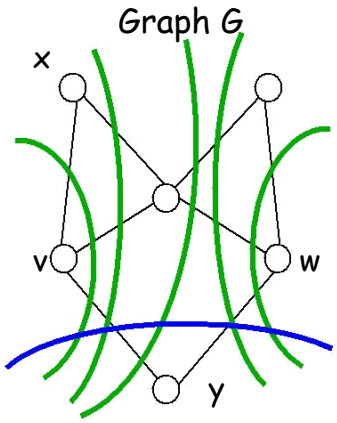
Outline

0. Introduction
1. Definition and notations
2. Minimum spanning tree problem
3. Shortest path problem
4. **Maximum flow problem**
   - **4.1 Network flow**
   - **4.2 Problem definition**
   - **4.3 Residual network concept**
   - **4.4 Flow augmenting path algorithm**
   - **4.5 Max-flow min-cut theorem**
   - **4.6 Application: calculation of connectivity**
5. Minimum cost flow problem
6. Extension: multi-commodity flow problems

Calculation of edge-connectivity

λ(v,w) = minimum cardinality of a v,w-edge-cutset

λ(G) = edge-connectivity of graph G = min{λ(v,w) | v,w ∈ V }.

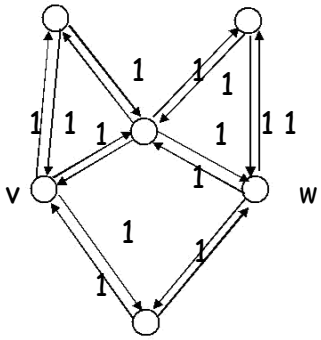


λ(v,w) = 3
λ(x,y) = 2
…
} → λ(G) = 2

How to calculate λ(v,w) ?

Max flow min cut theorem :
λ(v,w) = maximum flow F(v,w)
(if all link capacities = 1)

Residual Network :



λ(v,w) = max. # edge-disjoint paths between v and w

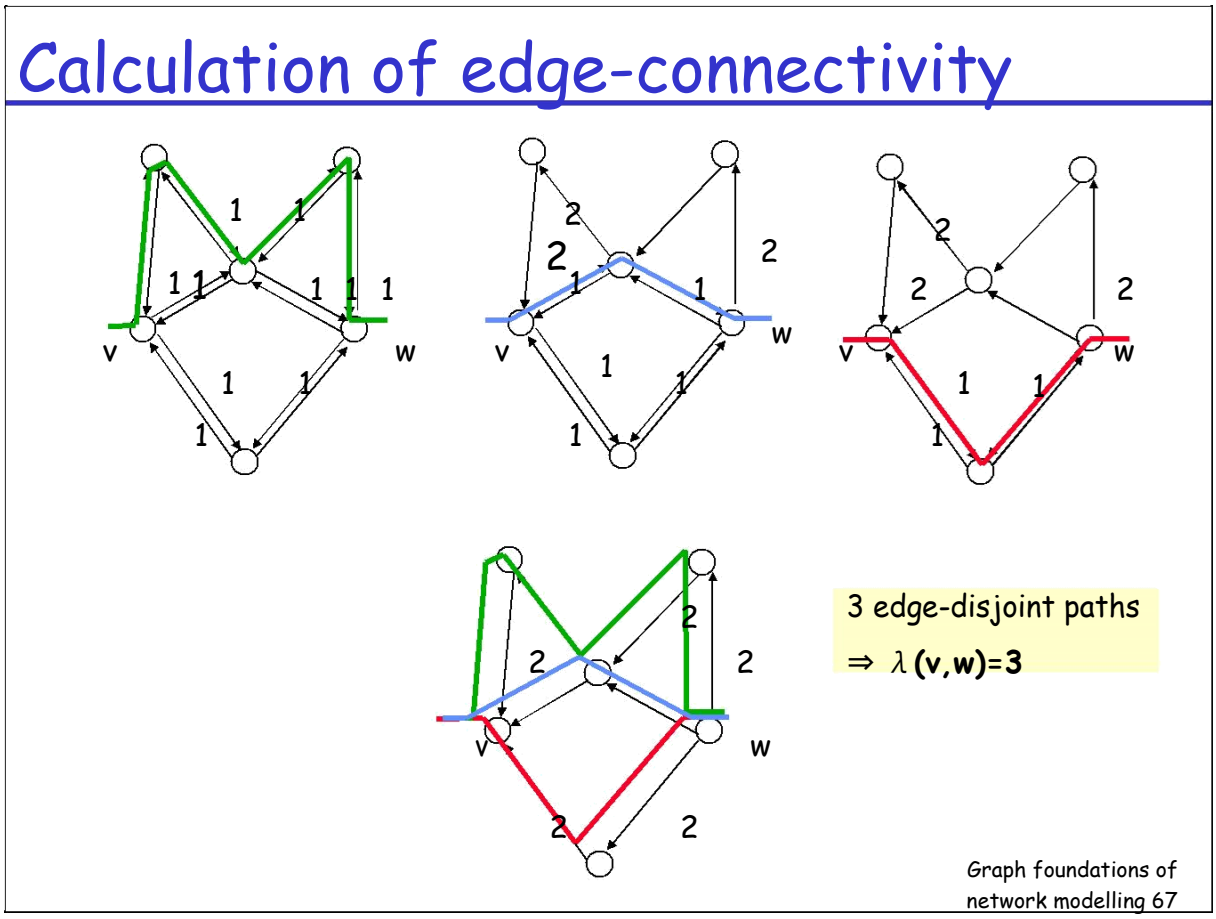
Graph foundations of network modelling 66

From subsection 1.1 one can easily see that the *edge-connectivity* λ(G) of a graph G=(V,E) can be defined as min{λ(v,w) | v,w ∈ V}, where λ(v,w) is the minimum cardinality of a v,w-edge-cutset (i.e. a edge-cutset that disconnects v from w). Now the maximum flow - minimum cut theorem can be used to calculate λ(G) as follows.

The minimum cardinality of a v,w-edge-cutset is in fact the minimum cut in a capacitated graph G', being G with all edge capacities set to 1. Thanks to the maximum flow - minimum cut theorem, finding this minimum cut is equivalent with determining the maximum flow F(v,w) from v to w (or vice versa) in the capacitated digraph G'', being G' where every edge {u,v} has been replaced by two anti-parallel arcs (u,v) and (v,u), each with capacity 1 (see picture above). By repeatedly applying the flow augmenting path algorithm, we can determine F(v,w) for every pair of nodes v and w. The minimal value for F(v,w) is the edge-connectivity λ(G) of the original graph G.

