Master Praktikum

Julius-Maximilians-
UNIVERSITÄT
WÜRZBURG

From Urban Network Visualization with the World Settlement Footprint to FISHNET

Lorenz Gruber
Department of Computer Science
Chair of Computer Science II (Software Engineering)

Dennis Kaiser, M. Sc
Advisor

Submission
12. January 2023

Abstract

To eliminate the shortcomings of the *WSF Urban Network Visualizer* it was transformed to framework FISHNET, which emerges to be a flexible and more generic framework to operate on network-related geographic information system data. Through the composition of the independent generic libraries of FISHNET, various heterogeneous use cases can be implemented. Moreover, major performance improvements in core algorithms of the framework are achieved by exploiting geometric and graph theoretical properties, lowering the asymptotic algorithmic complexity of for example the edge generation and edge contraction algorithms. The aforementioned performance improvements could be backed up with benchmarks, while the correctness of the software is checked using unit tests. Furthermore, FISHNET is brought to use in a cooperation with the University of Kassel, testing the robustness of the framework in various tasks. Ultimately, FISHNET strives to be a generic, reliable and flexible framework for network-related geographic data, while being easy to use for developers and users likewise.

Contents

1. Introduction

In the Bachelor thesis "Urban Network Visualization with the World Settlement Footprint "[1], a tool (*WSF Urban Network Visualizer*) for analysing and visualizing the relationships between settlements was development. Using leveraged satellite imagery from the World Settlement Footprint (WSF) Project [2] by the Deutsches Zentrum für Luft- und Raumfahrt / German Aerospace Center (DLR), a network of settlements can be constructed, which consequently can be utilized to determine central urban areas. With these result city planning in less developed regions can be improved, since reliably and up to date data often is not available [3]. Consequently, the *WSF Urban Network Visualizer* emerged to be a promising tool, with great interest of all involved parties to continue the joint project of DLR and the Chair of Computer Science II at the Univerity of Würzburg. Ultimately, the overall goal is the improvement of the software, especially regarding performance and flexibility, while also proving the software's correctness using extensive testing.

In this report the progress on achieving said goals will be documented by first giving a brief introduction about the *WSF Urban Network Visualizer*, mainly it's processing steps, use cases and flaws in Chapter 2. The following Chapter 3 introduces the Framework for Identification of Settlement Pattern Heuristics using Networks (FISHNET), by first outlining the major design principles of the framework and then providing details of the actual implementation, focusing on the *graph library*, which is the central component of the software. From Chapter 4 it can be comprehended, which concepts are used to improve the runtime of the software and how they were implemented. In the following Chapter 5 said performance improvements are verified by benchmarks. Moreover, this chapter outlines the extensive use of unit tests, to ensure the correctness of the software, as well. Thereafter a brief summary of the cooperation with the University of Kassel, which started during the workings on FISHNET, will be giving in Chapter 6. Finally the results of this master internship and possible future work will be summarized in Chapter 7.

2. Urban Network Visualization with WSF

This chapter provides background information about the *WSF Urban Network Visualizer* starting with the data source, in essence the WSF 2019 in Section 2.1. Thereafter, the necessary processing steps required to transform the input data to a network of settelements are highlighted in Section 2.2, with Section 2.3 showing possible use cases and outputs of the software. The final Section 2.4 discusses the shortcomings of the *WSF Urban Network Visualizer*, ultimately justifying the transformation to FISHNET.

2.1 World Settlement Footprint 2019

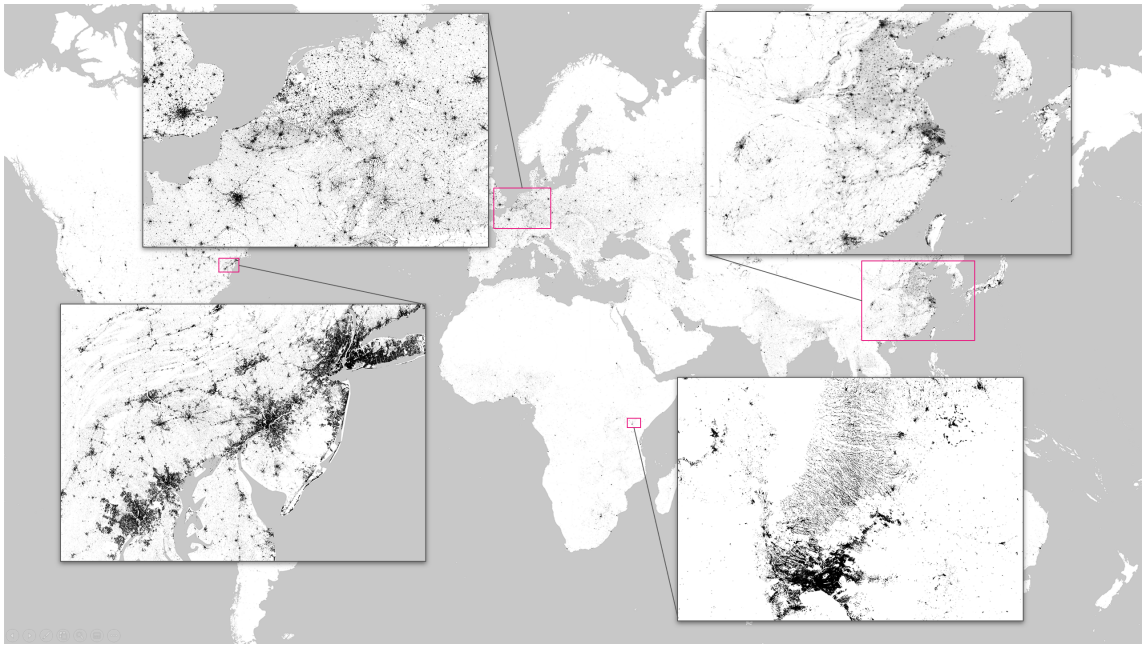

Figure 2.1: Binary map from the WSF 2015, highlighting the built-up areas in selected regions of the world, taken from [4]

The World Settlement Footprint 2019 Project by DLR, provides a worldwide delineation of settlements [2], which are publicly available in the *GeoTIFF* file format [5]. Like the WSF 2015, leveraged satellite data is utilized for training a classifier, which ultimately is able to distinguish areas of human built-up and regions without any settlements on

a ten by ten meter scale [4]. In addition to the binary map, which tells regions with human built-up apart from areas without human settlements, the WSF imperviousness and population dataset provide quantitative data about an area, by estimated the amount of sealed ground area as a percentage (imperviousness) and given an estimate about the amount of inhabitants resident in a certain ten by ten meter square. In Figure 2.1 an example of the binary map of the WSF 2015 is given. Said binary map for a region is used as an input to the *WSF Urban Network Visualizer* to determine the extent of a settlement [1].

2.2 Processing Steps of the WSF Urban Network Visualizer

With the input data from the WSF 2019 stored in the *GeoTIFF* [5] file format, the input first is transformed into a vector image, in essence into a shapefile. This is beneficial, since shapefiles allow the user to store additional attributes for each geometric feature displayed in the image [6]. After the conversion to shapefiles, the input data of geometric features, which might consist out of multiple WSF data sources, but at least the binary built-up layer, is adapted to settlement objects. Thereafter, edges between settlements are constructed, influenced on distance and impact of a settlement on its surroundings. Before computing centrality measures on the network to highlight important settlements, adjacent settlements can be aggregated into a common urban area. Finally the resulting settlements and edges, with their corresponding attributes, like area, population count or centrality values are written to a shapefile, which can be inspected with most common Geographic Information System (GIS) programms like QGIS[1]. These processing steps are abstractly visualized in Figure 2.2.

