Karlsruhe Institute of Technology

Zooming Out: Generalization of Geometric Graphs

Diploma Thesis of

Edith Brunel

At the Department of Informatics
Institute for Theoretical Computer Science
Algorithmics I
Karlsruhe Institute of Technology (KIT)

Reviewer:	Prof. Dr. Dorothea Wagner
Second reviewer:	Prof. Dr. Peter Sanders
Advisor:	Andreas Gemsa
Second advisor:	Marcus Krug
Third advisor:	Ignaz Rutter

Duration: 16. August 2010 – 15. February 2011

Disclaimer

I hereby declare hat I have completed this diploma thesis on my own, using no material or sources other than those indicated in the text and specified in the bibliography.

Karlsruhe, den 15.02.2011

Deutsche Zusammenfassung

Die Diplomarbeit beschäftigt sich mit der visuellen Abstraktion geometrischer Graphen, einer Problemstellung die in der Literatur bis jetzt noch nicht eingehend behandelt wurde. Das Hauptziel ist einen ersten Ansatz zur automatischen schrittweisen Generalisierung von Graphen zu schaffen. Dazu werden zunächst die notwendigen Grundlagen diskutiert und wichtige Teilprobleme und deren Komplexität erörtert. Den Hauptteil bilden mehrere grundlegende Abstraktionstechniken, sowohl für die Knoten- als auch die Kantenmenge eines Graphen, und deren Evaluation. Diese identifiziert hierbei einerseits einfachere Instanzen die mit solchen Methoden lösbar sind, aber zeigt auch Graphstrukturen und Eigenschaften auf die Schwierigkeiten für eine Abstraktion bieten und gibt einen Ausblick wie man diesen begegnen kann.

Contents

1. Introduction

When working with large amounts of data, it is usually desirable to display the information graphically in order to gain a basic understanding of inherent features and reveal underlying patterns. Geometric graphs are versatile tools for visualization of data correlations with a wide range of applications, as information can be encoded in both the graph's structure as well as its layout. While fast layouting techniques are available for reasonably complex [FLM95] or specialized [Tam97] instances, drawing large dense graphs in detail is unnecessarily expensive if only a quick rudimentary overview is required. Moreover, even with a good layout large graphs still clutter the display due to screen space limitations, and too much information obscures rather than helps. Therefore, it may be preferable to settle for a more abstract high-level representation which emphasizes the graphs defining characteristics and hides details that would be imperceptible anyway.

Aims

Our goal is a system for automatic gradual generalization of geometric graphs. This is a very diverse problem that covers a large class of subproblems and has many degrees of freedom. The most obvious application is zooming, which motivates the notion of a zoom factor, i.e., a parameter that regulates abstraction based the level of detail required. The abridged graph should visually resemble the original, akin to a mental map, and additionally preserve its structure and properties as much as possible, but be easily discernible at an appropriate zoom level. Figure 1.1 shows an example, the gradual abstraction of a regular grid. While this may generally be a trivial task for humans, for whom identifying important structures at a single glance comes naturally when categorizing graphs, artificial intelligence has a considerably harder time establishing what properties actually define a graph and distinguish it from others.

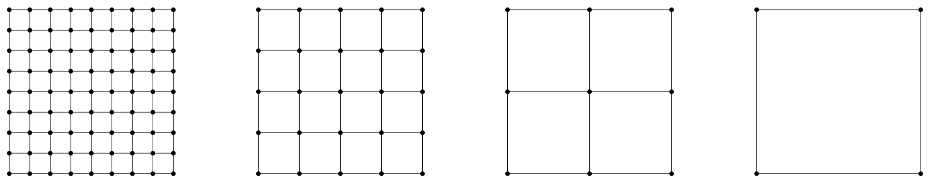


Figure 1.1.: Gradual abstraction of a regular grid.

Related Work

Abstraction of general large-scale geometric graphs has as of yet not been widely discussed in literature. Rather than computing a condensed representation of a graph when screen space is limited, visualization techniques focus on transforming its layout to allow a local zoom, such as fish-eye view [KRB95] or rubber sheet [SSTR93].

Published works mainly deal with very restricted graph classes and instances that are predestined for multilevel viewing due to their natural hierarchical properties, mostly as a side-product or in conjunction with the graph's construction. Examples are complex behavior graphs with states order [MPK96], modal logic graphs [BBKR08] and cluster graphs [EF97, QE01].

One approach that has been examined for general graphs is random sampling as as way to effectively visualize large networks [RC05].

Lastly, [HL04] briefly introduces two general methods for graph abstraction, k-Clique Minimization and Centrality Erosion. These work independent of the graph's layout, however.

Scope

As indicated, the focus of this thesis is on undirected geometric graphs with a predefined layout. The generalization therefore has to balance two objectives, namely pure visual similarity and identification of graph properties that are indicative of this on the one hand, as well as structural similarity and defining characteristics. This is not always easy to judge, as is apparent from the example in Figure 1.2, which delivers a good visual abstraction but loses the 3D mesh property apparent from the original in the process. We evaluate in how far a meaningful abstraction independent of graph class and with no restriction on layout is possible and hint towards harder instances which require special treatment as applicable. Continuous zoom, while a large motivation, requires that nodes should not move around between zoom levels, or disappear and reappear from one zoom level to the next. This demands consistency for the abstraction process, which is a strong restriction. We therefore focus on the static case, i.e., more or less one-time generalization at coarse zoom intervals.

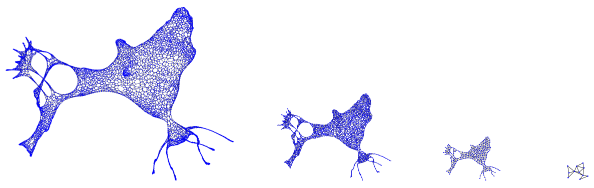


Figure 1.2.: Gradual abstraction of a graph with 3D structure.

Outline

The further structure is set up as follows. First, some preliminaries are explained in the subsequent Section 2. These include basic definitions and algorithms, a discussion on how resemblance of graphs can be measured and an overview of our testing framework. Next, several basic generalization methods focusing on abstracting the node set are introduced and evaluated in the main Section 3. While these techniques also produce an induced edge mapping, its quality varies. Section 4 presents several quick filters to improve the generalized edge set in a post-processing step, while Section 5 examines more advanced techniques for mapping edges that take structural properties of the original graph into account. A general overview and discussion concludes the thesis in Section 6.

2. Preliminaries

Starting with a formal definition of the generalization problem, some preliminaries required during the course of the thesis are explained. Apart from basic terminology and foundations, we also describe how the quality of an abstraction can be measured and introduce our visualization and testing framework.

2.1. Problem Particulars

The standard input for generalization are two-dimensional embeddings of undirected simple geometric graphs, more specifically Euclidean graphs.

Definition 2.1 (Euclidean Graph) *A Euclidean graph is a graph $G = (V, E)$ with node set V and edge set $E \subseteq V \times V$, whose nodes $v \in V$ are distinct points in $\mathbb{R}^2$ and whose edges $\{u, v\} \in E$ are straight line segments weighted with the Euclidean distance $\|u - v\| = \sqrt{\sum_{i=1}^{2} (u_i - v_i)^2}$ between their two endpoints.*

Note that while most of the techniques described herein are optimized for the Euclidean distance, they also work with a general metric as edge weight.

As mentioned above, we also require a zoom factor z regulating the degree of abstraction. The generalization process can then be formally defined as a transformation $\phi : (G, z) \mapsto G'$ that maps a geometric graph $G = (V, E)$ to a new geometric graph $G' = (V', E')$. We also consider the two restricted problems of generalizing the node and edge set separately. Note that abstraction methods for the node set which provide a mapping $\phi_v : V \to V'$ gain an induced mapping for the edge set by extending ϕ_v to $\phi_e : E \to E', \{u, v\} \mapsto \{\phi_v(u), \phi_v(v)\}$.

2.2. Measuring Graph Similarity

In order to judge the quality of our abstraction, a measure for graph similarity is required. We focus mainly on aesthetic constraints, which are intrinsically hard to define using graph properties.

Cheong et al. [CGK+09] propose two metrics for general geometric graphs. The first defines an edit distance, which can be computed using an ILP solver for small graphs but is not feasible for large-scale input. The second is a heuristic based on landmarks that only considers point set distance, which is not necessarily significant as high abstraction levels only retain a fraction of the original number of nodes.

We therefore do not rely on only one measure, but rate abstraction according to basic graph characteristics and visual qualities, such as

- **visual clutter**
  the entire graph should be clearly viewable at an appropriate zoom level
  - **node cluttering**
    nodes that are too close together may be indistinguishable
  - **edge cluttering**
    particularly short and long parallel edges may be hard to discern
- **node-to-edge ratio**
  the ratio $\frac{|V|}{|E|}$ should be approximated reasonably
- **node set properties, e.g.**
  - **degree distribution**
  - **density distribution**
- **edge set properties, e.g.**
  - **angle distribution**
  - **length distribution**

The demands on node and edge set, both visual and structural, are mostly independent of each other, which further suggests separating the generalization into node and edge set abstraction.

2.3. Planar 3SAT

We provide proof of NP-hardness for two related problems, which both rely on reduction from the NP-complete planar 3SAT problem [Lic82].

Definition 2.2 (Planar 3SAT) *For a given 3SAT formula φ, the variable-clause graph G_φ is constructed with variables and clauses as nodes. Variables are connected to each clause they occur in by an edge. Instances of planar 3SAT are Boolean 3SAT formulas for which G_φ is planar, i.e., can be embedded in the plane such that none of the clause edges cross.*

According to [KR92], a planar variable-clause graph can be laid out in polynomial time with clauses in a rectilinear configuration as shown in Figure 2.1, with the set of variables arranged on a straight line and connected by nested clauses as three-legged combs on either side. This makes an reduction easy for many geometrical problems by constructing the graph step-by-step out of smaller gadgets corresponding to variables, literals and clauses.

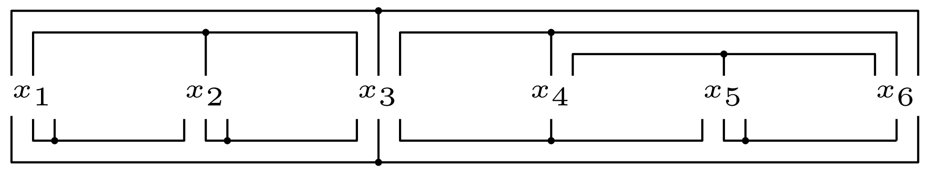


Figure 2.1.: Planar 3SAT variable-clause graph.

2.4. Visualization

Our visualization creates plain graph drawings[1], with the edge set painted first in black (anti-aliasing enabled) and nodes overlaid in blue. All input graphs have integer coordinates.

Standard labelling of graph figures for evaluation includes first an abbreviation denoting the generalization method, followed by the number of nodes, number of edges and level of abstraction, for example {No Generalizer n500 e500 l5}. The abstraction level corresponds to the zoom factor z, but is considered 0 for all levels for which the generalization does not modify the graph and is counted upwards.

2.5. Test Instances

This section documents the sample graphs used throughout the thesis. Since we measure the quality of an abstraction visually, a small representative set of test instances is used for evaluation. It contains both larger examples from real-world applications as well as selected smaller instances with special structural properties, such as regularity. For a quick overview on basic statistics refer to Table 2.1.

data set	graph name	# nodes	# edges
OpenStreetMap	Berlin	106 675	124 163
Sparse Matrix	Commanche	7 920	11 880
	Cep1	6 290	8 233
	Aug3d	24 300	34 992
Unit Disk	UR10	100	1 012
	UR30	500	3 523
yEd	Binary Tree	1 023	1 022
	Star	250	249
	Polygon	1 000	1 000
	Grid	900	1 740
	Clique	100	4 950
	Triangulation	100	294

Table 2.1.: Test graph statistics.

2.5.1. Street Graphs

One common class of graphs with fixed layout are road networks. They are valuable for testing, since specialized techniques that exploit additional information, such as road categories, are available and provide very good reference abstractions. All street graphs used for testing purposes are taken from the OpenStreetMap[2] data set available under the creative commons license. These graphs are usually sparse, have a disproportionately large number of very short edges, and feature a very distinctive structure that favours angles of about 90° and 180° degrees. Only one is examined here in detail, which is the representation of Berlin depicted in Figure 2.2 with 106 675 nodes and 124 163 edges.

[1] using the Qt SDK
[2] map data (c) OpenStreetMap (http://www.openstreetmap.org/) and contributors, CC-BY-SA (http://creativecommons.org/licenses/by-sa/2.0/)

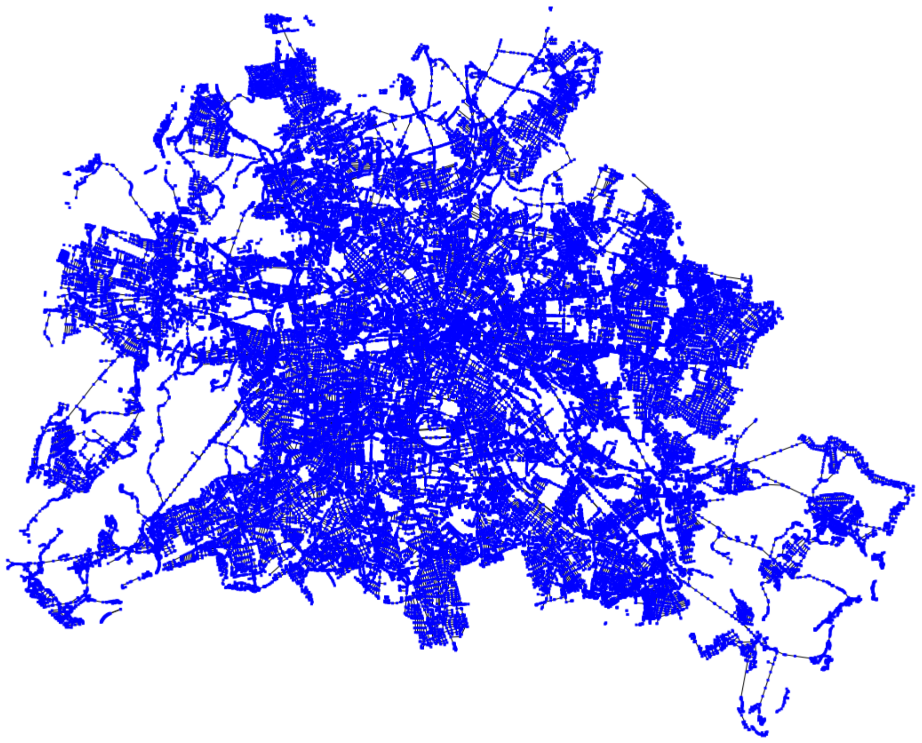


Figure 2.2.: OSM Berlin graph.

2.5.2. Sparse Matrix

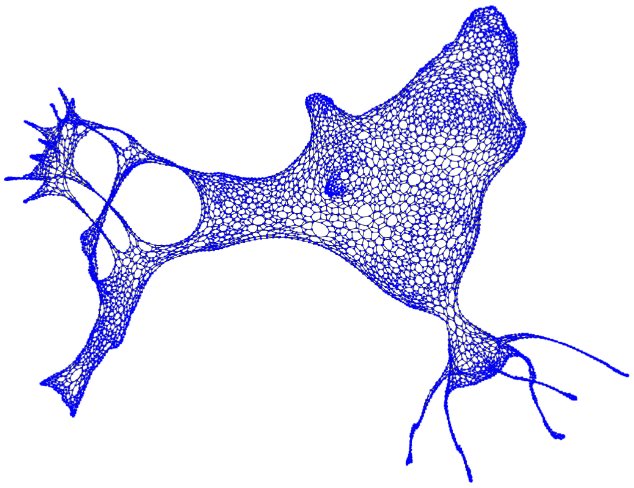
A large number of very diverse sparse graphs can be obtained from the University of Florida sparse matrix collection[3] (cf. [Dav97]). We picked three and computed a layout for each using the scalable force directed placement algorithm (sfdp) introduced in [Hu05]), which is a multilevel force directed algorithm available as part of the graphViz suite [EGK$^+$03].

The first instance is the Cep1 graph (Figure 2.3(b)), a deterministic equivalent of a stochastic linear programming problem. At first glance the drawing resembles a single star with spherical layout. However, the graph actually consists of four central hubs from which edge strings spread out, which are also weakly interconnected. Abstraction should reveal this.

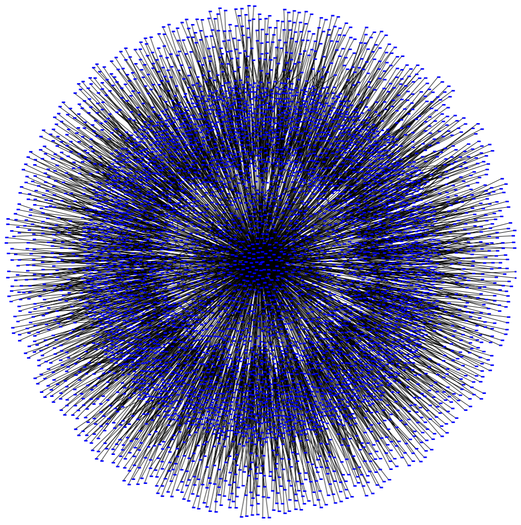
Next is a two-dimensional projection of a 3D-mesh, a model of the AH-66 Comanche Helicopter (cf. Figure 2.3(a)). All vertices have degree three. The main challenge for the generalization is here to preserve enough of the original structure to make sure it is still clearly recognizable as a 3D object.

The last example is another graph with a very unique structure, the representation of a quadratic journal bearing problem (cf. Figure 2.3(c)) which turns out as a cube containing multiple layers of edge strings. Although this graph's structure is three-dimensional as well, this time the focus lies more on its regularity, with angles almost exclusively around 45°, 90° and 135° degrees.

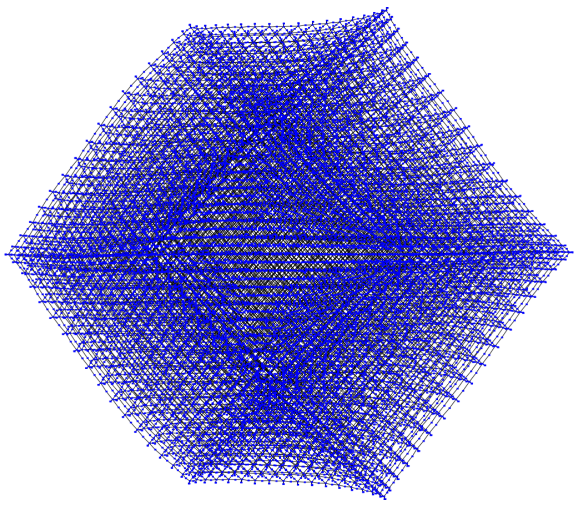
[3] University of Florida CISE (http://www.cise.ufl.edu/research/sparse/matrices/), maintained by Tim Davis and Yifan Hu



(a) Commanche



(b) Cep1



(c) Aug3D

Figure 2.3.: Sparse matrix graphs with sfdp layout.

2.5.3. Unit Disk Graphs

For somewhat denser common geometric instances we use unit disk graphs, which have a large application in sensor networks. Unit disk graphs are intersection graphs of equal-radius circles. Nodes represent the center points and two nodes are connected if their distance is below the radius threshold. Figure 2.4 shows two randomly generated examples for evaluation, UR30 (Figure 2.4(a)) with 100 nodes and a radius of 30% the graph's diameter, and UR10 (Figure 2.4(b)) with 500 nodes but only 10% radius. The generalization should respect both the maximum edge length property as well as the density distribution.

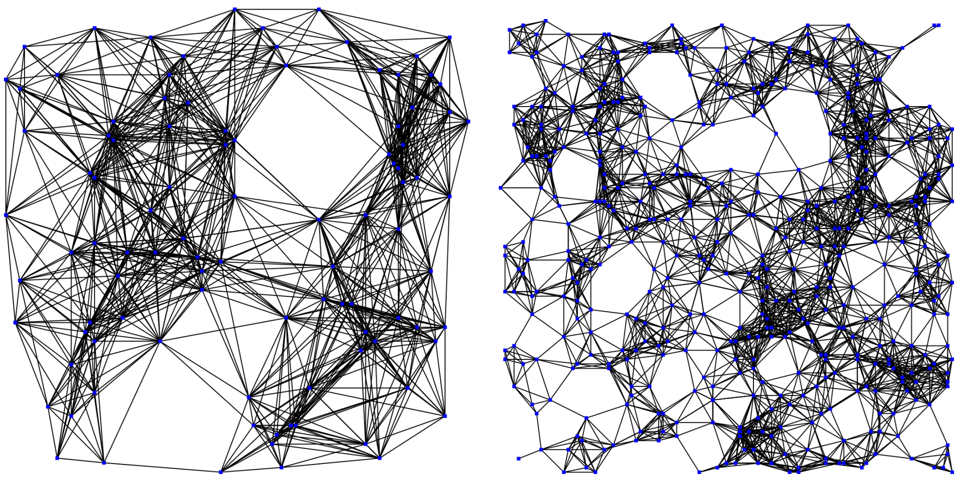

(a) 500 nodes, radius 30% (b) 500 nodes, radius 10%

Figure 2.4.: Unit disk graphs.

2.5.4. Specialized Instances

Also used throughout this thesis are a number of small instances of various graph classes to demonstrate specific abstraction qualities and issues with certain structural traits. All these graphs were produced with a freely available version of the yEd Graph Editing Software[4] released by yWorks.

Binary Tree

A complete binary tree on 1023 nodes, both with a canonical horizontal layout (Figure 2.5(a)) and a force directed organic layout (Figure 2.5(b)). The abstraction should preserve the tree property and also approximate the visual structure of the two layouts as closely as possible.

Polygon

A randomly generated non-intersecting polygon path with 1000 nodes (Figure 2.5(c)). This very basic instance can be expected to be reasonably easy, and gives an indication of how well simple shapes are approximated.

Regular Grid

A regular 30x30 grid (Figure 2.5(d)). Regular grids are instances that are hard for all general abstraction methods examined within the scope of this thesis, although their generalization is largely trivial from a human point of view. A major problem is that any irregularity in the generalized node set causes non-rectangular edge angles or holes in the regular grid structure, which are immediately obvious.

[4] yEd (c) 2011 yWorks (http://www.yworks.com/en/products_yed_about.html)

Triangulation

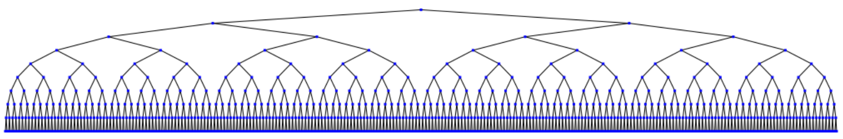
A randomly generated planar embedding of a triangulation of 100 nodes (Figure 2.5(e)), which contains close parallel edges on the outside that are nearly indistinguishable even at high zoom levels. The generalization should be able to discard them while simultaneously preserving the triangular structure.

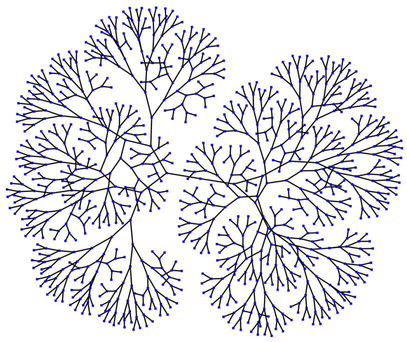
Uniform Star

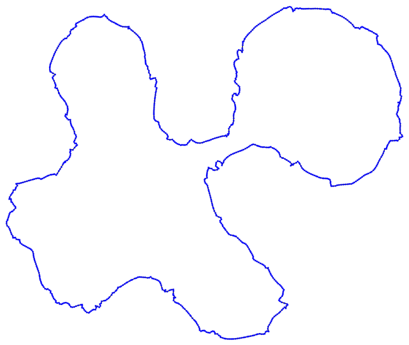
A star with 250 nodes and uniform edge length (Figure 2.5(f)). Another very regular instance, which should be easy, however, as only the node set requires abstraction and ideally no new edges should be inserted.

