Bachelor's Thesis

Generating Low-Voltage Grid Benchmarks with OpenStreetMap

Metty Kapgen

Examiner: Prof. Dr. Hannah Bast
Advisers: M.Sc. Matthias Hertel

University of Freiburg
Faculty of Engineering
Department of Computer Science
Chair of Algorithms and Data Structures

October 13th, 2022

Writing Period

13. 07. 2022 – 13. 10. 2022

Examiner

Prof. Dr. Hannah Bast

Advisers

M.Sc. Matthias Hertel

Declaration

I hereby declare that I am the sole author and composer of my thesis and that no other sources or learning aids, other than those listed, have been used. Furthermore, I declare that I have acknowledged the work of others by providing detailed references of said work.
I hereby also declare that my thesis has not been prepared for another examination or assignment, either wholly or excerpts thereof.

Freiburg, 13.10.2022

Place, Date	Signature

Abstract

This bachelor's thesis documents an implementation of a benchmark generation tool to evaluate low-voltage grid expansion algorithms. This is achieved through generating test grids with the help of OpenStreetMap. In a time in which the supply and expansion of energy infrastructure becomes more and more pressing it is especially important to develop efficient and correct expansion plan algorithms.

The emphasis of this thesis lies in the ability to customize such grids to fit the user's specialized needs. We provide the user of this benchmark tool, among other things, with a set of forest algorithms which can be selected to generate different states/scenarios on the same grid. Those grids can then be used to test, compare and verify different expansion plan algorithms on real-world scenarios. This implementation will return a dataset which can be digested by expansion planning algorithms that use PyPSA[1].

Zusammenfassung

Diese Bachelorarbeit dokumentiert die Implementierung eines Benchmark-Tools zur Erstellung und Evaluierung von Niederspannungsnetzausbau-Algorithmen. Dies wird durch die Generierung von Testnetzen mit Hilfe von OpenStreetMap erreicht. In einer Zeit, in der die Versorgung und der Ausbau der Energieinfrastruktur immer dringender wird, ist es besonders wichtig, effiziente und korrekte Ausbauplanungsalgorithmen zu entwickeln.

Der Schwerpunkt dieser Arbeit liegt darin, solche Netze an die speziellen Bedürfnisse des Nutzers anzupassen. Wir stellen dem Benutzer dabei unter anderem eine Reihe von Waldalgorithmen zur Verfügung, die ausgewählt werden können, um verschiedene Zustände/Szenarien auf dem gleichen Netz zu erzeugen. Diese Netze können dann verwendet werden, um verschiedene Ausbauplanungsalgorithmen auf realen Szenarien zu testen, zu vergleichen und zu verifizieren. Diese Implementierung wird einen Datensatz ausgeben, der von Ausbauplanungsalgorithmen, die PyPSA[1] verwenden, verarbeitet werden kann.

Contents

List of Figures

List of Tables

List of Algorithms

1 Introduction

Most European villages and cities have a low-voltage grid to feed residential and commercial buildings with electricity. Due to the rising demand of energy and the expansion of villages, grid operators have to regularly expand and improve their coverage. This development of expansion plans is a critical and difficult task, as many electrical grid norms must be respected while also finding the cheapest extensions possible. Usually this job is done by experts who manually develop such plans using their knowledge and experience.

However, in the past years more and more algorithms[2][3] are being designed in order to solve this same problem. Such algorithms can largely differ in time complexity, efficiency, price and quality of their proposed solution. Those different factors make it hard for operators to identify suited algorithms. Thus raising the essential question on how to compare and benchmark these different algorithms. The most intuitive approach would be to request grid data from different grid operators and manually generate a benchmark. This sensitive data, however, is usually not publicly available and thus difficult to get.

Therefore, we have to fall back on an open and publicly available source. The OpenStreetMap dataset enables us to extract not only roads and transmission lines, but also substations which usually feed villages by converting higher-voltage into low-voltage current and buildings which act as the consumers in the grid. With the OpenStreetMap data, some electrical configuration values and a specialized forest algorithm we can simulate a low-voltage grid of any given village. Our code will

return a set of CSV files which is directly compatible with the Python for Power System Analysis (PyPSA) library[1]. At last, a set of such village grids can be used to benchmark and compare different expansion plan algorithms against each other.

2 Related Work

OpenStreetMap data in modelling the urban energy infrastructure: a first assessment and analysis[4]

This paper states, that the OpenStreetMap data can be used to develop electrical grids. Additionally, it states that even though some OpenStreetMap data features are missing, it can still be useful for sustainable and transparent grid modelling.

OSMnx: New methods for acquiring, constructing, analyzing, and visualizing complex street networks[5]

OSMnx is an extremely powerful python library capable of turning OpenStreetMap data into NetworkX[6] objects. Its main goal lies in providing an easy to use, automated and reliable library for the extraction and processing of street networks.

Open Data in Power Grid Modelling: New Approaches Towards Transparent Grid Models[7]

In this paper, the authors developed two different methods of modelling power grids. One of them makes use of the powerlines in OpenStreetMap. However, low-voltage cables are mainly placed below the ground, which are not stored in the OSM dataset. In contrast to this paper, we will additionally make use of other village components (roads, buildings, etc.) to generate our grids.

Expansion Planning of Low-Voltage Grids Using Ant Colony Optimization[2]

This master thesis provides the reader with a low-voltage grid expansion planning algorithm called *ANTPOWER*. This algorithm uses the so-called ant colony optimization process to find cheap solutions for a grid expansion.

Testing and improving Antpower on the Simbench networks[8]

This blog post tests the above-mentioned *ANTPOWER* code on a synthetic set of low-voltage grids, called the SimBench networks[9]. The article states that it would like to have the ability to test the algorithm on a larger set of grids with more than one substation. Moreover, it would also like the grid to have new lines as candidates which are currently not installed on the grid for further potential expansions.

Generating low-voltage grid proxies in order to estimate grid capacity for residential end-use technologies: The case of residential solar PV[10]

This paper gives us an interesting approach on how to estimate the low-voltage grid capacity from photovoltaic generators. In future works it may be interesting combining such methods with those of this thesis, to get an even more accurate simulation/benchmark grid, by properly estimating generator and even grid values.

3 Background

3.1 Low-Voltage Grid Expansion

A low-voltage grid can be interpreted as a graph consisting of nodes and edges with attributes. Edges represent all available lines, meaning the lines that are currently part of the low-voltage grid and candidate lines that may be used for a potential expansion. Every line is described by a cable type which can have different electrical properties. Not every line must be part of the currently installed grid, as topological constraints restrict certain lines from being installed.

The topological constraints are:

1. A grid can not have circles
2. A grid can not have a path between transformers

In the following, a "ci-grid" will stand for the currently installed grid. The weight of an edge represents the physical distance of its two connected nodes. A node represents a so-called "bus" and can have different characteristics. It can be a generator, transformer, load/consumer or a simple intersection between lines. Sometimes a bus can be a consumer and generator at the same time. In the real world this could, for example, be interpreted as a residential building that has a photovoltaic system. Consumers and generators are connected to the grid, by a so-called "private line". Such a "private line" is ignored in the process of expansion planning, as it represents the line that is setup on the private property of its consumer. Thus, it can and

should not be upgraded by the grid operators. The lines that are part of the ci-grid can be upgraded by expansion planning and are called ”public lines”.

The main task of a low-voltage grid is to connect consumer and generator buses to transformers, without violating the above-mentioned electrical constraints. A village can have several transformers and generators to guarantee a sufficient supply to a grid. A correct ci-grid should additionally sustain load simulations through a grid-library like PyPSA's[1] *powerflow*. It is, however, up to the benchmark's user to determine a threshold when a line or node violates the constraints of the grid. The return values of PyPSA's *powerflow* should be used to determine which components violate the constraints of the grid. Moreover, PyPSA's *powerflow* can be run on several snapshots. A snapshot is a point of time for the feed and load values of every building. Snapshots usually represent the extreme cases on which the grid should be tested.

