Technische Universität Ilmenau 27. Februar 2024

Vorlesung

Netzalgorithmen

Prof. Dr.-Ing Günter Schäfer

NEXTEX

Zusammenfassung von
ADRIAN SCHOLLMEYER

Inhaltsverzeichnis

1. Introduction

1.1. Basic Types of Transmissions

Web
- Bunch of data to be transmitted
- No guaranteed arrival times
- Simplest Case: Server and Client are directly connected by a cable

Telephony
- Continuous flow of information
- Information must arrive in time
- Simplest Case: Two telephones are directly connected via a cable

1.2. Structuring a Network

As pairwise connection of all entities with each other (thus building a complete graph of all entities) does not work, other structures need to be established. We distinguish between *end systems* (user devices) and *switching elements* (switches, routers, etc.).

End systems connect to some form of uplink to access/provide information on the network. They do not forward requests of other systems.

Switching elements Forward incoming packets onto the next hop towards its destination. The "best" next hop is decided by various routing/forwarding tables and algorithms.

1.3. Routing Algorithms

Routers execute *routing* algorithms to decide which output line an in coming packet should be transmitted on.

Connection-oriented services Run the routing algorithm during connection setup and only once to find one path to forward the packets along for the whole lifetime of the connection.

Connectionless services Run the routing algorithm either for each packet or periodically, updating the router's forwarding table in the process.

Routing algorithms can take a *metric* into account that assigns costs to network links and allows administrators to influence routing decisions. Some possible metrics are:

- Financial cost for sending a packet over a link (e. g. when the link is charged per unit of data transferred).
- Delay (useful to penalize using a link with high delay when trying to prefer links with low delay)
- Number of hops (commonly used in most routing algorithms deployed on the Internet, aims to reduce the number of routers/networks to traverse to reach a destination)

The cheapest path is also commonly referred to as the *shortest path*.

Basic types of routing algorithms:

Non-adaptive routing algorithms do not base routing decisions on the current state of the network.

Adaptive routing algorithms take into account the current network state (e. g. distance vector routing, link state routing).

1.4. Flooding

Flooding is a simple strategy, sending every incoming packet to every outgoing link except the one it arrived on. This leads to many duplicated packets in the network, but leads to the packet almost certainly arriving at the destination (the only exception being broken links partitioning the network into two parts).

To reduce the number of duplicated packets, strategies can be used:

Solution 1: Hop counting Have a hop counter in the packet header, which is decremented by each router. If the packet stays in the network for too long, the hop counter goes to 0 and the packet is dropped. Ideally, the hop counter should be initialized to the length of the shortest path from the source to the destination.

Solution 2: Sequence numbers Each router maintains a sequence number and a table of sequence numbers it has seen from other routers. The first-hop router increments and adds its sequence number to each incoming packet from a host. Each router only forwards incoming packets, if it hasn't seen this sequence number from the first-hop router, yet. Thus, packets that have already been seen are discarded.

1.5. Adaptive Routing Algorithms

Non-adaptive routing algorithms pose problems:

- Non-adaptive routing algorithms can't cope with dramatic changes in traffic levels in different parts of the network.
- Non-adaptive routing algorithsm are usually based on average traffic conditions, but lots of computer traffic us extremely *bursty* (i. e. very variable in intensity).

Thus, adaptive routing algorithms are commonly used to make routing decisions.

Three types can be distinguished:

Centralized adaptive routing Has only one central routing controller making routing decisions.

Isolated adaptive routing Is based on information local to each router. No exchange of information between routers is required.

Distributed adaptive routing Uses periodic exchanges of information between routers to compute and update routing information to be stored in the local forwarding table.

1.5.1. Centralized Adaptive Routing

At the heart of Centralized Adaptive Routing is a central routing controller, which

- periodically collects link state information from routers
- calculates routing tables for each router
- dispatches updated routing tables to each router

The centralized approach is severely limited by the routing controller. If it goes down, the routing becomes non-adaptive, making the network vulnerable to outages. Furthermore, the controller needs to handle a great deal of routing information, making it not only a single point of failure, but also a bottleneck for scalability and performance of the network.

1.5.2. Isolated Adaptive Routing

The basic idea is to make routing decisions solely based on information available locally in each router, e. g.:

- *Hot potato*
- *Backward learning*

Hot potato routing:

- Forward the incoming packet to the output link with the shortest queue
- Do not care where the selected output link leads
- Not very effective

Backward learning:

- Maintain a local forwarding table with next hop, hop count and output link

- Incoming packets update the fowarding table entry of the sender if their hop count is better than the entry's current hop count
- Forward packets based on the forwarding table, random route (hot potato, flood) the packet if no entry for the destination exists.
- Remove/forget stale entries in the forwarding table to account for deterioration of routes (e. g. in case of link failures)

Ethernet switches commonly use backward learning to maintain forwarding tables for MAC addresses, usually falling back to flooding packets if no entry for the destination MAC exists in their local forwarding table.

1.5.3. Distributed Adaptive Routing

The central goal is to determine a "good" path (i. e. a sequence of routers) through the network from source to destination. For calculations, the network is abstracted into a graph consisting of:

Routers represented by nodes

Links represented by edges

Costs assigned to edges, representing link costs

Routing algorithms can be classified in several ways:

- Global or decentralized information

  Decentralized All routers know only a portion of the network, i. e. their physically connected neighbors and link costs to their neighbors. By exchanging information with neighbors, routes to other destinations can be calculated. Examples: BGP (path vector), RIP (distance vector)

  Global By exchanging information, routers gain knowledge of the full network topology and the cost of each link. This is used to compute cheap routes to each destination in the network. Examples: OSPF, IS-IS

- Static or dynamic routes

  Static Routes change only slowly over time, e. g. if they are statically configured by network administrator.

  Dynamic Routes change more quickly in response to link cost changes and require periodic updates of routing information.

1.5.4. Graph Model for Routing Algorithms

- $V = \{v_1, v_2, \ldots, v_n\}$ the set of nodes (routers)
- $E = \{e_1, e_2, \ldots, e_m\} \subseteq V^2$ the set of edges (links)
- $c : V \times V \to \mathbb{Z}_{>0}$ cost of an edge
- $s \in V$ start node (i. e. the node for which the shortest path shall be found)
- $d[i]$ cost from s to node v_i
- $p[i]$ index j of the predecessor v_j of v_i on the shortest path from s to v_i.
- $\delta(s, v)$ cost of the shortest path from s to v

1.5.5. Dijkstra's Algorithm for Shortest Paths

Maintain a set N (initially empty) of nodes for which shortest have been found. For each node i that is not directly connected to s, set $d[i] = \infty$, otherwise $d[i] = c(s, v_i)$. Starting from s, take $v_i \in V \setminus N$ with the lowest $d[i]$. The path from v_i's predecessor to v_i is the shortest path. Update $d[j]$ for neighbors $v_j \in V \setminus N$ of v_i, wherever $d[i] + c(v_i, v_j) < d[j]$ (i. e. the path from s to v_j via v_i is shorter than the previously known path from s to v_j). Set $N := N \cup \{v_i\}$. This results in finding the shortest paths from s to each node in the network.

Complexity:

- $\mathcal{O}(|V|^2)$
- Optimal in dense graphs with $|E| \approx |V|^2$
- Efficient implementations with $\mathcal{O}(|V| \cdot \log |V| + |E|)$ are possible when using Fibonacci-Heaps.

1.5.6. Distance Vector Routing

For distance vector routing, each node has its own table for $D^X(Y, Z)$, listing the cost from X to Y via Z as next hop. Distance vector routing has a few favorable properties, useful in large networks like the Internet:

Iterative The algorithm works iteratively, until no nodes exchange information (i. e. no updated information is transmitted through the network)

Asynchronous Information does not need to be exchanged in lock step. Instead any node can send updated information at any time.

Distributed Each node only needs to communicate with its directly attached neighbors.

Initially, all routers only know the costs to their neighbors and set the cost to any other destination to ∞ (aka. unreachable). Afterwards they notify neighbors of their costs to each destination they know of. Then, distance vector routing algorithms continuously wait for changes in local link costs or link update messages from neighbors. Whenever such a change occurs, the information is used to update the local distance table. If any changes to shortest (!) paths have occured, neighbors are notified of the updated shortest path costs.

The main problem of distance vector routing is the *count to infinity* problem. If the link cost for a link suddenly increases (possibly to ∞), the network might now have a shortest path to a destination going in a circle for some time, while the change in link cost is continuously increased in the network until the new cost for the link is reached or an alternative, shorter path is found. This increases the time to find a new shortest path dramatically! This issue can be mitigated with *poisoned reverse*, i. e. when Z routes to X via Z and Y notifies Z that the cost to X changed, Z notifies Y that its cost to X is ∞ to prevent Y from routing to X via Z (as Z would want to route to X via Y, making the packets go in a loop).

1.5.7. The Bellman-Ford Algorithm

The *Bellman-Ford algorithm* is capable of solving the problem of computing shortest path in graphs with edges with negative costs and is the basis for distance-vector routing. Its only limitation is that there must be no cycles with negative total cost, as otherwise the shortest path's cost will go to $-\infty$ (as continuously traversing such a cycle will inifinitely reduce the total cost). The algorithm can detect if such cycles with negative total cost exist.

The algorithm iteratively improves the estimated cost to reach a node by iterating $|V|-1$ times over all edges and check if the current estimate of the node can be improved by taking any of the connected edges, given the current estimate cost. As this algorithm always checks all edges, it has a higher running time of $\mathcal{O}(|V| \cdot |E|)$.

Negative cost cycles can be detected by running the algorithm $|V|$ times. If the cost to any node has changed in the $|V|$th iteration, there must be a negative cost cycle.

1.5.8. Comparison of Link State and Distance Vector Algorithms

Message complexity How many messages are exchanged?

- Link State: with n nodes, E links, $\mathcal{O}(n \cdot E)$ messages are sent by each node
- Distance Vector: exchange only between neighbors, but variable number of messages and variable convergence time

Speed of Convergence How long does it take until the routing table doesn't change after a link state change has occured?

- Link State: $\mathcal{O}(n^2)$ algorithm requires $\mathcal{O}(n \cdot E)$ messages
- Distance Vector: variable convergence time, in part caused by routing loops and the count-to-infinity problem

Robustness What happens if a router malfunctions?

- Link State:
  - Node can advertise incorrect link cost
  - Invalid routing table calculations only affect the malfunctioning router
- Distance Vector:
  - Nodes can advertise incorrect path costs
  - Each node's table is used (in part) by other routers, so errors can propagate through the network

1.6. Hierarchical Routing and Interconnected Networks

So far, an idealized scenario with identical routers and a “flat” network was assumed. In practice, networks scale to hundreds millions of destinations, making storing detailed routing tables for all destinations in the whole network technically impossible, due to memory limitations in routers and link overloads caused by routing table exchanges between routers. Furthermore, administrative autonomy in parts of the network (like in the Internet's autonomous systems) should allow for network administrators to control the routing (especially the routing protocol in use) in their network, independent of the rest of the network.

In *interconnected networks*, data transmission usually involves multiple networks. Routing can be distinguished into two levels:

Intradomain routing inside autonomouse systems.

Interdomain routing between autonomous systems

In the Internet, interdomain routing is done using the Border Gateway Protocol (BGP), which operates on the AS level and considers every AS as one hop. For intradomain routing, each network administrator can choose their AS's interior routing protocol (e. g. OSPF, RIP, iBGP, IS-IS).

For sending traffic between ASes, *Internet Service Providers* (ISPs) have peering agreements and connections with and to each other, making a data transfer possible. Depending on the policies and available links, traffic may not be able to be sent directly from the source ISP to the destination ISP, but needs to be sent to a different transport provider network first (transit).

Each network operator has to make decisions regarding how to handle the traffic in their network, including

- allocating enough capacity of routers and links,
- choosing a routing algorithm,
- setting link costs.

This requires estimation of *traffic demand* in the network. This can be displayed as *demand volume matrix* $H : \{1, \ldots, n\}^2 \to \mathbb{N}$, denoting the traffic demand volume between nodes v_i and v_j as $H[i, j]$, also abbreviated h_{ij} later on.

