Bachelorarbeit

Partitioning of Public Transit Networks

Matthias Hertel

07.09.2015

Albert-Ludwigs-Universität Freiburg im Breisgau
Technische Fakultät
Institut für Informatik

Bearbeitungszeitraum
06. 07. 2015 – 07. 09. 2015

Supervisorin
Dr. Sabine Storandt

Erklärung

Hiermit erkläre ich, dass ich diese Abschlussarbeit selbständig verfasst habe, keine anderen als die angegebenen Quellen/Hilfsmittel verwendet habe und alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten Schriften entnommen wurden, als solche kenntlich gemacht habe. Darüber hinaus erkläre ich, dass diese Abschlussarbeit nicht, auch nicht auszugsweise, bereits für eine andere Prüfung angefertigt wurde.

Contents

Zusammenfassung

Partitionierungen von öffentlichen Verkehrs-Netzwerken werden verwendet um die Berechnungszeit von Routen oder die Vorberechnungszeit bei Routenplanungs-Algorithmen zu beschleunigen.
Das Ziel der Partitionierung ist es, kleine Partitionen der Haltestellen zu finden, die möglichst viel Verkehr in ihrem Inneren beinhalten, sodass möglichst wenige Verbindungen zwischen den Partitionen verkehren. Um solche Partitionen für das öffentliche Verkehrs-Netzwerk von Deutschland berechnen zu können, wird das öffentliche Verkehrs-Netzwerk als Graph modelliert, wobei Knoten Haltestellen repräsentieren und Kanten mit der Häufigkeit gewichtet werden, mit der Fahrzeuge zwischen den Haltestellen verkehren.
Mit *k-means-clustering*, einem Fusions-basierten Algorithmus, *partitioning using natural cut heuristics* (PUNCH, engl. für "Partitionierung unter Verwendung von natürlichen Schnitt-Heuristiken"), und METIS werden vier Graph-Partitionierungs-Algorithmen getestet und mit einem Datensatz ausgewertet, der die Fahrpläne der Deutschen Bahn für das Jahr 2015 beinhaltet.
Ergebnis der Auswertung ist, dass der Fusions-basierte Algorithmus und METIS kurze Laufzeiten haben und kleine Partitionen mit einem kleinen Schnitt erzeugen. K-means-clustering löst das Problem besser als erwartet. PUNCH wurde für ungewichtete Graphen entwickelt und benötigt einige Anpassungen für gewichtete Graphen.

Abstract

Partitioning of public transit networks is a useful approach to speed up the query times or the preprocessing phase of public transit routing algorithms.
The aim of partitioning of public transit networks is to find small partitions of the stations of a given network such that most of the traffic lies in their inside while only little traffic goes between them. To find such partitions of the public transit network of Germany, the public transit network is modeled as a graph, where nodes represent stations and edges are weighted with the frequency of vehicles going between the stations.
With k-means-clustering, a merging-based algorithm, partitioning using natural cut heuristics (PUNCH) and METIS four graph partitioning algorithms are tested and evaluated on a dataset that contains the schedule of Germany of the year 2015.
It turns out that the merging-based algorithm and METIS have fast runtimes and produce small partitions with a small cut size. K-means-clustering solves the problem better than expected. PUNCH was invented for unweighted graphs and needs some adaptations for weighted graphs.

1 Introduction

1.1 The problem

For a given public transit network, the aim is to find a partitioning of the stations, such that the partitions are small and most of the traffic lies in the inside of the partitions while only little traffic goes between the partitions.
A trade-off between the two goals has to be made: If the partitioning is defined to consist of only one partition that contains the whole network, all traffic lies inside that partition, which is optimal, but on the other hand, the partition is as big as possible, which is pessimal. At the other extreme, every station of the network can be defined as its own partition. This optimizes the size of the partitions, since every partition contains only one station, but all traffic goes between the partitions, which is pessimal.
Similar problems are known to be NP-hard, e.g. partitioning a network into k partitions, such that the connections between the partitions are minimized ([1]). This means computing the optimal solution is not profitable. Instead, heuristic solutions are found and evaluated in this work.

1.2 Motivation

1.2.1 Transfer Pattern Routing

Bast et al. introduced the concept of Transfer Patterns ([2]). This means, for a given transit network, they compute rules like "If someone wants to travel from Freiburg to Zürich, optimal routes always go via Basel or are direct connections." By the use of precomputed Transfer Patterns, their algorithm achieves very fast query times.
For a given source, destination and time, the algorithm checks the train schedules for all the stored Transfer Patterns in order to find the fastest connection. In the given example this would be to find the next train from Freiburg to Basel, and then the successive train from Basel to Zürich, as well as the next direct connection from Freiburg to Zürich.
In order to achieve the fast query times, Transfer Pattern Routing has a heavy preprocessing phase, because the computation of the Transfer Patterns takes a lot of time. In theory, all optimal routes from any station to any other station at any

possible starting time have to be computed. This can be done with n runs of multi-criteria set-Dijkstra-algorithms, each starting at one of the n stations.
Even if not all optimal routes are computed, but only the optimal routes from some "important" stations to all other stations and from "unimportant" stations to the closest important stations, the precomputation is still slow.
For example, the algorithm of Bast et al. took 3203 hours to find the Transfer Patterns of the public transit network of North America, which means the non-parallelized computation runs for more than three months.

One idea to improve the runtime of the computation of Transfer Patterns is to use a partitioning of the network. The runtime of the preprocessing phase can be improved by only computing Transfer Patterns from every node to the other nodes of the same partition. Note that this means that the source and the target station of the Transfer Patterns are in the same partition, but paths can still cross other partitions.
With this approach and a good partitioning, many queries can already be answered. For example, most trips of users stay in the same city or local area, which means the source and the target station are likely to be in the same partition.
For long-distance queries, where the source and the target station are in different partitions, routes can be found by travelling from the source station to a station that is connected to the long-distance network (e.g. the main station of a city), then with a long-distance connection to any station of the target partition (e.g. the main station of another city), and finally from there to the target station. These routes are non-optimal, but their travel time gives an upper bound for the travel time of the optimal route.
The long-distance network is only a little part of the whole network. Therefore, the computation of Transfer Patterns on this subnetwork and for every partition of the network would cost only a fraction of the time that is needed to compute the Transfer Patterns of the whole network.

1.2.2 Routing on hierarchically partitioned public transit networks

Strasser and Wagner use partitionings of public transit networks to compute optimal routes very fast in [3].
They partition the network in a hierarchical way during a preprocessing phase. The top level of the hierarchy is the whole network. The next level contains partitions of the network. Each lower level contains partitions of the partitions of the upper level.
For each partition on the bottom level, connections that are part of optimal routes through the partition are computed and stored. This means for every station at the border of a partition, all optimal routes to every other border station of the partition are computed, and the connections that are part of these routes are stored.

The authors use an algorithm called Connection Scan for the computation of these connections.
The same is done for the partitions of the higher levels of the hierarchy, but now not all connections of a partition have to be taken into account, but only the connections that were stored for the subpartitions of the partition. That is because an optimal route through a partition can only consist of optimal routes through the subpartitions.
When a query is made, it is only necessary to search on the connections of the two partitions on the bottom level that contain the start node or target node and on the stored connections of the subpartitions of every partition that contains the start node or target node, in order to compute an optimal route. The query times of this approach are very fast, because this is only a small subset of all connections.

2 Preliminaries

2.1 Graph theory

In this work, the given public transit network is modelled as a graph in order to let graph partitioning algorithms run on it. This section introduces some terms and definitions that are used later.

2.1.1 Definitions

A **graph** $G = (V, E)$ contains of a set of nodes V and a set of edges E.
An **edge** is a tuple (u, v) of nodes from V.
In this work the public transit network is modelled as an undirected weighted graph.
In an **undirected graph**, the edge (u, v) and the edge (v, u) are the same.
In a **weighted graph**, each edge e has a weight $w(e)$ that can be an arbitrary number.
An example of an undirected weighted graph is given in Figure 2.1.

Two nodes u and v are **neighboured**, if an edge (u, v) exists in E.
The **neighbours** of a node v are the nodes that are neighboured to v.
The nodes u and v are **adjacent** to an edge (u, v) and vice versa.
The number of edges, that are adjacent to a node, is the **degree** of the node.
A **path** from a node u to a node v is a sequence of edges $(u, v_1), (v_1, v_2), ..., (v_k, v)$.
The number of edges of a path is its **length**. Edges can occur multiple times in a path and then are counted multiple times.
The nodes $u, v_1, v_2, ..., v_k, v$ are **induced** by this path.
A **shortest path** from a node u to a node v is one of the paths from u to v with the smallest length. The length of the shortest path from u to v is the **distance** from u to v. If no path from u to v exists, the distance is positive infinity.
If a path from u to v exists, u and v are **connected**.
If every node from V is connected to all other nodes from V, the graph is **connected**.

2.1.2 Contraction of nodes

Some algorithms contract nodes in order to make the graph smaller.
To provide this operation, sizes of nodes are introduced. Initially, every node v has

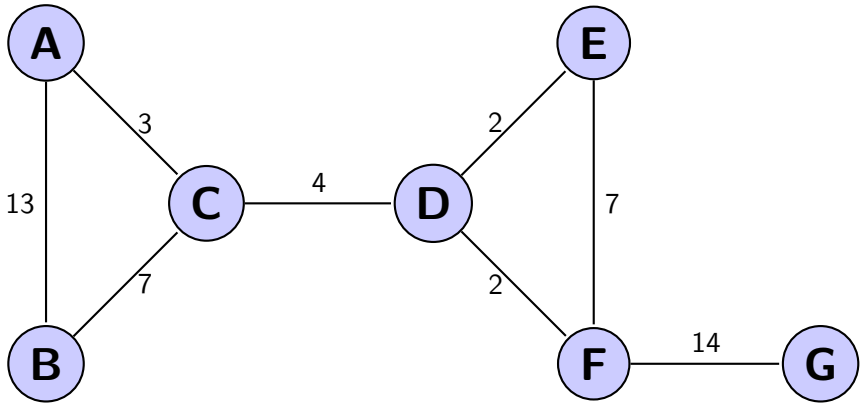

Figure 2.1: An undirected weighted graph.

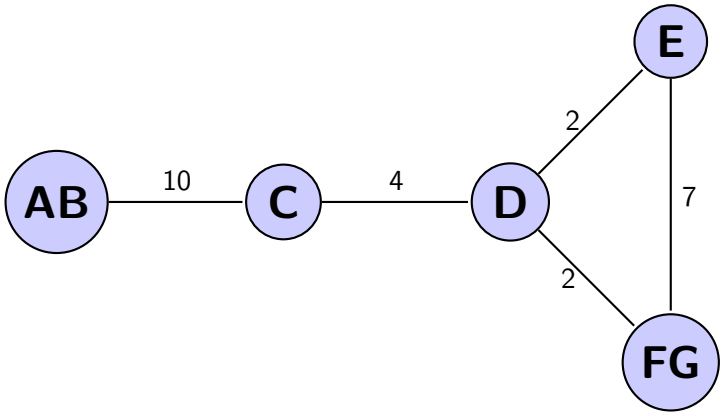

Figure 2.2: The graph from Figure 2.1 after contraction of nodes A with B and F with G. The new nodes AB and FG have size 2, the others size 1.

size $s(v) = 1$.
Contracting two nodes u and v means to replace them with a new node w with size $s(w) = s(u) + s(v)$, removing their adjacent edges and inserting new edges (w, x) for each edge (u, x) or (v, x). The new edges (w, x) have weight $w(u, x) = w(u, x) + w(v, x)$.
Contracting a set of nodes means contracting pairs of nodes in any order, since the result does not depend on the order.
A contracted version of the example graph from Figure 2.1 is shown in Figure 2.2.

A node v with size $s(v) > 1$ can be uncontracted. That means v gets eliminated and the nodes and edges that were contracted into v are again added to the graph.

2.1.3 Breadth-first-search

A breadth-first-search is a way of traversing the nodes of a graph. It is executed until a specific node is found or until all nodes that are connected to the start node are visited.

The breadth-first-search visits the nodes of the graph in the order of their distance to the start node. It begins at the start node and then visits all neighbours of the node, then the nodes that are neighbours of the neighbours and were not already visited, and so on.
When the nodes are numbered in the order in which they are visited in the depth-first-search, the labelling is called a level-order labelling. An example of a level-order labelling is shown in Figure 2.3.

Algorithm 1 uses a queue to store the nodes that will be visited next. The queue has a first-in-first-out ordering (FIFO).

Algorithm 1 breadth-first-search(start-node)

```
let Q be a queue
visit start-node
Q.enqueue(start-node)
while not Q.empty do
  node = Q.dequeue
  for all neighbours of node do
    if neighbour is not yet visited then
      visit neighbour
      Q.enqueue(neighbour)
    end if
  end for
end while
```

2.1.4 Depth-first-search

The depth-first-search differs from the breadth-first-search in the order of the visited nodes. The depth-first-search visits all neighbours of the first neighbour of a node, before visiting the other neighbours of the node. This means it goes as much into depth as possible before visiting the second neighbour that is close to the start node.
When the nodes are numbered in the order in which they are visited in the depth-first-search, the labelling is called a preorder labelling. An example of a preorder labelling is shown in Figure 2.4.

Algorithm 2 uses a stack instead of a queue to store the nodes that will be visited next. A stack has a last-in-first-out ordering (LIFO).

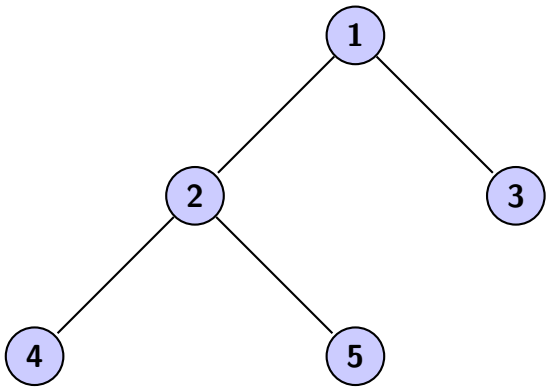

Figure 2.3: A breadth-first-search starting on the top visits the nodes from 1 to 5. The labelling is a level-order labelling.

Algorithm 2 depth-first-search(start-node)

```
let S be a stack
S.push(start-node)
while not S.empty do
  node = S.pop
  if node is not yet visited then
    visit node
    for all neighbours of node do
      S.push(neighbour)
    end for
  end if
end while
```

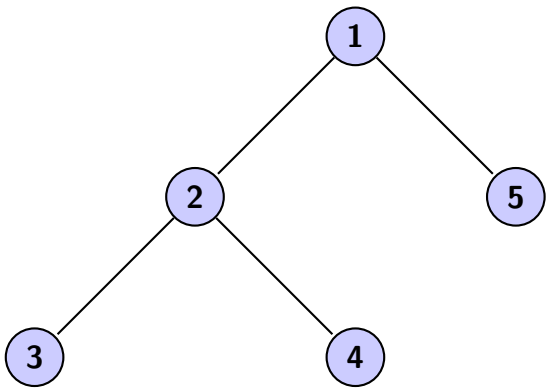

Figure 2.4: A depth-first-search starting on the top visits the nodes from 1 to 5. The labelling is a preorder labelling.

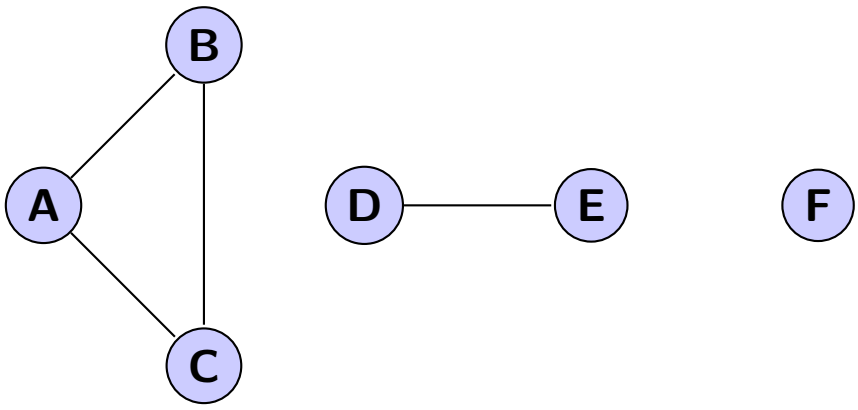

Figure 2.5: A graph with three connected components {A, B, C}, {D, E} and {F}.

2.1.5 Connected components

A subset of nodes $S \subseteq V$ is a **connected component** of G if every node from S is connected to all other nodes from S (i. e. a path from every node to every other node exists) and all nodes that are connected to any node from S are also in S.
A connected graph has only one connected component that contains all nodes of the graph. A graph that is not connected has multiple connected components.

The connected components of a graph can be computed by starting a breadth-first-search at any node and assigning all visited nodes to the same connected component. When the breadth-first-search stops, all nodes that are connected to the start node are visited and assigned to one connected component.
If not all nodes of the graph are connected to the start node, another breadth-first-search is started from a node, that is not already assigned to a connected component. This breadth-first-search finds the second connected component.
This procedure is repeated until every node of the graph is assigned to a connected component.
A graph with multiple connected components is shown in Figure 2.5.

2.1.6 Spanning trees

A **spanning tree** $T = (V_T, E_T)$ of a graph $G = (V, E)$ is a directed graph with $V_T = V$. For every edge $(u, v) \in E_T$ there must be an edge (u, v) in E.
There is one root node $r \in E_T$ that has no incoming edge. This means no directed edge (v, r) exists for any $v \in V_T$. All other nodes in V_T have exactly one incoming edge. If a directed edge (u, v) exists in V_T, u is called the parent of v and v is called the child of u.
All nodes of the graph G must be connected to the root node r in T to make T a spanning tree. Note that this is only possible if the graph G is connected. Otherwise

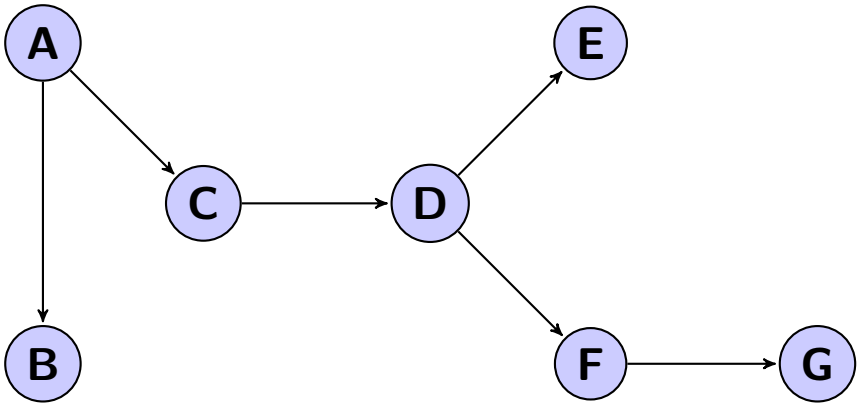

Figure 2.6: A spanning tree of the graph from Figure 2.1. A is the root node.

a spanning forest can be computed, which is a set of spanning trees, where each spanning tree contains the nodes of one connected component of the graph.
An example of a spanning tree is shown in Figure 2.6.

A spanning tree of a graph $G = (V, E)$ can be computed with a breadth-first-search. The search starts at the root r and adds all neighbours n and the connecting edge (r, n) as a directed edge to the tree, if the neighbour is not already in the tree. Then this procedure is repeated recursively with the neighbours.