The expansion planning process of these grids can be defined as the process of improving the grid stability and capability by replacing and/or upgrading the currently used lines, while also minimizing its overall cost. In short, an expansion planning algorithm tries to find the cheapest expansion that does not provoke any grid violations. If a ci-grid does not contain any violations, it generally does not need to be upgraded.

Throughout this paper we will use the term buildings for both consumer and generator nodes. Additionally, substations will be a synonym for transformers.

3.2 OpenStreetMap Data

OpenStreetMap (OSM) is a large and open dataset containing streets, buildings, locations, and territories of the entire planet. The raw data can be processed with the help of many libraries and is freely available on the web. The OSM data consists of three main types.

A node is the simplest form of data, as it can be interpreted as a simple point with several attributes. It contains a node ID, coordinates and several other tags. For example node 5212856530 has coordinates $47.9953250, 7.8575591$ (Freiburg i. Br., Germany) and has a tag ”power = transformer”.

Another important OSM element is the way datatype. It consists of a list of nodes and can also contain informational tags. It can be used among other things to define shapes like buildings and roads. Way 143746749, for example, represents the ”Umspannwerk Freiburg - Schlossbergring” which is a building that also contains the ”power = substation” tag.

The last important OSM element is the relation tag, this tag is used to unify different nodes into groups. This datatype, however, is not of great importance for this thesis.

4 Approach

4.1 Problem Definition

Expansion planning is algorithmically a very challenging task. Thus deciding on which solvers to pick for different scenarios is of crucial importance. If you want to upgrade a heavily violated grid, some algorithms may perform much better than others. The same is also true for a scenario with much fewer violations. This raises the essential question, on how to test and verify the capabilities of those algorithms.

Our OSM low-voltage grid benchmark tries to solve this exact problem. This tool enables developers to test grid expansion planning algorithms on real world scenarios. The benchmark aims to provide as much flexibility as possible by splitting up the problem generation process into four sequential steps, which are explained next.

Step 1: User picks a city or village

Step 2: User defines electrical grid configuration

Step 3: User defines physical grid configuration

Step 4: User picks a sb-forest algorithm

In the first step, the user is asked to select the city or village the grid should be originated from. This input must be a municipality in OSM as it is specifically used by OSMnx[5]. If this condition is not met, the benchmark may have trouble to find the proper location. After that, the user should define a set of electrical constants

that are necessary to properly configure the grid. Those variables are especially important for the PyPSA's[1] built-in *powerflow* function, which is indirectly used by the expansion planning algorithm to determine whether a certain grid is overloaded or not.

Next, we need the user to define several boundaries for the physical properties of grid. The benchmark wants to know, how many substations the grid should have at least, how many consumers it should have at most, and what the ratio between generators and consumers should be.

Lastly the benchmark will request the user to select between a set of modified forest algorithms, which will individually produce a different ci-grid. This selection enables the user to generate different expansion scenarios on the same grid. They can choose between a shortest paths, a random, a minimum and a maximum variant.

The code used to implement the benchmark is written in Python. This was done for two main reasons. On one hand, it is a widely used language thus already containing some OSM related libraries helping us with the extraction of data. On the other hand, we want expansion planning algorithms to fit PyPSA's[1] built-in methods, which are also based on Python.

4.2 OSM Data Extraction

In this part we are going to present how we extracted the OSM data that was needed to generate the grids from the user's input of a village or city name. To start off we need to mention that we used two main libraries, which both played a significant role in the development of the code. Firstly we used the OSMnx[5] library, which is able to generate a street grid in the NetworkX[6] format. A large percentage of low-voltage grid cables in villages and cities are placed below public roads, therefore this NetworkX street graph can be used as a foundation for the simulated electrical grid. This graph can be generated with the help of the OSMnx *graph_from_place*

method, with the network type set to ”drive”. This method will guarantee that the generated graph will always be connected, as the ”retain_all” variable is set to false by default.

This simple graph, however, does not take into account the overhead powerlines. Note that they should not be confused with high-voltage lines. Generally the OSM powerlines are an electrical equivalent to street cables. To get this data we can once again make use of OSMnx by generating another graph with the custom filter ”[”power” = ”line”]”. Uniting both graphs will give us a good estimate where low-voltage cables should be situated in the real world.

To connect both graphs to each other we firstly identified the endpoint nodes of the overhead powerline graph, by checking for the nodes of degree 1. After that we simply connected those nodes to their closest neighbor node in the street graph. The result will be used as the foundation for our grid.

In **Figure 1** we can see the unification of the both graphs on the village ”Bad Krozingen” near Freiburg im Breisgau. Black lines represent cables that are placed below the streets, while the green lines represent the ones above the ground. The orange lines represent the connecting cables between both graphs.

To turn our merged graphs into a grid, the next step is to add substations and buildings that will later act as consumers or generators or even both. OSMnx, however, is not properly suited to help us with this problem. Thus, we need a second more powerful OSM library, which is called PyOsmium[11]. PyOsmium is a Python derivative of the Osmium Library which runs in C++. This library enables us to read from an OSM file by directly filtering and returning specific OSM objects. Thus, we can look up specific infrastructure such as residential and commercial buildings and substations. Unfortunately the OSM dataset is not always properly standardized, resulting in several infrastructures being defined as ways instead of nodes. This forced us to take both into consideration. As we can not directly add ways to our simulated grid, we must first reduce said ways to their first node before adding them.

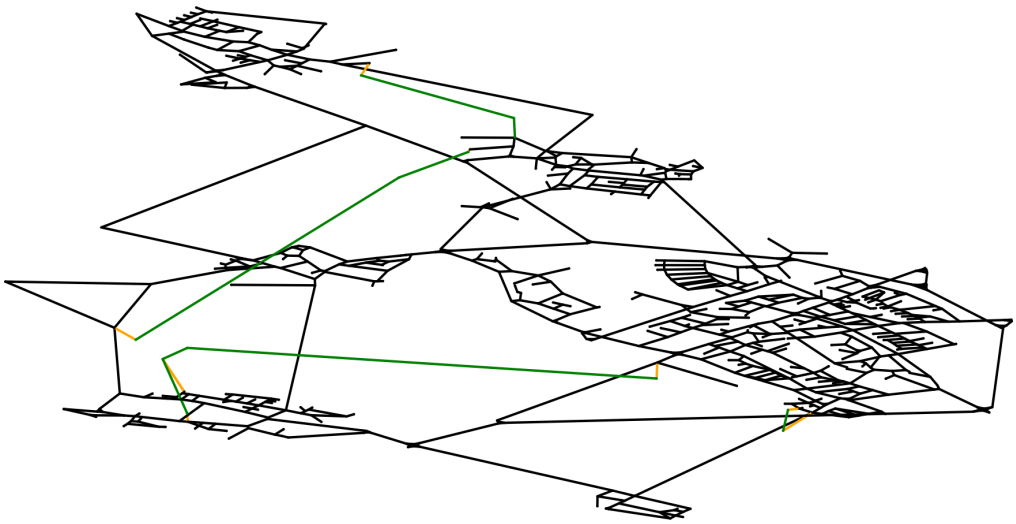

Figure 1: Merging the street graph and the powerline graph of Bad Krozingen

Having identified all the interesting nodes for this grid, the next step will be to connect them to the grid. Connecting a building to the grid will be done with the help of the private lines. They are usually connected to the grid in the shortest way possible as private entities aim to minimize their cable cost when initially linking up to the grid. Thus finding the closest, right-angled connection between a grid line and the individual buildings seems like the most reasonable approach. This is done by iterating over all the existing edges and computing the closest point on the nearest edge to the building's node. When we have identified that point we will split the belonging edge with the help of said point and connect it to the building node by a private line. The substations, on the other hand, can be connected to the grid in a much simpler way. Firstly we need to determine which substations could really take part in our low-voltage grid. We start of by calculating the distance between every OSM substation and the center of the village we want to simulate. If that distance falls within a certain threshold, we will connect it to the closest node of the grid. If none of the OSM substations is close enough to the village to fall within that threshold, we will choose the closest as the only substation of the grid.