1.7. Considerations on Traffic Demand and Link Utilization

To understand constraints on maximum link utilization, a few basic facts about the natur of Internet traffic need to be recapitulated:

- Packets are delayed in every router due to store-and-forward processing and queueing.
- Traffic congestion can occur in parts of the Internet.
- Packets may be dropped if arriving at a router with full output queues.

The task of a network designer is to design the network such that delay, congestion and the probability of packet drops are minimized, while also allowing for a reasonable utilization of the network. This is complicated by the fact that traffic arrival patterns and packet sizes in the Internet are random. In order to characterize Internet traffic behavior, large scale measurements are needed to gain insights about traffic arrival distribution and packet size distribution.

Important observation: Internet traffic does not follow commonly known distributions like normal or exponential distributions, but shows self-similar characteristics and can have *heavy-tailed* distributions, i. e. distributions with high skewness.

For simplicity, a few assumptions are made:

- Packets arrive according to a Poisson process with rate λ:

  $$P_n(t) = \frac{(\lambda t)^n}{n!} e^{-\lambda t} \tag{1.1}$$

  (on average, one arrival in every time interval of length $\frac{1}{\lambda}$).
- Packet size is exponentially distributed, leading to exponentially distributed service times with rate μ

Considering only one router, such a system can be thought of as an M/M/1 queueing system.

1.8. The Poisson Process

Let $A(t)$ $(t \geq 0)$ be the number of packets arriving in $(0, t]$. Requirements:

- $A(0) = 0$

- Idependence of teh number of arrivals in disjoint time periods
- Singularity of arrival events (packets never arrive in parallel)
- Stationary process of arrivals: the probability how many arrivals happen in a time interval only depends on the interval length.

Denote the probablity that n packets arrive in $(0, t]$ as follows:

$$P_n(t) := \Pr[A(t) = n] \tag{1.2}$$

Due to the aformentioned requirements, we have

$$P_0(0) = 1 \qquad \forall n > 0 : P_n(0) = 0 \tag{1.3}$$

After a bunch of math that no one in their right mind can memorize, we obtain:

$$P_n(t) = \frac{(\lambda t)^n}{n!} e^{-\lambda t} \tag{1.4}$$

1.9. Little's Law

Let $\text{Arrival}(T)$ be the number of packets arrived until time T, $W_i(T)$ be the waiting time of packet i at time T, $N(T)$ be the number of packets in the system at time T.

We are interested in the accumulated (total) waiting time of all jobs that ever arrived in the system until time T. This can be computed either as the sum of waiting times of all packets arrived until time T or as the integral over the number of packets in the system during $(0, T]$. Both methods lead to the same result.

Now, let $\lambda(T)$ be the average number of packets in $(0, T]$, $\overline{W}(T)$ be the average waiting time of a packet and $\overline{N}(T)$ be the average number of packets in the system. Then we get *Little's Law*:

$$\lambda \cdot \overline{W} = \overline{N} \tag{1.5}$$

1.10. M/M/1 System

The number of packets in the system (queue size) at discrete points in time δ can be described as a *Markov chain*, with the probability of the queue size increasing being $\lambda\delta$ and the probability of the queue size decreasing being $\mu\delta$. Let p_n denote the probability of the system having queue size n. We obtain

$$p_n = (1 - \rho)\rho^n \qquad \rho = \frac{\lambda}{\mu} \tag{1.6}$$

$$\overline{N} = \frac{\lambda}{\mu - \lambda} \tag{1.7}$$

$$\overline{W} = \frac{1}{\mu - \lambda} \tag{1.8}$$

Thus, with $\rho \to 1$ (i. e. the system load is so high that on average each packet takes as long to process as new packets arrive on average), the average waiting time and queue size go to infinity. Since in reality the queue size is limited, this will lead to packets being dropped.

If packets have average size K_p bits and link capacity is C bits per second, then the average service rate of the link is

$$\mu_p = \frac{C}{K_p} \text{ pps (packets per second)} \tag{1.9}$$

If the average arrival rate is λ_p pps, then the average delay is given by

$$D(\lambda_p, \mu_p) = \frac{1}{\mu_p - \lambda_p} \tag{1.10}$$

This leads to an important insight: To maintain low delays, link utilization should be kept low, e. g. below 50 %. Thus, when link utilization reaches a certain threshold, it should be upgraded. From a delay perspective, it's better to have one high bandwidth link than multiple lower bandwidth links. This is often referred to as the *statistical multiplexing gain*.

Contrary to this, fault tolerance may call for having multiple links. Also, on a single link, misbehaving traffic flows are difficult to control.

1.11. Notion of Routing and Flows

Routing can not only be interpreted as the decision how an individual packet may be transported in the networks, but also how ensemble traffic may be routed between the same two points (e. g. points of presence, data centers). For the remainder of the lecture, this second notion is used and instead of making routing decisions for individual packets, decisions for whole flows are made. Routing decisions then need to stay withing capacity constraints or can influence capacity decisions.

1.12. Multi-Level Networks

When doing interconnects over transit providers, the network architecture can be viewed both in the *transport view* (i. e. the (underlay) network of the transport provider) as well as the *traffic view* (i. e. the flow of traffic between nodes in the network).

Links in the traffic network are *logical links* and must be mapped to links/paths in the transport network. The mapping can change the properties of the network from one view to the other. There can be logical links between nodes in the traffic view for nodes that are not physically connected in the transport view, e. g. if traffic between these nodes needs to be transported over a few switches.

2. Modeling Network Design Problems

2.1. Link-Path Formulation

- $\hat{h}_{12} = 5$... undirected demand between node 1 and 2 is 5, also noted as $\langle 1, 2 \rangle$
- $\hat{x}_{132}$... amount of flow over path 1, 3, 2
- $\hat{c}_{12}$... capacity of link 1–2
- a^*... optimal solution for variable a (e. g. $\hat{x}^*_{132}$)

These can be combined to obtain systems of equations, which usually have multiple solutions. The answer to the question, which of these solutions is of best interest, depends on the goal of network design, e. g.:

- Minimize the total routing cost (if links are annotated with a link cost)
- Minimize congestion of the most congested link

If the objective is to minimize the total routing cost and the cost of routing one unit of traffic over one link is set to 1 for all links (i. e. the goal is to minimize the number of hops for each route), an objective function might be:

$$\mathbf{F} = \hat{x}_{12} + 2\hat{x}_{132} + \hat{x}_{13} + 2\hat{x}_{123} + \hat{x}_{23} + 2\hat{x}_{213} \tag{2.1}$$

Note that flows routed over two links are weighted with factor 2. Such an optimization task is called a *multi-commodity flow problem.* The inverse objective (try to avoid direct links) can also occur, e. g. in air travel networks.

Link-path formulation is one of multiple ways to describe network optimization problems. It is appropriate for networks with undirected links as well as with directed links.

2.2. Node-Link Formulation

In this scenario, links and demands are assumed to be directed, so a link 1–2 is substituted with two directed links ("*arcs*") $1 \to 2$ and $2 \to 1$. Instead of tracing all path flows realising the demand, the total link flow for the demand on each link is considered. Undirected demands $\langle 1, 2 \rangle$ are replaced with directed demands $\langle 1 : 2 \rangle$.

Looking from the point of view of a fixed node that is not end point of the flow, there are flows coming in and going out of that node. The total incoming flow must then be equal to the total outgoing flow. The demand of source nodes is the total outgoing flow minus the total incoming flow, while for sink nodes the demand is the total incoming flow minus the total outgoing flow.

This gives the following notation:

- $\tilde{x}_{13,12}$... flow over arc $1 \to 3$ for demand $\langle 1 : 2 \rangle$

For each demand and each node, all possible direction of paths (including backflows) are part of the total flow equation, although backflows can be safely set to 0, as they make no sense from a practical viewpoint. Additionally, the source node includes a $-\hat{h}$ term and the sink node includes a $\hat{h}$ term to account for the imbalance in incoming/outgoing flow caused by the demand (account for the conservation of flow).

A simple example for a demand $\langle 1 : 2 \rangle$ with nodes $1, 2, 3$ might be:

$$\tilde{x}_{12,12} + \tilde{x}_{13,12} = \hat{h}_{12} \tag{2.2}$$

$$-\tilde{x}_{13,12} + \tilde{x}_{32,12} = 0 \tag{2.3}$$

$$-\tilde{x}_{12,12} - \tilde{x}_{32,12} = -\hat{h}_{12} \tag{2.4}$$

Capacity constraints are modeled as before:

$$\tilde{x}_{12,12} + \tilde{x}_{12,13} \leq \hat{c}_{12} \tag{2.5}$$

Note that the two notations shown until now can become very cumbersome for larger networks:

- There might be no demand between some or many pairs of nodes
- There are no links between most pairs of nodes

Still, the two formulations would require inclusion of these cases, although they are not relevant to the solution at all.

2.3. Link-Demand-Path-Identifier-Based Notation

This formulation assignes indices to demands and capacities, yielding a simpler notation:

- h_i... the demand with index i
- c_j... the (known) capacity of the jth link
- y_j... the unknown capacity of the jth link (dimensioning problem)
- x_{ij}... the flow of demand i over link j
- $v_1 - v_2 - \ldots - v_{n+1}$... undirected path in node representation
- $v_1 \to v_2 \to \ldots \to v_{n+1}$... directed path in node representation
- $\{e_1, e_2, \ldots, e_n\}$... undirected path in link representation

- $(e_1, e_2, \ldots, e_n)$... directed path in link representation
- ξ_i... cost of link e_i
- P_{ij}... candidate path j for demand i
- $\delta_{edp} : e \times P_{ij} \to \{0, 1\}$... is link e_k on candidate path P_{ij}, *link-path incidence relation*

The vector of all flows is called the *flow allocation vector* or *flow vector*:

$$\mathbf{x} = (\mathbf{x}_1, \mathbf{x_2}, \ldots, \mathbf{x}_D) \quad (2.6)$$
$$= (x_{11}, x_{12}, \ldots, x_{1P_1}, \ldots, x_{D1}, x_{D2}, \ldots, x_{DP_D}) \quad (2.7)$$
$$= (x_{dp} \mid d = 1, 2, \ldots, D; p = 1, 2, \ldots, P_d) \quad (2.8)$$

The table for the link-path incidence relation δ_{edp} contains a 1 whenever a link e is used for satisfying a demand d over a path p, and 0 if the link is not used in that path. Note that δ_{edp} is not a variable, but fixed!

This us a notation for the *load* $\underline{y}_e$ of link e and capacity constraints:

$$\underline{y}_e = \sum_{d=1}^{D} \sum_{p=1}^{P_d} \delta_{edp} x_{dp} \quad (2.9)$$

$$\forall e \in \{1, 2, \ldots, E\} : \sum_{d=1}^{D} \sum_{p=1}^{P_d} \delta_{edp} x_{dp} \le y_e \quad (2.10)$$

The general formulation of the simple dimensioning problem is:

$$\min \mathbf{F} = \sum_{e=1}^{E} \xi_e y_e \quad (2.11)$$

subject to

$$\forall d \in \{1, \ldots, D\} : \sum_{p=1}^{P_d} x_{dp} = h_d \qquad \text{demand} \quad (2.12)$$

$$\forall e \in \{1, \ldots, E\} : \sum_{d=1}^{D} \sum_{p=1}^{P_d} \delta_{edp} x_{dp} \le y_e \qquad \text{capacity} \quad (2.13)$$

$$\mathbf{x} \ge 0 \qquad \text{variables} \quad (2.14)$$

$$\mathbf{y} \ge 0 \quad (2.15)$$

Depending on whether link capacites are known (fixed) or unknown (to be chosen), c_i and y_i are used, respectively. Problems with unknown link capacities are referred to as *dimensioning problems* or *uncapacitated problems*, contrary to *capacitated problems* where link capacities are known. When variables can take continuous values, then for any optimal solution, the capacity constraints become equalities, as otherwise cost would arise for unused capacity (which is never optimal).

The *cost* of a path P_{dp} is given by:

$$\zeta_{dp} = \sum_{e=1}^{E} \delta_{edp}\xi_e \qquad d \in \{1, \ldots, D\}, p \in \{1, \ldots, P_d\} \tag{2.16}$$

Shortest-Path Allocation Rule for Dimensioning Problems: For each demand, allocate its entire demand to its shortest path with respect to link costs and candidate paths. If there is more than one shortest path for a given demand, then the demand volume can be arbitrarily split among shortest paths.

