Bachelor thesis
Computer Science

Radboud University

Scoring Entity-Relationship Diagrams Drawn by a Computer Algorithm

Author:
Bart van de Put
S4400496

First supervisor/assessor:
prof. dr. F.W. Vaandrager
f.vaandrager@cs.ru.nl

Second assessor:
Dr. Peter Achten
p.achten@cs.ru.nl

April 2, 2018

Abstract

Graph drawing algorithms are used to automatically generate graph drawings. Ideally, these drawings are clear and readable and conform to one or more aesthetic criteria. This research focuses on the drawing of entity-relationship diagrams, which are a type of graph. We provide some insight into existing algorithms and implement metrics for five aesthetic criteria which can be used to score the generated drawings. We use a graph drawing algorithm based on one by Tamassia et al. and evaluate the generated drawings. We find that the algorithm scores really high for the *crossings* aesthetic. However, the other criteria which we have measured could see some improvement.

Contents

Chapter 1

Introduction

More and more manual tasks are becoming automated by clever computer programs and algorithms. This can save a lot of time and money. Even (parts of) the task of building software is being automated. A part of software development that, when automated, can save a large amount of time is the drawing of entity-relationship diagrams. This is difficult to do automatically, because you want to make your diagrams as clear and readable as possible. However, a computer does not know what is clear and readable to humans.

Entity-relationship diagrams are a type of graph. This means that to automatically draw an entity-relationship diagram we need a good *graph drawing algorithm*. A lot of work has been done in the field of *graph drawing algorithms*. See for instance the bibliography put together by Di Battista [7]. This bibliography shows that only a very small number of these algorithms were developed with entity-relationship diagrams in mind.

We want to represent graphs or diagrams in a readable and understandable way, and these *drawing algorithms* are often based on some *aesthetic criteria*. However, as pointed out by Purchase [19], the measurement of such criteria is often done informally and may differ between algorithms. In her paper she introduces formal measurements for seven commonly used aesthetic criteria.

In this thesis we will try to answer the following question: Can an algorithm produce clear and readable E/R diagrams and how can this be measured? To answer this we will give an overview of important work that has been done in the field of graph drawing, specifically regarding aesthetic criteria. We will implement metrics for a few important aesthetic criteria to enable us to score diagrams. We will then use an algorithm to generate E/R diagrams and score these drawings with the implemented criteria. The results we get can be used by other researchers to compare their algorithms with the one used in this research. Alternatively the implementations of the metrics for the aesthetic criteria can be used to compare their work with

related algorithms.

This thesis starts with some preliminary knowledge needed to understand the rest of the work (chapter 2). In chapter 3 an overview of important previous work is given. Implementations for metrics of aesthetic criteria are shown in chapter 4. The algorithm we used to generate E/R diagrams for testing is described in chapter 5. Following this we will present the results from testing the diagrams against the implemented metrics. Chapter 7 contains our conclusions and some future research.

Chapter 2

Preliminaries

2.1 Graph theory

There is a lot of literature covering graph theory. Within this literature you can find all kinds of slightly different ways to describe and represent graphs. We use the following description that can be found in 'Graph Algorithms', by Shimon Even [10]. A graph consists of a set of *nodes* (also called *vertices*) $V = \{v_1, v_2, ...\}$ and a set of *edges* $E = \{e_1, e_2, ...\}$. Each edge has two *endpoints*, which are nodes. For example, edge $e_1 = (v_1, v_2)$ means that the endpoints of e_1 are the nodes v_1 and v_2. Note that two distinct edges can have the same endpoints. And in some cases both endpoints of an edge are the same node. Such an edge is called a *self-loop*. An example of this is *edge* e_1, from the graph depicted in Figure 2.1.

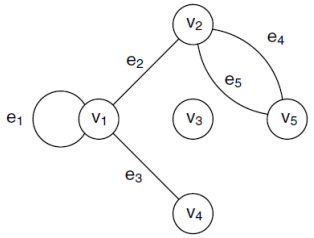

Figure 2.1: Example of a graph.