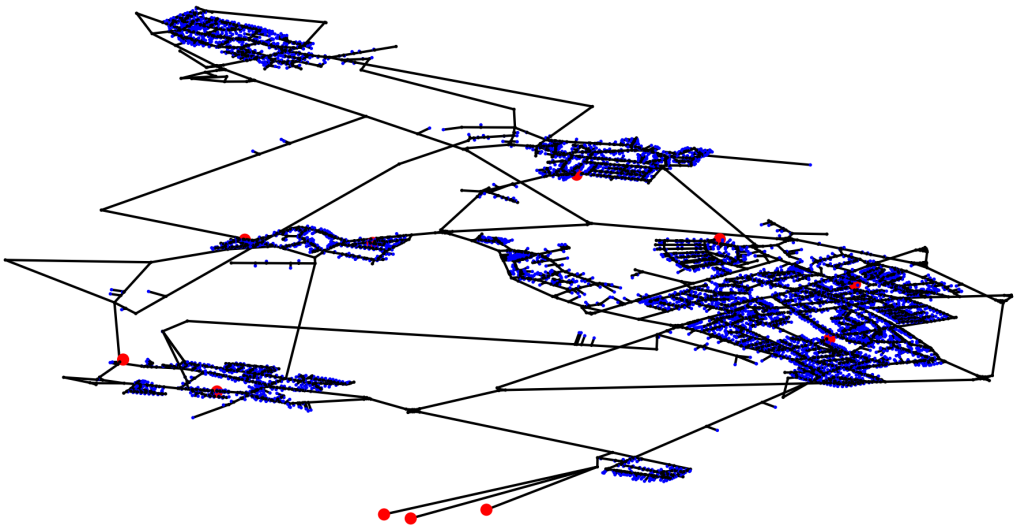

Figure 2: Adding consumers and substations to the grid of Bad Krozingen

In **Figure 2** we can see the buildings that were connected to the grid in blue, while the substations are colored in red.

Lastly, we will determine the weights of the edges by applying basic Pythagoras on the coordinates of the connected nodes, by calculating $weight = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$, where the two nodes have coordinates (x_1, y_1) and (x_2, y_2) respectively.

In general, the data extraction part of the benchmark is a time-consuming task. The extraction of Bad Krozingen for example can take up to ten minutes on an i7-7700HQ (mobile) processor with the OSM file being stored on a SSD. OSMnx will directly request the necessary data from an API. PyOsmium on the other hand will have to receive an OSM file to properly work. This library will iterate through every node and way of the OSM file resulting in large runtimes, depending on the size and spectrum of the files. Fortunately, there are many shrunk down versions for different districts. In this thesis we used the "Regierungsbezirk Freiburg" file.

Usually the extraction process must only be done once to retrieve the general grid for each village, thus it is useful to compress and separately store the grid with the

gpickle methods, which is part of the NetworkX library. In the next steps, such gpickle files will be unpacked for further processing.

4.3 Grid Configuration

In the second step of using the benchmark tool we will ask the user to configure several electrical variables. Those variables can be set through a configuration file. This configuration was derived from the inputs PyPSA's *powerflow* will need. Through this file we will be able to define a large quantity of component properties. On startup this file is filled with a set of default values derived from *ANTPOWER*'s test input. We introduce two different PyPSA snapshots, which represent different extreme scenarios in the grid. Either the grid can have an abundance of power with little consumer need(*feedcase*), or a lack of power with a large request for energy(*loadcase*).

Apart from the electrical configuration the user will also have to determine several limiting bounds for the amount of substations, buildings and generators.

Sometimes the OSM data can be incomplete and does not provide a realistic scenario for the amount and positioning of substations. Thus, the benchmark generation tool provides the user with the capability to determine a lower bound for the amount of substations the grid must have. This functionality was implemented by counting the actual amount of substations currently in the grid, and converting random buildings to substations if the requested quantity is not met.

Secondly, the user can determine an upper bound for the amount of buildings. This method was mainly added to give the user the freedom to reduce the complexity of the generated grid. The function was implemented by randomly deleting a set amount of building nodes and their private lines from the grid.

Lastly, the user is able to select a value for the ratio between consumers and generators. Every building in the grid will be a consumer, however the probability of it also being a generator is determined through that variable.

4.4 CI-Grid Generation

Now that we have the complete grid and its configuration data, we need to determine which lines are currently part of the actual ci-grid and which lines are available as upgrade options. The ci-grid can be seen as a specialized forest, that respects the above-mentioned topological constraints.

Note that we are not directly looking for **spanning forests** as results from our algorithms. Generating a spanning forest on the grid, would return us a result, that will contain every node. However, we are actually looking for a forest, that connects **every** building to **exactly** one substation. In this case, we will usually still end up with a set of edges and nodes that are not used in one of the trees. Such components will later be interpreted as the candidate lines for a potential upgrade of the grid. Thus, the following algorithms should **not** be interpreted as a spanning forest, but as a forest that guarantees that every building is connected to exactly one substation. They will have the **"SB"** (Substation-Building) prefix to clarify that they are in fact specialized forests that respect the beforehand mentioned condition.

To offer the user the most flexible grid simulation, we provided four different algorithms that can generate a ci-grid. Those different algorithmic options are especially important, as in a real world scenario, a ci-grid is only rarely a proper minimum spanning forest. A low-voltage grid is usually expanded through the historic development of the village or city. In general, this makes ci-grids prone to imperfections and inefficient layouts. Thus, to enhance the variety and also provide more challenging ci-grids, it is useful to offer the user a set of different algorithmic options to generate the sb-forest.

4.4.1 Shortest Paths SB-Forest

Algorithm

A first and intuitive way to generate a sb-forest is to connect every building to the closest substation. For that we make use of the shortest paths through the edges' weights, i.e. their physical length. Firstly we need to identify such paths from each building node to its closest substation. NetworkX, the library we are using to represent our grid in, already has a useful built-in method for this. The method is called *multi_source_dijkstra_path* and is an extension of Dijkstra's famous shortest path algorithm. Instead of the usual *dijkstra_path* which needs one specific source, the extended version can handle a set of sources.

In general, an implementation of Dijkstra's algorithm will sequentially iterate over the neighbor of the set of already visited nodes with the shortest distance to the source node. Thus, it keeps track and possibly updates the currently shortest path found for each node. It needs a queue to store the order in which different nodes should be selected. Note that with every visited node, the order of neighbor nodes in the queue can potentially change. Once a node is visited, it can no longer enter the queue again, as its shortest path was already found, assuming there are no negative edge weights in the grid. The only input Dijkstra's algorithm will need is a source node, from which we can sequentially enlarge the set of neighbor nodes and update the shortest paths.

The multi-source version of this algorithm shares large parts of the same process. Instead of starting the algorithm with only one source node, that has a shortest path of weight zero to itself, we will have several source nodes having this property. A node position in the queue will be determined by the shortest distance to one of the source nodes. The pseudocode for the *multi_source_dijkstra* algorithm can be seen in **Algorithm 1**.

Note that every node of the grid will be considered in a shortest path algorithm as long as the grid is connected, thus every node must also find a path to the sources. In our implementation, the sources will be the substations. This will return us a list of edges for every building representing the path to its closest substation. The paths that are generated for all non-building nodes during the method call can be discarded.

Algorithm 1 Shortest path sb-forest algorithm(grid=(V,E))

foreach *node* in grid **do**
  $dist[node] = \infty$
  $prev[node] = None$
end for
foreach *source* in *sources* **do**
  $dist[source] = 0$
end for
heapq = set of all nodes in grid
while *heapq* not empty **do**
  candidate = smallest *dist* node in *heqpq*
  $heapq = heapq \setminus \{candidate\}$
  foreach *neighbor* of *candidate* still in *heapq* **do**
    if $dist[candidate] + weight(candidate, neighbor) < dist[neighbor]$ **then**
      $dist[neighbor] = dist[candidate] + edge\_weight(candidate, neighbor)$
      $prev[neighbor] = candidate$
    end if
  end for
end while

Correctness

Dijkstra's algorithm will guarantee that every building will connect to at least one substation as the grid is connected. Note that it will not be possible for two nodes, that are directed to different substations, to cross or share part of their shortest paths. Otherwise, this would mean that there exists a node from which point onward there are two shortest paths to two different substations. For such a scenario to occur, both paths from the above-mentioned node onwards must have the same exact weight. However, NetworkX's method will, in this case, still deterministically chose

one of the paths. Therefore, we can conclude that every building is connected to exactly one substation, confirming the sb-forest condition.