This rule works for simple dimensioning problems, but might not work if further constraints are to be taken into account. Further constraints might very well be imposed on the problem:

- *Non-bifurcated flows* require each demand to be satisfied by exactly one path flow.
- To ensure graceful degradation in case of node or link failures, flows might need to be partitioned among several node-disjoint paths.

Depending on the demands and capacities, non-bifurcated solutions might not even be possible, although bifurcated solutions exist.

2.4. Capacitated Problems

In some cases, link capacities are given and the task is to find a solution that satisfies the specified demands, while staying within the capacity bounds. Such problems can be formulated in the following general notation:

$$\forall d \in \{1, \ldots, D\} : \sum_{p=1}^{P_d} x_{dp} = h_d \qquad \text{demand constraints} \tag{2.17}$$

$$\forall e \in \{1, \ldots, E\} : \sum_{d=1}^{D} \sum_{p=1}^{P_d} \delta_{edp} x_{dp} \leq c_e \qquad \text{capacity constraints} \tag{2.18}$$

$$\mathbf{x} \geq 0 \qquad \text{constraints on variables} \tag{2.19}$$

Sometimes there might be no objective function, rendering any feasible solution acceptable. If flow routing cost is to be minimized, these problems are similar to the first problem.

2.5. Shortest-Path Routing

Shortest-Path routing is commonly used in networks. Thus, the network design needs to anticipate that demands will only be routed on their shortest paths. The *length* of the path is determined by adding up link costs w_e according to some weight system $\mathbf{w}$.

Setting the link capacities to be equal to the computed link loads of the shortest-path routing solution gives a (trivially) feasible solution. In general, however, the objective is to find a solution that allows to respect the given link capacities and instead looks for the appropriate weight system.

Single Shortest Path Allocation Problem For given link capacities $\mathbf{c}$ and demand volumes $\mathbf{h}$, find a link weight system $\mathbf{w}$ such that the resulting shortest paths are unique and the resulting flow allocation vector is feasible.

This problem is usually complex, as non-bifurcated solutions may not exist even though bifurcated solutions do, non-bifurcated solutions (if they exist) are usually hard to determine and a weight system inducing an existing single-path flow solution might be impossible to find.

If there are multiple shortest paths, one might be interested in splitting demand volumes amoung multiple shortest paths. Such a rule which is used in OSPF routing is the *equal-cost multi-path* (ECMP) rule, aiminng to equally split the outgoing demand volume over all outgoing next hops with equal cost for a fixed destination. However, such a simple might fail if link weights are not set appropriately.

2.6. Fair Networks

In the Internet, demands are often not fixed but *elastic*, meaning that each demand can consume any bandwidth assigned to its path. In such a case, capacity constraints are given, for the demands h_d no particular values are assumed.

An obviously initial solution is to assign each demand volume on its lower bound. If this does not satisfy the capacity constraints, there is no feasible solution at all. If, however, feasibility is assured, being able to carry more than the minimum required bandwidth while at the same time giving a fair share of bandwidth to all flows might be desired.

The best-known general fairness criterion is *Max-Min-Fairness* (MMF), also called *equity*:

- If no lower bounds are specified, assign the same maximal value to all demands.
- If there is still capacity left, assign the same maximal value to all demands that can still make use of that capacity.

One might be interested in a compromise between MMF and greedily maximizing network throughput. One such fair allocation principle is *Proportional Fairness* (PF) and is realized by maximizing a logarithmic revenue function:

$$\forall d \in \{1, \dots, D\} \, \forall p \in \{1, \dots, P_d\} : \mathbf{R}(x) = \sum_d r_d \ln \left(\sum_p x_{dp} \right) \tag{2.20}$$

with r_d being the revenue associated with demand d. If all demands are of equal importance, then $r_d = 1$ for all demands d. This function is no longer linear. However, it ensures that no demand is allocated an overall path flow sum of 0 and makes assigning (unfairly) high values “unattractive”. By introducing a linear approximation of **R**, the PF problem can be solved, as is shown later.

Solving the MMF capacitated problem is more complicated, since in general it is not enough to find a flow allocation vector that maximizes the minimal flow X_d over all demands d. Even if such a flow vector X is found, then in general some link capacities might still be free and can be used to increase flow allocations for at least a subset of demands.

2.7. Topological Design

When installing a link in a network, there is usually a fixed cost that is independent of the capacity of the link (e. g. cabling cost). In order to account for this, such an *opening cost* κ_e needs to be modeled in the objective function (which is to be minimized):

$$\mathbf{F} = \sum_e \xi_e y_e + \sum_e \kappa_e u_e \tag{2.21}$$

where u_e is a binary variable indicating whether link e is installed or not.

To force the capacity y_e to be 0 whenever the link e is not installed, a large additional constant Δ together with additional constraints is introduced,

$$\forall e \in \{1, \dots, E\} : y_e \leq \Delta u_e \tag{2.22}$$

2.8. Restoration Design

So far, the network was always considered to be in operational state, without accounting for link or node failures. Now, let’s assume the following failure model:

- Links can be either fully functional or completely failed
- No more than one link fails at a time
- Failure state s ($s \in \{1, \dots, |E|\}$) indicates that s links have failed

To solve the *restoration design problem* (RDP), additional indexes s are introduced to the path flow variables x_{dps}, referring to that particular flow in case of failure state s. This also requires reformulating the capacity constraints, e. g.:

$$s = 0 : \qquad x_{120} + x_{310} \leq y_1 \tag{2.23}$$

$$s = 1 : \qquad x_{121} + x_{311} \leq 0 \tag{2.24}$$

$$s = 2 : \qquad x_{122} + x_{312} \leq y_1 \tag{2.25}$$

Additionally, the notation α_{es} is introduced, indicating whether link e is up or not obtaining the following constraints:

$$\forall s \in \{0, \dots, S\} \, \forall e \in \{1, \dots, E\} : \sum_d \sum_p \delta_{edp} x_{dps} \leq \alpha_{es} y_e \tag{2.26}$$

A robust network can be considerably more expensive than the cheapest network without failure considerations.

2.9. Intra-Domain Traffic Engineering for IP Networks

In this scenario, intra-domain routing is operated by an ISP that has control over the network topology, routing algorithm and link weight system. Due to service level agreements or experience obtained via measurements, the ISP knows the demands between nodes of their network. A common objective of intra-domain routing optimization is to minimize the (average) delay experience by data packets. Thus, the goal is to minimize the maximum utilization over all links.

A commonly used intra-domain routing protocol is OSPF, which is based on Dijkstra's algorithm, which calculates shortest paths based on some weight system $\mathbf{w}$. Thus, the goal is to identify a weight system $\mathbf{w}$ such that the maximum link utilization of the network is minimized while satisfying all given demands and staying within capacity constraints. This results in path flows and total link loads begin defined with respect to $\mathbf{w}$, as these are now induced by the weight system influencing how OSPF distributes traffic:

$$\forall d \in \{1,\dots,D\} : \sum_p x_{dp}(\mathbf{w}) = h_d \tag{2.27}$$

$$\forall e \in \{1,\dots,E\} : \underline{y}_e(\mathbf{w}) = \sum_d \sum_p \delta_{edp} x_{dp}(\mathbf{w}) \le c_e \tag{2.28}$$

The maximum r over all link utilizations can be computed and is needed to ensure that all link loads stay below $c_e r$:

$$r = \max\left\{ \frac{\underline{y}_e(\mathbf{w})}{c_e} \,\middle|\, e = 1,\dots,E \right\} \tag{2.29}$$

$$\forall e \in \{1,\dots,E\} : \underline{y_e}(\mathbf{w}) = \sum_d \sum_p \delta_{edp} x_{dp}(\mathbf{w}) \le c_e r \tag{2.30}$$

This leads to the following optimization problem:

$$\min \mathbf{F} = r \tag{2.31}$$

subject to

$$\forall d \in \{1,\dots,D\} : \sum_{p=1}^{P_d} x_{dp}(\mathbf{w}) = h_d \tag{2.32}$$

$$\forall e \in \{1,\dots,E\} : \underline{y}_e(\mathbf{w}) = \sum_{d=1}^{D} \sum_{p=1}^{P_d} \delta_{edp} x_{dp}(\mathbf{w}) \le c_e r \tag{2.33}$$

$$r = \max\left\{ \frac{\underline{y}_e(\mathbf{w})}{c_e} \,\middle|\, e = 1,\dots,E \right\} \tag{2.34}$$

With r being continuous and w_e being non-negative integers.

For this to work, k shortest paths for every attempted weight system vector $\mathbf{w}$ need to be found. If $r^* < 1$, no link will be overloaded. However, if r^* is close to 1, congestion is likely to occur. For $r^* > 1$ there is at least one overloaded link, i. e. the demands can not be satisifed appropriately.

2.10. Tunnel Optimization for MPLS Networks

Multi-Protocol Label Switching (MPLS) is an approach that introduces virtual connections into packet switched networks in order to speed up processing times for routers and allow for traffic engineering. In order to transport traffic in an MPLS network, a so-called *label switched path* is set up from the source (ingess MPLS node) to the destination (egress MPLS node). Tunneling (by making use of label stacking) can be used to handle "similar" traffic in an aggregated way, allowing for different traffic capabilities like putting traffic of similar QoS classes into the same tunnels for special treatment and easy re-routing in case of congestion or link failures.

In order not to overload routers with too many tunnels, which would increase the processing overhead, it is desirable to limit the number of tunnels per router and/or link. Thus the optimization challenge is to carry different traffic classes in an MPLS network through the creation of tunnels such that the number of tunnels per node/link is minimized and the load is balanced amoung routers/links. For this, the same notation as before can be used, with x_{dp} now denoting the fraction of demand that is routed over path P_{dp}, resulting in the demand constraint:

$$\forall d \in \{1,\dots,D\} : \sum_{p=1}^{P_d} x_{dp} = 1 \tag{2.35}$$

Note that $x_{dp} \in [0,1]$ and the absolute flow transported is now $h_d \cdot x_{dp}$.

To avoid path flows with very low fractions, a lower bound ε is introduced together with binary variables u_{dp} indicating whether the lower bound is satisfied or not:

$$\forall d \in \{1,\dots,D\}\, \forall p \in \{1,\dots,P_d\} : \varepsilon u_{dp} \leq x_{dp} \tag{2.36}$$

$$\forall d \in \{1,\dots,D\}\, \forall p \in \{1,\dots,P_d\} : x_{dp} \leq u_{dp} \tag{2.37}$$

Capacity feasibility constraints:

$$\forall e \in \{1,\dots,E\} : \sum_{d=1}^{D} h_d \sum_{p=1}^{P_d} \delta_{edp} x_{dp} \leq c_e \tag{2.38}$$

The number of tunnels on link e will be:

$$\sum_d \sum_p \delta_{edp} u_{dp} \tag{2.39}$$

The goal is now to minimize the number r representing the maximum number of tunnels over all links:

$$\min \mathbf{F} = r \tag{2.40}$$

subject to

$$\forall d \in \{1, \ldots, D\} : \sum_{p=1}^{P_d} x_{dp} = 1 \tag{2.41}$$

$$\forall e \in \{1, \ldots, E\} : \sum_{d=1}^{D} h_d \sum_{p=1}^{P_d} \delta_{edp} x_{dp} \leq c_e \tag{2.42}$$

$$\forall d \in \{1, \ldots, D\} \, \forall p \in \{1, \ldots, P_d\} : \varepsilon u_{dp} \leq x_{dp} \tag{2.43}$$

$$\forall d \in \{1, \ldots, D\} \, \forall p \in \{1, \ldots, P_d\} : x_{dp} \leq u_{dp} \tag{2.44}$$

$$\forall e \in \{1, \ldots, E\} : \sum_{d=1}^{D} \sum_{p=1}^{P_d} \delta_{edp} u_{dp} \leq r \tag{2.45}$$

and x_{dp} continuous and non-negative, u_{dp} binary and r integer.

This problem has both continuous and binary variables, whilch the constraints and objective function are linear. It is an example for a *mixed-integer linear programming problem* (MIP) . Finding exact solutions for MIP is more difficult than for linear optimization problems, branch-and-bound and banch-and-cut being established techniques to solve such problems.

