Information and Computing Sciences

Utrecht University

DNV Kema

Master thesis project

Automatic planning tools for power system design

Supervisors:

Marjan van den Akker (UU)
Han Hoogeveen (UU)
Han LaPoutré (UU)
Roger Cremers (KEMA)

Author:

Alexandru-Nicolae Dimitriu

Utrecht, 2012

Abstract

The purpose of this thesis is to investigate different techniques for designing electrical power network grids and demonstrate how algorithms can be used and combined in order to solve two power network design subproblems.

In the first part, we focus on the problem of connecting a new energy consumer to an existing electrical grid based on its distance to the possible connection points. The terrain is divided into convex or concave polygons, each having a cost for laying cable within its boundary. To solve this problem we use an approach based on Dijkstra's algorithm in order to produce close to optimal solutions. For this particular problem we conducted computational performance experiments on two identical implementations written in C++ and PowerFactory. The results support the view that C++ is a much more efficient programming language than the Digsilent Programming Language (DPL) used in PowerFactory.

In the second part of the thesis, we analyze the problem of routing electricity from producers to consumers in a simplified electrical network, given that connections are characterized by capacity and cost. The goal is to find a solution which maximizes the amount of energy sent through the network (i.e. satisfy as much demand as the network can handle). If there are multiple such solutions, we are interested in the one with the minimum cost. The cost of sending energy is determined by the cost of acquiring connections and the cost of sending energy through those connections. We propose a branch-and-bound approach which obtains an exact solution, a simulated annealing approach and a heuristic which varies the cost of the connections in order to explore the solution space.

Contents

Abstract **ii**

1 Introduction **1**
- 1.1 Project goal . . . 2
- 1.2 Problem statement . . . 2

2 Optimal paths in a non-uniform weighted two-dimensional space **4**
- 2.1 Simplified weighted region problem . . . 5
- 2.2 Generalized weighted region problem . . . 6
  - 2.2.1 Implementation and results . . . 8

3 Network flow applications in power system design **10**
- 3.1 Minimum-cost maximum-flow problem with edge demands (MCMFP-ED) 11
  - 3.1.1 Identifying a feasible flow in the network. . . . 13
  - 3.1.2 Finding the maximum flow with the minimum cost in the network. 15
- 3.2 Minimum-cost maximum-flow problem with fixed cost on edges (MCMFP-FC) 20
  - 3.2.1 Branch-and-bound algorithm (BB) . . . 22
  - 3.2.2 Simulated annealing heuristic . . . 28
  - 3.2.3 Cost function slope heuristic . . . 30
  - 3.2.4 Experiments and results . . . 31

4 Conclusions **36**
- 4.1 Future work . . . 37

A Dijkstra's algorithm **39**

B Bellman-Ford algorithm **40**

C Edmonds-Karp algorithm **41**

D Checking if a line segment intersects the interior of a polygon **43**
  - D.0.1 Finding the intersection point between two segments . . . 44
  - D.0.2 Checking if a point is inside a polygon . . . 45

E Input format for the C++ implementations **47**
- E.1 Optimal paths in a non-uniform weighted two-dimensional space . . . 47
- E.2 MCMFP-FC . . . 48

Chapter 1

Introduction

An electric power system is a network of electrical components used to supply, transmit and use electric power. It is broadly divided into the generators that supply the power, the transmission system that transports the power from the generating centers to the load centers and the distribution system that feeds the power to local homes and industries. The system starts with generation, by which electrical energy is produced in power plants and transformed to high-voltage electrical energy that is more suitable for efficient long-distance transportation. High-voltage power lines transport the electrical energy over long distances to the distribution substations. There, the high-voltage energy is transformed back into lower-voltage energy that is transmitted over distribution power lines for residential, commercial or industrial consumption. Although the electric power system was initially developed in the late 1800s, it is still undergoing change and continues to evolve in our current time, with a focus on decentralized generation sources.

A full-scale electric power system is much more complex than the basic generation-transmission-distribution paradigm, as it involves a large number of elements to take into consideration during its design:

- Generation - What kind of energy generation should be used, i.e. fossil fuel, nuclear energy, renewable energy (wind, sun). Where should the generators be located in the network and how much energy should they generate?
- Storage - Where to place the storage equipment? What properties should they have?
- Cables - What type of cables should be used?
- Layout and planning - Where to build substations? How to efficiently connect them (what type of topology should be used)? Should the network be overground or underground?
- Scalability - How easy can the network be extended? Can it accommodate future increase in the demand of electrical energy?
- Protective devices (protective relays, circuit breakers, batteries).
- Safety standards and reliability.

1.1 Project goal

Designing an electric power system (or a part of it) typically consists of three functional steps: identifying alternatives, evaluating the alternatives against criteria and desired attributes and selecting the best alternative. These steps can be particularly difficult, mainly because the problem usually has millions of alternatives which need to be analyzed. While a domain expert can use his judgment and experience in order to reduce the set of possible alternatives, the evaluation of these alternatives is still a tedious and lengthy process, time consuming and expensive, as well as prone to error if done manually. Therefore, the goal of this project is to develop automatic planning tools in order to *help* a domain expert solve electrical power system design subproblems.

1.2 Problem statement

In the second chapter of this paper we develop a tool which can solve the following layout problem: we want to expand an electrical power network by connecting a new entity (load, substation, etc.) to the existing grid. The domain expert identifies multiple points in the grid where the new entity can be connected, but he has to determine which of those connection points to use, based on the cost of laying cable from the new entity to each of the connection points. We model the terrain by using a map which is divided into convex or concave polygons, with different cost coefficients, representing the cost of laying cable across that region. We use an optimization algorithm to determine the best way to expand the grid by adding the new entity. The output of the automated tool represents an exact layout of how the new entity is connected to the existing grid, in terms of where the cables should be placed in the terrain. We approach this problem by first identifying a solution for the simplified problem where the polygons' coefficient can only be 1 (normal terrain) or ∞ (obstacles). We then extend the algorithm to work for any positive coefficient values, which makes the model more realistic: besides normal terrain and obstacles, we can model areas where the cost of laying cable is smaller (for example, areas where overground cables are allowed, i.e. no digging is required) or larger (for example, remote areas).

In the third chapter we tackle a more difficult problem: we are given input data of a network containing nodes which can supply energy (producers), nodes which consume energy (consumers) and intermediate nodes which are used as connections points between these entities. We know the production of each supply node and the demand of each consumer node and we want to identify the optimum layout in terms of the total cost we need to spend for the connections we choose to use. The logic behind choosing the connections is that we want to maximize the flow of energy in the network, i.e. satisfy as much demand of the customers as possible. Given the large complexity of the problem we make the following assumptions on the model:

- The production and the demand values are given for a fixed moment in time. We choose this approach in order to simplify the model and reduce the number of alternatives we have to analyze.
- In an electrical power system, the connections are bi-directional, meaning that

current can be transmitted through the cable in both directions depending on the state of the production and demand patterns. As we choose to analyze the network for a fixed moment in time, the cables in our model are directed, as current can only flow in one direction at a specific moment. However, we permit our input to have any number of connections between two nodes in the network. For example, if we do not know beforehand in which direction the current should flow between two nodes u and v, we can have two edges in the input, one from u to v and one from v to u. The final design will only contain one of these edges.

- Cables are characterized by minimum capacity, maximum capacity, cost per unit of flow and fixed cost. The minimum capacity represents the minimum amount of current we want to send on the connection. By default this value is 0, but if we want to force the algorithm to use a connection in the final solution we can put a strictly positive value as the minimum capacity of that connection. The maximum capacity represents the maximum amount of current we can send on the connection. The cost per unit of flow is the cost we have to pay for each unit of current we send across the connection. This value is usually derived from the type of cable used for the connection (for example, a longer cable will have a larger resistance, meaning a larger voltage drop which implies larger costs). The fixed cost represents the cost of acquiring the connection, and includes the cost of the cable and the cost of laying the cable in the terrain.
- We do not account for the voltage constraints which exist in a regular electrical network. However, we do use the flow conservation constraint, meaning that the amount of current entering a node must be equal to the amount of current leaving that node (with the exception of nodes which produce or consume energy).

While some of these assumptions make the model less realistic, they also make it feasible to be used for solving medium-sized instances of the problem (dozens of nodes, hundreds of connections) in a reasonable amount of time. The solution proposed by our automatic tool should not be considered a final design, but a starting solution for a further analysis, either conducted by another tool or by a domain expert.

Chapter 2

Optimal paths in a non-uniform weighted two-dimensional space

In power system design, electrical engineers have to find the optimal way to connect a point (e.g. a house) to another point from a set (e.g. the possible existing grid connection points). Laying cable across the terrain has different costs, or is even forbidden for some regions. We assume that we can subdivide the study case map into simple polygons (convex or concave) with different cost coefficients. The default cost coefficient is 1, but there can be *forbidden* areas with infinite cost (regions which can not be crossed by cables) or *custom* areas which have a cost coefficient larger than 0 but smaller than infinity. For example, if the law forbids a cable crossing a certain region, we will assign a polygon with infinite cost for that region; however, if for some other region laying cable is cheaper (because maybe the digging for the cable was previously done) we can assign a polygon with cost 0.2 for that specific region.

We will use the cost coefficient of a polygon to quantify the actual cost for laying one unit of cable crossing that region. One unit represents a distance of 1 in the Euclidean space in which we are given the polygons. For a region belonging to multiple polygons (overlapping) the following priority rule is used to determine its cost: if one of the overlapping polygons is a forbidden polygon then the cost of the region is infinite, otherwise the cost will be the minimum cost of the custom overlapping polygons. The total cost of a path between two points is measured according to the weighted Euclidean metric - the cost of a path is defined to be the weighted sum of Euclidean lengths of the sub paths within each region.

In the field of computer science this problem is known as the *weighted region problem* and it was first introduced by Mitchell and Papadimitriou [28] as a generalization of finding the shortest path in a two-dimensional space with obstacles. It has been recently proven by Carufel et al. [4] that this problem is unsolvable in the Algebraic Computation Model over the Rational Numbers (ACM$\mathbb{Q}$[1]). This result, combined with the fact that modeling a real terrain using cost coefficient polygons is by its nature inaccurate, makes approximation algorithms an appealing alternative for solving this problem. In our

[1] The ACM$\mathbb{Q}$ can compute exactly any number that can be obtained from the rationals $\mathbb{Q}$ by applying a finite number of operations from $+, -, \times, \backslash, \sqrt[k]{\ }$, for any integer $k \geq 2$.

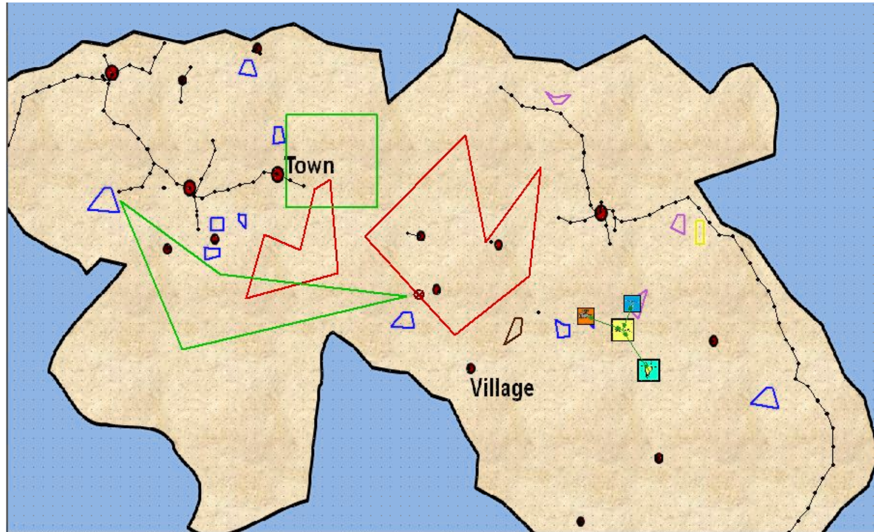

Figure 2.1. Example of a power system design map. Forbidden polygons are displayed in red color (infinite cost), while custom polygons are displayed in green color.

project we have analyzed and solved the weighted region problem for two cases:

- The simplified case, where the polygons can either have a cost coefficient of 1 or infinity.
- The generalized case, where the polygons can have any positive cost coefficient.

We will proceed to describe in detail the solution chosen for each of these cases.

2.1 Simplified weighted region problem

In the simplified weighted region problem we are interested to find the shortest path from a *start* point s to a set of *end* points T, given that the map is divided into regions with cost coefficient either 1 or infinity. We use the following theorem as a cornerstone of our shortest path algorithm:

Theorem 1. *Any shortest path between s and an end point $t \in T$, among a set Q of polygonal obstacles (polygons with infinite cost coefficient) is a polygonal path (a connected series of line segments) whose inner vertices are vertices of Q.*

The proof of this theorem is given by Mark de Berg et all [8, p. 325]. Given this theorem, we can create a graph G with the following vertices:

- the start point s.
- the end points in T.
- the vertices of the forbidden polygons.

There will be an edge between any two vertices in this graph if and only if the line connecting the two points does not cross the interior of any forbidden polygon. Constructing the set of edges of the graph can be trivially computed in $O\left(n^3\right)$ time (where

n is the number of vertices in the graph) by taking each line connecting a pair of vertices and testing for intersection with each forbidden polygon (see D for the algorithm used to check for an intersection between a line and the interior of a polygon). The cost of such an edge is the Euclidean distance between the two vertices.

The set of edges of G can be determined in a better fashion if the pair of vertices for which we test the intersection with the forbidden polygons are not chosen in arbitrary order, but instead concentrate on one vertex at a time and identify all vertices which are *visible* from it. A vertex u is visible for some other vertex v, if the segment (v, u) does not intersect any forbidden polygons. This approach is known as computing the *visibility graph* and can be optimally computed in $O\left(n^2 logn\right)$ time (de Berg et al. [8, p. 325]). However, this approach was not implemented in our solution.

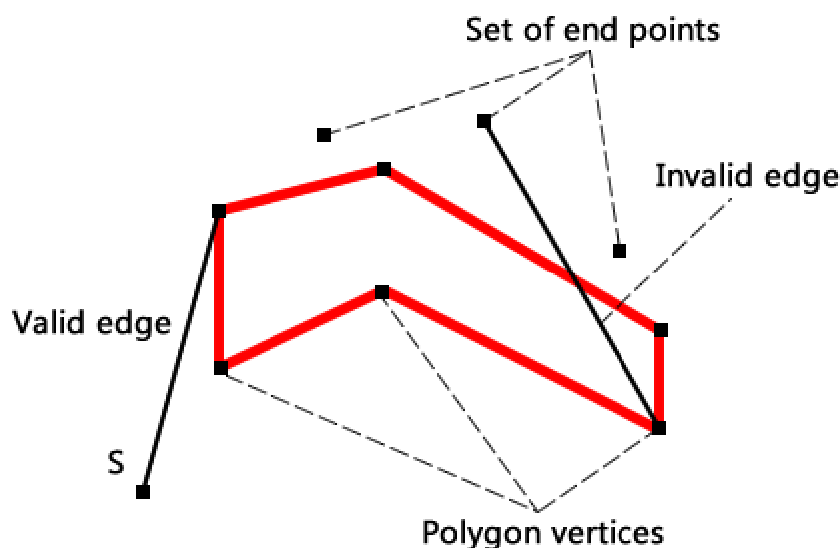

Figure 2.2. Construction of graph G.