Complexity

The complexity of this overall method is determined by the amount of nodes $|V|$ and the amount of edges $|E|$ in the grid. NetworkX *multi_source_dijkstra_path* uses Python's internal *heapq* data structure to efficiently determine the next closest nodes. *Multi_source_dijkstra_path* should not add a relevant time constraints to the general Dijkstra's algorithm, as the modifications can be done in constant time. Assuming Dijkstra's algorithm was implemented by NetworkX along the state of the art, we will receive a runtime complexity of $\mathcal{O}(|V|\log(|V|) + |E|)$. The latter runtime includes *heapq*, for which the worst case boundary for every operation can be set to $\mathcal{O}(\log|V|)$. As the *multi_source_dijkstra_path* will return us a set of paths for each node, we will iterate through every of those paths to mark down which edges and nodes will become part of the ci-grid. This can be done in $\mathcal{O}(|V| \cdot |E|)$. Thus, the overall complexity of the shortest path sb-forest algorithm will be $\mathcal{O}(|V|\log(|V|)+|E|+|V|\cdot|E|) = \mathcal{O}(|V|^2)$. The latter equation can be explained through the fact, that for every node we will have a constant amount of edges. Note that a usual street node will at most have a hand-full/constant amount of edges in case of a large crossroad. From this we can conclude that $\mathcal{O}(|V|) = \mathcal{O}(|E|)$.

On average the complexity of the iteration through the output paths of NetworkX's *multi_source_dijkstra_path* can be simplified to $\mathcal{O}(h \cdot |V|)$, where h stands for the longest (amount of edges) "shortest" path found by the algorithm. Thus, the average complexity can be reduced to $\mathcal{O}(|V|\log(|V|) + h \cdot |V|)$.

Why is there no Longest Paths SB-Forest in the benchmark?
An algorithm that would produce us the longest paths sb-forest would rely on an algorithm able to produce maximum paths. This problem, however, is NP-hard[12].

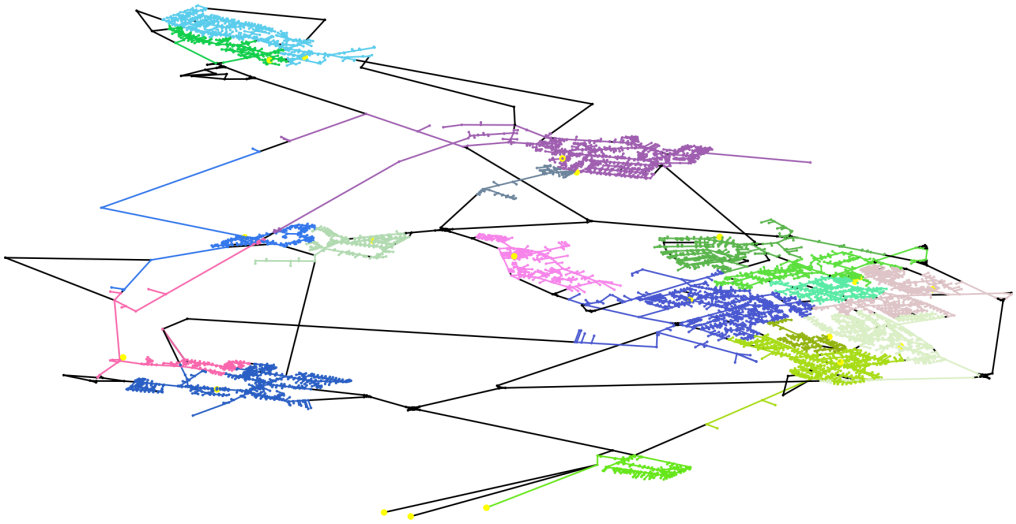

Figure 3: Shortest Paths SB-Forest of Bad Krozingen with 20 substations

Such an implementation would resemble a ”brute force” algorithm, which would have a factorial runtime complexity. As even the smallest grids do contain hundreds of edges, such an approach would struggle to finish. Thus, we declared such an implementation to be useless for our benchmark tool.

In **Figure 3** we can see the shortest paths sb-forest algorithm applied to ”Bad Krozingen”. The substations amount was set to 20 and the upper bound for buildings was set to infinity. The substations themselves are colored yellow, while their connected nodes take on a random but uniform color. Nodes and edges that stay black are not part of the sb-forest/ci-grid.

Algorithm 1 shows the pseudocode for the shortest path sb-forest algorithm.

4.4.2 Random SB-Forest

Algorithm

An alternative to generating the sb-forest through a modified shortest path algorithm, is to generate it randomly. While developing this algorithm we wanted to set a main emphasis on having approximately even sizes of the individual trees. To achieve this we let every substation sequentially pick one of its neighbor nodes to become part of its tree until every substation tree finds no more node that can be connected. This approach will return us a spanning forest that will match **every** node to one of the substations. Such a forest however does also contain nodes and lines that are not part of the ci-grid, as they are not directly on the path between a building and its substation. This can occur as we can have several nodes in the grid that are not needed to connect a building to the grid. To solve this issue, we developed a separate method called the *traceback_forest* method. This method will check for every node in the entire grid if it is an endpoint (degree = 1) of a spanning tree and neither a house nor a substation. If such a node is found, it will be removed from the forest. This method will be run as long as nodes are getting removed. In the end we will thus receive a proper sb-forest, that may also have several candidate lines. **Algorithm 2** shows the pseudocode for the random forest while **Algorithm 3** presents the *traceback_forest* method.

Correctness

As we know that the grid is connected and the individual substation trees must be as expanded as possible, we can assume that we generated a spanning forest. Note, that there is no guarantee that every substation tree will have the same amount of nodes in its set, as it can happen that certain selections can ”cut off” substations from the rest of the grid. This will stop a substation tree from further expanding, as it will not find any more available neighbors that are not already part of another tree.

Algorithm 2 Random spanning forest algorithm(grid=(V,E))

```
foreach substation in grid do
    subset[substation] = {}
end for
visited = {substations}
while visited ≠ V do
    sub = Get next substation
    candidate = Get random neighbor of subset[sub] not in visited
    if candidate exists then
        Add candidate to subset[sub]
        Add candidate to visited
    end if
end while
```

Algorithm 3 Traceback forest algorithm(grid=(V,E))

```
while nodes are getting removed do
    foreach node n in grid do
        if degree(node) = 1 & node ≠ substation & node ≠ building then
            Remove node from forest
        end if
    end for
end while
```

Using the *traceback_forest* method will sequentially shorten and remove paths that are not relevant for the sb-forest condition. As the *traceback_forest* will iteratively delete only nodes that do have degree 1 that are not buildings and substations, we can guarantee, that no node which is part of the sb-forest will be deleted.

Complexity

The runtime complexity of this algorithm can be split up into two parts. In the first step (Line 1 - 4) some initializations are executed which can be done in constant time. After that we enter the while loop (Line 5) which will run for $|E|$ iterations. Inside the loop the most time draining operation is the *get_random_neighbors* method. As we did not use advanced data structures(i.e. Union-Find) to implement this method, this method will only run in $\mathcal{O}(|V|)$. The beforehand complexity can be

explained through the lookup of the neighbor, which can take up to $\mathcal{O}(|V|)$ for each method call, as we need to initially update said set for the recently added node. The selection of the random neighbor out of this set, however, can be done in $\mathcal{O}(1)$. Secondly, in the worst case scenario of the *traceback_forest*, we would have a grid only consisting of one path, that is on one side connected to a substation while the other nodes are all simple non-building nodes. If our iterations through the $|V|$ nodes are unlucky we will only reduce such a path by one node in each step. Thus forcing us to repeat that step $|V|$ times, returning us an overall complexity for the *traceback_forest* method of $\mathcal{O}(|V|^2)$. If we however assume, that the longest path of a tree that has no building at the end has length of k, we can reduce the runtime complexity of the *traceback_forest* method to be $\mathcal{O}(k|V|)$. Nonetheless, the overall complexity of the random sb-forest algorithm will be $\mathcal{O}(|V||E| + k|V|) = \mathcal{O}(|V|^2)$. Again, we can conclude that $\mathcal{O}(|V|) = \mathcal{O}(|E|)$, with the same argument as before. Due to the lack of specialized data structures in the implementation of the random approach, we will also receive an average runtime complexity of $\mathcal{O}(V^2)$.