3. Optimization Methods

3.1. Optimization Problems

An *optimization problem* is given by a set M and a function $f\colon X \to \mathbb{R}$ with $M \subseteq X$. The usual terminology is as follows:

- M is called the *feasible set*
- f is called the *objective function*
- Elements of M are called *feasible solutions*
- $x^* \in M$ is an *optimal solution* if $\forall x \in M : f(x^*) \geq f(x)$, i. e.

$$f(x^*) = \max \{f(x) \mid x \in M\} \tag{3.1}$$

An *optimization method* is an algorithm that computes an optimal solution x^* given the input (M, f), if there is any.

There is no need to deal with minimaztion problems separately as they can be easily converted into a maximization problem.

3.2. Linear Optimization Problems (LOP)

An optimization problem (M, f) is a *linear optimization problem* (LOP) if $M \subseteq \mathbb{R}^n$ for some $n \in \mathbb{N}$ consists of all $\vec{x} \in \mathbb{R}^n$ satisfying a finite set of linear inequalitites and/or linear equations, and $f\colon \mathbb{R}^n \to \mathbb{R}$ is a linear function.

Example:

$$\max f(x, y) = x + 3y \tag{3.2}$$

subject to

$$-x + y \leq 1 \tag{3.3}$$

$$x + y \leq 2 \tag{3.4}$$

$$x, y \geq 0 \tag{3.5}$$

Notation:

- $\mathbb{R}^n$ is the set of all column vectors $\vec{x} = (x_1, \ldots, x_n)^\top$ with $x_1, \ldots, x_n \in \mathbb{R}$
- $\vec{a} \leq \vec{b}$ iff $a_k < b_k$ for all $k \in \{1, \ldots, n\}$

- $\vec{o}$ denotes a zero vector
- I denotes an identity matrix
- $\|\vec{x}\| = \sqrt{x_1^2 + \ldots + x_n^2}$ denotes the euclidian norm of $\vec{x}$

3.2.1. Solving LOPs with Two Variables Graphically

Each inequality defines a boundary line, which can be plotted in a diagram. These together are boundaries to the feasible set. By setting $c = f(\vec{x})$ for some c, a contour line can be drawn, showing where $f(\vec{x}) = c$ holds. By increasing c, this contour line is shifted towards the maximum value, until it reaches a "corner" with the maximum value.

3.2.2. Canonical form LOPs

$$\max f(\vec{x}) = \vec{c}^\top \vec{x} \tag{3.6}$$

subject to

$$A\vec{x} \leq \vec{b} \tag{3.7}$$

$$\vec{x} \geq \vec{o} \tag{3.8}$$

With $A \in \mathbb{R}^{m\times n}$ being a real matrix with m rows and n columns and $\vec{b} \in \mathbb{R}^m$ being a real column vector with m entries.

To transform any LOP into a LOP in canonical form:

- If there is a variable x_k subject to $x_k \leq 0$ (the variable must be non-positive), replace every appearance of x_k with $-x_k$ and replace $x_k \leq 0$ with $x_k \geq 0$.
- If a variable x_k is unrestricted (can be both positive and negative), replace every appearance of x_k by $x_k^+ - x_k^-$ and let $x_k^+ \geq 0$ and $x_k^- \geq 0$.
- Replace $\underline{a}_k\vec{x} \leq b_k$ with $-\underline{a}_k\vec{x} \geq -b_k$ and $\underline{a}_k\vec{x} = b_k$ with $\underline{a}_k\vec{x} \leq b_k$ and $\underline{a}_k\vec{x} \geq b_k$.

3.2.3. Standard Form LOPs

$$\max f(\vec{x}) = \vec{c}^\top \vec{x} \tag{3.9}$$

subject to

$$A\vec{x} = \vec{b} \tag{3.10}$$

$$\vec{x} \geq \vec{o} \tag{3.11}$$

With $A \in \mathbb{R}^{m\times n}$ and $\vec{b} \in \mathbb{R}^m$ with $\vec{b} \geq \vec{o}$.

To transform a LOP into standard form, every occurence of $\leq$ must be replaced with $=$ as well as the introduction of *slack variables* $\vec{y}$, which are simply added to the left-hand side of the equation to allow the actual x_k to be lower than b_k. The resulting equations can be rewritten as

$$(A|I)\begin{pmatrix}\vec{x}\\ \vec{y}\end{pmatrix} = \vec{b} \tag{3.12}$$

If $\vec{b}$ does not satisfy $\vec{b} \geq \vec{o}$ (i. e. if some b_k are negative), multiply these rows by -1.

LOPs in standard form are systems of linear equations $A\vec{x} = \vec{b}$. As such, the number of irredundant equations can be compared to the number of variables to get information about how many solutions exist:

- If there are more irredundant equations than variables, no solution to the system of equation exists.
- If there are as many irredundant equations as variables, there is exactly one solution.
- If there are less irredundant equations as variables, there are infinitely many solutions (and as such, an optimal solution must be determined).

3.3. The Structure of the Feasible Set

A subset $X \subseteq \mathbb{R}^n$ is called:

convex if for any two points $\vec{x}_1, \vec{x}_2 \in X$ the whole straight line segment is in X, i. e. the straight line connecting $\vec{x}_1$ and $\vec{x}_2$ never leaves X

closed if the limit of every convergent sequence in X is in X too

bounded if there is $K \in \mathbb{R}$ such that $\|\vec{x}\| \leq K$ for all $\vec{x} \in X$, i. e. there is an upper limit to the distance of all points in X from $\vec{o}$.

Any feasible set M for a LOP in standard form is convex and closed. If M is also nonempty and bounded, f attains its maximum on M.

3.4. Extreme Points and Basic Solutions

Let M be a convex set. A point $\vec{x}_0 \in M$ is called an *extreme point* of M if $\vec{x}_0$ cannot be expressed as a convex linear combination $\alpha\vec{x}_1 + (1-\alpha)\,\vec{x}_2$ with $\alpha \in [0,1]$ of two points $\vec{x}_1, \vec{x}_2 \in M$ with $\vec{x}_1 \neq \vec{x}_0$ and $\vec{x}_2 \neq \vec{x}_0$.

If the set M is nonempty, then it has at most finitely many and at least one extreme point. If M is in addition bounded, then M is the convex hull of its extreme points $\vec{x}_1, \ldots, \vec{x}_k$, i. e. any point $\vec{x} \in M$ can be expressed as convex linear combination of the extreme points.

If M is nonempty and bounded, then there is an extreme point $\vec{x}^* \in M$ such that $f(\vec{x}^*) \geq f(\vec{x}) \quad \forall \vec{x} \in M$. Thus, one of the extreme points is the optimal solution to the LOP.

If M is nonempty and unbounded, then there need not be an optimal solution. However, if there is one, then the following theorem holds:

If M is nonempty and there exists a K such that $f(\vec{x}) \leq K$ for all $\vec{x} \in M$, then there is an extreme point $\vec{x}^* \in M$ such that $f(\vec{x}^*) \geq f(\vec{x})$ for all $\vec{x} \in M$.

Let $A \in \mathbb{R}^{m\times n}$ be a matrix with column vectors $\vec{a}_1, \ldots, \vec{a}_n$, $M = \left\{ \vec{x} \in \mathbb{R}^n \,\middle|\, \left(A\vec{x} = \vec{b}\right) \wedge (\vec{x} \geq \vec{o}) \right\}$ and $S(\vec{x}) = \{\vec{a}_i \,|\, (i \in \{1, \ldots, n\}) \wedge (x_i \neq 0)\}$ for $\vec{x} \in M$. Then $\vec{x}$ is called a *basic solution* if $S(\vec{x})$ is linear independent.

$\vec{x} \in M$ is an extreme point of M iff $\vec{x}$ is a basic solution.

3.5. The Simplex Algorithm

3.5.1. Overview

Consider the LOP in standard form (L) as follows:

$$\max f(\vec{x}) = \vec{c}^\top \vec{x} \tag{3.13}$$

subject to

$$A\vec{x} = \vec{b} \tag{3.14}$$

$$\vec{x} \geq \vec{o} \tag{3.15}$$

$$A \in \mathbb{R}^{m\times n} \tag{3.16}$$

$$\vec{b} \geq \vec{o} \tag{3.17}$$

$$rk(A) = m \tag{3.18}$$

A few facts are already known:

- M has at least one extreme point
- Any extreme point of M is a basic solution of (L) and vice versa
- (L) has at most $\binom{n}{m}$ basic solutions
- If there is an optimal solution of (L), then there is an optimal solution of (L) that is a basic solution.
- If M is nonempty and bounded, then there is an optimal solution of (L).

Thus, if M is nonempty and bounded, there is a finite algorithm that finds an optimal (basic) solution of (L).

One such algorithm is the Simplex Algorithm. It consists of two parts, called Phase 1 and Phase 2. The input of Phase 2 is a feasible basic solution. The algorithm stops when either an optimal basic solution has been found or if it has been detected that the objective function is unbounded on M. In the latter case, there is no optimal solution of (L).

If no feasible basic solution, Phase 1 must be executed, which applies Phase 2 to an auxiliary LOP. Phase 1 stops when a feasible basic solution of (L) Has been found or if it has been detected that M is empty.

3.5.2. Phase 2

Let $\vec{x}$ be a feasible basic solution of (L). Letz $T(\vec{x}) \subseteq \{\vec{a}_1, \ldots, \vec{a}_n\}$ be a maximal linear independent set of column vectors of A such that $x_k \neq 0 \implies \vec{a}_k \in T(\vec{x})$, i. e. T gives the set of columns of A where the basic solution's value is not 0. It is called a *basis* of the linear subspace generated by all column vectors of A. Thus, each column vector $\vec{a}_j$ can be represented as a linear combination of the elements of $T(\vec{x})$:

$$\vec{a}_j = \sum_{k \in B} t_{k,j} \vec{a}_k \tag{3.19}$$

Let $B = \{k \in \{1, \ldots, n\} \mid \vec{a}_k \in T(\vec{x})\}$ and $N = \{1, \ldots, n\} \setminus B$, i. e. B is the set of column indices where there is corresponding $\vec{a}_k$ to the basic solution and N is the remainder of column indices.

If $j \in B$, then $t_{j,j} = 1$ and $t_{k,j} = 0$ if $k \neq j$. For $j \in N$ let $u_j = \sum_{k \in B} t_{k,j} c_k$ and $d_j = u_j - c_j$. These values can be used to transform $f(\vec{x})$ into $f(\vec{y})$ for an arbitrary feasible solution $\vec{y}$.

Three cases can now be distinguished:

- All offset factors d_i for $i \in N$ are positive. Then $f(\vec{y}) \leq f(\vec{x})$, i. e. the basic solution is better than or equal to $f(\vec{y})$ for all feasible solutions, i. e. the feasible basic solution $\vec{x}$ is also optimal. The algorithm stops.
- For some $j \in N$, all $d_j < 0$ and $t_{k,j} \leq 0$ for all $k \in B$. Then the objective function f is unbounded from above on M, i. e. the objective function can get arbitrarily high. The algorithm stops.
- For some $j \in N$ with $d_j < 0$ there is an index $l \in B$ such that $t_{l,j} > 0$. Then a new feasible basic solution $\vec{z}$ is calculated and the algorithm starts over.

Special care is neeeded to avoid "cycling"; this is done by applying *Bland's rule*, i. e. always take the lowest such j and the lowest such basis index to find a new $\vec{z}$.

3.5.3. Phase 1

If the original LOP is given in standard form (L), and no feasible basic solution is known, an auxiliary LOP can be used to find one:

$$\min g(\vec{x}, \vec{y}) = \sum_{i=1}^{m} y_i \tag{3.20}$$

subject to

$$A\vec{x} + \vec{y} = \vec{b} \tag{3.21}$$

$$\vec{x} \geq \vec{o} \tag{3.22}$$

$$\vec{y} \geq \vec{o} \tag{3.23}$$

A feasible basic solution for the auxiliary problem is $\vec{y} = \vec{b}, \vec{x} = \vec{o}$. Thus, the feasible set of the auxiliary LOP is nonempty.

The auxiliary LOP has an optimal solution $(\vec{y}^*, \vec{x}^*)$ and the original LOP has a feasible solution iff $\vec{y}^* = \vec{o}$. Thus, the objective is to find an optimal solution with $\vec{y} = \vec{o}$.