Once we have constructed graph G, we can find the shortest paths from s to any other vertex in G by applying Dijkstra's algorithm (see Appendix A for the algorithm description). This algorithm has a running time of $O\left(n^2\right)$ and computes the optimal solution for the simplified weighted region problem.

2.2 Generalized weighted region problem

The difference between the generalized and the simplified weighted region problem is that the polygon coefficients are not limited to 1 or infinity but they can take any positive value. Intuitively, this problem is much more complex than the previous one, as in this case, the optimal path can pass through polygons and it is hard to determine through which polygons it should pass and what should be the entry and the exit points for those polygons. As solving this problem to optimality is not possible in polynomial time, we have implemented an approximation algorithm which uses as starting point Snell's law ([1]). A light ray traveling through different media will always choose the

shortest path, and according to Snell's Law, the only *bending* points on the trajectory will occur at the border of two different isotropic media. In our problem, the shortest path between two points will only have bending points on the edges of the *custom* polygons. As a result, we could use the same Dijkstra algorithm as in the simplified weighted region problem on an extended graph containing all the points on the edges of the custom polygons. However, there are an infinity of such points. To solve this issue, we have used an *edge sampling* approach, selecting only certain possible bend points on the edges of the custom polygons.

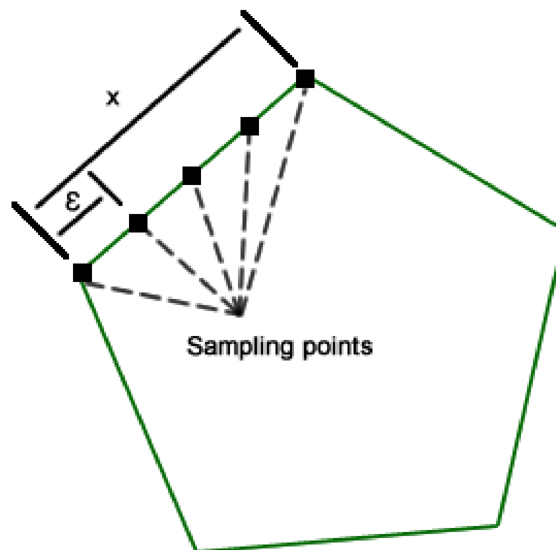

Figure 2.3. Sampling using the accuracy parameter ϵ.

The precision of this approximation algorithm lies in how many points we chose to sample on each edge. A larger number will provide a more accurate solution but it will also increase the running time. One possible way to sample the points would be to let the user select an accuracy parameter ϵ, such that on an edge of a custom polygon with length x we would have $\frac{x}{\epsilon} + 1$ sampling points. However, we think that choosing a good value for this parameter is not an easy task for the user as the same parameter ϵ will yield different running times depending on the length of the edges of the custom polygons. Instead, we use the following approach:

- The user chooses a maximum number of points n that graph G will contain, instead of the accuracy parameter ϵ.
- Let $m = n - 1 - |T| - k$ be the number of sample points. This value is obtained by subtracting from the maximum number of points, the start point, the end points and the total number of vertices of the forbidden polygons (k).
- Let L be the total length of the edges of the custom polygons.
- We know that we have to sample m points on custom polygon edges which have a total length of L. Therefore, for a custom polygon edge of length l we will sample $\frac{ml}{L}$ points, equally distanced on that edge.

In this approach the Dijkstra algorithm will run for a fixed sized graph and the running time will always be $O\left(n^3\right)$ (given by the construction of the visibility graph).

2.2.1 Implementation and results

We have implemented the generalized weighted region problem in two different settings: PowerFactory and C++. PowerFactory[1] is one of the most used application for studying large interconnected power systems. Its main advantages are the electrical modeling capabilities as well as the load flow calculation algorithm for an AC or DC network topology. The results in tables 2.1 and 2.2 show a comparison of the run times for the two (identical) implementations for two test cases.

	Running time (s)		
n	PowerFactory	C++	Shortest distance
100	6	0.026	136.487
250	56	0.122	136.487
500	-	0.483	136.487
1000	-	2	136.487
3000	-	18.176	136.487

Table 2.1. Comparison of the run times for the PowerFactory and C++ implementations of the generalized weighted region problem - test case 1. The shortest distance does not improve when we increase the number of points (n) because the optimal solution goes through the vertices of the polygons.

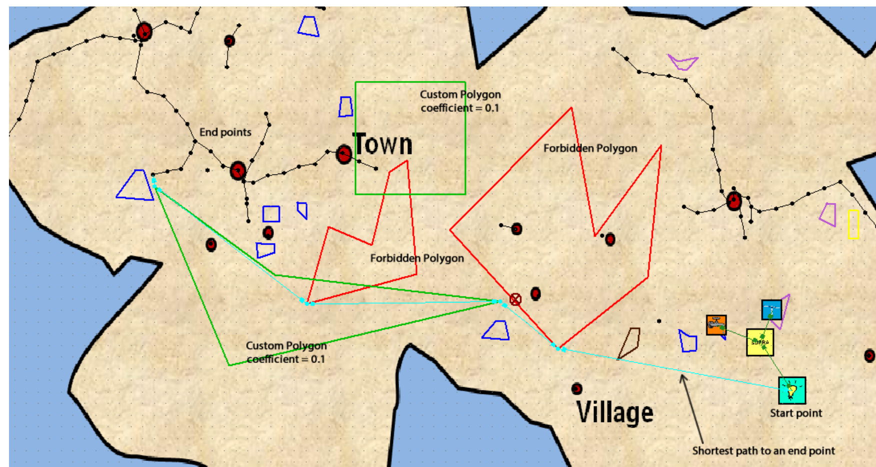

Figure 2.4. Sample output of the Power Factory implementation (for the test case in table 2.1).

[1] http://www.digsilent.de/index.php/products-powerfactory.html

	Running time (s)		
n	PowerFactory	C++	Shortest distance
100	6	0.054	166.683
250	41	0.133	166.252
500	-	0.572	166.239
1000	-	1.771	166.235
3000	-	16.180	166.235

Table 2.2. Comparison of the run times for the PowerFactory and C++ implementations of the generalized weighted region problem - test case 2. The shortest distance improves as we increase the number of points (n).

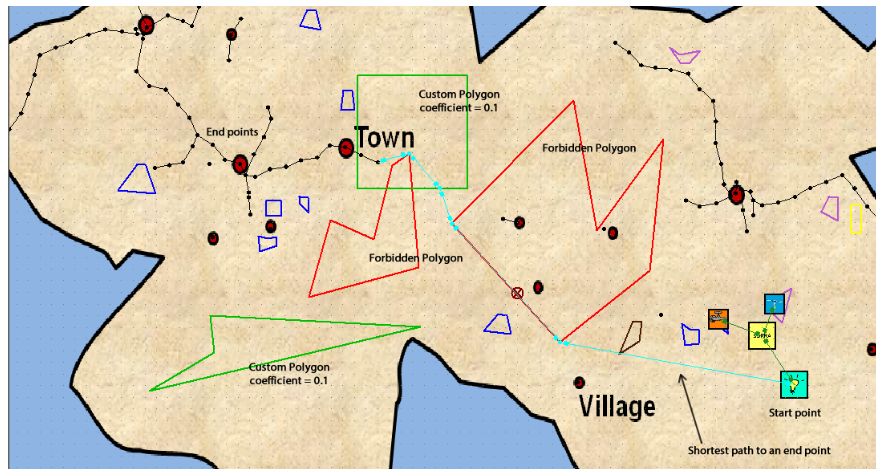

Figure 2.5. Sample output of the Power Factory implementation (for the test case in table 2.2).

Chapter 3

Network flow applications in power system design

One important problem in power system design is to identify a good way to connect the end users (consumers) to the energy suppliers (producers). The design usually involves intermediate points (substations) which do not consume or produce energy, but act as connection points. In such networks, there are multiple viable designs but one has to choose the most cost-efficient solution. Such a problem can be modeled as a classical network flow problem, where the goal is to maximize the flow (i.e. satisfying as much of the customer demand as possible) while in the same time choosing the most inexpensive way to send that flow across the network. In order to solve this problem we use a two phase approach:

- In Section 3.1, we try to solve the problem for the case when the connections have no fixed cost, using a standard minimum-cost maximum-flow algorithm. This is a known polynomial time algorithm which obtains the optimal solution fast.
- In Section 3.2, we add the fixed cost of the connections in the model. In the computer science literature, this is known as the *capacitated single-commodity fixed charge network flow problem.* It is part of the larger category of fixed charge network flow problems which have been studied by researchers in multiple forms: capacitated or uncapacitated, single-commodity or multicommodity, single source or multiple sources. Most solution approaches utilize branch-and-bound techniques to find an exact solution ([3, 14, 17, 20, 21, 25, 29, 30]). However, finding an exact solution is computationally expensive and is infeasible for large instances, so heuristics have also been proposed in order to find fast reasonable good solutions ([6, 9, 13, 22–24, 27]). We propose three different optimization techniques in order to solve it: a branch-and-bound algorithm, a simulated annealing approach and a heuristic based on the approximation of the fixed cost using a modified cost per unit of flow for each edge.

3.1 Minimum-cost maximum-flow problem with edge demands (MCMFP-ED)

The network flow model we use has the following properties:

- There are n nodes in the network $G = (V, E)$, divided into three categories: producers, substations, consumers. For each node u, we know its demand D_u, which is negative for producers, zero for substations and positive for consumers. Each demand has an associated cost C_u, representing the cost of producing / consuming one unit of flow.
- The nodes are interconnected with m directed edges. An edge $e = (u, v)$ connects two nodes and has the following characteristics:
  - Minimum capacity $d(u, v)$ (the minimum amount of flow which has to be sent on edge (u, v)), Maximum capacity $c(u, v)$ (the maximum amount of flow which can be sent on edge (u, v)). These represent capacity constraints.
  - Cost per unit of flow $cost(u, v)$, i.e. the cost of sending one unit of flow across edge (u, v). This cost can be determined by the type and the length of the cable used.
  - Flow $f(u, v)$, i.e. the amount of current sent across edge (u, v).

  The model allows multiple edges between two nodes, in the same or opposite directions. The cost of the connections can also be negative, but no negative-weighted cycles are allowed.
- The amount of flow leaving a node must be equal to the amount of flow entering the node minus the (partial) demand of that node (flow conservation constraint).
- The flow of the network is equal to the sum of the flows of the edges leaving from the producer nodes. Let F be the maximum flow the network can support while satisfying all the constraints (capacity and flow conservation constraints).

While the flow $f(u, v)$ for each edge $(u, v) \in E$ (and implicitly F as well) is a computed value and represents the solution of the problem, all the other data is given as input.

In order to have a single source (producer) and a single sink (consumer) we will artificially modify the network as follows:

- Add node s, representing the source. We connect this node to each of the initial producers (nodes $u \in V$ with $D_u < 0$). Each such edge (s, u) will have $d(s, u) = 0, c(s, u) = -D_u$ and $cost(s, u) = C_u$.
- Add node t, representing the sink. We connect all the initial consumers (nodes $u \in V$ with $D_u > 0$) to t. Each such edge (u, t) will have $d(u, t) = 0, c(u, t) = D_u$ and $cost(u, t) = C_u$. Allowing negative weighted edges in the graph is needed for these edges, as the value C_u for the demand nodes should be negative, representing the cost that consumers pay for the each unit of flow they receive. Having different values of C_u for different customers can be useful when there is not enough supply to satisfy all the demand. In this situation, based on the cost they are able to

pay, and the layout and cost of the connections in the network, we can decide to which customers it is optimal to send the flow to.

We introduce the following notations:

- $N(u) = \{v \in V, \text{ s.t. } (u,v) \in E\}$, the set of *successors* of node u.
- $M(u) = \{v \in V, \text{ s.t. } (v,u) \in E\}$, the set of *predecessors* of node u.

To compute the maximum flow F of the network, we can solve the following linear program:

$$
\begin{aligned}
\text{maximize} \quad & \sum_{u \in N(s)} f(s,u) & & \\
\text{subject to} \quad & d(u,v) \leq f(u,v) \leq c(u,v), & \forall (u,v) \in E \quad & (1) \\
& \sum_{i \in M(u)} f(i,u) = \sum_{j \in N(u)} f(u,j), & \forall u \in V, u \neq s, u \neq t \quad & (2)
\end{aligned}
$$

The goal of the LP formulation is to maximize the flow on the edges leaving the source. The first constraint ensures that the amount of flow on each edge respects the minimum and the maximum capacity of that edge. The second constraint makes sure that flow conservation is achieved, i.e. the amount of flow entering a node equals the amount of flow leaving that node (for all nodes except for the source s and the sink t).

Once we computed F, we can find the most cost-efficient solution with that maximum flow using the following linear programming formulation:

$$
\begin{aligned}
\text{minimize} \quad & \sum_{(u,v) \in E} cost(u,v) f(u,v) & & \\
\text{subject to} \quad & d(u,v) \leq f(u,v) \leq c(u,v), & \forall (u,v) \in E \quad & (1) \\
& \sum_{i \in M(u)} f(i,u) = \sum_{j \in N(u)} f(u,j), & \forall u \in V, u \neq s, u \neq t \quad & (2) \\
& \sum_{u \in N(s)} f(s,u) = F & & (3)
\end{aligned}
$$

While the first and second constraints are the same as in the previous LP formulation, the last constraint ensures that the flow in the network is maximum, as we want to find the minimum cost flow while still having the highest possible flow in the network.

Instead of using one of the available LP solvers, we have chosen a graph theory based approach for solving the MCMFP-ED (Minimum-cost maximum-flow problem with edge demands):

- Find a feasible flow in the network, i.e. a flow which satisfies the first two constraints of the LP formulation (inspired by the lecture notes of Erickson [12]).

- Augment the feasible flow such that the flow is maximized and the cost is minimized.

The reason why we use a two-step approach is that we first want to eliminate the minimum capacity constraints in order to simplify the problem. Once we have found a feasible flow, we can augment the flow in the network to obtain the maximum one.