In **Figure 4** we can see the random sb-forest algorithm applied to "Bad Krozingen". The substations amount was set to 20 and the upper bound for buildings was set to infinity. The substations themselves are colored yellow, while their connected nodes take on a random but uniform color. Nodes and edges that stay black are not part of the sb-forest/ci-grid.

4.4.3 Minimum SB-Forest and Maximum SB-Forest

Algorithm

To start of, we will introduce a new temporary node to the grid. This node will be connected to every substation node through an edge with a weight of 0. Now we can use the built-in NetworkX method called *minimum_spanning_edges* which can be configured to use Kruskal's algorithm. Firstly, Kruskal's algorithm will sort all

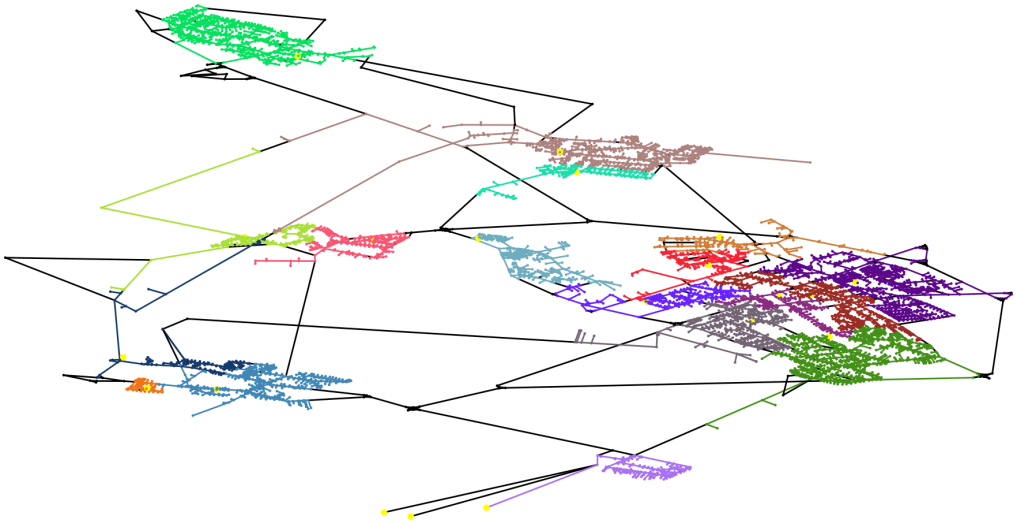

Figure 4: Random SB-Forest of Bad Krozingen with 20 substations

edges of the grid by their weights and store them in a list. After that, the algorithm will sequentially add the edges of the beforehand list, and check if the adding of the edge has resulted in a circle in the grid. If so, the edge will be discarded from the spanning forest. As we picked weight values of 0 for the edges connecting the added node to the substations, those edges will be the first to be selected to the spanning tree. After the algorithm finished running through the sorted edge list, we have a spanning tree. Now we can remove the temporary node and its linked edges. This will return us a spanning forest, for which we can once again use the *traceback_forest* method. In the end we will thus end up with a minimal sb-forest.

The maximum sb-forest can be generated in an identical fashion. The only difference is that we will have to invert the edges weights, so that the largest weights will become the smallest and vice versa. After the *minimum_spanning_edges* is called we can revert the weights to the original values. The rest of the process will be the same. **Algorithm 4** shows the pseudocode for a minimum spanning forest algorithm. It was inspired from the following slides[13].

Algorithm 4 Minimum spanning forest algorithm(grid=(V,E,w))

```
Connect temporary_node to every substation with weight 0
MST = {}
foreach node in grid do
    Create subset for node
end for
Sort edges in ascending order of weights
foreach edge(u, v) in grid (in ascending weight order) do
    if find(u) ≠ find(v) then
        MST ← MST ∪ edge(u, v)
        union(find(u), find(v))
    end if
end for
Remove temporary_node
Apply traceback_forest algorithm
```

Correctness

To prove that this algorithm will actually return us a sb-forest, we will initially have a look at Kruskal's algorithm. Kruskal's algorithm will always return a minimal spanning tree. Firstly, we need to introduce the definition of a subtree. In the following explanation, a subtree will be a subset of all nodes, that are connected and do not have a circle. The smallest subtree possible is a single node that is not connected to any other node. Kruskal will order the edges with the help of their weights. After that, it will iteratively suggest an edge to be part of the spanning tree. Now we can differentiate between two cases. Firstly, the selected edge can connect two nodes of the same subtree resulting in a cycle in that subtree. A tree should not contain cycles, thus we will not add the selected edge, and move on to next heavier weighted edge. If however, the selected edge does not produce a cycle, meaning it connects two different subtrees, they will be merged together into one larger subtree. Iterating over all the edges will force the subtrees to slowly merge into one large tree, which is connected to every individual node. To manage the different subtrees and keep track of which node belongs to which subtree, Kruskal's algorithm uses the Union-Find data structure[14]. In short this data structure can be represented

as a large array linking nodes through sequential references to their root/subtree. After receiving the spanning tree, we will remove the temporary node, which will break down the spanning tree into a spanning forest, where each subtree is exactly connected to one substation. Once again, the *traceback_forest* method will turn the spanning forest into a sb-forest.

Complexity

The runtime complexity of Kruskal's algorithm can be extracted from the combination of the runtime of sorting the list of edges and the checks if the grid contains a circle. Firstly, sorting the list will take $\mathcal{O}(|E|log|E|)$ where E represents the amount of edges. Such a time complexity can be achieved either through a heap sort or a merge sort algorithm. Secondly, checking for circles in each step can be done with the help of a Union-Find structure, which has an overall runtime of $\mathcal{O}(|E| + |V| \log |V|)$. The $|E|$ in the beforehand complexity comes from the algorithm iterating and testing each of the lines. The other term can be explained through the fact that we can at most combine the subsets $|V| \log |V|$ times until there is only one big subset left. After applying Kruskal to the grid, we once again apply *traceback_forest* which brings the worst-case runtime of the minimum and maximum sb-forest algorithm to $\mathcal{O}(|E| \log |E| + |E| + |V| \log |V| + |V|^2) = \mathcal{O}(|V|^2)$. We again used the observation that $\mathcal{O}(|V|) = \mathcal{O}(|E|)$.

However, if we consider the average case, the *traceback_forest* method's complexity could be simplified to $\mathcal{O}(k \cdot |V|)$, where k is the longest path (amount of edges) of a tree that has no building at the end. Thus, we can conclude that this algorithm has an overall average runtime of $\mathcal{O}(|V| \log |V| + k \cdot |V|)$. This complexity is probably a much more accurate estimate of the real runtimes.