An optimal solution can be found by applying Phase 2 to the auxiliary LOP. A basic feasible solution for the auxiliary LOP is $\vec{y} = \vec{b}$, $\vec{x} = \vec{o}$.

3.5.4. Simplex Tableaus

The plain algorithm description can be applied to a tabular view of the system of equations. The system of linear equations representing $A\vec{x} = \vec{b}$ can be transformed by the Gauss-Jordan transformation into equivalent systems of equations. The following operations can be performed:

- Switch two equations (rows)
- Multiply one equation (row) with a *nonzero* factor
- Add a multiple of an equation (row) to another equation (row)

The resulting system of equations has exactly the same solutions as the original one.

With these operations, the system of linear equations $A\vec{x} = \vec{b}$ can be transformed into an equivalent system of linear equations $A\vec{x} = \vec{b}$ such that the first m columns form an $m \times m$ identity matrix I.

In simplex tableaus, we extend the system of equations $\hat{A}\vec{x} = \hat{\vec{b}}$ by adding another row $\left(\hat{\vec{c}}^\top \middle| z\right)$ with $\hat{z} = \hat{\vec{c}}^\top \vec{x}$ being the transformed objective function value and

$$\hat{c}_i = \begin{cases} 0 & i \in B \\ c_i - \sum_{k \in B} c_k t_{k,j} & i \in N \end{cases} \tag{3.24}$$

$$\hat{z} = z - \sum_{k \in B} c_k b_k \tag{3.25}$$

This allows us to produce new basic solutions as follows:

- Choose $j \in N$ such that $d_j < 0$.
- Choose $l \in B$ such that $t_{l,j} > 0$ and $\frac{x_l}{t_{l,j}}$ is minimal.
- If multiple such j and l exist, apply Bland's rule.
- Multiply the lth equation by $\frac{1}{t_{l,j}}$ and then, add the resulting lth equation multiplied by $-t_{k,j}$ to the kth equation for $k \in \{1, \dots, m\}$ with $k \neq l$.
- Add the lth equation multiplied by $-\hat{c}_j = -d_j$ to the equation $\vec{c}^\top \vec{x} = z$, giving $\hat{\vec{c}}^\top \vec{x} = \hat{z}$.

This translates into the following steps:

- Find the column j with the highest c_j (= pivot column).
- Divide the values of the last column by the corresponding values from the pivot column. Find the row where the result is the lowest (= pivot row).
- The intersection between the pivot column and pivot row is the pivot element.
- Use Gauss-Jordan transformations to bring all cells of the pivot column except the pivot element and the last row to 0. While doing so, make sure that $\vec{b}$ remains positive!
- Repeat until:
  - All c_j are ≤ 0. An optimal solution has been found. Stop.
  - For some j with $c_j > 0$ all cells a_{ij} above it are $a_{ij} \leq 0$. Then the solution is unbounded.

3.5.5. The Complexity of the Simplex Algorithm

While there are examples where the Simplex Algorithm starting with a specified feasible basic solution will pass through all vertices, and the number of vertices is exponential in the number n of variables. Still, in practice the Simplex Algorithm is very efficient in approach. In many relevant cases, its running time is proportional to $m + n$.

Polynomail time algorithms for LOPs also exist (Khachian, Karmarkar). Furthermore, there are also polynomial time variations of the Simplex Algorithm for special cases like single-commodity network flow problems.

3.6. Duality

To every *primal* problem (i. e. any problem)

$$\max \vec{c}^\top \vec{x} \qquad \text{s. t. } A\vec{x} \leq \vec{b}, \vec{x} \geq \vec{o} \tag{3.26}$$

there is a *dual* problem

$$\min \vec{b}^\top \vec{y} \qquad \text{s. t. } A^\top \vec{y} \geq \vec{c}, \vec{y} \geq \vec{o}. \tag{3.27}$$

Thus, in the dual problem, the number of variables and constraints are switched with each other. If the primal problem had 2 variables and 3 constraints, the dual problem has 3 variables and 2 constraints. The optimal solution for both problems is the same and any feasible solution is a bound for the optimal solution. If the primal optimal solution is unbounded, then the dual problem has no solution. Vice versa, if the primal problem has no solution, the dual problem's optimal solution is unbounded.

If there are feasible solutions of the primal problem and its objective function $\vec{c}^\top \vec{x}$ is bounded from above, then both the primal and dual problem have optimal feasible solutions $\vec{x}^*$ and $\vec{y}^*$ and

$$\vec{c}^\top \vec{x}^* = (\vec{y}^*)^\top \vec{b}. \tag{3.28}$$

If there are feasible solutions of the dual problem and its objective function $\vec{y}^\top \vec{b}$ is bounded from below, both the primal and dual problem have optimal feasible solutions $\vec{x}^*$ and $\vec{y}^*$ and

$$\vec{c}^\top \vec{x}^* = (\vec{y}^*)^\top \vec{b}. \tag{3.29}$$

The primal problem can be transformed into the dual problem as shown in Tabelle 3.1.

Tabelle 3.1.: Transformation between the primal and dual LOP

	Primal LOP	Dual LOP
Variables	$x_1, \dots, x_n$	$y_1, \dots, y_m$
Matrix	A	$A^\top$
Right-hand side	$\vec{b}$	$\vec{c}$
Objective function	$\max \vec{c}^\top \vec{x}$	$\min \vec{b}^\top \vec{y}$
Constraints	ith constraint has $\leq$	$y_i \geq 0$
	ith constraint has $\geq$	$y_i \leq 0$
	ith constraint has $=$	$y_i \in \mathbb{R}$
	$x_j \geq 0$	jth constraint has $\geq$
	$x_j \leq 0$	jth constraint has $\leq$
	$x_j \in \mathbb{R}$	jth constraint has $=$

3.7. Branch and Bound

Consider a (hard to solve) optimization problem

$$\min f(x) \text{s. t. } x \in M. \tag{3.30}$$

Associate a *relaxed* optimization problem

$$\min g(x) \text{s. t. } x \in R \tag{3.31}$$

such that

- $M \subseteq R$,
- if $x \in R$, then $g(x) = f(x)$, and
- The relaxed problem can be solved efficiently.

Then, if y^* is an optimal solution for the relaxed problem, then $f(x) \geq f(y^*)$ for all $x \in M$, i. e. the optimal solution of the relaxed problem is an upper bound for the optimal solution of the hard problem. If $y^* \in M$, then y^* is also an optimal solution to the hard problem.

The idea of *branch and bound* is now to partition the input space M into M_1 and M_2 and solve the optimization problem for both M_1 and M_2 individually (possibly splitting these problems up again), compare both solution and choose the better one. This solution then is the optmial solution for $x \in M$.

When solving for some input space M_i, first solve the relaxed problem g. If it has an optimal solution $y^* \in M_i$, then immediately return this solution as solution for the hard problem. Otherwise, split up M_i into two partitions as describe above and solve those.

If the optimal solution for g in each step turns out ot be worse than the currently known best solution, don't continue on this path at all (as the solution can only get worse than the currently known best solution).

The branch and bound method recursively splits up the input space, and obtains optimal solutions for each part of the partition, then chooses the best one. To reduce calculation effort and eventually obtain a solution, the relaxed problem is solved in each step and check for whether it yields a solution that matches one input from the current input space. Furthermore, if at some point it is clear that the solution for the relaxed problem is worse than the best known solution, the solution for the hard problem can only get worse down this recursion path, so calculation stops there. Eventually, all branches have been either checked or discarded and the optimal solution has been obtained. However, in the worst case, the whole input space has been searched, leading to extremely large run times.

3.7.1. Branch and Bound for Mixed Integer Problems

Consider the MIP

$$\min f(\mathbf{x}) = \mathbf{c}\mathbf{x} \tag{3.32}$$

s. t.

$$\mathbf{A}\mathbf{x} = \mathbf{b} \tag{3.33}$$

$$\mathbf{x} \geq \mathbf{0} \tag{3.34}$$

And x_i is integer for all $j \in I$ where $\mathbf{x} = (x_1, \ldots, x_n)$, $\mathbf{b} \geq \mathbf{0}$, $\mathbf{A} \in \mathbb{R}^{m \times n}$ has rank $m \leq n$ and $I \subseteq \{1, \ldots, n\}$.

Thus, some subset of variables x_i must be integer.

A relaxed problem to this hard problem can be chosen by simply removing the integer constraint (i. e. the x_i that must be integer in the original problem may now be real). This is called *LP-relaxation* and results in the following LOP:

$$\min f(\mathbf{x}) = \mathbf{c}\mathbf{x} \tag{3.35}$$

s. t.

$$\mathbf{A}\mathbf{x} = \mathbf{b} \tag{3.36}$$

$$\mathbf{x} \geq \mathbf{0} \tag{3.37}$$

where $\mathbf{x} = (x_1, \ldots, x_n)$ and $\mathbf{b} \geq \mathbf{0}$, $\mathbf{A} \in \mathbb{R}^{m \times n}$ has rank $m \leq n$.

The relaxed problem can be solved with the Simplex Algorithm, while the MIP must be solved with branch and bound. During the branch step, the input space is split up by choosing some integer variable y_k and its optimal value y_k^* for the relaxed problem. The two sets of possible input values are now:

$$B_1 = \{y \in B \,|\, y_k \leq \lfloor y_k^* \rfloor\} \tag{3.38}$$
$$B_2 = \{y \in B \,|\, y_k \geq \lfloor y_k^* \rfloor\} \tag{3.39}$$

if B is the input space from the "parent" operation.

3.7.2. Cutting planes for MIP

Given a MIP and its LP-relaxation as before, the idea is to find a valid cut $\mathbf{dx} \leq q$ such that

$$\{\mathbf{x} \,|\, \mathbf{Ax} = \mathbf{b}, \mathbf{x} \geq \mathbf{0}, \mathbf{dx} \leq q\} \neq R \text{ and} \tag{3.40}$$
$$\{\mathbf{x} \,|\, \mathbf{Ax} = \mathbf{b}, \mathbf{x} \geq \mathbf{0}, x_j \text{ is integer for all } j \in I, \mathbf{dx} \leq q\} = M \tag{3.41}$$

Such a cut reduces M by the optimal solution of the LP-relaxation. Thus, running the solver for the LP-relaxation again now gives a different optimal solution, which hopefully is closer to the optimal integer solution. This is process is repeated until the optimal solution to the LP-relaxation is integer, in which case it is also an optimal solution to the MIP.

4. Network Design Problems

4.1. Simple Design Problem

- Indices:
  - $d = 1, 2, \ldots, D$... demands
  - $p = 1, 2, \ldots, P_d$... candidate aths for flows realizing demand d
  - $e = 1, 2, \ldots, E$... links
- Constants:
  - δ_{edp}... if link e belongs to path p realizing demand d
  - h_d volume of demand d
  - ξ_e unit (marginal) cost of link e
  - x_{dp} flow allocated to path p of demand d
  - y_e capacity of link e
- Objective: $\min \mathbf{F} = \sum_e \xi_e y_e$ (bandwidth cost)
- Constraints:

$$\sum_{p=1}^{P_d} x_{dp} = h_d, \qquad \forall d \in \{1, \ldots, D\} \text{ (demand constraints)} \tag{4.1}$$

$$\sum_{d=1}^{D} \sum_{p=1}^{P_d} \delta_{edp} x_{dp} \leq y_e \qquad \forall e \in \{1, \ldots, E\} \text{ (capacity constraints)} \tag{4.2}$$

Solutions to this optimization problem will have y_e equal to the load on the links, as otherwise the costs could be lowered (and thus the solution wouldn't be optimal). y_e can be substituted in the cost function with $y_e = \sum_d \sum_p \delta_{edp} x_{dp}$ and $\zeta_{dp} = \sum_e \xi_e \delta_{edp}$ denoting the cost of path p for demand d:

$$\mathbf{F} = \sum_{d=1}^{D} \sum_{p=1}^{P_d} \zeta_{dp} x_{dp} \tag{4.3}$$

This leads to the SDP-Decoupled Link-Path formulation (SDP/DLPF):

- Variables: x_{dp} flow variable allocated to path p of demand d

- Objective: $\min \mathbf{F} = \sum_{d=1}^{D} \sum_{p=1}^{P_d} \zeta_{dp} x_{dp}$
- Constraints: $\forall d \in \{1, \ldots, D\} : \sum_{p=1}^{P_d} x_{dp} = h_d$

This problem formulation has fewer variables! Due to the linear structure of **F**, this is actually a set of decoupled optimization problems, with an optimal solution to these problems allocating all demand on the shortest path. If there are multiple shortest paths, the demand can be arbitrarily split among these paths.