We chose this particular definition of graphs because it corresponds with the way we work with graphs in the `Open Graph Drawing Framework` [6], which is the framework used to implement our algorithm. Here a graph object also contains the set of nodes and the set of edges. Edges have references to their endpoints, these points are called *source* and *target*.

2.1.1 Graph drawing

When drawing graphs, nodes are usually represented by dots, circles, boxes or some other shape, and edges connect these shapes with lines drawn between them. There are various *graphic standards* for the representation of graphs in the plane. In [7], we are given some examples. A commonly adapted standard is a drawing where an edge is represented by a polygonal chain. This is called a *poly-line* drawing. This standard comes in two variants. (See Figure 2.2).

1. A *straight-line* drawing maps each edge into a straight-line segment.
2. An *orthogonal* drawing maps each edge into a chain of horizontal and vertical segments.

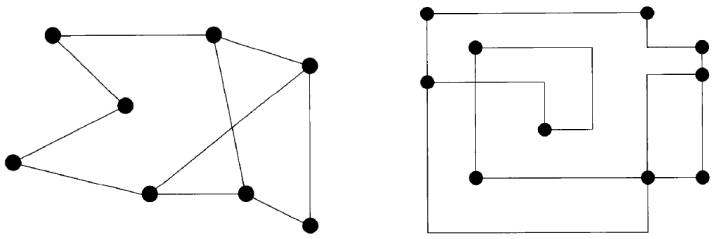

Figure 2.2: Straight-line drawing and orthogonal drawing.

Drawing a graph by hand is relatively easy for small graphs. As a graph gets larger this task becomes more difficult and time consuming. At a certain point drawing a graph by hand becomes unfeasible. This is where graph drawing algorithms are very helpful. They receive as input a combinatorial description of a graph and produce a drawing of that graph conform a given graphic standard.

The goal of a graph drawing algorithm is to produce a readable drawing, as the usefulness of a drawing depends on the *readability*. Readability can be defined as the capability to convey the meaning of a diagram quickly and clearly [7]. Purchase [20] describes it as follows: a graph drawing should help the user to understand and remember the information being visualized. To achieve better readability, graph drawing algorithms try to optimize one or more *aesthetics*. Some examples of such aesthetics are:

- Minimizing edge crossings.
- Minimizing bends in edges.
- Maximizing symmetry.

2.1.2 Planar graph

A graph is *planar* if it can be embedded in the plane. This means that it can be drawn in such a way that edges may only intersect in their endpoints, the nodes. Planar graphs can be characterized with *Kuratowski's theorem*. This theorem states that a finite graph can only be planar when it does not contain a *subgraph* that is a *subdivision* of K_5 (complete graph of five nodes) or $K_{3,3}$ (complete bipartite graph of six nodes, where three nodes connect to the other three).

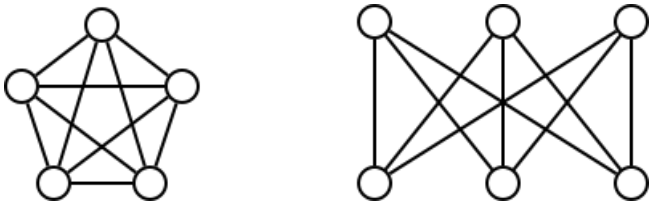

Figure 2.3: K_5 and $K_{3,3}$

A subdivision of a graph is a graph that is the result of a sequence of *edge subdivisions*. Edge subdivision is an operation where an edge $\{v1, v2\}$ is deleted from a graph and two new edges $\{v1, v3\}$ and $\{v3, v2\}$ are added as well as the new node $v3$. This basically means splitting an edge into two pieces by introducing a new node, as illustrated in Figure 2.4.

Figure 2.4: Before (left) and after (right) edge subdivision.

Maximum planar subgraph

When working with a non-planar graph, it is often useful to find a planar subgraph. The `maximum` planar subgraph problem [4], is defined as follows: given a graph G, find a planar subgraph of G with the maximum number of edges.