It is important to note that finding a maximum flow from v to w in the graph G'' comes in fact down to determining the maximum number of pairwise *edge-disjoint* v,w-paths in G. Two v,w-paths are edge-disjoint if they do not have any edges in common. This leads to an alternative definition of edge-connectivity of a graph : a graph G = (V,E) is *k-edge-connected* if and only if every two vertices of G are connected by at least k edge-disjoint paths.

Task : Of course, the edge-connectivity of a graph can also be found by simply searching for the minimum number of edges that must be removed to disconnect the graph. For which type of graphs is this latter method to be preferred ; for which type of graphs will the maximum flow calculations be more efficient ?



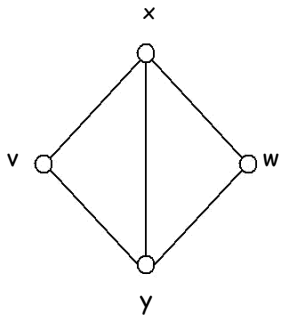
The figure shows an example of the calculation of edge-connectivity between v and w by using the maximum flow.

Calculation of vertex-connectivity

$\kappa(v,w)$ = minimum cardinality of a v,w-vertex-cutset (v,w non adjacent)

$\kappa(G)=\min\{\kappa(v,w) \mid v,w \in V, vw \notin E\}$

Graph G



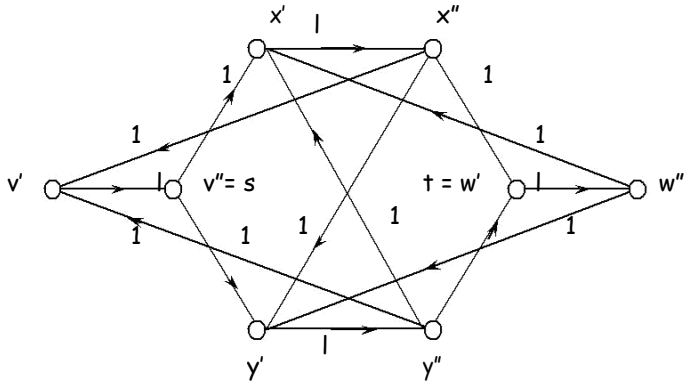
κ(v,w) = 2

κ(G) = 2

How to calculate κ(v,w)

κ(v,w) = max. # vertex-disjoint paths between v and w

- Split vertices : v', v''
- Interconnect vertices : v'→v'', v'←w'', v''→w' (all link capacities = 1)



- κ(v,w) = maximum flow F(v'',w')

Graph foundations of network modelling 68

The flow augmenting path algorithm can also be used to determine the *vertex-connectivity* $\kappa(G)$ of a graph G=(V,E).

For two non-adjacent (why ?) vertices v,w of G, we define κ(v,w) as the minimum cardinality of a v,w-vertex-cutset (i.e. a vertex-cutset (not containing v and w) that disconnects v from w). It is clear that $\kappa(G)=\min\{\kappa(v,w) \mid v,w \in V, vw \notin E\}$ (except if the graph G is complete, i.e. every pair of vertices is connected by a link). In a similar way as for edge -connectivity, we can prove that κ(v,w) equals the number of pairwise vertex-disjoint v,w - paths in G. Two v,w-paths are vertex-disjoint if they don't have any vertices in common (except for the vertices v and w themselves, of course). To find vertex-disjoint paths by applying the flow augmenting path algorithm, the graph G must be transformed as follows.

1. create a capacitated graph G', being graph G with all link capacities set to 1

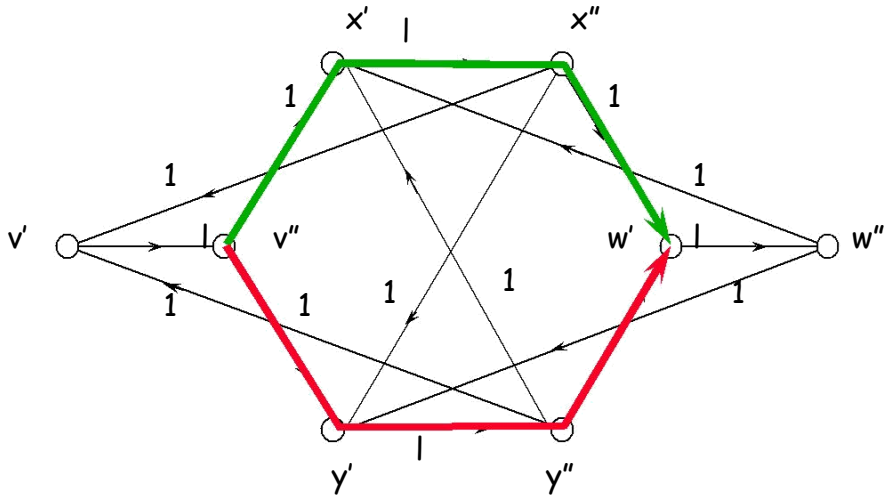
2. create a capacitated digraph G'', being G' where every edge {u,v} has been replaced by two anti-parallel arcs (u,v) and (v,u), each with capacity 1

3. create a capacitated digraph G'''. Each vertex v of G'' is mapped to two vertices v' and v'' of G'''. Every arc incident to v in G'' is mapped to an arc incident to v' in G''', every arc incident from v in G'' is mapped to an arc incident from v'' in G'''. Finally, an arc is created from v' to v'' in G'''. Give each arc in G''' a capacity equal to 1.

By determining the maximum possible flow value between v'' and w' in G''', we find pairwise vertex-disjoint v,w-paths in the original graph G (why ?).