4.2. Capacitated Problems

4.2.1. Pure Allocation Problem

- Constants:
  - δ_{edp} as before
  - h_d as before
  - c_e... capacity of link e
- Variables:
  - x_{dp} as before
- Constraints:

$$\sum_{p=1}^{P_d} x_{dp} = h_d \qquad \forall d \in \{1, \ldots, D\} \tag{4.4}$$

$$\sum_{d=1}^{D} \sum_{p=1}^{P_d} \delta_{edp} x_{dp} \leq c_e \qquad \forall e \in \{1, \ldots, E\} \tag{4.5}$$

This problem has no objective function to optimize, so in this basic form the goal is just to find *any* solution that satisfies the constraints.

Another auxiliary variable can be introduced to define an objective. This yields the PAP Modified Link Path Formulation:

- Variables:
  - x_{dp} as before
  - z... auxiliary continuous variable (of unrestricted sign)
- Objective:

$$\min z \tag{4.6}$$

- Constraints:

$$\sum_{p=1}^{P_d} x_{dp} = h_d \qquad \forall d \in \{1, \dots, D\} \tag{4.7}$$

$$\sum_{d=1}^{D} \sum_{p=1}^{P_d} \delta_{edp} x_{dp} \leq z + c_e \qquad \forall e \in \{1, \dots, E\} \tag{4.8}$$

This problem always has a solution and if $z^* \leq 0$, then x^*_{dp} represent a solution to the original PAP. If PAP is feasible, then a solution $\mathbf{x}$ with at most $D + E$ non-zero flows exists.

In many cases, some objective function to find the best solution shall be found. The above problem maximizes the minimum unused capacity. However, it is also possible to maximize the total unused capacity after flow allocation:

$$\max \mathbf{F} = \sum_{e=1}^{E} r_e \left(c_e - \sum_{d=1}^{D} \sum_{p=1}^{P_d} \delta_{edp} x_{dp} \right) \tag{4.9}$$

$$= \sum_{e=1}^{E} r_e \left(c_e - \underline{y}_e \right) \tag{4.10}$$

4.2.2. Bounded Link Capacities

A variation of a mixed dimensioning/capacitated problem, where link capacities y_e shall be dimensioned with upper bounds c_e can also be considered:

- Objective:

$$\min F = \sum_{e=1}^{E} \xi_e y_e \tag{4.11}$$

- Constraints:

$$\sum_{p=1}^{P_d} x_{dp} = h_d \qquad \forall d \in \{1, \dots, D\} \tag{4.12}$$

$$\sum_{d=1}^{D} \sum_{p=1}^{P_d} \delta_{edp} x_{dp} \leq y_e \qquad \forall e \in \{1, \dots, E\} \tag{4.13}$$

$$y_e \leq c_e \qquad \forall e \in \{1, \dots, E\} \tag{4.14}$$

4.2.3. Path Diversity (PD)

Sometimes, flows shall be allocated in a way that no single path flow x_{dp} carries more than a fraction of the demand (expressed by n_d... the number of different paths among which h_d is to be split).

- Variables:
  - x_{dp} as before
- Constraints:

$$\sum_{p=1}^{P_d} x_{dp} = h_d \qquad \forall d \in \{1, \ldots, D\} \tag{4.15}$$

$$\sum_{d=1}^{D} \sum_{p=1}^{P_d} \delta_{edp} x_{dp} \leq c_e \qquad \forall e \in \{1, \ldots, E\} \tag{4.16}$$

$$x_{dp} \leq \frac{h_d}{n_d} \qquad \forall d \in \{1, \ldots, D\}, p \in \{1, \ldots, P_d\} \tag{4.17}$$

If n_d is an integer, it will force the demand d to be split amon onto at least n_d different paths. If all P_d candidate paths for a demand d are link disjoint, this can guarantee that a single link failure leads to demand d loosing at most $\frac{100\,\%}{n_d}$ of its volume.

4.2.4. Generalized Diversity (GD)

The diversity constraint from PD can also be formulated in a stricter way, allowing to pass arbitrary candidate path lists[1]:

- Constraints:

$$\sum_{p=1}^{P_d} x_{dp} = h_d \qquad \forall d \in \{1, \ldots, D\} \tag{4.18}$$

$$\sum_{d=1}^{D} \sum_{p=1}^{P_d} \delta_{edp} x_{dp} \leq c_e \qquad \forall e \in \{1, \ldots, E\} \tag{4.19}$$

$$\sum_{p=1}^{P_d} \delta_{edp} x_{dp} \leq \frac{h_d}{n_d} \qquad \forall d \in \{1, \ldots, D\}, e \in \{1, \ldots, E\} \tag{4.20}$$

The modified constraint ensures that no link e of path P_{dp} carries more than $\frac{100\,\%}{n_d}$ of demand d. As multiple candidate paths for one demand may use one specific link, the sum over all candidate paths must be calculated. A major drawback of this problem is the high number of constraints.

The shortest path allocation rule from the original problem can be used in a modified version:

- First, look up the shortest path for a demand an allocate $\frac{h_d}{n_d}$ to it.
- Then, allocate the next fraction $\frac{h_d}{n_d}$ the next shortest path and so on.

[1]Note the inclusion of δ_{edp} in the last constraint set here.

4.2.5. Lower Bounded Flows

Sometimes, the flow over a path shall be restricted from below to avoid having a flow to be partitioned among too many paths. This is somehow opposite to path diversity. It requires modeling a lower bound to non-zero flows. thus, new constants and binary variables need to be introduced, leading to the followig problem:

- Constraints:

$$\sum_{p=1}^{P_d} x_{dp} = h_d \qquad \forall d \in \{1,\ldots,D\} \tag{4.21}$$

$$\sum_{d=1}^{D}\sum_{p=1}^{P_d} \delta_{edp} x_{dp} \le c_e \qquad \forall e \in \{1,\ldots,E\} \tag{4.22}$$

$$x_{dp} \le h_d u_{dp} \qquad \forall d \in \{1,\ldots,D\}, p \in \{1,\ldots,P_d\} \tag{4.23}$$

$$b_d u_{dp} \le x_{dp} \qquad \forall d \in \{1,\ldots,D\}, p \in \{1,\ldots,P_d\} \tag{4.24}$$

The binary variables u_{dp} make this problem difficult to solve, as most methods are not significantly different from trying out all combinations.

4.2.6. Limited Demand Split

The idea is to limit among how many non-zero path flows a demand d can be split, represented by the constant k_d. Dependig on k_d, this can result in a number of optimization problems and always introduces binary variables u_{dp}.

Single-Path-Allocation (SPA, $k_d = 1$):

- Constraints:

$$x_{dp} = h_d u_{dp} \qquad \forall d \in \{1,\ldots,D\}, p \in \{1,\ldots,P_d\} \tag{4.25}$$

$$\sum_{p=1}^{P_d} u_{dp} = 1 \qquad \forall d \in \{1,\ldots,D\} \tag{4.26}$$

$$\sum_{d=1}^{D}\sum_{p=1}^{P_d} \delta_{edp} x_{dp} \le c_e \qquad \forall e \in \{1,\ldots,E\} \tag{4.27}$$

The second constraint enforces that for each demand exactly one binary variable $u_{dp} = 1$ and all others are $= 0$. The first constraint enforces that $x_{dp} = 0$ whenever $u_{dp} = 0$, i. e. non-zero flows can only exists for paths and demands where $u_{dp} = 1$.

This problem is known to be NP-complete.

Integral Flow Pure Allocation Problem: For a set of demands find an integral solution (all x_{dp} are integers) so that capacity constraints of all edges are not exceeded.

The SPA formulation can be simplified by eliminating flow variables x_{dp}:

- Variables:
  - u_{dp}... binary variable, flow allocated to path p of demand d
- Constraints:

$$\sum_{p=1}^{P_d} u_{dp} = 1 \qquad \forall d \in \{1, \ldots, D\} \tag{4.28}$$

$$\sum_{d=1}^{D} h_d \sum_{p=1}^{P_d} \delta_{edp} u_{dp} \leq c_e \qquad \forall e \in \{1, \ldots, E\} \tag{4.29}$$

By using binary variables, the formulation that a demand d must be split equally among k_d candidate paths is also possible:

- Additional constants:
  - k_d... predetermined number of paths for demand d
- Constraints:

$$\sum_{p=1}^{P_d} u_{dp} = k_d \qquad \forall d \in \{1, \ldots, D\} \tag{4.30}$$

$$\sum_{d=1}^{D} \left(\sum_{p=1}^{P_d} \delta_{edp} u_{dp} \right) \frac{h_d}{k_d} \leq c_e \qquad e \in \{1, \ldots, E\} \tag{4.31}$$

Arbitrary Split Among k Paths:

- Constraints:

$$\sum_{p=1}^{P_d} x_{dp} = h_d \qquad d \in \{1, \ldots, D\} \tag{4.32}$$

$$\sum_{p=1}^{P_d} u_{dp} = k_d \qquad d \in \{1, \ldots, D\} \tag{4.33}$$

$$x_{dp} \leq u_{dp} h_d \qquad d \in \{1, \ldots, D\}, p \in \{1, \ldots, P_d\} \tag{4.34}$$

$$\sum_{d=1}^{D} \sum_{p=1}^{P_d} \delta_{edp} x_{dp} \leq c_e \qquad e \in \{1, \ldots, E\} \tag{4.35}$$

4.3. Modular Flow Allocation

In transport networks, demand volumes are usually given in terms of modular units, as the underlying technology normally only allows certain discrete network speeds (e. g.SFP modules, nBASE-T standards). In such cases, demands d can be modeled as a number H_d of demand modules, each with capacity L_d (thus $h_d = L_d \cdot H_d$).

- Additional constants:
  - L_d... demand module for demand d
  - H_d... volume of demand d expressed as the number of demand modules
- Constraints:

$$x_{dp} = L_d u_{dp} \qquad \forall d \in \{1,\dots,D\}, p \in \{1,\dots,P_d\} \tag{4.36}$$

$$\sum_{p=1}^{P_d} x_{dp} = h_d \qquad \forall d \in \{1,\dots D\} \tag{4.37}$$

$$\sum_{d=1}^{D}\sum_{p=1}^{P_d} \delta_{edp} x_{dp} \le c_e \qquad \forall e \in \{1,\dots,E\} \tag{4.38}$$

As $x_{dp} = L_d \cdot u_{dp}$, the variables x_{dp} can be eliminated to simplify the formulation:

- Additional constants:
  - L_d... demand module for demand d
  - H_d... volume of demand d expressed as the number of demand modules
- Constraints:

$$\sum_{p=1}^{P_d} u_{dp} = H_d \qquad \forall d \in \{1,\dots,D\} \tag{4.39}$$

$$\sum_{d=1}^{D} L_d \sum_{p=1}^{P_d} \delta_{edp} u_{dp} \le c_e \qquad \forall e \in \{1,\dots,E\} \tag{4.40}$$

4.4. Non-Linear Link Dimenioning, Cost and Delay Functions

So far, the assumption was that link capacities are equal to the link loads for uncapacitated problems. However, typically the link cost function is built upon the notion of the link dimensioning function $F_e\left(\underline{y}_e\right)$ which determines the relationship between the link load $\underline{y}_e$ and the minimal required link capacity y_e.

As the link cost was computed as the capacity times a cost coefficient ξ_e, the link cost was always considered linear. The following problems extend this by considering also module links, links with convex cost and links with concave cost.

4.4.1. Modular Links

In practice, links are often of modular size. Assume that the size of one *link capacity module* is M. The variable y_e denotes the number of link capacity modules. The link dimensioning function is:

$$F_e\left(\underline{y}_e\right) = k\xi_e \text{ with } (k-1)\,M \le \underline{y}_e \le kM \tag{4.41}$$

This gives the optimization problem for *modular links* (ML):

- Additional constants:
  - ξ_e... cost of one capacity module on link e
  - M... unique size of the link capacity module
- Objective:

$$\min \mathbf{F} = \sum_{e=1}^{E} \xi_e y_e \tag{4.42}$$

- Constraints:

$$\sum_{p=1}^{P_d} x_{dp} = h_d \qquad \forall d \in \{1, \ldots, D\} \tag{4.43}$$

$$\sum_{d=1}^{D} \sum_{p=1}^{P_d} \delta_{edp} x_{dp} \leq M y_e \qquad \forall e \in \{1, \ldots, E\} \tag{4.44}$$

The dimensioning problem with modular links ML is NP-complete.