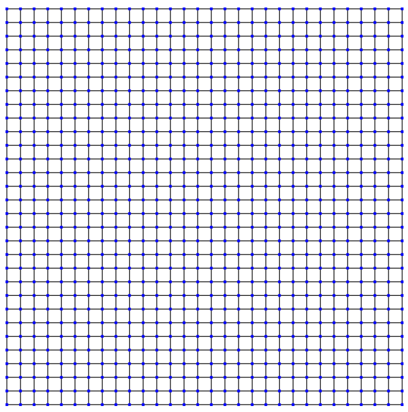
Circular Clique

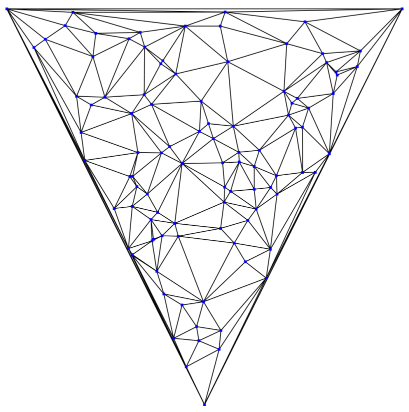
A 100-node clique arranged in a circular fashion (Figure 2.5(g)). Cliques are an interesting instance because the large number of edges generally causes visual cluttering even on zoom levels where the node set is clearly discernible. Therefore, the edge set is the main focus of abstraction, as it needs to be thinned out without sacrificing too much structure.

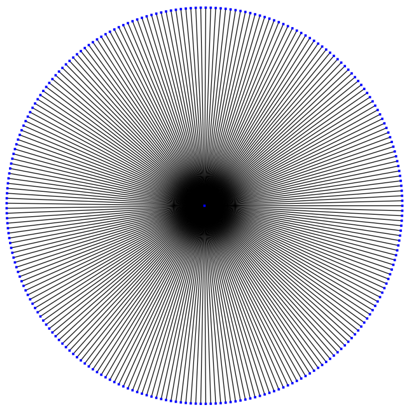

(a) Complete Binary Tree


(b) Complete Binary Tree


(c) Poylgon


(d) Regular Grid


(e) Random Triangulation


(f) Uniform Star

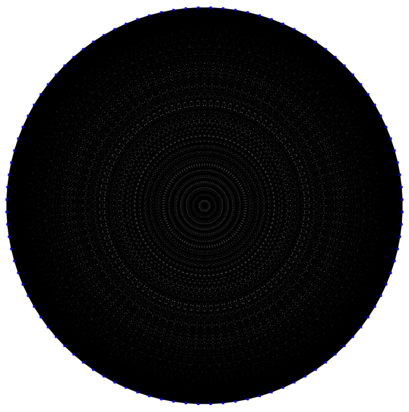

(g) Circular Clique

Figure 2.5.: Special graph class instances.

3. Node Set Generalization

In the following several basic generalization techniques are introduced and evaluated. While the focus lies on finding a good abstraction for a graph's node set, the induced mapping is also discussed for each introduced generalizer.

3.1. Random Sampling

A very straightforward approach with linear running time is to simply pick a subset of nodes to discard at random. Disregarding the inherent issue that any sample can not be guaranteed to be representative of the graph, this approach is still severely lacking. Density can be expected to be approximated reasonably well, but higher abstraction levels are generally meaningless, as very small random samples are not able to capture the graph's structure reliably. Conversely, on low zoom levels visual shortcomings are prohibitive. Extremely short edges as well as close neighbors are prevalent and cause cluttering, as is apparent from Figure 3.1 which shows an example of sampling on the OSM Berlin graph.

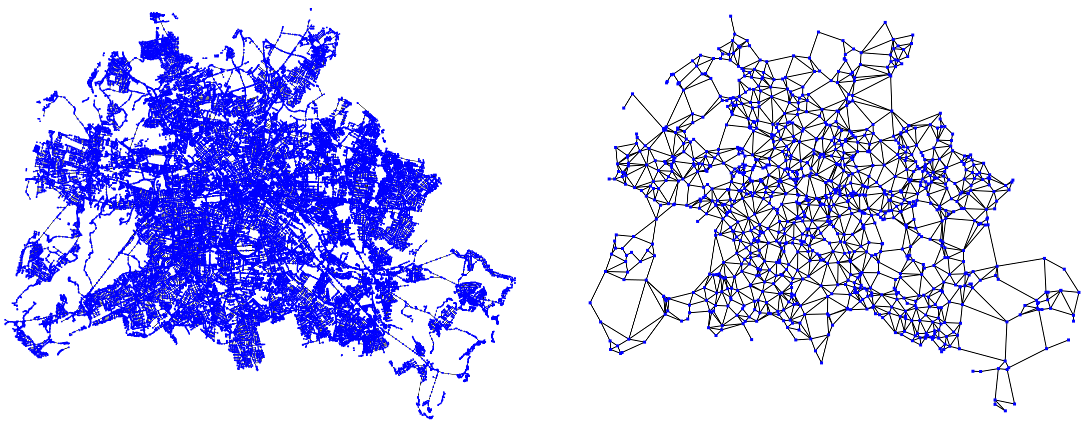

(a) No Generalizer n106675 e124163 (b) Sampling n1111 e2620 l7

Figure 3.1.: Random sampling on the OSM Berlin graph.

3.2. Edge Contraction

In order to solve the problems of edge and node cluttering apparent with random sampling, this method enforces minimum edge length constraints in the generalized graph by running a simple greedy contraction algorithm on the edges. The target minimum edge length and node distance is computed as $\varepsilon_{\text{diff}} = \varepsilon \cdot 2^{\text{zoomLevel}}$, which corresponds to a fixed distance of ε on any given zoom level. An edge is contracted by removing it from the graph and merging its source and target vertices. Pseudo-code for the edge contraction generalizer is given in Algorithm 3.2.1.

Algorithm 3.2.1: Edge Contraction Generalizer

```
Input: graph $G = (V, E)$, distance threshold $\varepsilon_{\text{diff}}$
Output: graph $G' = (V', E') \mid \forall e \in E' : \texttt{Length}(e) \geq \varepsilon_{\text{diff}}$
while $\exists e \in E : \texttt{Length}(e) < \varepsilon_{\text{diff}}$ do
    G ← LimitedEdgeContraction $(G, \varepsilon_{\text{diff}})$;
end
return G;
```

The graph is repeatedly contracted by the limited edge contraction routine (cf. Algorithm 3.2.2), until all remaining edges are longer than $\varepsilon_{\text{diff}}$. This iterative contraction helps to prevent long curved paths from turning into a single long edge, as many consecutive short distance edges are contracted uniformly.

The limited edge contraction routine works as follows. Edges are first sorted by their length in ascending order. As long as an edge of length smaller than $\varepsilon_{\text{diff}}$ remains whose nodes haven't been moved yet, the shortest edge is taken from the edge sequence and its target node is moved onto its source node. All edges except those removed by the contraction routine remain in the graph and have their source and target nodes updated accordingly. Loops and parallel edges are discarded.

Algorithm 3.2.2: Limited Edge Contraction

```
Input: graph $G = (V, E)$, distance threshold $\varepsilon_{\text{diff}}$
Output: graph $G' = (V', E')$
foreach $v \in V$ do Moved($v$) ← false;
while $\exists e \in E : \texttt{Length}(e) < \varepsilon_{\text{diff}}$ do
    e ← edge with smallest length in E;
    if not Moved(Target($e$)) and not Moved(Source($e$)) then
        Source($e$) ← Target($e$);
        Moved(Target($e$)) ← true;
        Moved(Source($e$)) ← true;
    end
    remove e from E;
end
return G;
```

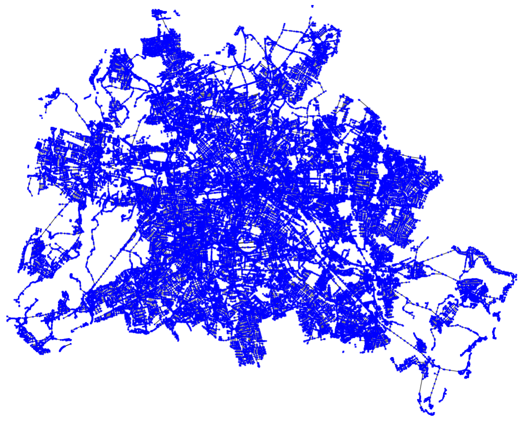
Theorem 3.1 *The running time of Edge Contraction is in $\mathcal{O}((|V| - |V'| + 1) \cdot |E| \log |E|)$.*

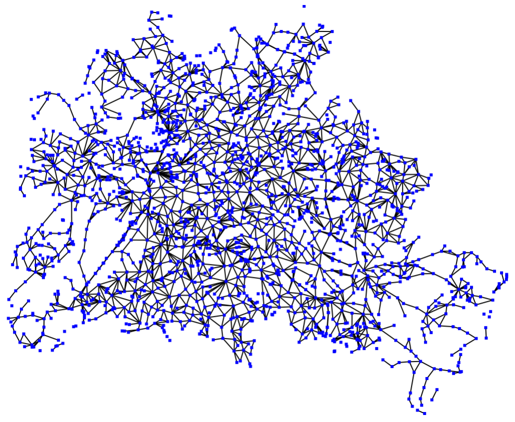
Proof

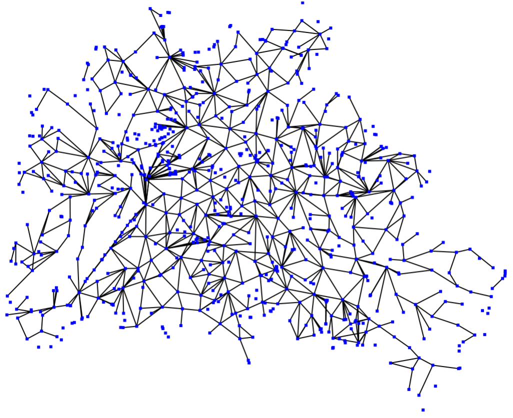
During each iteration of the limited edge contraction routine at least one node is removed, which limits the number of iterations to at most $|V| - |V'|$. Each iteration requires sorting and scanning the edges which amounts to $\mathcal{O}(|E| \log |E|)$, and the graph has to be remapped afterwards. This can be done scanning the edge set and replacing nodes accordingly, which takes $\mathcal{O}(|E|)$ time and allows for contracting a single edge in $\mathcal{O}(1)$. After the algorithm finishes, all edges are guaranteed to have length at least $\varepsilon_{\text{diff}}$.

$\square$

Note that running time is considerably faster for sparse graphs in practice.


(a) No Generalizer n106675 e124163


(b) EC n1964 e2827 l16


(c) EC n771 e875 l17

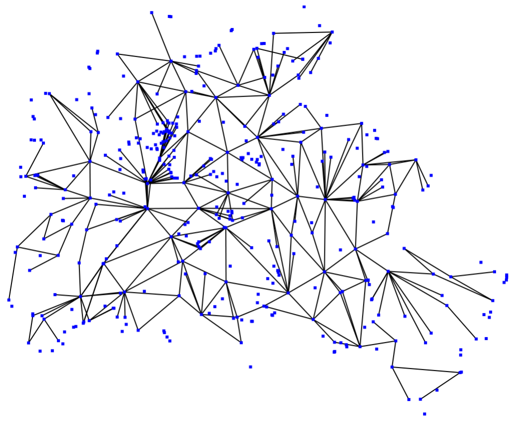
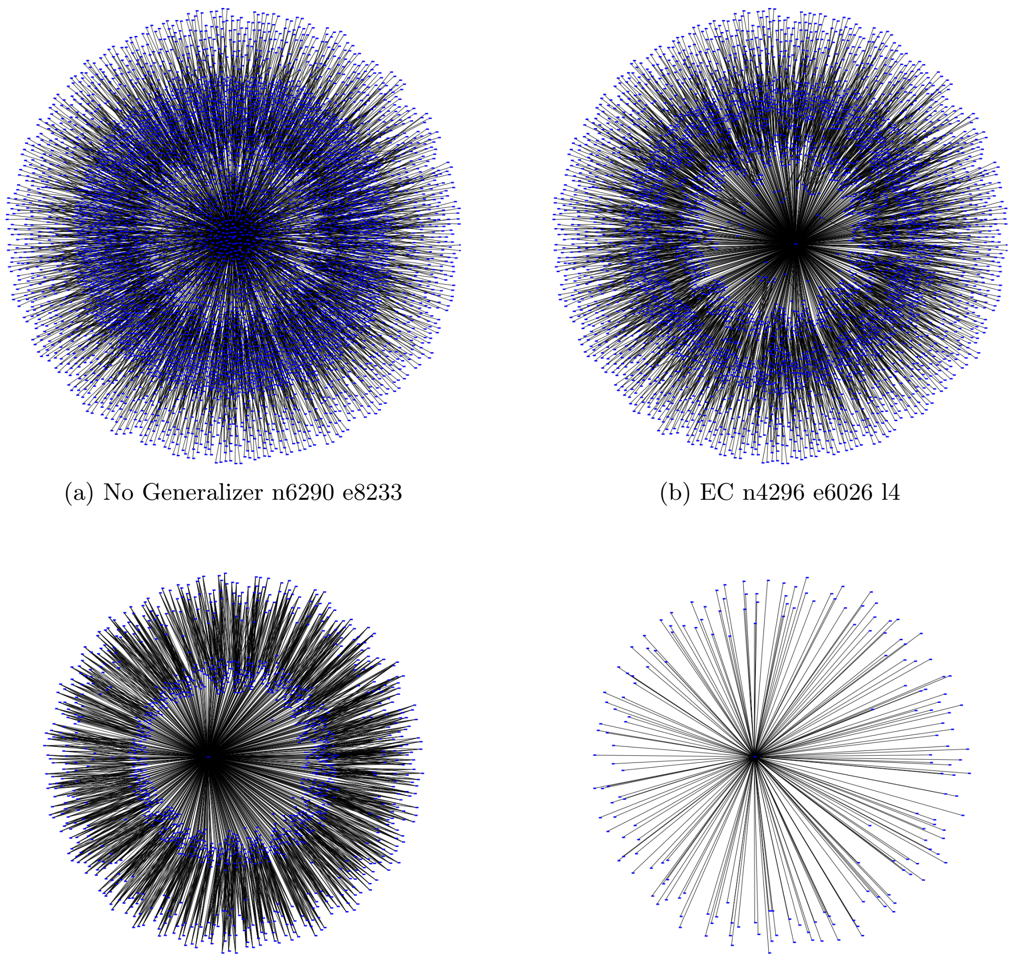

(d) EC n404 e301 l18

Figure 3.2.: Edge Contraction on the OSM Berlin graph.

Figure 3.2 shows the results of edge contraction on the OSM Berlin graph at various zoom levels. Unlike random sampling, edge contraction actually reveals that the graph is disconnected, as for each connected component only one solitary node remains at high zoom levels. While the abstraction generally captures structure better than random sampling, there are significant drawbacks. One important issue is that the heuristic still cannot reliably ensure minimum node distances in the generalized graph. Adjacent nodes always have distance at least $\varepsilon_{\text{diff}}$, but geometrically close nodes that are no immediate neighbors can remain in their place for a long time, if the heuristic opts to contract edges between common neighbors instead or all their incident edges are already very long to begin with.

While the minimum edge length constraint has improved edge cluttering in one regard, the induced mapping still has issues with long parallel edges and high degree nodes in particular, where single edges are impossible to discern. This is especially apparent on star graphs and similar structures, such as the Cep1 graph (cf. Figure 3.3).



(a) No Generalizer n6290 e8233

(b) EC n4296 e6026 l4

(c) EC n1251 e2805 l5

(d) EC n185 e184 l6

Figure 3.3.: Edge Contraction on the Cep1 graph.

Since the edge contraction heuristic depends solely on edge length, uniform or near-uniform edge length is a special worst case input for which the order of contraction is random. This is, for example, the primary cause for the failure of edge contraction on regular Grids (cf. Figure 3.4), since central nodes move in an arbitrary direction and the generalization can not be guaranteed to maintain the original graph's regularity, which is its most distinctive visual feature. Note that our implementation is deterministic and it is actually the input order of nodes which determines the results in this case, which explains the somewhat regular shift to the left. With a truly random input node order better results can be expected for this instance.

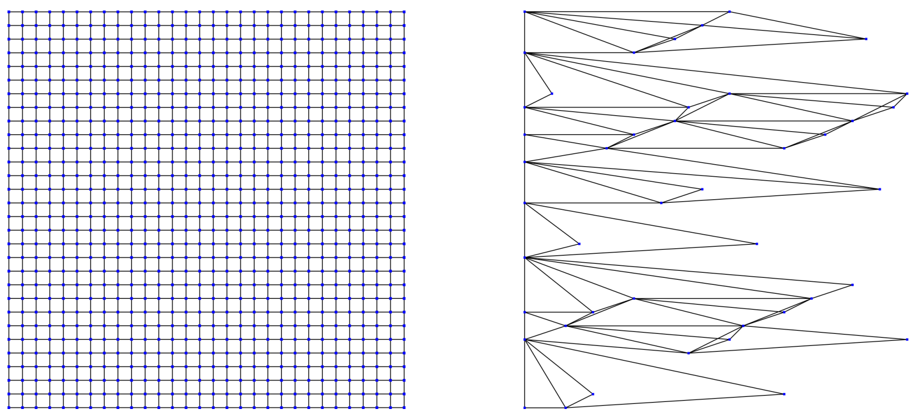

(a) No Generalizer n900 e1740 (b) EC n44 e94 l1

Figure 3.4.: Edge Contraction on the regular 30x30 grid.

A worst case example for uniform edge length is the regular star depicted in Figure 3.5, however, where the heuristic only allows for a single degree of abstraction as the entire star is contracted into one node in a single iteration.

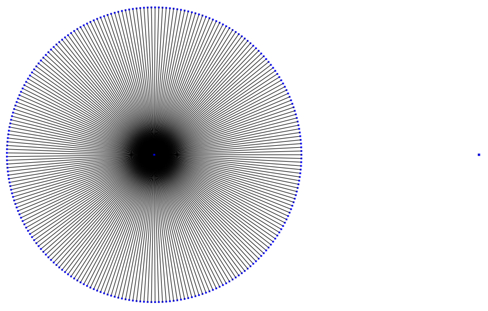

(a) No Generalizer n250 e249 (b) EC n1 e0 l1

Figure 3.5.: Edge Contraction on the regular star with uniform edge length.

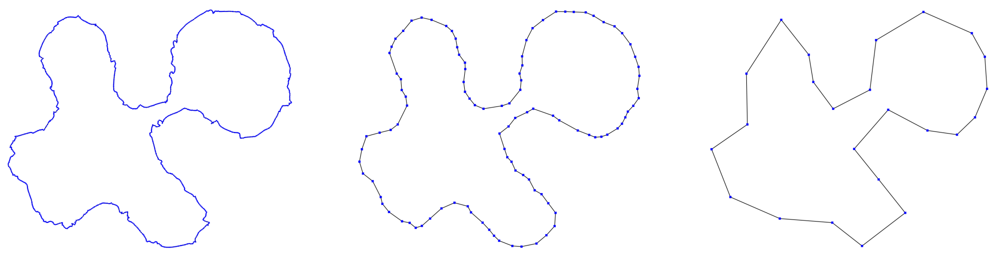

(a) No Generalizer n1000 e1000 (b) EC n104 e104 l4 (c) EC n24 e24 l6

Figure 3.6.: Edge Contraction on the 1000-node polygon.

Edge contraction with induced mapping works well with some basic graph structures such as simple polygons (cf. Figure 3.6) and trees (cf. the complete binary tree of depth six in Figure 3.7). Due to the connectivity-preserving nature of the contraction the result is guaranteed to be a tree as well, independent of the graph's layout. In that regard it has an advantage over all other abstraction methods examined within the scope of this thesis, which in certain cases may allow a generalized tree to be disconnected or contain circles, respectively. However, node degree is not preserved (cf. the complete binary tree of depth six in Figure 3.8).

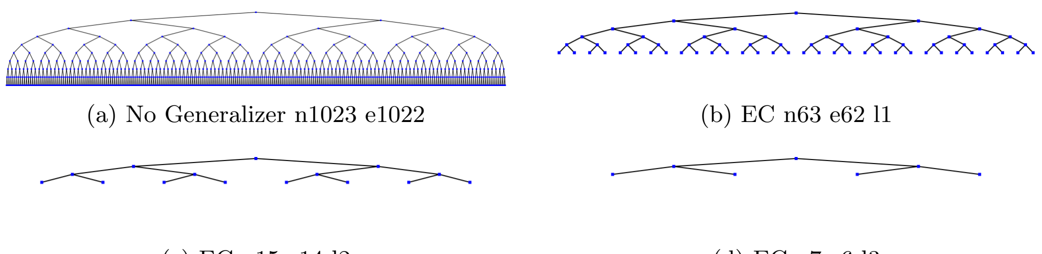

(a) No Generalizer n1023 e1022 (b) EC n63 e62 l1 (c) EC n15 e14 l2 (d) EC n7 e6 l3

Figure 3.7.: Edge Contraction on the complete binary tree with canonic layout.

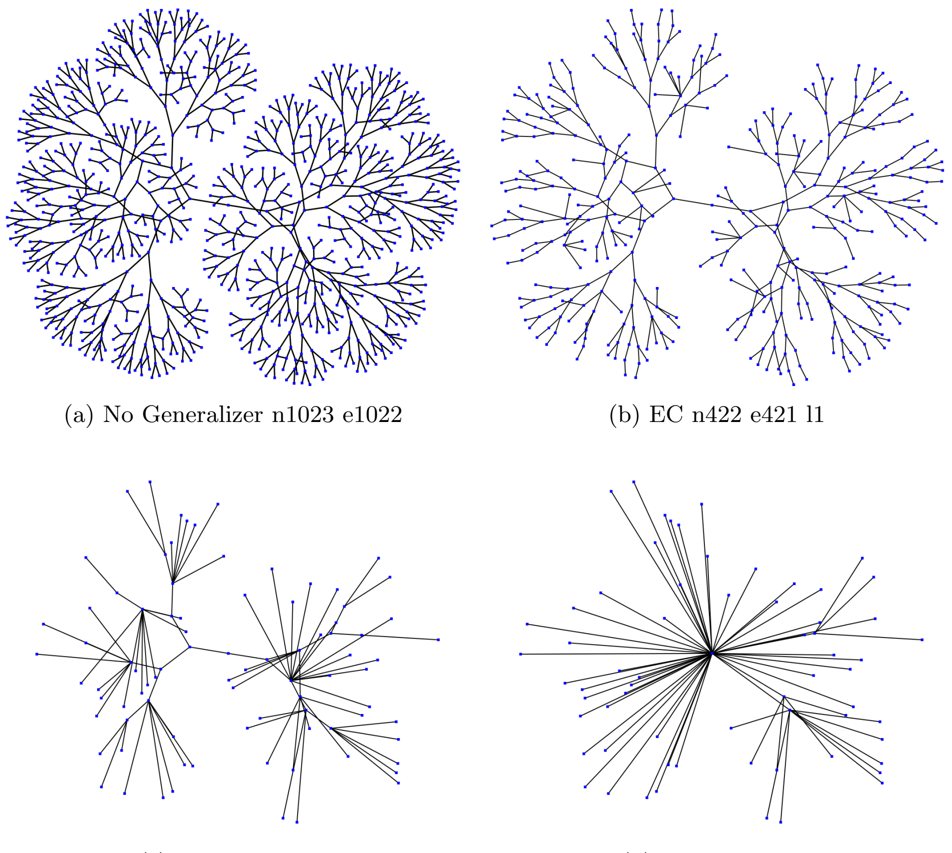
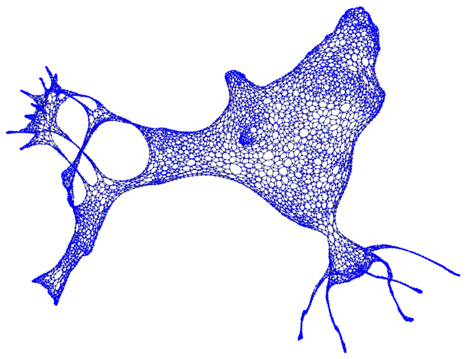
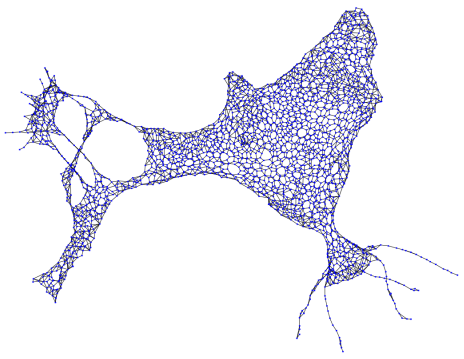
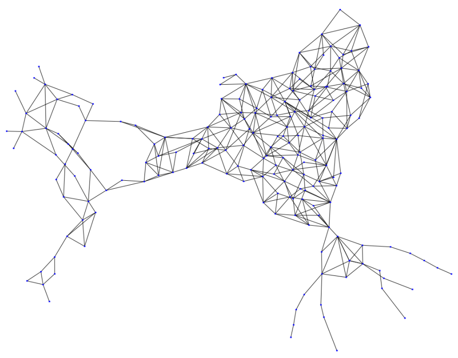

(a) No Generalizer n1023 e1022 (b) EC n422 e421 l1 (c) EC n86 e85 l2 (d) EC n65 e64 l3

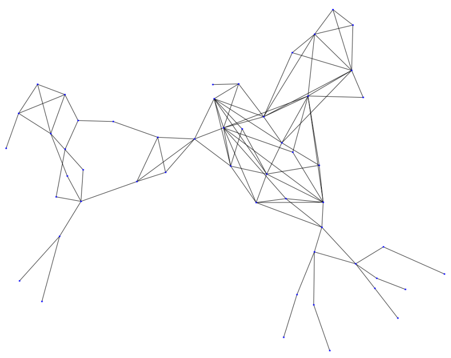
Figure 3.8.: Edge Contraction on the complete binary tree with organic layout.