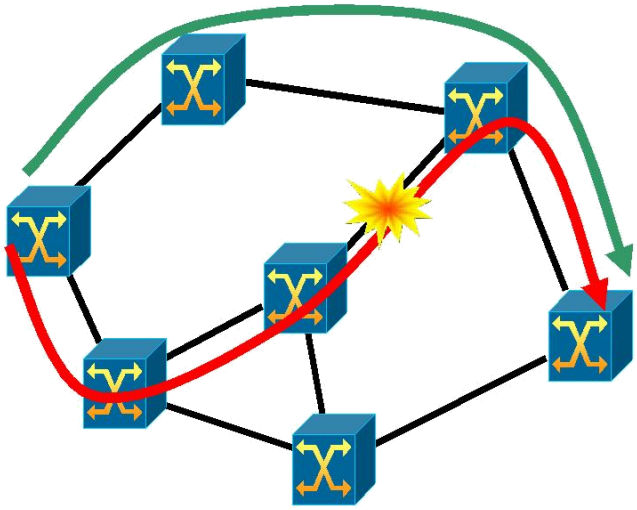
A graph G = (V,E) is *k-vertex-connected* if and only if every two vertices of G are connected by at least k vertex-disjoint paths.

Calculation of vertex-connectivity



The figure shows an example of the calculation of vertex connectivity between v and w by using the maximum flow.

Connectivity: applications



Network survivability:

- to cope with single link failures: edge-connectivity ≥ 2
- to cope with single node failures: vertex-connectivity ≥ 2
- to cope with two links failing simultaneously: edge-connectivity ≥ 3
- ...

Graph foundations of network modelling 70

One of the main applications of the graph connectivity concept is found in network survivability and recovery mechanisms. A network topology with edge- and vertex-connectivity equal to 1 can provide connections between every two nodes in the network (as long as there is enough capacity on the links, of course). However, if a failure occurs, large amounts of (possibly crucial) data can be lost.

In many cases, especially with geographically large networks, single link failures are the most probable failures (due to civil works for instance). For instance, in a pan-European network with about 10000 km of cable, a link failure can be expected about every two weeks ! As it is unacceptable for many services and customers that some part of the network becomes unreachable for some hours (time to detect, locate and repair the failure locally) every two weeks, the network should be able to cope with at least a single link failure. This imposes some constraints on the network topology: its edge-connectivity should at least be 2. As indicated in the previous slides, this implies that we can find two edge-disjoint paths between every source and destination node. These two edge-disjoint paths are candidate paths for a primary and backup path in a network recovery scheme.

If also single node failures (e.g. due to software or hardware failure in a router) have a high probability (the probability of a node failure is typically in the same order of magnitude as a link failure if we consider networks within a smaller geographical region), the limitation on the network topology comes down to a vertex-connectivity of 2. This means that if a single node A fails, all traffic that is not terminated in node A can be rerouted.

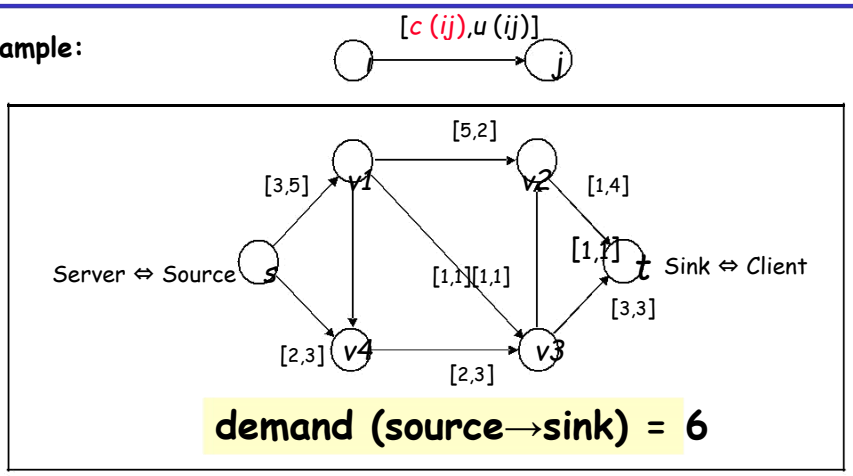
The probability of having two links failing at the same time (for instance one link is still being repaired while another goes down) is typically much smaller than having a single (link or node) failure. For extremely critical services, this can also be taken into account when designing the network topology, leading to a minimum edge-connectivity of 3.

Outline

0. Introduction
1. Definition and notations
2. Minimum spanning tree problem
3. Shortest path problem
4. Maximum flow problem
5. **Minimum cost flow problem**
   - 5.1 Introduction
   - 5.2 Problem definition
   - 5.3 Residual network
   - 5.4 Busacker-Gowen algorithm
   - 5.5 Several supply/demand vertices
6. Extension: multi-commodity flow problems

Introduction

Example:



Minimize the total cost of flow
with respect to how to route the
flow subject to
- the flow value on each arc respects the arc capacity
- the total flow fulfills the demand between *s* and *t*

Graph foundations of network modelling 72

Consider the same IP-network as in subsection 4.2., with a server at one location (i.e. a vertex) and a client located at another point of the network. This time, instead of searching the maximum transfer amount we can achieve via the network, we want to route a certain given demand from the server to the client through the network. Moreover, we have to take into account the *costs* incurred for using the link facilities. We assume that this flow cost varies linearly with the amount of flow on each arc.

Our objective is to find a flow of minimum total cost, which transfers the given amount of information between the server and the client (we refer to the amount of data units that must be transferred as the *demand d*). We still have to respect the capacities of the transmission facilities. This implies that in our example (see slide above), we cannot send everything along a shortest path. Instead, we will have to decompose the flow into different paths.

The resulting network flow problem is called the ***minimum cost flow problem***, which can be stated as follows. We consider a network $N = (V,A)$ as defined in subsection 4.1. We wish to determine the flow $f(a)$ on the arcs $a \in A$ with minimum total cost $\sum f(a)c(a)$, that respects the capacities $u(a)$ of the arcs $a \in A$, and fulfills the demand d between s and t.

Outline

0. Introduction
1. Definition and notations
2. Minimum spanning tree problem
3. Shortest path problem
4. Maximum flow problem
5. **Minimum cost flow problem**
   - **5.1 Introduction**
   - **5.2 Problem definition**
   - **5.3 Residual network**
   - **5.4 Busacker-Gowen algorithm**
   - **5.5 Several supply/demand vertices**
6. Extension: multi-commodity flow problems

Problem definition

Optimisation model for minimum cost flow problem

$$\text{minimize} \sum_{a \in A} c(a)\, f(a)$$

subject to

$$\sum_{a \in I(v)} f(a) - \sum_{a \in I'(v)} f(a) = \begin{cases} d, & \text{if } v = s \\ 0, & \forall v \in V \setminus \{s, t\} \\ -d, & \text{if } v = t \end{cases}$$

$$0 \le f(a) \le u(a), \qquad \forall a \in A$$

Graph foundations of network modelling 74

Let $N = (V,A)$ be a network, i.e. a digraph with a cost $c(a)$ and capacity $u(a)$ associated with every arc $a \in A$. Assume that we want to send d units between two vertices of V, s and t. The corresponding minimum cost flow problem can then be formulated mathematically as shown in the slide above.