2.2 Computing cuts

A **cut** of a connected graph $G = (V, E)$ is a subset of edges $C \subseteq E$ that separates two regions of the graph. If the edges in C are removed from the graph, it is no longer connected.
An **s-t-cut** is a cut with the property that the nodes s and t are no longer connected if the cut edges are removed.
The **cut size** $w(S)$ of a cut S is the sum of the weights of it edges.

$$w(S) = \sum_{e \in S} w(e)$$

A cut that consists of exactly one edge is called a **bridge**.
A cut that consists of two edges, where none of these two edges is a bridge, is called a **two-cut**.
A **two-cut class** is a set of edges, where no edge is a bridge and any pair of edges forms a two-cut. An edge can only belong to one two-cut class. A graph can have multiple two-cut classes.
The cut with the minimum cut size is the **minimum cut**.

Algorithms for the computation of bridges, two-cut classes and minimum cuts are presented in this section.

2.2.1 Bridges

An edge (u, v) is a bridge if all paths from u to v go over the edge (u, v).
Bridges are edges that are necessary for the connectivity of the graph. If we remove a bridge, the graph decomposes in two parts.

The bridges of a graph can be computed with an algorithm that is similar to the one described by Tarjan in [4] and has linear runtime.

First, a spanning tree is built from the graph and the nodes are numbered in pre-order. Both is done with a depth-first-search. The depth-first-search starts at any node and adds its neighbours and the edge that connects the node and the neighbour to the tree, if the neighbour is not already part of the tree. This procedure is executed recursively with the neighbours, until all nodes of the graph are included in the tree. The nodes are labelled with numbers starting with zero in the order they appear in the depth-first-search.

The number of descendants $ND(v)$ of each node v is also computed recursively. Descendants are the nodes that can be reached in the tree by only travelling edges from parent-node to child-node. Note that a node is always a descendant of itself. The number of descendants of a node is equal to the sum of the descendants of its child nodes plus one (the node itself).

Now $L(v)$ and $H(v)$ are computed, which are the lowest and the highest label reachable from each node v with a path, that only consists of tree edges from parent-nodes to child-nodes and possibly has one edge in the end, that is a non-tree edge of the graph. Again, this can be done recursively. The lowest label, that is reachable from a node by such a path, is equal to the lowest label reachable by such a path from its child nodes or the lowest label of a neighbour-node that is connected with the node itself in the graph, but not in the tree. The same holds for the highest label.

After the computation of the labels, $ND(v)$, $L(v)$ and $H(v)$ for each node, finding bridges is quite simple. A tree edge (u, v), where u is the parent-node and v is a child-node, is a bridge of the graph, if and only if the following formula holds:

$$L(w) = label(v) \wedge H(v) < label(v) + ND(v)$$

The correctness of the formula can be seen as follows:
Only tree edges can be bridges. Proof: Assume that a non-tree edge (u, v) is a bridge. There would be a path from u to v that only consists of tree edges, because every pair of nodes is connected in the tree. That contradicts the assumption.
In the preorder labelling, the descendants (without the node itself) of a node v with label k are labelled with numbers from $k+1$ to $k+ND(v)-1$. The descendants of v are the nodes in the subtree that is rooted by v. The edge (u, v) is a bridge if a node, that is not part of the subtree, is only reachable from v by passing the edge (u, v). That means that no label smaller than $label(v)$ or greater than $label(v)+ND(v)-1$ can be found with the described procedure.

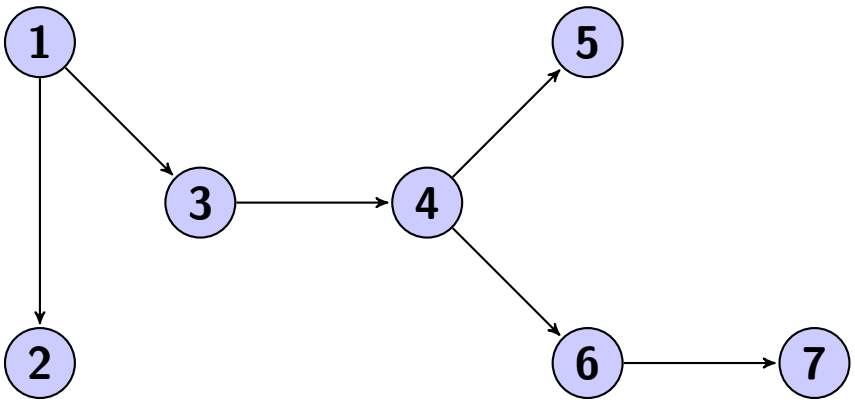

Figure 2.7: A preorder labelling of the graph from Figure 2.1 with usage of the spanning tree from Figure 2.6.

An example is given with the preorder labelling in Figure 2.7, the lowest and highest labels in Figure 2.8, and the logical value of the formula for each node in Table 2.1.

2.2.2 Two-cuts

An algorithm that computes two-cut classes is presented in [5]. It is a Monte Carlo algorithm, which means it has a bound runtime but the result may be incorrect. Though, the probability of getting the right result is $p = 1 - \frac{1}{n}$ for a graph with n nodes. So the probability is close to 100% for big graphs.

The algorithm presented in [5] computes binary circulations of a graph $G = (V, E)$.

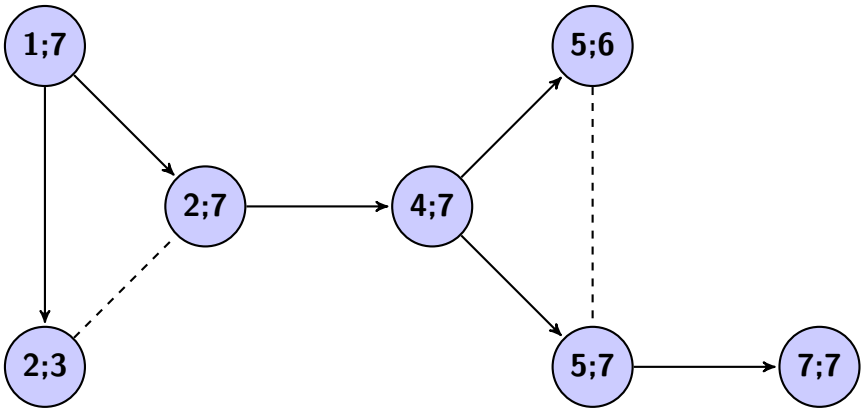

Figure 2.8: The lowest and highest label found in the graph from Figure 2.1 with usage of the spanning tree from Figure 2.6. Tree edges are drawn solid, non-tree edges dashed.

node v	label	ND(v)	L(v)	H(v)	$L(w) = label(v) \wedge$ $H(v) < label(v) + ND(v)$	bridge edge
A	1	7	1	7	true	-
B	2	1	2	3	false	-
C	3	5	2	7	false	-
D	4	4	4	7	true	(C, D)
E	5	1	5	6	false	-
F	6	2	5	7	false	-
G	7	1	7	7	true	(F, G)

Table 2.1: The results of the bridge computing algorithm on the graph from Figure 2.1. A is the root node and therefore has no edge to a parent node that can be marked as a bridge.

Binary circulations are subsets $S \subseteq E$ such that every node in the graph $G_S = (V, S)$ has even degree. If two edges are in the same binary circulations, the probability that they are in the same two-cut class is high. This probability increases with the number of different binary circulations that are computed. The authors use a parameter $b = \lceil log_2(|V| \cdot |E|) \rceil$ to distinguish the number of binary circulations they compute.

The algorithm uses a spanning tree T to compute random binary circulations.
First, a graph G_{bc} is constructed that contains the same nodes as G. Every edge of G that is not in T is added to G_{bc} with probability 1/2. Edges that are in T are not added to G_{bc} in the beginning.
Then, the tree T is travelled from bottom to top. If a node n has odd degree in G_{bc}, the edge that connects n to its parent in T is added to G_{bc}. When the algorithm reaches the root, every node in G_{bc} has even degree. The root has no edge to a parent node that can be added. But it is impossible that exactly one node in a graph has odd degree, because every edge has two endpoints, so the sum of the degrees of all nodes must be even. Since the algorithm provides every node except the root of T with an even degree, the root must have an even degree in the end.

If at least two edges are part of the same set of the b binary circulations, they are put together to a two-cut class C.

The compuation of a random binary circulation is shown in Figure 2.9 and Figure 2.10. The two-cut classes of the example graph from Figure 2.1 is shown in Figure 2.11.

2.2.3 Minimum cuts

The Push-Relabel Algorithm of Goldberg and Tarjan presented in [6] computes minimum s-t-cuts.

The aim of the algorithm is to find a maximum flow from the source node s to the sink node t of a graph $G = (V, E)$. The maximum flow can then be used to find the

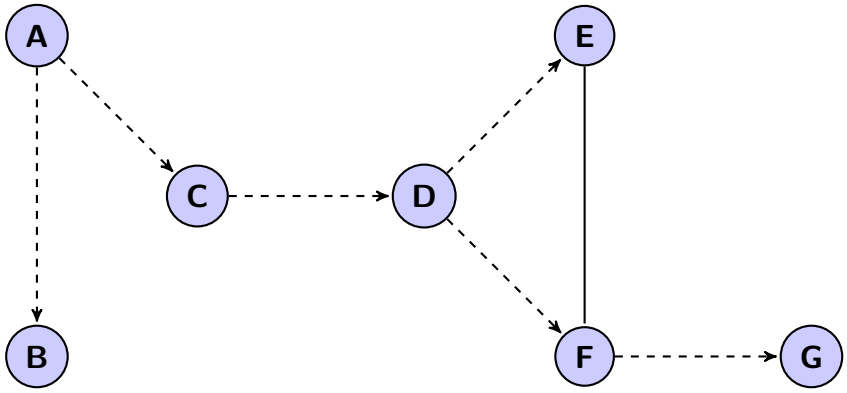

Figure 2.9: Random edges (solid) of the graph from Figure 2.1 that are not in the spanning tree (dashed) from Figure 2.6.

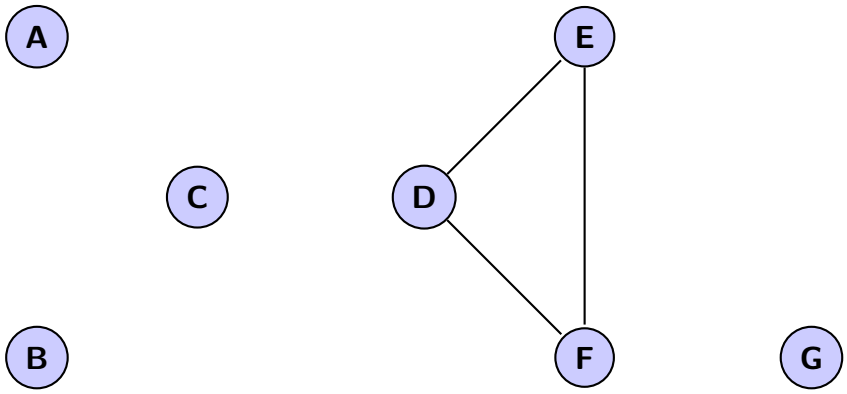

Figure 2.10: The random circulation that results from Figure 2.9.

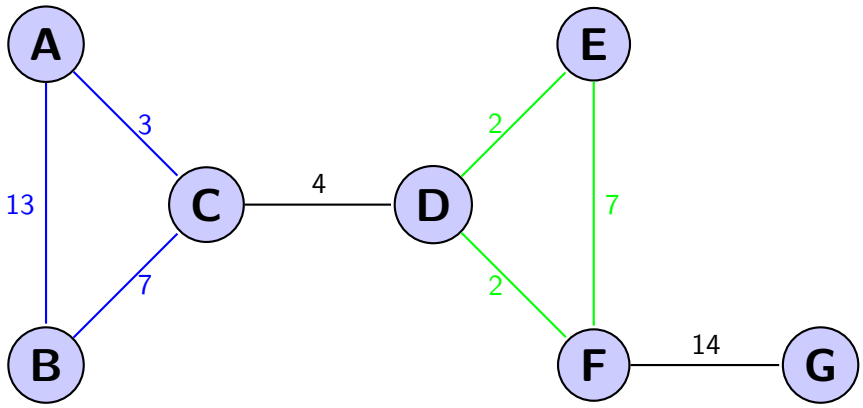

Figure 2.11: The green and blue edges form two-cut classes of the graph from Figure 2.1.

minimum cut.
The weights of the edges distinguish the capacity of each edge:

$$c(u,v) = w(u,v),\ \forall (u,v) \in E$$
$$c(u,v) = 0,\ \forall (u,v) \notin E$$

A flow is a function $f : V \times V \to \mathbb{N}$ that satisfies the following constraints:

$$f(u,v) \leq c(u,v),\ \forall u,v \in V$$
$$f(u,v) = -f(v,u),\ \forall u,v \in V$$
$$\textstyle\sum_{v \in V} f(u,v) = 0,\ \forall u \in V \setminus \{s,t\}$$

The first constraint means that the flow of an edge can not be grater than its capacity. The second constraint forbids positive flow in both directions of an edge. The last constraint defines that the incoming flow must be equal to the outgoing flow of a node. Exceptions are made for the source node, which has only outgoing flow, and the sink node, which has only incoming flow.

While the algorithms runs, it manages a preflow. In a preflow a node can have a positive excess $e(v)$:

$$\textstyle\sum_{v \in V} f(u,v) \leq 0,\ \forall u \in V \setminus \{s\}$$
$$e(v) = \textstyle\sum_{u \in V} f(u,v)$$

This means a node can have more incoming than outgoing flow.

A residual edge $(u,v)_f$ is a directed edge between u and v with $f(u,v) < c(u,v)$. The residual graph is the graph with all nodes and all residual edges.
The algorithm also computes a labelling $h(v)$ of the nodes. The source node is labelled with $|V|$ and the sink node with 0. All other nodes are labelled in a way that their label is always a lower bound of the distance from the node to the sink node, if the sink node is connected to the node in the residual graph.

A vertex is active, if it has a positive excess. In the case that the algorithm shall only compute the minimum cut and not the maximum flow, it is enough to define a vertex to be active if it has a positive excess and its label is smaller than $|V|$. This leads to an earlier termination of the algorithm.

For the initialization, the flow of all edges from the source node is set to the maximum value:

$$f(s,v) := c(s,v)$$

After the initialization, the algorithm applies the push and relabel operations described in algorithm 3 and algorithm 4 to active nodes, until no operation is possible. The push operation pushes excess from an active node to a neighboured node with

Algorithm 3 push(u, v)

assert u is active and h(u) = h(v) + 1
delta = min(e(u), c(u, v) - f(u, v))
f(u, v) = f(u, v) + delta
f(v, u) = f(v, u) - delta

Algorithm 4 relabel(u)

assert u is active and h(u) $\leq$ h(v) for all neighbours v with f(u, v) $<$ c(u,v)
h(v) = min(h(v) for all neighbours v with f(u, v) $<$ c(u,v)) + 1

a lower label, if the edge that connects them is a residual edge. The relabel operation increases the label of an active node, depending on the labels of its active neighbours. After a relabel operation, the node can push excess to at least one neighboured node.

When the algorithm terminates, all nodes that are connected to the sink node in the residual graph form one part of the minimum cut, the other nodes form the other part.
An example of a maximum flow is shown in Figure 2.12. The resulting residual graph and minimum cut are presented in Figure 2.13 and Figure 2.14.

The faster version of the algorithm uses a datastructure called Dynamic Trees (see [7]) and has a runtime of $O(n \cdot m \cdot log(\frac{n^2}{m}))$.
The Dynamic Trees are used to store paths from nodes towards the sink node, where for a node v and its successor u on the path holds that $h(v) = h(u) + 1$. The flow can then be sent directly from the beginning of a path to its end, instead of sending the flow from each node to the successor.

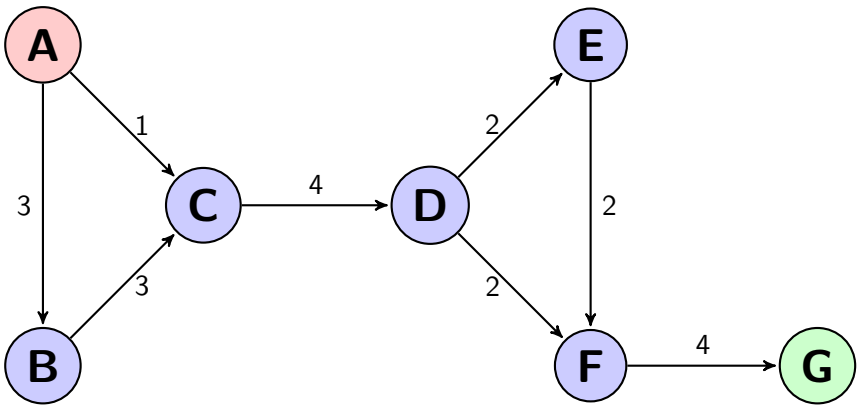

Figure 2.12: The maximum flow of the graph from Figure 2.1, with source node A and sink node G.

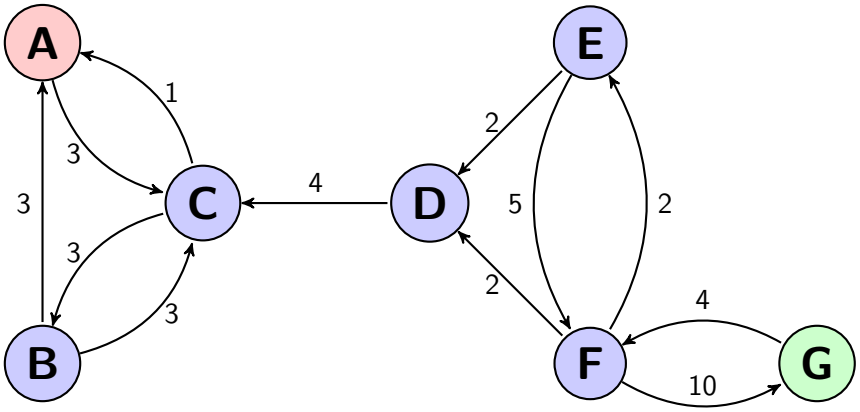

Figure 2.13: The residual graph of the maximum flow from Figure 2.12 with usage of the capacities from Figure 2.1. E and F are connected to the sink G (i. e. paths from E and F to G exist), the other nodes are not.

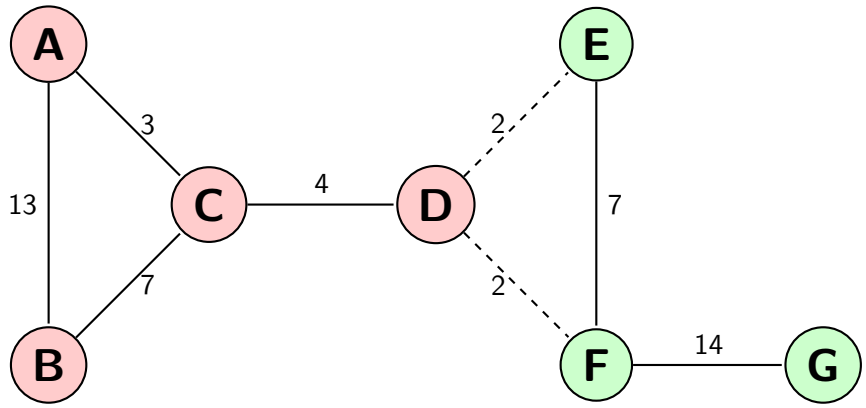

Figure 2.14: The minimum cut between A and G that results from the residual graph from Figure 2.13. The dashed edges are the cut edges.