3.1.1 Identifying a feasible flow in the network.

In order to find a feasible flow in network G (or determine that none exists[1]), we will create a new network $G^{'}(V^{'}, E^{'})$ and apply one of the standard maximum flow algorithms. $G^{'}$ is obtained as follows:

- Add all nodes $u \in V$.
- Add a new source $s^{'}$ and a new sink $t^{'}$.
- For each edge $e = (u, v) \in E$:
  - Add an edge $e^{'} = (u, v)$ with capacity $c^{'}(u, v) = c(u, v) - d(u, v)$. This edge is the equivalent in $G^{'}$ of edge e in G.
  - Add an edge $\left(s^{'}, v\right)$ with capacity $c^{'}(s^{'}, v) = d(u, v)$.
  - Add an edge $\left(u, t^{'}\right)$ with capacity $c^{'}(u, t^{'}) = d(u, v)$.
- Add a new edge (t, s) with infinite capacity.

If $G^{'}$ contains multiple edges from the new source $s^{'}$ to another node u, we can merge them all into a single edge with capacity equal to the sum of the capacities of the merged edges. The same happens for the edges from any one node u to the new sink $t^{'}$.

Definition 1. *An (s, t)-flow f is a saturating flow if all edges leaving s and all edges entering t are saturated, i.e. $\forall v \in N(s), f(s, v) = c(s, v)$ and $\forall u \in M(t), f(u, t) = c(u, t)$.*

Lemma 1. *G has a feasible (s, t)-flow f if and only if $G^{'}$ has a saturating $\left(s^{'}, t^{'}\right)$-flow $f^{'}$, where $f(u, v) = f^{'}(u, v) + d(u, v), \forall(u, v) \in E$.*

Proof. We will only prove the right-to-left implication of the lemma.

The flow f is feasible if it satisfies two constraints:

$$d(u, v) \leq f(u, v) \leq c(u, v), \qquad \forall(u, v) \in E \tag{1}$$

$$\sum_{i \in M(u)} f(i, u) = \sum_{j \in N(u)} f(u, j), \qquad \forall u \in V, u \neq s, u \neq t \tag{2}$$

[1] An example in which no feasible flow exists: edge (u, v) has a minimum capacity $d(u, v)$ and the sum of the maximum capacities of all edges leaving from v is less than $d(u, v)$, i.e. $d(u, v) > \sum_{i \in N(v)} c(v, i)$.

The first contraint is satisfied as $\forall e = (u, v) \in E$ we have:

$$0 \leq f^{'}(u, v) \leq c^{'}(u, v) \Rightarrow$$
$$0 \leq f(u, v) - d(u, v) \leq c(u, v) - d(u, v) \Rightarrow$$
$$d(u, v) \leq f(u, v) \leq c(u, v).$$

The second constraint is satisfied as well, as for any node $u \in V$ such that $u \neq s$ and $u \neq t$ we have[1]:

$$\sum_{i \in M^{'}(u)} f^{'}(i, u) = \sum_{j \in N^{'}(u)} f^{'}(u, j) \Rightarrow$$
$$\sum_{i \in M(u)} (f^{'}(i, u)) + f^{'}(s^{'}, u) = \sum_{j \in N(u)} (f^{'}(u, j)) + f^{'}(u, t^{'}) \Rightarrow \text{(as } f^{'} \text{ is saturating)}$$
$$\sum_{i \in M(u)} (f^{'}(i, u)) + c^{'}(s^{'}, u) = \sum_{j \in N(u)} (f^{'}(u, j)) + c^{'}(u, t^{'}) \Rightarrow$$
$$\sum_{i \in M(u)} f^{'}(i, u) + \sum_{i \in M(u)} d(i, u) = \sum_{j \in N(u)} f^{'}(u, j) + \sum_{j \in N(u)} d(u, j) \Rightarrow$$
$$\sum_{i \in M(u)} f(i, u) = \sum_{j \in N(u)} f(u, j).$$

□

We have used the Edmonds-Karp method (described in Appendix C) to find a saturating flow in $G^{'}$. Once we have identified a saturating flow, we can obtain the feasible flow in G by adding to the flow of each edge the minimum capacity of that edge. Figure 3.1 illustrates all the steps involved in the process of finding a feasible flow for a network.

When building the residual network[2] G_f of the feasible flow f, it is important to take note of a slight modification in the capacity of the residual edges:

$$c_f(u, v) = \begin{cases} c(u, v) - f(u, v), & \text{if } (u, v) \in E. \\ f(v, u) - d(v, u), & \text{if } (v, u) \in E\text{; in this case the edge } (u, v) \text{ in } G_f \text{ is called "return edge".} \\ 0, & \text{otherwise.} \end{cases}$$

For any edge $(u, v) \in E$ we cannot *return* $f(u, v)$ flow as that will cause the edge to violate the minimum capacity constraint if $d(u, v) > 0$. Instead, we allow that only $f(u, v) - d(u, v)$ flow can be *returned* on that edge.

[1] $M^{'} = M \cup \{s^{'}\}$, $N^{'} = N \cup \{t^{'}\}$.

[2] The residual network is defined in the Edmonds-Karp algorithm described in Appendix C

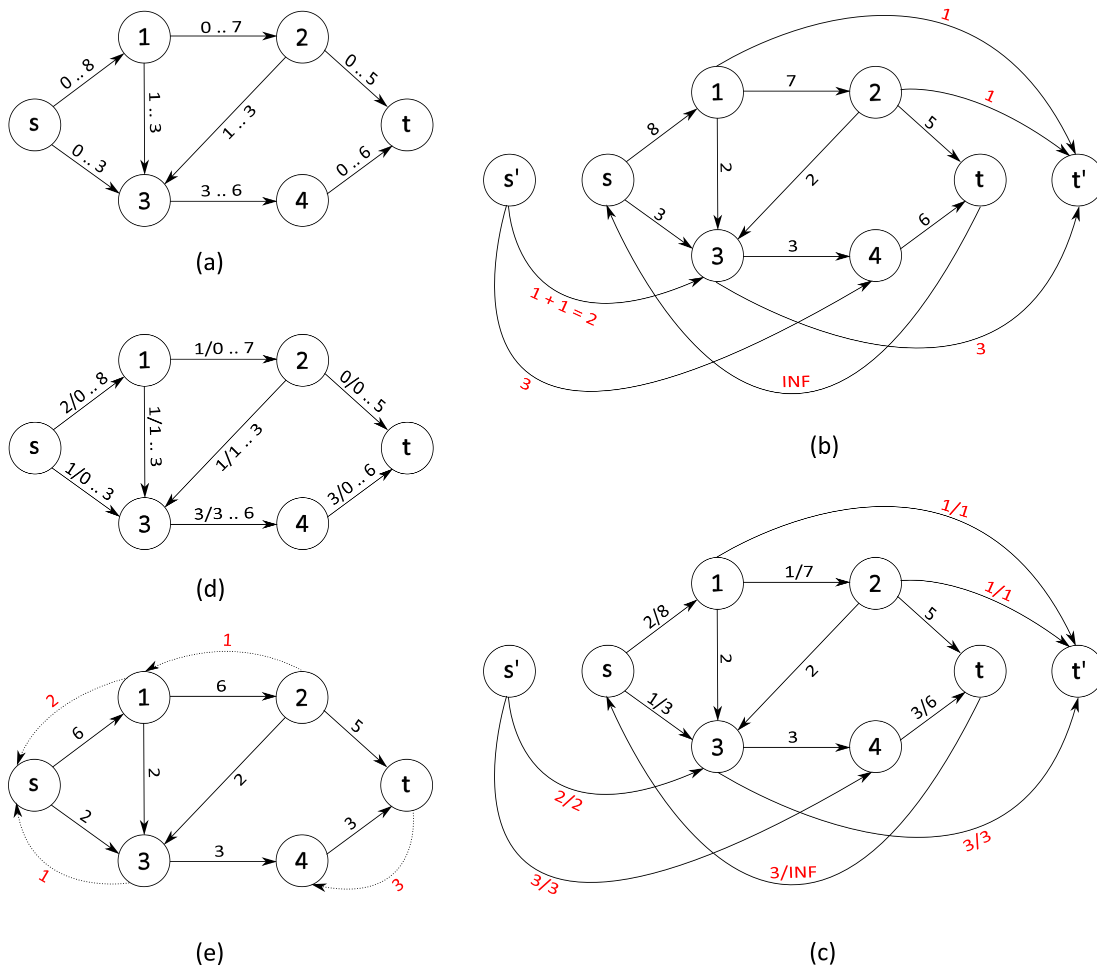

Figure 3.1. a) The initial network G; the edge capacities are given as $d..c$. **b)** The new network $G^{'}$. **c)** A saturating $\left(s^{'}, t^{'}\right)$-flow $f^{'}$ in $G^{'}$. **d)** A feasible (s, t)-flow f in G. **e)** The residual network G_f for the feasible flow f.

3.1.2 Finding the maximum flow with the minimum cost in the network.

Once we have computed a feasible flow f for the initial network G, we can use the residual network G_f as a starting point for finding the maximum flow with the minimum cost. If we would only be interested in finding the maximum flow of the network, we could apply the Edmonds-Karp method just like we did for finding a saturating flow in $G^{'}$. However, we want to find the minimum cost solution with the maximum flow. The first step is to assign costs to the edges in the residual network:

$$cost_f(u,v) = \begin{cases} cost(u,v), & \text{if } (u,v) \in E. \\ -cost(v,u), & \text{if } (v,u) \in E. \\ 0, & \text{otherwise.} \end{cases}$$

The method we have implemented for finding the maximum flow with the minimum cost is the *Successive Shortest Path Algorithm* (described by Ahuja et al. [2]). This algorithm

searches for the maximum flow and optimizes the objective function simultaneously. The initial network G can have edges with negative costs, but we assume that it has no negative-weighted cycles (with regard to the cost), as then we cannot say exactly what the "shortest path" is. At each iteration, we will find the shortest path (minimum cost path) from s to t in the residual network G_f and augment that path. The algorithm terminates when the residual network contains no such path, i.e. the flow is maximal. By searching for the shortest path at each iteration, we guarantee that the residual network will not contain any negative-weighted cycles (according to Lemma 2) which means that the final solution will be optimal (Lemma 3).

Lemma 2. *If a residual network G_f has no negative-weighted cycles, augmenting the flow along the shortest path from the source s to the sink t will not induce any negative-weighted cycles in G_f.*

Proof. (Refer to Figure 3.2.)
Suppose that after augmenting the flow along the shortest path P from s to t, a negative-weighted cycle Q appears in the residual network. As before augmenting, G_f did not have any negative-weighted cycles, it means that there was a subpath $(i \cdots j)$ in P the reversal of which closed cycle Q after the augmentation. Q is a negative-costed cycle, so:

$$cost(j \cdots i) + cost(Q \setminus (j \cdots i)) < 0 \Leftrightarrow$$
$$cost(Q \setminus (j \cdots i)) < cost(i \cdots j)$$

This implies that we could find another path from s to t, which goes from s to i, then from i to j along the edges of Q, then from j to t which has a lower cost than path P. We have a contradiction to the assumption that P is the shortest path from s to t.

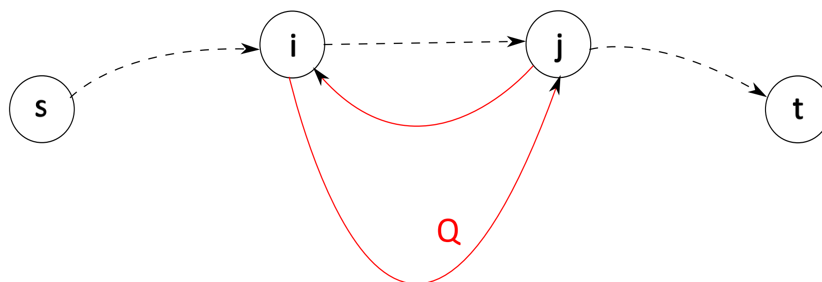

Figure 3.2. The dotted black line represents the shortest path P from source s to sink t. The red line represents a possible cycle Q formed after the augmentation of flow along the edges of P.

□

Lemma 3. *Let f be a feasible solution of a minimum cost flow problem. Then f is an optimal solution if and only if the residual network G_f contains no negative-weighted cycle.*
(Proof by Ahuja et al. [2]).

As we didn't consider costs when identifying a feasible flow for G, it is possible that the residual network G_f has negative-weighted cycles. Before applying the successive shortest path algorithm we want to eliminate these cycles from the residual network. For

identifying such cycles we have used the Bellman-Ford algorithm described in Appendix B. Using this algorithm we either find a negative-weighted cycle or determine that no such cycle exists in G_f. When we find a negative-weighted cycle, if it contains no *return* edge, then that cycle also exists in the initial network G, which makes it unfeasible for our algorithm. However, if the cycle contains a *return* edge, then we will augment the cycle with the minimum capacity along the edges of that cycle and apply the Bellman-Ford algorithm again, until no negative-weighted cycles are found. Figure 3.3 illustrates the concept of cycle-canceling on a simple example.

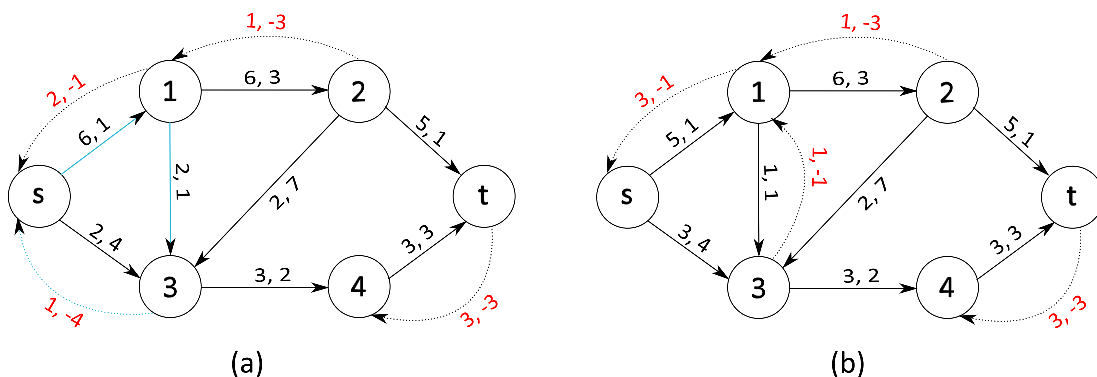

Figure 3.3. **a)** The residual network G_f of the feasible flow f; the edge descriptors indicate its capacity and cost: $c, cost$. There exists a negative-weighted cycle $s-1-3-s$ which can be augmented with 1 flow (the minimum capacity along the edges of the cycle; in this case the capacity of the *return* edge $3-s$). **b)** The residual network G_f after the negative-weighted cycle has been augmented. Now it doesn't contain any negative-weighted cycles.

After all the negative-weighted cycles in the residual network G_f have been eliminated we need to find the shortest augmenting paths from s to t. As the network can have negative-weighted edges we cannot apply Dijkstra's algorithm, but instead we could use the Bellman-Ford algorithm. The disadvantage is that Bellman-Ford has a larger running time than Dijkstra. However, we can apply a reweighing technique described by Cormen et al. [7, p. 701] which assigns only positive weights to all the edges without influencing the optimal solution. By having all positive-weighted edges we can then apply Dijkstra's algorithm and obtain a better overall time complexity.