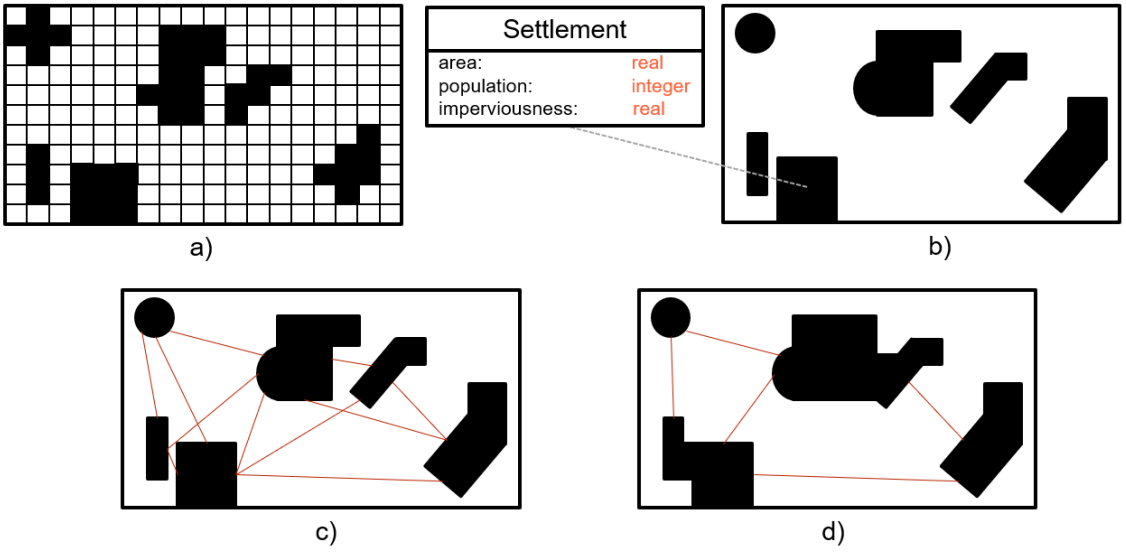

Figure 2.2: Illustration of the processing steps of the *WSF Urban Network Visualizer* taken from [1]

2.3 Use Cases and Examples

The major use case of the *WSF Urban Network Visualizer* is the construction of a settlement network based on up to three input files from the WSF datasets. Figure 2.3 illustrates how the software could be used from the command line, by providing a path to a *GeoTIFF* file covering a certain area for each of the processable datasets, namely the

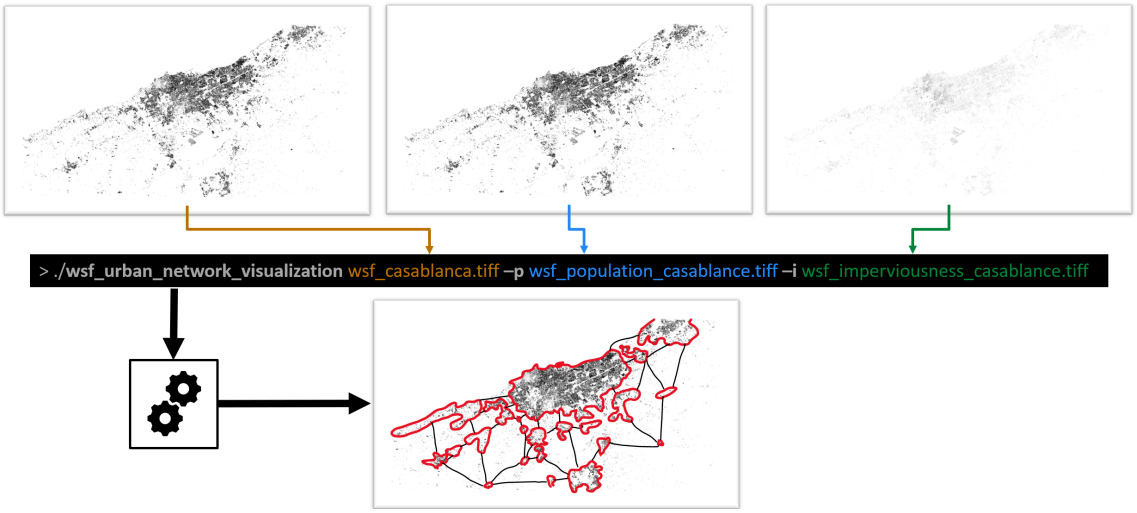

Figure 2.3: Schematic visualization of the main use case of the *WSF Urban Network Visualizer* [1]

binary built-up layer, the imperviousness and the population dataset. An example output viewed in QGIS is displayed in Figure 2.5, which is computed using the built-up WSF layer in the region to the west of the city of Casablanca, depicted in Figure 2.4, as an input. Moreover, adjacent settlements were merged in the process.

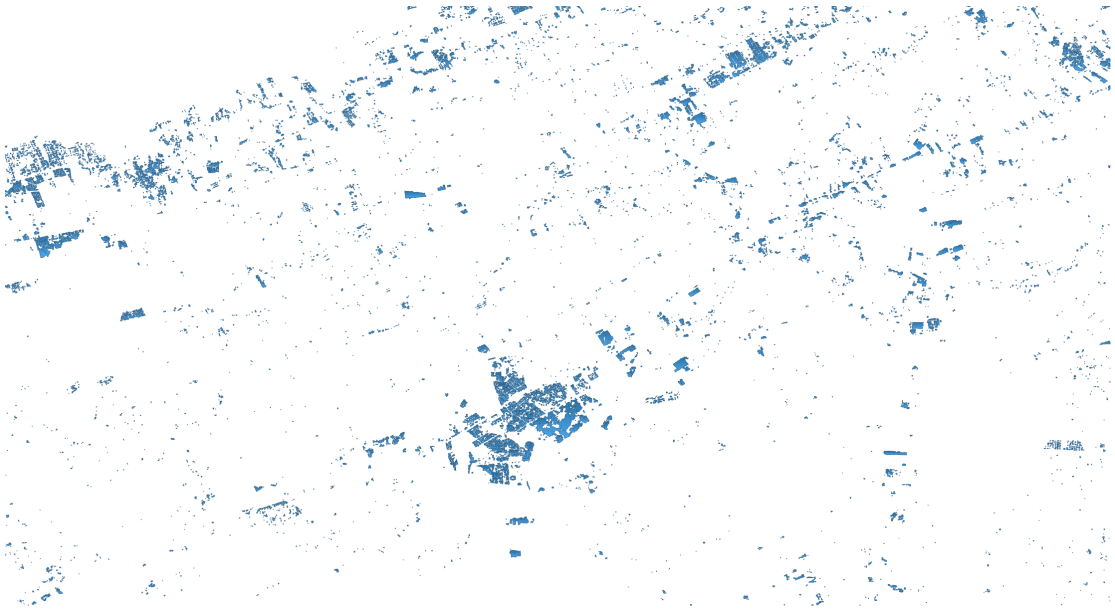

Figure 2.4: WSF built-up layer of the western suburbs of Casablanca, Morocco [2]

2.4 Shortcomings and Motivation for FISHNET

While the output of the software itself is a satisfying result according to experts of the DLR [1], the processing time of the software is not optimal and may take about two weeks to complete the computations on a moderate size dataset, e.g. the whole metropolitan area of Casablanca. Moreover, the *WSF Urban Network Visualizer* supports only the quite narrow use cases introduced in Section 2.3, while a lot of code could theoretically be

[1] https://www.qgis.org/de/site/forusers/download.html (last visited: 04.01.2023)