3 Dataset

3.1 Given format

The dataset contains the schedule of Deutsche Bahn AG of the year 2015.
The data is in the 'HaCon Fahrplan-Auskunfts-System' (HAFAS) format ([8]).
It contains one file with information about the stations of the public transit network of Germany. This file includes station IDs, latitude and longitude coordinates and station names.
The stations are used as nodes of the graph that will be partitioned.
Other files contain the bus and train lines as sequences of stations. An edge between two nodes is added if at least one bus or train line goes from one of the two corresponding stations to the other corresponding station without a stop in between.
The graph is only built from local buses and trains, excluding long-distance traffic. This is achieved by using the vehicle types, which are stored together with each line. Each vehicle type has a category, reaching from category 'A' to 'D'. Category A contains ICEs and TGVs, category B contains ICs, category C contains buses and local trains and category D contains 'others', e.g. ferry boats.
Edges are included, if the corresponding vehicles are from category C or D.
An excerpt of the dataset is presented in Figure 3.1.

```
*Z 00000 vagSBU    01 002 060
*G Bus 0808254 0808282
*L 11    0808254 0808282
*R 1 0808282 0808254 0808282
*A VE 0808254 0808282 001946
0808254 Munzinger Straße, Fr          00607
0808091 Hauptbahnhof, Freibu  00629   00629
0808282 Technische Fakultät,  00639
```

Figure 3.1: Excerpt of the dataset that shows a bus which starts at the station with ID 0808254 at 6:07 a.m.. The next stops have IDs 0808091 and 0808282. The first line contains the information that this bus goes two times with 60 minutes in between. The vehicle type "Bus" is linked to category C in another file. This bus goes on the days that are specified by the bitfield with ID 001946.

line	stations	starting time	repetitions	every ... minutes	days of the calendar
1	A → B → C	8:00 a.m.	13	60	364
2	A → B → C	8:30 a.m.	12	60	260
3	C → B → A	8:00 a.m.	13	60	364
4	C → B → A	8:30 a.m.	12	60	260
5	B → D	10:00 a.m.	4	120	84
6	D → B	10:30 a.m.	4	120	84

Table 3.1: An example timetable. A train goes from station A over B to C every half hour from 8:00 a.m. to 8:00 p.m. on workdays (line 1 and 2), but only every full hour on weekends (line 1). The same happens in the other direction (line 3 and 4). A train from B to D goes on weekends at 10:00 a.m., 12:00 a.m., 2:00 p.m. and 4:00 p.m., another one from D to B every half an hour later (line 5 and 6).

3.2 Edge weights

The edges between stations in the graph are weighted with the number of direct connections between the stations.
Each line can occur multiple times in the dataset when it starts at different times of the day or goes on different days of the year.
Every line of the dataset is linked with an ID of a bitfield. The bitfields are stored in a separate file. The bitfields themselves are hexadecimal numbers. The hexadecimal numbers can be translated to binary numbers. The binary numbers then have length 364 and each position of the number stands for one day of the schedule.
If the bit at position p is set to 1, that means that the lines linked to this bitfield go on the p^{th} day of the schedule.
Therefore, the number of ones in a bitfield is equal to the number of days of the schedule, at which the lines linked to this bitfield go.
Some lines go multiple times a day at intervals of a given length. For example, a train could go at 8:00 a.m. and 20 times every 10 minutes.
This information is also important for the weighting of the edges. If a line goes t times a day at d days in the schedule, it is weighted by $w_{line} = t \cdot d$.
The weights of all the lines that go between two stations are summed up to compute the weight of the edge between the corresponding nodes in the graph.
An example timetable is presented in Table 3.1 and the resulting graph is shown in Figure 3.2.

3.3 Footpaths

After inserting edges for the short-distance trains and buses to the graph, the graph has more than 1,200 connected components.

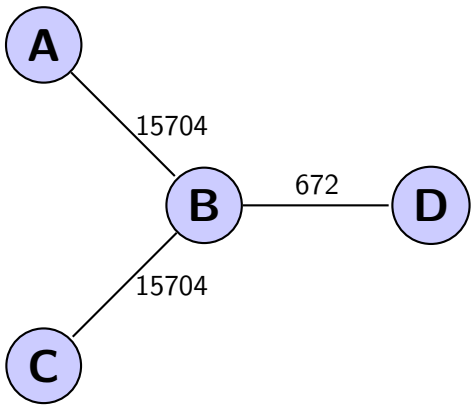

Figure 3.2: An example graph that results from the timetable in Table 3.1.

One reason is that the dataset contains stations that are only connected via long-distance trains, for example stations outside of Germany. The dataset contains long-distance trains that go from stations in Germany to stations outside of Germany or the other direction, but no short-distance transportation of other countries. Therefore, with not including long-distance transportation to the graph, the nodes that correspond to stations in other countries have no edges.
The graph size is reduced by removing nodes with degree zero, that means nodes without edges.

After removing the nodes with degree zero, the graph still has 84 connected components. The biggest connected component contains 99.58% of the nodes of the graph, while the other connected components are rather small.
The reason for the non-connectivity of the graph is that some stations of the public transit network are modelled as multiple stations. For example, the main station of a city can be modelled as one station for the train racks, one for the underground, one for buses, one for trams, and so on. At some points this leads to the fact that single train or bus lines or little parts of the network are no longer connected to the rest.

In order to improve the connectivity of the graph and to model a more realistic situation, footpaths are added to the graph. The dataset does not contain information about footpaths, so a heuristic model is used. Additional edges are added between any two stations with a distance ≤ 400 meters.

The edges corresponding to footpaths get a high weight. With a low weight the algorithms would probably cut the footpaths in order to create a partitioning with a small cut size. A partitioning in which the train tracks and bus station of a main station lies in different partitions seems not to be realistic. This situation is prohibited by introducing footpath edges with a weight of 200,000.
If a footpath is added between two stations that are already neighboured, there is

no new edge introduced, but the weight is added to the weight of the existing edge.

For the selection of pairs of nodes with distance ≤ 400 meters, the nodes are stored in a spatial grid. This has the advantage that not all distances of pairs of nodes have to be computed, but only the distances from all nodes to the nodes that are stored in the same or neighboured grid cells.

3.4 Biggest connected component

Even after adding footpath edges between stations with distance ≤ 400 meters the graph still has 4 connected components.
This is a disadvantage for algorithms that put only connected nodes into the same partition. The size of 3 partitions is bounded by the size of the 3 smaller connected components.
For a better comparison between the algorithms, and also because partitioning a non-connected transportation network is not realistic, the nodes that are not in the biggest connected component are removed from the graph.
What remains is a connected graph that has close to 100% of the nodes of the public transit network of Germany.
Statistics about the datasets are given in Table 3.2 and the distributions of the edge weights are presented in Figure 3.3 and Figure 3.4.

	without footpaths	with footpaths
number of nodes	249,067	250,092
number of edges	420,654	550,385
minimum edge weight	1	1
average edge weight	8,400.36	90,322.62
maximum edge weight	781,457	981,457
total edge weight	3,543,502,016	49,713,302,016

Table 3.2: Comparison of the dataset without and with footpaths.

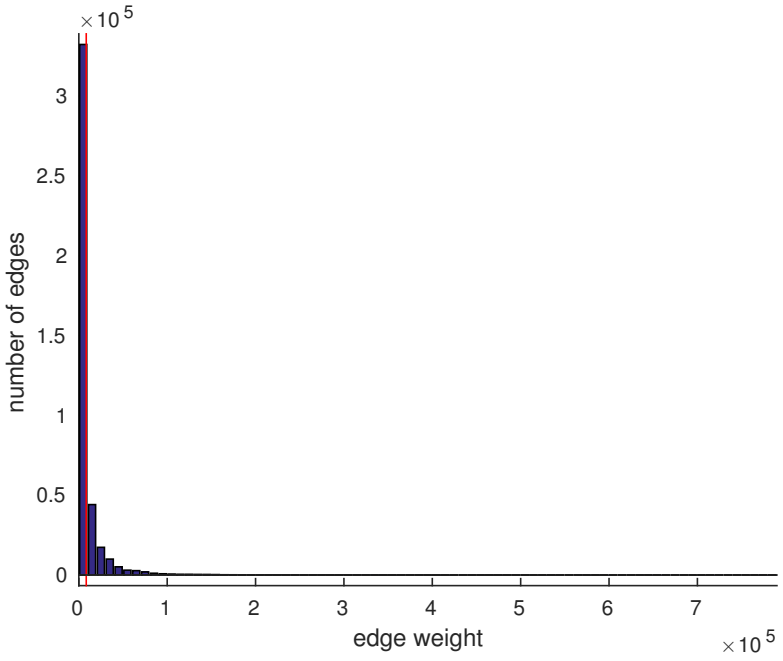

Figure 3.3: Distribution of edge weights without footpaths.

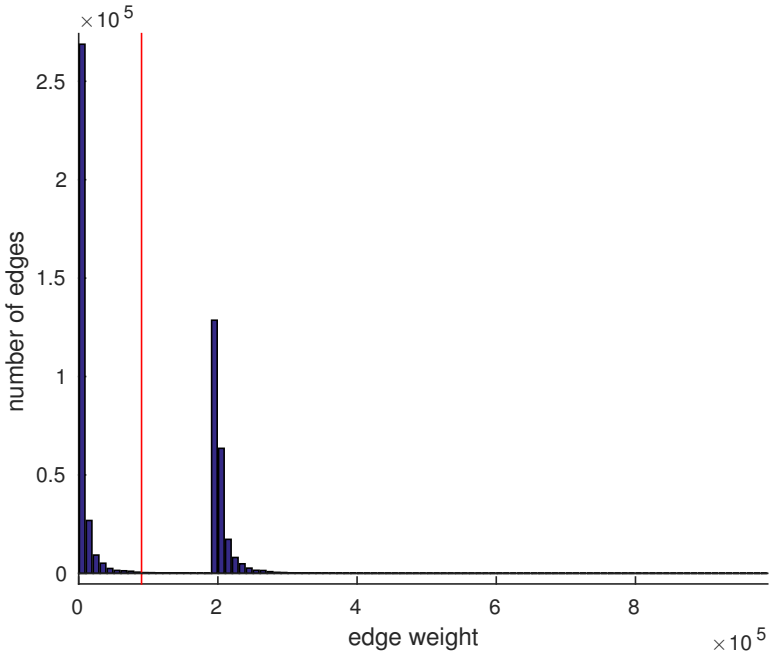

Figure 3.4: Distribution of edge weights with footpaths between stations with distance ≤ 400 meters and weight $200,000$.

4 Algorithms

4.1 K-means-clustering

The k-means-algorithm, which was first published in [9], introduces k clusters and assigns each data point to a cluster such that some distance measure is minimized. K is a hyperparameter that has to be set by the user.
In contrast to the other approaches discussed in this work, k-means does not use the properties of the public transit network as a graph. In particular the edges of the graph and their weights are ignored.
Only the stations of the public transit network are used as input data. Each station has a position, which is stored as latitude and longitude coordinates.
K-means minimizes the overall distance from the data points to the midpoints of the clusters they are assigned to. This means that the algorithm finds a local minimum of the function

$$\sum_{i=1}^{k} \sum_{j=1}^{n} d(x_j, \mu_i) \cdot \mathbf{1}(x_j \in S_i)$$

where d is the distance function, μ_i is the center of the i^{th} cluster and $\mathbf{1}(x_i \in S_j)$ is 1 if x_j is assigned to the i^{th} cluster and 0 else. This function is called the distortion measure.
The distortion measure is minimized in two alternating procedures, as shown in algorithm 5.

Algorithm 5 k-means-clustering

```
initialize
while assignments change do
    update assignments
    update means
end while
```

For the initialization the coordinates of k data points are randomly selected as means of the clusters.
In the assignment step each data point is assigned to the cluster with the mean that is closest to the data point. If all data points are assigned to the same cluster as they were assigned to before, which means that no changes are made, the algorithm stops.

In the update step the means of the clusters are recomputed with the following formula:

$$\mu_i = \frac{\sum_{j=1}^{k} x_j \cdot \mathbf{1}(x_j \in S_i)}{\sum_{j=1}^{k} \mathbf{1}(x_j \in S_i)}$$

The k-means-algorithm is guaranteed to converge to a local minimum of the distortion measure function. The final result depends on the initialization. Different runs can have very different results.
Since every station is assigned to exactly one cluster, we can use the clusters as partitions of the public transit network.
Note that the k-means-algorithm does not necessarily compute connected partitions, since it completely ignores the structure of the network and uses only spatial data to form the clusters.

4.2 Merging algorithm

Van der Horst uses an hierarchical approach to find a partitioning of a road network ([10]).
The road network is modelled as an unweighted graph, but the algorithm can easily be adjusted to use edge weights.

In the beginning, every node of a graph is seen as its own partition with size 1. Neighboured partitions are merged to a single partition until they reach an upper bound size U.
Merging two partitions means to remove them and add a new partition instead, that contains the nodes of both merged partitions.
The order of the pairs of partitions that are merged is determined with a utility function. Whenever the algorithm selects a pair of partitions, it selects the two partitions with the maximum value of the utility function out of the set of pairs of partitions that are neighboured and that together have a size smaller than or equal to U.

The following utility functions are introduced and compared in this work, where $s(u)$ and $s(v)$ are the sizes of partitions u and v and $w(u, v)$ is the sum of the weights of all edges with one endpoint in u and one endpoint in v. In the unweighted case, $w(u, v)$ is the number of edges with one endpoint in u and one endpoint in v.

$$f_2(u,v) = \frac{w(u,v)}{(s(u)+s(v))^2} \tag{4.1}$$

$$f_4(u,v) = \frac{w(u,v)}{(s(u)+s(v))^4} \tag{4.2}$$

$$f_8(u,v) = \frac{w(u,v)}{(s(u)+s(v))^8} \tag{4.3}$$

$$f_{exp}(u,v) = \frac{w(u,v)}{exp(s(u)+s(v))} \tag{4.4}$$

$$f_{PUNCH}(u,v) = \frac{w(u,v)}{\sqrt{s(u)}} + \frac{w(u,v)}{\sqrt{s(v)}} \tag{4.5}$$

$$f_{improved}(u,v) = \frac{1}{s(u)\cdot s(v)} \cdot \left(\frac{w(u,v)}{\sqrt{s(u)}} + \frac{w(u,v)}{\sqrt{s(v)}}\right) \tag{4.6}$$

$$f_{equal}(u,v) = \frac{1}{\sqrt{s(u)}} + \frac{1}{\sqrt{s(v)}} \tag{4.7}$$

The utility functions f_2, f_4, and f_8 are inspired by [10]. The utility function f_{PUNCH} is taken from [1]. $f_{improved}$ is like f_{PUNCH}, but with a balancing factor that is also used in [10].

Since the aim of the algorithm is to find a partitioning with small partitions and small edge weights between the partitions, the utility functions punish big partitions and reward pairs of partitions with high edge weights between them. After merging two partitions, the edges between them are no longer cut edges in the partitioning. It follows that merging partitions with high edge weights between them reduces the cut size more than merging partitions with low edge weights in between.

The algorithm stops, when no pair of neighboured partitions is left, that together has a size smaller than or equal to U.
Instead of the upper bound size U, a parameter k can be introduced, that determines the number of partitions in the final result. Then the algorithm merges partitions until only k partitions are left, without concerning the size of the partitions.
In this work both termination criteria are combined. That means the algorithm runs until k partitions are left or no more pairs of partitions with total size smaller than or equal to U are left. U is set to the total number of nodes and k is set to 1 if only the effect of the other criterion is tested.

4.3 METIS

Karypis and Kumar present a framework for graph partitioning in [11], that is called METIS. It can be downloaded from their webpage ([12]) and used as a standalone program.

The program runs on an input file that stores the given graph. The output is a file, that includes the partitioning, that was found by the program.
The program takes an integer k as an input argument, that determines the number of partitions in the resulting partitioning.
The option *-contig* can be used to enforce the algorithm to compute contiguous partitions. This means, that in the resulting partitioning, for any pair of nodes of the same partition there must be a path that connects the two nodes and only induces nodes of this partition.
With the option *-ufactor=u* a disbalancing ratio r can be defined, that is computed as follows:

$$r = 1 + \frac{u}{1000}$$

The ratio distinguishes how much the size $s(p)$ of a partition p can differ from the average size a.

$$s(p) \leq r \cdot a$$

The program uses a multilevel graph partitioning scheme, that runs in three phases. The first phase is the coarsening phase, where the size of the graph is reduced. In the second phase, a partitioning of the coarse graph is found. In the last phase, the partitioning is projected back to the original graph and refined.
The three phases are shown in Figure 4.1.

4.3.1 File formats

To use the METIS program, the input graph must be stored in a textfile. The format of the input file is described in the METIS manual ([13]).

The first line of the input file is a header line.
It contains the number of nodes and the number of edges of the graph, as well as a bitsequence of length 3, that describes the graph's format. The three bits determine whether the graph has vertice weights, vertice sizes and edge weights. For the partitioning of public transit networks, the bitsequence is set to 001, which means that the graph has edge weights, but no vertice weights and vertice sizes.

The following lines store the edges of the graph. Each line stores the edges adjacent to one vertex. The line consists of an even number of integers that are separated by blank spaces. Every integer i at an odd position stands for an edge to the node that is stored in the i^{th} line. The following integer stands for the edge weight.
To store an undirected graph, every edge (u, v) must appear in the line of u and in the line of v.

The output file simply contains one line per node, that stores an ID of the partition the node is assigned to.

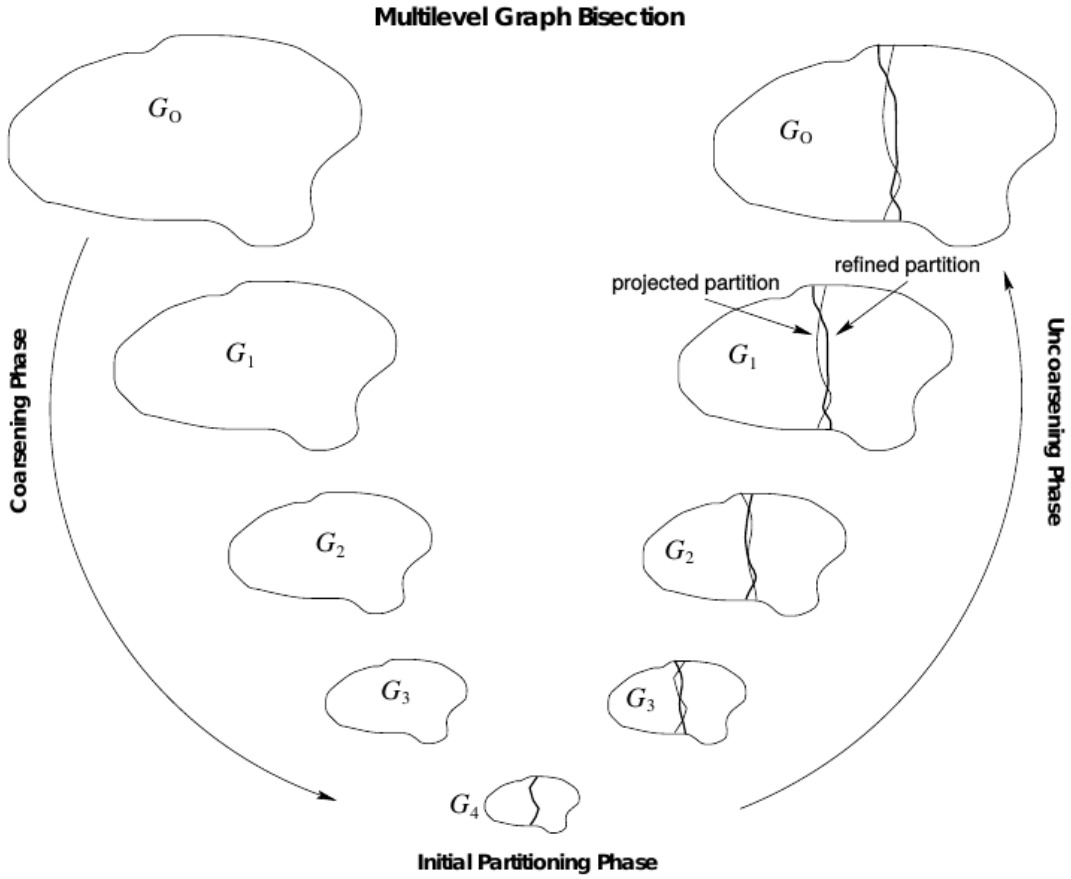

Figure 4.1: The three phases of METIS. (Source: [11])

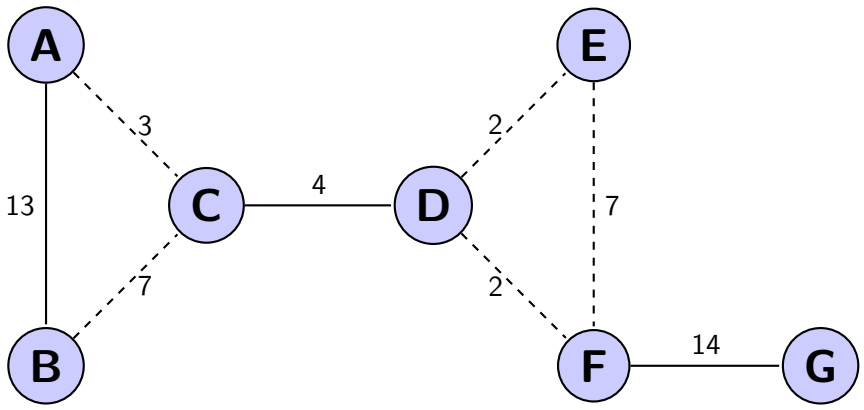

Figure 4.2: A maximal matching (solid) of the graph from Figure 2.1 (dashed).