Positive edge reweighing

Lemma 4. *Given a weighted, directed graph $G(V, E)$ with weight function $cost : E \to \mathbb{R}$, let $\pi : V \to \mathbb{R}$ be any function mapping nodes to real numbers. We call $\pi(u)$ the potential of node u. For each edge $(u, v) \in E$ we can define a new weight function:*
$cost^\pi(u, v) = cost(u, v) + \pi(u) - \pi(v)$.
The optimal solution for the minimum-cost maximum-flow problem for graph G will be the same for either of the two weighting functions.

Proof. We start by showing that:
$cost^\pi(p) = cost(p) + \pi(u_0) - \pi(u_k)$, where $p = \langle u_0, u_1, \cdots, u_k \rangle$ is a path from node u_0 to node u_k.

We have:

$$\begin{aligned} cost^{\pi}(p) &= \sum_{i=1}^{k} cost^{\pi}(u_{i-1}, u_i) \\ &= \sum_{i=1}^{k}(cost(u_{i-1}, u_i) + \pi(u_{i-1}) + \pi(u_i)) \\ &= \sum_{i=1}^{k} cost(u_{i-1}, u_i) + \pi(u_0) - \pi(u_k) \text{ (because the sum telescopes)} \\ &= cost(p) + \pi(u_0) - \pi(u_k). \end{aligned}$$

We will now show that the optimal solution is not influenced by the new weighting function. We know that an $(s - t)$ flow can be decomposed into multiple augmenting paths $p_1, p_2, \cdots, p_l$ each path i sending $f(p_i)$ flow along that path. Therefore:

$$\begin{aligned} \sum_{(u,v)\in E} cost^{\pi}(u, v)f(u, v) &= \sum_{i=1}^{l} cost^{\pi}(p_i)f(p_i) \\ &= \sum_{i=1}^{l} f(p_i)(cost(p_i) + \pi(s) - \pi(t)) \\ &= \sum_{i=1}^{l} cost(p_i)f(p_i) + (\pi(s) - \pi(t))\sum_{i=1}^{l} f(p_i) \\ &= \sum_{(u,v)\in E} cost(u, v)f(u, v) + (\pi(s) - \pi(t))F. \end{aligned}$$

For fixed node potentials π, the difference in the objective function is given by a constant value $(\pi(s)-\pi(t))F$. Therefore, a flow which minimizes the objective function for some new weights $cost^{\pi}$ will also minimize it for the original weights *cost*. □

Lemma 5. *If we assign the potential of a node $\pi(u)$ to be equal to the shortest distance (in terms of cost) from s to u in a graph $G(V, E), \forall u \in V$, then $cost^{\pi}(u, v) \geq 0, \forall (u, v) \in E$.*

Proof. We know that $\pi(v) \leq cost(u, v) + \pi(u)$, because otherwise the shortest distance from s to v could be improved by going through node u and using edge (u, v) to reach node v. This means that: $cost(u, v) + \pi(u) - \pi(v) \geq 0 \Rightarrow cost^{\pi}(u, v) \geq 0$. □

Successive shortest path algorithm

Once all the edges in the network have a positive weight we can successfully apply Dijkstra's algorithm to find the shortest paths from the source s to all other nodes. If there exists a shortest path P from s to t in the residual network, then it is an augmenting path and we can increase the flow on the edges of P. As we are augmenting the path, we need to add *return* edges which will have $cost(v, u) = -cost(u, v), \forall (u, v) \in$

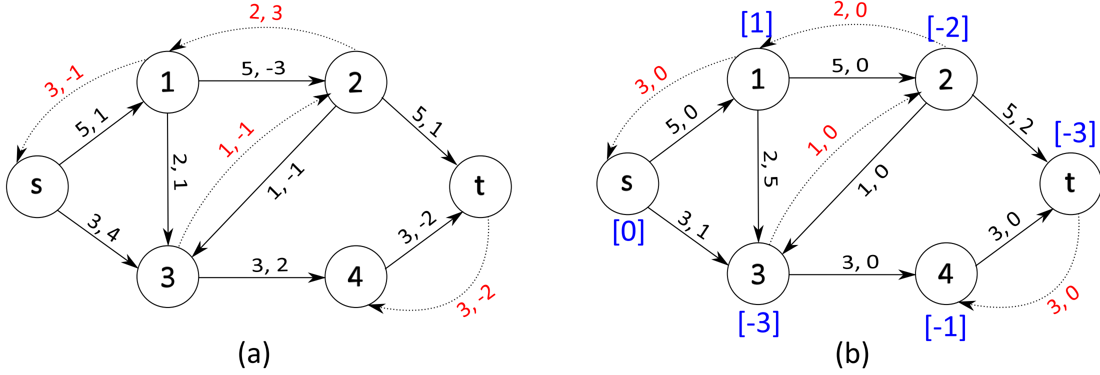

Figure 3.4. The positive edge reweighing process. **a)** The initial network. **b)** The edges of the network have been reweighed such that all have positive values; each node u has its potential displayed as $[\pi_u]$

.

P. However, if we update the potentials π with the new values of the shortest paths obtained by the latest Dijkstra run, we can easily prove that the return edges will have a weight of 0.

Lemma 6. *Any return edge in the residual network G_f obtained by augmenting the flow on the shortest path from s to t will have a weight of 0, as long as we use the node potentials $\pi(u), \forall u \in V$, where $\pi(u)$ is equal to the length of the shortest path from s to u.*

Proof. We assume without loss of generality that we want to augment edge $(u, v) \in P$, where P is the shortest path from s to t in the residual network G_f. As the edge (u, v) is on the shortest path from s to v then we know that:

$$\pi(v) = \pi(u) + cost(u, v).$$

The return edge (v, u) in the residual network will have the weight:

$$\begin{aligned} cost^\pi(v, u) &= -cost^\pi(u, v) \\ &= -(cost(u, v) + \pi(u) - \pi(v)) \\ &= \pi(v) - (\pi(u) + cost(u, v)) \\ &= 0 \end{aligned}$$

□

We can summarize the Successive shortest path algorithm used to compute the minimum-cost maximum-flow in a network with the steps described in Algorithm 1.

We use Bellman-Ford's algorithm only once to initialize the node potentials. It takes $O(|V|\,|E|)$ time. Then, the loop on line 3 has at most F iterations, as at every step we increase the maximum flow in the network by at least 1. The complexity of one loop iteration is given by Dijkstra's algorithm, which takes $O(|V|^2)$ time. Summing up, we obtain a complexity of $O(|V|\,|E| + F\,|V|^2)$ which can be further improved if better implementations are used for Dijkstra's algorithm.

Algorithm 1 Successive shortest path

Require: G_f - the residual network of a feasible flow f in G containing no negative-weighted cycles
1: Use Bellman-Ford's algorithm to establish the initial potentials π.
2: Reweigh the edges of G_f based on the potentials π.
3: **while** G_f contains a path from s to t **do**
4: Find the shortest path P from s to t in G_f using Dijkstra's algorithm.
5: Update the node potentials π with the new shortest path values.
6: Reweigh the edges of G_f based on the new potentials π.
7: Augment path P.

3.2 Minimum-cost maximum-flow problem with fixed cost on edges (MCMFP-FC)

One of the limitations of the previously described model is that the cost associated with each edge is defined as the *cost per flow*, i.e. the amount we pay for sending one unit of flow on that edge. However, this is not very realistic for electrical networks, where the highest cost for a connection (edge) is a fixed cost determined by the length of the connection (digging costs) and the type of the connection (equipment costs). The new problem can be formulated as an integer linear program (ILP):

$$\text{minimize} \sum_{(u,v)\in E} (cost(u,v)f(u,v) + fixedCost(u,v)y(u,v))$$

$$\text{subject to } d(u,v) \leq f(u,v) \leq c(u,v)y(u,v), \qquad \forall (u,v) \in E \qquad (1)$$

$$\sum_{i\in M(u)} f(i,u) = \sum_{j\in N(u)} f(u,j), \qquad \forall u \in V, u \neq s, u \neq t \qquad (2)$$

$$\sum_{u\in N(s)} f(s,u) = F \qquad (3)$$

$$y(u,v) \in \{0,1\}, \qquad \forall (u,v) \in E \qquad (4)$$

We can notice the addition of the binary variable $y(u,v)$ which indicates if an edge $(u,v) \in E$ is used or not in the maximum flow solution. If the flow $f(u,v)$ is strictly positive, $y(u,v)$ will be 1 and $fixedCost(u,v)$ will be payed for using edge (u,v).

Lemma 7. *MCMFP-FC (Minimum-cost maximum-flow problem with fixed cost on edges) is NP-hard.*

Proof. Lemma 7 has been proven by Guisewite et. al [18] for a generalized version of MCMFP-FC (where the cost of an edge can be any concave function). We propose a proof for our more specific problem (MCMFP-FC), based on a reduction from the directed Steiner tree problem which has been proven to be NP-hard by Garey and Johnson [15]. The directed Steiner tree problem is the following: given a directed graph

$G = (V, E)$ with weights $w(u, v)$ on edges, a set of terminals $S \subseteq V$, and a root vertex r, find a minimum weight out-branching T rooted at r such that all vertices in S are included in T. We will construct our MCMFP-FC instance $G^{'}$ as follows:

- $G^{'} = (V, E), D_u \leftarrow 0, C_u \leftarrow 0, \forall u \in V$
- $D_r \leftarrow -|S|$
- $D_u \leftarrow D_u + 1, \forall u \in S$ (note that defining $D_u \leftarrow 1, \forall u \in S$ is not correct if $r \in S$)
- $d(u, v) \leftarrow 0, c(u, v) \leftarrow \infty, \forall (u, v) \in E$
- $cost(u, v) \leftarrow 0, fixedCost(u, v) \leftarrow w(u, v), \forall (u, v) \in E$

It is easy to see that the instance of the directed Steiner tree has a solution if and only if there is a solution of the MCMFP-FC applied on $G^{'}$. Figure 3.5 illustrates the above-mentioned reduction on a simple example. □

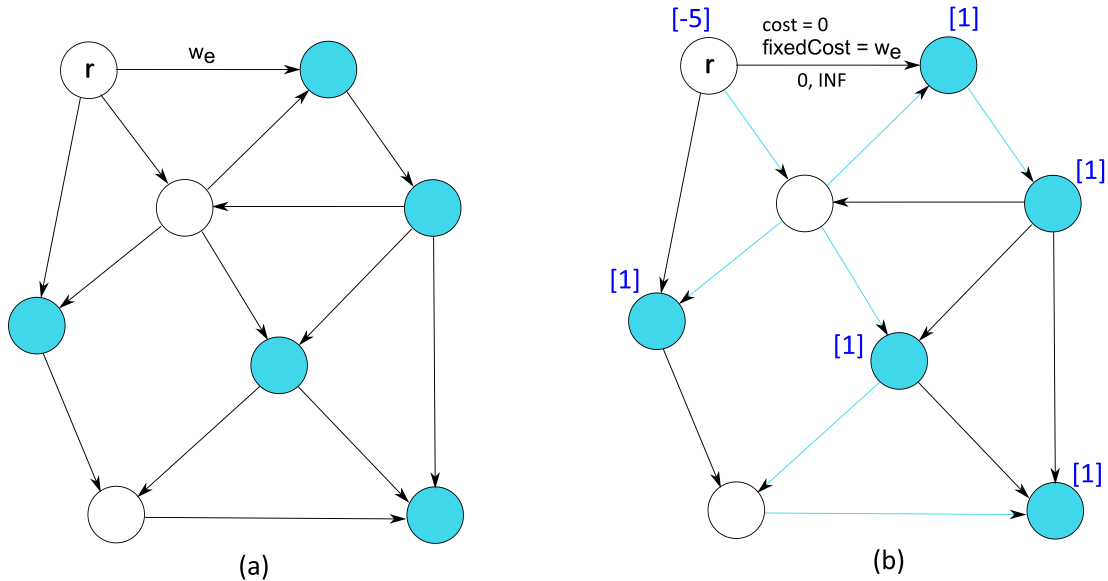

Figure 3.5. An example of the reduction from the directed Steiner tree problem to the MCMFP-FC. **a)** The directed Steiner tree problem instance G; colored nodes represent subset S. **b)** The MCMFP-FC instance $G^{'}$; the supply / demand of the producer / consumers is displayed in square brackets; the colored edges represent a possible solution.

We have implemented and analyzed three different approaches for solving the MCMFP-FC:

- A branch-and-bound algorithm which computes an optimal solution.
- A simulated annealing heuristic.
- A cost-function slope heuristic.

While similar branch-and-bound algorithms have already been used in the literature for solving the MCMFP-FC, the other two methods represent new approaches to the problem.

3.2.1 Branch-and-bound algorithm (BB)

Branch and bound[1] is an algorithm used for finding optimal solutions of optimization problems. It works by analyzing all candidate solutions, but discarding worthless candidate subsets, by using upper and lower estimated bounds of the function being optimized. The *branching* step of the algorithm consists in splitting a set of candidates S into two or more smaller subsets $S_1, S_2, \cdots$ whose union covers S. The recursive application of branching creates a tree in which for every node *bounding* is applied to find an estimated lower and upper bound for that particular subset of S. The key idea of the BB algorithm is the *pruning* process: for the minimization variant of the BB (which we use for MCMFP-FC), if a *lower* bound of a tree node S_1 is greater than the *upper* bound of some other node S_2, then all the candidate solutions in S_1 can be discarded, as they are all worse than some solution in S_2. The algorithm ends when the upper bound of the set of candidates S equals the lower bound of S.

In order to solve MCMFP-FC using the BB method we need to define the branching and bounding steps of the algorithm.

Branching

The complexity of MCMFP-FC is determined by the existence of the fixed cost of the edges. If for a particular graph instance $G = (V, E)$, we fix the values $y(u, v)$ defined in the ILP formulation of the problem, we obtain an LP and MCMFP-FC can be solved by adding the fixed cost of the edges $(u, v) \in E$ with $y(u, v) = 1$ to the solution of MCMFP-ED applied on a graph G' derived from G such that:

- All edges $(u, v) \in E$ with $y(u, v) = 0$ will have their capacity $c(u, v)$ set to 0 to prevent them from taking part in the solution. Of course, edges with minimum capacity $d(u, v) > 0$ will not be able to have $y(u, v) = 0$.
- All edges $(u, v) \in E$ with $y(u, v) = 1$ will have their minimum capacity $d(u, v)$ set to 1 unless they already have a strictly positive value (if they already have $d(u, v) > 0$ then the edge will be taken in the solution anyway).