It is interesting to note that both shortest path and maximum flow problem are in fact *special cases* of the minimum cost flow problem. The shortest path problem models situations where we want to send flow through an uncapacitated network (i.e. $u(a) = \infty$, $\forall\ a \in A$). Indeed, if we wish to send d units of flow from vertex s to vertex t and the capacity of every arc in the network is unlimited, then we would send the complete flow along a shortest path from s to t. The maximum flow problem models situations where we want to send a huge amount of information (as much as possible) through a network with capacity restrictions but without costs (i.e. $c(a) = 0$, $\forall\ a \in A$).

Since minimum cost flow problems combine the problem ingredients of both problems, a lot of material (algorithms, theory, ..) developed for shortest path and maximum flow can be extended for the minimum cost flow problem. E.g. the Busacker-Gowen algorithm, discussed in subsection 5.4, extends the idea of flow augmenting paths (cf. maximum flow problem) to a network with costs using a shortest path algorithm as a subprocedure.

Outline

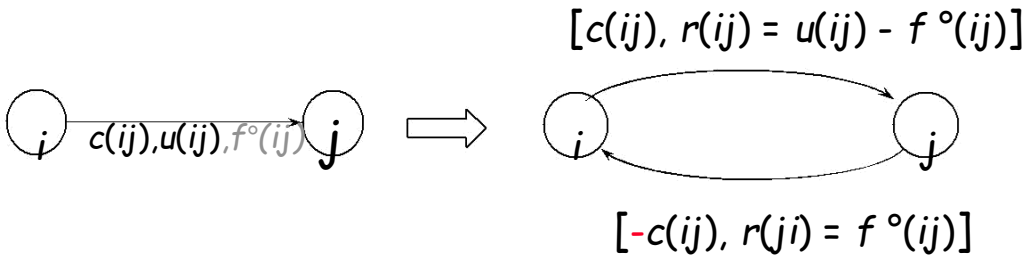
0. Introduction
1. Definition and notations
2. Minimum spanning tree problem
3. Shortest path problem
4. Maximum flow problem
5. **Minimum cost flow problem**
   - **5.1 Introduction**
   - **5.2 Problem definition**
   - **5.3 Residual network**
   - **5.4 Busacker-Gowen algorithm**
   - **5.5 Several supply/demand vertices**
6. Extension: multi-commodity flow problems

Residual network concept

original network N
with a certain flow f °(ij)

corresponding
residual network N(f °)

every arc (i,j) in original network ⇒ two arcs (i,j) and (j,i) in residual network



Graph foundations of
network modelling 76

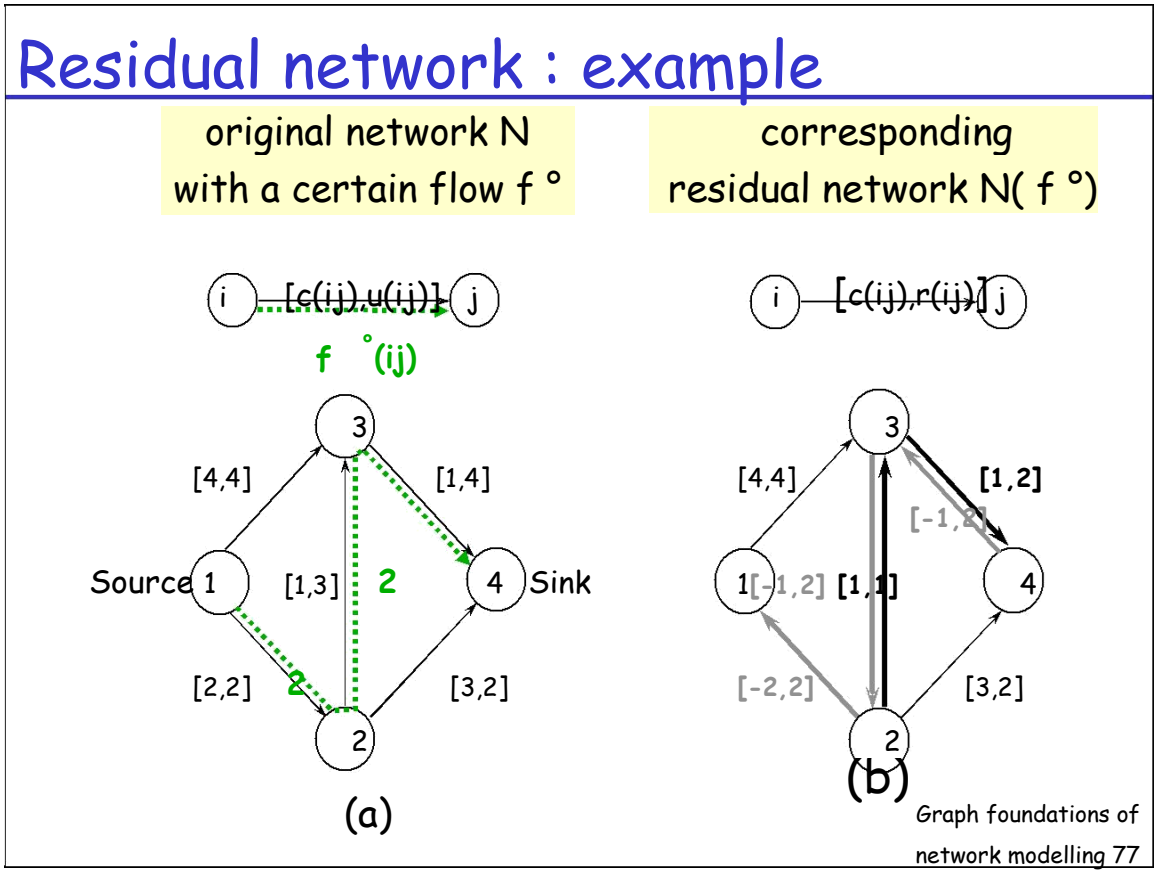
We already introduced residual networks as a useful concept for solving the maximum flow problem (see subsection 4.3): The residual network functioned as the temporal network situation during the calculation of the maximum flow in the flow augmenting path algorithm.