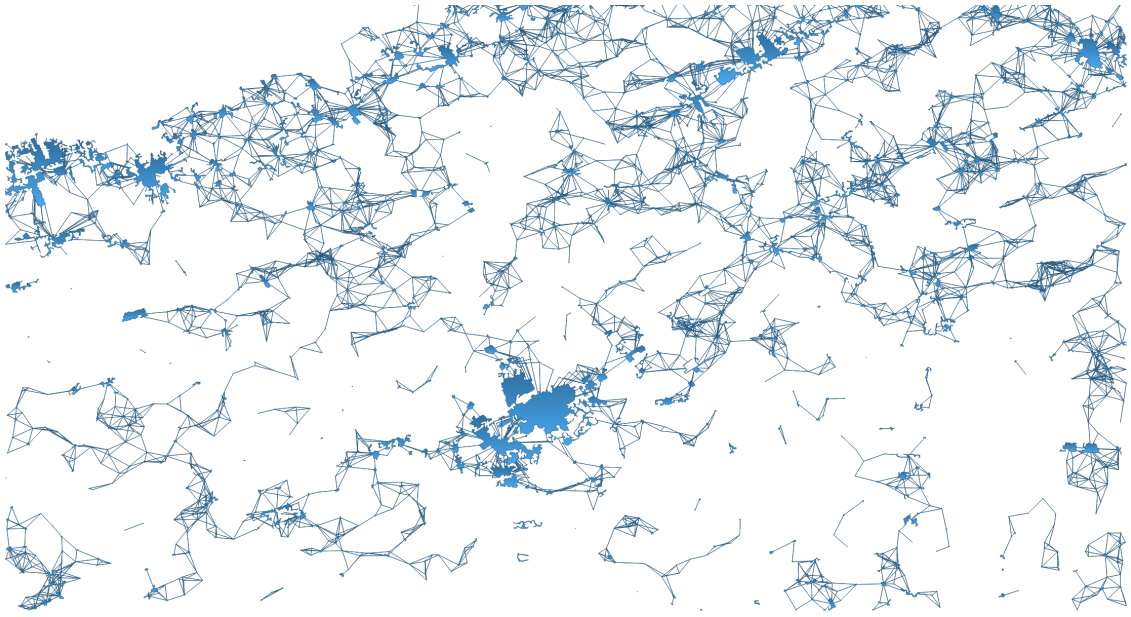

Figure 2.5: Output network of the *WSF Urban Network Visualizer* with enabled merging, taken from [1]

reused for related applications, e.g. the cooperation with the University in Kassel, which will be examined in Chapter 6. However, the composition of the software was designed with only one usage in mind. Therefore, it is lacking abstraction and genericity. The following listing summarizes the shortcomings of the *WSF Urban Network Visualizer*, which itself represents the motivation for the development of FISHNET:

- Develop independent, generic libraries for the different components of the software, namely a library for graphs, geometry-related operations, I/O operations and other supporting libraries.
- Provide filter options on the input dataset, e.g. discard small settlements to reduce the amount of vertices in the graph.
- Improve edge construction algorithm from a runtime of $\mathcal{O}(n^2)$ to a asymptotic complexity of $\mathcal{O}(n \log n)$
- Improve the contract algorithm (p.30 [1]) from a runtime of $\Theta(|E|^2)$, where $|E|$ is the amount of edges in the dataset, to a runtime of $\mathcal{O}(|V| + |E|)$, with $|V|$ being the amount of nodes in the graph.
- Speed up merge procedure by merging independent patches of settlements concurrently.

3. The Framwork FISHNET

Motivated by the shortcomings of the *WSF Urban Network Visualizer*, FISHNET is developed as an abstracting framework for the computation of network heuristics in a earth observation context. Therefore the framework must provided generic interfaces and algorithms to allow various heterogeneous use cases. Section 3.1 introduces import design principles and concept that are used in the development of fishnet, while Section 3.2 provides more details regarding the implementation of selected libraries.

3.1 Design Principles of FISHNET

One major aspect in the design of fishnet is the separation of concerns [7], by splitting the logic into multiple independent, generic libraries. For example, can the *graph* interface from the graph library used with standard *C++* data types like *int*, but also with geometric objects like polygons, as well as settlement objects, which them self contain polygons. To achieve this behavior, *C++* template classes are used extensively, as well as concepts[2], which were introduced with *C++ 20* and can constrain the type of template. For example, there are concepts which require the template parameter to be a number, to ensure that numeric operations compile.

Moreover, static polymorphism is utilized instead of dynamic virtual function, which require an additional overhead during runtime to dispatch the function call to appropriate child class. To achieve static polymorphism the Curiously Recurring Template Pattern (CRTP)[3] is applied. As illustrated in Figure 3.1 both *Derived1* and *Derived2* inherit from a common base, however the operation *function()* is not marked as virtual, instead the object is casted to its child class during compile time and calls the required function on the child class. When calling a function that requires an implementation of the base class, the type can be inferred by the compiler, as depicted in the function calls to *print()* in line 13 and 14 in Figure 3.1. Additionally, this procedure does not require pointers to achieve polymorphistic behavior, which is required when utilizing dynamic polymorphism with virtual functions. Since handling pointers is infamously error prone, this is another benefit of using CRTP.

[2]https://en.cppreference.com/w/cpp/language/constraints (last visited: 05.01.2023)
[3]https://en.wikipedia.org/wiki/Curiously_recurring_template_pattern (last visited: 05.01.2023)

```cpp
#pragma once


template<class T>
class Base{
public:
    int function()const{
        return static_cast<const T*>(this)->function();
    }
};

class Derived1 : public Base<Derived1>{
public:
    int function()const{
        return 42;
    }
};

class Derived2: public Base<Derived2>{
public:
    int function()const{
        return -1;
    }
};
```

```cpp
#include "CRTPExample.h"
#include <iostream>

template<class Impl>
static void print(const Base<Impl> & base){
    std::cout << base.function() << std::endl;
}

int main(){
    Derived1 d1;
    Derived2 d2;

    print(d1); // 42
    print(d2); // -1

    return 0;
}
```

Figure 3.1: Static Polymorphism using the CRTP to eliminate the runtime overhead of virtual functions

To implement a use case with FISHNET, the independent libraries can be linked with CMAKE[8], which is the default build tool for FISHNET.

3.2 Implementation of Independent Libraries

This Section provides more in-depth information about the different libraries composed in FISHNET, highlighting the arguably most relevant component, the graph library in Subsection 3.2.1 and introducing other relevant modules in Subsection 3.2.2.

3.2.1 Graph Library

The graph library can be split into two major components. The *graph_model* library defines the structure of a graph object, as well as the methods that operate on it. Furthermore, several concepts are necessary to e.g. constrain the types that can be used as vertices of the graph or that a weight function must produce an optional of the edge annotation when given two vertices as input. How this can be formalized is depicted in Figure 3.2. These concepts can be used to constrain the template parameters in a graph instantiation. As shown in Figure 3.3, the class *UndirectedGraph* is templated with a four parameters. The first parameter specifies the type of the vertices of the graph, which have to fulfill the *Node* constraint, therefore requiring the inserted type to be applicable to `std::hash` (concept Hashable from Figure 3.2) and to be comparable using the `==` operator. The edge annotation has to fulfill the *Number* constraint, while the third template parameter has to be weight function, which takes two *Node* instances as an input and produces and optional of the annotation type as an output. Designing generic code with these constraints, ensures that the required operations are defined an no undefined behavior can occur. Moreover, *C++* concepts generate more readable compiler errors, whenever a constraint is not satisfied.