4.3.2 Coarsening phase

During the coarsening phase, the size of the graph is reduced iteratively.
In each iteration, pairs of neighboured nodes of the graph $G_i = (V_i, E_i)$ (where G_1 is the initial graph) are selected and each pair is combined to a single node of the graph $G_{i+1} = (V_{i+1}, E_{i+1})$ of the next iteration. To select many pairs at once, a maximal matching is computed.
A matching of a graph $G = (V, E)$ is a subset $S \subseteq E$, such that no multiple edges in S are adjacent to the same node. A matching S is maximal, if every edge in E has at least one endpoint that is adjacent to an edge in S. It follows, that in a maximal matching S no more edge of E can be added to S without destroying the property that S is a matching. An example of a maximal matching is provided in Figure 4.2.
A random maximal matching can be computed by visiting the nodes in V in random order and adding one edge to S, that is randomly selected out of the adjacent edges with endpoints that are not already adjacent to an edge in S, if at least one such edge exists.
In the METIS program, the nodes are visited in random order, but instead of selecting the edges randomly, the edge with the highest weight is selected. It follows, that edges with smaller weights are preserved in the coarser graph. Since the initial partition is computed on the coarsest graph, a partition with a smaller edge-cut can be found.

4.3.3 Initial partitioning phase

In the initial partitioning phase, a partitioning of the coarsest graph $G_m = (V_m, E_m)$ is computed. A partitioning with two partitions can be computed by growing one partition with a breadth-first-search started at a randomly selected vertex. The breadth-first-search stops when the partition contains half of the nodes of V_m.
A partitioning with k partitions can be computed by computing a partitioning with

two partitions and then recursively partitioning the partitions with the same procedure. After $log_2(k)$ recursive phases, the graph is partitioned into k partitions.

The initial partitioning is then refined with the Kernighan-Lin Algorithm (see [14]). The Kernighan-Lin Algorithm swaps nodes between the partitions, if such a swap reduces the cut size.

4.3.4 Uncoarsening phase

The uncoarsening phase runs iteratively again. In each iteration, the partitioning of the graph $G_{i+1} = (V_{i+1}, E_{i+1})$ is transformed to a partitioning of the less coarse graph $G_i = (V_i, E_i)$. After the transformation, the partitioning is again refined with the Kernighan-Lin Algorithm during each iteration.

4.4 PUNCH

Delling et al. developed an algorithm in [1], that they called 'Partitioning Using Natural Cut Heuristics' (PUNCH). It consists of two phases.
The first phase is the filtering phase. During this phase, the input graph is shrunk without changing its natural properties. In the second phase, that is called assembly phase, an initial partitioning of the shrunk graph is computed. Afterwards, some parts of the shrunk graph are revisited in order to improve the partitioning.
The idea behind the algorithm is that the partitioning can be computed faster on the shrunk graph than on the original graph. Since the aim is to find a partitioning on the original graph, it is important that a good partitioning of the shrunk graph can be transformed to a good partitioning of the original graph. For that reason the filtering phase must preserve the structure of the graph.

4.4.1 Filtering phase

The aim of the filtering phase is to reduce the size of the graph while preserving the structure of the graph. This is done by contracting edges that are not likely to appear in any cut of the graph.
Several cuts of small regions of the graph are computed and the edges that are part of these cuts are preserved, but the other edges are contracted.
The filtering phase has four passes: first it computes bridges of the graph, second it contracts simple paths, third it finds two-cuts and fourth it computes so-called natural cuts.
The nodes, that remain after the filtering phase, are called fragments.

4.4.1.1 Bridges

In the first pass of the filtering phase, small regions of the graph, that are separated from the rest of the graph by bridges, are contracted.
The bridges are computed with the algorithm described in subsection 2.2.1.

The bridge-connected components of the graph form a tree. The biggest component is chosen as the root of the tree. The tree is then travelled from top to bottom. Whenever a subtree of size smaller than or equal to U is entered, all nodes of this subtree are contracted.
This procedure reduces the number of nodes but preserves the bridges of the graph, which are important edges for finding a partitioning.

4.4.1.2 Simple paths

During the second pass of the filtering phase, paths induced by nodes of degree two are contracted. This comes from the idea that it is not important which one of the edges of such a path is cut, and that there is no need to cut multiple edges of such a path in order to disconnect its endpoints.
In a public transit network many such paths can be found, because there are stations that are only served by a single train or bus line.
The procedure is very intuitive: take neighboured nodes with degree two and total size smaller than or equal to U and merge them, until no more mergable pairs are left.

4.4.1.3 Two-cuts

After contracting bridge-separated regions and paths of nodes with degree two, the next step is to find two-cut classes. The two-cut classes are computed with the algorithm described in subsection 2.2.2.
For each two-cut class C the connected components of the graph $G_C = (V, E \setminus C)$ are computed. This means a graph similar to G is built, that does not contain the edges of the two-cut class, and then the connected components of that graph are computed.
For reasons of simplicity, all two-cut classes are applied at once in the implementation of this work.
If the size of a connected component of G_C is smaller than or equal to U, the nodes of the component are contracted in G.

4.4.1.4 Natural cuts

The aim of the last pass of the filtering stage is to find what Delling et al. call 'natural cuts' ([1]). A natural cut separates a region of the graph in a 'natural' way,

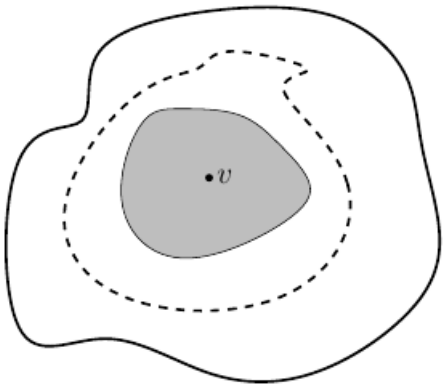

Figure 4.3: Finding a natural cut that separates node v from the rest of the graph. The grey part is the core, the solid line is the ring. The dashed line is the minimum cut between core and ring, i. e. the natural cut. (Source: [1])

like a river or another impassable area does.
One could select two nodes of the graph and compute a minimum cut between them, but this would produce many trivial cuts that separate a single node from the rest. In order to separate bigger regions of the graph, the authors choose two parameters

$$0 < \alpha \leq 1$$
$$f > 1$$

and start a breadth-first-search from a node. They combine all nodes that are visited before the sum of the sizes of the visited nodes exceeds $\frac{\alpha \cdot U}{f}$ to a source node of the minimum cut. The breadth-first-search goes on while the sum of the sizes of the visited nodes is smaller than or equal to $\alpha \cdot U$. After the breadth-first-search stops, the neighbours of the visited nodes, that themselves are not visited, form the target-node of the minimum cut.
The minimum cut is then computed with the algorithm described in subsection 2.2.3. The selection of the core and ring is presented in Figure 4.3.

The algorithm starts breadth-first-searches from nodes that have not been transformed to a source node and computes minimum cuts until every node is at least once transformed to a source node.
The connected components of the graph $G_C = (V, E \setminus C)$ are computed, where C is the union of the cut edges of all natural cuts. Each connected component is contracted.
The constraints that α is smaller than or equal to one and that each node is at least inside one natural cut imply that no connected component is bigger than U. That is because every node is in the inside of at least one cut and the cut regions have a maximum size of $\alpha \cdot U \leq U$.

4.4.2 Assembly phase

In the assembly phase an initial partitioning is found in a greedy fashion. Afterwards the solution is refined with a local optimization.

4.4.2.1 Initial partitioning

The initial solution is found with an algorithm that is similar to the merging algorithm described in section 4.2.
In the beginning, every node of the filtered graph is its own partition. The size of the node refers to the size of the partition.
Neighboured partitions are merged, if their size is together smaller than or equal as the upper bound U. The utility function f distinguishes the order of the pairs of partitions that are merged. The function is randomized with a term $r(1, 1.01)$ that stands for a uniformly distributed number between 1 and 1.01.

$$f(u, v) = (\frac{w(u,v)}{s(u)} + \frac{w(u,v)}{s(v)}) \cdot r(1, 1.01)$$

The pair of partitions u and v that maximizes the function $f(u, v)$ is chosen to be merged, where $w(u, v)$ is the weight of the edge between u and v and $s(u)$ and $s(v)$ are the sizes of the nodes u and v.
The algorithm stops when no more pairs of partitions exist that together have a size that is smaller than or equal as U.

4.4.2.2 Local optimization

After the greedy solution is found, small parts of the graph are uncontracted and revisited in order to improve the solution.

In every refinement step an edge is chosen and its two adjacent partitions are uncontracted into their fragments. Then the merging algorithm is run again on the uncontracted part of the graph. This can be done in three different ways, which are shown in Figure 4.4:

1. uncontract the partitions and run the merging algorithm on the uncontracted nodes
2. uncontract the partitions and run the merging algorithm on the uncontracted nodes and their contracted neighbours
3. uncontract the partitions and their neighboured partitions and run the merging algorithm on the uncontracted nodes

After each refinement step, the old and the new solution are evaluated. If the new solution has a smaller cut size than the old solution, the new solution is kept for the next refinement step. Otherwise the new solution is discarded and the old solution is kept.
The optimization phase runs until every edge was tested once.

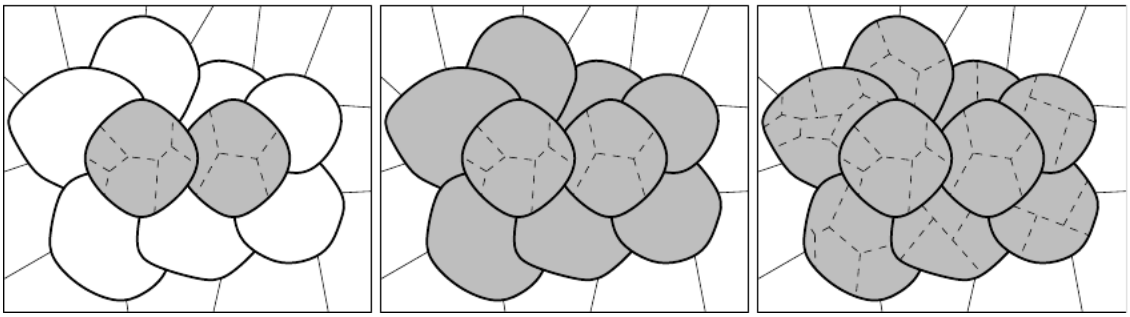

Figure 4.4: Three ways of uncontracting partitions in PUNCH. Partitions are separated by solid lines. Fragments are separated by dashed lines. The reoptimization step is run on the grey parts. (Source: [1])

4.4.3 Further improvements

4.4.3.1 Multistart

Since the selection of the cores during the computation of natural cuts, the initial partitioning phase and the local optimization are randomized, multiple runs of the algorithm may have different results.
It can be useful to run the algorithm multiple times and take the best outcome as the final result.

4.4.3.2 Combination

In order to get multiple results that differ from each other, an evolutionary programming technique is used. A set of solutions is maintained over multiple runs of the algorithm.

After a new solution is found by the algorithm, it is combined with a randomly selected solution of the set of solutions. During combination, another solution is built, that shares the features of the new and the old solution. The combinatorial solution adds edges to the set of cut edges in a probabilistic fashion. The probability of adding an edge is high for edges that are in both solutions, which means both solutions agree that this edge is a cut edge. It is medium for edges that are in one of the solutions, because these are unsure, and low for edges that are not in any of the solutions.

After the combinatorial solution is created, it is compared to the solutions of the set of solutions. If all solutions in the set are better than the combinatorial solution, the combinatorial solution gets discarded. Otherwise the set of solutions that are worse than the combinatorial solution is built, and the most similar solution of this set is replaced by the combinatorial solution. Similarity is distinguished by the number of edges that are in both solutions.

This procedure can run for a given number of iterations or until the last k combinatorial solutions were not added to the set of solutions.
In the end the solution with the lowest cut size is taken as the final result.

4.4.3.3 Parallelization

To speed up the algorithm, some parts of it can be parallelized.
The computation of cores and minimum cuts to find natural cuts can be done in parallel.
The local optimization of different parts of the graph can run on multiple cores in parallel.
For the multistart and combination the algorithm can run on multiple cores to produce new solutions.

4.5 Overview

The following table gives an overview about the properties of the different algorithms. It says whether they have a hyperparameter k to distinguish the number of partitions and a hyperparameter U to set the upper bound partition size, which data model they use and if they are randomized, whether they use a multilevel approach (which means they coarsen the graph before partitioning it and refine the result in the end) as well as what happens in the different phases.

algorithm	k	U	data model	randomized	multilevel	coarsening	partitioning	refinement
k-means	yes	no	spatial data	yes	no	-	sets means of clusters such that the distortion measure is minimized, assigns nodes to the cluster with the closest mean	-
Merging	yes	yes	graph model	no	no	-	merges neighbours that are highly connected using a utility function	-
METIS	yes	no	graph model	yes	yes	iteratively finds maximal matchings and contracts matched nodes	runs a breadth-first-search until 1/2 of the nodes are visited, then runs the Kernighan-Lin-algorithm	runs the Kernighan-Lin-algorithm for each iteration of the coarsening phase
PUNCH	no	yes	graph model	yes	yes	contracts regions separated by tiny cuts or natural cuts	merges neighbours that are highly connected using a utility function	uncontracts small regions and reruns the merging algorithm to find a new partitioning

5 Evaluation

K-means, the merging algorithm and PUNCH were implemented in C++ and compiled with *g++* with the optimization flag *O3*. METIS was downloaded from [12] and installed.
All algorithms were run on a single core of a *Intel Core i7-2670QM* processor with 2.20 GHz and 6 GB of RAM.
The runtimes that are reported in this section contain only the processing time of the algorithms. Neither time for input and output nor for the evaluation of the algorithms is included.

5.1 Criteria

The criteria for a "good" partitioning are not easy to define and depend on the application that will use the partitioning.
The following criteria are used in this section:

- the **number of partitions**
- the **maximum partition size** - the number of stations in the biggest partition
- the **cut size** - the sum of the weights of all edges with adjacent nodes in different partitions
- the **number of cut edges** - the number of edges with adjacent nodes in different partitions
- the **number of border nodes** - the number of nodes with at least one neighbour in another partition

5.2 Baseline

The given dataset contains a file that stores assignments of every station of the public transit network to a local transport company. This information is used to create a baseline partitioning. The statistics of the baseline evaluated on the datasets without footpaths and with footpaths are shown in Table 5.1.

	without footpaths	with footpaths
number of partitions	179	181
maximum partition size	33,230	33,302
cut size	94,519,779	5,540,781,631
number of cut edges	20,628	44,353
number of border nodes	21,643	44,285

Table 5.1: Statistics of the baseline on both datasets.

5.3 K-means: does it even optimize the cut-size?

Since the k-means algorithm ignores the graph structure of the given public transit network and instead uses only spatial data to minimize the distortion measure, it may be questioned whether the algorithm does optimize the cut size and the maximum partition size.

Figure 5.1 shows the evolution of the cut size for multiple runs of the k-means algorithm with different choices of the hyperparameter k. During all runs, the cut size is improved fast in the first 20 to 30 rounds. In the later rounds it has up- and downturns with little improvement.
Separate from the up- and downturns, similar behaviour can be seen in Figure 5.2 for the distortion measure. One can conclude that the cut size is optimized together with the distortion measure.

Figure 5.3 shows the evolution of the maximum partition size for multiple runs with different choices of k. Again, fast improvements are made in the first rounds. After 10 to 20 rounds, a point is reached where the maximum partition size stays the same.

Due to the up- and downturns of the cut size and the steadiness of the maximum partition size, the algorithm sometimes creates solutions in the intermediate rounds, that are better than the final solution. It can be useful to store the solution with the smallest cut size and return it in the end, instead of returning the final solution, that can have a bigger cut size.

The figures also demonstrate that the number of rounds does not depend on the hyperparameter k but on the random initialization.

5.4 Merging with different utility functions

5.4.1 Hyperparameter choice

For every run of the merging algorithm, the hyperparameters k and U have to be set by the user. K distinguishes the minimum number of partitions created by the

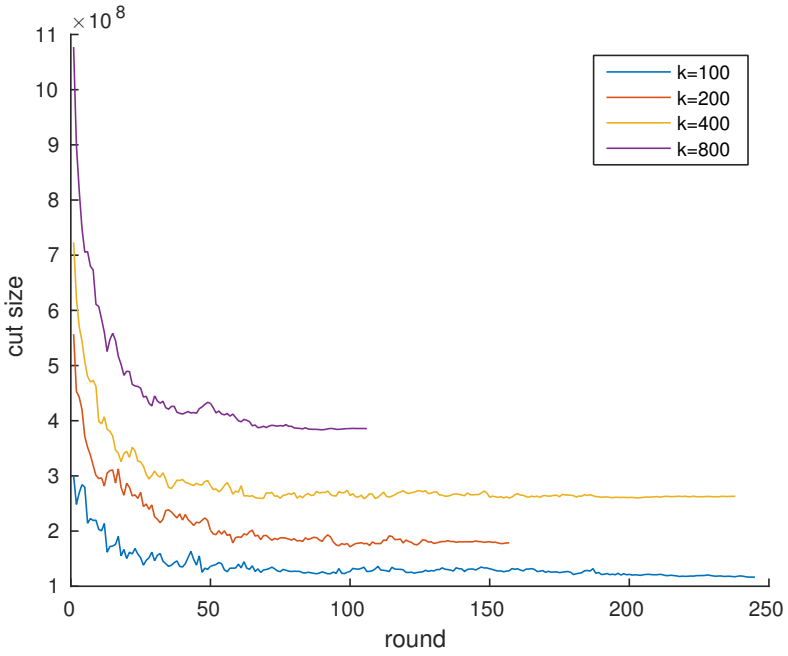

Figure 5.1: Evolution of the cut size for different runs of k-means.