When we want to use the ideas of residual networks for solving the minimum cost flow problem as well, the reasoning of subsection 4.3 must first be extended to residual networks with costs. To find out which cost should be associated to each arc, we must think about what 'cost' really means here.

Suppose that arc (i,j) carries $f°(ij)$ units of flow. Because we have to respect the arc capacities, we can only send $u(ij) - f°(ij)$ units of additional flow from vertex i to vertex j along arc (i,j). We also noticed that we can send up to $f°(ij)$ units of flow from vertex j to vertex i over the arc (i,j), which amounts to canceling the existing flow on the arc (i,j). To build up the residual network, we replace each arc (i,j) in the original network by two arcs (i,j) and (j,i): the arc (i,j) has a residual capacity $r(ij) = u(ij) - f°(ij)$, and the artificial arc (j,i) has a residual capacity $r(ji) = f°(ij)$.

The cost of the original arc (i,j) in the residual network is the price we have to pay per unit of flow augmentation on that arc. However, augmenting the flow on an artificial arc (j,i) (which is always associated with an original arc (i,j)) corresponds to reducing the flow on the original arc (i,j). This means that whereas sending a unit of flow from vertex i to vertex j on arc (i,j) increases the total flow cost by $c(ij)$ units, sending a unit of flow from vertex j to vertex i on the same arc decreases the total flow cost by $c(ij)$ units.

Hence, flow augmentation on an artificial arc should be rewarded rather than penalized. Therefore, we assign a 'cost' of $-c(ij)$ to the artificial arc (j,i), where $c(ij)$ is the cost of the corresponding original arc.



In summary, the construction of the residual network $N(f°)$ with respect to a given flow $f°$ goes as follows.

We replace each arc (i,j) in the original network by two arcs (i,j) and (j,i): the arc (i,j) has a residual capacity $r(ij) = u(ij) - f°(ij)$ and cost $c(ij)$, and the arc (j,i) has a residual capacity $r(ji) = f°(ij)$ and cost $-c(ij)$. By convention, only arcs with a strictly positive residual capacity are included in the residual network.

If a network N contains both the arcs (i,j) and (j,i), the residual network may contain two *parallel arcs* from vertex i to vertex j, and/or two (parallel) arcs from vertex j to vertex i. It is important to highlight that, in contrast to residual network for the maximum flow problem, we can **not** merge both of the parallel arcs into a single arc. Indeed, in most of the residual networks $c(ji) \neq -c(ij)$, which means that the parallel arcs will have different costs.

In the figure above, we give an example of a network N with a certain flow $f°$ and its corresponding residual network $N(f°)$. In the residual network, the flow is not explicitly mentioned, but is instead implicitly contained in the residual capacities.

The gray arcs in the residual network are also called 'artificial arcs', because they do not exist in the original network. Each arc in the original network with a strictly positive flow value has such an associated artificial arc in the residual network corresponding to that flow. Augmenting flow on artificial arcs in the residual network corresponds to a decrease of the flow on the associated arcs in the original network.

Outline

0. Introduction
1. Definition and notations
2. Minimum spanning tree problem
3. Shortest path problem
4. Maximum flow problem
5. **Minimum cost flow problem**
   - **5.1 Introduction**
   - **5.2 Problem definition**
   - **5.3 Residual network**
   - **5.4 Busacker-Gowen algorithm**
   - **5.5 Several supply/demand vertices**
6. Extension: multi-commodity flow problems

The algorithm of Busacker and Gowen

algorithm to determine a minimum cost flow f in a network N from a source vertex s to a sink vertex t with value d (demand).

(1) $f := 0$; // initially there is no flow in the network
(2) **while** ($F < d$) **do**
(3) identify the shortest augmenting path P from s to t in $N(f)$;
(4) $\delta := \min \{ \min\{ r(ij) : ij \in P \}, (d - F) \}$;
(5) augment the flow f along P with δ;
(6) update $N(f)$;
(7) $F := F + \delta$;
(8) **od**

Remarks : (i) It is assumed that a feasible flow from s to t with value d is possible.

(ii) Flow value = $F = \sum_{a \in I(s)} f(a)$

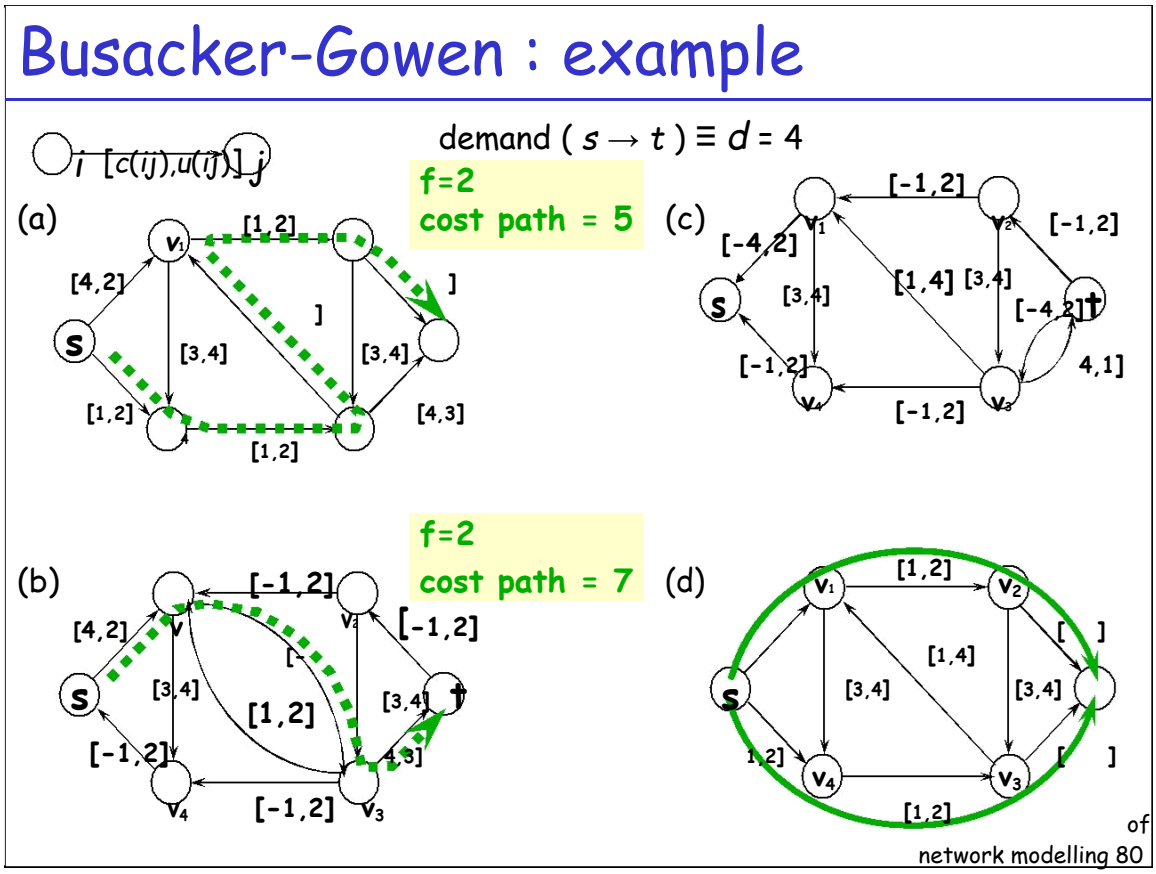
Graph foundations of network modelling 79