The maximum planar subgraph problem is known to be NP-complete. This means there exists no known algorithm that can solve this problem in polynomial time. However, there are various approximation algorithms for this problem. One of these is finding a `maximal` planar subgraph: A subgraph G' of a non-planar graph G is a maximal planar subgraph if G' is planar and adding any edge not present in G' results in a non-planar subgraph of G [15].

2.2 Entity-relationship diagrams

An entity-relationship diagram (E/R diagram) is a visual representation of an entity-relationship model (E/R model). E/R models were introduced in 1976 by Chen [5]. They are used to describe relations between different entities within a data set. E/R diagrams are often used as a tool for database design, because they represent the structure of a relational database very well.

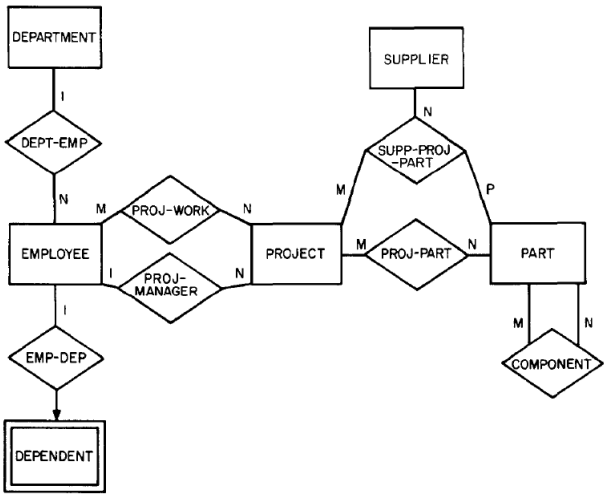

Figure 2.5: Example of an E/R diagram.

In his book on database systems, Garcia-Molina [12] describes them as follows: in an *entity-relationship model*, the structure of data is represented graphically by an *entity-relationship diagram* using three basic elements:

1. entity sets,
2. relations,
3. attributes.

2.2.1 Entity sets

An entity is some object and an entity set is a collection of similar entities. In his book, Garcia-Molina uses the example of a database with movies, stars in the movies and the studios that produce movies. A movie is an entity and the set of all movies is an entity set. Likewise, stars and studios are entity sets.

2.2.2 Relations

Relations describe a connection between two (or sometimes more) entity sets. Following the example from the book, the relation *Stars-in* connects stars to movies. The intent of this relation is that a movie entity m is related to a star entity s if s appears in m.

2.2.3 Attributes

Attributes, associated with entity sets are properties of entities of that set. For example the entity set *Movies* might have the attributes *title* and *length*.

2.3 Entity-relationship diagram drawing

There exists no single 'correct' way of drawing E/R diagrams. You can find different graphical representations of E/R models. However, all of these are some form of a graph in a plane. The difference lies in what parts are drawn and how they are drawn. There is the original notation, as introduced by Chen [5]. In his notation, entity sets are drawn as rectangles and relations have a diamond shape. Entity sets are connected to relations with lines. This can be seen as the entity sets and relations being the nodes of a graph, and the lines between them the edges. In the original paper, Chen does not mention a specific shape for attributes, but when talking about *Chen's notation*, they are often drawn with an oval shape and connected to entity sets (or relations) with lines. See Figure 2.5 for the example diagram from Chen's paper.

Another notation type is the SSADM style [25]. With this style of notation, relationships are drawn as lines, no diamond shapes are used. Another difference with Chen's notation style is the way in which the cardinality of relationships is indicated. For our E/R diagram drawings we will use the SSADM notation. We will not draw attributes, because a lot of attributes can make an E/R diagram look unclear. A common way to include attributes is to put them within the rectangle of an entity set. However, we will omit them completely as we want to focus on the positioning of nodes and edges.

2.4 Open Graph Drawing Framework

For all implementations we are using the *Open Graph Drawing Framework* (OGDF) [6]. This is a C++ library of algorithms and data structures for graph drawing. We chose this framework because it contains implementations for a lot of graph drawing algorithms that we might want to use. Some of these implementations are rather complex and would take a lot of time to implement ourselves.