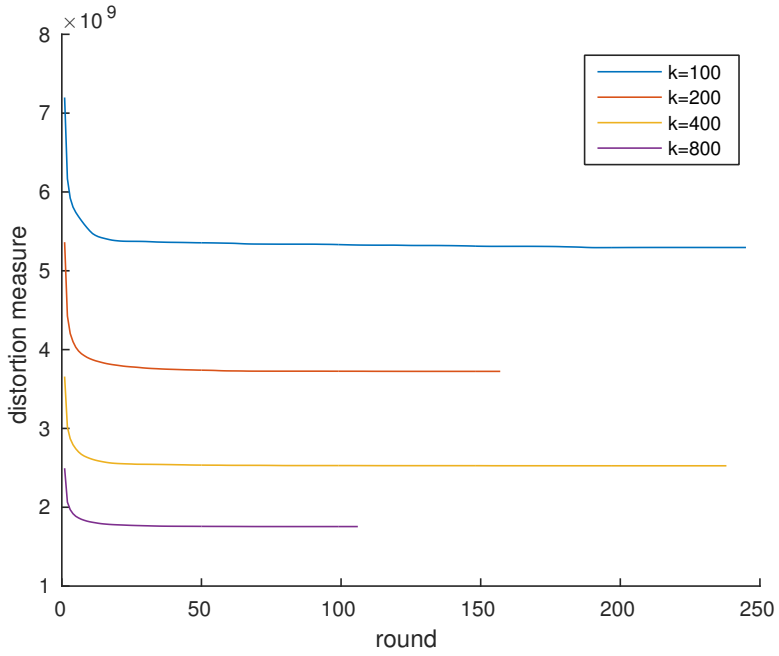

Figure 5.2: Evolution of the distortion measure for different runs of k-means.

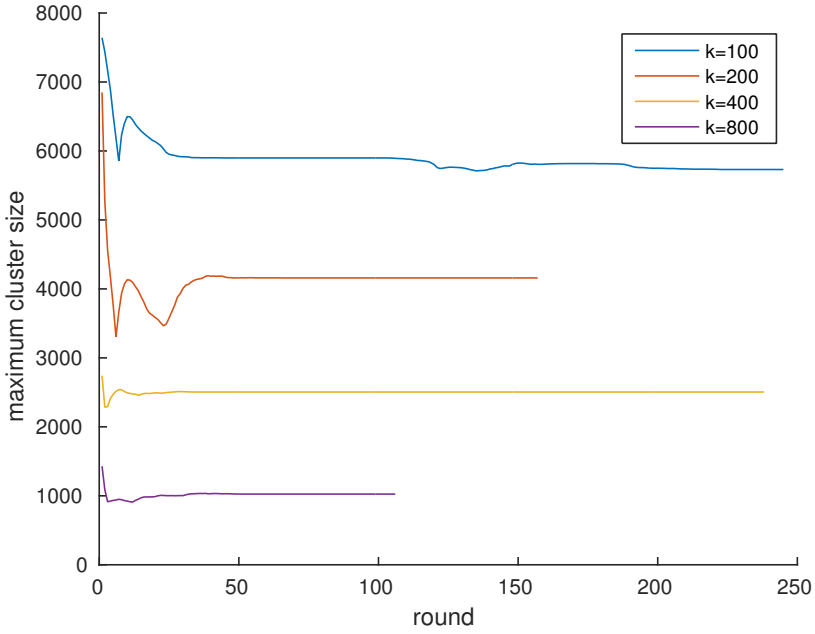

Figure 5.3: Evolution of the maximum partition size for different runs of k-means.

algorithm, and U distinguishes the maximum size of a partition.
Using k and U together might be useful for some applications, but it leads to an earlier termination than in the case where only one hyperparameter is used. Since the algorithm improves the cut size and reduces the number of partitions, the number of cut edges and the number of border nodes in each merging step, an earlier termination is not wanted during the search for optimal solutions. Therefore the algorithm is evaluated with either k or U in this chapter.
When the algorithm is run with k set to a specific value, U is set to the number of nodes in the graph. Therefore, no pair of partitions can violate the constraint that their sizes together must be smaller than or equal to U to allow merging the pair.
When the algorithm is run with U set to a specific value, k is set to one. Therefore, the algorithm runs until no neighbouring pair of partitions exists, that together is smaller than or equal to U.

When the merging algorithm runs with f_2, f_4, f_8, f_{exp} or f_{PUNCH} as the utility function and without the upper bound partition size U set, it creates one big partition, that contains most of the nodes, and many very small partitions. Since a small maximum partition size is a criterion for a good partitioning, these runs seem not useful. Therefore these runs are not included in the plots of this section.

In this section, the utility functions are evaluated with k set to 25, 50, 100, 200, 400 and 800, as well as with U set to $1,000$, $2,000$, $4,000$, $8,000$, $16,000$ and $32,000$.

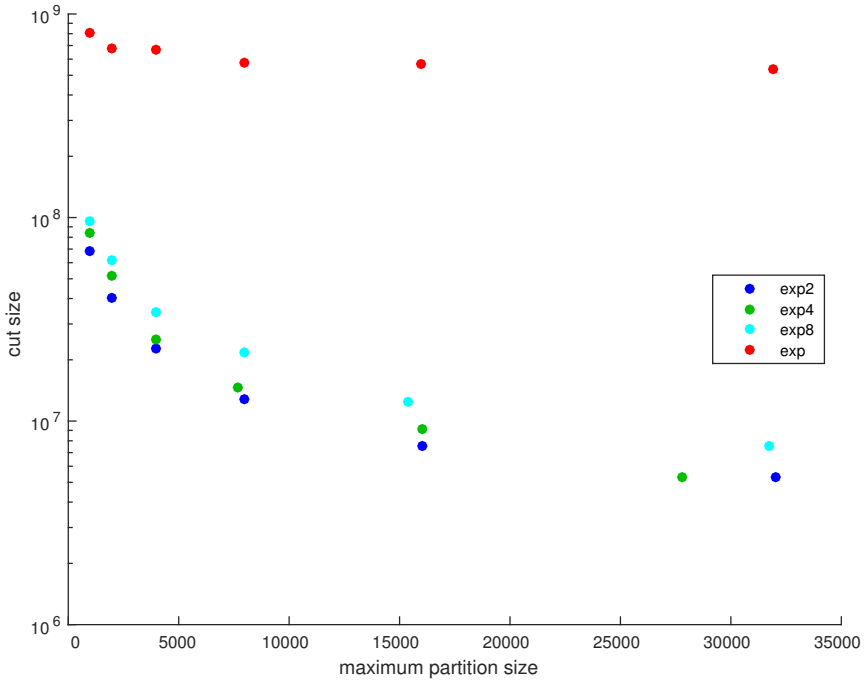

Figure 5.4: Relation between maximum partition size and cut size for the merging algorithm with functions f_2, f_4, f_8 and f_{exp}. Points represent results of runs with different parameters.

5.4.2 f_2, f_4, f_8 and f_{exp}

The utility functions f_2, f_4, f_8 and f_{exp} are of the same form, but have different weightings of the size of the partition in the function's denominator.
f_2 is the most greedy function concerning the cut size. The other functions punish pairs with many nodes more than f_2 does.

Figure 5.4 presents the cut size of results of the merging algorithm with different parameters. It shows that the weighting of the denominator affects the cut size of the resulting partitioning. The more a function punishes bigger partitions, the less is it concerning the weights of the edge between two partitions. Therefore, functions with a higher weighting of the partitions' sizes, create partitionings with a higher cut size. f_2 has the best cut size, followed by f_4 and f_8. f_{exp} has the worst cut size.

Figure 5.5 presents the number of partitions that are created in different runs of the merging algorithm. It shows that the more greedy functions concerning the cut size produce partitionings with more partitions.
When a greedy function is used, the merging algorithm merges pairs of partitions, that are connected by edges with high weights, until they reach the upper bound size U. In the end, many small partitions remain, that are only connected to partitions that already have a size close to U, and therefore can not be merged.
Less greedy functions merge small partitions first. That leads to solutions with less

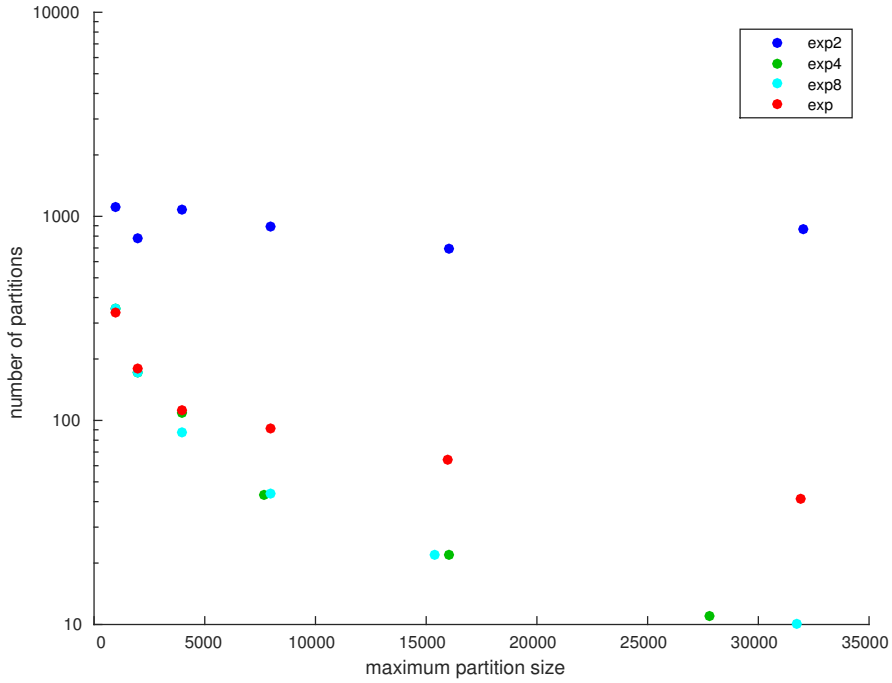

Figure 5.5: Relation between maximum partition size and number of partitions for the merging algorithm with functions f_2, f_4, f_8 and f_{exp}. Points represent results of runs with different parameters.

remaining partitions.
f_2 produces the most partitions, whereas with f_4 and f_8 only a few partitions remain. f_{exp} works well for producing many small partitions, but leads to more partitions than f_4 and f_8 when U is 8000 or greater.

In Figure 5.6 the runtimes of the merging algorithm with the four different utility functions and different parameters is presented.
When run on a graph with N nodes, the merging algorithm initially introduces N partitions. With every merging step, two partitions are removed and one new partition is introduced. So the number of partitions is reduced by one. To find a partitioning with k partitions of a graph with N nodes, the merging algorithm does $N - k$ merging operations. This means the number of operations only depends on the number of partitions in the solution.
The cost of each merging operation depends on the number of edges that are adjacent to the two partitions that are merged. The edges of the new partition must be computed and inserted into the graph. This takes longer if the new partition has many edges. Big partitions normally have more edges, that connect them to other partitions. Therefore, merging big partitions takes more time than merging small partitions. This leads to the assumption that less greedy utility functions, that merge small partitions first, have a better runtime than more greedy utility functions.
Figure 5.6 supports this assumption. The merging algorithm with f_8 has the best

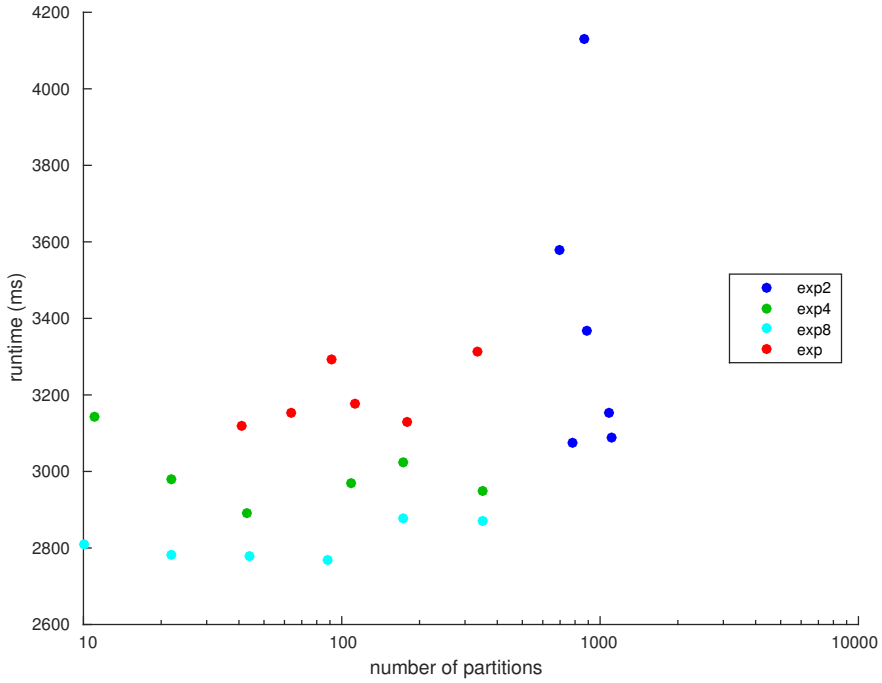

Figure 5.6: Relation between number of partitions and runtime for the merging algorithm with functions f_2, f_4, f_8 and f_{exp}. Points represent results of runs with different parameters.

runtime, followed by f_4. f_2 has the worst runtime.

With f_4 a utility function is found, that leads to good results in all criteria. It has the second-best cut size, produces only a few partitions, cuts the least edges and therefore creates the least border nodes. The runtime is also very fast when f_4 is used.
With f_8 the algorithm is even faster than with f_2, but the results are a little bit worse.
When only the cut size is of interest, f_2 is the best choice out of the four utility functions, but it produces many small partitions.

5.4.3 f_{PUNCH} and $f_{improved}$

The utility functions f_{PUNCH} and $f_{improved}$ look similar, but f_{PUNCH} is more greedy concerning the cut size, whereas $f_{improved}$ has a normalizing factor $\frac{1}{s(u)\cdot s(v)}$ that should lead to more balanced partitions.
Pairs of partitions with unbalanced sizes are merged before pairs of partitions with balanced sizes. For example, a pair with sizes 1 and 199 is punished less than a pair with sizes 100 and 100, even if the partitions, that result from merging any of the two pairs, have the same size.

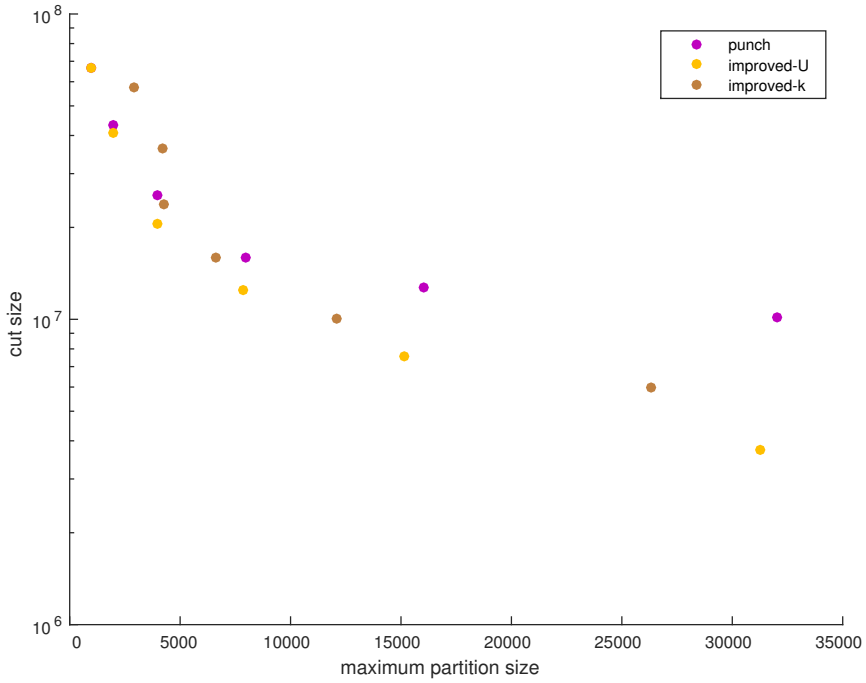

Figure 5.7: Relation between maximum partition size and cut size for the merging algorithm with functions f_{PUNCH} (with U set) and $f_{improved}$ (with k or U set). Points represent results of runs with different parameters.

The utility function $f_{improved}$ can be used either with the hyperparameter k or with the hyperparameter U. Both variants are evaluated in this section.

Figure 5.7 presents the cut size of multiple runs of the algorithm.
$f_{improved}$ with hyperparameter U leads to the best cut size. The results with the same function, but with hyperparameter k, are a little bit worse. When k is used, the result can contain neighbouring partitions with a combined size that is smaller than or equal to the size of the biggest partition. If so, the cut size could be improved by merging these partitions and the maximum partition size would not increase. When U is used, the result is guaranteed to contain no neighbouring partitions with a combined size $\leq U$.
f_{PUNCH} creates a bigger cut size than $f_{improved}$. This might be confusing, because f_{PUNCH} concentrates more on the edge weights than $f_{improved}$ and one could think that this improves the cut size.
An explanation is given with Figure 5.8, which presents the number of partitions created by the merging algorithm with the three utility functions.
Due to the greediness of f_{PUNCH}, many small partitions remain in the end that can not be merged with their neighbours, which have a size close to U.

The runtimes of the merging algorithm with f_{PUNCH} and $f_{improved}$ can be compared in Figure 5.9.
Again, f_{PUNCH} suffers from its greediness and takes much longer than $f_{improved}$.

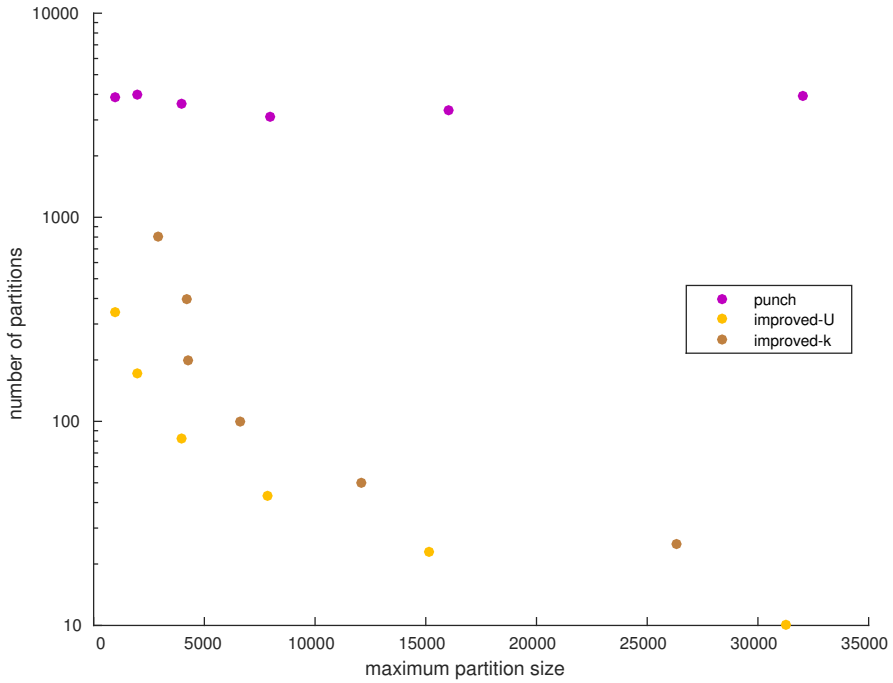

Figure 5.8: Relation between maximum partition size and number of partitions for the merging algorithm with functions f_{PUNCH} (with U set) and $f_{improved}$ (with k or U set). Points represent results of runs with different parameters.