The *algorithm of Busacker and Gowen* (also called the *successive shortest path algorithm)*, can be used to solve the minimum cost flow problem defined in the subsection 5.2, where there is one supply vertex (called source vertex *s*) and one demand vertex (called sink vertex *t*). The algorithm successively identifies shortest augmenting paths in the residual network, and augments flow along such paths until the total flow fulfills the demand.

When the algorithm is executed, the residual network $N(f)$ will change as the optimal flowvector f is constructed. As illustrated on the previous pages, $N(f)$ contains original arcs (that were also present in N) as well as artificial arcs. As the artificial arcs normally have a negative cost (assuming that all original costs are positive), the calculation of the shortest path in step (3) must be done using the Ford-Belman-Moore algorithm (cf. subsection 3.3).

Notice the resemblance between this algorithm and the flow augmenting path algorithm of subsection 4.4 for the maximum flow problem. They both identify successive augmenting paths in the residual network. However, in the maximum flow algorithm any augmenting path can be used, whereas Busacker-Gowen searches for the shortest among the augmenting paths. This implies that a shortest path procedure is required, while in the maximum flow algorithm, a generic 'path-searching' procedure (which can be less complex) can do the job.

We thus perceive the relation between the three discussed network problems. In general, many algorithms for solving minimum cost flow problems indeed combine ingredients of both shortest path algorithms and maximum flow problems.



We now illustrate the algorithm of Busacker and Gowen on an example network (weighted capacitated digraph). We are going to determine the minimum cost flow between *s* and *t* with total flow value (demand) equal to 4.

Figure (a) shows the original network together with the first flow-augmentation. The first flow augmenting path is *s*-v_4-v_3-v_1-v_2-*t*, the 'cost' of this path is 5 and the capacity available along this path is 2. The cost of the first flow augmentation is 10 (amount of flow augmentation times the 'cost' of the path).

Figure (b) shows the residual network after the first flow augmentation and the second flow-augmentation. The second flow augmenting path is *s*-v_1-v_3-*t*, which has a cost equal to 7. The capacity available along this path is 2. We need an augmentation of 2 to have a total flow of 4. The cost of the second flow augmentation is 14 (amount of flow augmentation times the 'cost' of the path).

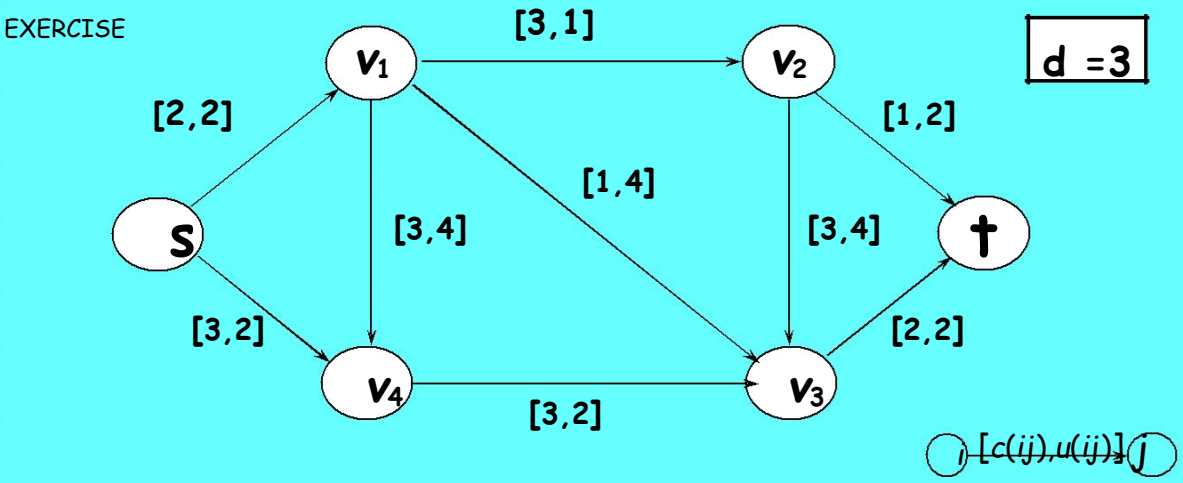
Figure (c) shows the residual network after the second flow augmentation.

Figure (d) shows the resulting minimum cost flow with total cost 10+14=24. The resulting flow pattern consists of a flow of value 2 along *s*-v_1-v_2-*t* and a flow of value 2 along *s*-v_4-v_3-*t*.

It may be instructive to see what happens when the value of the minimum cost flow we are looking for is equal to 3. In that case the optimal flow pattern will be different.