Chapter 3

State of the art

3.1 Graph drawing algorithms

As stated before, a lot of work has already been done in the field of automated graph drawing [7]. Because drawing an E/R diagram is basically drawing a specific type of graph, it is good to know what types of algorithms for graph drawing exist. The first step in answering our research question is to get to know these existing algorithms. We will describe the basic ideas and workings of some important types, or categories, of graph drawing algorithms.

3.1.1 Force-directed graph drawing

A commonly used type of algorithm is called *force-directed graph drawing*. The idea stems from a paper by Eades [8] and is as follows. Each edge is replaced with a spring to form a mechanical system. The nodes are then placed in some initial layout and let go so that the spring forces move the system to a state of minimal energy. Algorithms based on this idea are also called *spring embedders* and belong to the *straight line* graphic standard. Different versions that use the same basic idea have been proposed over the years. As mentioned by Fruchterman and Reingold [11], force-directed algorithms perform well with the following aesthetic criteria:

- Distributing vertices evenly.
- Keeping edge lengths uniform.
- Reflecting inherent symmetry.

An example of a spring embedder can be seen in Figure 3.1. Each edge is modeled as a spring and after applying the appropriate forces a layout is formed.

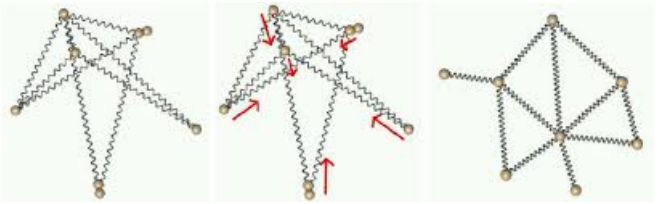

Figure 3.1: Spring embedder

3.1.2 Planar graph drawing

An aesthetic criterion that is often associated with appealing and readable graphs is the following:

- Minimizing edge crossings.

Because of this, a common theme within graph drawing algorithms is finding a planar representation of a graph. The first step in finding a planar representation is to test whether a given graph is planar. The first algorithm to be able to do this in linear time was given by Hopcroft and Tarjan [14]. After determining that a graph is indeed planar the graph can than be embedded in the plane. An example of a planar embedding algorithm is one introduced by Boyer and Myrvold [2]. Their algorithm uses *depth-first search* to define an order in which to add edges to embedded *biconnected components*. Partial embedded connected components are merged to form a complete planar embedding.

To be able to apply a planar graph drawing algorithm to a *non-planar* graph, you first have to *planarize* that graph. Planarization of a non-planar graph is done in two steps:

1. Finding a planar subgraph.
2. Reinserting removed edges.

Ideally we want to find a maximum planar subgraph. However, as this problem is NP-complete, this is commonly approximated by finding a maximal planar subgraph (an example algorithm is presented in [3]). When we have a planar subgraph, some edges from our original graph are removed. We reinsert these edges and introduce dummy nodes at the crossings to make sure we keep the graph planar. These dummy nodes can be removed after a planar graph drawing algorithm has been executed.

3.1.3 Orthogonal graph drawing

As discussed before, there is a distinction between straight-line and orthogonal graph drawings. In [24], Tamassia et al. present a general approach

for creating an orthogonal graph drawing. Their approach consists of the following steps:

1. Planarization:
   The first step is finding a planar representation, if the graph is non-planar, this means first planarizing it as described in chapter 3.1.2.

2. Orthogonalization:
   In this step the planar representation is transformed into an orthogonal representation.The sequence of angles along each edge is specified.

3. Compaction:
   The final step determines the metrics of the drawing, the size of the area used and the length of edges.

An algorithm that follows this approach is presented by Batini et al. [1]. This algorithm is one of the very few that is specifically designed with E/R diagrams in mind. They assume that the criteria listed below make a pleasant E/R diagram drawing. As these are assumptions, they acknowledge that research has to be done to test their relevance in a real scenario.