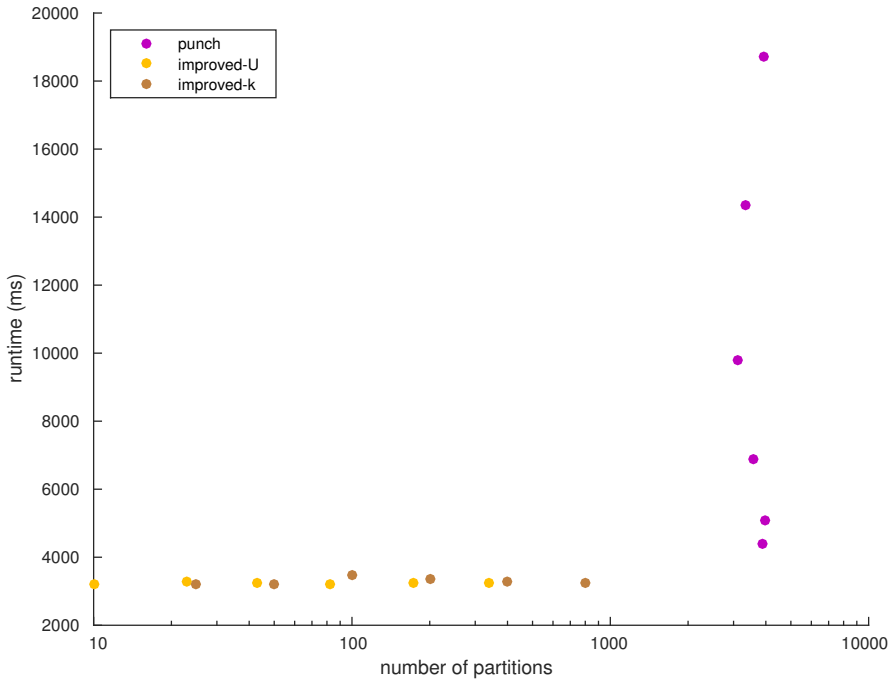

Figure 5.9: Relation between number of partitions and runtime for the merging algorithm with functions f_{PUNCH} (with U set) and $f_{improved}$ (with k or U set). Points represent results of runs with different parameters.

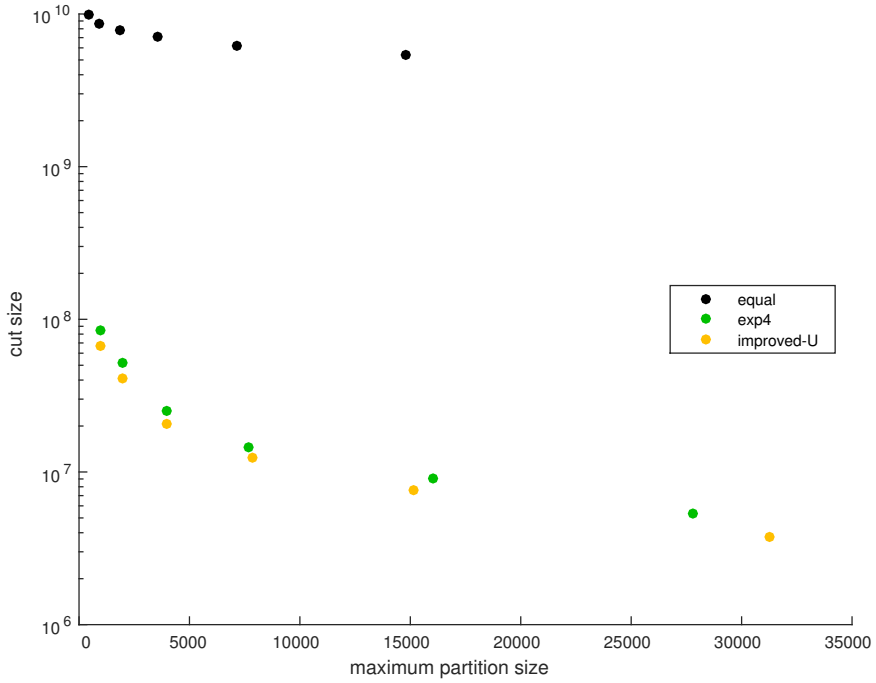

Figure 5.10: Relation between maximum partition size and cut size for the merging algorithm with functions f_{equal}, f_4 and $f_{improved}$. Points represent results of runs with different parameters.

The merging algorithm with the utility function $f_{improved}$ and hyperparameter U produces the best results among the three functions and terminates very fast.

5.4.4 Overall comparison

It was shown that the utility function f_4 creates better results than f_2, f_8 and f_{exp}. $f_{improved}$ was shown to behave better than f_{PUNCH}.
In this section, these two utility functions are compared to f_{equal}.

f_{equal} ignores the edge weights and only tries to create small partitions. This results in a very high cut size (see Figure 5.10).
$f_{improved}$ creates partitionings with lower cut sizes than f_4.

All the three utility functions create comparable numbers of partitions, as shown in Figure 5.11.

Since f_{equal} merges pairs of partitions in ascending order of their sizes, which means that in the beginning only small partitions are merged, it has the best runtime (see Figure 5.12).
The runtime of f_4 is better than the runtime of $f_{improved}$.

If the aim is to find a partitioning with few cells of equal size very fast, f_{equal} is a valid choice as the utility function of the merging algorithm.

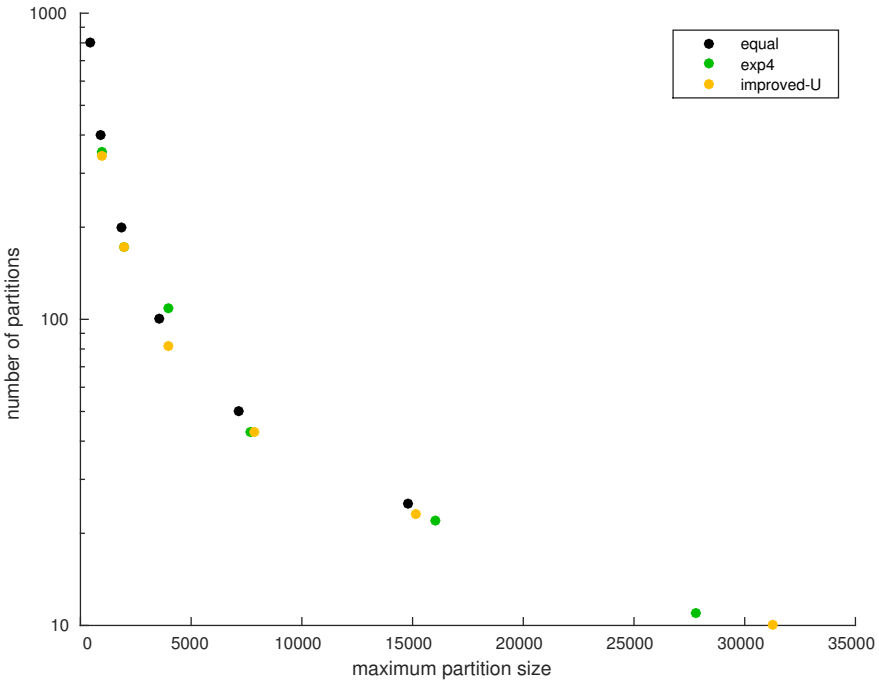

Figure 5.11: Relation between maximum partition size and number of partitions for the merging algorithm with functions f_{equal}, f_4 and $f_{improved}$. Points represent results of runs with different parameters.

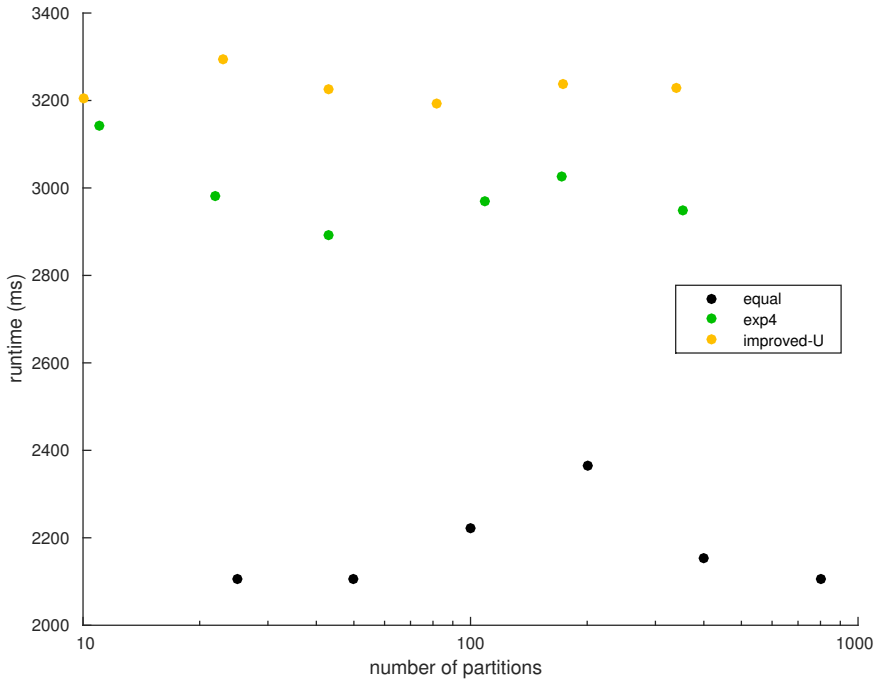

Figure 5.12: Relation between number of partitions and runtime for the merging algorithm with functions f_{equal}, f_4 and $f_{improved}$. Points represent results of runs with different parameters.

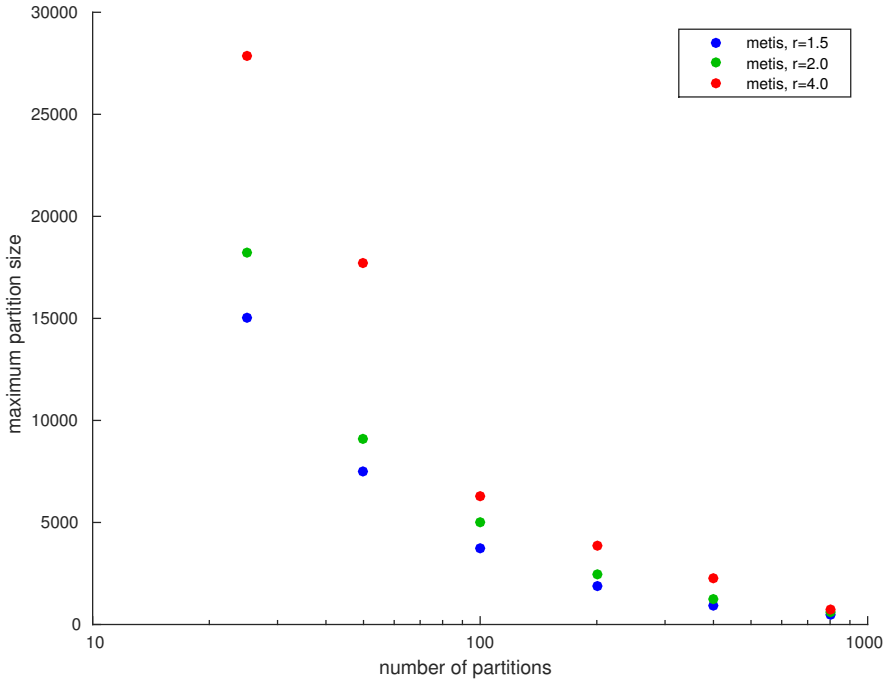

Figure 5.13: Relation between number of partitions and maximum partition size for METIS with different unbalancing ratios. Points represent results of runs with different parameters.

In all other criteria, $f_{improved}$ was shown to create the best results.

5.5 METIS: the unbalancing ratio

With METIS the trade-off between maximum partition size and cut size can be regulated with the unbalancing ratio r.
If r is small, the sizes of the partitions can not be much bigger than the average size. Therefore, small partitions are produced, but big cities might be divided, which leads to a bigger cut size.
If r is big, bigger partitions can be produced, but the algorithm can find a smaller cut size.
The unbalancing ratios 1.5, 2 and 4 are evaluated in this section for k equal to 25, 50, 100, 200, 400 and 800.

Figure 5.13 shows how the maximum partition size increases for a greater r.
With $r = 4$ METIS creates partitions that have double the size of the partitions that are created with $u = 1.5$. The partitions could even be bigger, but METIS does not necessarily exploit the total range of allowed partition sizes.

In Figure 5.14 the cut sizes are presented. The cut size decreases with a greater u.
For very small and very big partitions, the cut size for $u = 1.5$ is very high compared

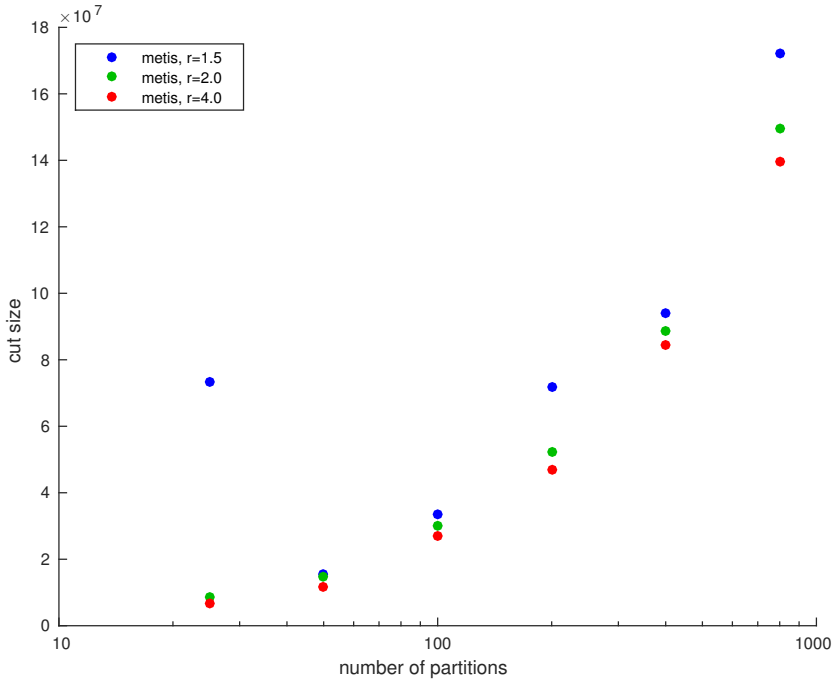

Figure 5.14: Relation between number of partitions and cut size for METIS with different unbalancing ratios. Points represent results of runs with different parameters.

to $u = 2$ and $u = 4$. In between, the cut size does not increase that much when u is chosen small.

5.6 PUNCH

In all runs of PUNCH, the third uncontraction strategy was used during the optimization phase, which uncontracts a pair of partitions and all neighbours. This was shown to create the best results in [1].
The parameter α was set to 0.1 in order to speed up the computation of natural cuts, whereas f was set to 10 as in [1].

5.6.1 The filtering phase

The first pass of the filtering phase takes about 600 milliseconds and reduces the number of nodes of the graph with footpaths from 250,092 to 244,580. The biggest bridge-separated component in the graph has size 87. So the result of the first pass does not depend on the upper bound cell size U, as long as U is greater than or equal to 87.

U	runtime (s)	nodes remaining
1,000	3.0	35,649
2,000	4.3	29,960
4,000	7.1	26,776
8,000	15.2	26,721
16,000	39.6	26,221
32,000	111.4	23,877

Table 5.2: Runtime and number of remaining nodes after the fourth pass of the filtering phase on the dataset with footpaths for different choices of U.

The second pass of the filtering phase goes even faster than the first pass. It takes about 100 milliseconds. The graph size is reduced to 221,989. Again, this is independent from the choice of the hyperparameter U, because the paths of nodes with degree two, that are merged, do not consist of many nodes.

The computation of two-cuts classes in the third pass of the filtering phase is more costly than the first two passes, but the gain is also greater. It takes about 22 seconds and reduces the number of nodes to 54,216. The result is the same for all tested hyperparameters U, because all two-cut-separated regions are smaller than 1,000 nodes, which is the smallest U that was tested.
On the dataset without footpaths the result of the third pass is different with different choices of U. There are two-cut-separated regions with more than $2,000$ nodes in the graph with footpaths.

The computation of natural cuts is a bottleneck in the non-parallelized version of PUNCH.
The runtime depends on the hyperparameter U. If U is great, the minimum cuts are computed on bigger subgraphs, which takes longer. On the other hand, the cores are greater when U is greater. Since the natural cuts are computed until every node was at least once part of a core, less runs of the push-relabel-algorithm are needed. It turns out that the runtime lies between 3 seconds and 2 minutes for U between 1,000 and 32,000.
The outcome also depends on U, because with greater U the merged regions are greater, and therefore less nodes remain.
The exact results can be seen in Table 5.2.

5.6.2 Adaptations for weighted graphs

PUNCH is developed for road networks, that are modelled as unweighted graphs. In [1] is shown that PUNCH produces a small number of partitions and runs very fast on road networks.

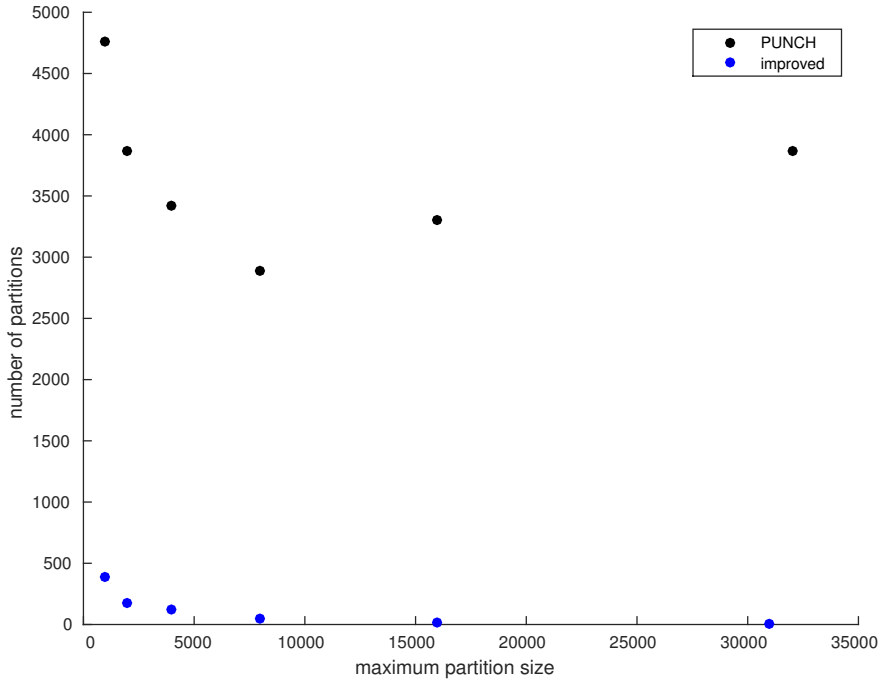

Figure 5.15: Relation between number of partitions and maximum partition size for PUNCH with utility functions f_{PUNCH} and $f_{improved}$. Points represent results of runs with different parameters.

In subsection 5.4.3, the merging algorithm, which is also used in the assembly phase of PUNCH, was tested on the weighted graph that represents the public transit network with footpaths. It was shown that the utility function f_{PUNCH} leads to many partitions and runs slow.
The merging algorithm with $f_{improved}$ is faster and produces better partitionings. As an adaptation for weighted graphs, using a randomized version of $f_{improved}$ in the assembly phase of PUNCH is tested and expected to fasten up the algorithm. The randomized utility function looks like the following:

$$f(u, v) = r(1, 1.01) \cdot f_{improved}(u, v)$$

The randomization term $r(1, 1.01)$ produces random numbers between 1 and 1.01. The number of partitions of the initial solution of PUNCH with randomized versions of f_{PUNCH} and $f_{improved}$ is shown in Figure 5.15. With $f_{improved}$ the number of partitions is significantly smaller than with f_{PUNCH}.