Therefore, the initial set S of candidate solutions of the BB algorithm consists of all the possible valid combinations of values for the binary variables $y(u, v)$. We define S as the sequence $(y_1, y_2, \cdots, y_{|E|}) = (*_1, *_2, \cdots, *_{|E|})$ where $*_e$ can be either 0 or 1. We can split S into two subsets $S_1 \cup S_2 = S$ by fixing the value of $*_1$ to 0 or 1[2]. Each of these subsets can further be split into two other subsets by fixing the value of $*_2$. The splitting continues until the set contains only one candidate, i.e. all values $*_e$ have been fixed. However, there is an exception for this rule: if a node in the branching tree is pruned or it contains no feasible solution then no branching will occur for that node. Figure 3.6 illustrates a complete branching process for a graph with 3 edges.

[1] Fore a more detailed description of the main principles of BB, I recommend the work of Clausen [5].

[2] For edges e with $d_e > 0$, $*_e$ cannot take the value 0.

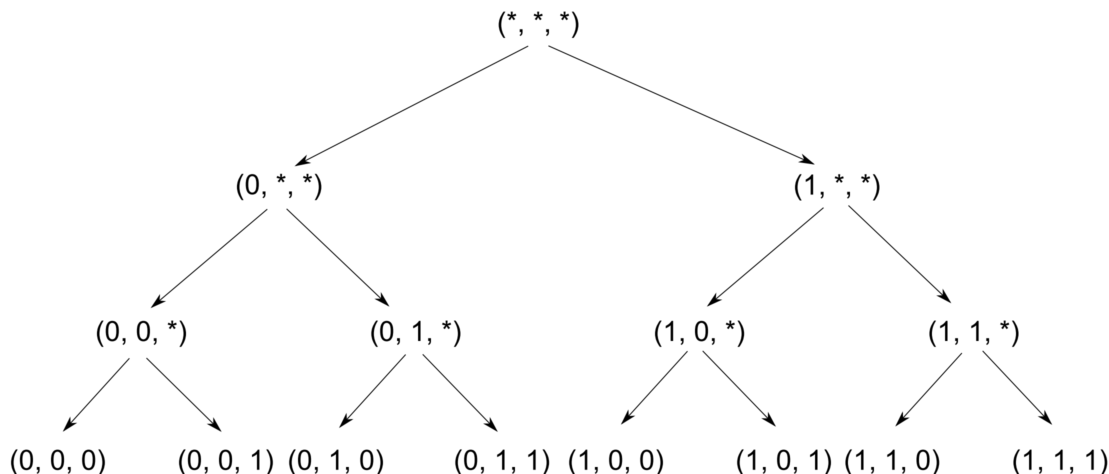

Figure 3.6. An example of branching for a graph with 3 edges and no minimum capacities. However, some of the branches could be *pruned* depending on their lower and upper bound values.

Bounding

For each node in the branching tree, we need to compute the lower and upper bounds for the subset of candidate solutions associated with that node. A node t placed on level l in the branching tree (the root has level 0) will be characterized by the sequence $S_t = (y_1, y_2, \cdots, y_l, *_{l+1}, *_{l+2}, \cdots, *_{|E|})$, where the values of y_e are fixed to either 0 or 1.

The lower bound L_t of node t represents the minimum value that a candidate solution matching the sequence S_t could take, i.e. there is no candidate solution in the subset of candidates of node t that can be better than L_t. We start by constructing graph G' based on the fixed values $y_1, y_2, \cdots, y_l$ from the sequence S_t. For the rest of the edges for which we don't know whether they should or shouldn't be in the maximum flow solution, we will define a new cost per flow as $cost'(u,v) = cost(u,v) + \frac{fixedCost(u,v)}{c(u,v)}$ and a new fixed cost as $fixedCost'(u,v) = 0$ (example illustrated in figure 3.7).

Lemma 8. *The cost of sending flow on an edge (u,v) with $cost'(u,v) = cost(u,v) + \frac{fixedCost(u,v)}{c(u,v)}$ and $fixedCost'(u,v) = 0$, will always be less or equal than the original cost of sending flow on that edge (valid for any amount of flow).*

Proof. We want to show that for any edge (u,v), s.t. $cost'(u,v) = cost(u,v) + \frac{fixedCost(u,v)}{c(u,v)}$ and $fixedCost'(u,v) = 0$, the following inequality holds:

$$cost'(u,v)f(u,v) + fixedCost'(u,v)y(u,v) \leq cost(u,v)f(u,v) + fixedCost(u,v)y(u,v).$$

To prove the inequality we simply replace the terms in the left-hand side with their

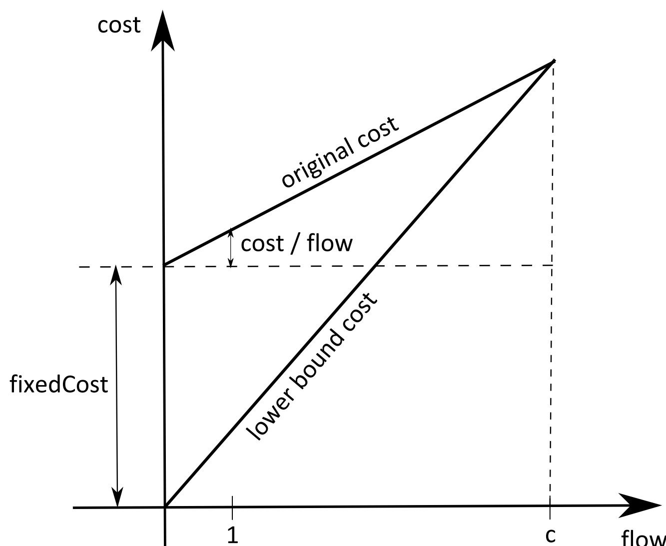

Figure 3.7. The new lower bound cost is lower than the real cost for any valid value of the flow.

definition and use the capacity contraint inequality $f(u,v) \leq c(u,v)y(u,v)$:

$$cost^{'}(u,v)f(u,v) + fixedCost^{'}(u,v)y(u,v) =$$
$$cost^{'}(u,v)f(u,v) =$$
$$cost(u,v)f(u,v) + \frac{fixedCost(u,v)}{c(u,v)}f(u,v) \leq$$
$$cost(u,v)f(u,v) + fixedCost(u,v)y(u,v).$$

□

To compute the value of the lower bound L, we apply the MCMFP-ED algorithm on $G^{'}$ and add the fixed cost of the edges $(u,v) \in E$ with $y(u,v) = 1$ (*fixed* edges).

Lemma 9. *L is a lower bound for the optimal solution.*

Proof. Let f be the maximum flow of the optimal solution. f will also be a valid maximum flow in $G^{'}$ (as only costs differ). The cost OPT of the optimal solution is equal to the sum of the cost of sending the flow $f(u,v)$ on each edge $(u,v) \in E$. By Lemma 8, we know that each such cost will be higher than or equal to the cost of sending the same flow in $G^{'}$. This means that the cost COST_f of sending the flow f in $G^{'}$ will be less or equal than OPT. As L represents the optimal solution in $G^{'}$ from all the possible maximum flows, it will be less or equal than COST_f, which is less of equal than OPT. Therefore L is a lower bound for the optimal solution. □

The upper bound U_t of node t simply represents a feasible solution matching the sequence S_t. We construct the same graph $G^{'}$ as for the lower bound case, however for edges which have not been fixed we do not modify any costs. After we apply the MCMFP-ED algorithm on $G^{'}$ we add the fixed cost of all the edges which have a strictly positive flow. This represents a feasible solution, but not necessarily the optimal one.

Pruning

Pruning is important in order to limit the number of nodes which are analyzed. The runtime of the BB algorithm is directly proportional to the amount of pruning which can be done. For each node t in the branching tree, after we compute the upper bound U_t, we update the global upper bound U, which represents the best feasible solution found so far. If the lower bound L_t of a node t is higher than or equal to the global upper bound, that means that we have already found a feasible solution which can not be improved by any solution which matches the pattern of the sequence S_t, so we can prune the subtree rooted in t. Unfortunately, it is possible that no pruning occurs, which means that the BB algorithm will explore the whole branching tree which has $2^{|E|+1} - 1$ nodes, yielding an exponential running time.

Algorithm and optimizations

An overview of the BB algorithm we have implemented:

Algorithm 2 BB algorithm for MCMFP-FC

Require: $G = (V, E)$ - the network for which we want to find the minimum-cost maximum-flow solution.

1: $root \leftarrow (*_1, *_2, \cdots, *_{|E|})$ ▷ The sequence of the root of the branching tree.
2: $L \leftarrow L_{root}$ ▷ The global lower bound.
3: $U \leftarrow U_{root}$ ▷ The global upper bound.
4: $Q \leftarrow \{root\}$ ▷ The set of unexplored nodes - initially contains only the root.
5: **while** $L < U$ **do**
6: $t \leftarrow$ the node with the minimum lower bound among the nodes in Q
7: $l \leftarrow level(t)$ ▷ The level of node t in the branching tree.
8: $Q \leftarrow Q \setminus t$
9: Branch t into t_0 and t_1 by fixing y_l to 0 respectively 1.
10: Compute $L_{t_0}, U_{t_0}, L_{t_1}, U_{t_1}$.
11: $Q \leftarrow Q \cup \{t_0, t_1\}$
12: Update L and U.
13: Prune Q.

Given the exponential running time of the BB algorithm it is important to pay attention to various implementation details. For this particular BB algorithm (applied for MCMFP-FC) there are several possible optimizations related to improving both the memory usage as well as the running time.

Optimization #1 The first optimization minimizes the memory usage of the nodes in the branching tree. For each such node t, we need to store the sequence $S_t = (y_1, y_2, \cdots, y_l, *_{l+1}, *_{l+2}, \cdots, *_{|E|})$, where l is the level of node t. The first observation is that we only need to store the values of y_i for $i \leq l$. Let $P(t)$ be the parent of node t in the branching tree and $t^{'}$ its other offspring. $t^{'}$ will be characterized by the sequence $S_{t'} = (y_1, y_2, \cdots, \overline{y_l}, *_{l+1}, *_{l+2}, \cdots, *_{|E|})$. The sequences $S_{P(t)}$, S_t and $S_{t'}$ have the common subsequence $y_1, y_2, \cdots, y_{l-1}$, the only difference between them is the value

of y_l. Therefore, if we know the sequence $S_{P(t)}$ we only need to store the value of y_l for nodes t and t'. The same reasoning can be made to show that for $P(t)$ we only need to store the value of y_{l-1}. By recursion, each node in the branching tree will only store the value of y_l, where l is their level in the tree. So, if for a node t we want to determine S_t, we can simply rebuild it by going from node t up in the branching tree up to the root. This way, for each node in the tree we only need to store one value, instead of $|E|$.

Optimization #2 The second optimization illustrates a tradeoff between memory usage and running time. The pruning process in the BB algorithm 2 occurs at every iteration and consists in going through all the nodes stored in Q and check if their lower bounds are higher than the global upper bound. Going through all the nodes in Q at every iteration ensures that there will not be any nodes stored which cannot improve the solution, however this is done at the cost of a higher running time. By only pruning when evaluating a node, we get a better run time but a higher memory usage as more nodes will be stored throughout the algorithm. We want to avoid extremes (higher memory usage or high run time) so we chose to apply the pruning only at every k iterations (in our experiments $k = 1000$). This way we get a better run time but still have a moderate upper bound for the memory used.

Optimization #3 The third optimization consists in the way of choosing the node to expand at every iteration. We have chosen to use the node in Q with the lowest lower bound for the following reason: as the BB algorithm has an exponential running time, for some test cases we can't afford to let the program run until it successfully explores all the nodes in the branching tree due to time constraints; instead we can let it run for a fixed number of iterations and analyze the lower and upper bound values up to that point. We are interested to have an as high as possible global lower bound because that way we can know for certain that no better solution than the value of that lower bound exists. By choosing to expand the node with the lowest lower bound we increase the global lower bound at every iteration (as that node is removed from Q and the two descendants will have a higher or equal lower bound). The disadvantage of choosing the lowest lower bound node at every iteration is that in a naive implementation we would have to go through all the nodes in Q to find the minimum. However, we used a heap data structure to hold the nodes in Q, yielding a logarithmic time complexity for the operations of extracting the minimum or adding new nodes into the heap.

Optimization #4 For each node in the branching tree we have to compute its lower and upper bound. If computing one of the bounds results in an unfeasible solution, i.e. the maximum flow cannot be achieved anymore, then there is no point in computing the other bound as the two graphs used in computing the bounds have identical structures (nodes, edges, capacities), only different costs.

Optimization #5 For some nodes in the branching tree, we can avoid computing their lower or upper bound if some conditions are fulfilled. We will analyze two types of nodes:

- Let t be a node in the branching tree placed on level l with $y_l = 0$ and let $P(t)$ be its parent. If the solution for the lower bound of $P(t)$ did not use edge l, then it will also be the solution for the lower bound of t. The same applies for the upper bound.
- Let t be a node in the branching tree placed on level l with $y_l = 1$ and let $P(t)$ be its parent. The lower bound of t always has to be computed as the cost of edge l differs in the two graphs on which the lower bound is calculated. However, for the case of the upper bound, if the solution for the upper bound of $P(t)$ used edge l, then it will also be the solution for the upper bound of t.

Whenever we compute a lower or upper bound for a node placed on level l, we need to store the flow value for the edges $e_{l+1}, e_{l+2}, \cdots$. When analyzing whether we need to compute a lower or upper bound for a node t placed on level l in the branching tree, we look at the flow value of edge e_l in the computed bound of its parent. However, it is possible that its parent did not compute that bound (as it was not necessary by using the same logic) so we have to look at the parent of the parent of node t. This situation can occur multiple times, so in order to find the correct flow information we need to find the the closest ancestor for node t which has that bound computed.

To minimize the memory usage, we can delete the flow information of the lower bound of a node t whenever t is not the closest ancestor with a computed lower bound for any of its descendants (not only the direct descendants). This optimization cannot be applied for the upper bound case: it can be easily shown that when branching node t into nodes t_1 and t_2, exactly one of the upper bounds of t_1 and t_2 will be computed. Without any loss of generality let t_1 be the node for which we don't compute the upper bound. This means that t is the closest ancestor with a computed upper bound for node t_1. When branching t_1 the same situation occurs, as t will remain the closest ancestor with a computed upper bound for one of the direct descendants of node t_1. By induction, t will always remain the closest ancestor with a computed upper bound for one of its descendants, therefore we can never delete the flow information of the upper bound of node t.

Optimization #6 The lower the value of the global upper bound is, the more pruning will occur, which means that less nodes in the branching tree have to be analyzed. In order to improve the global upper bound, we can first apply a heuristic algorithm for the MCMFP-FC, when no edges are fixed (for the root node of the branching tree). We could further improve the global upper bound by applying the heuristic for each node in the branching tree, however, from our experiments, this approach did not increase the pruning (by a significant amount), but greatly increased the running time. Therefore, we only apply the heuristic for the root node. The heuristic we used is the Cost Function Slope heuristic which will be described in section 3.2.3.