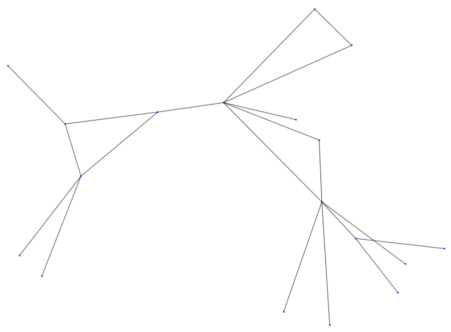
Figure 3.9 shows edge contraction on the Commanche graph, where the induced mapping manages to capture the three-dimensional structure quite well.


(a) No Generalizer n7920 e11880


(b) EC n2914 e5145 l5


(c) EC n190 e432 l7


(d) EC n53 e99 l8


(e) EC n18 e20 l9

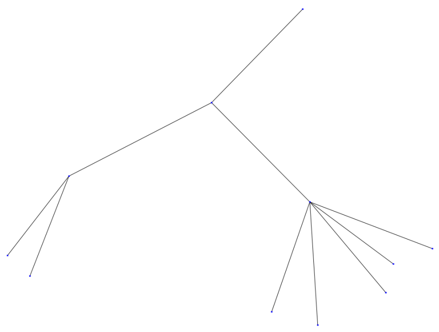

(f) EC n11 e10 l10

Figure 3.9.: Edge Contraction on the commanche sparse matrix with sfdp layout.

3.3. Node Contraction

This approach concentrates on purely visual abstraction. It borrows its name and the central idea from a method used in route planning by speed-up techniques, such as Contraction Hierarchies [GSSD08], where nodes are replaced by a set of edges while preserving shortest path lengths. Here we strive to maintain as much visual information as possible by replacing a node and its adjacent edges depending on their relative angles when contracting. An example for the contraction is given in Figure 3.10.

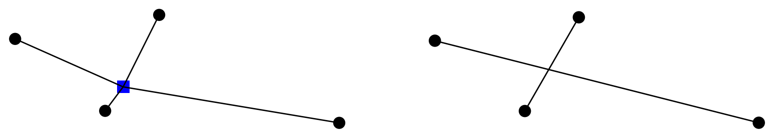


Figure 3.10.: Node contraction Example. The central node is contracted.

Algorithm 3.3.1 describes the edge insertion criterium in pseudo-code. For each pair of neighbors u_1, u_2 of the node v to be contracted, an edge is inserted if the angle $\angle(u_1, v, u_2)$ lies within a specified range $[a_{\min}, a_{\max}]$. In order to avoid severe disconnectivity in the generalization, we define a node to be contractable if for each of its neighbors at least one incident edge would be added upon contraction. Restricting the generalization to contractable nodes indicates the degree of abstraction possible without notable loss of visual structure.

Algorithm 3.3.1: Insert Edges

Input: graph $G = (V, E)$, node $v \in V$, subset $U \subseteq V$, parameters $a_{\min}, a_{\max}$
Output: graph $G = (V, E')$

```
1 foreach distinct pair $u_1, u_2 \in U$ do
2     if Angle($u_1, v, u_2$) $\in [a_{\min}, a_{\max}]$ then
3         add $\{u_1, u_2\}$ to $E$;
4     end
5 end
6 return $G$;
```

The node contraction generalization process is described in Algorithm 3.3.2. Nodes are ordered by a tuple $(\delta_c(v), \delta_i(v), \delta_\ell(v))$, where $\delta_c(v)$ indicates whether the node v is contractable or not and $\delta_\ell(v)$ denotes the length of its shortest adjacent edge. The third parameter $\delta_i(v)$ starts out as zero for all nodes, but whenever a node v is contracted it is set to $\max(\delta_i(u), \delta_i(v) + 1)$ for each neighbor u. Factoring in this additive term helps to make the contraction more uniform. The algorithm contracts nodes in order and inserts edges as described above. Note that the node order needs to be updated dynamically, which is expensive. We only execute updates lazily, i.e., when inspecting the next node to be contracted. While lazy updates are much faster in praxis, they are only correct in case a node's priority decreases. However, the only parameter change that can actually lead to a priority increase is when the shortest adjacent edge length $\delta_\ell(v)$ decreases, which can be expected to happen fairly rarely.

Algorithm 3.3.2: Node Contraction Generalizer

Input: graph $G = (V, E)$, factor $\varepsilon_{\text{diff}}$
Output: graph $G' = (V', E')$

```
foreach $v \in V$ do
    $\delta_c(v) \leftarrow$ IsContractable($v$);
    $\delta_i(v) \leftarrow 0$;
    $\delta_\ell(v) \leftarrow$ MinAdjacentEdgeLength($v$);
end
order $V$ by $(\delta_c(v), \delta_i(v), \delta_\ell(v))$;
foreach $v \in V$ in order do
    remove $v$ from $V$;
    if LazyOrderUpdate($v$) then
        add $v$ to $V$;
        continue;
    else
        if $\delta_\ell(v) > \varepsilon_{\text{diff}}$ then continue;
        $U \leftarrow$ neighbors of $v$ in $G$;
        InsertEdges($G$, $v$, $U$);
        foreach $u \in U$ do
            $\delta_i(u) \leftarrow \max(\delta_i(u), \delta_i(v) + 1)$;
            remove $\{v, u\}$ from $E$;
        end
    end
end
return $G$;
```

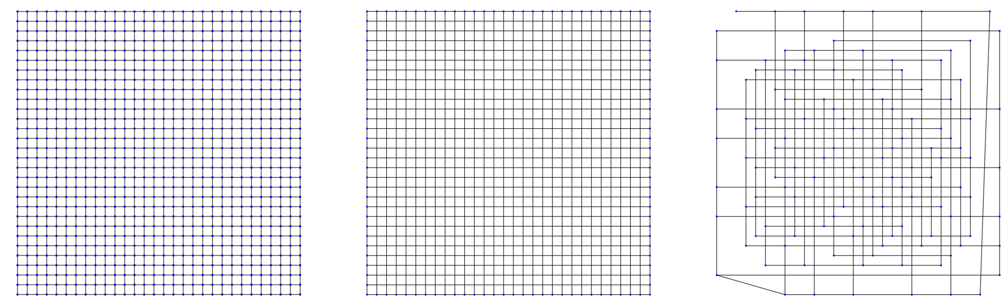
Theorem 3.2 *Node Contraction has a worst case running time of* $\mathcal{O}((|V| - |V'| + 1) \cdot |V|^2)$.

Proof

Contracting a single node takes time $\mathcal{O}(\triangle^2)$, where $\triangle$ denotes the node's degree, as for each pair of neighbors we have to check whether an edge should be inserted. The degree $\triangle$ is bounded only by $|V|$, since contraction usually increases the degree of neighboring nodes. Initializing the priority queue requires simulating the contraction for each node, which takes time $\mathcal{O}(|V| \cdot \triangle_{\max}^2)$. Updating is the most expensive part. Each node's priority can change once per contracted neighbor. In the worst case, all nodes have to be updated after a single contraction. Since we perform exactly $V - V'$ contractions, this amounts to $\mathcal{O}((|V| - |V'|)(|V|^2 + \log |V|))$. Overall, the worst case running time is therefore in $\mathcal{O}((|V| - |V'| + 1) \cdot |V|^2)$.

□

With the restricted variant of node contraction, which operates on contractable nodes only, a good low level visual abstraction is possible for general graphs, but a large non-contractable core usually remains. A prime example is the regular 30x30 grid as shown in Figure 3.11(b), where a reduction of 80% the number of both nodes and edges is possible without changing the visual structure at all. As can be seen from Figure 3.11(c) this is not the case for the unrestricted variant. The angle range is set to $[135, 225]$ in both cases.

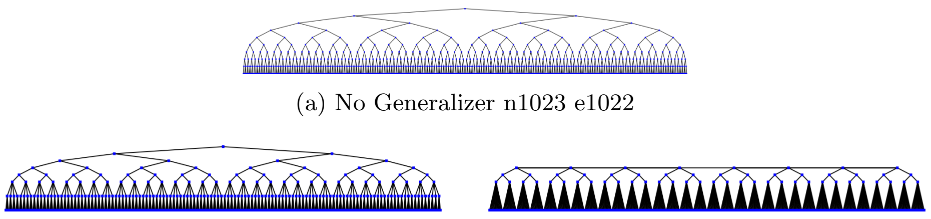

(a) No Generalizer n900 e1740

(b) NC Restricted n116 e172 l5

(c) NC n119 e183 l3

Figure 3.11.: Node Contraction on the regular 30x30 grid.

However, there are severe issues with edge cluttering, as can be seen in Figure 3.12, which shows node contraction on the complete binary tree with canonic layout.

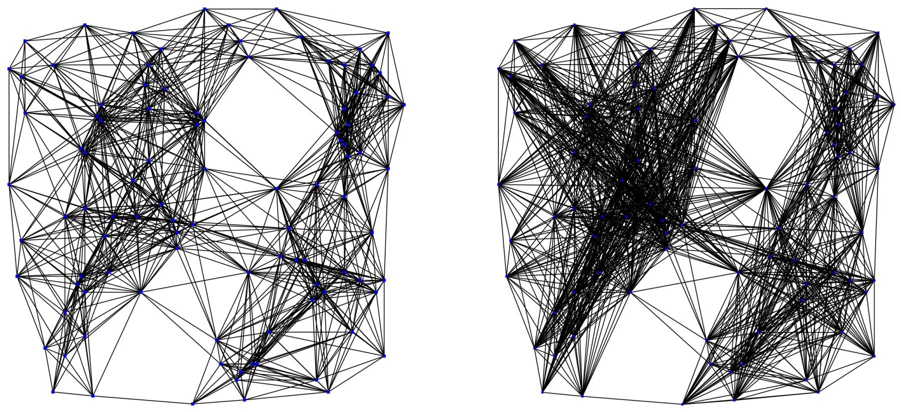

(a) No Generalizer n1023 e1022

(b) NC Restricted n703 e702 l1

(c) NC Restricted n568 e567 l4

Figure 3.12.: Node Contraction on the complete binary tree with canonic layout.

Another heavy drawback is that commonly too many edges are inserted and the node-to-edge ratio is hard to regulate, as the angle range for edge insertion would have to be adapted dynamically. This is apparent from the results on the UD30 unit disk graph displayed in Figure 3.13, where the same range used above, $[135, 225]$, actually allows for more edges than were present in the original graph at low abstraction levels.


(a) No Generalizer n100 e1012

(b) NC n87 e1219 l2

Figure 3.13.: Node contraction on the UD30 unit disk graph.

3.4. Neighborhood Contraction

One of the most obvious shortcomings of the previous approaches was that they failed to reduce node cluttering sufficiently, even with the total number of nodes greatly reduced. The neighborhood contraction generalizer remedies this by extending the restrictions on edge length to a fixed minimum distance of $\varepsilon_{\text{diff}} = \varepsilon \cdot 2^{\text{zoomLevel}}$ between any two nodes at a given zoom level, which leads to the following general layout problem.

Problem 3.1 (Minimum Distance Vertex Layout) *Given a finite set $V \subset \mathbb{R}^2$ of vertices and two thresholds $\varepsilon_{\text{move}} > 0$ and $\varepsilon_{\text{diff}} > 0$, is it possible to move the points of V in the plane within a radius of $\varepsilon_{\text{move}}$, such that all vertices $v_1, v_2 \in V$ either satisfy $\|v_1 - v_2\| \geq \varepsilon_{\text{diff}}$ or coincide?*

Note that if the parameters are restricted to $\varepsilon_{\text{diff}} \leq \varepsilon_{\text{move}}$ this problem always has a trivial solution, i.e., as long as points v_1 and v_2 with $\|v_1 - v_2\| < \varepsilon_{\text{diff}}$ exist in the graph collapse the $\varepsilon_{\text{move}}$-neighborhood of either point onto itself.

Theorem 3.3 *The minimum distance vertex layout problem is NP-hard.*

Proof

The proof follows by reduction from the NP-complete planar 3SAT problem introduced in the Preliminaries Section 2.3. We construct a planar variable-clause graph corresponding to a 3SAT formula φ as illustrated in Figure 3.14, for which the vertex layout problem is solvable if, and only if, the formula is satisfiable.

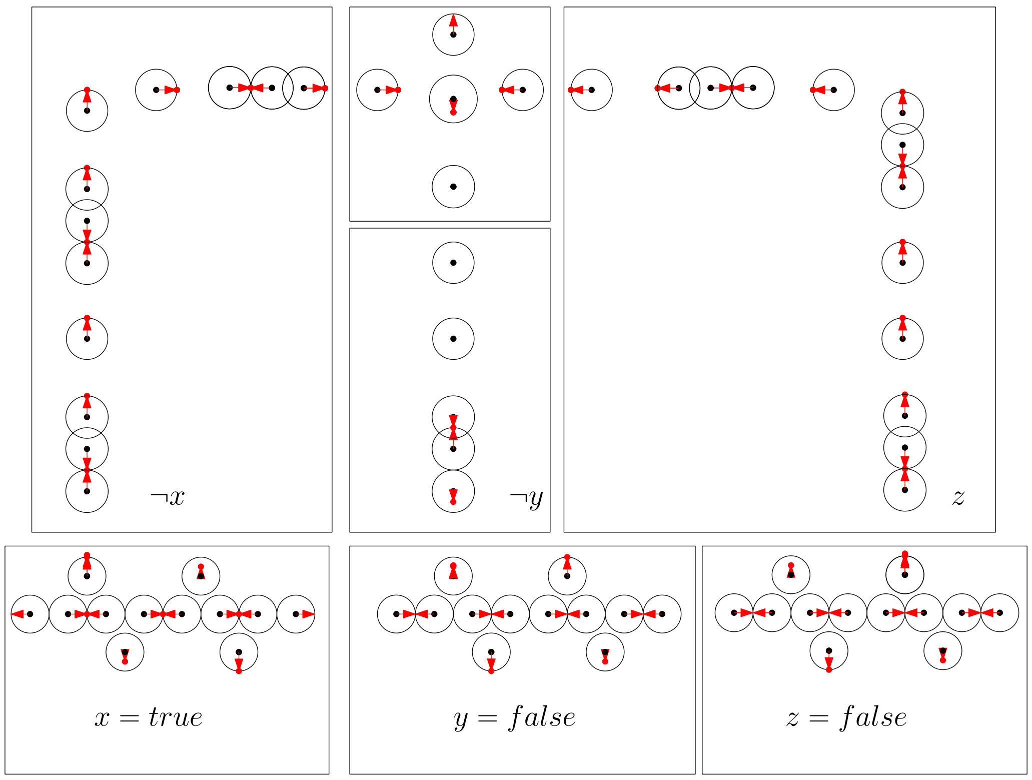


Figure 3.14.: Satisfying assignment for the formula $\varphi = \neg x \vee \neg y \vee z$

Variable gadgets.

A variable x is represented by a string of $10 \cdot n_x + 2$ points, where n_x is the number of occurrences of x in the formula. For every such occurrence, ten points are positioned in the plane as shown in Figure 3.15. Six of them are arranged on a straight horizontal line parallel to the x-axis, at distances exactly $2 \cdot \varepsilon_{\text{move}}$, while the four remaining ones are positioned above and below the line, respectively, at a distance of $\varepsilon_{\text{diff}}$. Two extra points are present per variable, the left- and the rightmost point.

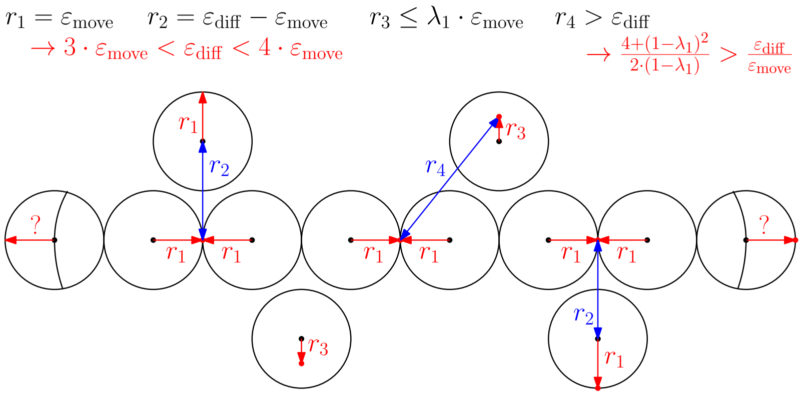
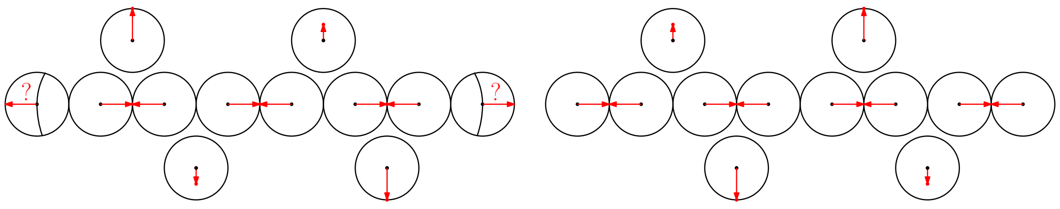

$r_1 = \varepsilon_{\text{move}}$
$r_2 = \varepsilon_{\text{diff}} - \varepsilon_{\text{move}}$
$r_3 \leq \lambda_1 \cdot \varepsilon_{\text{move}}$
$r_4 > \varepsilon_{\text{diff}}$
$\rightarrow 3 \cdot \varepsilon_{\text{move}} < \varepsilon_{\text{diff}} < 4 \cdot \varepsilon_{\text{move}}$
$\rightarrow \frac{4+(1-\lambda_1)^2}{2\cdot(1-\lambda_1)} > \frac{\varepsilon_{\text{diff}}}{\varepsilon_{\text{move}}}$
r_1
r_2
r_3
r_4
?

Figure 3.15.: Basic variable gadget layout and constraints.

By choosing $3 \cdot \varepsilon_{\text{move}} < \varepsilon_{\text{diff}} < 4 \cdot \varepsilon_{\text{move}}$ we ensure that are only two distinct states are possible for a variable gadget, depending on the direction in which the leftmost point has to be shifted to satisfy the constraints. If it moves to the left the resulting configuration is assigned the 'true' state (Figure 3.16(a)), otherwise the gadget assumes the 'false' state (Figure 3.16(b)).

The 'true' configuration forces both the left point on the top line and the right point on the bottom line to move by $\varepsilon_{\text{move}}$ parallel to the y-axis, fixing their position. The other two points on the top and bottom line have to shift in the same direction, albeit only by an amount of less than $\lambda_1 \cdot \varepsilon_{\text{move}}$, where λ_1 is configurable but enforces constraints on the quotient $\frac{\varepsilon_{\text{move}}}{\varepsilon_{\text{diff}}}$. Choosing $\lambda_1 = \frac{1}{2} \cdot \varepsilon_{\text{move}}$ requires $\frac{4}{17} < \frac{\varepsilon_{\text{move}}}{\varepsilon_{\text{diff}}}$, which together with the upper bound $\frac{\varepsilon_{\text{move}}}{\varepsilon_{\text{diff}}} < \frac{1}{3}$ from above results in $\frac{12}{51} \cdot \varepsilon_{\text{diff}} < \varepsilon_{\text{move}} < \frac{17}{51} \cdot \varepsilon_{\text{diff}}$. For the 'false' state the arrangement of the four bottom and top line points is exactly reversed, and the same calculations apply.

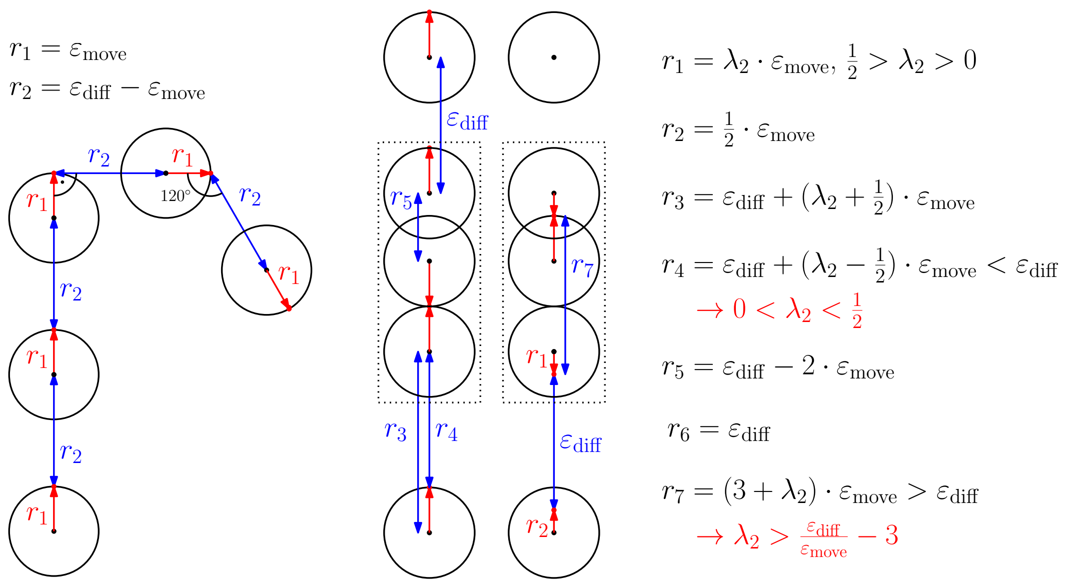


(a) True State (b) False State

Figure 3.16.: True and false configuration for the basic variable gadget.

Literal gadgets.

Literal gadgets consist of an alternating chain of node strings and buffer structures. The strings are simple sequences of co-linear points placed at intervals of $\varepsilon_{\text{diff}}$ which translate a shift of $\varepsilon_{\text{move}}$ as depicted in Figure3.17(a). Buffer structures consist of three nodes that are also aligned co-linearly with distances $\varepsilon_{\text{diff}} - 2 \cdot \varepsilon_{\text{move}}$ and $2 \cdot \varepsilon_{\text{move}}$ between them, respectively (see Figure 3.17(b)). These buffers connect to a target node at distance $\varepsilon_{\text{diff}}$ and a source node at $\varepsilon_{\text{diff}} + (\lambda_2 + \frac{1}{2}) \cdot \varepsilon_{\text{move}}$. The parameter λ_2 is configurable, but limited by $\frac{\varepsilon_{\text{move}}}{\varepsilon_{\text{diff}}} - 3 < \lambda_2 < \frac{1}{2}$. A ratio of $\frac{\varepsilon_{\text{move}}}{\varepsilon_{\text{diff}}} = \frac{15}{51}$, which is a valid choice according to variable gadget constraints, allows for any $\lambda_2 \in (\frac{2}{5}, \frac{1}{2})$.


$r_1 = \varepsilon_{\text{move}}$
$r_2 = \varepsilon_{\text{diff}} - \varepsilon_{\text{move}}$
r_1
r_2
120°
$\varepsilon_{\text{diff}}$
r_3
r_4
r_5
r_7
$r_1 = \lambda_2 \cdot \varepsilon_{\text{move}}, \frac{1}{2} > \lambda_2 > 0$
$r_2 = \frac{1}{2} \cdot \varepsilon_{\text{move}}$
$r_3 = \varepsilon_{\text{diff}} + (\lambda_2 + \frac{1}{2}) \cdot \varepsilon_{\text{move}}$
$r_4 = \varepsilon_{\text{diff}} + (\lambda_2 - \frac{1}{2}) \cdot \varepsilon_{\text{move}} < \varepsilon_{\text{diff}}$
$\rightarrow 0 < \lambda_2 < \frac{1}{2}$
$r_5 = \varepsilon_{\text{diff}} - 2 \cdot \varepsilon_{\text{move}}$
$r_6 = \varepsilon_{\text{diff}}$
$r_7 = (3 + \lambda_2) \cdot \varepsilon_{\text{move}} > \varepsilon_{\text{diff}}$
$\rightarrow \lambda_2 > \frac{\varepsilon_{\text{diff}}}{\varepsilon_{\text{move}}} - 3$

(a) String (b) Buffer

Figure 3.17.: String and buffer segments of literal gadgets.