In addition to the *graph_model* component, the graph library also provides a set of standard graph algorithms, which are bundled in the *graph_algo* (sub)library. These algorithms include Breadth-First-Search, Depth-First-Search, Connected Components and Contraction, which are formally specified in [8]. Additionally, another module for generic centrality

[4]https://cmake.org/overview/ (last visited: 05.01.2023)

```cpp
template<typename A>
concept Annotation = Number<A>;


template<typename N>
concept Node = Hashable<N> && std::equality_comparable<N>;

template<typename f, typename n,typename a>
concept WeightFn = Node<n> && Annotation<a> && requires (f, const n & n1, const n & n2){
    {f()(n1,n2)} -> std::convertible_to<std::optional<a>>;
};

template<typename N,typename A> requires Number<A> && Node<N>
struct EmptyWeightFunction
{
    std::optional<A> operator()(const N & n1, const N &n2)const {
        return std::nullopt;
    }
};

template<typename f, typename n>
concept NodeBiPredicate = Node<n> && (requires (f, const n & n1, const n & n2){
    {f()(n1,n2)} -> std::convertible_to<bool>;
})|| std::same_as<f,std::function<bool(const n &,const n &)>>;


template<typename f, typename N>
concept NodeBiFunction = Node<N> && requires(f, const N & n1, const  N & n2){
    {f()(n1,n2)} -> std::convertible_to<N>;
};
```

Figure 3.2: Implementation of template constrains using *C++* concepts

```cpp
template<Node N,Annotation A,WeightFn<N,A> WeightFunction = EmptyWeightFunction<N,A>, HashFunction<N> Hash = std::hash<N>, NodeBiPredicate<N> Equal = std::equal_to<N>>
class UndirectedGraph: public Graph<UndirectedGraph<N,A,WeightFunction,Hash,Equal>,N,A,WeightFunction,Hash,Equal,false>
```

Figure 3.3: Declaration of the *UndirectedGraph* template class, constraining the template parameters using concepts and inheriting from a common base graph, by applying CRTP

computations is planned, which do not require a specific type, for example for the computation of the Degree Centrality of a vertex, the underlying type is irrelevant, since only the number of outbound edges must be determined.

3.2.2 Further Supporting Libraries

Apart from the graph library, several other libraries are included in FISHNET, which will be listed in the following:

- `geometry`
  The geometry library, provides access to fundamental geometric operations on points, lines, rings and polygons. Furthermore, it acts as an adapter to calls to Geospatial Data Abstraction Library (GDAL) [9], which implements a lot of operations on vector data in the OGR module. Moreover, the geometry library defines algorithms for merging polygons and cooperates with the Computational Geometry Algorithms Library (CGAL)[5] to compute Delaunay Triangulations or Voronoi Diagrams [10].

- `io`
  This module is responsible for input and output operations on GIS files, mainly *GeoTIFF* and *Shapefiles*. It provides conversions from one file format into another, once again utilizing GDAL.

- `settlement`
  This library defines settlements as the composition of polygons and further attributes, like population count or imperviousness. Therefore its goal is to define a type, which can be applied to geometric operations and stored in a graph as well, while still maintaining a set of attributes.

- `util`
  The utility library bundles helper classes, which range from data structure like blocking queues, which will be explained in Section 4.1.2, calculating distances on an ellipsoid or some abstract concepts, which are required by several libraries (e.g. Determine if a type is hashable).

- `visualization`
  Finally the visualization module implements the visualization of vertices and edges of a network and exports them to the io library.

Combining these libraries can implement different heterogeneous use cases, for example the `io` library can read data from a shapefile, with the polygon and the attributes of each feature stored an a settlement object from `settlement` library. These settlements can be stored in a graph, given that a settlement fulfills the *Node* concept. Later on, adjacent settlements may be merged into a single settlement using the contraction algorithm, which requires a function, that determines how, in this case, two settlements are reduced to single one. To combine settlements, the *characteristic shape* [11] of both involved point sets is desirable, which is implemented in the `geometry` library. Finally the graph can be visualized with `visualization` library. However, there might be use cases, which do not require each component. An example for this scenario is given in Chapter 6.

[5] https://doc.cgal.org/latest/Manual/packages.html (last visited: 05.01.2023)

4. Performance Improvements and Implementation Details

After discussing the design principles and major components of FISHNET, this chapter highlights the improvement of important algorithms, highlighting the new contraction procedure in Section 4.1 and giving a brief explanation how to generate the edges of a graph faster using a Voronoi Diagram in Section 4.2.

4.1 Merging Vertices using Connected Components

While the former contraction algorithm applied in the *WSF Urban Network Visualizer* resembles a brute force approach with at least quadratic algorithmic complexity on the number of edges, the idea of the new contraction procedure, is to compute the set of vertices to be merged into one in advance and then reduce each independent set of vertices to a single vertex in parallel. To determine the a set of vertices which will be merged together, an adaption of *Connected Components* is used, which will be explained in Subsection 4.1.1. To parallelize the computations, the producer consumer pattern is applied, which is explained in Subsection 4.1.2. The final Subsection 4.1.3 provides more details on the implementation of the contraction algorithm that exploits connected components.

4.1.1 Connected Components

A connected component is a set of vertices where each vertex in the connected components is reachable from every other vertex by a path of edges [8]. A set of these components is inherently disjoint, because otherwise there would be a path between the two vertices of different components, so therefore they could be joined into a single connected component. Figure 4.1 illustrates a simple example of a graph with seven vertices, which form three disjoint connected components. Connected components can be efficiently computed using Breadth-First-Search (BFS) in $\mathcal{O}(|V| + |E|)$ (with $|V|$ = number of vertices and $|E|$ = number of edges). For the contraction step, a *predicate* on the edge, i.e. a *bipredicate* on the pair of nodes, determines whether an edge should be contracted with adjacent vertices merged together. Therefore, these modified connected components, are sets of vertices, which are reachable from another with a path of edges, and on each of these edges (pair of adjacent vertices) the predicate must be true. Ultimately, this results in disjoint sets of vertices, which will be merged together and the corresponding edges being contracted. This procedure is explain in detail in Section 4.1.3.

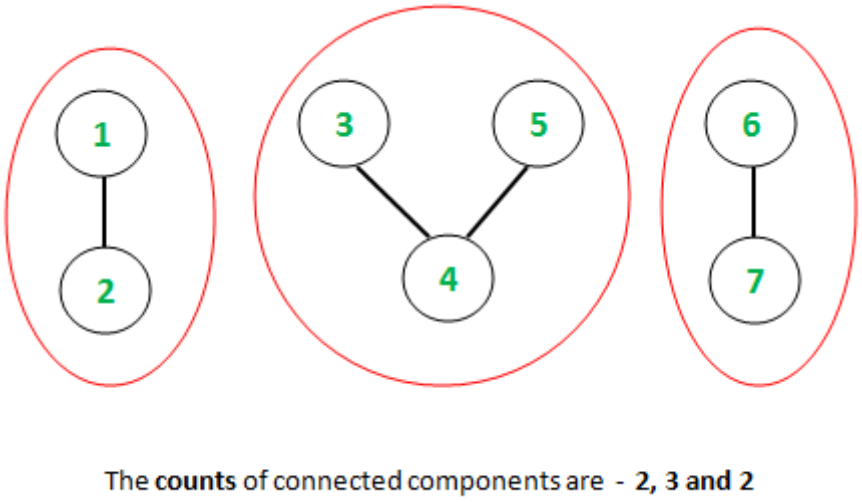

Figure 4.1: Simple example of three connected components, taken from https://www.geeksforgeeks.org/count-of-unique-lengths-of-connected-components-for-an-undirected-graph-using-stl/ (last visited: 06.01.2023) The connected components are: $\{1, 2\}, \{3, 4, 5\}, \{6, 7\}$

4.1.2 Producer Consumer Pattern

In order to parallelize an application the Producer-Consumer pattern can be applied, first mentioned by Edsger W. Dijkstra in [12]. It consists of three major components, the producer(s), the consumer(s) and a blocking queue, which acts as a buffer. As depicted in Figure 4.2, one (or possibly multiple) producer add data/workload to the buffer, which usually is implemented with a blocking queue. One or multiple consumer remove data/-workload from the buffer to process it. The Consumer will be blocked (it has to wait), whenever the queue is empty. The main advantage of this pattern, is that the producer can independently produce data to be processed, while the consumer can start processing, even though the producer might not be finished yet. Furthermore, this pattern is scales well, by just adding more producer or consumer instances.