Optimization #7
Definition 2. *A **cut** is a partition of the vertices of a graph into two disjoint subsets.*
Definition 3. *The **cut-set** of the cut is the set of edges whose endpoints are in different subsets of the cut.*

Definition 4. *In a flow network, an **s-t cut** is a cut that requires the source s and the sink t to be in different subsets. The **capacity** of an s-t cut is equal to the sum of the capacities of the directed edges going from one node from the subset containing the source to one node from the subset containing the sink.*

Theorem 2. *The maximum value of an s-t flow is equal to the minimum capacity over all s-t cuts.*
(Proof by Elias et al. [11])

We can use the max-flow min-cut theorem stated above to deduce that if we identify the minimum capacity s-t cut, and remove any of the edges which cross the cut (directed edges from a node in the subset containing the source to a node in the other subset), then the minimum capacity over all s-t cuts will decrease, therefore the maximum flow will also decrease. This is not acceptable for MCMFP-FC as we want to obtain the maximum flow possible. This means that the edges crossing the minimum capacity s-t cut must always be included in the maximum flow solution, so we can fix the value of y for those edges to 1. Having more binary variables y fixed before applying the BB algorithm, results in less branching and a lower run time.

In order to find the minimum capacity s-t cut we can use the following simple algorithm:

Algorithm 3 Minimum capacity s-t cut algorithm

Require: $G = (V, E)$ - the flow network.
1: Apply MCMFP-ED on G.
2: Do DFS on the residual network G_f starting from the source s.
3: The visited nodes will represent the subset containing the source s. The unvisited nodes represent the subset containing the sink t. These two subsets represent the minimum capacity s-t cut for G. ▷ s and t will be in different subsets as there is no path in G_f from s to t, i.e. the sink will be unvisited.
4: All the edges $e = (u, v) \in E$ with u visited and v unvisited, *cross* the cut.

3.2.2 Simulated annealing heuristic

Simulated annealing[1] (SA) is a local search heuristic used for finding a good approximation to the global minimum (or maximum) of a given function in a large search space. The SA algorithm starts from a random solution and at each iteration attempts to replace the current solution by a random solution taken from the neighborhood of the current solution. The new solution is always accepted if it is better than the current one, but it can also be accepted when it is not better, based on a probability function that depends both on the difference between the values of the two solutions and also on a global parameter T (called temperature), that is gradually decreased during the process. The most common used probability function for the minimization SA algorithm is $e^{\frac{cost(S)-cost(S')}{T}}$, where S is the current solution and S' is the new random solution. We can notice that if the difference in cost between the two solutions is low or if the value

[1]Fore a more detailed description of the main principles of SA, I recommend the work of Henderson et. all [19].

of T is really high, then the new solution is likely to be accepted. As the temperature approaches zero, the exponent approaches $-\infty$, and the probability approaches zero. The general SA algorithm has the following structure:

Algorithm 4 SA algorithm

Require: T - initial temperature, ϵ - final temperature, $\alpha \in [0, 1)$ - temperature decrement, L - number of candidate solutions analyzed for a fixed temperature.

1: Initialize S with a random solution.
2: **while** $T > \epsilon$ **do**
3:     **for** $i \leftarrow 1, L$ **do**
4:         Generate random solution $S^{'}$ in the neighborhood of S.
5:         **if** $cost(S) > cost(S^{'}) \vee random(0, 1) \leq e^{\frac{cost(S)-cost(S^{'})}{T}}$ **then**
6:             $S \leftarrow S^{'}$
7:     $T \leftarrow T * \alpha$

A more detailed description of the input parameters for the SA algorithm (T, ϵ, α and L) is given in the Experiments and Results section 3.2.4.

To apply the SA algorithm on MCMFP-FC[1], we need to start from any maximum flow solution and find a method to generate a random solution in the neighborhood of the current one. The way we generate the random candidate solution from the current solution S is as follows:

- Select a random edge $e = (u, v)$ with $f_e > 0$ from S which is not in the minimum capacity s-t cut-set (otherwise by removing this edge the maximum flow will decrease and the new solution will be rejected).
- Drain f_e flow sent from s to u.
- Drain f_e flow sent from v to t.
- Remove edge e from the G by setting its capacity to zero.
- Find random augmenting paths from s to t in the residual network G_f, until no such paths exist.
- Discard the new solution $S^{'}$ if it doesn't have the same maximum flow as S. Otherwise, reset the capacity of edge e to its original value, so that if this solution will be accepted, the edge e will have the chance to be used again in the next iterations.

Draining flow between two nodes v_1 and v_2 consists in finding random paths from v_1 to v_2 considering only edges with positive flow. When such a path is found, we decrease flow on all its edges equal to the minimum flow value among those edges (or the amount of flow left to be drained on edge (u, v), whichever value is lower). When f_e flow has been drained the process is complete.

In order to find a random path between two nodes v_1 and v_2, we use a slightly modified depth-first search algorithm: if from a node u we can move to k other unvisited nodes,

[1] For the simulated annealing approach we do not consider edge demands in order to simplify the process of finding a random solution in the neighborhood.

each will have a probability of $\frac{1}{k}$ to be selected as the next node. This method only finds a pseudo-random path. To find a true random path the probability to select neighbor v of u should be: $\frac{paths(v)}{\sum_{i}^{i \in N(u), !visited(i)} paths(i)}$, where $paths(v)$ denotes the total number of paths from v to v_2. However, computing $paths(v)$ is NP-hard as there may be an exponential number of simple paths between two nodes in graph.

3.2.3 Cost function slope heuristic

In the Cost function slope (CFS) heuristic we intend to replace the cost function of all edges (consisting in the fixed cost and cost / flow components) with cost functions which only contain a cost / flow component and to apply the polynomial running time algorithm for MCMFP-ED on the resulting graph. To explore the search space, we vary the slope of the new cost-function for each edge $(u, v) \in E$, at every iteration, based on the formula: $cost^{'}(u, v) = cost(u, v) + \frac{fixedCost(u,v)}{\frac{totalFlow(u,v)}{counter(u,v)}}(1 - \frac{decay(u,v)}{100})$, where:

- $cost^{'}(u, v)$ - the new cost function for edge (u, v).
- $cost(u, v)$ - the cost per flow component of the cost function of edge (u, v) in the original graph G.
- $fixedCost(u, v)$ - the fixed cost component of the cost function of edge (u, v) in the original graph G.
- $totalFlow(u, v)$ - the total amount of flow sent on edge (u, v) from the beginning of the CFS algorithm up to the current iteration.
- $counter(u, v)$ - the number of times edge (u, v) had a positive flow from the beginning of the CFS algorithm up to the current iteration. The fraction $\frac{totalFlow(u,v)}{counter(u,v)}$ represents an arithmetic mean for the flow sent on edge (u, v) up to the current iteration.
- $decay(u, v)$ - the decay coefficient for edge (u, v), taking values between 0 and 100. Whenever we send zero flow on edge (u, v) during an iteration, we increase the decay coefficient by a small amount. When the flow on edge (u, v) becomes positive, we reset the decay coefficient to 0. The reason for adding this extra parameter to the formula is that the cost function could sometimes remain blocked in a really steep position, meaning that it will hardly ever receive any flow in the subsequent iterations. This way, we make the slope less steeper with every iteration when the amount of flow on that edge is 0.

For the decay coefficient we use a small increment which is determined by: $0.2 * (rand() \%5 + 1)$. To further increase the exploration, we have added a small chance to reset the variables $totalFlow(u, v)$ and $counter(u, v)$ in order to erase the history of the flow sent on edge (u, v). The probability to reset the history of an edge is directly proportional to the size of its history: $counter(u, v)\%$. Figure 3.8 illustrates how the cost function could vary over time.

The CFS heuristic runs for a fixed number of iterations, during which the cost functions of the edges are modified. The higher the number of iterations, the higher the chance

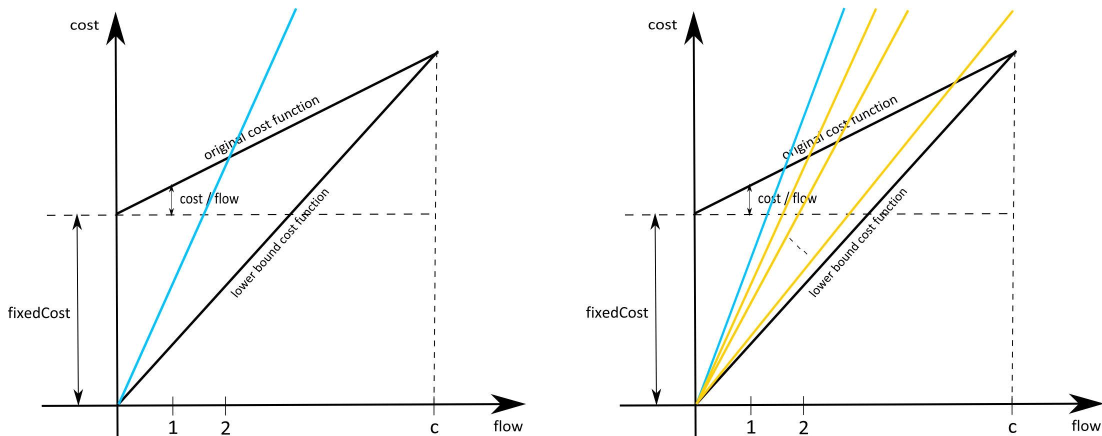

Figure 3.8. The cost function of an edge during the CFS iterations. In the first iteration of the algorithm we use the lower-bound cost function and we get 2 units of flow on that edge. The blue colored cost function in the leftmost image represents the cost function of the edge for the second iteration. In the second iteration we get only 1 unit of flow on the edge and we update the cost function using the formula of the heuristic. The blue colored cost function in the rightmost image represents the cost function of the edge for the third iteration. In the third iteration there is no flow sent on the edge, and the decay is increased resulting in a less steep cost function for the next iteration (yellow cost function). This situation can occur multiple times and the cost function will become less and less steep.

of obtaining a better solution, as the result of the heuristic represents the best solution encountered during the iterations. The intuition behind this method is that we try to approximate the real cost of sending flow on an edge (which contains the fixed cost and cost per flow components) with a cost function which does not have a fixed cost. By doing this, we can apply the MCMFP-ED algorithm to find a solution for the problem in polynomial time.

3.2.4 Experiments and results

The three methods for solving MCMFP-FC have been implemented in C++ and have been tested on a computer with an Intel Core 2 Duo @ 3.00GHz processor, 3GB of RAM, running Windows XP SP3. Each method receives as input a graph instance as well as some run-time parameters:

- Branch-and-bound
  - **# of iterations**: the maximum number of nodes which will be explored in the branching tree. This value is used to limit the running time of the method, as in the worst case scenario (when no pruning can be done) the algorithm goes through all the nodes in the branching tree.
  - ϵ: If we do not need the optimal solution, but just a solution close to the optimal, we can set a positive value to ϵ and stop the BB algorithm whenever $U - L \leq \epsilon$. By default this parameter is 0, meaning that we require the optimal solution.
- Simulated annealing

- **T**: the initial temperature for the SA method. This value should be large enough such that a move to almost any neighborhood state is allowed during the first iterations of the algorithm (when the temperature is high). However, choosing a too high initial temperature will transform the algorithm in a random search during the early stages, until the temperature is cool enough to start acting as a simulated annealing.
- ϵ: the final temperature for the SA method. When the temperature reaches this value, the algorithm stops. Ideally this temperature should be close to 0, but this will cause the algorithm to run for a long time. In practice, when the temperature approaches 0, the chances of accepting a worse solution are almost the same as when the the temperature is namely close to 0.
- **L**: the number of iterations at each temperature.
- α: the temperature drop. After each L iterations, the temperature is multiplied by α in order to obtain a slightly lower temperature. Therefore, the value of this parameter should be in the range of $[0.8, 0.95]$, depending on how fast the cooling should be.

- Cost-function slope heuristic
  - **# of iterations**: the number of iterations for one CFS heuristic run.
  - **# of runs**: we have empirically determined that restarting the heuristic may increase the chance of obtaining a better solution. This parameter indicates the number of independent CFS heuristic runs.

We have tested the performance of the three methods both in terms of running time and quality of the solutions. The input graphs were generated using a model developed by Klingman et. all [26] in "NETGEN: A program for generating large scale capacitated assignment, transportation and minimum-cost flow network problems"[1].

The first input graph instance (netgen_40_120) has the following properties:

- 40 nodes, 120 edges.
- 3 producers, 4 consumers, supply = demand = 400.
- Minimum capacity 0 for all edges. Having a minimum edge capacity larger then 0 only makes the instance easier to solve, as it implies less binary decision variables $(y(u, v) = 1, \forall (u, v) \in E,\ \text{s.t.}\ d(u, v) > 0)$. Maximum capacity has values between 1 and 400.
- The cost per unit of flow of connections is randomly generated with values between 1 and 50.
- The fixed cost of connections is randomly generated with values between 1 and 1000.

Table 3.1 contains the results of the experiments for the netgen_40_120 graph instance: the cost of the solution and the time expressed in seconds. The BB algorithm obtains

[1] We have used the C implementation of N. Schlenker for the NETGEN model. The source code is available at: http://elib.zib.de/pub/Packages/mp-testdata/generators/netgen/index.html

algorithm	cost of solution	time (s)
BB($100k, 0$)	39791	797 (it $\approx 80k$)
SA($3000, 1, 100, 0.9$)	41337	43
CFS($500, 50$)	39791	13

Table 3.1. The results of the BB, SA and CFS heuristic for the netgen_40_120 graph instance. The values in brackets next to each algorithm represent the run-time parameters described in this section. The cost of the solution for the SA algorithm is averaged over 5 independent runs.

the optimal solution before it reaches its maximum number of iterations. However, the solution is found in more than 13 minutes of run time. The same solution is found by the CFS heuristic in only 13 seconds. The SA algorithm obtains a good solution (compared to cost of other possible solutions), in a reasonable amount of time. One explanation of the fact that SA does not reach the optimal solution is that while BB does an exhaustive search, and the CFS heuristic uses convex functions approximations for the real cost functions of the connections, the SA does not use any cost information when searching for new solutions in the neighborhood. Figure 3.9 illustrates the evolution in time (number of iterations) of the cost of the current and best solution for the SA method. When the temperature is high (in early stages), the cost of the current solution decreases, but it can also have *jumps*. As the temperature decreases, the *jumps* are less frequent. We can also notice that the final temperature has a good value, as by the end of the iterations, the plot of the best solution can be approximated with a horizontal line.