Each literal requires at least one such buffer structure to connect to a variable gadget. If this variable is negated in the clause, the literal is attached to either the upper left or lower right point of the gadget. Otherwise, the connection is made at the upper right or lower left point.

Clause gadget.

Clause gadgets are composed of five points arranged as shown in Figure 3.18. Three points, aligned as the corners of an inverted pyramid, connect to the literal gadget. A fourth central point is situated on the pyramid base with equal distance $\varepsilon_{\text{diff}} + (1 - \lambda_3) \cdot \varepsilon_{\text{move}}$ to all three corners, and distance $\varepsilon_{\text{diff}} - \varepsilon_{\text{move}}$ to another point above it. If a variable gadget assumes a wrong state, its corresponding corner vertex is pushed towards the center point by $\varepsilon_{\text{move}}$ as the literal gadget transfers pressure from the variable gadget. If this happens for all three variables, the distances to the center point shrink by $\varepsilon_{\text{move}}$ and since it can not escape upwards the layout problem is not solvable any more. In case only two or less variables are assigned a wrong value, however, the central point can sidestep by $(1 - \lambda_3) \cdot \varepsilon_{\text{move}}$ in the direction of whichever literal connection does not push inwards. The ratio $\frac{\varepsilon_{\text{move}}}{\varepsilon_{\text{diff}}} = \frac{15}{51}$ allows for $\lambda_3 = 0.1$ which meets the constraints.

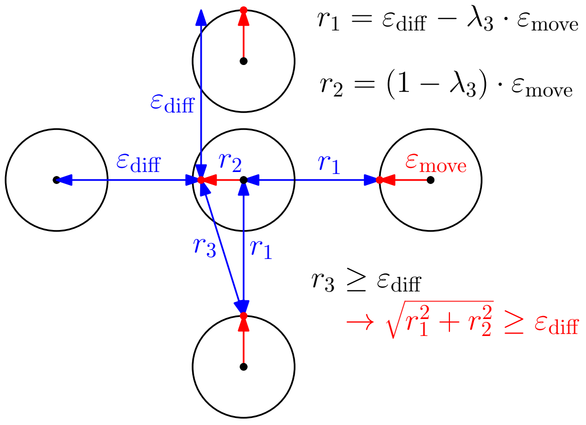

$r_1 = \varepsilon_{\text{diff}} - \lambda_3 \cdot \varepsilon_{\text{move}}$
$r_2 = (1 - \lambda_3) \cdot \varepsilon_{\text{move}}$
$\varepsilon_{\text{diff}}$
$\varepsilon_{\text{diff}}$
r_2
r_1
$\varepsilon_{\text{move}}$
r_3
r_1
$r_3 \geq \varepsilon_{\text{diff}}$
$\rightarrow \sqrt{r_1^2 + r_2^2} \geq \varepsilon_{\text{diff}}$

Figure 3.18.: Clause gadget layout and constraints.

The layout problem for this variable-clause graph is therefore solvable for any variable assignment that satisfies the formula φ. Conversely, if the assignment evaluates to false for any clause, no valid layout can be found for the corresponding gadget. This graph can be constructed in polynomial time.

□

While the general problem is NP-hard, if the generalized node set is restricted to a subset of the original graph's nodes the special case $\varepsilon_{\text{diff}} \leq \varepsilon_{\text{move}}$ applies, and the neighborhood contraction algorithm suggested in Theorem 3.1 can be used for generalization. Algorithm 3.4.1 describes the procedure in pseudo-code. Nodes are sorted according to a fixed order, e.g., the number of nodes in their $\varepsilon_{\text{diff}}$ neighborhood. The generalizer then iterates over all nodes in this order and collapses all unmoved neighbors remaining in the current node's $\varepsilon_{\text{diff}}$ radius onto it, if the node itself has not been moved yet. Loops and parallel edges are removed from the output graph and the source and target nodes for all other edges are updated as necessary.

Theorem 3.4 *Neighborhood Contraction (Algorithm 3.4.1) can be implemented to run in* $\mathcal{O}(|V| \cdot (\sqrt{|V|} + k_{\max}))$ *time, where* $k_{\max}$ *denotes the maximum* $\varepsilon_{\text{diff}}$*-neighborhood size.*

Proof

Using a KD-Tree for maintaining the point set, a single neighborhood query takes time $\mathcal{O}(\sqrt{|V|} + k_{\max})$. Computing the node order requires a neighborhood query for each node, which raises total running time to $\mathcal{O}(|V| \cdot (\sqrt{|V|} + k_{\max}))$. This dominates the effort of $\mathcal{O}(|V| \cdot \log(|V|))$ necessary for sorting the nodes as well as the actual neighborhood contraction itself.

□

Note that the node order has considerable impact on the outcome. Two straightforward approaches have been tested with varying effects on the generalized graph.

- **by ascending $\varepsilon_{\text{diff}}$ neighbourhood size**
  often better to keep the graph outline intact
- **by descending $\varepsilon_{\text{diff}}$ neighbourhood size**
  tends to keep the total number of nodes small

Algorithm 3.4.1: Neighbourhood Contraction Generalizer

```
Input: graph $G = (V, E)$, factor $\varepsilon_{\text{diff}}$
Output: graph $G' = (V', E') \mid \forall v_1, v_2 \in V' : \texttt{Distance}(v_1, v_2) \geq \varepsilon_{\text{diff}}$
1 $V \leftarrow$ SortNodes (V);
2 while not Empty($V$) do
3     $v \leftarrow$ NextNode (V);
4     forall the nodes $u$ within radius $\varepsilon_{\text{diff}}$ of $v$ do
5         u ← v;
6     end
7 end
8 return $G$;
```

Since neighborhood contraction relies on the size of the nodes' ε neighborhoods to determine which ones are moved first and ultimately remain in the graph, this heuristic is controlled to a great extent by the graph's layout and only secondarily by graph class characteristics. Also, the focus lies more on the generalized node set, since an edge mapping does not arise naturally as with the edge and node contraction approach. Figure 3.19 shows the results on the OSM Berlin graph. The main advantage of neighborhood contraction is that it guarantees minimum node distances in the generalized graph, which solves one of the main issues of edge contraction.

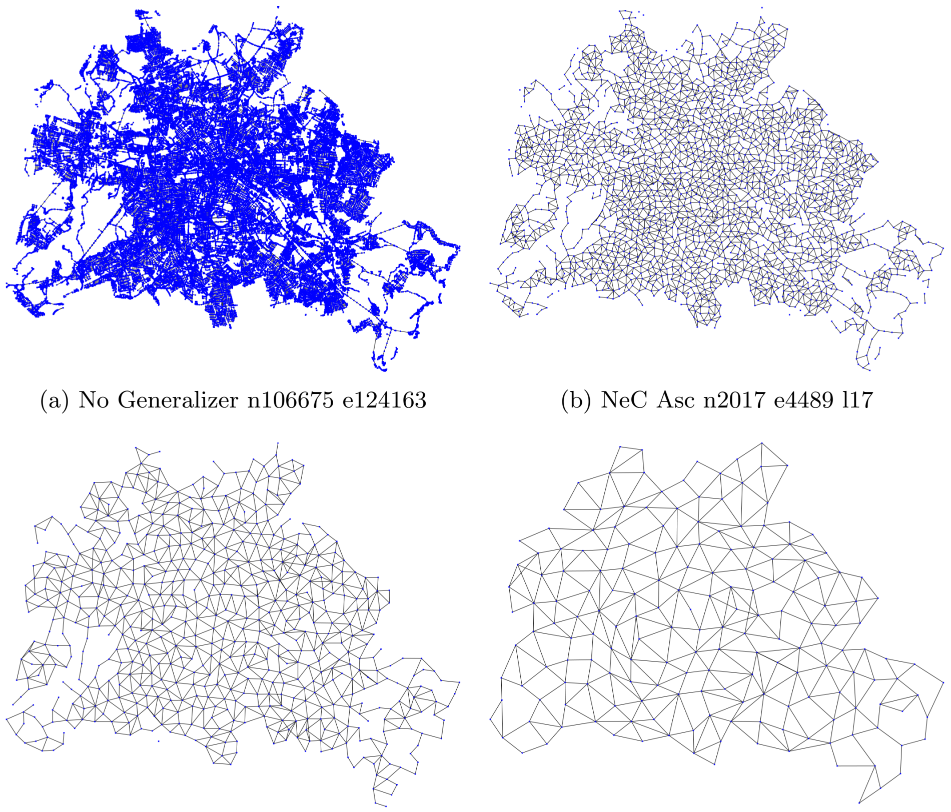

(a) No Generalizer n106675 e124163
(b) NeC Asc n2017 e4489 l17
(c) NeC Asc n617 e1440 l18
(d) NeC Asc n182 e441 l19

Figure 3.19.: Neighborhood Contraction on the OSM Berlin graph.

While the minimum node distance constraint now guarantees that nodes are easily discernible even at high zoom levels, the induced mapping is more problematic as edge cluttering still occurs and it is usually not representative of the original structure, regardless of the chosen node order. Most of the original structure is lost, and the generalized graph turns out nearly triangulated.

Another obvious shortcoming is that the original graph's node density is not respected by the generalizer, whose node density distribution tends to be almost uniform. Algorithm 3.4.2 gives pseudo-code for the adjusted heuristic, which adapts the attraction radius of each node according to its neighborhood size. Nodes whose $2 \cdot \varepsilon_{\text{diff}}$ neighborhood is very densely populated retain the original neighborhood radius $\varepsilon_{\text{diff}}$, while those with a very sparse neighborhood are assigned a radius of up to $2 \cdot \varepsilon_{\text{diff}}$. Figure 3.20 shows the results on the OSM Berlin graph.

Algorithm 3.4.2: Adaptive Neighbourhood Contraction Generalizer

Input: graph $G = (V, E)$, factor $\varepsilon_{\text{diff}}$
Output: graph $G' = (V', E') \mid \forall v_1, v_2 \in V' : \texttt{Distance}(v_1, v_2) \geq \varepsilon_{\text{diff}}$

```
1 while not Empty($V$) do
2     $v \leftarrow$ node with densest $2 \cdot \varepsilon_{\text{diff}}$ neighbourhood in $V$;
3     $d_{\text{n}} \leftarrow 2 - \frac{\texttt{NumberOfNeighbours}(v)}{\texttt{MaxNumberOfNeighbours}(G)}$;
4     forall the nodes $u$ within radius $d_{\text{n}} \cdot \varepsilon_{\text{diff}}$ of $v$ do
5         u ← v;
6     end
7 end
8 return $G$;
```

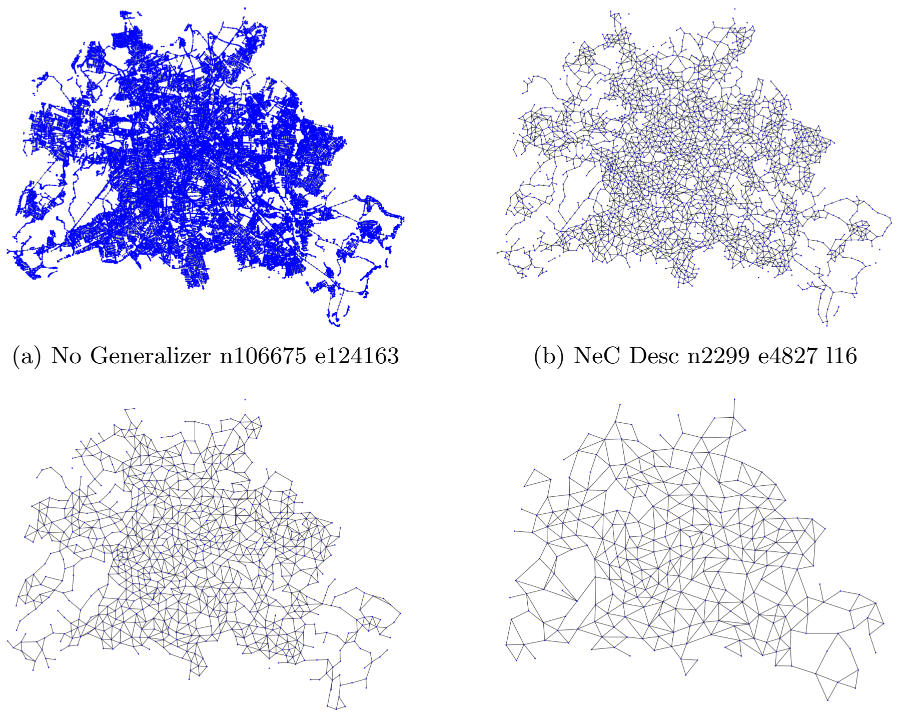

(a) No Generalizer n106675 e124163
(b) NeC Desc n2299 e4827 l16
(c) NeC Desc n838 e1867 l17
(d) NeC Desc n283 e664 l18

Figure 3.20.: Adaptive Neighbourhood Contraction on the OSM Berlin graph.

Nevertheless, the neighborhood heuristic does yield very good results with induced edge mapping for some of the specialized tree instances that are problematic for edge contraction, in particular the regular star (cf. Figure 3.21) with uniform or near-uniform edge length, which can be considered a best case instance for ordering by neighborhood size. The induced edge mapping suffices for an ideal generalization, whereas with the length-based edge contraction no gradual abstraction of a regular star is possible at all.

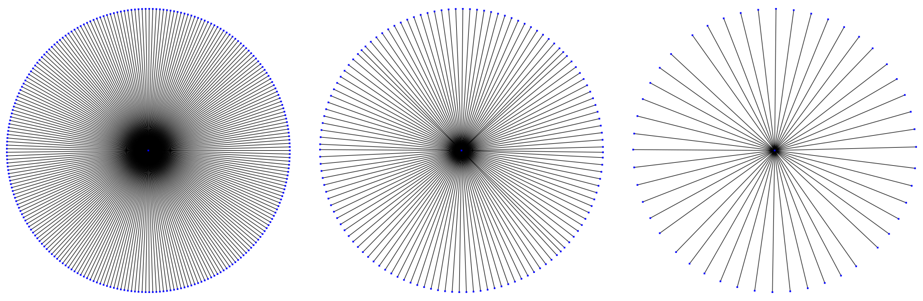
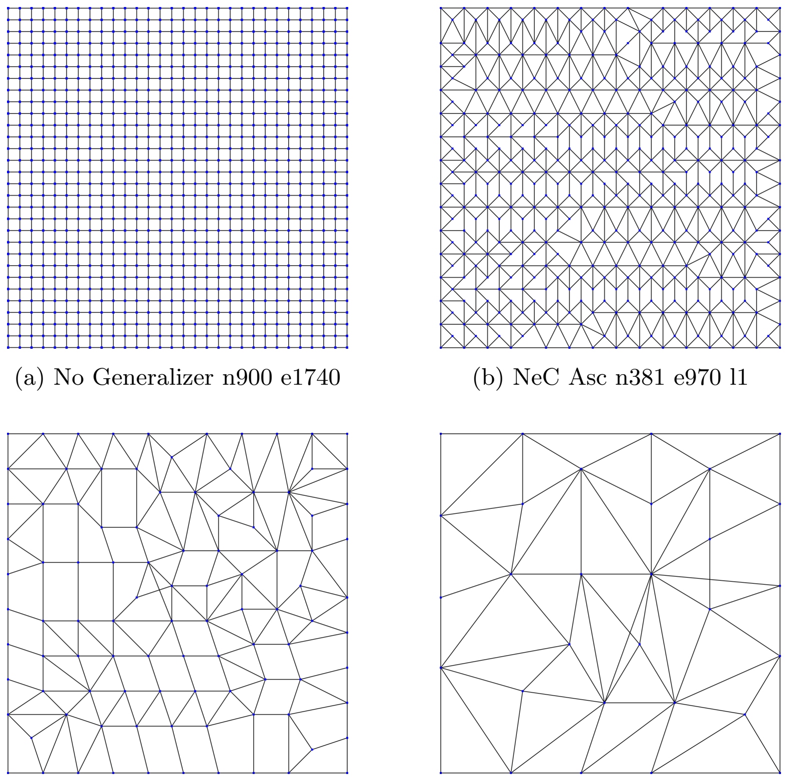

(a) No Generalizer n250 e249 (b) NeC Desc n128 e127 l2 (c) NeC Desc n49 e48 l4

Figure 3.21.: Neighbourhood Contraction on the regular star with uniform edge length.

Random order of contraction is still an issue though. Figure 3.22 shows the generalization for a regular 10x10 grid, which is a lot better better than the edge contraction heuristic's results. Contracting by ascending neighborhood size has the benefit that the grid's corner vertices remain fixed and the generalized graph has a bounding square. However, the order of contraction among the leftover boundary vertices, and later the central vertices, is random for both the ascending and descending variant. Therefore, the generalized node set is generally still too erratic to map a regular rectangular structure on.

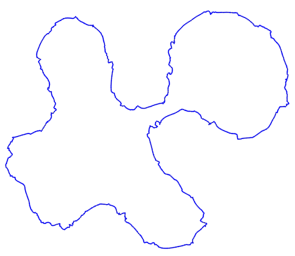

(a) No Generalizer n900 e1740

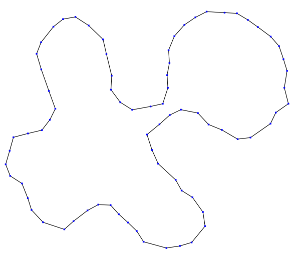
(b) NeC Asc n381 e970 l1

(c) NeC Asc n99 e229 l2

(d) NeC Asc n30 e73 l3

Figure 3.22.: Neighborhood contraction on the regular 30x30 grid.


(a) No Generalizer n1000 e1000


(b) NeC Asc n76 e76 l5

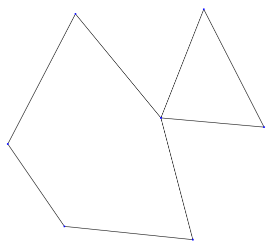
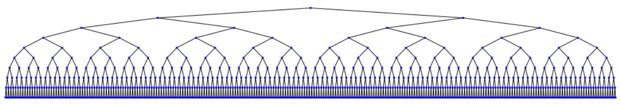

(c) NeC Asc n7 e8 l8

Figure 3.23.: Neighborhood contraction on the 1000-node polygon.

As mentioned above, graph properties and structure generally are not abstracted well. The polygon, for example, now self-intersects (cf. Figure 3.23).

When run on trees, neighborhood contraction generally does somewhat worse than edge contraction, due to its comparatively heavy dependency on the graph layout. Especially the induced mapping is problematic. Figures 3.24 and 3.25 show the results for the complete binary tree of depth six with canonic and organic layout, respectively. Ascending order of contraction works somewhat better than descending order, but neither delivers a generalized node set regular enough to map a complete binary tree on.


(a) No Generalizer n1023 e1022

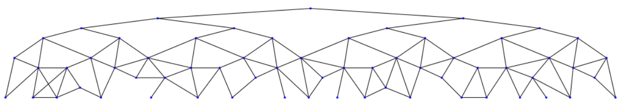
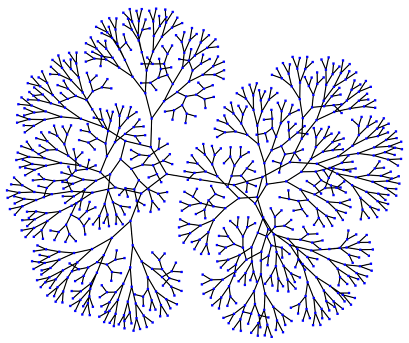

(b) NeC Asc n63 e117 l5

Figure 3.24.: Neighborhood contraction on the complete binary tree with canonic layout.


(a) No Generalizer n1023 e1022

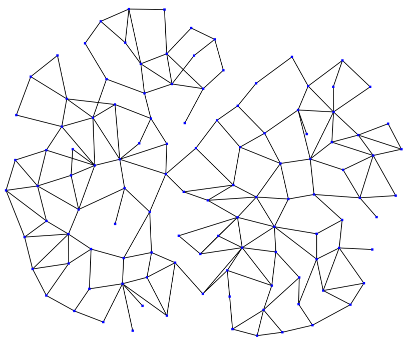
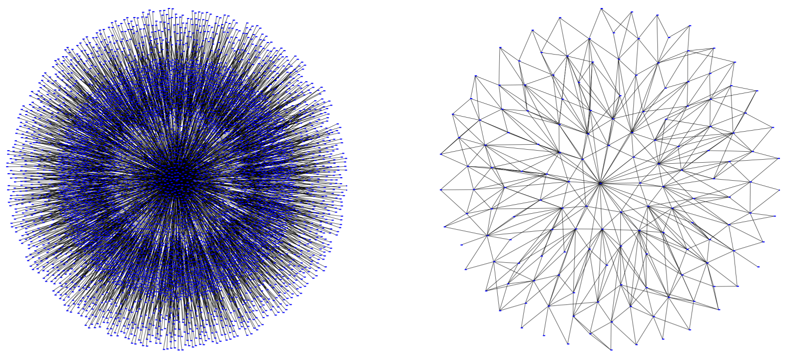

(b) NeC Asc n114 e209 l3

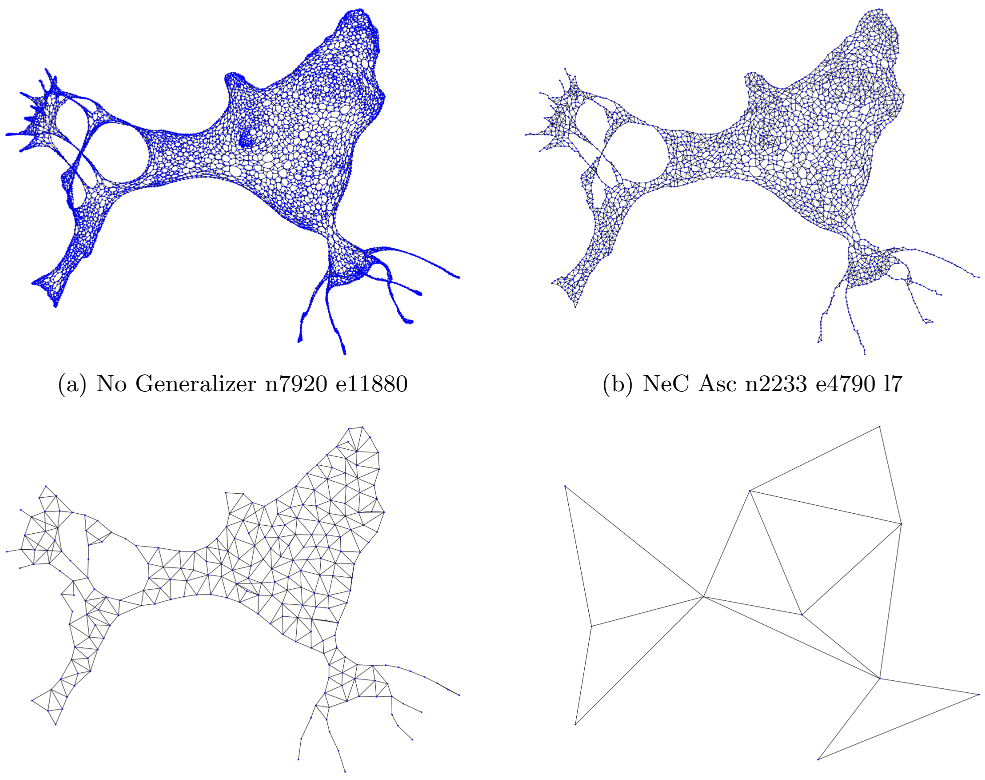
Figure 3.25.: Neighborhood contraction on the complete binary tree with organic layout.

Even in cases where the neighborhood contraction delivers a notably better node set abstraction than edge contraction, such as the Cep1 sparse matrix graph in Figure 3.26, the induced mapping is generally much worse. However, it works well with nets and similar structures. While the generalization of the Commanche sparse matrix graph, for example, is not recognizable as a 3D mesh anymore, it retains both shape and general structure of the original graph (cf. Figure 3.27).