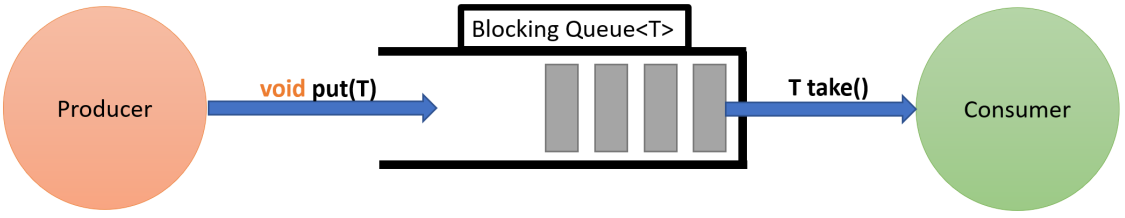

Figure 4.2: Illustration of the Producer-Consumer Pattern

4.1.3 Implementation of the Contraction Algorithm

The new contraction algorithm, combines both a modified version of connected components and the Producer-Consumer pattern. The algorithm is schematically visualized in Figure 4.3 and gets a graph as an input. This graph is then traversed using BFS, with vertices, which share a connection through possibly multiple edges and with each involved edge fulfilling a contraction predicate (e.g. distance $\leq 100m$), being grouped together in a *Connected Merge-Component*. These disjoint components can then be processed by one or multiple merge entities (act as consumer), to merge a set of vertices into a single one. After the BFS is finished and all *Connected Merge-Component* are reduced to a single vertex, the edges that were not contracted are added to the new output graph entity, which is the output of the procedure. The overall runtime of the procedure is $\mathcal{O}(|V| + |E|)$, since BFS takes $\mathcal{O}(|V| + |E|)$ time and the merging of nodes is in $|\mathcal{V}|$. This improved contraction

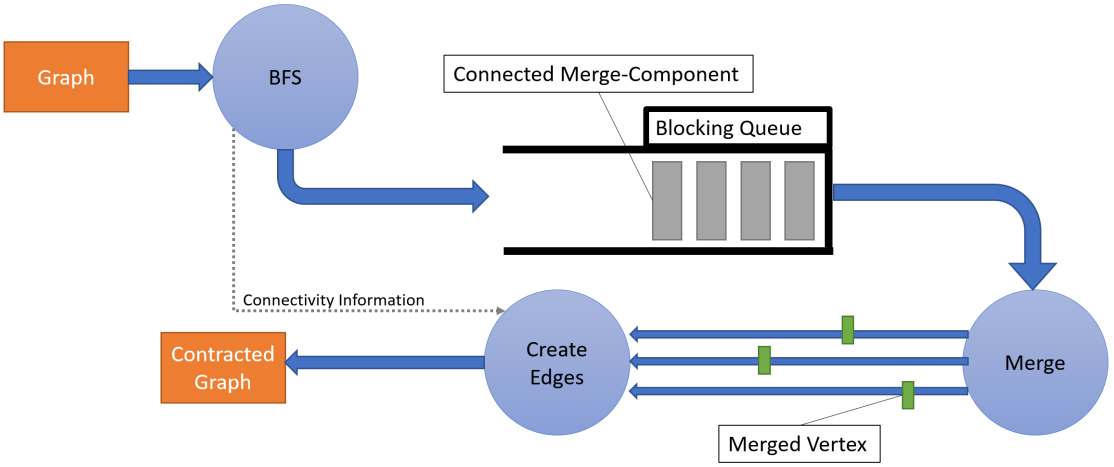

Figure 4.3: Schematic Visualization of the Contraction Algorithm. The input graph is first traversed using BFS, gathering connected components consisting out of merge-able vertices. While the BFS is still processing, the Merge-Components can already start reducing a set of vertices into a single vertex. Finally, the remaining edges are constructed using connectivity information obtained during the BFS.

algorithm vastly outperforms the former brute force approach in asymptotic complexity and in actual benchmarks as well, as Section 5.2 shows.

4.2 Edge Generation with Voronoi Diagrams

The major drawback in the legacy edge generation algorithm, was that for each vertex v, all other n vertices were inspected, with the best k vertices (according to a comparison function, e.g. the distance between the vertices) getting connected with v through an edge. Therefore, this procedure has quadratic complexity in the number of vertices. In order to improve performance, another approach is need with a explicitly smaller than $\Omega(n^2)$, for example with a complexity of $\mathcal{O}(n \log n)$. To achieve this improved runtime, the amount of possible neighbour candidates must be reduced, such that the number of possible neighbour candidates is not linear on average. Computing Voronoi Diagrams [10], produces a desirable division of the plane, which will be explained in Subsection 4.2.1. How to use said diagrams to generate the edges of a graph more efficiently is described in Subsection 4.2.2.

4.2.1 Voronoi Diagram

A Voronoi diagram consists of sites (input data), *Voronoi Edges*, *Voronoi Faces* and *Voronoi Vertices* [10]. In Figure 4.4 an example of a Voronoi diagram is given. The black dots are the sites, which is the input to the construction of a Voronoi diagram. Each of the colored polygonal areas, is a *Voronoi Face*, with each point in the face, being closest to the site that defines the face. The boundary between to faces is a *Voronoi Edge*, with all points on that edge, being equally close to the neighbouring sites. A *Voronoi Vertex* simple is the intersection of multiple *Voronoi Edges*, so a *Voronoi Vertex* is equally far away from at least three sites. A Voronoi diagram, given a set of input points, can be compute efficiently in $\mathcal{O}(n \log n)$ [10].