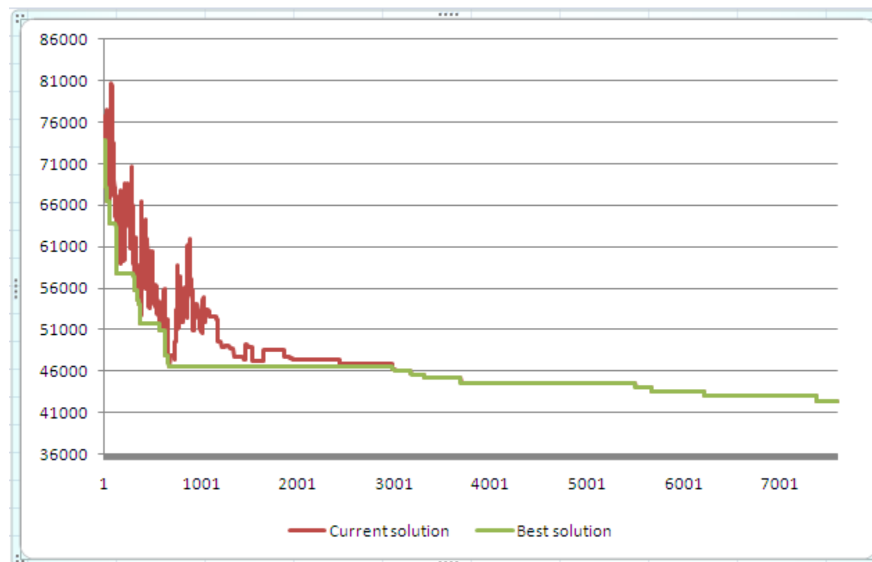

Figure 3.9. The evolution in time of the cost of the current and best solution for the SA algorithm (on netgen_40_120 graph instance). The two plots overlap beginning from iteration ≈ 3000.

We have also used for testing a larger graph instance (netgen_150_1000) with the following properties:

- 150 nodes, 1000 edges.
- 5 producers, 8 consumers, supply = demand = 800.

- Minimum capacity 0 for all edges. Maximum capacity has values between 1 and 800.
- The cost per unit of flow of connections is randomly generated with values between 1 and 50.
- The fixed cost of connections is randomly generated with values between 1 and 1000.

algorithm	cost of solution	time (s)
BB$(10k, 0)$	55592$(L = 47978.6)$	1412 (it $= 10k$)
SA$(20000, 1, 50, 0.9)$	124545	225
CFS$(500, 10)$	55373	311

Table 3.2. The results of the BB, SA and CFS heuristic for the netgen_150_1000 graph instance. The values in brackets next to each algorithm represent the run-time parameters described in this section. The cost of the solution for the SA algorithm is averaged over 5 independent runs.

The results in Table 3.2 indicate that increasing the number of edges significantly increases the running time of the algorithms, even if some of the parameters were reduced. (like the number of iterations of BB or the number of runs for the CFS heuristic). For this graph instance, the BB algorithm does not manage to find the optimal solution in the maximum number of iterations, but provides a lower bound (L) which can be used to evaluate the results of the other two methods. The CFS heuristic outperforms the SA method and is *at most* $\frac{55373}{47978.6} = 1.154$ times worse than the optimal solution.

In order to make our results more reliable, we have tested our algorithms on more generated graphs with the same properties as the two graph instances: netgen_40_-120 and netgen_150_1000. Tables 3.3 and 3.4 contain the results of these additional experiments which indicate that the implemented methods have a consistent behavior.

Test #	BB$(100k, 0)$		SA$(3000, 1, 100, 0.9)$		CFS$(500, 50)$	
	cost	time(s)	cost	time(s)	cost	time(s)
1	41796	809 (it $\approx 83k$)	45195	42	41796	13
2	38314	789 (it $\approx 77k$)	40112	42	38314	13
3	40970	896 (it $\approx 91k$)	42637	43	40970	14
4	43843	815 (it $\approx 84k$)	45864	43	43843	13
5	39181	772 (it $\approx 75k$)	43613	43	39181	12
6	37880	818 (it $\approx 84k$)	39030	42	37880	12
7	39558	761 (it $\approx 73k$)	41114	43	39558	14
8	42201	899 (it $\approx 91k$)	43347	43	42201	12
9	41029	848 (it $\approx 88k$)	45671	43	41029	12
10	42492	758 (it $\approx 73k$)	44249	43	42492	13

Table 3.3. The results of the BB, SA and CFS heuristic for additional netgen_40_120 graph instances. The values in brackets next to each algorithm represent the run-time parameters described in this section. The cost of the solution for the SA algorithm is averaged over 5 independent runs.

Test #	BB($10k, 0$)		SA($20k, 1, 50, 0.9$)		CFS($500, 10$)	
	cost	time	cost	time	cost	time
1	51712($L = 47443$)	1421($it = 10k$)	119707	229	52187	313
2	58940($L = 49530$)	1417($it = 10k$)	134630	220	58445	309
3	60138($L = 49701$)	1420($it = 10k$)	140649	228	59643	315
4	58137($L = 51000$)	1416($it = 10k$)	137661	225	58064	314
5	59138($L = 47831$)	1411($it = 10k$)	142206	223	58928	311
6	57816($L = 51162$)	1410($it = 10k$)	133996	226	57923	310
7	56179($L = 48449$)	1406($it = 10k$)	131659	221	55924	310
8	55296($L = 46060$)	1410($it = 10k$)	122825	221	55364	312
9	55173($L = 46759$)	1420($it = 10k$)	119055	224	54896	316
10	56219($L = 48620$)	1419($it = 10k$)	130253	223	56127	315

Table 3.4. The results of the BB, SA and CFS heuristic for additional netgen_100_1500 graph instances. The values in brackets next to each algorithm represent the run-time parameters described in this section. The cost of the solution for the SA algorithm is averaged over 5 independent runs.

Chapter 4

Conclusions

Designing electrical power network grids is a challenging and complex problem, which is hard to be modeled and solved as a single computer program. Instead, we can try to solve several subproblems, sometimes using some simplifying assumptions. In the current paper we have successfully researched and solved two such subproblems: connecting a new point to an existing electrical grid based on Euclidean distances in a non-uniform weighted space and choosing the cost-optimum design for a new electrical network in which we are given information about the producers, the consumers and the possible connections between points in the network.

For the first subproblem we have shown that Dijkstra's algorithm combined with a point sampling approach can be used to find an approximate solution, the quality of the result being determined by an input parameter which also determines the run time of the method. We have implemented this solution in both C++ and Powerfactory (DPL) and we have empirically determined that C++ is a much more efficient programming language. Given this result, we recommend that all code in a Powerfactory application should be written in C++ and then called within Powerfactory using a DLL.

In the second part of this paper we have modeled the problem of designing a new electrical network as a maximum flow problem for which connections do not only have a cost for each unit of flow sent, but also a fixed cost, which has to be payed if the connection is used in the network. We proposed three different approaches for solving this problem: a branch-and-bound (BB) algorithm which yields an exact solution, a simulated annealing algorithm and a cost-function slope (CFS) heuristic which ignores the fixed cost of the connections and varies the cost per unit of flow in order to explore the solution space. Experiments have shown that while the branch-and-bound method guarantees the optimum solution, the run time of this approach is too high for large problem instances. However, using the cost-function slope heuristic we obtain a good solution in a reasonable amount of time. We come to the conclusion that the BB algorithm should be used for small input instances, to obtain the best possible solution. For larger input instances, the cost-function slope heuristic is preferred. Even for these large instances, the BB algorithm can be run for a limited number of iterations in order to obtain a lower bound which can be used as a reference for determining the quality of the CFS heuristic solution.

4.1 Future work

While modeling the problem of designing an electrical network as a maximum flow problem is interesting from an algorithmic point of view and allows efficient computation of good solutions, there are certain drawbacks to this approach, which could be addressed in the future:

- *Additional electricity constraints for a more realistic model.*
  Besides the capacity constraints of the connections, the only electricity constraint which is currently implemented in the model is the flow conservation constraint: the flow leaving a vertex minus the flow entering the vertex should equal the production/consumption of flow at that vertex. By only having these constraints, the model can take advantage of the known algorithms for solving maximum flow problems. In the future, more electricity constraints could be added to the model to make it more realistic (e.g. voltage constraints). By adding more constraints, the linear program will not be solvable by a maximum flow algorithm anymore, but LP solvers could be used instead. It is important to notice that the complexity of the linear program increases by adding more constraints, which will result in a higher running time of the algorithms.

- *Production and demand patterns*
  In our model, the production and demand values are constant (more exactly, we find the optimal solution for a specific moment in time). In reality, the production and demands vary over time, meaning that the solution of our algorithm can become invalid for other production and demand values. For example, imagine that we compute the optimal design when the demand is low. In this situation, the optimal solution will probably use the cheaper, lower capacity edges. If the demand rises, the capacity of the chosen connections will not be able to satisfy the new increased demand, and the flow in the network will no longer be a maximum flow. One idea to solve this issue would be to run the algorithm for multiple time moments (select the most critical timeframes), however, the problem remains in how to efficiently merge the solutions into a single global one.

- *Change the maximum flow constraint from a hard to a soft constraint*
  In some situations, obtaining the maximum flow in the network can only be done at a really high cost. What if we could find a solution which *almost* maximizes the flow, but at a much lower cost? In order to be able to find such solutions, we could remove the constraint that the flow should be maximized and instead add a penalty in the objective function for the difference between the flow of the current solution and the maximum possible flow. The value of the penalty can be adjusted to reflect the importance of having an as high as possible flow in the network.

- *Storage*
  The current model does not offer any possibility for energy storage. For storing energy at a certain node in the grid, we could add a "virtual" connection between that node and the sink of the network with the following properties: fixed cost equal to the cost of buying the storage equipment, cost per unit of flow equal to the cost of storing one unit of flow and maximum capacity equal to the size of

the storage device. However, we only want to store energy if there is an excess of production, i.e. all customer demands have been satisfied. Therefore, we need to introduce a penalty in the objective function directly proportional to the amount of unsatisfied demand.

- *Automatic detection of connection costs*
  Currently, the connection costs have to be manually set in the input file. To automatically obtain a good approximation for the fixed cost of a connection, we can integrate the first subproblem discussed in this paper by multiplying the length of the (close to) optimum path between the endpoints of the connection with the cost (per unit) of the type of cable used (a higher capacity cable will probably cost more than a lower capacity one).

Appendix A

Dijkstra's algorithm

Dijkstra's algorithm computes the shortest path between a single source vertex and every other vertex in a positive-weighted graph $G(V, E)$. Depending on the specific implementation, the algorithm can run in $O\left(|V|^2\right)$, $O\left(|E|\, log\, |V|\right)$ or even $O\left(|E| + |V|\, log\, |V|\right)$ time.

The algorithm will assign some initial distance values and will try to improve them step by step:

1. Assign to every vertex a tentative distance value from the source vertex: set it to zero for the source vertex and to infinity for all other vertices.
2. Mark all vertices as *unvisited*.
3. Search for an unvisited vertex with the lowest distance value (initially this will be the source vertex). Let this vertex be u.
4. Let $d[u]$ be the tentative distance from the source vertex to u. Take each neighbor v of vertex u. Let $cost[u, v]$ be the cost of the edge (u, v). If $d[u] + cost[u, v] < d[v]$, update the tentative distance for vertex v. This step is reffered to as *relaxing* edge (u, v).
5. Mark vertex u as visited.
6. Go back to step 3 as long as there are unvisited vertices.

A more comprehensive and detailed description of Dijkstra's algorithm is given by Cormen et al. [7, p. 658].

Appendix B

Bellman-Ford algorithm

Similar to Dijkstra's algorithm, the Bellman-Ford algorithm computes the single-source shortest paths in a weighted graph $G(V, E)$. However, the difference is that Bellman-Ford can be applied on graphs with negative-weighted edges. In such graphs, the algorithm will either find the shortest paths or deduct that a negative-weighted cycle exists.

The algorithm is based on a dynamic-programming approach and has a similar structure to Dijkstra's algorithm. Instead of selecting the node with the minimum distance from the source, and relaxing the edges of that node, Bellman-Ford relaxes all the edges at every iteration. As the shortest path will contain at most $|V|$ nodes, we need to relax all the edges at most $|V| - 1$ times. Therefore, the complexity of the algorithm is $O(|V| \, |E|)$.

To identify if the graph contains a negative-weighted cycle we can relax all the edges one more time after the first $|V| - 1$ iterations. If any path is improved in this iteration it means that the path has more than $|V|$ vertices which implies that there exists a negative-weighted cycle.

The proof of correctness of this algorithm is given by Cormen et al. [7, p. 653].

Appendix C

Edmonds-Karp algorithm

Let $G(V, E)$ be a directed graph, and for each edge $(u, v) \in E$ let $c(u, v)$ be the capacity and $f(u, v)$ be the flow. We want to find the maximum flow from the source s to the sink t. At each step of the algorithm the following constraints have to be satisfied:

$$f(u,v) \leq c(u,v), \qquad \forall (u,v) \in E \qquad (1)$$

$$\sum_{i \in M(u)} f(i,u) = \sum_{j \in N(u)} f(u,j), \qquad \forall u \in V, u \neq s, u \neq t \qquad (2)$$

The first constraints ensures that the maximum capacity of an edge is respected while the second constraint guarantees that the flow entering a node is equal to the flow leaving that node (for all nodes except the source and the sink).

We define the residual network $G_f(V, E_f)$ as the network which reflects the amount of *available* capacity:

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

We can see that the residual network indicates how much flow we can still send on an edge without exceeding the maximum capacity of that edge, as well as how much flow we can *return* on that edge.

The idea behind the Edmonds-Karp algorithm is very simple. As long as there is a path from the source to the sink with available capacity on all edges in the path, we send flow along those edges. Then we find another path, and so on. A path with available capacity is called an *augmenting* path. When there is no augmenting path, the algorithm has found the maximum flow.

To find an augmenting path we will apply a breadth-first search in the residual network G_f starting from the source node s until we reach the sink t. We identify the minimum available capacity along the path, τ; we increase the flow on each edge of the path by τ and update the residual network accordingly.

The running time of the algorithm is $O(VE^2)$ and is found by showing that each augmenting path can be found in $O(E)$ time (the complexity of the breadth-first search), that every time at least one of the E edges becomes saturated (when we increase the flow of the edges of a path by τ, one of those edges will become saturated), that the distance from the saturated edge to the source along the augmenting path must be longer than last time it was saturated, and that the length is at most V. The complete proof is given by Edmonds and Karp [10].

Appendix D

Checking if a line segment intersects the interior of a polygon

We are interested to check if a line segment intersects the interior of a simple bounded polygon. The interior of a polygon is defined by all the points *inside* the polygon, *not* including the points on the edges of the polygon. We use the following approach for solving this problem:

- Compute the axis aligned bounding box BB of the polygon. The bounding box is a rectangle defined by the points: $(min_x, min_y), (max_x, min_y), (max_x, max_y)$, (min_x, max_y), where min_x represents the minimum x-coordinate of the vertices of the polygon (similar definition for min_y, max_x, max_y). Check if the line segment intersects BB. If it does not, then it will not intersect the polygon either. We use this step as an optimization, as it is computationally inexpensive and can filter out some of the cases.
- Given the line segment AB, find all intersection points between AB and the edges of the polygon. Sort these points by their X coordinate (or Y coordinate if they all have the same X coordinate): $A, I_1, I_2, \cdots, I_k, B$. Let T be this sorted list.
- Take any two consecutive points P_1P_2 from the sorted list T. These two points represent a segment which is part of the initial segment AB. As P_1 and P_2 are consecutive points in T, the segment P_1P_2 will not intersect any of the polygon edges except possibly in P_1 or P_2. This means that the segment will either be completely *outside* or *inside* the polygon. To distinguish between the two cases we can select any point inside the segment P_1P_2. Let M_{12} be the middle point on this segment. If M_{12} is inside the polygon, P_1P_2 will be inside the polygon as well, otherwise it be outside.
- If there is any subsegment P_1P_2 which is *inside* the polygon then we can say that AB intersects the interior of the polygon.