(a) No Generalizer n6290 e8233 (b) NeC Asc n150 e527 l5

Figure 3.26.: Neighborhood contraction on the Cep1 sparse matrix graph.



(a) No Generalizer n7920 e11880 (b) NeC Asc n2233 e4790 l7

(c) NeC Asc n241 e547 l9 (d) NeC Asc n11 e18 l12

Figure 3.27.: Neighborhood contraction on the Commanche sparse matrix graph.

4. Edge Filtering

The introduced contraction-based generalization methods, with the possible exception of node contraction, are mainly geared towards creating a good abstraction of the node set. While an edge mapping arises naturally for some specialized instances, such as edge contraction on trees and neighborhood contraction on wheel graphs, finding such an edge set approximation for general graphs is problematic. Due to the comparatively much larger visual impact, a bad edge mapping ruins the abstraction regardless of the quality of the chosen node subset. Also, while it is nearly impossible to attain a good generalization on a poorly abstracted node set (as is the case with the grid instances), selecting the edges carefully can usually improve the results significantly. Determining an entirely new set of edges (see Section 5) is costly, however, and not always necessary. This section discusses alternative solutions for two common edge mapping problems.

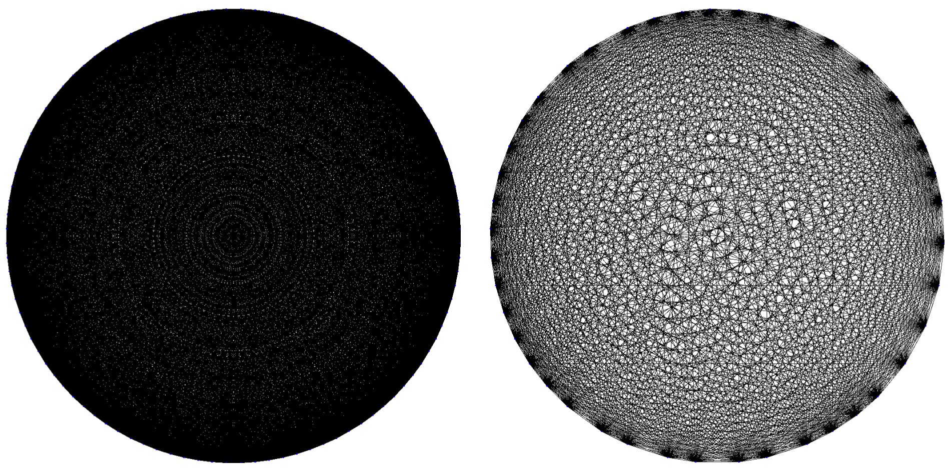

(a) No Generalizer n100 e4950 (b) NeC Asc n43 e903 l1

Figure 4.1.: Edge cluttering on the circular 100-node clique

First, for very dense graphs the large number of edges generally causes visual cluttering even on zoom levels where node proximity is not an issue and no node set abstraction is required. If the graph layout is disadvantageous, none of the examined generalization methods by themselves provide an adequate solution. A worst case example are circular cliques, where single edges remain indiscernible for all but the highest abstraction levels (cf. Figure 4.1 for a 100-node clique with circular layout).

Second, the simple mapped edge set induced by node movement is generally not optimal. As a purely node-based heuristic, the neighborhood contraction generalizer in particular tends to result in a mapping with much lower node-to-edge ratio and very different edge distribution compared to the original graph. On road networks, such as the OpenStreetMap graphs, neighborhood contraction produces increasingly triangulated graphs for higher abstraction levels (cf. level 17 of OSM Berlin in Figure 4.2). This is an issue for most sparse graphs, and also applies, albeit to a much lesser extent, for the edge and node contraction methods with induced edge mapping.

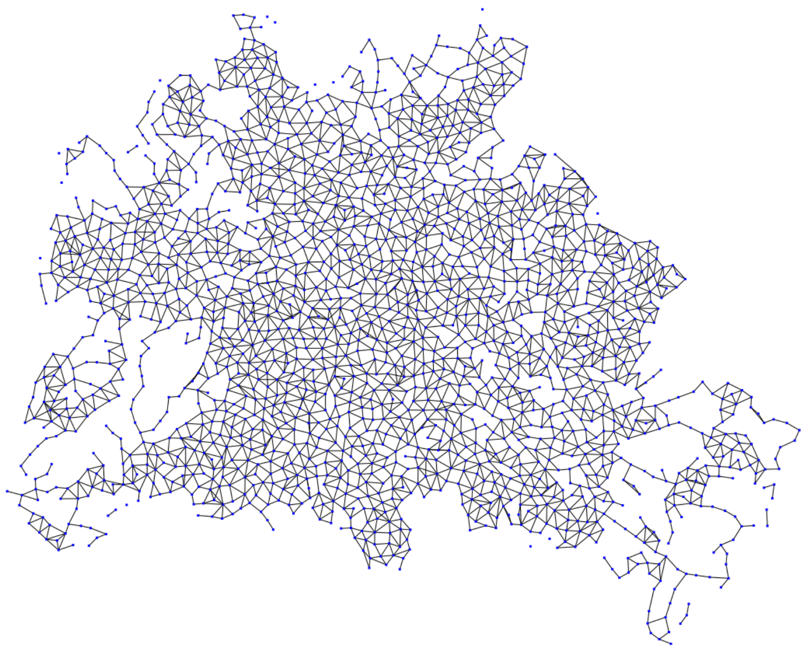


Figure 4.2.: OSM Berlin NeC Asc n2017 e4489 l17.

Several fast measures to alleviate problems of this nature in a post-processing step are discussed in the following. They rely mainly on identifying unwanted edges and filtering them out according to desired generalization properties. As such, the introduced filters are all one-dimensional in that they only optimize a single property of the graph, and each is tailored to address generalization problems arising for specific graphs classes.

4.1. T-Spanner

One of the most meaningful characteristics to preserve while pruning edges is shortest path length, particularly for road networks and similar graphs. This can be achieved by computing a t-spanner (cf. Figure 4.3) on the generalized graph and discarding only unnecessary edges.

Definition 4.1 (T-Spanner) *Given an undirected connected geometric graph $G = (V, E)$ and a real number $t \geq 1$. A t-spanner of G is a spanning subgraph $G' = (V, E')$ that satisfies $\forall u, v \in G : d_{G'}(u, v) \leq t \cdot d_G(u, v)$, where $d_G(u, v)$ denotes the length of the shortest path between u and v in G.*

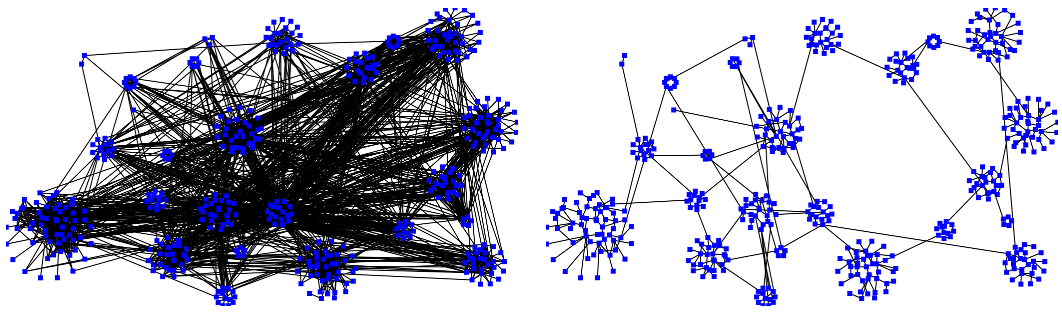

(a) Original graph (b) 2-Spanner

Figure 4.3.: T-Spanner example.

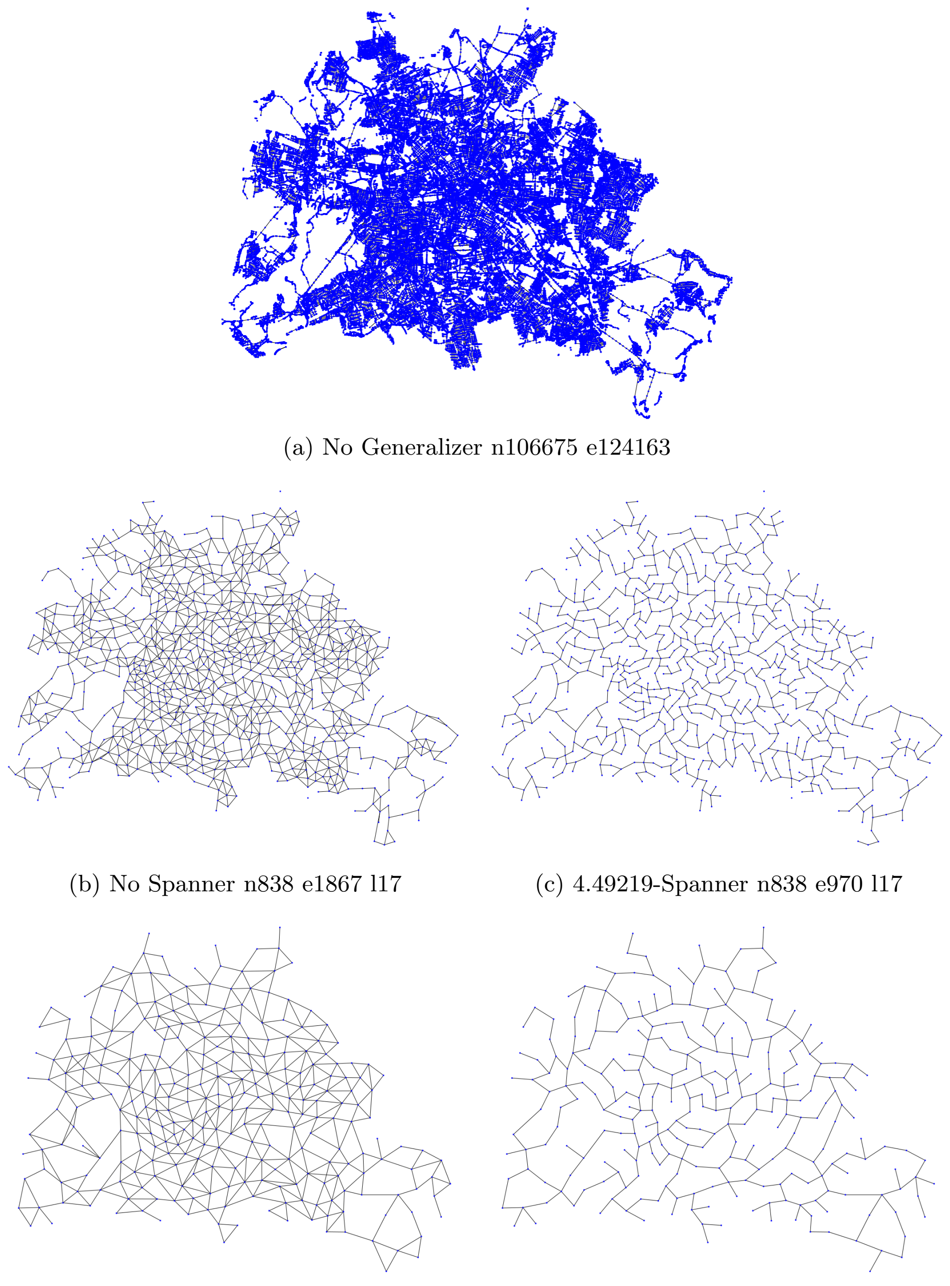
A simple spanner can be constructed greedily with Algorithm 4.1.1 in $\mathcal{O}(|E| \cdot (|E| + |V| \log |V|))$ time, as described by Gudmundsson, Levcopoulos and Narasimhan [GLN02]. Edges are sorted by length and added iteratively to the spanner if they are necessary to meet the shortest path length constraint (cf. Definition 4.1) for its source and target node. Once this constraint holds for all adjacent nodes, it is satisfied for arbitrary node pairs as well. Since the spanner graph starts out empty and usually remains disconnected for a large part of the computation, the Dijkstra queries waste much time exploring increasingly larger connected components before determining that a target node cannot be reached at all. To speed up the process, we use Union Find (see Tarjan [Tar75] for a detailed description and analysis of the data structure) to check if nodes are connected before computing their shortest path distance. On suitably sparse graphs it is generally desirable for the generalization to approximate the original node-to-edge ratio, which can be achieved by a binary search on the t-value.

Algorithm 4.1.1: Greedy T-Spanner

Input: Graph $G = (V, E)$, factor t
Output: T-Spanner $S = (V, E')$ of G

```
1 $S \leftarrow (V, E' = \emptyset)$;
2 while not Empty($E$) do
3     $e \leftarrow$ shortest edge in $E$;
4     if ShortestPath($S$, Source($e$), Target($e$)) $> t \cdot$ Length($e$) then add $e$ to E';
5     remove $e$ from E;
6 end
7 return $S$;
```

Filtering by spanner property is particularly suited for improving neighborhood contraction with induced edge mapping, since a t-value larger than $\sqrt{2}$ suffices to eliminate triangulated structures in the generalization. While this is at times difficult to reconcile with the target node-to-edge ratio for denser graphs, it allows for a good and reasonably fast generalization of road networks and similar sparse graphs. Figure 4.4 shows a comparison of the adaptive neighborhood contraction heuristic with and without t-spanner post-processing on the OSM Berlin data.



(a) No Generalizer n106675 e124163

(b) No Spanner n838 e1867 l17

(c) 4.49219-Spanner n838 e970 l17

(d) No Spanner n283 e664 l18

(e) 4.10156-Spanner n283 e332 l18

Figure 4.4.: Adaptive neighborhood contraction with t-spanner on the OSM Berlin graph.

While the t-spanner method works well for adjusting the node-to-edge ratio as much as possible and to avoid triangulation, a lot of the original graph's edge structure is lost in the process. Depending on individual graph characteristics, it may be preferable to instead filter the generalized edge set to match specific desired graph properties to a degree. Since unrestricted filtering tends to leave the graph disconnected, these filters on their own are mainly useful for special problem instances and if the node-to-edge-ratio deviates by a great amount. In most other cases, it is more beneficial to instead compute a corresponding order of removal for the greedy spanner algorithm to use in place of edge length, and adjust the node-to-edge ratio by binary search on the t-value as before.

4.2. Histogram Filters

This section introduces several quick filters that attempt to improve the generalization with regard to edge attributes and their distribution, by matching a normalized histogram of the generalized graph to that of the original as closely as possible.

Problem 4.1 (Histogram Matching Problem) *Given two graphs $G = (V, E)$ and $G^{\text{ref}} = (V^{\text{ref}}, E^{\text{ref}})$. Consider the finite normalized histograms H and H^{ref} of any edge property (e.g. angle distribution) with histogram buckets labeled h_{i}, size of a bucket $|h_i|$ and number of buckets $|H|$. Determine a subset of edges to discard from E such that $\sum_{\text{i}}(|h'_{\text{i}}| - |h^{\text{ref}}_{\text{i}}|)^2$ is minimal for the remainder graph G'.*

The histogram-based filters use the greedy strategy described in Algorithm 4.2.1 to approximate an optimal solution. Edges are removed from the graph as long as the gap value $v = (|h| - |h^{\text{ref}}|)^2 - (|h| - |h^{\text{ref}}| - 1)^2$ between their buckets h and h^{ref} in the respective histograms is positive. Figure 4.5 shows an example.

If a target value for $|E'|$ is given, the generalization histogram's sum can be normalized to that number instead of $|E|$. In this case the greedy approach then trivially yields an optimal solution as long as each edge only counts towards a single histogram bucket, since then all edges have the same value and the order of removal is irrelevant. This is the case for both the angle and 4D filter method, which are discussed in the following.

Algorithm 4.2.1: Greedy Histogram Matching

```
Input: Graphs $G = (V, E)$, $G^{\text{ref}} = (V^{\text{ref}}, E^{\text{ref}})$, histograms $H$, $H^{\text{ref}}$
Output: Graph $G' = (V, E')$ with minimized $\sum_{\text{i}}(|h'_{\text{i}}| - |h^{\text{ref}}_{\text{i}}|)^2$
1  scale $\sum_{\text{i}} h_{\text{i}}$ to $|E|$;
2  while not Empty($E$) do
3      h ← $h_i \in H$ that maximizes $v_i = (|h_i| - |h^{\text{ref}}_{\text{i}}|)^2 - (|h_i| - |h^{\text{ref}}_{\text{i}}| - 1)^2$;
4      if v > 0 then
5          $e$ ← top edge from h;
6          remove $e$ from h and $E$;
7      else
8          break;
9      end
10 end
11 return $(V, E)$;
```

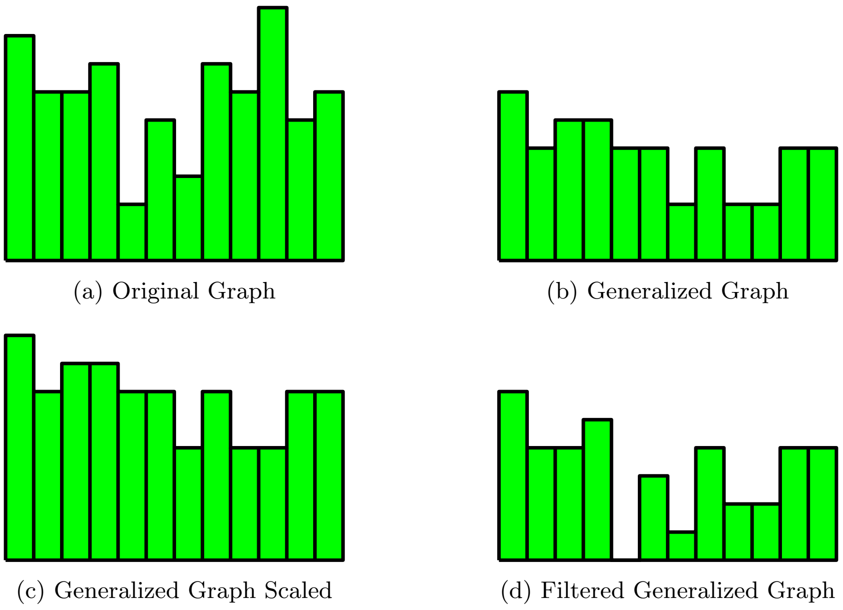

(a) Original Graph (b) Generalized Graph (c) Generalized Graph Scaled (d) Filtered Generalized Graph

Figure 4.5.: Greedy histogram matching.

4.2.1. Angle Filter

Angles range from 0° to 180° and are sorted into k buckets of equal size, where k is a tuning parameter. Since nodes move around, the majority of the remaining edges usually do not fit the original graph's edge angle distribution. A broad distribution is therefore very hard to reasonably approximate by filtering and the divergent edges mostly do not have much visual impact, anyway. For some very regularly shaped graphs, where only very few specific edge angles actually occur, however, edges that deviate from the norm stand out very much visually and removing them can greatly improve the end result. Primary examples are regular grids and similar structures.

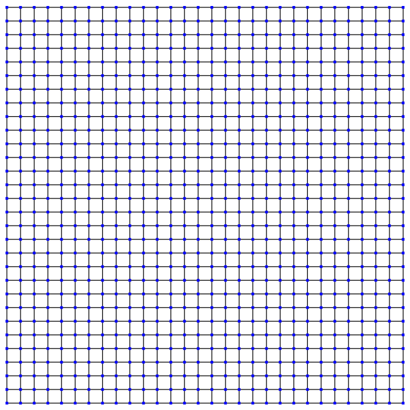
Figure 4.6 shows a comparison of neighborhood contraction on the regular 30x30 grid, both with angle based order of removal ($k = 36$) for t-spanner post-processing and the angle filter on its own. Since the generalized graph's node-to-edge ratio is only marginally lower than that of the original graph, the t-spanner method is preferable. The angle filter has to discard a lot of edges that actually match the original edge angle distribution in order to adjust the ratio of vertical to horizontal edges, which leaves the graph disconnected.

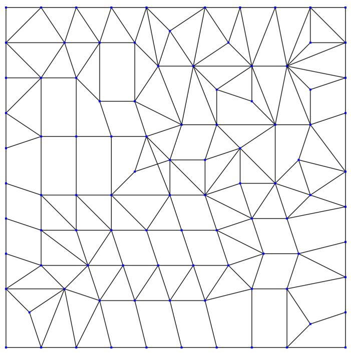
4.2.2. 4D Filter

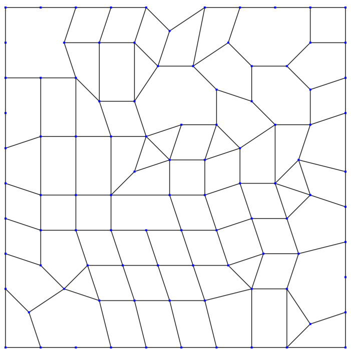
An edge's histogram bucket is computed based on its source coordinates, length and angle as in Formula 4.1. To make the computation symmetric, the source is defined to be the endpoint with the lexicographically smaller coordinate.

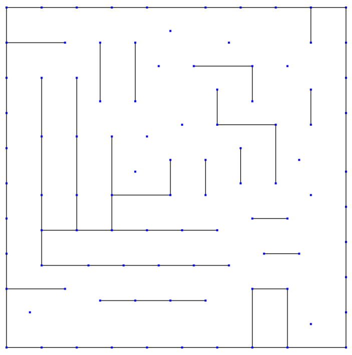
$$i_{\text{bucket}} = i_{\text{length}} + k \cdot (i_{\text{angle}} + k \cdot (i_{\text{x}} + k \cdot i_{\text{y}})) \tag{4.1}$$

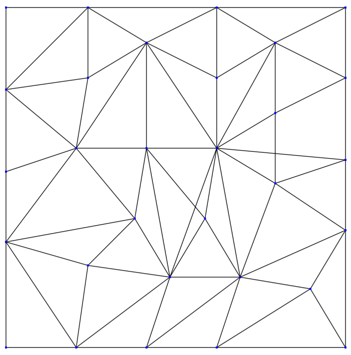
The tunable parameter k determines the histogram's size $|H|$ and allows a tradeoff between filter quality and computation time, which is in $\mathcal{O}(|E| \cdot \log |H|)$. Figure 4.7 shows some results for post-processing of neighborhood contraction on the Aug3d sparse matrix graph. When using the 4D filter with t-spanner post-processing, the generalization retains a lot more of the original graph's acute angles.

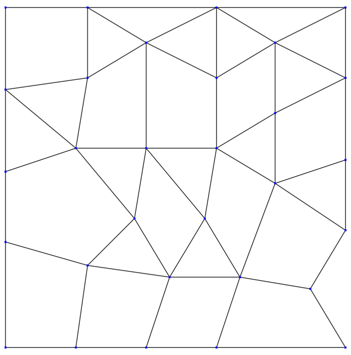

(a) No Generalizer n900 e1740


(b) No Filter n99 e229 l2


(c) T-Spanner n99 e170 l2


(d) Angle Filter n99 e78 l2


(e) No Filter n30 e73 l3


(f) T-Spanner n30 e57 l3

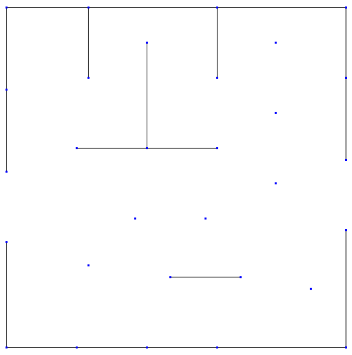
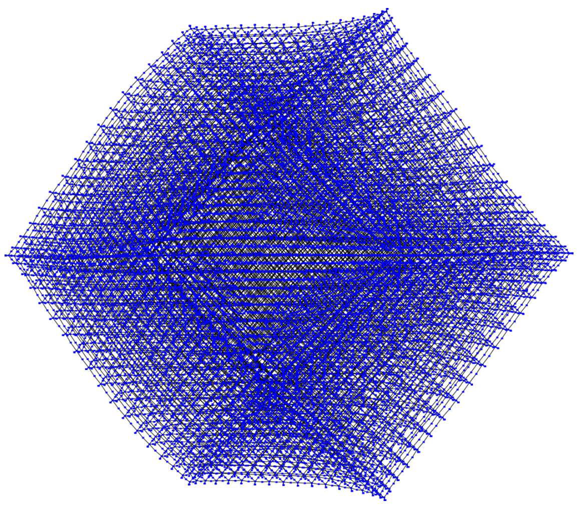
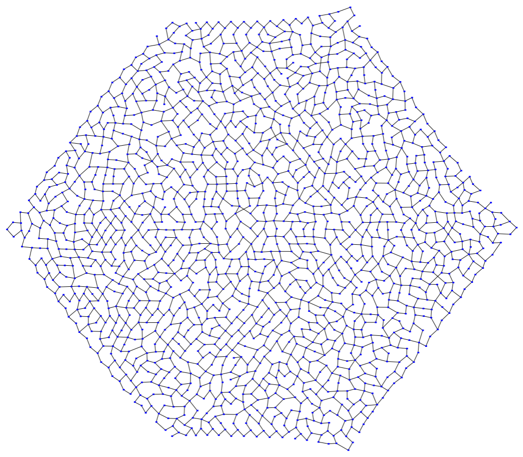
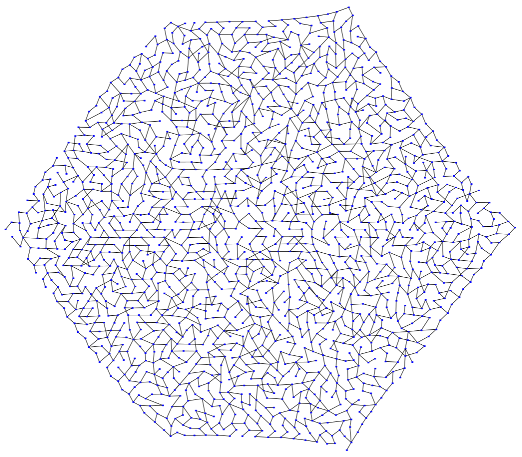
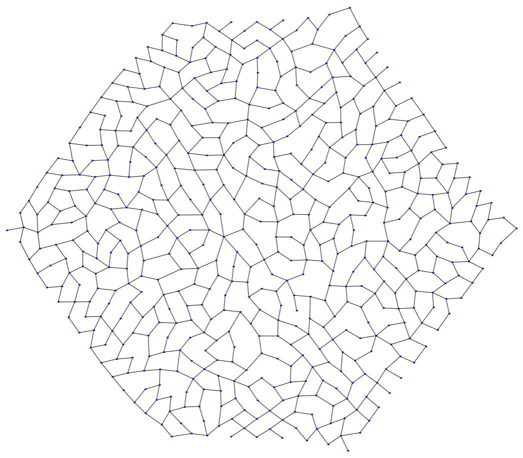

(g) Angle Filter n30 e19 l3

Figure 4.6.: Neighborhood Contraction with Angle Filter on the regular 30x30 grid.