In **Figure 5** we can see the minimum sb-forest algorithm applied to "Bad Krozingen". The substations amount was set to 20 and the upper bound for buildings was set to

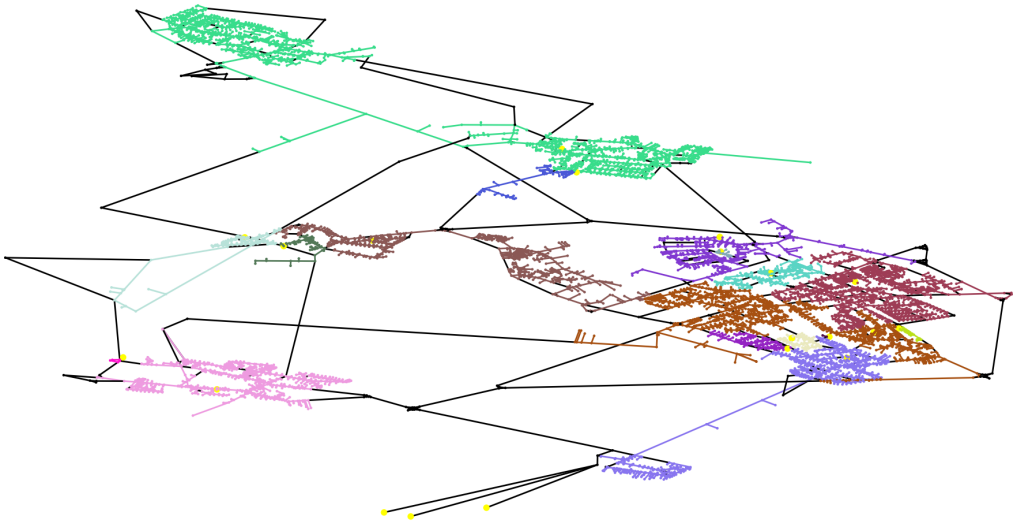

Figure 5: Minimum SB-Forest of Bad Krozingen with 20 substations

infinity. The substations themselves are colored yellow, while their connected nodes take on a random but uniform color. Nodes and edges that stay black are not part of the sb-forest/ci-grid.

In **Figure 6** we can see the maximum sb-forest algorithm applied to "Bad Krozingen". The substations amount was set to 20 and the upper bound for buildings was set to infinity. The substations themselves are colored yellow, while their connected nodes take on a random but uniform color. Nodes and edges that stay black are not part of the sb-forest/ci-grid.

4.5 Generating Output Files

Before we can properly generate the CSV files that will be interpreted by expansion planning algorithms, we need to tackle one more issue we are having with the present state of the grid. As for Lukas' Gebhard *ANTPOWER*, the algorithm assumes that every available new line that may be suggested to be added to the grid, must result in a circle in the grid. However, by looking at any of sb-forest algorithm figures, it

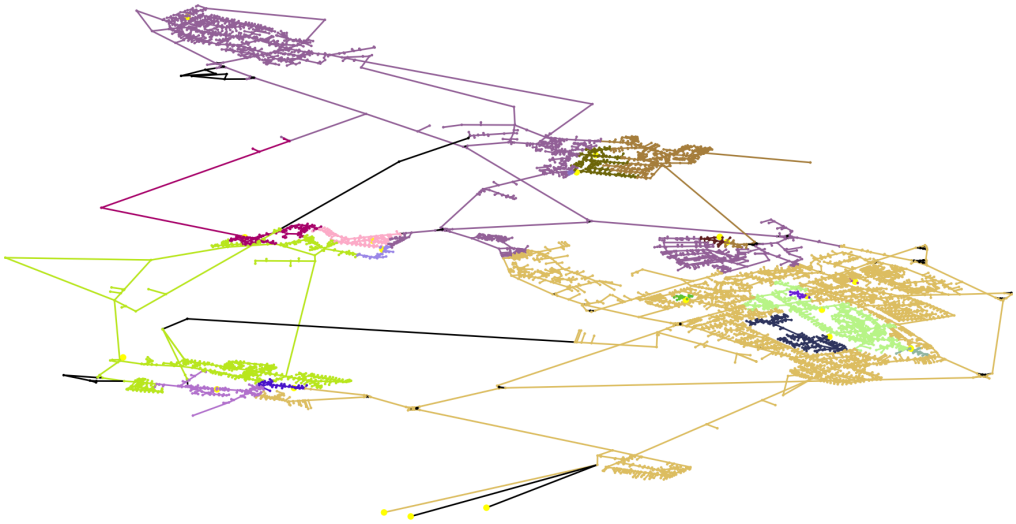

Figure 6: Maximum SB-Forest of Bad Krozingen with 20 substations

becomes clear that we can have cases, where an upgrade path should be considered instead of a single line, to produce such a circle. To fit this requested condition of *ANTPOWER*'s input, we developed a special *grid_smoothing* method. The goal of this algorithm is to unify possible upgrade paths into single edges, while not losing any information of distances and potential connection possibilities. The main idea behind the algorithm, is to iterate over all the nodes that are currently not part of the ci-grid, delete them and connect each of its neighbors with each other, while also adding the appropriate weights to those edges.

Note that in special cases, this implementation can worsen the actual length of the edges when "smoothing" a node. Imagine the special scenario, where an expansion planning algorithm would like to connect a smoothed node twice. With our implementation, the algorithm would than have to select two smoothed edges which are longer than the combination of the shortest path of the first two nodes we want to connect over the "smoothed" node coupled with the edge between the third node and the smoothed node.

Algorithm 5 shows the pseudocode for the grid smoothing algorithm.

Algorithm 5 Grid smoothing algorithm

```
foreach node in grid do
    if node not part of sb-forest then
        neighbors = Get neighbors of node
        foreach {neighbor1, neighbor2} in neighbors do
            tmp = weight(node, neighbor1) + weight(node, neighbor2)
            Add edge (neighbor1, neighbor2) with weight tmp
        end for
        Remove node and connected edges
    end if
end for
```

In the last step of the implementation, we have used a built-in CSV writer to help us return the grid and candidate lines as files. Those files are specially tailored to fit PyPSA's *import_network* methods.

5 Evaluation

In the following section we are going to describe how we evaluated the different sb-forest algorithms and determine which sb-forests are the most interesting in different scenarios.

5.1 Tests

Before we can look at the actual data, we need to define the setup and grids we tested the individual sb-forest algorithms on. The following evaluation was done on four different villages in the southwest of Germany. They have different sizes (amount of nodes and edges) and graph topologies.

Village	Nodes	Edges	Substation	Buildings
Wieden	434	439	1	206
Kirchzarten	4669	4732	14	2457
Bad Krozingen	9254	9510	11	4353
Freiburg i.Br.	60942	62490	91	29650

Table 1: Evaluation villages

Analyzing the three villages we can note that Bad Krozingen has fewer substations than Kirchzarten, even though Bad Krozingen is nearly double the size in nodes, buildings and edges. This is a good showcase of the lack of OSM data accuracy in different locations. We can however assume that Bad Krozingen must have more

than the 11 substations. We probably did not find them through our data extraction filters.

Nonetheless, we can simulate a proper grid by setting an arbitrary value for the lower bound of the quantity of substations. To simplify the evaluation and to get a better comparison, we will raise the lower bound of the substations for every of the three grids to 20.

Note that for Wieden, 20 substations may be unrealistic from the perspective of a grid operator. Nonetheless, we chose to pick this value as it will facilitate the comparison between the different algorithms in this evaluation part.

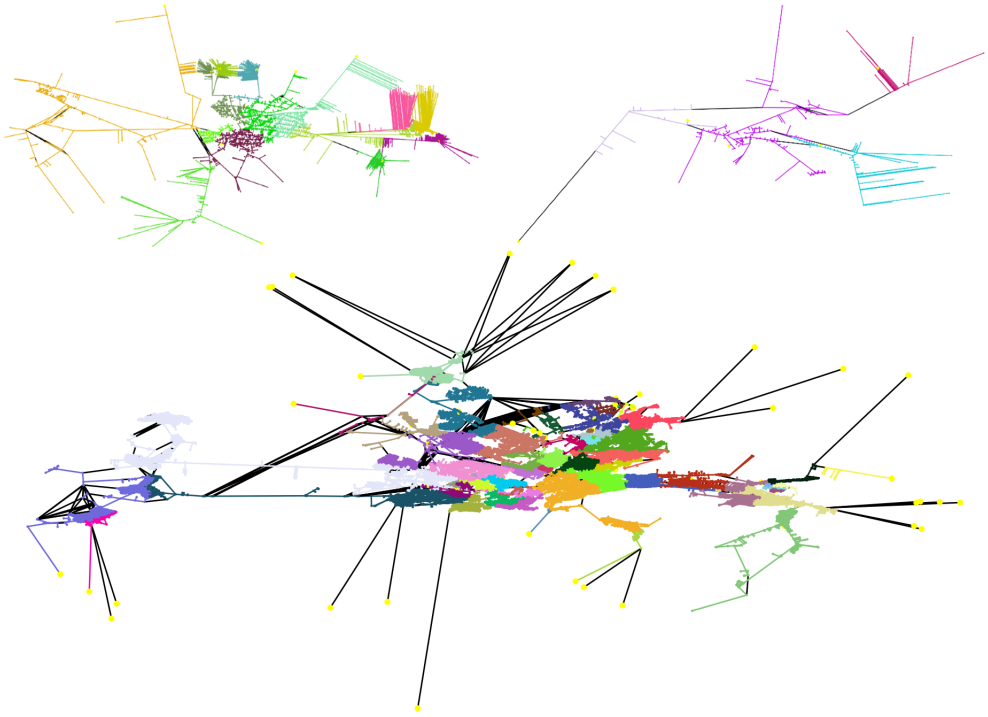

Figure 7: Left: Wieden, Right: Krichzarten, Bottom: Freibrug im Breisgau

In the following tests every algorithm was run 100 times on all villages except Freiburg im Breisgau. The largest grid was only tested 10 times per algorithm, and limited to at least 200 substations. It is important to note that the insertion of additional

substations at random positions is crucial for generating different forests for each test iteration. If we would not add new substations to the grid, every algorithm except for the random sb-forest algorithm will return a deterministic result.