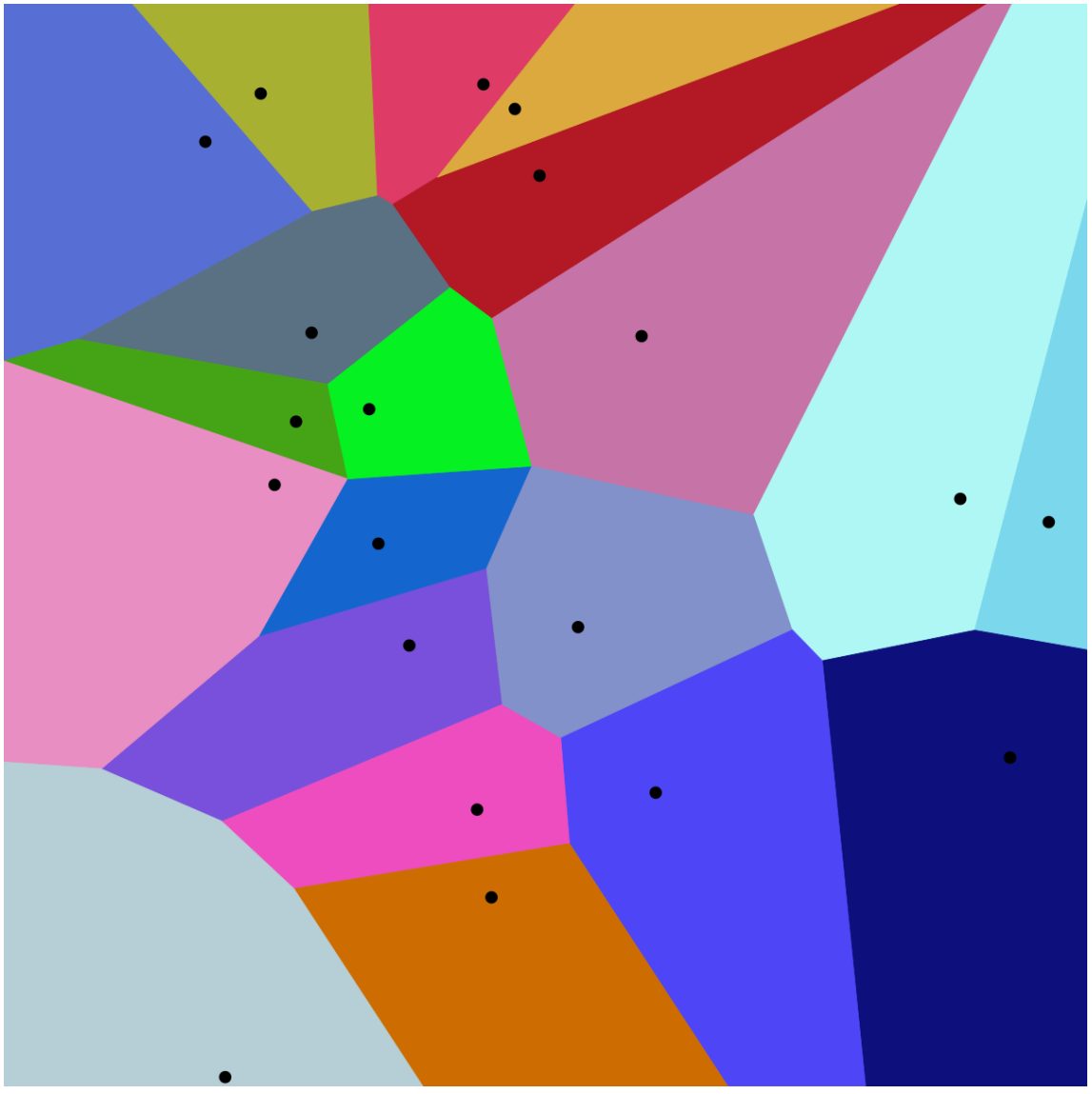

Figure 4.4: Voronoi Diagram example, with *Voronoi Faces* depicted in distinct colors[6]

[6] https://upload.wikimedia.org/wikipedia/commons/thumb/5/54/Euclidean_Voronoi_diagram.svg/1200px-Euclidean_Voronoi_diagram.svg.png

4.2.2 Edge Generation Algorithm

For this approach the *Voronoi Edges* are of fundamentally important, since these only exists between two sites p and q if and only if there is at least one point in the plane which is equally distant form p and q and all other sites are more far away. Therefore, the property of a *Voronoi Edge* can be used to reduce the number of possible neighbours in a euclidean graph scenario, since all sites that do not share an *Voronoi Edge* with a specific site p, cannot be closest to p, therefore only the vertices that share an *Voronoi Edge* can be considered, reducing the number of candidates significantly. Since most GIS application use polygons instead of single points, the centroid point (explained in [1]) can be used instead.

The overall procedure to generate the edges, starts with computing the Voronoi diagram on a set of centroid points of polygons in $\mathcal{O}(n \log n)$, while storing for each site (each polygon centroid), all associated *Voronoi Edges*. Finally, for each vertex only amortized constant amount of possible neighbours must be considered, resulting in a amortized linear runtime. So overall the algorithmic complexity of the edge generation using Voronoi diagrams is in $\mathcal{O}(n \log n)$, beating the brute force approach with quadratic complexity.

5. Evaluation

This chapter is dedicated to presenting the testing methods used to ensure the correctness of the software in Section 5.1 and proving the performance improvements of the contraction/merge algorithm using benchmarks in Section 5.2.

5.1 Unit Testing

During the development of the *WSF Urban Network Visualizer*, testing was not a priority, due to the limited time frame of a bachelor thesis. In order to provide a reliable framework, FISHNET introduced unit tests for each data structure and algorithm. FISHNET utilizes the *GTest*[7] (GoogleTest) testing framework, which provides a macros to validate that an actual computation is equal to the expected result, fixtures, that can be used to guarantee certain preconditions for each test (e.g. construct a test graph before applying the contraction algorithm). *GTest* can be linked with *CMAKE*[8] and is compatible with *CMAKE's* CTest. During past and ongoing development of FISHNET, testing is a fundamental part, with each component (data structure or algorithm) tested to verify the correctness of computations and internal state changes.

5.2 Performance Benchmarks

In this Subsection, the benchmark to measure the performance difference between the legacy contraction procedure and the contraction algorithm based on connected components is presented, as well as the impact of multiple merge entities compared to a single one (See Section 4.1.3 for more details on the algorithm).

At first the performance difference from the legacy contraction procedure to the connected component contraction algorithm will be measure. The test set consists of undirected graphs, with a at least 10 and up to 900 vertices. Each vertex has three neighbours and about 10% of edges are eligible for contraction. The results in Figure 5.1 show a major improvement in runtime using the new contraction procedure, taking always less than 5s, while the brute force approach may take multiple minutes to complete. The benchmark was discontinued after 900 nodes, since the runtime of the legacy approach skyrocketed. In contrast, the new contraction procedure achieves acceptable runtime, even with larger

[7] https://google.github.io/googletest/primer.html (last visited: 06.01.2023)
[8] https://cmake.org/overview/ (last visited: 05.01.2023)

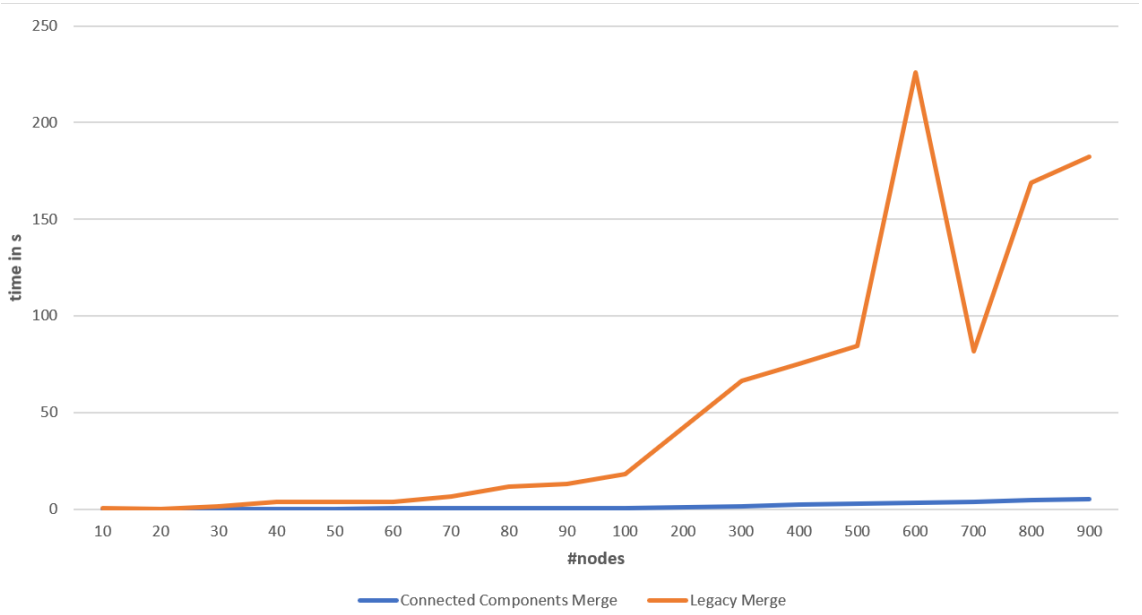

Figure 5.1: Runtime in seconds of the brute force approach (orange) and the connected components contraction algorithm (blue) on graphs with varying size, from 10 to 900 nodes. Each vertex has three neighbours before the contraction and 10% of edges will be contracted

input sizes. As depicted in Figure 5.2, the connected components contraction algorithm, completes in about one minute for 9000 nodes, when applying the same preconditions as in Figure 5.1. However, when the amount of neighbours per vertex is reduced to one and only about the 1% of edges are eligible for contraction, more threads are faster, since the amount of disjoint components is larger, that must be extracted from the queue. As depicted in Figure 5.3, using eight merger threads achieves the best performance. Increasing the number of neighbours and the percentage of contract-able edges to five neighbours and 20% once again does not benefit nor suffer from multiple threads as depicted in Figure 7.1 in the appendix.