(a) No Generalizer n24300 e34992


(b) No Filter n1829 e2617 l8


(c) 4D Filter n1829 e2553 l8


(d) No Filter n565 e815 l9

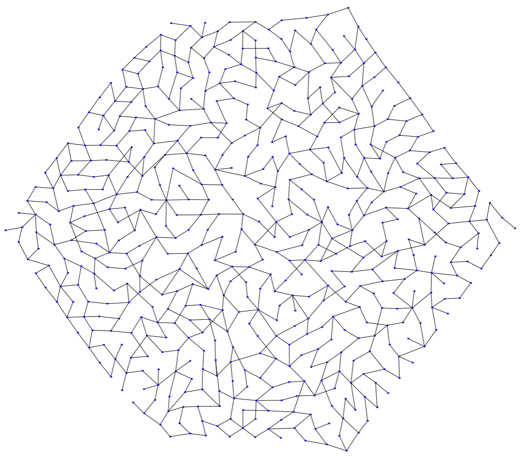
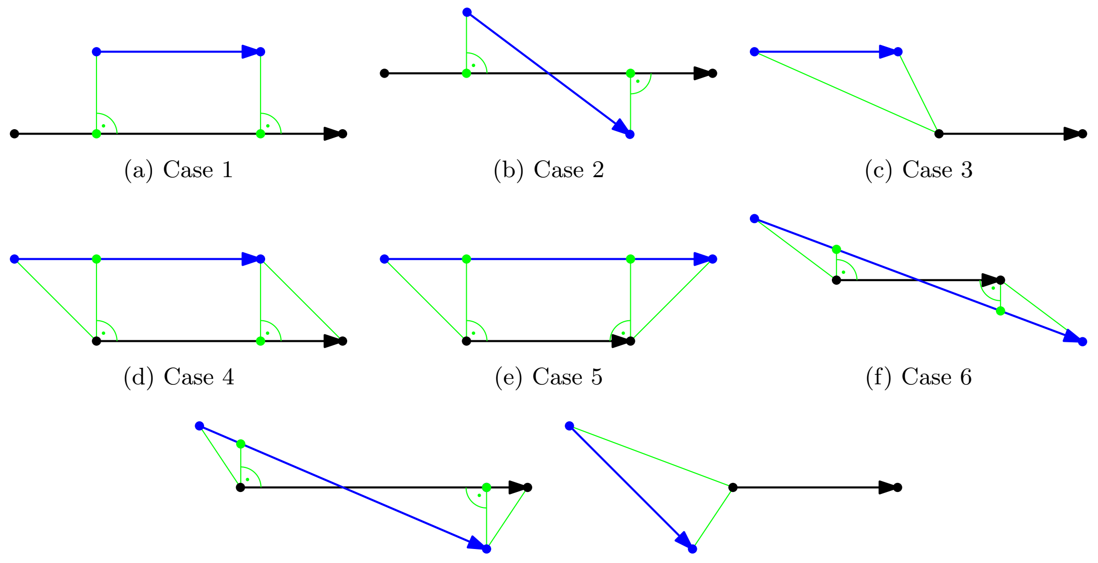

(e) 4D Filter n565 e795 l9

Figure 4.7.: Neighborhood contraction with angle filter on the Aug3d graph.

4.3. Average Distance Filter

Unlike the previous histogram-based filtering methods, the average distance filter does not attempt to improve the generalization by enhancing likeness to the original graph. It instead focuses on eliminating edge cluttering caused by edge proximity and redundant parallel edges, which is a problem that frequently occurs for all three basic generalizers and can not be adequately solved by using a t-spanner. The filter algorithm checks each edge against all others in the set. If the similarity of two edges exceeds a zoom level dependent threshold the shorter one is discarded.

Edge similarity is assessed based on the average distance of two edges, which is calculated as the definite integral of an edge's distance function. A lower value means greater similarity. This computation is not symmetric, the average distance from a short to a long edge is smaller than in the reverse case. Depending on relative edge position and length, there are eight distinct cases for the computation to distinguish (cf. Figure 4.8).



(a) Case 1 (b) Case 2 (c) Case 3

(d) Case 4 (e) Case 5 (f) Case 6

(g) Case 7 (h) Case 8

Figure 4.8.: All distinct edge constellations for avg. distance computation.

Depending on the case at hand, the distance value is made up of one or more of three basic distance function integrals.

Given start and end points $\vec{p}_{\text{start}} = \vec{e}_{\text{source}} + s_1\vec{e}_{\text{dir}}$ and $\vec{p}_{\text{end}} = \vec{e}_{\text{source}} + s_2\vec{e}_{\text{dir}}$ on the edge, we find their respective nearest points $\vec{p}_1$ and $\vec{p}_2$ on the reference edge ($\vec{p}$ if they coincide) and compute the Euclidean distances $l_1 = \|\vec{p}_{\text{start}} - \vec{p}_1\|$ and $l_2 = \|\vec{p}_{\text{end}} - \vec{p}_2\|$. The distance functions $d(s)$ and corresponding integrals for the first three cases can then be expressed as follows.

Case 1 : Line to Line

$$d(s) = l_1 + (l_2 - l_1)s \tag{4.2}$$

$$\int_{s1}^{s2} d(s)\,\mathrm{d}s = \frac{1}{2}(l_1 + l_2)(s_2 - s_1) \tag{4.3}$$

Case 2 : Crossing Line to Line

$$d(s) = \begin{cases} l_1 - s(l_1 + l_2) & : 0 \leq s \leq \frac{l_1}{l_1+l_2} \\ (l_1 + l_2)(s - \frac{l_1}{l_1+l_2}) & : \frac{l_1}{l_1+l_2} \leq s \leq 1 \end{cases} \tag{4.4}$$

$$\int_{s1}^{s2} d(s)\,\mathrm{d}s = \frac{l_1^2}{l_2 + l_1} - \frac{1}{2}(l_2 - l_1)(s2 - s1) \tag{4.5}$$

Case 3 : Line to Point

$$a = \vec{e}_{\text{source}} \cdot \vec{e}_{\text{source}} + \vec{p} \cdot \vec{p} + \vec{p} \cdot \vec{e}_{\text{source}}$$

$$b = 2\ (\vec{e}_{\text{source}} \cdot \vec{e}_{\text{dir}}) - \vec{p} \cdot \vec{e}_{\text{dir}}$$

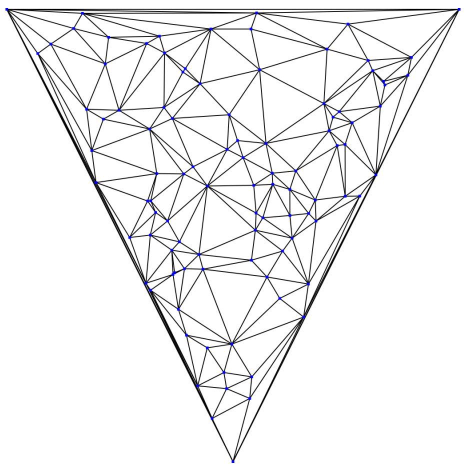
$$c = \vec{e}_{\text{dir}} \cdot \vec{e}_{\text{dir}}$$

$$d(s) = \sqrt{a + bs + cs^2} \tag{4.6}$$

$$\int d(s)\,\mathrm{d}s = \frac{2\sqrt{c}(b + 2cs)d(s) - (b^2 - 4ac)\log(b + 2cs + 2\sqrt{c} \cdot d(s))}{8c^{\frac{3}{2}}} \tag{4.7}$$

All other cases can be reduced to appropriate combinations of these three cases. Case 4, for example, is computed as the sum of two line to point distances (Case 3) and one line to line distance (Case 1).

Figure 4.9 shows the application of the average distance filter as a post-processing step for neighborhood contraction. The example graph is a random triangulation of 100 nodes with planar layout. Filtering by average distance effectively removes most indiscernible edges, in particular the longer parallel ones on the outside and some very short inner ones.


(a) No Filter n99 e291 l2

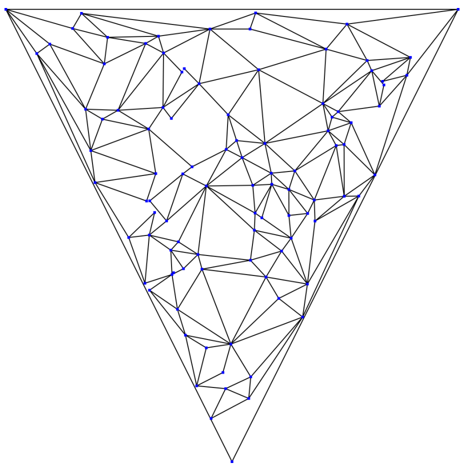

(b) Distance Filter n99 e231 l2

Figure 4.9.: Neighborhood Contraction on the triangulation of 100 nodes.

4.4. Density Filter

Like the previous average distance filter, the density filter aims to reduce edge cluttering. Instead of determining individually whether any given pair of edges is indistinguishable, this approach overlays the graph with a grid and enforces zoom level dependent constraints on the number of edges that may cross a cell. This leads to the following general problem.

Problem 4.2 (Density Grid) *Given a graph $G = (V, E)$, threshold values $c_{\min}, c_{\max} \in \mathbb{Z}$ with $0 \leq c_{\min} \leq c_{\max} \leq |E|$ and a regular grid Λ with cell width ℓ_Λ. Let c_λ^{E} denote the number of edges in E that cross grid cell λ. Is it possible to discard a number of edges from E such that the remaining subset E' satisfies $\forall \lambda \in \Lambda \mid c_\lambda^{\mathrm{E}} > 0 : c_{\min} \leq c_\lambda^{\mathrm{E}'} \leq c_{\max}$?*

While for $c_{\min} = 0$ this filtering problem is always trivially solvable, the related problem of finding a solution with maximal number of edges might be worth consideration.

Theorem 4.1 *The density grid problem is NP-hard.*

Proof

As with the Minimum Distance Vertex Layout problem examined in section 3.4, the NP-hardness proof follows by reduction of planar 3SAT. It proceeds analogously for a given instance of planar 3SAT by constructing an instance of the density grid problem that is solvable if and only if the 3SAT formula is satisfiable. This instance is composed of variable, literal and clause gadget which are described subsequently. The chosen threshold values are $c_{\min} = c_{\max} = 1$. Figure 4.10 gives an example which illustrates the concept. Remaining and discarded edges are colored blue and green, respectively.

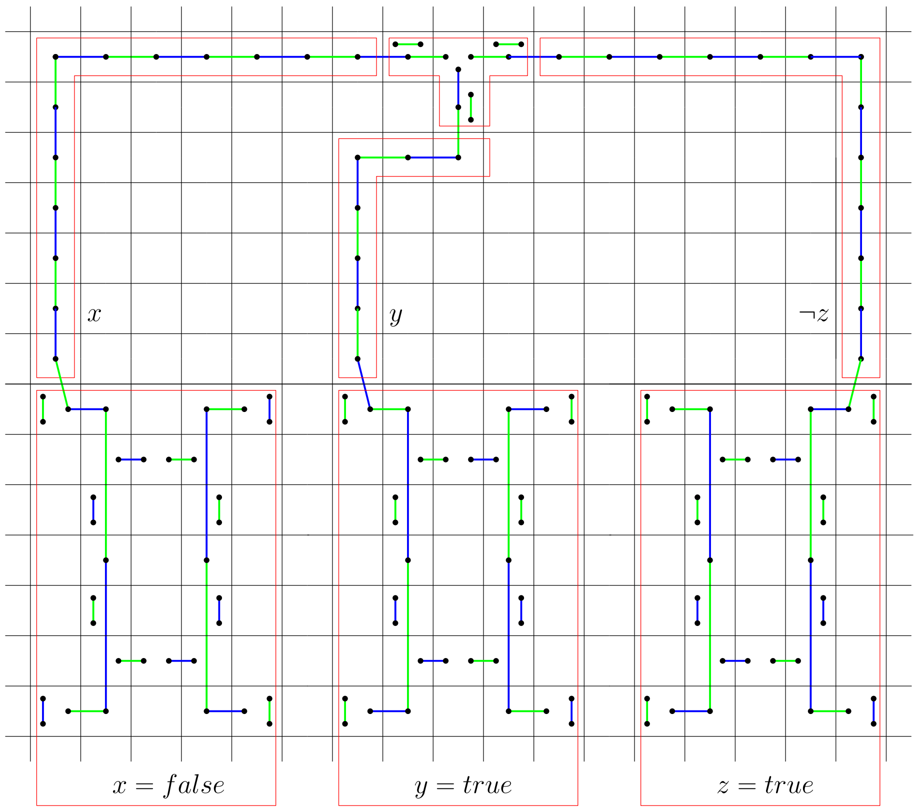


Figure 4.10.: Satisfying assignment for the formula $\varphi = x \vee y \vee \neg z$.

Variable gadgets.

The basic component for variable gadgets is the symmetrical arrangement of 20 edges on 34 nodes depicted in Figure 4.11(a). It consists of two vertically mirrored four-edge structures (staples, blue), two central links of two edges each (green), four outer buffer edges (red) and four additional inner buffer edges (orange).

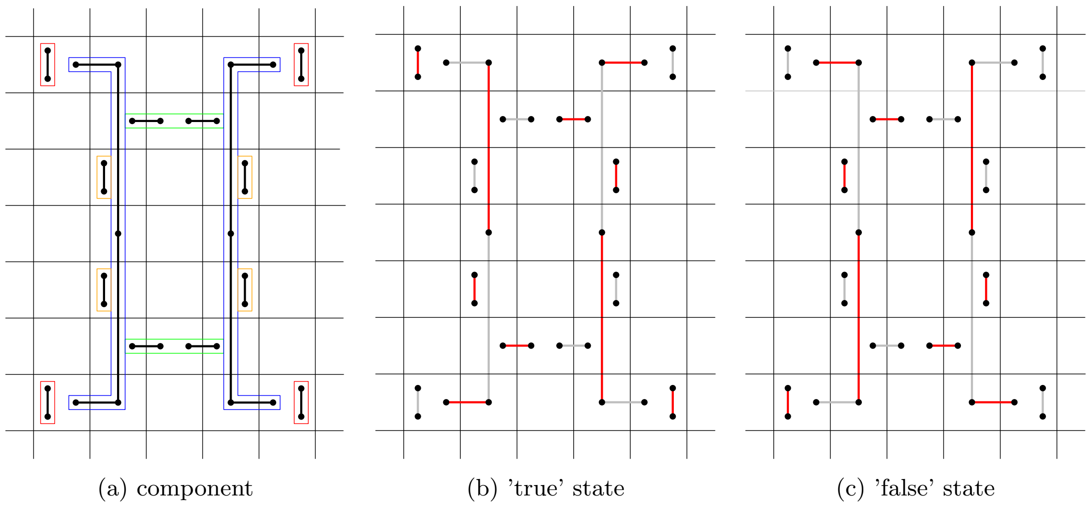

(a) component (b) 'true' state (c) 'false' state

Figure 4.11.: Variable gadget main component.

Literal gadgets connect to either staple's top or bottom edge, respectively. Depending on whether the left staple's top edge remains in the graph or is discarded, the component assumes 'true' or 'false' state (cf. Figures 4.11(b) and 4.11(c), remaining edges are colored red.) .

A variable gadget itself is comprised of a chain of n_x such components, according to the number of the variable's occurrences in the formula, which are connected as shown in Figure 4.12. Remaining and discarded edges are coloured blue and green, respectively. Dashed edges indicate possible literal gadget connections. Only the first and last components in the chain retain their inner buffer edges, for all others these are replaced by a two-edge link segment each.

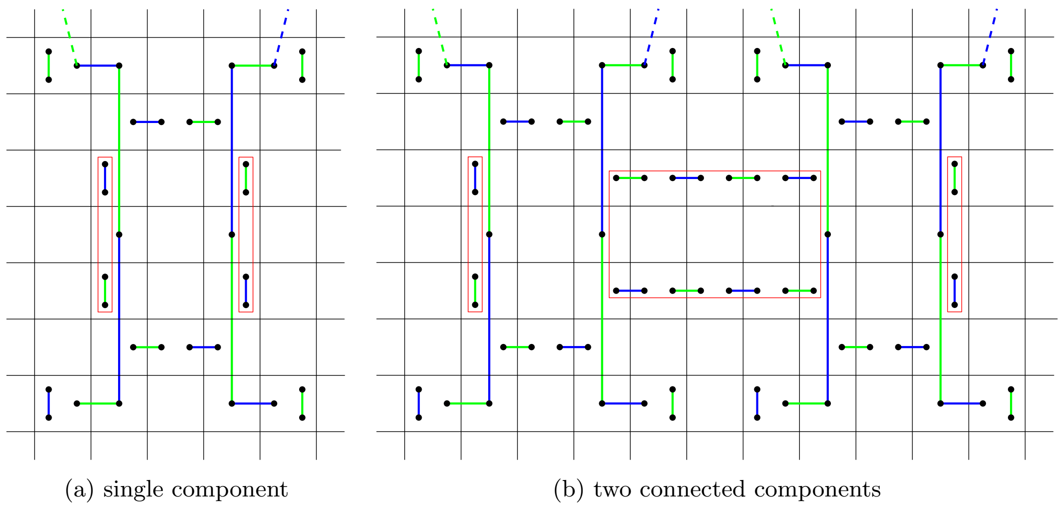
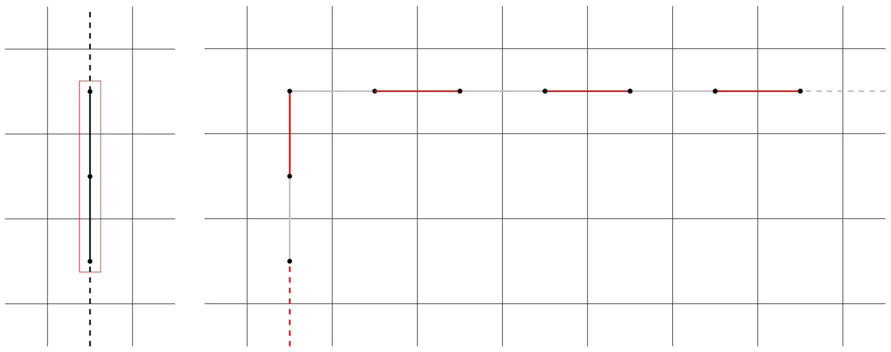

(a) single component (b) two connected components

Figure 4.12.: Variable gadget main component and composition connection.

Literal gadgets.

Literal gadgets are simple strings of edges with length ℓ_Λ (cell width) which connect to the variable gadgets and are aligned in such a way that any two consecutive edges cross the same cell (cf. Figure 4.13). If a literal is not negated in the formula, its corresponding string is attached to the variable component's top left or the bottom right outlet, otherwise it connects on the top right or bottom left, respectively.

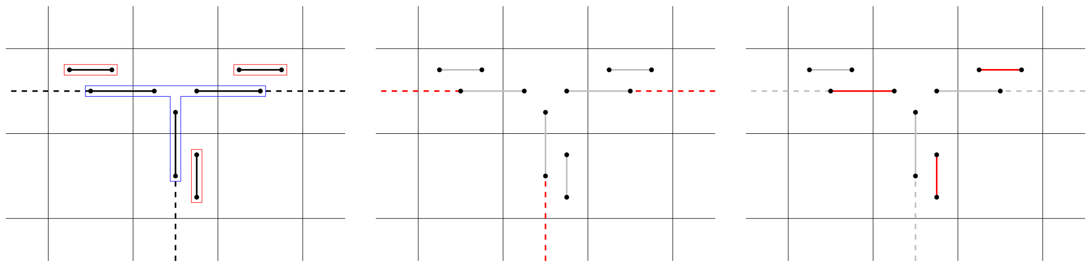


(a) basic link (b) literal gadget string

Figure 4.13.: Literal gadget segments. Remaining edges are coloured red.

Clause gadget.

Figure 4.14(a) shows the clause gadget. It consists of a three-edge junction structure intersecting a single cell and three small buffer edges. Literal gadgets connect to each of the junction's edges. If all variables are assigned incorrectly, all clause structure edges have to be discarded and the central cell remains empty, which is an invalid configuration (cf. Figure 4.14(b)). Conversely, if one or more of the literals evaluate to true, one of their junction edges can be kept in the graph and the others switch to their respective buffer edges accordingly. Figure 4.14(c) shows one possible solution for a case where all variables are assigned correctly.



(a) clause gadget (b) contradiction (c) possible solution

Figure 4.14.: Clause gadget, contradiction and one resolving configuration.

Given a planar 3SAT formula φ, the variable-clause graph can be constructed in polynomial time with gadgets as described above. The density grid problem for this graph can be solved if and only if a fulfilling assignment for the formula exists. Such a variable assignment can in turn be obtained by checking the state of the corresponding gadgets in a valid solution. □

While the density grid problem is NP-hard for constraints on both maximum and minimum number of edges per cell, it can be solved efficiently with a modified version of Bresenham's line algorithm [Bre98] if there is only an upper limit $c_{\max}$ (cf. Algorithm 4.4.1). The same method can also be used to adjust the generalization's density distribution according to that of the original graph, if instead of a global limit appropriate thresholds are precomputed for each cell.

Algorithm 4.4.1: Density Filter

Input: Graph $G = (V, E)$, grid Λ, threshold $c_{\max} \in \mathbb{Z}_0^+$
Output: Graph $G = (V, E')$ that satisfies $\forall \lambda \in \Lambda : c_\lambda^{E'} \leq c_{\max}$

```
1  sort $E$ by length;
2  $E' \leftarrow \emptyset$;
3  while not Empty($E$) do
4      $e \leftarrow$ first edge in $E$;
5      if GridConstrainsSatisfied($\Lambda$, $e$) then
6          add $e$ to $E'$;
7      end
8      remove $e$ from $E$;
9  end
10 return $E'$;
```

Figure 4.15 shows the result of t-spanner post-processing using the density filter order on the UD10 unit disk graph.