This problem could be solved heuristically by assuming a linear approximation of the link dimensioning function F_e, solving the respective linear programming problem and then rounding up the obtained link capacities. This, however, can lead to solutions that are far from optimal.

ML can also be generalized to cover multiple module sizes $M_1, \ldots, M_K$ where K Is the number of module types and variable y_{ek} denotes the number of modules of size M_k installed on link e: Links With Multiple Modular Sizes (LMMS) :

- Additional constants:
  - ξ_{ek}... cost of one capacity module of type k on link e
  - M_k... size of the link capacity module of type k
- Objective:

$$\min \mathbf{F} = \sum_{e=1}^{E} \sum_{k=1}^{K} \xi_{ek} y_{ek} \tag{4.45}$$

- Constraints:

$$\sum_{p=1}^{P_d} x_{dp} = h_d \qquad \forall d \in \{1, \ldots, D\} \tag{4.46}$$

$$\sum_{d=1}^{D} \sum_{p=1}^{P_d} \delta_{edp} x_{dp} \leq \sum_{k=1}^{K} M_k y_{ek} \qquad \forall e \in \{1, \ldots, E\} \tag{4.47}$$

Modeling K different module sizes increases the number of required variables by a factor of K, as one set of variables is needed for each modular unit type. Yet another way of introducing module cost function with different modules is the incremental characterization:

- K denotes the number of steps.
- The incremental sizes of the link capacity module of type k are modeled with $m_1, m_2, \dots, m_k$ (if the load on a link passes one of these values, the respective cost function for this link "jumps").
- The cost of each incremental module m_k on link e is ξ_{ek}.
- Binary variables u_{ek} are used to indicate whether the incremental module of type k is installed on link e or not.

This gives the problem *Links With Incremental Modules* (LIM):

- Additional constants:
  - ξ_{ek}... cost of one capacity module of type k on link e
  - m_k... incremental size of the link capacity module of type k
- Variables:
  - x_{dp}... flow allocated to path p of demand d
  - u_{ek}... binary variable indicating if module of type k is installed on link e
- Objective:

$$\min \mathbf{F} = \sum_{e=1}^{E} \sum_{k=1}^{K} \xi_{ek} u_{ek} \tag{4.48}$$

- Constraints:

$$\sum_{p=1}^{P_d} x_{dp} = h_d \qquad \forall d \in \{1, \dots, D\} \tag{4.49}$$

$$\sum_{d=1}^{D} \sum_{p=1}^{P_d} \delta_{edp} x_{dp} \leq \sum_{k=1}^{K} m_k u_{ek} \qquad \forall e \in \{1, \dots, E\} \tag{4.50}$$

$$u_{e1} \geq u_{e2} \geq \dots \geq u_{eK} \qquad \forall e \in \{1, \dots, E\} \tag{4.51}$$

4.4.2. Convex Cost and Delay Functions

A real-valued function f defined on the interval $[0, \infty)$ is called *convex*, if for any two points $z_1, z_2 \in [0, \infty)$ and any $\alpha \in [0, 1]$ we have:

$$\alpha f(z_1) + (1 - \alpha) f(z_2) \geq f(\alpha z_1 + (1 - \alpha) z_1) \tag{4.52}$$

Pictorically, a function is called convex if the function lies below or on the straight line segment connecting two points, for any two points in the interval.

Allocation With Convex Cost Function (CCF):

- Additional constants:
  - $F_e(\cdot)$... convex cost function of link e
- Variables:
  - $\underline{y}_e$... load of link e
- Objective:

$$\min \mathbf{F} = \sum_{e=1}^{E} F_e\left(\underline{y}_e\right) \tag{4.53}$$

- Constraints:

$$\sum_{p=1}^{P_d} x_{dp} = h_d \qquad \forall d \in \{1,\ldots,D\} \tag{4.54}$$

$$\sum_{d=1}^{D}\sum_{p=1}^{P_d} \delta_{edp} x_{dp} = \underline{y}_e \qquad \forall e \in \{1,\ldots,E\} \tag{4.55}$$

$$\underline{y}_e \leq c_e \qquad \forall e \in \{1,\ldots,E\} \tag{4.56}$$

Convex function can be used to convert capacitated flow allocation problems to uncapacitated ones by using *penalty functions*. This requires that the penalty function is convex and incurs a high cost if the link capacity is violated. The uncapacitated problem can then be obtained by omitting the constraints

$$\underline{y}_e \leq c_e \qquad \forall e \in \{1,\ldots,E\} \tag{4.57}$$

To solve convex optimization problems with the techniques learned so far, linear approximations of the convex functions must be made to obtain a "corresponding" linear problem. In the general case, the convex function is approximated with a series $g_k(z)$ of linear functions:

- $g(z) = g_k(z) = a_k z + b_k$
- $s_{k-1} \leq z \leq s_k, k \in \{1,\ldots,K\}$
- $s_1 = 0, s_k = \infty$

As linear programming problems have their optimum solutions in the edges of the polytope describing the area of valid solutions, the solution computed will be a point where the linear approximation is equal to the actual convex function. Thus, an optimal solution to the approximative problem is also a valid solution to the original problem.

However, it is not necessarily an optimal solution! Different approximations can lead to different solutions.

Convex Penalty Function with Piecewise Linear Approximation (CPF/PLA):

- Additional incdices and constants:
  - $k = 1, \ldots, K_e$... consecutive pieces of the linear approximation of $F_e(\cdot)$
  - a_{ek}, b_{ek}... coefficients of the linear pieces of the linear approximation of $F_e(\cdot)$
- Variables:
  - r_e... continuous variable approximating $F_e\left(\underline{y}_e\right)$
- Objective:

$$\min \mathbf{F} = \sum_{e=1}^{E} r_e \tag{4.58}$$

- Constraints:

$$\sum_{p=1}^{P_d} x_{dp} = h_d \qquad \forall d \in \{1, \ldots, D\} \tag{4.59}$$

$$\sum_{d=1}^{D} \sum_{p=1}^{P_d} \delta_{edp} x_{dp} = \underline{y}_e \qquad \forall e \in \{1, \ldots, E\} \tag{4.60}$$

$$r_e \geq a_{ek}\underline{y}_e + b_{ek} \qquad \forall e \in \{1, \ldots, E\}, k \in \{1, \ldots, K_e\} \tag{4.61}$$

Thus, convex mathematical programming problems can be transformed into linear programming problems. Also, in many applications, it is not important to know the exact equations of the linear pieces of the approximation, but only the slopes a_{ek} and points s_k where they change matter.

$$\min a_1 z_1 + a_2 z_2 + \ldots + a_K z_K \tag{4.62}$$

subject to

$$y = z_1 + z_2 + \ldots + z_K \tag{4.63}$$

$$0 \leq z_1 \leq s_1 - s_0 \tag{4.64}$$

$$0 \leq z_2 \leq s_2 - s_1 \tag{4.65}$$

$$\vdots \tag{4.66}$$

$$0 \leq z_K \leq s_K - s_{K-1} \tag{4.67}$$

This works, because due to convexity we have $a_1 < a_2 < \ldots < a_K$.

Problems with a linear cost function, but convex constraints are also possible, e. g. minimizing capacity cost for a fixed routing under the constraint that a given average acceptable delay $\hat{D}$ has to be met. This can be solved by various methods:

- Karush-Kuhn-Tucker conditions
- Classical Lagrangian multiplier method
- Piecewise linear approximation as described before

Capacity Design With Fixed Routing and Delay Constraint:

- Constants:
  - $\underline{y}_e$... load on link e induced by fixed routing
  - $\hat{D}$... acceptable delay
  - H... total traffic volume $H = \sum_d H_d$
- Variables:
  - y_e... capacity of link e
- Objective:

$$\min \mathbf{F} = \sum_e^E \xi_e y_e \tag{4.68}$$

- Constraints:

$$y_e \geq \underline{y}_e \qquad \forall e \in \{1, \dots, E\} \tag{4.69}$$

$$\frac{1}{H} \sum_e^E \frac{y_e}{y_e - \underline{y}_e} \leq \hat{D} \tag{4.70}$$

4.4.3. Concave Link Dimensioning Functions

A real-valued function f defined on the interval $[0, \infty)$ is called *concave*, if for any two points $z_1, z_2 \in [0, \infty)$ and any $\alpha \in [0, 1]$ we have:

$$\alpha f(z_1) + (1 - \alpha) f(z_2) \leq f(\alpha z_1 + (1 - \alpha) z_2) \tag{4.71}$$

Pictorically, a function is called concave if the function lies above or on the straight line segment connecting two points, for any two points in the interval.

In networks, concave functions often appear to describe link dimensioning functions, as growth in link costs often adheres to the following relation:

$$\frac{f(z_1)}{z_1} \geq \frac{f(z_2)}{z_2} \text{ for } z_1 < z_2 \tag{4.72}$$

Concave Link Dimensioning Functions (CDF):

- Additional constants:

  - $F_e(\cdot)$... non-decreasing concave dimensioning function of link e

- Variables:
  - $\underline{y}_e$... load of link e
- Objective:

$$\min \mathbf{F} = \sum_{e}^{E} \xi_e F_e \left(\underline{y}_e\right) \tag{4.73}$$

- Constraints:

$$\sum_{p=1}^{P_d} x_{dp} = h_d \qquad \forall d \in \{1, \dots, D\} \tag{4.74}$$

$$\sum_{d=1}^{D} \sum_{p}^{P_d} \delta_{edp} x_{dp} = \underline{y}_e \qquad \forall e \in \{1, \dots, E\} \tag{4.75}$$

Optimal solutions to these kinds of problems are non-bifurcated as allocating one big link is cheaper than allocating two small links. As these problems require minimizing a concave objective function subject to linear constraints, they in general can have numerous local minima on the extreme points of the feasible region defined by the constraints. Thus, finding the global minimum can be a very difficult task.

Piecewise linear approximation as was done for convex functions does not lead to a linear programming problem for concave functions. However, approximation leads to a mixed-integer programming program. In general:

$$g(z) := g_k(z) = a_k z + b_k, s_{k-1} \leq z < s_k, k \in \{1, \dots, K\} \tag{4.76}$$

To avoid multiplying two variables (which is forbidden in mixed integer programming problems), additional variables y_k must be introduced and limited by additional constraints:

$$\min \sum_{k=1}^{K} (a_k y_k + b_k u_k) \tag{4.77}$$

subject to

$$\sum_{k}^{K} y_k = y \tag{4.78}$$

$$y_k \leq \Delta u_k \qquad \forall k \in \{1, \dots, K\} \tag{4.79}$$

$$y_k \quad \text{nonnegative continuous} \tag{4.80}$$

$$u_k \quad \text{binary} \tag{4.81}$$

$$\Delta \quad \text{number larger than any potential value } y \tag{4.82}$$

These constraints force that exactly one value (the right one) y_k will be non-zero and equal to y in the optimal solution.

Concave Dimensioning Functions with Piecewise Mixed-Integer Approximation (CDF/ PMIA):

- Additional Indices and Constants:
  - $k = 1, \dots, K_e$... consecutive pieces of the linear approximation
  - a_{ek}, b_{ek}... coefficients of the linear pieces of the approximation
- Variables:
  - y_{ek}, u_{ek} continuous/binary variables for link e
- Objective:

$$\min \mathbf{F} = \sum_e \sum_k (a_{ek} y_{ek} + b_{ek} u_{ek}) \tag{4.83}$$

- Constraints:

$$\sum_{p=1}^{P_d} x_{dp} = h_d \qquad \forall d \in \{1, \dots, D\} \tag{4.84}$$

$$\sum_{d=1}^{D} \sum_{p=1}^{P_d} \delta_{edp} x_{dp} = \underline{y}_e \qquad \forall e \in \{1, \dots, E\} \tag{4.85}$$

$$\sum_{k=1}^{K} y_{ek} = \underline{y}_e \qquad \forall e \in \{1, \dots, E\} \tag{4.86}$$

$$\sum_{k=1}^{K} u_{ek} = 1 \qquad \forall e \in \{1, \dots, E\} \tag{4.87}$$

$$y_{ek} \leq \Delta u_{ek} \qquad \forall e \in \{1, \dots, E\}, k \in \{1, \dots, K\} \tag{4.88}$$

Assuming that the piecewise linear approximation involves the same number K of pieces for every link, the above problem contains:

- $E \cdot K$ additional continuous variables y_{ek}
- $E \cdot K$ additional binary variables u_{ek}
- $E \cdot (K + 2)$ additional constraints

This indicates that the problem is difficult to solve. Ther are no algorithms known that are significantly better than the full search in the space of binary variables.