The cut size after the initial solution is not affected much when $f_{improved}$ is used instead of f_{PUNCH}, as shown in Figure 5.16. Indeed, there is one outlier where $f_{improved}$ results in a much worse cut size than f_{PUNCH}.

The runtime of the algorithm is improved when $f_{improved}$ is used.
Since the initial solution contains less partitions, less pairs of partitions are tested

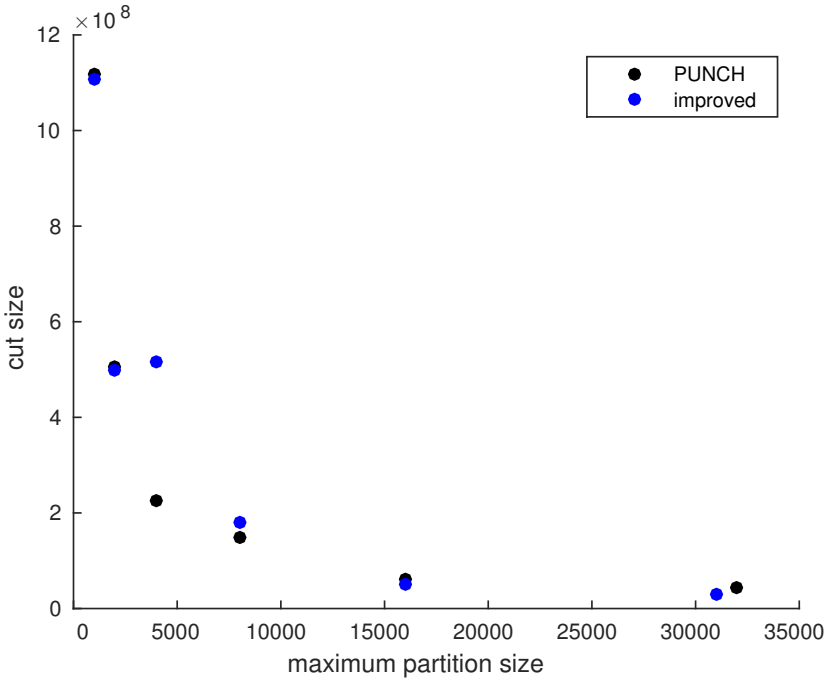

Figure 5.16: Relation between number of partitions and cut size for METIS with different unbalancing ratios. Points represent results of runs with different parameters.

during the local optimization phase. The local optimization steps are also faster with $f_{improved}$, as shown in subsection 5.4.3.
In an example run on the graph without footpaths with $f_{improved}$ and $U = 1,000$, the local optimization phase took 8 minutes and 7 seconds. When f_{PUNCH} was used with the same U, the local optimization phase took 70 minutes and 14 seconds. The cut size produced with $f_{improved}$ was also better. So it seems to be a valid adaptation to use $f_{improved}$ instead of f_{PUNCH} for the assembly phase of PUNCH on weighted graphs.

Another possible adaptation is to increase the randomization. When PUNCH is used for street networks, the weights of all edges are initially set to 1. Therefore, the randomization term $r(1.0, 1.01)$ brings the edges in random order. In the weighted graph model used in this work, the weights differ from 1 to over 700,000 without footpaths, and even more when footpaths are added. Therefore, the randomization term $r(1.0, 1.01)$ has little influence on the order of the edges. Adding more randomization can increase the cut size after the initial solution, but more different solutions are tried in the local optimization phase, which can improve the cut size of the final result.

Table 5.3 shows the cut sizes of the final results of PUNCH, when executed on the graph with footpaths, with $U = 1,000$ and the utility function $f_{improved}$ with different randomization terms. Each parameter setting was tested multiple times and the

r	cut size (in.)	cut size (fin.)	part. (in.)	part. (fin.)	runtime
$r(1, 1.01)$	1,106,215,655	1,100,042,443	387	391	2 min 31 sec
$r(1, 1.10)$	1,109,642,160	1,066,319,778	387	491	6 min 47 sec
$r(1, 2.00)$	1,140,211,963	960,667,235	382	574	8 min 00 sec
$r(1, 11.00)$	1,145,235,708	943,703,125	423	698	10 min 45 sec
$r(1, 101.00)$	1,153,222,445	951,516,392	435	690	8 min 30 sec

Table 5.3: The cut size of the initial solution, the cut size of the solution after the optimization phase, the number of partitions of the initial solution, the number of partitions after the optimization phase and the runtime of the optimization phase with different randomization terms.

best final result is reported in the table.
A greater randomization term leads to a greater cut size after the initial solution. When $r(1, 101)$ is used instead of $r(1, 1.01)$, the cut size of the initial solution increases by 4.3%. During the local optimization phase, better solutions are found with more randomization. With $r(1, 101)$ the initial cut size is improved by 17.5% during the local optimization phase, which is much more than the improvement by 0.5% that is made with $r(1, 1.01)$. The cut size of the final result was improved by 13.4% when $r(1, 11)$ was used instead of $r(1, 1.01)$.
A greater randomization term leads to longer runtimes, because new solutions are accepted more often, which means more new partitions are created and more edges are inserted, which then are also tested in an optimization step.
Even if the cut size decreases with more randomization, the algorithm creates more partitions during the assembly phase, which is a disadvantage of this adaptation. PUNCH created 698 partitions with $r(1, 11)$, but only 391 partitions with $r(1, 1.01)$. Therefore, a greater randomization term should only be used when the number of partitions in the result is not important.

5.7 Comparison of the algorithms

In this section the results of the four algorithms are analysed for several runs with different parameters.
K-means and METIS are executed with k equal to 25, 50, 100, 200, 400 and 800. For k-means the result with the smallest cut size out of three runs is reported for every choice of k. The unbalancing factor u for METIS was chosen equal to 4.
The merging algorithm is executed with $f_{improved}$ as the utility function. PUNCH is executed with $f_{improved}$ as the utility function as well and with the randomization term $r(1, 1.01)$. The upper bound partition size U for the merging algorithm and PUNCH is chosen equal to 1,000, 2,000, 4,000, 8,000, 16,000 and 32,000.

Figure 5.17 shows the relation between the number of partitions and the maximum

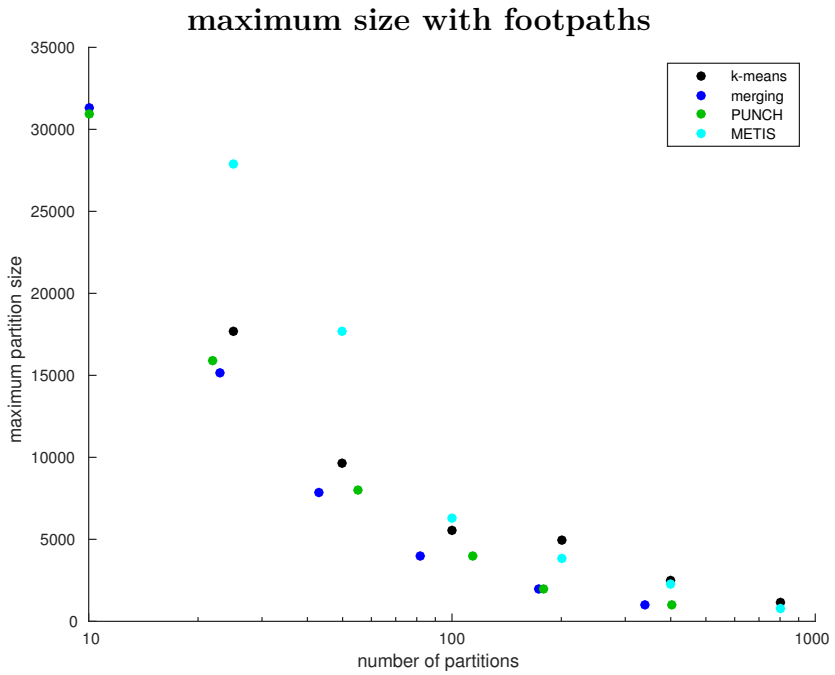

Figure 5.17: Relation between maximum partition size and number of partitions for the algorithms. Points represent results of runs with different parameters on the dataset with footpaths.

partition size.
For the merging algorithm and PUNCH the maximum partition size is distinguished by the hyperparameter U, so it does not differ in the results of these two algorithms. Though, the merging algorithm creates less partitions than PUNCH for the same choice of U.
For METIS and k-means the number of partitions is distinguished by the hyperparameter k. METIS creates bigger partitions than k-means when k is chosen small but less partitions when U is chosen great.
For comparable numbers of partitions, k-means and METIS both create bigger partitions than the merging algorithm and PUNCH.

Figure 5.20 presents the cut sizes on the dataset with footpaths.
As expected, the cut size increases when more partitions are created or when the upper bound partition size is chosen small. Figure 5.18 and Figure 5.19 show cut-outs of the solutions of k-means with $k = 200$ and $k = 800$. The cut-outs show the city of Berlin, which is cut into many small partitions for $k = 800$, which results in a great cut size, whereas the whole city forms one partition with $k = 200$.

The merging algorithm produces the smallest cut sizes, followed by METIS.
K-means produces better results than PUNCH for small partitions. This is not the case on the dataset without footpaths, which is presented in Figure 5.21. K-means can deal better with the footpaths than PUNCH, even if it does not take

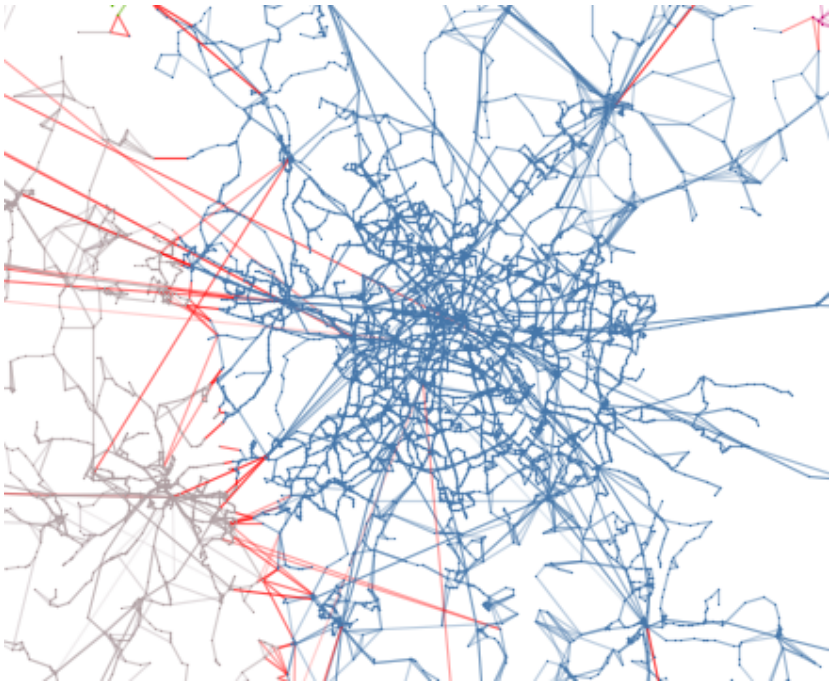

Figure 5.18: Cut-out of the solution of k-means with $k = 200$ on the dataset with footpaths.

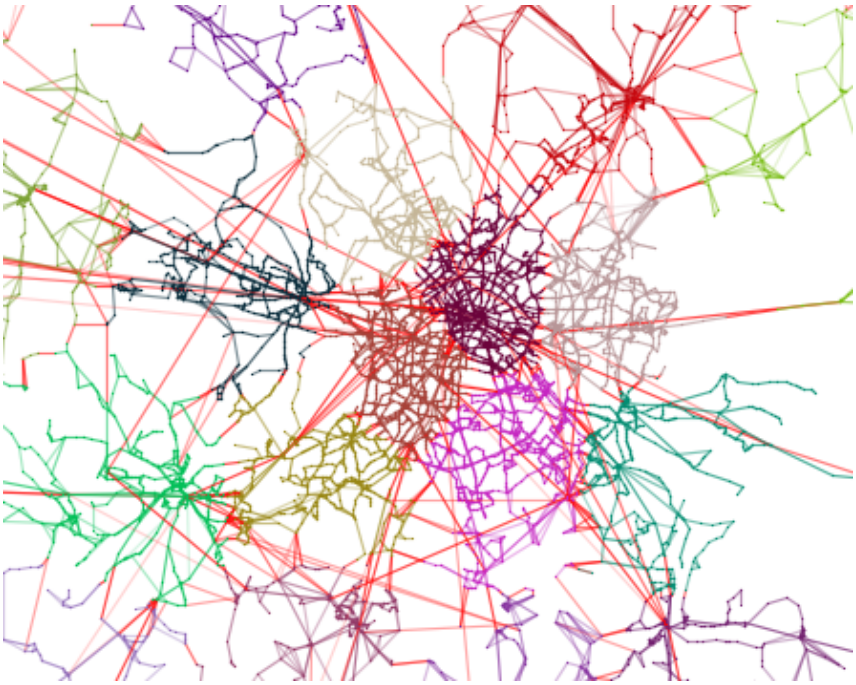

Figure 5.19: Cut-out of the solution of k-means with $k = 800$ on the dataset with footpaths.

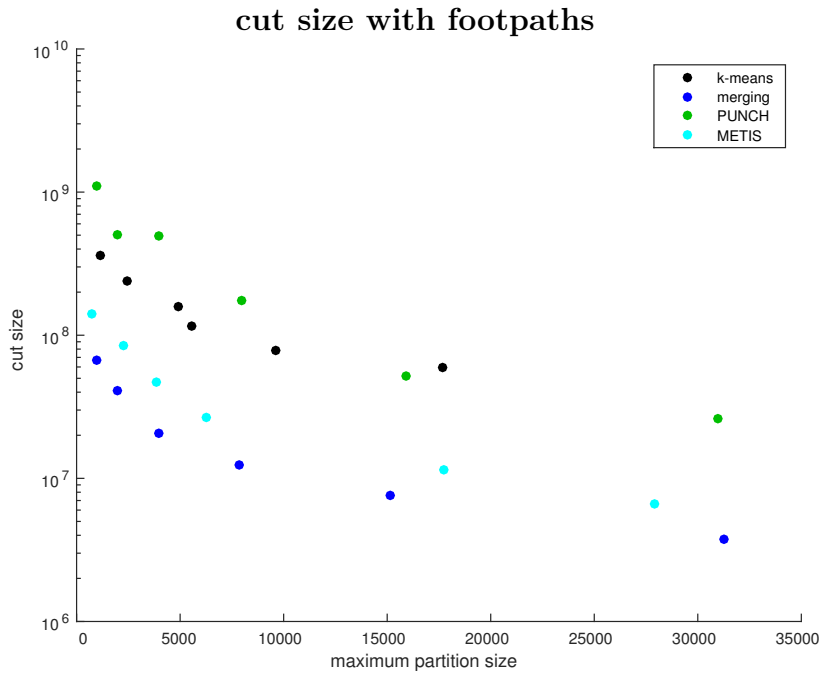

Figure 5.20: Relation between cut size and maximum partition size for the algorithms. Points represent results of runs with different parameters on the dataset with footpaths.

the edges into account. A possible explanation is that the heuristic footpaths are added with respect to spatial data. Since k-means finds spatial clusters of stations, it automatically finds clusters of footpaths. On the other hand, PUNCH does not use the edge weights in the filtering phase, too, but uses the graph structure. If many footpaths are edges of small cuts, they remain after the filtering phase, but edges with lower weights, that are not part of small cuts, are merged.
This is underlined by Figure 5.22, which shows the number of cut edges on the dataset with footpaths. Although k-means produces smaller cut sizes than PUNCH on the dataset with footpaths, it cuts more edges. One can follow that PUNCH cuts the "wrong" edges, i. e. edges with high weights.
The merging algorithm cuts the least edges, followed by METIS.

Figure 5.21 contains an outlier: Unexpectedly, PUNCH creates a very big cut size for $U = 32,000$ on the dataset without footpaths. This is not the case on the dataset with footpaths (Figure 5.20). A cut-out of the result of the initial solution is shown in Figure 5.23. PUNCH works fine in some areas, like the big partition coloured in green. In other areas, many small partitions remain. The grey partition is very big and therefore can not be merged with any of the small partitions, which are coloured in different colours. This results in many cut edges, which are coloured red.

In Figure 5.24 one can see the number of border nodes of the results of the four algorithms.

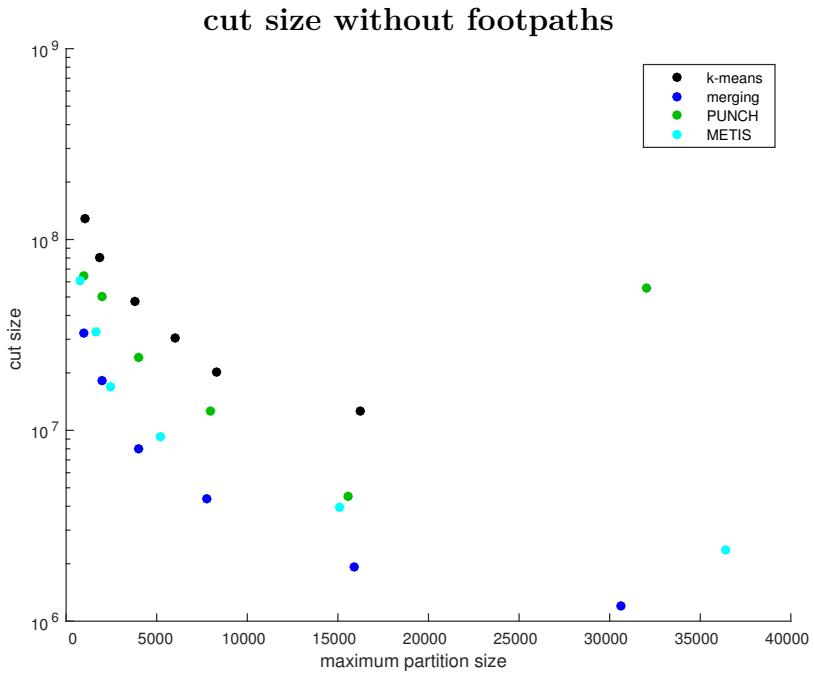

Figure 5.21: Relation between cut size and maximum partition size for the algorithms. Points represent results of runs with different parameters on the dataset without footpaths.

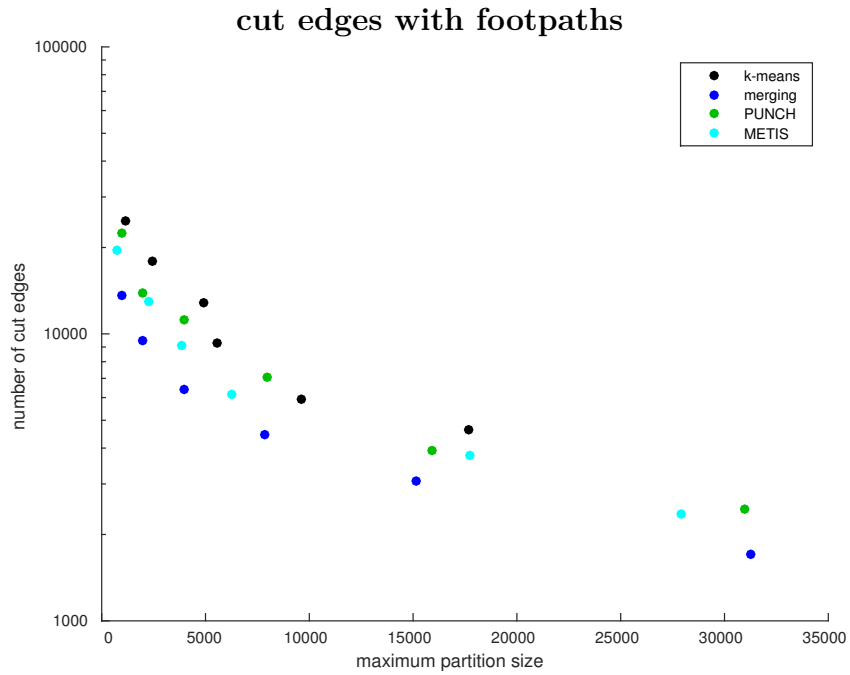

Figure 5.22: Relation between number of cut edges and maximum partition size for the algorithms. Points represent results of runs with different parameters on the dataset with footpaths.