Busacker-Gowen

EXERCISE



Outline

0. Introduction
1. Definition and notations
2. Minimum spanning tree problem
3. Shortest path problem
4. Maximum flow problem
5. **Minimum cost flow problem**
   - **5.1 Introduction**
   - **5.2 Problem definition**
   - **5.3 Residual network**
   - **5.4 Busacker-Gowen algorithm**
   - **5.5 Several supply/demand vertices**
6. Extension: multi-commodity flow problems

Several supply/demand vertices

Optimisation model :

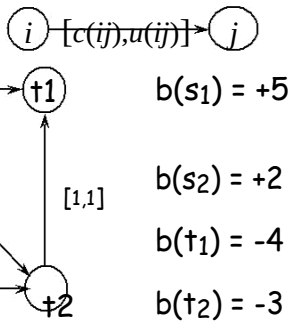
$$\text{minimize} \sum_{a \in A} c(a)\, f(a)$$

subject to

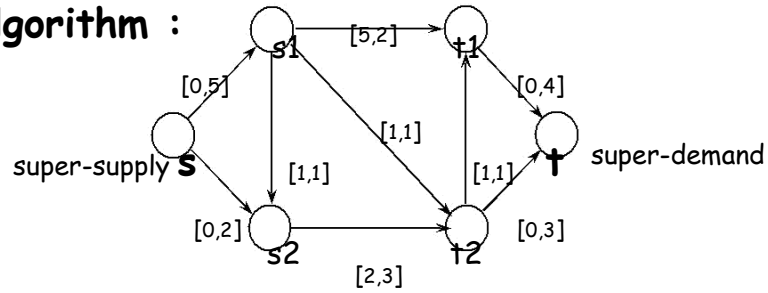
$$\sum_{a \in I(v)} f(a) - \sum_{a \in I'(v)} f(a) = b(v) \;, \quad \forall\, v \in V$$

$$0 \le f(a) \le u(a) \;, \quad \forall\, a \in A$$

Example :



Min cost flow algorithm :



demand = 5+2 = 4+3 = 7

We also note that it is possible to formulate a more general min cost flow model, with more than one demand and/or more than one supply vertex. In this case, we associate a function $b(v)$ to the vertices $v \in V$, which represents the supply/demand of the vertex v. ($b(v) > 0 \rightarrow$ supply vertex, $b(v) < 0 \rightarrow$ demand vertex). This leads to the optimisation model mentioned above. A simple example network is also depicted above.

At first sight, it might seem much more difficult to find an algorithm which solves this more general problem. However, such an algorithm can easily be derived from the preceding Busacker-Gowen algorithm as follows (the supply vertices are $s_1, \ldots, s_m$, the demand vertices are $t_1, \ldots, t_n$) :

1. Add a 'super-supply vertex' σ to the network and connect σ to every supply vertex s_i (i=1,…,m) with an arc (σ, s_i) with cost 0 and capacity $b(s_i)$. Add a 'super-demand vertex' τ to the network and connect every demand vertex t_i ($i = 1,\ldots,n$) to τ with an arc (t_i, τ) with cost 0 and capacity $-b(t_i)$ (see picture above).

2. Apply the Busacker-Gowen algorithm on the min cost flow problem with one supply vertex σ and one demand vertex τ to transfer the demand

$$d = \sum_{i=1}^{m} b(s) = \sum_{i=1}^{n} -b(t_i)$$

It is obvious that this two-phase procedure will yield the optimal solution for the min cost flow problem with several supply/demand vertices.

Outline

0. Introduction
1. Definition and notations
2. Minimum spanning tree problem
3. Shortest path problem
4. Maximum flow problem
5. Minimum cost flow problem
6. **Extension: multi-commodity flow problems**

The concept 'commodity'

A commodity = one uniform product

Sections 3, 4 and 5 : single-commodity flow problems

e.g. oil / electricity distribution system

This section : multi-commodity flow problems

e.g. telephone network

Several multi-commodity problems : natural extensions of the single-commodity problems

e.g. multi-commodity minimum cost flow problem

Graph foundations of network modelling 85

A *commodity* is a uniform product, which means that no distinction can be made between different units of that product.

An important distinction between different network problems can be made depending on the question whether all products transported by the network are uniform (a so-called *single-commodity problem*) or not (a so-called *multi-commodity problem*). E.g.

- In an oil distribution system, all oil is considered to be the same. This has important consequences : if we consider some supply vertices (places where oil is pumped into the network) and some demand vertices (users), it does not matter for the user from which supply vertex the oil is coming.
- If, on the contrary, we consider a telephone network, a totally different situation arises. Here the supply vertices are the people that have dialed a number, the demand vertices are the people that are phoned. Since a bitstream associated to a telephone call typically has one particular source and one particular destination, it does really matter from which source the bitstream is coming and to which destination it is going. We can express this enhanced complexity by considering each phone call as a separate commodity. A network problem which models a telephone network situation should incorporate the fact that several commodities must be transported by the network.

In the previous sections of this chapter, only single-commodity problems were considered. (Do not confuse the several supply/demand vertices min cost flow problem of the previous page with a multi-commodity situation !) For each single-commodity problem, a multi-commodity counterpart exists. As an example, the multi-commodity minimum cost flow problem is described in the next slide.

Multi-commodity min cost flow problem

Optimisation model for the multi-commodity min cost flow problem :

minimize $\sum_{a \in A} c(a) \sum_{k \in K} f^k(a)$ weighted capacitated digraph

subject to $\sum_{a \in I(v)} f^k(a) - \sum_{a \in I'(v)} f^k(a) = b_k(v)$, $\forall v \in V$, $\forall k \in K$

$0 \le \sum_{k \in K} f^k(a) \le u(a)$, $\forall a \in A$

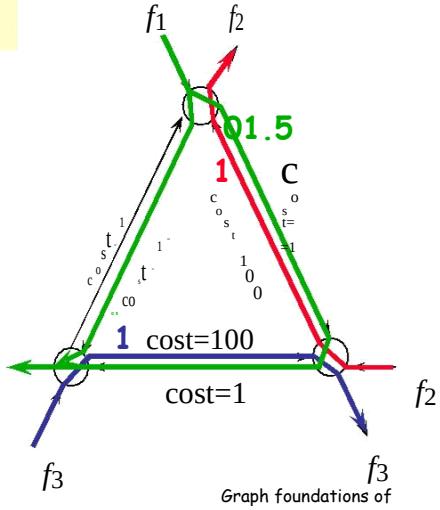
Distinction between two problems :

- $f^k(a)$ integer : integer MCMCF problem
- $f^k(a)$ real : linear MCMCF problem

Capacity of arc = 1

$f_1=f_2=f_3=1$ f_1 ⟵ integer MCMCF : cost = 202

linear MCMCF : cost = 153



Graph foundations of network modelling 86

The givens in a multi-commodity minimum cost flow problem are :

- a network (=weighted capacitated digraph) N=(V,A);
- a set K of commodities, each commodity k is specified by the integers $b_k(v)$, $\forall v \in V$ ($b_k(v) > 0$ if v is a supply vertex for commodity k, $b_k(v) < 0$ if v is a demand vertex for commodity k).