Similar to convex functions, sometimes just looking at slopes and the points where the slopes change might be enough instead of doing a full approximation.

4.5. Budget Constraints

A possible alternative optimization goal to minimizing cost is to stay within a given budget constraint and choose a different optimization goal, e. g. when optimizing for throughput, while staying in budget constraints:

Budget Constraint (BC):

- Additional Constants:
  - B... given budget
  - h_d... reference volume of demand d
- Variables:
  - y_e... capacity of link e
  - r... proportion of the realized demand volumes
- Objective:

$$\max r \tag{4.89}$$

- Constraints:

$$\sum_{p=1}^{P_d} x_{dp} \geq rh_d \quad \forall d \in \{1, \dots, D\} \sum_{d=1}^{D} \sum_{p=1}^{P_d} \delta_{edp} x_{dp} \quad \leq y_e \forall e \quad \in \{1, \dots, E\} \sum_{e=1}^{E} \xi_e y_e \leq B \tag{4.90}$$

4.6. Incremental Network Design Problems

Often networks are not designed from scratch, but have to be extended with additional resources. In such cases, there are already existing link capacities c_e and the task is to add additional capacities y_e to account for an increase in the demand volume.

For this, add to the existing problem capacity constraints for all links which are already present. The cost of such a problem is generally higher than the optimal solution of a pure network design problem.

Simple Extension Problem (SEP):

- Additional Constants:
  - c_e... existing capacity of link e
  - ξ_e... unit cost of link e
- Variables:
  - y_e... extra capacity of link e on top of c_e

- Objective:

$$\min \mathbf{F} = \sum_{e=1}^{E} \xi_e y_e \tag{4.91}$$

- Constraints:

$$\sum_{p=1}^{P_d} x_{dp} = h_d \qquad \forall d \in \{1, \dots, D\} \tag{4.92}$$

$$\sum_{d=1}^{D} \sum_{p=1}^{P_d} \delta_{edp} x_{dp} \leq y_e + c_e \qquad \forall e \in \{1, \dots, E\} \tag{4.93}$$

$$\tag{4.94}$$

4.7. Representing Nodes

So far, only link capacities were considered, but node capacities can be limited or imply costs as well. These can be modeled by splitting up each node into an ingress node, an egress node and a connecting internal link. All ingress links are connected to the ingress node and all egress links are connected to the egress node. The link capacity of the internal link can then be used to model the capacity of the node from the original graph.

This can be used both for limiting capacity constraints on nodes or node failures (by failing its internal link).

5. Network Resilience

5.1. Introduction

Network resilience denote the property of a network to sustain the ability to communicate even if parts (nodes, links) of the network fail. Quantification of random failures often computes the probability of certain conditions, e. g. partitioning of a network, etc. Quantification of failures due to deliberate attacks often computes the worst case damage, e. g. smallest number of attacked/failed links or nodes so that the remaining network is partitioned, etc. Additionally, the smallest number of additional links that need to be added in order to increase the resilience of a network against random failures or deliberate attacks can be of interest.

Definitions from Graph Theory:

- Two paths p_1 and p_2 from x to y in G are *edge independent*, if they have no link in common.
- Two paths p_1 and p_2 from x to y in G are *node independent*, if they only have nodes x and y in commong.
- If there is at least one path linking every pair of actors in the graph then the graph is called *connected.*
- If there are k-edge-independent paths connecting every pair, the graph is *k-edge-connected.*
- If there are k node-independent paths connecting every pair, the graph is *k-node-connected.*
- The biggest number k for which G is k-edge-connected is called the *edge-connectivity* of G.
- The biggest number k for which G is k-node-connected is called the *node-connectivity* of G. Graphs that are 2-node-connected are also called *biconnected.*
- In any connected component, the path(s) linking two non-adjacent nodes must pass through a subset of other nodes, which if removed, would disconnect them.
  - For two nodes s and t the set $T \subseteq V\ \{s, t\}$ is called an *s-t-cutting-node-set* if every path connecting s and t passes through at least one node of T, that is there is no path from s to t in $G \setminus T$.

- A set T is called a *cutting-node-set* if T is an s-t-cutting-node-set for two nodes s and t.
- For two nodes s and t the set $F \subseteq E$ is called an *s-t-cutting-edge-set* if every path connecting s and t traverses at least one edge of F, that is there is no path from s to t in $G \setminus F$.
- A set F is called a *cutting-edge-set* if F is an s-t-cutting-edge-set for two nodes s and t.

5.2. Menger's and Whitney's Theorems

Menger's Theorem: For non-adjacent nodes s and t in an undirected graph, the maximum number of node independent paths is equal to the minimum size of s-t-cutting-node-set. For nodes s and t in an undirected graph, the maximum number of edge independent paths is equal to the minimum size of an s-t-cutting-edge-set.

Whitney's Theorem: An undirected graph with at least $k+1$ nodes is k-node-connected iff each cutting-node-set in G contains at least k nodes. An undirected graph is k-edge-connected iff each cutting-edge-set in G contains at least k edges.

Implications for communication networks:

- If a communication network is supposed to allow communication between arbitrary nodes even in case of failure of r arbitrary nodes, its topology must be at least $(r+1)$-node-connected.
- If a communication network is supposed to allow communication betwee arbitrary nodes even in case of failure of s arbitrary links, its topology must be at least $(s+1)$-edge-connected.

Thus, the network graph can be analyzed to get information about resilience by calculating the highest k for which G is k-edge/node-connected. This can be solved in polynomial time.

If a graph is not k-edge/node-connected augmentations to G can be searched to find a minimum set of edges/nodes to add so that G becomes k-edge/node-connected. For edge-connectivity, this can be solved in polynomial time. For node-connectivity, polynomial algorithms are only known for $k \leq 4$. The weighted variant with the objective of weight minimization is NP-hard already for $k = 2$.

5.3. Block Structure of Graphs

Let $G = (V, E)$ be an undirected Graph with $|V| \geq 3$. Of interest are all subgraphs of maximum sizie that are biconnected. G is biconnected iff either G is a single edge or for each tuple of vertices (u, v) there are at least to node disjoint paths. The intersection of two maximum size biconnected components consists of at most one vertex, which is called *articulation node*. A graph G with at least 3 nodes is biconnected, iff G does not contain isolated nodes and every pair of nodes is on a common single cycle.

Let $e_1, e_2 \in E$ and $\equiv$ be defined as $e_1 \equiv e_2$ iff e_1 and e_2 lie on a common simple cycle. This equivalence relation partitions E into sets $E_1, E_2, \ldots, E_h$. Let $G_i = G[E_i]$ for $i \in \{1, \ldots, h\}$. These subgraphs are called *blocks*. Blocks with at least 2 edges are the maximum sized biconnected components of G.

Let $G_i = G[E_i] = (V_i, E_i)$ be the blocks of G, then

- $\forall i \neq j \in \{1, \ldots, h\} : |V_i \cap V_j| \leq 1$.
- A node $a \in V$ is an articulation node iff $\exists i \neq j \in \{1, \ldots, h\} : V_i \cap V_j = \{a\}$.

The block structure of a graph can be describe with a so-called *block structure graph* $B(G)$ that contains nodes v_a for each articulation node a and nodes v_b for each block b with each v_a being connected to the nodes v_b denoting the respective biconnected components b that node a is connected to in G. $B(G)$ is a tree.

Calculating $B(G)$ allows to identify articulation nodes as well as the blocks of a graph. It also allows network designers to see which nodes in the network are more important to protect or between which components of the network additional links are needed.

5.4. DFS Spanning Trees on Network Graphs

5.4.1. Classification of Edges

Articulation nodes can be found in $\mathcal{O}(|V|(|V| + |E|))$, by deleting each node and using DFS to calculate a spanning tree to see if it's still connected. During DFS, *preorder* and *postorder* numbers can be calculated, which are incremented for each node before and after recursion, respectively.

Classification of edges of G with respect to a spanning tree T:

- An edge $vw \in E(T)$ is called a *tree edge*.
- An edge $vw \in E(G) \setminus E(T)$ is called a *back edge* if v is a descendant or ancestor of w.
- Else, vw is called a *cross edge* (these don't exist if T is a depth-first-search tree).

Let D_v be the number of descendants of v. Then w is descendant of v iff $v.pre < w.pre \leq v.pre + D_v$. The following notation is used for edges with respect to the DFS-tree T:

- $u \rightarrow v$ iff $uv \in E(T)$
- $u \overset{*}{\rightarrow} v$ iff u is an ancestor of v
- $w \dashrightarrow u$ iff wu is back edge if wu in $E(G) \setminus E(T)$ with either $w \overset{*}{\rightarrow} u$ or $u \overset{*}{\rightarrow} w$.

5.4.2. Computing Articulation Nodes

If w is descendant of v and wu is back edge such that $u.pre < v.pre$, then u is a proper ancestor of v. In a DFS tree T, a node v other than the root is an articuation node iff v is not a leaf and some subtree of v has no back edge incident to a proper ancestor of v and some subtree of v has no back edge incident to a proper ancestor of v.

To efficiently implement this test, the so-called *low value* is used: For each vertex v define

$$low(v) := \min\left(\{v.pre\} \cup \left\{w.pre \,\middle|\, v \overset{*}{\rightarrow} \text{---}\, w\right\}\right) \tag{5.1}$$

With $v \overset{*}{\rightarrow} \text{---}\, w$ meaning that v is connect to w through a path of tree edges and potentially one additional back edge as the last edge.

$$low(v) = \min\left(\{v.pre\} \cup \{low(w) \mid v \rightarrow w\} \cup \{w.pre \mid v \text{---}\, w\}\right) \tag{5.2}$$

$low(v) = v.low$ can be computed for all nodes $v \in V(G)$ by using DFS and evaluationg preorder values of incident nodes as each node is visited. For each node v that is visited during DFS, set $v.pre$, initialize $v.low := v.pre$ and consider all edges of v:

- For tree edges to unvisited nodes w, perform a recursive call and after it returns and $w.low$ has been computed properly, set $v.low := \min\{v.low, w.low\}$.
- For back edges to nodes w that have already been visited, set $v.low := \min\{v.low, w.pre\}$.

A node a is an articulation node iff either the node a is the DFS tree root with ≥ 2 tree children or the node a is not the DFS tree root, but it has a tree child v with $low(v) \geq a.pre$. Thus, the articulation nodes of a graph can be calculated with a slightly modified DFS in $\mathcal{O}(|V| + |E|)$.

5.4.3. Computing the Blocks of a Graph

In order to also compute the blocks of G while computing articulation points, an additional stack s of edges is introduced:

- Whenever a tree edge $v \rightarrow w$ is found, push it to s prior to making the recursive call.
- Whenever a back edge is found, push it to s.
- Whenever a recursive call for node w returns to v and $w.low \geq v.pre$, then all edges on top of the stack up to $v \rightarrow w$ form the next identified block.

With all operations for the stack being done in $\mathcal{O}(|V| + |E|)$ (holds true when memory is preallocated), the running time of the algorithm is $\mathcal{O}(|V| + |E|)$.

A. Letter Salad Decryption Manual

In case one does no longer know which letter means what, here is a semi-comprehensive list of letters used throughout the lecture:

v Node

e Edge/Link

P Path

δ Indicator wheter a link is on a path

x Flow

y Link capacity (uncapacitated problems)

c Link capacity (capacitated problems)

h Demand

ξ Link cost

ζ Path cost

w Weight

κ Opening cost

s Failure state

α Link up in failure state

u binary variable

ε Lower bound

n_d Minimum number of path splits

$\hat{D}$ Delay

B Budget

Stichwortverzeichnis