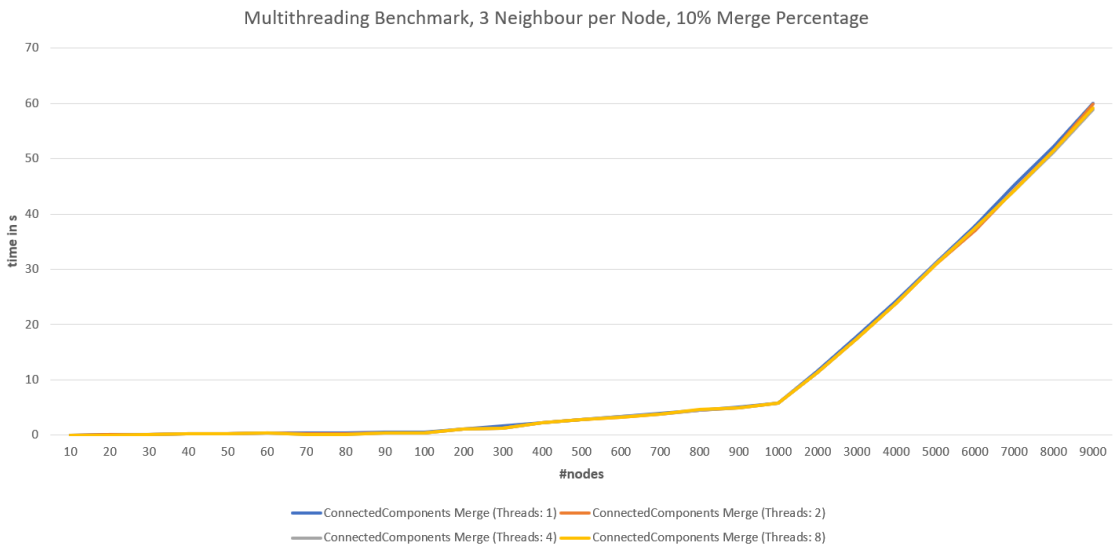

Figure 5.2: Runtime in seconds of the connected components contraction algorithm with different amount of merge instances. In this scenario the amount of threads does not make a significant difference.

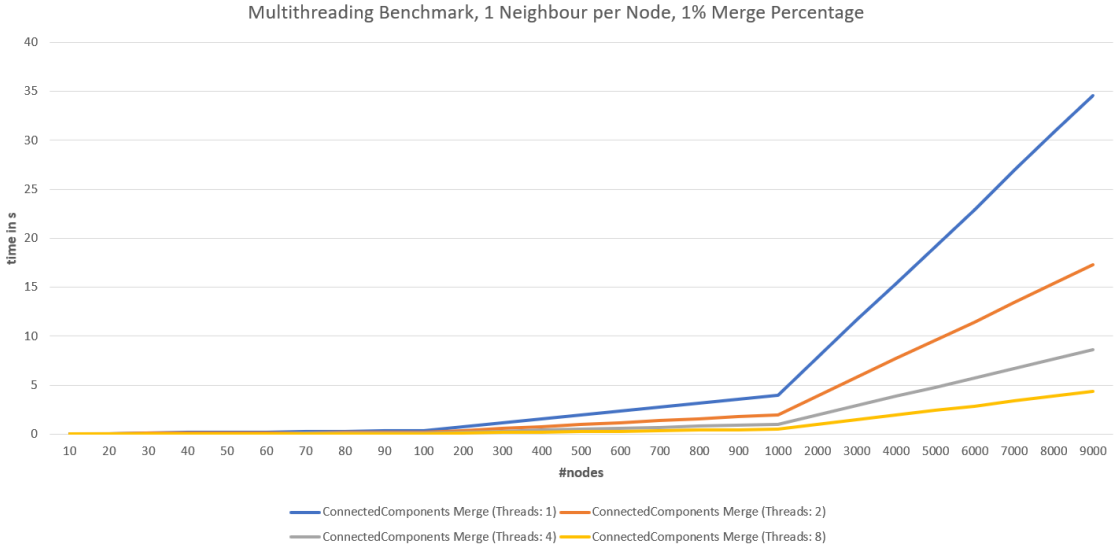

Figure 5.3: Runtime in seconds of the connected components contraction algorithm with varying amount of threads. The input consists out of 10 and up to 9000 nodes, with one neighbour per node and 1% of edges will be contracted.

6. Cooperation with the University of Kassel

During the transformation from *WSF Urban Network Visualizer* to FISHNET a cooperation with the DLR and a geographic research group from the University of Kassel was initialized. In their paper "Spatial Patterns of Urbanising Landscapes in the North Indian Punjab show Features predicated by Fractal Theory", Dr. Nguyen and her research group analyzed the spatial patterns in Punjab manually, using software like QGIS [13]. However, the laborious task of manually analysis is not desirable for further large-scale pattern analysis in other regions. Therefore, FISHNET shall be utilized as an automatic analysis tool, to free up resources in the research team. Since the cooperation is only at the beginning, the hole procedure is not implemented yet, however, an analysis on the settlement size distribution in Punjab was already automated, validating their manual results. Figure 6.1 shows the settlement size distribution in Punjab, compute with FISHNET, while their original results for the distribution of settlement sizes can be comprehend on page four in their paper [13]. Both approaches yield similar distributions. Nguyen

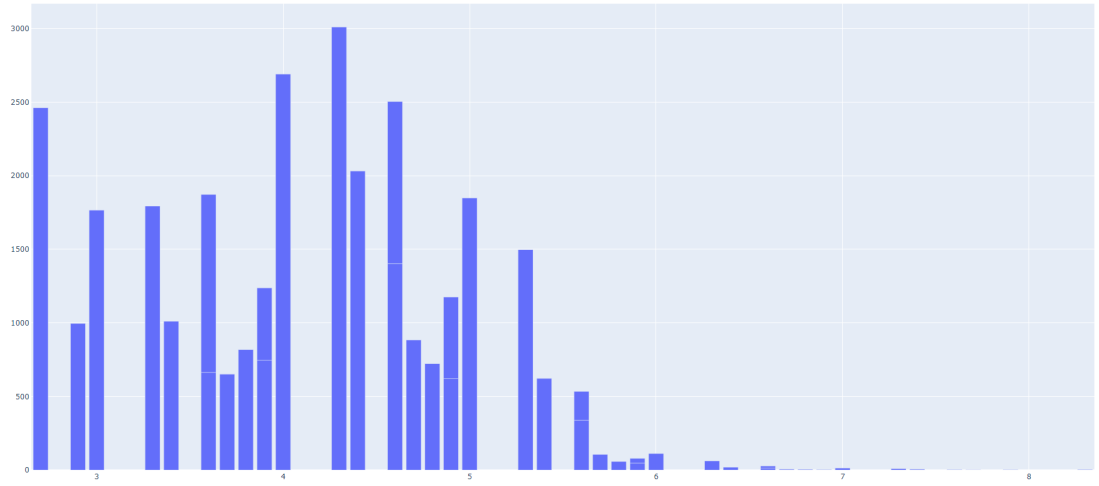

Figure 6.1: Settlement size distribution in Punjab, India computed using FISHNET. The x-axis encodes the settlement area on a logarithmic scale, with the labels being exponents of 10 in square meters. The y-axis accounts for the quantity of settlements with a certain size.

7. Conclusion

Coming from the *WSF Urban Network Visualizer*, the more abstracted and generic framework FISHNET improves flexibility to implement other heterogeneous use cases, that vastly benefits the current cooperation with the University of Kassel (mentioned in Chapter 6). Furthermore, the asymptotic complexity of core algorithms is provably improved, as Chapter 4, dealing with the algorithmic and implementation, and Chapter 5, proving the performance gains in benchmarks, show. Through extensive testing, the correctness of the operations is ensured, uncovering and solving multiple flaws from *WSF Urban Network Visualizer*. Ultimately, development on FISHNET will be continued, with focus on improving the `io` and `geometry` library in the near future and providing convenience features, like intermediate saving and a preview option, before applying the framework to huge dataset. The cooperation with University of Kassel will be continued, with contiguous extension and improvement on FISHNET, with the overall goal to construct a reliable, flexible and user-friendly framework for network-related GIS use cases.