The following results were averaged from the amount of iterations of each algorithm and grid. The *grid smoothing* method, that is needed for *ANTPOWER*, was explicitly not considered for the data extracted from the following results, as it would change the total amount of edges and nodes. This would obfuscate the actual performance of the algorithms.

5.1.1 Standard Deviation for Buildings per Substation

Firstly we wanted to have a look at the standard deviation for the quantity of buildings per substation. In short, we want to know how many buildings every substation deviates from the average amount of buildings it should have. This value essentially tells us if buildings are fairly distributed between substations, or if certain substations are struggling to feed a large amount of buildings, while others would still be able to feed more.

	σ(Shortest Path)	σ(Random)	σ(Minimum)	σ(Maximum)
Wieden	10.03	6.75	12.40	13.37
Kirchzarten	143.39	72.03	211.20	254.24
Bad Krozingen	196.60	128.99	276.68	573.36
Freiburg i. Br.	163.85	116.31	260.87	392.29

Table 2: Average standard deviation for buildings per substation

From **Table 2** we can see that the random sb-forest algorithm seems to generate the ”fairest” grids. This aligns with what we would expect, as it will sequentially distribute nodes between the individual substations. The minimum and maximum sb-forest algorithms seem to produce the largest inequalities in the distribution of buildings between the substations. This can be explained by the greedy nature of

those algorithms. They will not consider ”fairness” as they will only look to minimize resp. maximize their own conditions.

The question that may arise is why the minimum sb-forest algorithm is considerably ”fairer” than its maximum counterpart. This can be explained through the prioritization of edge selection of the both algorithms. While the maximum algorithm will preferably pick large edges, the minimum algorithm will firstly pick the small edges that are often in the close neighborhood of the substations. Additionally, we added synthetic substations, which will be placed at random positions instead of buildings. A village topology usually contains sets of buildings that are connected to each other, which can be interpreted as housing estates, residential or industrial areas. In the minimum case, if a substation is placed right inside such a grouping it is likely to be spanned to the entire set, as the edges used inside such sets are very short. On the other hand, the maximum sb-forest algorithm will firstly prioritize large edges, which are usually situated outside such groupings of buildings.

From this data we can conclude that the minimum and maximum sb-forest algorithm will generate more challenging grids. Such grids will have many consumers connected to few substations, raising the chance of electrical constraint violations in the grid. On the other hand, the shortest path and random sb-forest algorithm seem to produce much more balanced and distributed grids. Thus overloaded components become much less probable.

5.1.2 Weight Coverage

In this section we are going to have a look at the average overall length/weight coverage of all the cables that are part of the sb-forest. For this we will initially sum up all the edges weights that are part of the ci-grid, and divide them by the weight of all the edges in the entire grid (including candidate lines). Averaging those values over the amount of iterations of each algorithm will return us a percentage of

cable length that was used to generate the sb-forest. We will call this percentage the weight coverage of the sb-forest. Note that a 100% coverage would mean that every edge of the village was used to build the sb-forest.

	Shortest Path	Random	Minimum	Maximum
Wieden	83.76%	94.64%	80.88%	98.64%
Kirchzarten	96.12%	98.60%	94.11%	99.31%
Bad Krozingen	81.63%	93.07%	76.84%	95.35%
Freiburg i. Br.	83.01%	93.35%	79.93%	95.81%

Table 3: Average Weight coverage for 100 iterations

As expected, having a look at **Table 3**, the minimum sb-forest will return us the least coverage by far. On the other hand, the maximum algorithm has the completest coverage.

If a current grid has a large weight coverage, this will mean that a large quantity of cable is already placed in the ground. This leaves a good chance for potential upgrades by grid optimization algorithms.

Additionally, for grid operators, the total weight/length of the ci-grid is a useful metric as it directly correlates to the price of material and construction that was needed to build up that specific grid.

For the sake of completeness, the overall weights of the grids, meaning the sum of the ci-grid weights and the candidate lines can be read out of **Tabular 4**.

	Wieden	Kirchzarten	Bad Krozingen	Freiburg i. Br.
Length in km	36.62	297.37	222.26	1673.36

Table 4: Average overall grid weight in km

5.1.3 Runtimes

In this section we are going to have a look at the different examined runtimes of the algorithms. They were all run on an i7-7700HQ (mobile) processor. The following runtimes are measured in milliseconds.

	Shortest Path	Random	Minimum	Maximum
Wieden	11.69	25.06	21.47	25.54
Kirchzarten	287.97	738.80	271.85	250.16
Bad Krozingen	582.05	2969.30	880.65	694.11
Freiburg i. Br.	4108.22	37347.56	4902.56	4599.00

Table 5: Average runtime in milliseconds

As we can see from the data, the shortest path and the minimum/maximum sb-forest algorithm perform fairly the same. For the random sb-forest algorithm we received the largest average runtime during our tests. Looking at the different runtime complexities that we determined in chapter 4.4.1 to 4.4.3, we will note that in the worst case, every complexity is $\mathcal{O}(|V|^2)$. However, as we are averaging the runtimes over a large set of runs, we should also consider the average case complexities of the different algorithms. From this we can conclude that the shortest path and minimum/maximum sb-forest algorithm will outperform their forth competitor.

5.2 *ANTPOWER* on Wieden

To show the real application of this benchmark tool, we tested Lukas Gebhard's *ANTPOWER* with one of our grids. We chose the small grid of Wieden with five substations with no limitations on the amount of buildings and a 50% ratio for generators. Unfortunately the generated grid does not immediately run with the default electrical values. The problem is that PyPSA's *powerflow* will not always converge, and thus generate a very large error. This is a well known error and can

be troubleshoot[15]. This is probably related to us not properly being able to set up the electrical values.

Nonetheless, to prove the concept of our implementation we applied a length factor to all the lines in the grid. This will multiply the length of every line by a specific value. In our case we chose the value of 0.1 reducing the actual edge weights to 10% of their real weight. Now PyPSA's *powerflow* converges and *ANTPOWER* is able to optimize the grid. In the concrete example, we were able to reduce the initial ten constraint violations into zero over 200 iterations of *ANTPOWER*.