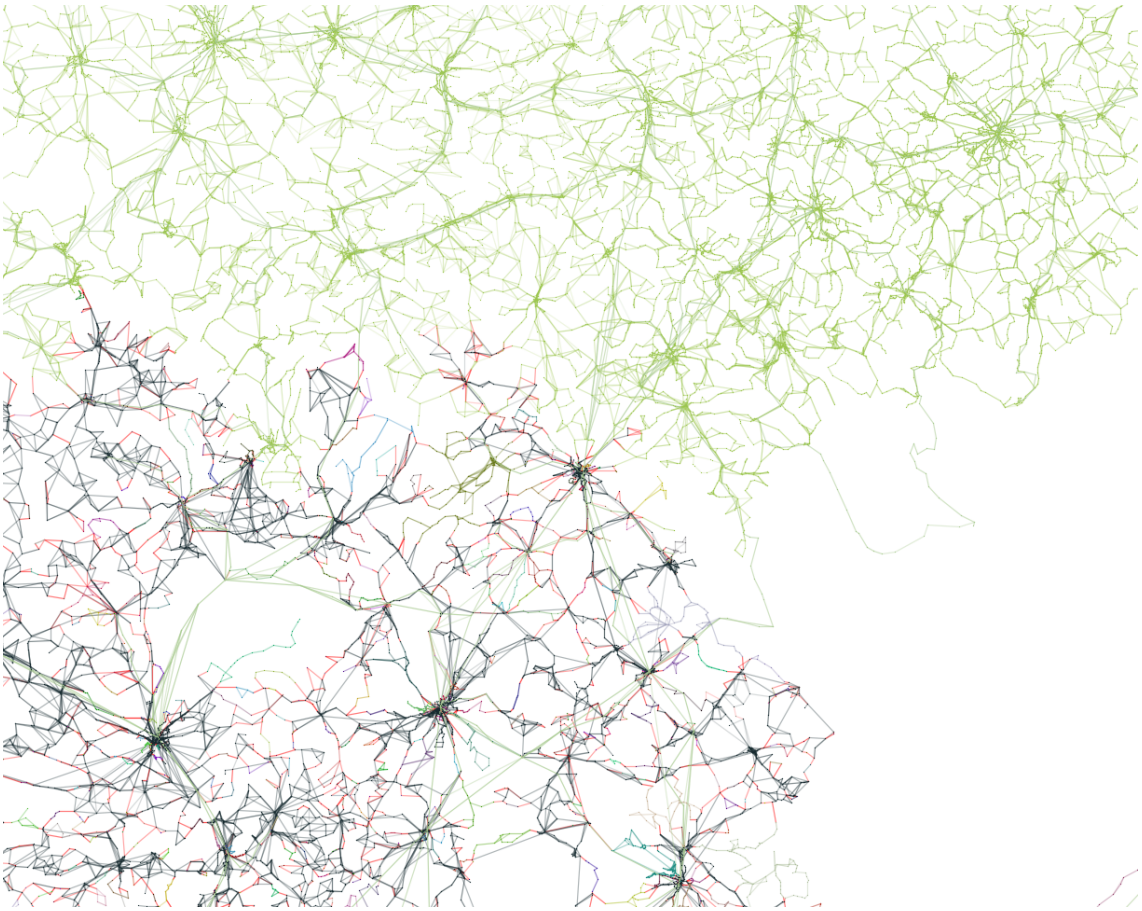

Figure 5.23: Cut-out of the initial solution of PUNCH with $U = 32,000$ on the dataset without footpaths.

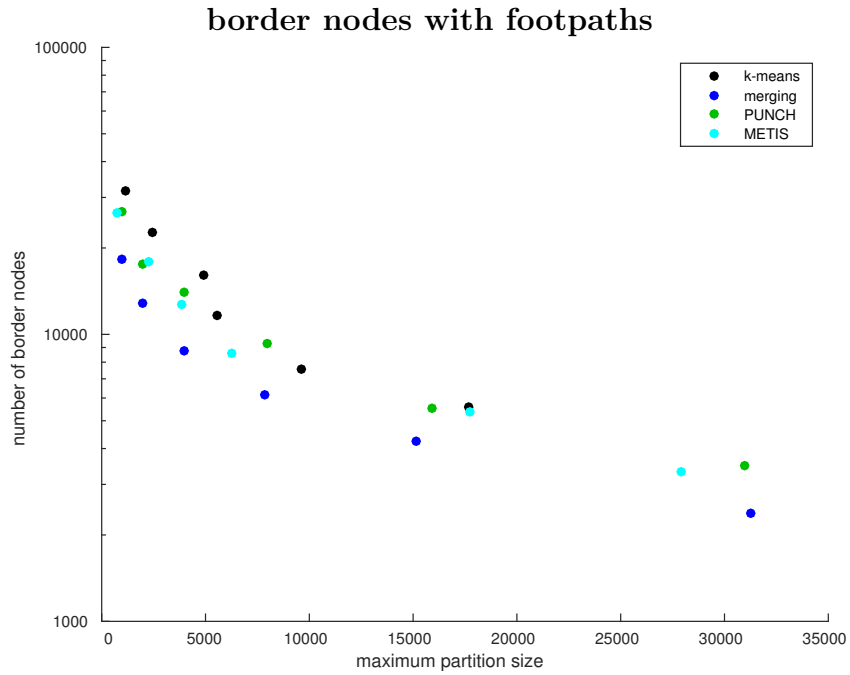

Figure 5.24: Relation between number of border nodes and maximum partition size for the algorithms. Points represent results of runs with different parameters on the dataset with footpaths.

Again, the merging algorithm produces the best results, followed by METIS and PUNCH. K-means creates the most border nodes. Though, the number of border nodes created by the four algorithms does not vary much.

The runtime of the four algorithms can be compared in Figure 5.25.
METIS is very fast and runs in less than one second. Though, the runtime of METIS increases when more partitions are created.
The runtime of the merging algorithm lies between two and three seconds. It is rather constant for the tested choices of U.
The runtime of k-means is about 5 seconds for $k = 25$ and increases with greater k's.
PUNCH has the longest runtime of the four algorithms. It runs between 5 and 42 minutes.

A comparison of the results of the four algorithms and the baseline on the dataset with footpaths is made in Table 5.4. Example runs were taken that produced nearly the same number of partitions as the baseline.
It turns out that the four algorithms outperform the baseline in every criterion.
The merging algorithm produces the smallest partitions with a maximum partition size of 1,873. It has also the best cut size, followed by METIS. METIS cuts the least edges and produces the least border nodes. Only 1.6% of the edges are cut by METIS and only 4.8% of the nodes become border nodes. The percentages of cut

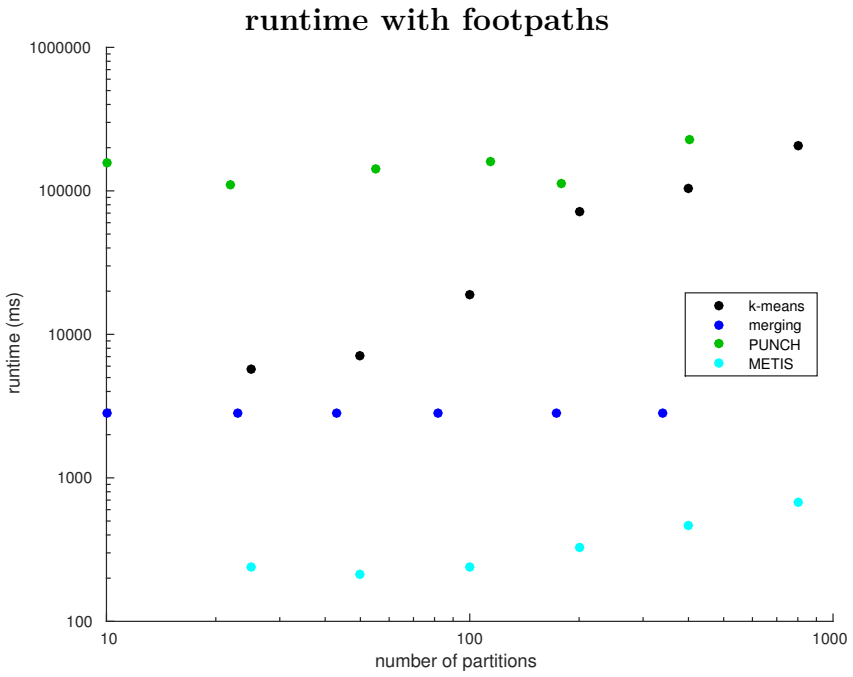

Figure 5.25: Relation between runtime and number of partitions. Points represent results of runs with different parameters on the dataset with footpaths.

edges and border nodes of the merging algorithm are close to that of METIS. METIS is the fastest algorithm and runs for less than one second. The merging algorithm runs for 3 seconds. K-means and PUNCH take longer. K-means runs for less than one minute, PUNCH runs about two minutes.

5.7.1 The gain of adding footpaths

Footpaths were added to the data model in order to include more stations to the biggest connected component of the graph and to create more realistic partitionings. Figure 5.26 and Figure 5.27 show that the latter is achieved by adding footpaths. The first figure shows a solution of the merging algorithm on the graph without footpaths. An overlaying partition is seen, that contains some stations of Berlin and some stations far away from Berlin. This overlaying partition is badly connected to the rest of the graph. Only a few cut edges are produced by this partitioning. The latter figure shows a solution of the same algorithm with the same parameters on the graph with footpaths. The overlaying partition disappeared, but many cut edges are introduced. This means that the cut size is high, but the gain is that the partitions look more realistic and match better with the expectation of “good” partitions.

	baseline	k-means	merging	PUNCH	METIS
partitions	181	181	181	176	181
max. part. size	33,302	4,015	**1,873**	1,975	3,132
cut size	$5{,}540.8{\cdot}10^6$	$154.7{\cdot}10^6$	$\mathbf{42.8{\cdot}10^6}$	$496.4{\cdot}10^6$	$45.5{\cdot}10^6$
cut edges	44,353	12,273	9,497	13,917	**8,562**
cut edges (%)	8.1	2.2	1.7	2.5	**1.6**
border nodes	44,285	15,564	12,954	17,669	**12,010**
border nodes (%)	17.7	6.2	5.2	7.1	**4.8**
runtime (s)	-	53.8	2.9	118.3	**0.3**

Table 5.4: Baseline and results of the four algorithms with about 181 partitions on the dataset with footpaths.

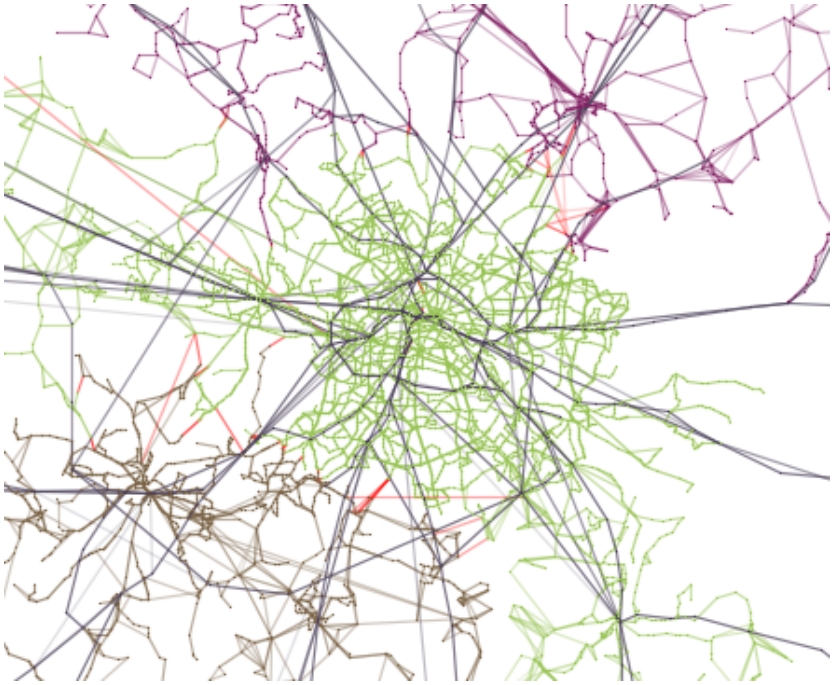

Figure 5.26: Cut-out of the solution of the merging algorithm with $U = 4,000$ on the dataset without footpaths.

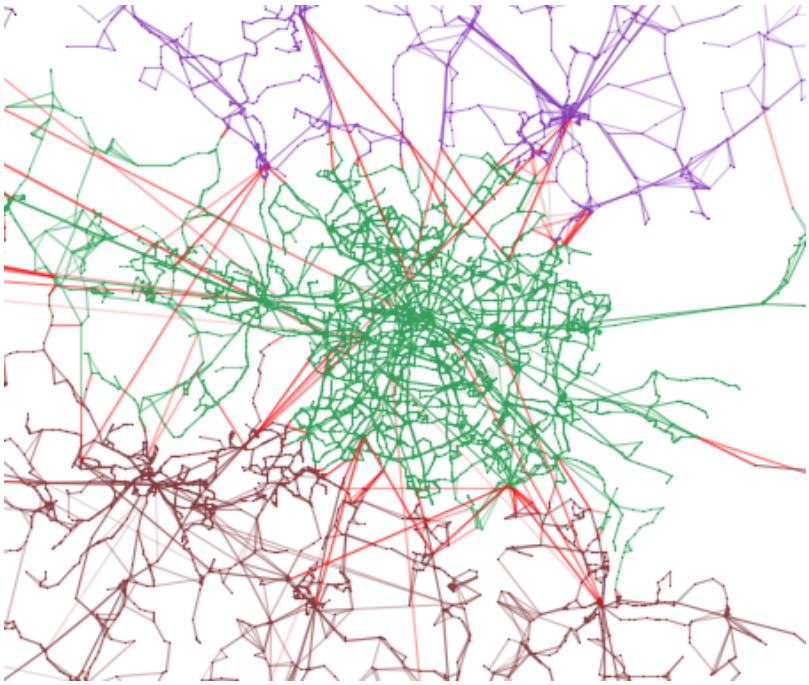

Figure 5.27: Cut-out of the solution of the merging algorithm with $U = 4,000$ on the dataset with footpaths.

6 Conclusions and future work

The k-means-clustering-algorithm performed better than expected on the problem of partitioning public transit networks. Not only the stations of a public transit network are spatially clustered, but also the traffic between the stations. The same holds for footpaths. K-means-clustering is useful for distinguishing the number of partitions that is optimal for a given public transit network.

The merging algorithm and METIS are fast algorithms and produce the best results, i. e. partitionings with the smallest cut size, the least cut edges and the least border nodes.
In METIS the trade-off between cut size and maximum partition size can be controlled with the unbalancing factor, which makes the program very useful.
Finding the best utility function for the merging algorithm is an open field. Arbitrary functions can be used. The requirements of an application that will use the partitioning can directly be modelled as a formula and used to distinguish how the partitioning shall look like. The balancing term used in the utility function $f_{improved}$ is helpful, because it fastens up the algorithm and leads to less partitions in the final result.
Especially the merging algorithm has to be tested on bigger public transit networks.

PUNCH uses a filtering phase to shrink the graph size before partitioning it. In the implementation of this work, the filtering phase took longer than the computation of the partitioning - this might not be the case on bigger public transit networks.
The heuristics used in the filtering phase lead to worse results. A filtering phase has to be invented that does not only take the graph structure into account, but also uses edge weights. PUNCH can also be equipped with a utility function that fits the requirements of a given application.
Still, the optimization phase of PUNCH improves the cut size of the partitionings and might also be used for the results of k-means, METIS or the merging algorithm.

Adding footpaths to the data model is not necessary, but was very useful in this work. Not only did the biggest connected component of the graph contain more stations after footpaths were added, but also did the high-weighted footpaths prohibit geographically overlapping partitions.

Finally, the partitionings produced by the algorithms have to be tested in public transit routing, e.g. together with Transfer Patterns ([2]) or the Connection Scan ([3]), in order to verify the benefit of partitioning of public transit networks.

Acknowledgement

I'd like to thank Prof. Dr. Hannah Bast for her inspiring lectures and for giving me the chance to write my thesis at her chair.
I want to thank Dr. Sabine Storandt for her support during all phases of my thesis.
I also want to thank my family, who always encourages me.
Last but not forgotten, I want to thank you, the reader, for your interest in my thesis.

Bibliography

[1] D. Delling, A. V. Goldberg, I. Razenshteyn, and R. F. Werneck, "Graph partitioning with natural cuts," in *Parallel & Distributed Processing Symposium (IPDPS), 2011 IEEE International*. IEEE, 2011, pp. 1135–1146.

[2] H. Bast, E. Carlsson, A. Eigenwillig, R. Geisberger, C. Harrelson, V. Raychev, and F. Viger, "Fast routing in very large public transportation networks using transfer patterns," in *Algorithms–ESA 2010*. Springer, 2010, pp. 290–301.

[3] B. Strasser and D. Wagner, "Connection scan accelerated," in *Proceedings of the Meeting on Algorithm Engineering & Expermiments*. Society for Industrial and Applied Mathematics, 2014, pp. 125–137.

[4] R. E. Tarjan, "A note on finding the bridges of a graph," *Information Processing Letters*, vol. 2, no. 6, pp. 160–161, 1974.

[5] D. Pritchard and R. Thurimella, "Fast computation of small cuts via cycle space sampling," *ACM Transactions on Algorithms (TALG)*, vol. 7, no. 4, p. 46, 2011.

[6] A. V. Goldberg and R. E. Tarjan, "A new approach to the maximum-flow problem," *Journal of the ACM (JACM)*, vol. 35, no. 4, pp. 921–940, 1988.

[7] D. D. Sleator and R. E. Tarjan, "A data structure for dynamic trees," in *Proceedings of the thirteenth annual ACM symposium on Theory of computing*. ACM, 1981, pp. 114–122.

[8] "Hacon fahrplan-auskunfts-system," http://www.hacon.de, accessed: 2015-08-18.

[9] J. MacQueen, "Some methods for classification and analysis of multivariate observations," in *Proceedings of the fifth Berkeley symposium on mathematical statistics and probability*, vol. 1, no. 14. Oakland, CA, USA., 1967, pp. 281–297.

[10] M. G. van der Horst, "Optimal route planning for car navigation systems," Master's thesis, Technische Universität Eindhoven, 2003.

[11] G. Karypis and V. Kumar, "A fast and high quality multilevel scheme for partitioning irregular graphs," *SIAM Journal on scientific Computing*, vol. 20, no. 1, pp. 359–392, 1998.

[12] G. Karypis, "Metis - serial graph partitioning and fill-reducing matrix ordering," http://glaros.dtc.umn.edu/gkhome/metis/metis/overview, accessed: 2015-08-01.

[13] ——, “Metis - a software package for partitioning unstructured graphs, partitioning meshes, and computing fill-reducing orderings of sparse matrices - version 5.1.0,” http://glaros.dtc.umn.edu/gkhome/fetch/sw/metis/manual.pdf, 2013, accessed: 2015-08-01.

[14] B. W. Kernighan and S. Lin, “An efficient heuristic procedure for partitioning graphs,” *Bell system technical journal*, vol. 49, no. 2, pp. 291–307, 1970.