List of Figures

Acronyms

WSF World Settlement Footprint

DLR Deutsches Zentrum für Luft- und Raumfahrt / German Aerospace Center

GIS Geographic Information System

GDAL Geospatial Data Abstraction Library

FISHNET Framework for Identification of Settlement Pattern Heuristics using Networks

CGAL Computational Geometry Algorithms Library

CRTP Curiously Recurring Template Pattern

BFS Breadth-First-Search

Bibliography

[1] L. Gruber, "Urban Network Visualization with the World Settlement Footprint," 2022.

[2] Thomas Esch, Mattia Marconcini, Julian Zeidler, "German Aerospace Center (DLR): World Settlement Footprint 2019," Confidential Dataset, 2021.

[3] T. Esch and V. Deparday, "Satellite Monitoring Service of Urbanization in Africa - Final Report (not published yet)," Technical Report, 30.10.2021.

[4] M. Marconcini, A. Metz-Marconcini, S. Üreyen, D. Palacios-Lopez, W. Hanke, F. Bachofer, J. Zeidler, T. Esch, N. Gorelick, A. Kakarla, M. Paganini, and E. Strano, "Outlining Where Humans live, the World Settlement Footprint 2015," *Scientific Data*, vol. 7, no. 1, p. 242, 2020.

[5] Open Geospatial Consortium, "OGC GeoTIFF Standard," 14.09.2019. [Online]. Available: http://www.opengis.net/doc/IS/GeoTIFF/1.1 (Accessed 17.01.2022).

[6] Environmental Systems Research Institute, "ESRI Shapefile Technical Description," July 1998. [Online]. Available: https://www.esri.com/content/dam/esrisites/sitecore-archive/Files/Pdfs/library/whitepapers/pdfs/shapefile.pdf (Accessed 17.01.2022).

[7] E. Gamma, R. Helm, R. E. Johnson, and J. Vlissides, *Design Patterns: Elements of Reuseable Object-Oriented Software*, 1994.

[8] Thomas H. Cormen Charles E. Leiserson Ronald L. Rivest Clifford Stein, *Introduction to Algorithms*, 3rd ed. Cambridge, Mass.: MIT Press, 2009.

[9] C.-Z. Qin and L.-J. Zhu, "GDAL/OGR and Geospatial Data IO Libraries," *Geographic Information Science & Technology Body of Knowledge*, vol. 2020, no. Q4, 2020.

[10] M. T. de Berg, O. Cheong, M. van Kreveld, and M. H. Overmars, *Computational Geometry: Algorithms and Applications*, 3rd ed. Berlin and Heidelberg: Springer, 2008.

[11] M. Duckham, L. Kulik, M. Worboys, and A. Galton, "Efficient Generation of Simple Polygons for Characterizing the Shape of a Set of Points in the Plane," *Pattern Recognition*, vol. 41, no. 10, pp. 3224–3236, 2008. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0031320308001180 (Accessed 24.01.2022).

[12] E. Dijkstra, *Cooperating Sequential Processes*. Technological university, 1965. [Online]. Available: https://books.google.de/books?id=i_F_GwAACAAJ

[13] T. T. Nguyen, E. Hoffmann, and A. Buerkert, "Spatial patterns of urbanising landscapes in the north indian punjab show features predicted by fractal theory," *Scientific Reports*, 2022.

Appendix

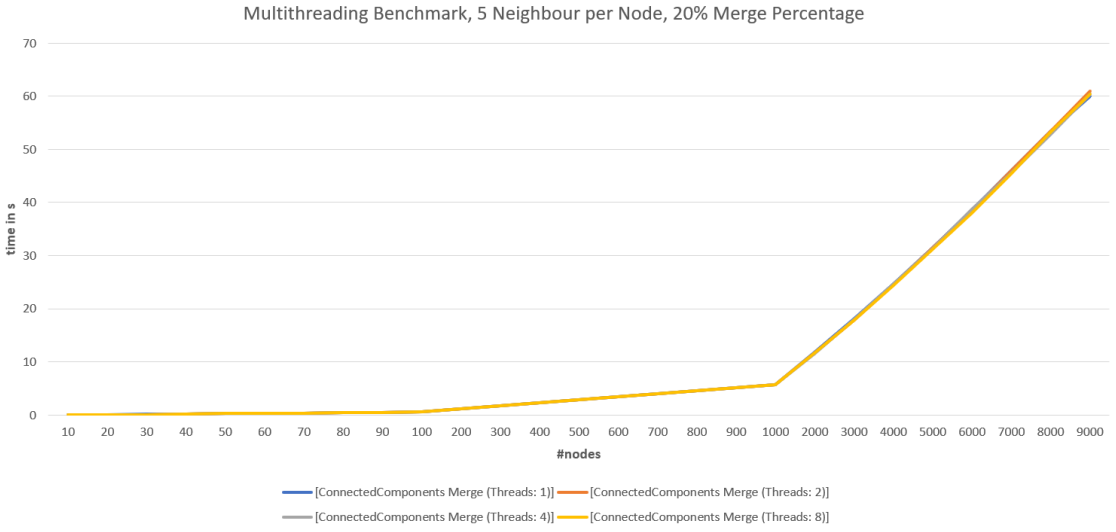

Figure 7.1: Runtime in seconds of the connected components contraction algorithm with varying amount of threads. The input consists out of 10 and up to 9000 nodes, with five neighbour per node and 20% of edges will be contracted

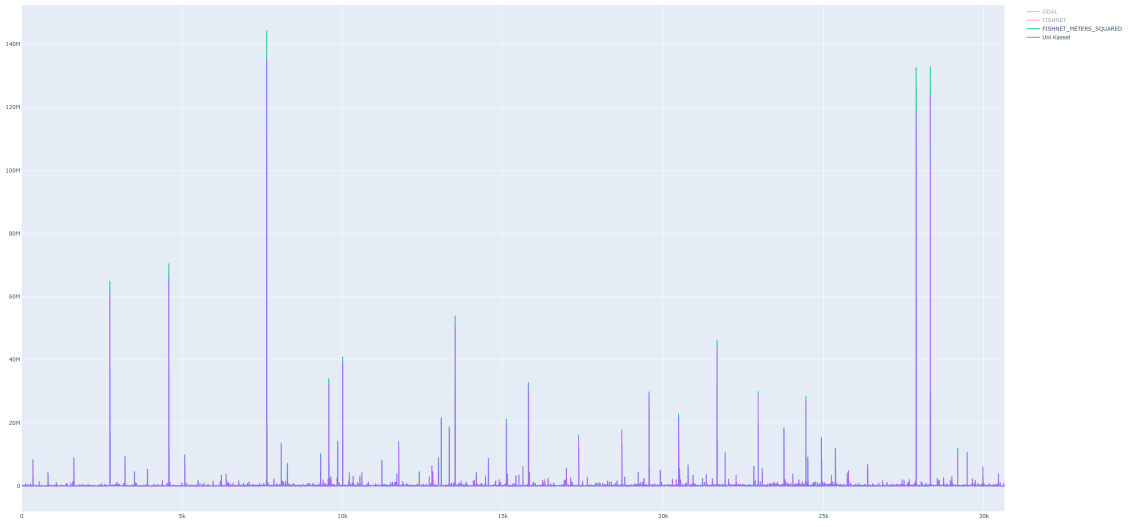

Figure 7.2: Area Computation Deviation of FISHNET versus manual computation by Dr. Nguyen, yielding almost identical results. The x-axis just enumerates the polygon dataset, while the y-axis accounts for the settlement size in square meters. The minor deviation is explain by the fact, that FISHNET disregard inner holes in the polygons, therefore yielding a bit larger area for some polygons