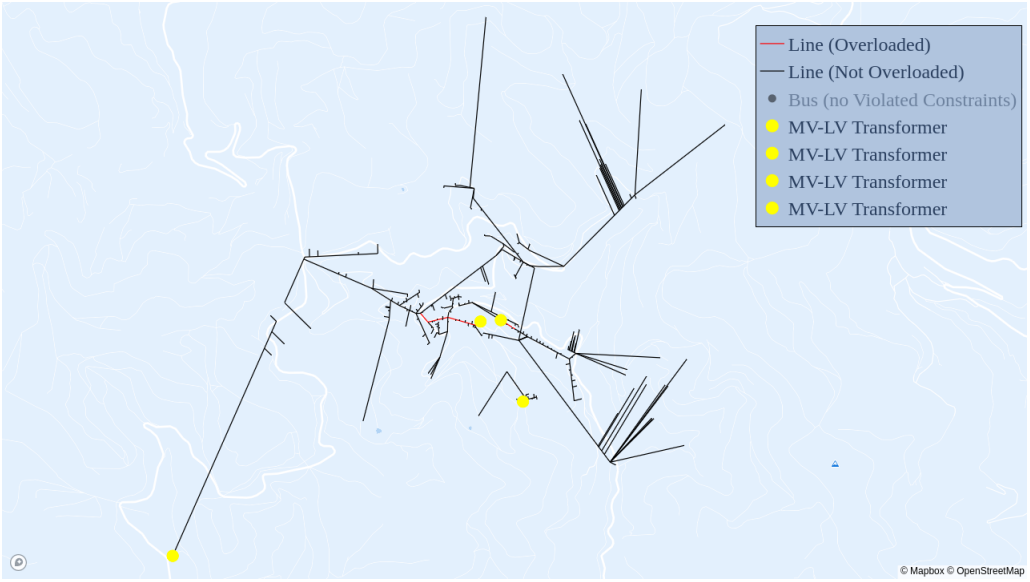

Figure 8: An example of constraint violations for Wieden with four substations

In **Figure 8** we can see an example grid of Wieden with four substations. In **Figure 9** we can see a zoomed in version of the same grid.

We expect that grid operators will be able to properly adjust electrical values to generate and test grids with their correct length of edges.

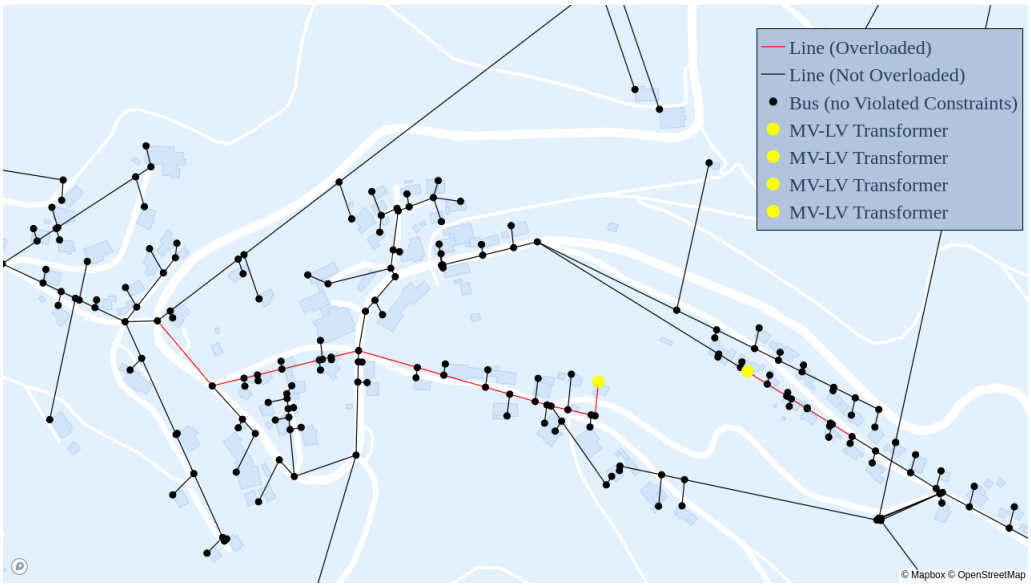

Figure 9: Zoomed in example of constraint violations for Wieden with 4 substations

6 Conclusion

With this thesis we propose an open source option to benchmark low-voltage grid planning algorithms. The benchmark tool offers a large flexibility and enables the user to specify and generate low-voltage grid problems with additional candidate lines. The user is not only able to define a large amount of electrical and physical variables, but can also choose one of the four sb-forest algorithms. Every sb-forest algorithm comes with its set of properties, advantages and disadvantages. Thus, the user can decide to make the low-voltage grid expansion problem a harder but also more unrealistic by generating a maximum sb-forest through this tool. Such problems tend to be much more synthetic, and we expect them to have many more constraint violations. The alternative is for the user to generate a more realistic problem, by selecting either the random, the shortest path or the minimal sb-forest algorithm. In that case the user will be able to test his expansion planning algorithm on a problem that should only have few constraint violations.

7 Acknowledgments

First and foremost, I would like to thank Matthias Hertel for proposing me the topic for this thesis. He supported me throughout my entire bachelor's project and thesis and motivated me to tackle the work. I am happy, that we could work together.

Secondly I would like to thank Prof. Dr. Hannah Bast for giving me the opportunity to present this work to.

Next in line is Felix Schoellen. He is a great friend of mine who stuck with me through large parts of my academic career and never let me down. He always made time to explain me subjects and learn with me for future exams. I would not be writing this thesis if it was not for your patience and persistence.
Merci Felix!

After that I would like to give a special thanks to Isabelle Desbordes and Georges Kapgen, my parents. They will not admit it, but they gave up a lot to make studying possible for my brothers and me. I hope the day will come where I can give you back the joy and good times I had in Zurich and Freiburg.
Merci Mamm a Papp!

My last thank you goes out to everyone that believed in me throughout this adventure. To my brothers, girlfriend, family and friends.
Thank you a lot!

Bibliography

[1] T. Brown, J. Hörsch, and D. Schlachtberger, "PyPSA: Python for Power System Analysis," *Journal of Open Research Software*, vol. 6, no. 4, 2018.

[2] L. Gebhard, "Expansion planning of low-voltage grids using ant colony optimization," 2021. Chair of Algorithms and Data Structures, University of Freiburg.

[3] J. Saat, S. Stein, M. Müllender, and A. Ulbig, "Planning and design of urban low-voltage dc grids," *Electric Power Systems Research*, vol. 211, 2022.

[4] A. Alhamwi, W. Medjroubi, T. Vogt, and C. Agert, "Openstreetmap data in modelling the urban energy infrastructure: a first assessment and analysis," *Energy Procedia*, vol. 142, pp. 1968–1976, 2017. Proceedings of the 9th International Conference on Applied Energy.

[5] G. Boeing, "Osmnx: New methods for acquiring, constructing, analyzing, and visualizing complex street networks," *Computers Environment and Urban Systems*, vol. 65, pp. 126–139, 07 2017.

[6] A. A. Hagberg, D. A. Schult, and P. J. Swart, "Exploring network structure, dynamics, and function using networkx," in *Proceedings of the 7th Python in Science Conference* (G. Varoquaux, T. Vaught, and J. Millman, eds.), (Pasadena, CA USA), pp. 11 – 15, 2008.

[7] W. Medjroubi, U. P. Müller, M. Scharf, C. Matke, and D. Kleinhans, "Open data in power grid modelling: New approaches towards transparent grid models," *Energy Reports*, vol. 3, pp. 14–21, 2017.

[8] M. Kapgen, "Testing and improving antpower on the simbench networks," [https://ad-blog.informatik.uni-freiburg.de/post/testing-and-improving-antpower-on-the-simbench-networks/; accessed 10-October-2022].

[9] "Simbench networks," [https://simbench.de/de/; accessed 10-October-2022].

[10] E. Hartvigsson, M. Odenberger, P. Chen, and E. Nyholm, "Generating low-voltage grid proxies in order to estimate grid capacity for residential end-use technologies: The case of residential solar PV," *MethodsX*, vol. 8, p. 101431, 2021.

[11] "Pyosmium," [https://osmcode.org/pyosmium/; accessed 10-October-2022].

[12] C. H. Thomas, L. E. Charles, R. L. Ronald, and S. Clifford, "Introduction to algorithms," vol. 2, p. 978, 2001.

[13] Uni-Paderborn, "Datenstrukturen und Algorithmen, Kapitel 16," [https://cs.uni-paderborn.de/fileadmin/informatik/fg/ti/Lehre/SS_2017/DuA/16.pdf; accessed 10-October-2022].

[14] Wikipedia, "Union-Find-Struktur — Wikipedia, the free encyclopedia," 2022. [https://de.wikipedia.org/wiki/Union-Find-Struktur; accessed 10-October-2022].

[15] "Pypsa - troubleshooting," [https://pypsa.readthedocs.io/en/latest/troubleshooting.html; accessed 10-October-2022].