D.0.1 Finding the intersection point between two segments

In order to find the intersection point between segment AB and an edge of a polygon we use a vector cross product approach inspired by the work of Ronald Goldman [16, p. 304]. The cross product of two vectors a and b is denoted by $a \times b$ and is defined as a vector c that is perpendicular to both a and b, having a direction given by the right-hand rule and a length equal to the area of the parallelogram determined by the two vectors a and b.

Suppose we have two line segments which run from p to $p+r$ and from q to $q+s$. Then any point on the first line can be represented as $p + tr$ (for a scalar parameter t) and any point on the second line as $q + us$ (for a scalar parameter u). The two lines will intersect if we can find t and u such that: $p + tr = q + us$. By crossing both sides with s, we get $(p + tr) \times s = (q + us) \times s$. Since $s \times s = 0$, this means $t\,(r \times s) = (q - p) \times s$. Solving it for t we get $t = \frac{(q-p)\times s}{r\times s}$. In a similar way we get $u = \frac{(q-p)\times r}{r\times s}$.

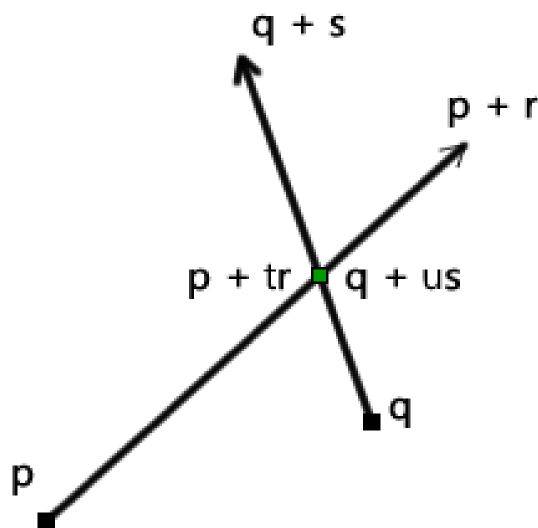

Figure D.1. The vector cross product approach for finding the intersection point between two line segments.

There are four possible cases:

1. $r \times s = 0$ and $(q - p) \times r = 0$, the two line segments are collinear.
2. $r \times s = 0$ and $(q - p) \times r \neq 0$, the two line segments are parallel but not collinear.
3. $r \times s \neq 0$ and $0 \leq t \leq 1$ and $0 \leq u \leq 1$, the two line segments intersect.
4. $r \times s \neq 0$ but t or u do not have values between 0 and 1, the two lines intersect, but the two line *segments* do not intersect.

If $r \times s = 0$, the angle θ between r and s is either 0 or 180 degrees, such that $sin(\theta) = 0$. This implies that r and s are parallel. If r and s are parallel, there are two situations: either they are collinear or not. The vector $q - p$ is the vector spanning from p to q, the start endpoints of the two segments. If this vector is parallel to r it means that the points p, q and $p + r$ are collinear, and as the two segments r and s are parallel it also

means that these points are collinear with $q + s$, meaning that r and s are collinear.

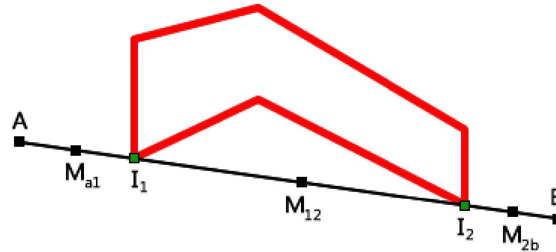

Figure D.2. Segment AB intersects the polygon in I_1 and I_2. The subsegments of AB will be AI_1, I_1I_2 and I_2B. For each of these subsegments, the middle points M_{a1}, M_{12} and M_{2b} are outside the polygon, i.e. the subsegments are outside the polygon as well. This means AB does *not* intersect the interior of the polygon.

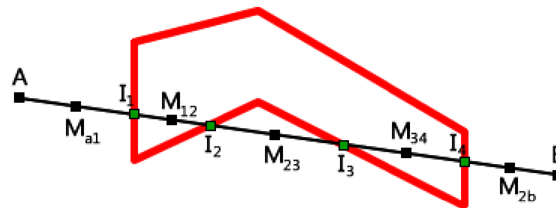

Figure D.3. Segment AB intersects the polygon in I_1, I_2, I_3 and I_4. The subsegments of AB will be AI_1, I_1I_2, I_2I_3, I_3I_4 and I_4B. As the middle point M_{12} is inside the polygon, the subsegment I_1I_2 is inside the polygon as well, i.e. AB intersects the interior of the polygon.

D.0.2 Checking if a point is inside a polygon

In computational geometry, the point-in-polygon problem asks whether a given point in the plane lies inside or outside a polygon. This problem can be solved using the ray casting algorithm: given a ray starting from the point and going in any fixed direction, we can count the number of times it intersects the edges of the polygon. If the point is outside the polygon, the number of intersections will be an even number; if the point is inside the polygon, the number of intersections will be an odd number. The algorithm is based on a simple observation that if a point moves along a ray from infinity to the probe point and if it crosses the boundary of a polygon, possibly several times, then it alternately goes from the outside to inside, then from the inside to the outside, etc.

If the point in question lies very close to the boundary of the polygon then the results may be incorrect because of the rounding errors caused by the finite precision arithmetic used in the code implementation.

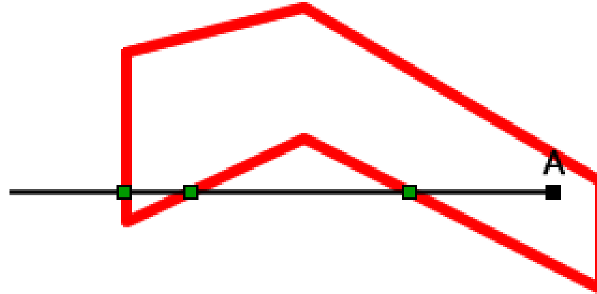

Figure D.4. The horizontal ray starting from A intersects the polygon in 3 points. Therefore, the point lies inside the polygon.

Appendix E

Input format for the C++ implementations

E.1 Optimal paths in a non-uniform weighted two-dimensional space

---

1 $x_s \quad y_s$
2 N_e
3 $x_{e1} \quad y_{e1}$
4 $x_{e2} \quad y_{e2}$
5 $\dots$
6 $x_{eN_e} \quad y_{eN_e}$
7 N_{fp}
8 M_1
9 $x_{1,1} \quad y_{1,1}$
10 $x_{1,2} \quad y_{1,2}$
11 $\dots$
12 $x_{1,M_1} \quad y_{1,M_1}$
13 $\dots$
14 $\dots$
15 $M_{N_{fp}}$
16 $x_{N_{fp},1} \quad y_{N_{fp},1}$
17 $x_{N_{fp},2} \quad y_{N_{fp},2}$
18 $\dots$
19 $x_{N_{fp},M_{N_{fp}}} \quad y_{N_{fp},M_{N_{fp}}}$
20 N_{cp}
21 P_1
22 c_1
23 $x_{1,1} \quad y_{1,1}$
24 $x_{1,2} \quad y_{1,2}$
25 $\dots$
26 $x_{1,P_1} \quad y_{1,P_1}$

```
27 ...
28 ...
29 P_{N_cp}
30 c_{N_cp}
31 x_{N_cp,1}  y_{N_cp,1}
32 x_{N_cp,2}  y_{N_cp,2}
33 ...
34 x_{N_cp,P_{N_cp}}  y_{N_cp,P_{N_cp}}
35 X_multiplier  Y_multiplier
36 N
```

Listing E.1. Input file format

Symbols in the listing:

- x_s y_s - coordinates of the start point.
- N_e - number of end points.
- x_{ei} y_{ei} - coordinates of the ith end point.
- N_{fp} - number of forbidden polygons.
- M_i - number of vertices of the ith forbidden polygon.
- P_i - number of vertices of the ith custom polygon.
- $x_{i,j}$ $y_{i,j}$ - coordinates of the jth vertex of polygon i (either forbidden or custom).
- c_i - coefficient of the ith custom polygon.
- $X_{multiplier}$ $Y_{multiplier}$ - scaling factors for the X and Y coordinates.
- N - desired total number of points.

E.2 MCMFP-FC

```
1 n m
2 D_1 D_2 ... D_n
3 C_1 C_2 ... C_n
4 u_1 v_1 d_1 c_1 cost_1 fixedCost_1
5 u_2 v_2 d_2 c_2 cost_2 fixedCost_2
6 ...
7 u_m v_m d_m c_m cost_m fixedCost_m
```

Listing E.2. Input file format

Symbols in the listing:

- n - number of nodes; m - number of edges.
- D_i - production/demand of node i.

- C_i - cost of producing/consuming one unit of flow at node i.
- u_i, v_i - endpoints of edge i; d_i, c_i - minimum and maximum capacity of edge i; $cost_i$ - cost per unit of flow of edge i; $fixedCost_i$ - fixed cost of edge i.

Bibliography

[1] *Snell's law, http://en.wikipedia.org/wiki/snell's_ law.*

[2] R.K. Ahuja, T.L. Magnanti, and J.B. Orlin, *Network flows: theory, algorithms, and applications*, Prentice Hall, 1993.

[3] A.V. Cabot and S.S. Erenguc, *Some branch-and-bound procedures for fixed-cost transportation problems*, Naval research logistics quarterly **31** (2006), no. 1, 145–154.

[4] Jean-Lou De Carufel, Carsten U. Grimm, Anil Maheshwari, Megan Owen, and Michiel Smid, *Unsolvability of the weighted region shortest path problem*, The European Workshop on Computational Geometry (2012).

[5] Jens Clausen, *Branch and bound algorithms - principles and examples*, 1999.

[6] L. Cooper and C. Drebes, *An approximate solution method for the fixed charge problem*, Naval Research Logistics Quarterly **14** (1967), no. 1, 101–113.

[7] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein, *Introduction to algorithms (3. ed.)*, MIT Press, 2009.

[8] Mark de Berg, Otfried Cheong, Marc van Kreveld, and Mark Overmars, *Computational geometry*, third ed., 2008.

[9] M. Diaby, *Successive linear approximation procedure for generalized fixed-charge transportation problems*, Journal of the Operational Research Society (1991), 991–1001.

[10] Jack Edmonds and Richard M. Karp, *Theoretical improvements in algorithmic efficiency for network flow problems*, J. ACM **19** (1972), no. 2, 248–264.

[11] P. Elias, A. Feinstein, and C. Shannon, *A note on the maximum flow through a network*, Information Theory, IEEE Transactions on **2** (Dec 1956), no. 4, 117–119.

[12] Jeff Erickson, *Algorithms course materials.*

[13] D.B.M.M. Fontes and J.F. Gonçalves, *Heuristic solutions for general concave minimum cost network flow problems*, Networks **50** (2007), no. 1, 67–76.

[14] D.B.M.M. Fontes, E. Hadjiconstantinou, and N. Christofides, *A branch-and-bound algorithm for concave network flow problems*, Journal of global optimization **34** (2006), no. 1, 127–155.

[15] Michael R. Garey and David S. Johnson, *Computers and intractability: A guide to the theory of np-completeness*, W. H. Freeman & Co., New York, NY, USA, 1979.

[16] Ronald Goldman, *Graphics gems*, Academic Press Professional, Inc., San Diego, CA, USA, 1990.

[17] P. Gray, *Technical noteŮexact solution of the fixed-charge transportation problem*, Operations Research **19** (1971), no. 6, 1529–1538.

[18] G. M. Guisewite and P. M. Pardalos, *Minimum concave-cost network flow problems: applications, complexity, and algorithms*, Ann. Oper. Res. **25** (1991), no. 1-4, 75–100.

[19] Darrall Henderson, SheldonH. Jacobson, and AlanW. Johnson, *The theory and practice of simulated annealing*, Handbook of Metaheuristics (Fred Glover and GaryA. Kochenberger, eds.), International Series in Operations Research and Management Science, vol. 57, Springer US, 2003, pp. 287–319.

[20] W.M. Hirsch and G.B. Dantzig, *The fixed charge problem*, Naval Research Logistics Quarterly **15** (1968), no. 3, 413–424.

[21] J. Kennington and E. Unger, *A new branch-and-bound algorithm for the fixed-charge transportation problem*, Management Science **22** (1976), no. 10, 1116–1126.

[22] D.B. Khang and O. Fujiwara, *Approximate solutions of capacitated fixed-charge minimum cost network flow problems*, Networks **21** (1991), no. 6, 689–704.

[23] D. Kim, X. Pan, and P.M. Pardalos, *An enhanced dynamic slope scaling procedure with tabu scheme for fixed charge network flow problems*, Computational Economics **27** (2006), no. 2, 273–293.

[24] D. Kim and P.M. Pardalos, *A solution approach to the fixed charge network flow problem using a dynamic slope scaling procedure*, Operations Research Letters **24** (1999), no. 4, 195–203.

[25] Dukwon Kim and Panos M. Pardalos, *A solution approach to the fixed charge network flow problem using a dynamic slope scaling procedure*, Oper. Res. Lett. **24** (1999), no. 4, 195–203.

[26] D. Klingman, A. Napier, J. Stutz, and TEXAS UNIV AUSTIN CENTER FOR CYBERNETIC STUDIES., *Netgen: A program for generating large scale (un)capacitated assignment, transportation, and minimum cost flow network problems*, Defense Technical Information Center, 1973.

[27] H.W. Kuhn and W.J. Baumol, *An approximative algorithm for the fixed-charges transportation problem*, Naval Research Logistics Quarterly **9** (1962), no. 1, 1–15.

[28] Joseph S. B. Mitchell and Christos H. Papadimitriou, *The weighted region problem: Finding shortest paths through a weighted planar subdivision*, J. ACM **38** (1991), no. 1, 18–73.

[29] F. Ortega and L.A. Wolsey, *A branch-and-cut algorithm for the single-commodity, uncapacitated, fixed-charge network flow problem*, Networks **41** (2003), no. 3, 143–158.

[30] U.S. Palekar, M.H. Karwan, and S. Zionts, *A branch-and-bound method for the fixed charge transportation problem*, Management Science **36** (1990), no. 9, 1092–1105.