The optimisation model for the multi-commodity minimum cost flow problem is mentioned above. The decision variables are $f^k(a)$, i.e. the flow of commodity k on arc a. The objective is to minimize the total cost of the flow through the network. The constraints are the flow conservation constraints for each commodity individually and the capacity constraints (the sum of the flows of the different commodities on an arc can not exceed the capacity of the arc).

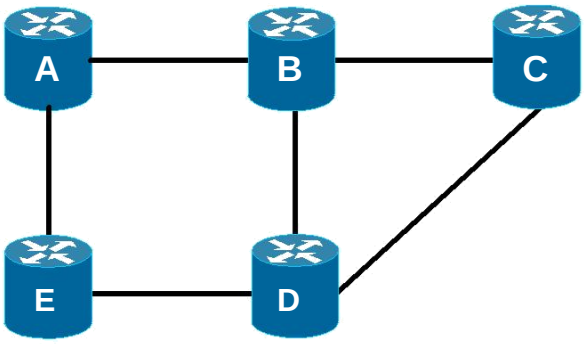
When we considered the single-commodity minimum cost flow problem (see section 5), no distinction was made between the problem with real variables f(a) and the problem with integer variables f(a). The reason is that for integer arc capacities there is always an integer optimal solution (a proof of this statement can be found in literature).

However, this property does not hold for multi-commodity problems anymore ! Even with integer arc capacities, the optimal solution might have fractional flow values. This is illustrated on the simple triangular network situation above, where all arc capacities are set to one. We want to send one unit flow from vertex 1 to vertex 2 (f_1), one unit from vertex 3 to vertex 1 (f_2) and one unit from vertex 2 to vertex 3 (f_3). The calculation of the optimal solution for the linear and the integer problem is left to the reader (solution : total cost = 153 (linear problem) and 202 (integer problem)).

For some applications, fractional flow values are not allowed (e.g. in SDH it is not possible to send half a VC-12 container along a link). In these situations, we have to solve the integer multi-commodity minimum cost flow problem (which is far more complex than the linear equivalent).

Application: IP routing vs. MPLS Traffic Eng.

Problem: demands for traffic between every pair of IP(/MPLS) routers



IP:

only shortest paths allowed

⇗ algorithm of Dijkstra

IP/MPLS:

explicit routes possible ('traffic engineering')

mathematical formulation: multi-commodity flow problem very refined granularity (one IP packet = some bytes ↔ typical arc capacity = 100s Mbit/s) ⇒flow variable has an almost continuous nature

⇒ real $f^k(a)$ variables allowed

Graph foundations of network modelling 87

For instance, if we consider the routing decisions in IP networks, with or without MPLS functionality, two clearly distinct situations arise.

In case of a pure IP network (without MPLS functionality), the situation is simple. Based on the OSPF protocol, all traffic is routed along its shortest path from source to destination. Hence, a simple shortest path algorithm is sufficient to decide upon the routing. A clear disadvantage of this routing mechanism is that it leaves no opportunity to avoid traffic congestion by routing some traffic along alternative paths.

In a IP network with MPLS functionality, it is possible to set-up explicit routes which may be different from the shortest path. This allows for traffic engineering: one can exploit the resources of the whole network to spread the traffic as good as possible to avoid traffic congestion. If we want the optimal routing of different traffic flows (coming from different source routers, going to different destination routers), we are now facing a multi-commodity flow problem ! Since the size of an IP packet (typically a few tens to a few thousands of bytes) is very small in comparison with the capacity of an IP link, we can choose the flow on an IP link almost continuously. Hence, assuming that the flow variables are real numbers is a decent approximation.

Limitations of graph algorithms

Example of multi-commodity flow problems:
- distinction between real and integer problem arises
- no efficient graph algorithms as solution method

More in general:
- graph algorithms for 'easy' problems
- or parts of 'difficult' problems (e.g. network design)

⇒ next chapter:
- general optimisation techniques
- real vs. integer

The extension towards multi-commodity flow problems clearly indicates some limitations of the graph algorithms described in the previous sections. For many multi-commodity flow problems, no efficient graph algorithms have been found (and probably don't exist). The essential difference between single- and multi-commodity flow problems is also apparent from the distinction that appeared between real-valued and integer-valued problems for multi-commodity problems, a distinction which was not present when considering single-commodity problems.

The algorithms in the sections 2 to 5 are algorithms that allow us to solve a number of basic and ever-recurring network problems quite efficiently. For more complex problems (like many of the problems encountered in network design, see also chapter 1), these algorithms will not be sufficient (although in many cases they will still be very useful to solve smaller subproblems of these more complex problems). Therefore, in chapter 3 some general optimisation techniques will be introduced, allowing us to fill the gap that is left by the graph algorithms of sections 2 to 5. The distinction between real-valued and integer-valued problems will turn up there as well.

References

[1] N. Christofedes: Graph Theory: an Algorithmic Approach (Academic Press, New York, 1975).

[2] G. Chartrand, O. Oellermann: Applied and Algorithmic Graph Theory (McGraw-Hill, New York, 1993).

[3] R.K.Ahuja, T.L.Magnanti, J.B.Orlin, Network Flows (Prentice-Hall, New Jersey, 1993).

[4] M.O.Ball, T.L.Magnanti, C.L.Monma, G.L.Nemhauser, Handbooks in Operations Research and Management Science, vol. 7, Network Models (North-Holland, Amsterdam, 1995).

For further information on graph algorithms and networking applications, we refer the interested reader to the literature mentioned above. [1] and [2] are two standard works giving a broad overview of fundamental graph theorems and graph algorithms. A comprehensive overview of a large number of graph algorithms for network flow modelling is given in [3].