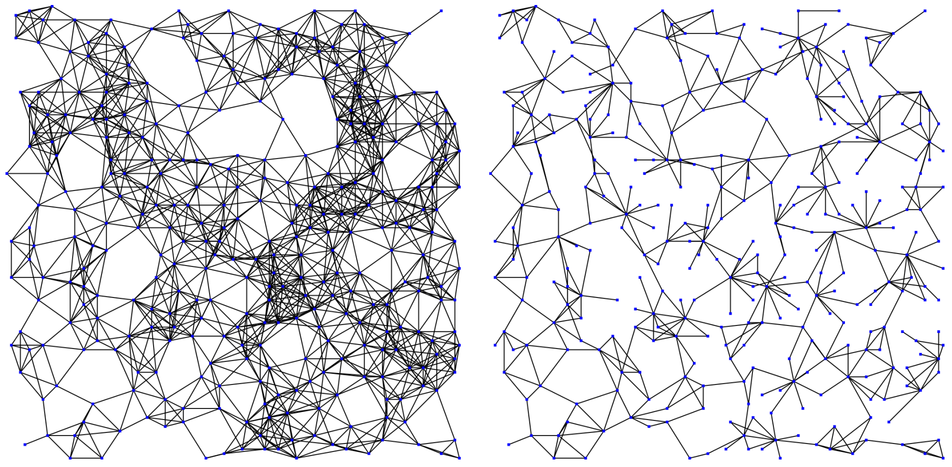

(a) No Filter n336 e1650 l2 (b) Density Filter n336 e566 l2

Figure 4.15.: Neighborhood contraction with density filter on the UD10 unit disk graph.

5. Edge Set Abstraction

The node set abstraction techniques introduced in Section 3 also induce an edge set for the generalization, which requires no additional computation and allows for a fast abstraction that preserves a lot of structure at low abstraction levels. Although this induced mapping yields reasonably good results for a large number of graphs, such as meshes and highly connected structures, it can be lacking with respect to particular desiderata like density or degree distribution. It is also random to a degree, and only optimal for very restricted special instances. In cases where the induced edge set contains too few edges the filtering methods discussed in the previous section are not applicable. Especially for high zoom levels, where the generalized node set is small, more complex methods are feasible to optimize the generalization. Given an abstraction of the node set, this leads to the following general optimization problem.

Problem 5.1 (Edge Set Abstraction) *Given a graph G with node set V, edge set $E \subseteq V \times V$ and a subset $V' \subseteq V$ (or arbitrary set of points V'). Find a set of edges $E' \subseteq V' \times V'$, such that E' is optimized with regard to a target function $t(G, G') \mapsto \mathbb{R}$.*

Theorem 5.1 *The Edge Set Abstraction Problem is NP-hard for general target functions $t(G, G')$, even if these can be evaluated efficiently.*

Proof

The proof follows by reduction from the NP-hard density grid problem introduced in Section 4.2. Given an instance $G = (V, E)$ of the density grid problem, a corresponding instance of the edge set abstraction problem can be constructed as follows. Choose G as the input graph and $V' = V$. The target function $t(G, G')$ assigns the value zero if G' is a valid solution according to the density constraints, and one otherwise. Note that any G' that is not a subgraph of G is automatically rated one, since a valid solution may only contain edges from E. It follows that $t(G, G') = 0$ if and only if G' is a feasible solution of the density grid problem.

$\square$

Note that complexity can generally be even higher, as no constraints at all are put on the target function's complexity class. However, if the target function can be computed in polynomial time we can formulate an integer linear program, and find an optimal solution using an ILP solver.

In the following two polynomial heuristics are discussed. First is an iterative method that selects edges according to properties which are indicative of graph structure and improves the generalization step by step. While such a simple approach would be preferable, only taking single edges into consideration does not suffice in many cases, as much of a graph's structure is encoded in subgraphs. We therefore also discuss an alternative method which inserts multiple edges at a time by identifying and mapping entire representative paths.

5.1. Iterative Insertion

We use an iterative greedy heuristic that starts out with an empty edge set and examines all undirected non-loop edges in $V' \times V'$ for the generalization. They are rated according to the improvement their addition would bring for the generalization and the locally optimal edge is inserted, up to the desired node-to-edge ratio. Algorithm 5.1.1 describes the process in pseudo-code. Note that depending on the target function computation may be slow nonetheless, as during each step the function has to be recomputed for every potential next edge to find an optimal one.

Algorithm 5.1.1: Greedy Edge Insertion Routine

```
Input: graph $G = (V, E)$, node set V', target function t
Output: graph $G' = (V', E')$ with t(G, G') $\approx 0$
1 E' ← $\emptyset$;
2 E" ← all undirected non-loop edges in $V' \times V'$;
3 while $\frac{E'}{V'} < \frac{E}{V}$ do
4     e ← edge in E" with best improvement t(G, G' $\cup$ e) over t(G, G');
5     add e to E';
6     remove e from E";
7 end
8 return G';
```

5.1.1. Desiderata

As discussed in the Preliminaries Section 2.2, finding a single function to measure graph similarity reasonably is problematic. We therefore opt for a weighted linear combination of several simple, easy to formulate target functions most of which are based on the techniques used for edge filtering in Section 4. Each tries to improve the generalized graph with regard to a single graph property.

Histograms

A number of graph attributes can be measured with histograms. These include node degree and length distribution, as well as the 4D histogram introduced in Section 4.2.2. For two histograms H^{G} and $H^{\mathrm{G}'}$ with histogram entries $h \in H$, the target function $t(G, G')$ is computed as

$$t(G, G') = \sum_{h \in H} |h^{\mathrm{G}} - h^{\mathrm{G}'}|^2.$$

Average Distance

For each edge e in the original graph, the minimum average distance as described in Section 4.3 to any edge that is part of the generalization is computed as

$$v_e = \min_{e' \in E'} d_{\text{avg}}(e, e').$$

The target function to minimize is then the sum over all such values v_e

$$t(G, G') = \sum_{e \in E} v_e.$$

Edge Density Grid

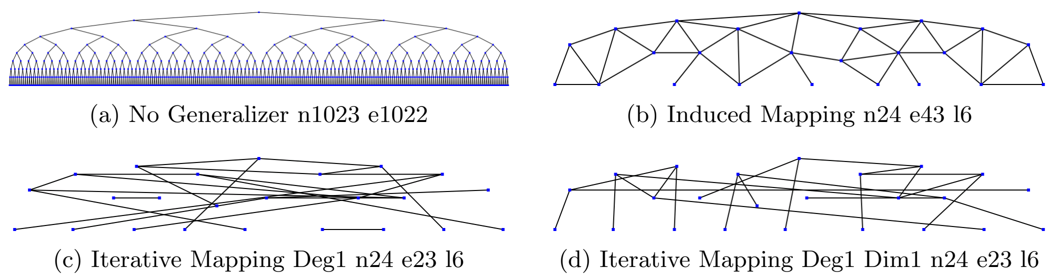
As in Section 4.4 given a grid Λ with grid cells $\lambda \in \Lambda$ we denote the number of edges in E that cross grid cell λ as c_λ^{E}. The target function is

$$t(G, G') = \sum_{\lambda \in \Lambda} |c_\lambda^{\text{E}} - c_\lambda^{\text{E}'}|^2$$

None of these heuristics are optimal on their own, the challenge lies in finding a combination of parameters that work well in general. They also vary in complexity. Histogram based functions, such as length and node degree distribution, are much faster to compute than the average edge distance.

5.1.2. Evaluation

The iterative insertion heuristic does not yield very good results even when fine tuning the parameters by hand. Figure 5.1 shows a comparison with the induced mapping for neighborhood contraction on the complete binary tree with canonic layout. Inserting by degree alone produces a generalization that matches the original graph's distribution exactly, but is not at all visually similar. This can be expected for all histogram based target functions, as they can only provide a very coarse classification. Factoring in the 4D histogram at an equal weight improves the results somewhat, but considering more properties is actually detrimental for this instance. Generally, a lot more basic target functions than the few introduced above are required for a combination to be able to reliably capture the structure of a graph.


(a) No Generalizer n1023 e1022 (b) Induced Mapping n24 e43 l6

(c) Iterative Mapping Deg1 n24 e23 l6 (d) Iterative Mapping Deg1 Dim1 n24 e23 l6

Figure 5.1.: Iterative Edge Insertion on the complete binary tree with canonic layout.

The density grid and distance functions, while more representative of the original graph than the histogram based ones, do not work well in combination. Even for instances where the induced mapping does not contain nearly enough edges to approach the original graph's node-to-edge ratio, such as the UD30 unit disk graph, density is better represented than with the iterative insertion heuristic tuned accordingly (Figure 5.2).

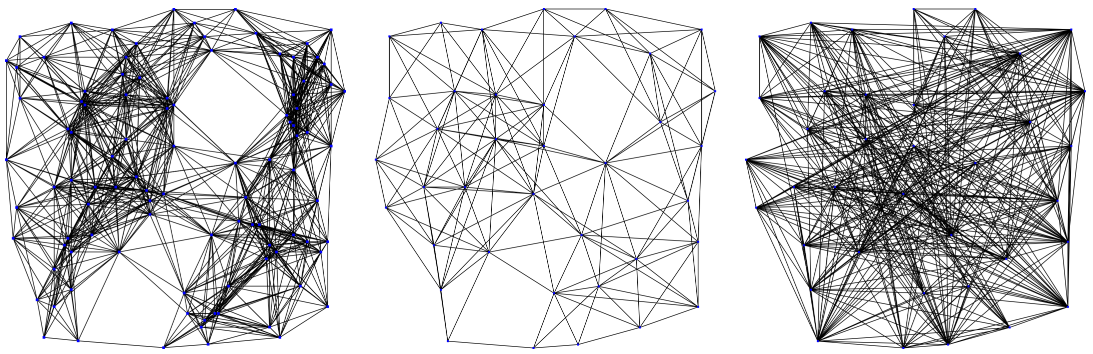

(a) No Generalizer n100 e1012 (b) Induced Mapping n38 e168 l5 (c) Iterative Mapping n38 e263 l5 Den1 Dist1

Figure 5.2.: Iterative Edge Insertion on the UR30 unit disk graph.

5.2. Path Mapping

All of the edge insertion methods discussed above concentrate on mapping edges one at a time, or selecting single edges according to their attributes in order to enhance the generalization. Since a lot of the original graph's structure is ignored, they do not succeed in maintaining path characteristics and connectivity, in particular. The following approach attempts to improve on this by identifying important paths in the graph and mapping them, as closely as possible, to the generalized node set. This mapping process is regulated by a number of degrees of freedom, which are described subsequently. First of all, in order to be able to judge the quality of mapped paths a measure to rate their similarity to the original path is required. Also, the set of generalization nodes for a path to be mapped on needs to be narrowed down, in order to make computation feasible for low abstraction levels. This is achieved by constructing a bounding structure around the original path, which rules out generalization nodes that are too geometrically distant to be useful as part of the mapped path. Given a subset of admissible vertices, an optimal path with regard to the chosen distance measure has then to be determined. Lastly, the question remains which paths of the original graph give a good representation of the original graph's structure, and how many of them should be mapped for a good balance between computation time and quality. These points are discussed in detail in the following.

5.2.1. Distance Measure

In order to determine the similarity of two paths P_1 and P_2, two ways to evaluate path distance are examined. A straightforward property to minimize is the divergence of their respective node sets, which can be measured by the Hausdorff distance (Formula 5.1, cf. [RW98] for an introduction).

$$d_{\mathrm{H}}(P_1, P_2) = \max\{\sup_{p \in P_1} \inf_{p' \in P_2} \|p - p'\|, \sup_{p' \in P_2} \inf_{p \in P_1} \|p - p'\|\} \tag{5.1}$$

While fast to compute, this approach has significant drawbacks. A low Hausdorff distance only indicates the proximity of the paths' node sets and does not necessarily mean that the paths themselves are visually close. In fact, Figure 5.3(a) shows two paths with distance value $d_{\mathrm{H}}(P_1, P_2) = 0$ which are not remotely similar. Also, there are cases where locally optimal path segments are ignored, since only the maximum distance among either path's nodes counts towards the end value (cf. Figure 5.3(b) for an example).

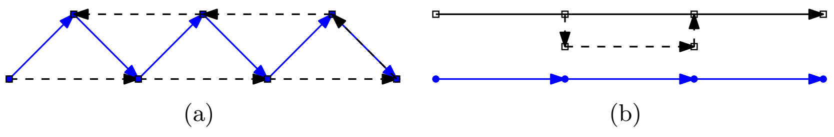


Figure 5.3.: Cases in which the Hausdorrf metric is sub-optimal.

In order to counter some of these problems, we introduce a second method which takes path edges into consideration. By connecting the two paths to be compared, their distance can be measured as the area of the resulting polygon $P = (p_1, ..., p_n)$. The computation (Formula 5.2) is not significantly more complex than the Hausdorff metric.

$$A_{\text{polygon}}(P) = \frac{1}{2} \cdot \sum_{i=1}^{n-1} (x_i y_{i+1} - x_{i+1} y_i) \tag{5.2}$$

Depending on which orientation is chosen for either path, the distance value can differ (see Figure 5.4 for an example). However, this issue can be resolved easily by fixing source and target for both the mapped and reference path beforehand, and restricting the mapping process to the remaining path nodes.

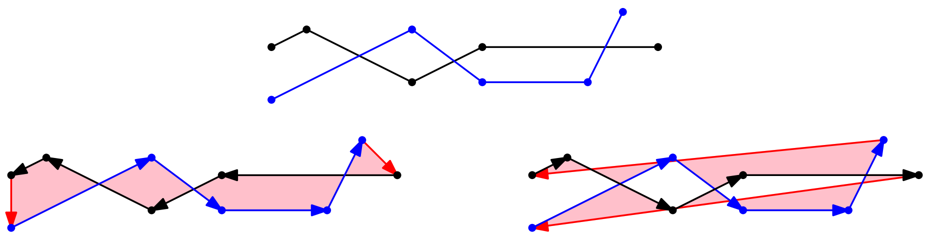


Figure 5.4.: Two paths and possible path polygons according to orientation.

As before, zero distance does not imply equality (cf. Figure 5.5), since nodes and coincident edges can be added and shifted along the path without influencing the area value. When disregarding nodes, however, two paths with no area between them are at least visually indistinguishable.

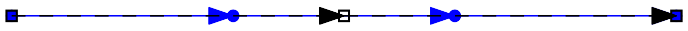


Figure 5.5.: Two paths that are equal according to path polygon distance.

5.2.2. Boundary Polygon

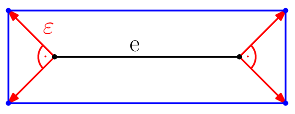
In order to reduce the number of generalization vertices that need to be examined when mapping a path, we compute a geometrical boundary around the original graph. Only nodes and edges inside this bounding structure are considered close enough to to be potentially useful as part of the new path. Its size and shape therefore determine whether a reasonably similar path can be found in the generalized graph or not.

A simple polygon boundary can be obtained by constructing a bounding rectangle of distance ε for every path edge and taking their union, as shown in Algorithm 5.2.1. The union operation is implemented as part of the GPC Library [Mur98], which uses a modified version of the Vatti clipping algorithm [Vat92] with quadratic running time.

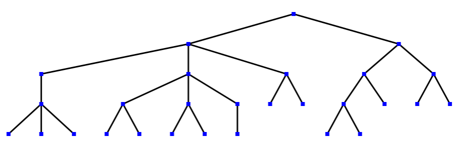
Algorithm 5.2.1: Bounding Polygon

```
Input: path $P = (v_1, ..., v_n)$, distance $\varepsilon$
Output: bounding polygon $B$
for $i = 1$ to $n - 1$ do
    $B_i \leftarrow$ BoundingRectangle($(v_i, v_{i+1})$, $\varepsilon$);
end
$B \leftarrow$ Union($B_1$, ..., $B_{n-1}$);
return B;
```

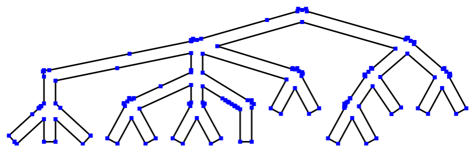
This polygon ensures that only nodes and edges that deviate by at most ε are part of the mapped path. A binary search on the parameter ε finds a path with minimal distance from any of its nodes to the original path. Figure 5.6 shows an example of the construction.



(a) Bounding Rectangle



(b) Binary Tree



(c) Bounding Polygon

Figure 5.6.: Bounding Rectangle and Bounding Polygon.

5.2.3. Finding Mapped Paths

The subset of generalization nodes for the path to be mapped on is decided by the boundary polygon. While this set is generally quite small, the number of possible paths to consider can grow exponentially with its size, in the worst case. Finding an optimal path with respect to a given distance measure is therefore not trivial and leads to the following optimization problem.

Problem 5.2 (Path Mapping) *Given a graph $G = (V, E)$, a path $P = (v_1, ..., v_n)$ in G, a node set V' and vertices v'_1, v'_m. Find a path $P' = (v'_1, ..., v'_m)$ on V' such that $d(P, P')$ is minimal for a given distance measure d.*

The complexity of this problem depends on the chosen distance measure d, as well as properties of both the reference path P and mapped path P'. A special instance are monotone paths, i.e., paths for which the order of nodes does not differ from that of their projection onto the straight line between their source to target. Identifying monotone paths are a somewhat natural approach when mapping, since they generally catch the human eye first and represent important connections for many graphs, such as road networks.

5.2.3.1. Monotone Paths

We present a polynomial time algorithm for the following restricted variant of the path mapping problem introduced above, which requires both the reference and mapped path to be monotone.

Problem 5.3 (Monotone Path Mapping) *Given a graph $G = (V, E)$, a monotone path $P = (v_1, ..., v_n)$ in G, a node set V' and vertices v'_1, v'_m. Find a monotone path $P' = (v'_1, ..., v'_m)$ on V' such that $d(P, P')$ is minimal for a given distance measure d.*

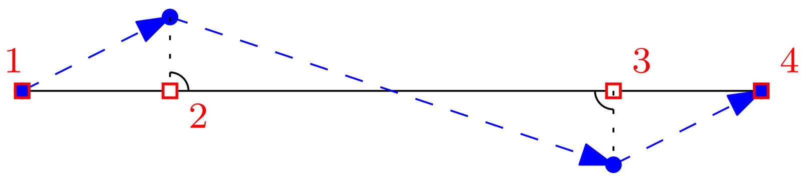


Figure 5.7.: Monotone path.

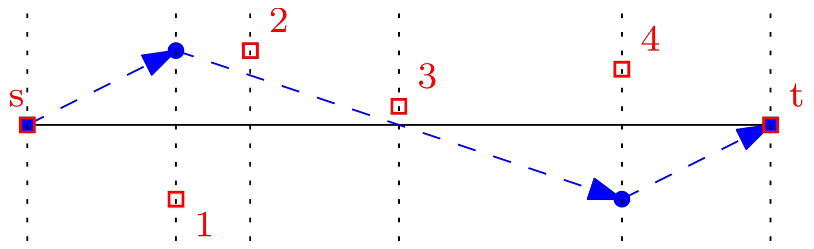
A path is here defined to be monotone if and only if the order of its nodes coincides with that of their projections onto the straight line through path source and target. Figure 5.7 gives an example.

Algorithm 5.2.2: Monotone Path Mapping

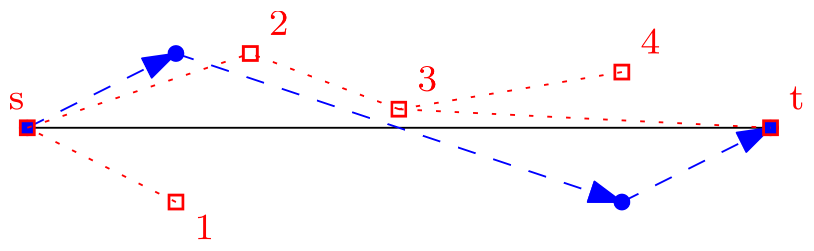
Input: Ordered node set $V = (v_1, ..., v_n)$, monotone reference path P^{ref}
Output: Monotone path P on V with minimal distance $d(P, P^{\text{ref}})$

```
foreach $v \in V$ do
    MinDistance($(v_1, ..., v)$, $P^{\text{ref}}$) $\leftarrow \infty$;
    Predecessor (v) $\leftarrow v_1$;
end
MinDistance($(v_1, ..., v_1)$, $P^{\text{ref}}$) $\leftarrow d_R((v_1, v_1), P^{\text{ref}})$;
foreach $v_i \in V$ in order do
    foreach $j < i$ do
        distance $\leftarrow$ MinDistance($(v_1, ..., v_j)$, $P^{\text{ref}}$) $\circ\ d_R((v_j, v_i), P^{\text{ref}})$;
        if distance $<$ MinDistance($(v_1, ..., v_i)$, $P^{\text{ref}}$) then
            MinDistance($(v_1, ..., v_i)$, $P^{\text{ref}}$) $\leftarrow$ distance;
            Predecessor $(v_i) \leftarrow v_j$;
        end
    end
end
return ReconstructPath(Predecessor);
```

Algorithm 5.2.2 finds an optimal solution using dynamic programming for both the polygon area and Hausdorff distance measures. The $\circ$ operator denotes addition and maximum of, respectively. Because monotonicity is required for the mapped path, the nodes in V can first be arranged to match their relative position in the sequence of their projections onto the reference path's source-target line. When examining V in this order optimal prefix solutions can be computed for each node in succession by determining the best predecessor. The distance function d_R denotes the restriction of the path polygon area measure to path segments, as indicated by the dotted lines in Figure 5.8(a) for an example path. Figure 5.8(a) shows all optimal prefix paths with respect to minimum polygon area distance for the same example.



(a) Computation



(b) Solution

Figure 5.8.: Monotone to monotone path mapping algorithm example.

Theorem 5.2 *Monotone path mapping can be solved in $\mathcal{O}(|V|^2 \cdot |P^{\mathrm{ref}}|)$ time, where $|P^{\mathrm{ref}}|$ is the length of the reference path.*

Proof

We use Algorithm 5.2.2. Sorting the nodes according to their projection takes $\mathcal{O}(|V| \cdot \log |V|)$ time. For each node u the prefix solution for each previous node v has to be checked, which requires computing the distance $d_R((v,u), P^{\mathrm{ref}})$. This takes time at most $|P^{\mathrm{ref}}|$. Total running time is therefore in $\mathcal{O}(|V|^2 \cdot |P^{\mathrm{ref}}|)$.

□

While monotone path mapping is in $\mathcal{P}$, complexity for the general problem can be expected to be higher, since locally subobtimal path segments may be part of an optimal solution if the restriction on monotonicity of both paths is dropped. As dynamic programming can not be applied in this case, there is no immediately obvious solution for finding an optimal path, other than checking all possible paths.

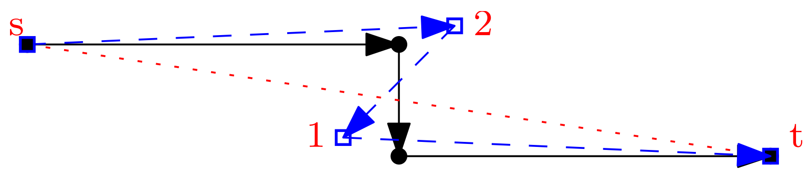


Figure 5.9.: Optimal mapped path is not monotone.

However, there are common cases where the optimal mapped path is not monotone, even if the reference path has this property (cf. Figure 5.9 for an example). We therefore pursue a different strategy.

5.2.3.2. BFS Paths

A simple visibility constrained breadth-first search (BFS), as described in Algorithm 5.2.3, finds a path with minimal hop count. Only nodes inside a bounding polygon are admissible for the search, and a target cannot be reached by the BFS if the corresponding edge would cross the boundary.

Algorithm 5.2.3: Path Finding BFS

Input: path $P = (v_1, ..., v_n)$, parameter ε, node set V', $u', v' \in V'$
Output: mapped path $P' = (u', ..., v')$

```
1 $B \leftarrow$ BoundingPolygon($P$, $\varepsilon$);
2 $G' \leftarrow (V', \{(v'_1, v'_2) \in V' \mid$ EdgeInPolygon($(v'_1, v'_2)$, $B$)$\})$;
3 $P' \leftarrow$ BFSPath($u'$, $v'$, $G'$);
4 return P';
```

Figure 5.10 illustrates the process. Our implementation precomputes only the node set for the visibility graph, and checks whether an edge is valid for the search on demand.

For the point in polygon check we use the Ray Casting method as described in Haines [Hai94]. The algorithm (cf. Algorithm 5.2.4) counts the number of intersections of an infinite ray, starting from the point in question and directed along the y-axis, with the polygon. This number is odd if and only if the point is included in the polygon.

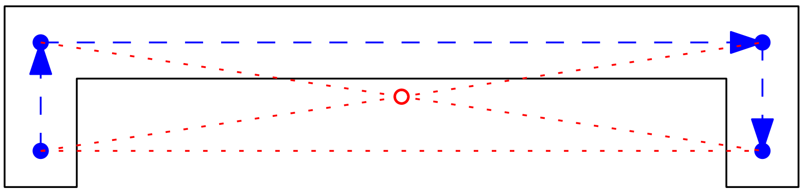


Figure 5.10.: Visibility constrained BFS.

Algorithm 5.2.4: Point in Polygon

Input: Polygon $P = (p_1, ..., p_n)$, point p
Output: True if p is in P, false otherwise

```
1 result ← false;
2 foreach edge $(p_i, p_{i+1}) \in P$ do
3     if $(\mathtt{y}(p_i) > \mathtt{y}(p)) = (\mathtt{y}(p_{i+1}) > \mathtt{y}(p))$ then continue;
4     if $\mathtt{x}(p) < \mathtt{x}(p_i) + \frac{(\mathtt{x}(p_{i+1}) - \mathtt{x}(p_i))(\mathtt{y}(p) - \mathtt{y}(p_i))}{\mathtt{y}(p_{i+1}) - \mathtt{y}(p_i)}$ then continue;
5     result ← not result;
6 end
7 return result;
```

In order to decide whether an edge lies inside a given polygon, first the point in polygon test is performed for both its endpoints. It is then tested for intersection with all polygon edges as follows. The intersection point for two lines can be computed with Formula 5.3.

$$\begin{pmatrix} x(i) \\ y(i) \end{pmatrix} = \begin{pmatrix} x(s_1) \\ y(s_1) \end{pmatrix} + u_1 \begin{pmatrix} x(t_1) - x(s_1) \\ y(t_1) - y(s_1) \end{pmatrix} = \begin{pmatrix} x(s_2) \\ y(s_2) \end{pmatrix} + u_2 \begin{pmatrix} x(t_2) - x(s_2) \\ y(t_2) - y(s_2) \end{pmatrix} \tag{5.3}$$

To determine whether two line segments $\ell_1 = s_1 + u_1(t_1 - s_1)$ and $\ell_2 = s_2 + u_2(t_2 - s_2)$ intersect, it suffices to compute u_1 and u_2 according to Formula 5.4 and 5.5, respectively, and check if both values lie in the interval $[0, 1]$.

$$d = (y(t_2) - y(s_2))(x(t_1) - x(s_1)) - (x(t_2) - x(s_2))(y(t_1) - y(s_1))$$

$$u_1 = \frac{(x(t_2) - x(s_2))(y(s_1) - y(s_2)) - (y(t_2) - y(s_2))(x(s_1) - x(s_2))}{d} \tag{5.4}$$

$$u_2 = \frac{(x(t_1) - x(s_1))(y(s_1) - y(s_2)) - (y(t_1) - y(s_1))(x(s_1) - x(s_2))}{d} \tag{5.5}$$

Theorem 5.3 *Algorithm 5.2.3 has a running time of $\mathcal{O}(|V|^2 \cdot |B|)$, where $|B|$ denotes the size of the bounding polygon.*

Proof

In order to determine the subset of nodes admissible for the BFS, the point in polygon check is performed for each node. During each step of the BFS, the visibility of any node in the polygon is determined using the edge in polygon test. Both polygon inclusion tests are in $\mathcal{O}(|B|)$. This leads to a worst case complexity of $\mathcal{O}(|V|^2 \cdot |B|)$ for the entire algorithm. □

While BFS paths do not necessarily have any particularly desirable property of their own, they are fast to compute and provide the basis for a quick heuristic solution (cf. Algorithm 5.2.5) to finding an optimized path, by recursively expanding edges whenever this results in an improvement. For each path edge in order, the detour over every reachable node that is not yet part of the path is computed. If the rerouted path is determined to be closer to the reference path, the path edge is replaced by the two alternative edges and optimization continues with the first of them. Figure 5.11 gives an example.

Algorithm 5.2.5: Path Optimization Heuristic

Input: Node set V, path $P = (v_1, ..., v_n)$ on V, reference path P^{ref}
Output: Path $P' = (v'_1, ..., v'_m), m \geq n$ on V

```
modified ← $false$;
$i \leftarrow 1$;
while $i < m$ do
    foreach $v \in V$ do
        $P' \leftarrow (v_1, ..., v_i, v, v_{i+1}, ..., v_n)$;
        if $d(P', P^{\text{ref}}) < d(P, P^{\text{ref}})$ then
            modified ← $true$;
            $P \leftarrow P'$;
            break;
        end
    end
    if modified = $true$ then
        modified ← $false$;
    else
        $i \leftarrow i + 1$;
    end
end
return P;
```

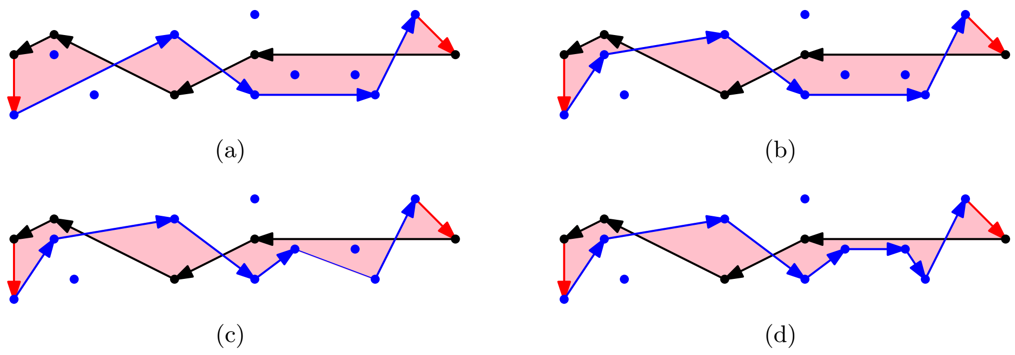


Figure 5.11.: Example of heuristic optimization by path expansion.

Since the algorithm never again looks at nodes it has determined to be part of an optimal path, it fails in cases where the best solution contains a suboptimal path segment. Also, none of the original BFS path's nodes are ever discarded and optimal solutions these are not part of can therefore never be reached, as shown in Figure 5.12.

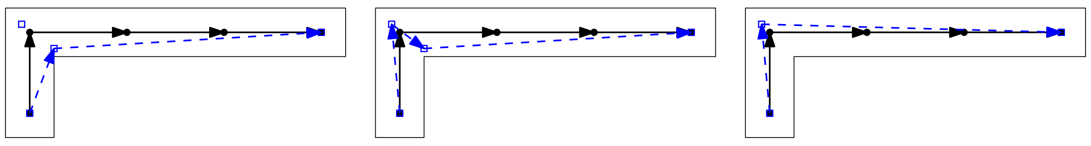

(a) BFS Path (b) Heuristic Solution (c) Optimal Solution

Figure 5.12.: Heuristic optimization fails, since the original BFS node can never be discarded.

5.2.4. Selecting Paths To Map

Path selection takes two factors into account. First and foremost are desired properties of the paths themselves, which should fulfill the following desiderata.

- **Length** Paths should not be too short, in order to be meaningful as a representation of the graph's structure. On the other hand, the chance that no mapped path can be found at all, or is sub-optimal, grows with path length.
- **Simplicity** Loops in the reference path are undesirable, as are multiply contained edges, since they cause unnecessary computational overhead and complicate mapped path finding.
- **Straightness** Zigzagging and self-crossing paths are detrimental to the bounding structure approach. More or less straight paths are preferable.

Second, selected paths should also be disjoint, since otherwise a lot of the mapping is redundant and cluttering due to erroneously mapped edges may occur. An edge's mapped equivalents usually conflict, as they are likely to be very similar and therefore hard to distinguish. Ideally, only a path cover of the original graph is mapped. Determining appropriate covering paths is generally computation-intensive, however. For shortest paths finding a minimum number is NP-hard (cf. Boothe et al. [BDFP07]).

5.2.4.1. All Pairs

Mapping paths for all pairs of nodes in the original graph is usually not feasible. The computational effort is high and increasingly redundant for higher abstraction levels. A more viable strategy is to only map paths for all node pairs in the generalized set. Corresponding source and target nodes in the original graph can be found by a KD-Tree (cf. [dBvKOS00]) query with radius $\varepsilon_{\text{diff}}$. A lot of edges can be expected to be mapped multiple times as part of different paths.

5.2.4.2. Shortest Path Graph Cover

A simple shortest path graph cover can be computed as described in Algorithm 5.2.6, by selecting random edges that are as of yet uncovered and mapping the entire shortest path tree from one of its end nodes. However, edges are covered by multiple paths and a lot of the paths that are mapped may be short.

5.2.4.3. Priority Based

Algorithm 5.2.7 shows a priority based strategy, which selects source and target for the mapped path in the generalized node set according to desired properties of nodes in the generalization, such as degree. The two nodes with the largest gap to their target value are chosen as source and target nodes of the next path to be mapped. This approach can be used to address specific issues, such as disconnectedness.

Algorithm 5.2.6: Map Shortest Path Graph Cover

```
Input: graph $G = (V, E)$, node set $V'$, mapping m : $V \to V'$
Output: edge set $E'$
$E' \leftarrow \emptyset$;
foreach $e \in E$ do Covered($e$) $\leftarrow$ false;
foreach $e \in E$ do
    if Covered($e$) then continue;
    $u \leftarrow$ SourceNode (e);
    foreach $v \in V$ do
        $P \leftarrow$ ShortestPath($u$, $v$, $G$);
        foreach edge $e_i$ on $P$ do Covered($e_i$) $\leftarrow$ true;
        $P' \leftarrow$ MapPath($P$, $V'$, m($u$), m($v$));
        add $P'$ to $E'$;
    end
end
return $E'$;
```

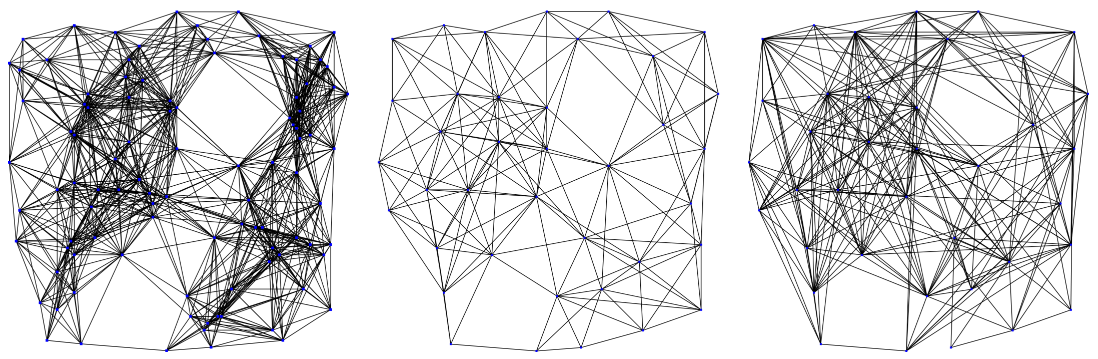
Algorithm 5.2.7: Map Gap Based

```
Input: graph $G = (V, E)$, node set $V'$, gap function Gap : $V' \to \mathbb{Z}$
Output: edge set $E'$
E' $\leftarrow \emptyset$;
while $\exists v' \in V'$ | Gap($v'$) $> 0$ do
    $u' \leftarrow$ node in $V'$ with highest Gap($u'$);
    $v' \leftarrow$ node in $V'$ with next highest Gap($v'$);
    $u \leftarrow$ NearestNeighbour($u'$, $V$);
    $v \leftarrow$ NearestNeighbour($v'$, $V$);
    $P \leftarrow$ ShortestPath($u$, $v$, $G$);
    $P' \leftarrow$ MapPath($P$, $V'$, $u'$, $v'$);
    add $P'$ to $E'$;
    foreach node $v_i'$ on $P'$ do update Gap($v_i'$);
end
return E';
```

5.2.5. Evaluation

Like iterative edge insertion, path mapping is most suited for dense graphs, where the considerably less expensive induced mapping is generally too sparse to be of any use. Filtering by average edge distance works well as a post-processing step, since edges tend to be similar if too many are inserted. The most useful application for path mapping are low to medium zoom levels, where the computation is significantly faster than iterative edge insertion. For higher levels of abstraction the results tend to be poor and do not justify the high computational cost.

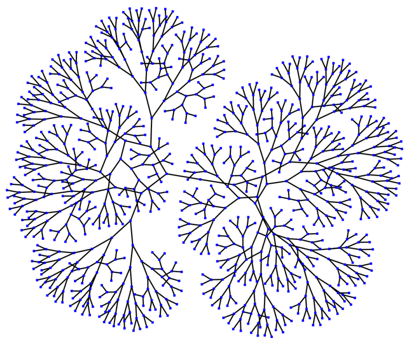
Figure 5.13 shows a comparison on a unit disk graph with 100 nodes and a node-to-edge ratio of about 1:10. Edge insertion is necessary here to improve the generalization, since the induced mapping only achieves a much higher node-to-edge ratio of about [1:4]. While path mapping brings some visual cluttering, the result represents both density and structure of the original graph much better.



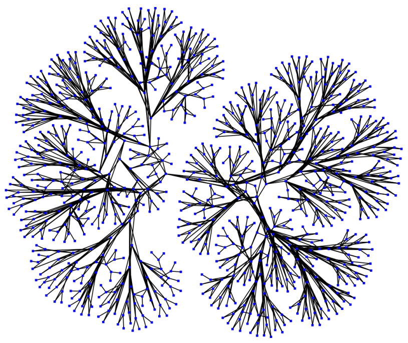
(a) No Generalizer n100 e1012 (b) Induced Mapping n38 e168 l5 (c) Path Mapping n38 e263 l5

Figure 5.13.: Neighborhood contraction with path mapping on the UR30 unit disk graph.

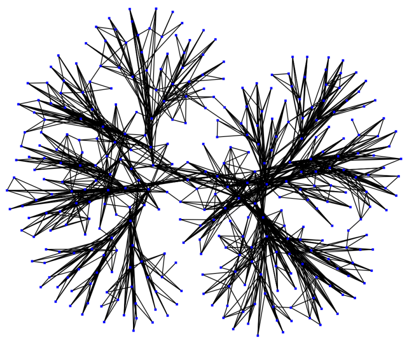
Unfortunately, there are severe issues that still need to be ironed out. Figure 5.14 shows the results of path mapping on the complete binary tree with 1023 nodes and organic layout at various zoom levels. At medium abstraction levels, path mapping manages to capture the tree structure quite well, but unfortunately leads to massive cluttering due to multiply mapped edges. Modifying the path finding algorithm to favor previously inserted edges might solve this problem.



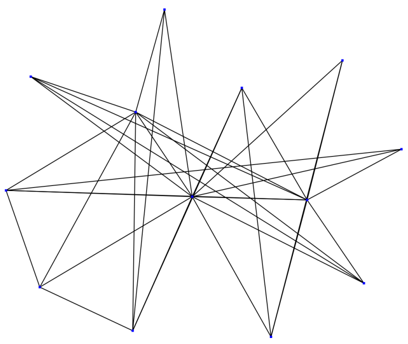
(a) No Generalizer n1023 e1022



(b) NeC Asc n974 e1877 l1



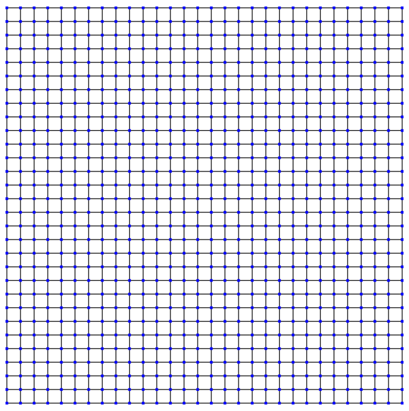
(c) NeC Asc n343 e1482 l2



(d) NeC Asc n13 e34 l5

Figure 5.14.: Neighborhood contraction with path mapping on the complete binary tree.

Path mapping on the regular grid with the original node set (cf. Figure 5.15) shows the inadequacy of the polygon distance measure on its own. The generalized edge set actually contains twice as many edges as the original graph, with no visual distinction between the two whatsoever.


(a) No Generalizer n900 e1740

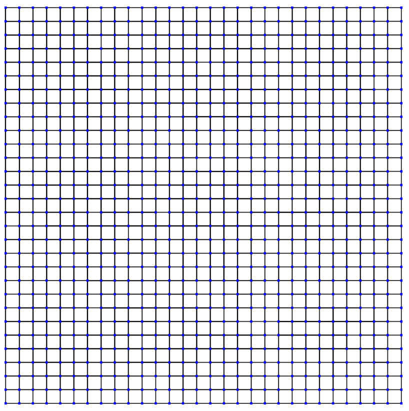

(b) NeC Asc n900 e3036 l1

Figure 5.15.: Neighborhood contraction with path mapping on the regular 30x30 grid.

6. Conclusion

Our goal was automatic generalization of general geometric graphs, a diverse and complex problem. We gave an outline of fundamental subproblems, starting with how the similarity of graphs can even be measured reasonably, and discussed criteria to determine what defines a graph and how to balance visual and structural abstraction. A number of relevant subproblems turned out to be NP-hard, which substantiates the difficulty of the general problem. The evaluation of several basic approaches for graph generalization showed some promising first results, however.

Three polynomial-time algorithms for node set abstraction were introduced, namely edge contraction, node contraction and neighborhood contraction. All of these have flaws, but give an indication of what is important for a good generalization. Problematic are mainly very regular structures. While specific graph characteristics can help generalization, the layout also greatly influences the end result, and good as well as bad instances can be found regardless of graph class. Dense graphs, particularly cliques, require special attention due to their large number of edges.

The induced mapping works surprisingly well in many cases, but is inherently random. Improving this mapping by filtering, if applicable, yields good results and allows for a reasonably fast generalization of sparse graphs. Specialized methods for edge set abstraction turned out to be less successful, however. We discussed two techniques, iterative edge insertion and path mapping, both of which are only feasible for higher zoom levels and can not always compare with the induced mapping.

In general, while the results are far from comprehensive, we were able to identify a number of easier instances that can be solved using basic techniques and give some indication on how to approach a general solution.

Future Work. There is still much left to do, as far as an automatic generalization is concerned we barely scratched the surface. None of the related subproblems have been analyzed exhaustively and not many of the results are really satisfactory.

First and foremost, a feasible measure for the similarity of two geometric graphs is needed. Only few measures have been proposed in literature, none of which are applicable, and the combination of simple terms we considered is sorely lacking.

Another problem that could be investigated further are specific layouting constraints and corresponding node orders to eliminate the degree of randomness that is an issue with all the node set abstraction methods that have been introduced. Computation of a matching layout, which emphasizes important graph properties visually the way other techniques already do for specialized graph instances, might be helpful.

Also, while the iterative insertion method for edge set generalization yielded no usable results, the approach looks promising as only a good target function needs to be found. The same applies for the path mapping technique, which mainly requires a better method for path selection and merits further testing.

Lastly, instances like regular grids demonstrate the importance of a graphs outline and shape, which catch the human eye. It might be worth developing techniques to determine a reasonably complex boundary and map it separately.

Appendix

A. List of Tables

B. List of Figures

Bibliography

[BBKR08] Jörg Bauer, Iovka Boneva, Marcos E. Kurbán, and Arend Rensink, *A modal-logic based graph abstraction*, Proceedings of the 4th international conference on Graph Transformations, Springer-Verlag, 2008, pp. 321–335. 2

[BDFP07] Peter Boothe, Zdeněk Dvořák, Arthur M. Farley, and Andrzej Proskurowski, *Graph covering via shortest paths*, Congressus Numerantium (2007), pp. 145–155. 56

[Bre98] J. E. Bresenham, *Algorithm for computer control of a digital plotter*, Seminal graphics, ACM, 1998, pp. 1–6. 44

[CGK$^+$09] Otfried Cheong, Joachim Gudmundsson, Hyo-Sil Kim, Daria Schymura, and Fabian Stehn, *Measuring the similarity of geometric graphs*, Proceedings of the 8th International Symposium on Experimental Algorithms, Springer-Verlag, 2009, pp. 101–112. 3

[Dav97] Timothy A. Davis, *University of florida sparse matrix collection [online] `http://www.cise.ufl.edu/research/sparse`*, NA Digest, 1997. 6

[dBvKOS00] Mark de Berg, Marc van Kreveld, Mark Overmars, and Otfried Schwarzkopf, *Computational geometry: Algorithms and applications*, second ed., Springer-Verlag, 2000. 56

[EF97] Peter Eades and Qing-Wen Feng, *Multilevel visualization of clustered graphs*, Proceedings of the Symposium on Graph Drawing, Springer-Verlag, 1997, pp. 101–112. 2

[EGK$^+$03] John Ellson, Emden R. Gansner, Eleftherios Koutsofios, Stephen C. North, and Gordon Woodhull, *Graphviz and dynagraph – static and dynamic graph drawing tools*, Graph Drawing Software, Springer-Verlag, 2003, pp. 127–148. 6

[FLM95] Arne Frick, Andreas Ludwig, and Heiko Mehldau, *A fast adaptive layout algorithm for undirected graphs*, Proceedings of the DIMACS International Workshop on Graph Drawing, Springer-Verlag, 1995, pp. 388–403. 1

[GLN02] Joachim Gudmundsson, Christos Levcopoulos, and Giri Narasimhan, *Fast greedy algorithms for constructing sparse geometric spanners*, SIAM Journal on Computing, vol. 31, Society for Industrial and Applied Mathematics, 2002, pp. 1479–1500. 33

[GSSD08] Robert Geisberger, Peter Sanders, Dominik Schultes, and Daniel Delling, *Contraction hierarchies: faster and simpler hierarchical routing in road networks*, Proceedings of the 7th international conference on Experimental algorithms, Springer-Verlag, 2008, pp. 319–333. 18

[Hai94] Eric Haines, *Point in polygon strategies*, Graphics gems IV, Academic Press Professional, Inc., 1994, pp. 24–46. 53

[HL04] Drew Harry and Daniel Lindquist, *Graph abstraction through centrality erosion and k-clique minimization [online]* `http://web.media.mit.edu/~dharry/old_portfolio/projects/files/GraphMinimiz.pdf`, Olin College, 2004. 2

[Hu05] Yifan Hu, *Efficient and high quality force-directed graph drawing*, The Mathematica Journal (2005), pp. 37–71. 6

[KR92] Donald E. Knuth and Arvind Raghunathan, *The problem of compatible representatives*, SIAM Journal on Discrete Mathematics, vol. 5, Society for Industrial and Applied Mathematics, 1992, pp. 422–427. 4

[KRB95] Karlis Kaugars, Juris Reinfelds, and Alvis Brazma, *A simple algorithm for drawing large graphs on small screens*, Proceedings of the DIMACS International Workshop on Graph Drawing, Springer-Verlag, 1995, pp. 278–281. 2

[Lic82] David Lichtenstein, *Planar Formulae and Their Uses*, SIAM Journal on Computing, vol. 11, Society for Industrial and Applied Mathematics, 1982, pp. 329–343. 4

[MPK96] Richard S. Mallory, Bruce W. Porter, and Benjamin J. Kuipers, *Comprehending complex behavior graphs through abstraction*, Tenth international workshop on qualitative physics, AAAI Press, 1996, pp. 137–146. 2

[Mur98] Alan Murta, *A general polygon clipping library [online]* `http://www.cs.man.ac.uk/~toby/alan/software/gpc.html`, University of Manchester, 1998. 50

[QE01] Aaron Quigley and Peter Eades, *Fade: Graph drawing, clustering, and visual abstraction*, Proceedings of the 8th International Symposium on Graph Drawing, Springer-Verlag, 2001, pp. 197–210. 2

[RC05] Davood Rafiei and Stephen Curial, *Effectively visualizing large networks through sampling*, Visualization Conference, IEEE Computer Society, 2005, pp. 48–56. 2

[RW98] R. T. Rockafellar and R. J-B. Wets, *Variational analysis*, Springer-Verlag, 1998. 49

[SSTR93] Manojit Sarkar, Scott S. Snibbe, Oren J. Tversky, and Steven P. Reiss, *Stretching the rubber sheet: a metaphor for viewing large layouts on small screens*, Proceedings of the 6th annual ACM symposium on User interface software and technology, ACM, 1993, pp. 81–91. 2

[Tam97] Roberto Tamassia, *Graph drawing*, Lecture Notes in Computer Science, CRC Press, 1997, pp. 815–832. 1

[Tar75] Robert Endre Tarjan, *Efficiency of a good but not linear set union algorithm*, ACM Journal, vol. 22, ACM, 1975, pp. 215–225. 33

[Vat92] Bala R. Vatti, *A generic solution to polygon clipping*, Communications of the ACM, vol. 35, ACM, 1992, pp. 56–63. 50