1. Minimization of crossings between edges.

2. Minimization of the global number of bends in (edge) lines.

3. Minimization of the global length of edges.

4. Equality of the length of edges between relationship boxes and related entity boxes.

3.2 Aesthetic criteria

Basically every paper that introduces a graph drawing algorithm mentions some aesthetic criteria. However, the choice of criteria is not often substantiated with research or proof. Also, often no method of measuring how well a certain criteria has been accomplished is given. Research has been done to validate some criteria and to find out which criterion has the most impact on the readability of graphs. We give an overview of these studies.

The first study to address the effect of aesthetics on the general understanding of graphs is by Purchase et al. [20] and tries to validate three aesthetics. This was done by performing empirical studies of human understanding of graphs drawn using various aesthetics. The aesthetics were formulated as hypotheses.

- `bends`
  Increasing the number of arc (edge) bends in a graph drawing decreases the understandability of the graph.

- `crossings`
  Increasing the number of arc (edge) crossings in a graph drawing decreases the understandability of the graph.

- `symmetry`
  Increasing the local symmetry displayed in a graph drawing increases the understandability of the graph.

Subjects had to answer graph-theoretic questions about experimental graphs within a certain time limit. The study confirms the first two hypotheses. The results show that minimizing edge bends and edge crossings do indeed help with understanding a graph. The third hypothesis is not confirmed, the results are inconclusive.

Another study trying to improve upon these results was done by Purchase [18]. She claims this study is superior to the previous one, because five aesthetics are considered (instead of three), attempts were made to place a priority order on the relative importance of the aesthetics, and the experiments were run online. Another important improvement is the fact that both the time taken to complete questions about a drawing as well as the number of errors were measured. The previous study only measured the amount of errors. Two more hypotheses were added to the previously mentioned three.

- `orthogonality`
  Fixing nodes and edges to an orthogonal grid increases the understandability of the graph.

- `angles`
  Maximizing the minimum angle between edges leaving the nodes in a graph drawing increases the understandability of the graph.

The results of this study show that the `crossings` aesthetic affects graph readability the most. The `bends` hypothesis is supported for the time that was needed to answer the questions and the `symmetry` hypothesis is supported in regard to the number of errors made. The `orthogonality` and `angles` criteria had no impact on the subjects' graph reading.

These studies both tried to validate aesthetics for the understanding of general graphs. While E/R diagrams are a type of graph, this does not necessarily mean that these results also apply to E/R diagrams. No comparable empirical study has been done to determine the impact aesthetics have on E/R diagrams. However, such a study has been done for *UML class diagrams* [21], which has some similarities to E/R diagrams. As the previous studies have already shown that the edge crossings aesthetic has impact on the understandability of a graph, this was not one of the aesthetics in this research. The first part of the study included the following aesthetics:

- minimize bends,
- node distribution,
- uniform edge lengths,
- (consistent) direction of flow,
- orthogonality.

After this first part a second experiment was done, and two more heuristics were added based on the results from the first part:

- edge length,
  Edge lengths should be short, but not too short.
- symmetry.

After conducting two experiments, little to no concrete results were obtained. It seemed that only the bends criterion had a little impact on the understandability of a UML class diagram. But even that may be due to other factors influencing the diagrams that were used for this experiment.

While there has not been a study on the effect these criteria have on E/R diagrams, research has been done on the effect different E/R diagram notations have on the understandability. In [22], Purchase et al. conducted an empirical study to find out whether `Chen` [5] or `SSADM` [25] E/R diagrams are easier to comprehend. See Figure 3.2 for a comparison of both notations.

The subjects' task was to match a given textual specification against a set of diagrams. They were also asked whether or not each diagram correctly represented the textual specification. Both correct and incorrect diagrams from each of the notations could be found in the set of diagrams. The number of errors and the time it took to answer were measured. Subjects were also asked which notation they preferred for the representation of cardinality, participation, and relationships and for the overall diagram.

The study shows that the SSADM notation is faster overall, and also faster for cardinality identification and relationship naming operations. Overall, the SSADM notation was slightly preferred, and significantly preferred regarding the representation of cardinality. These results show that the SSADM notation is easier to understand than the Chen notation.

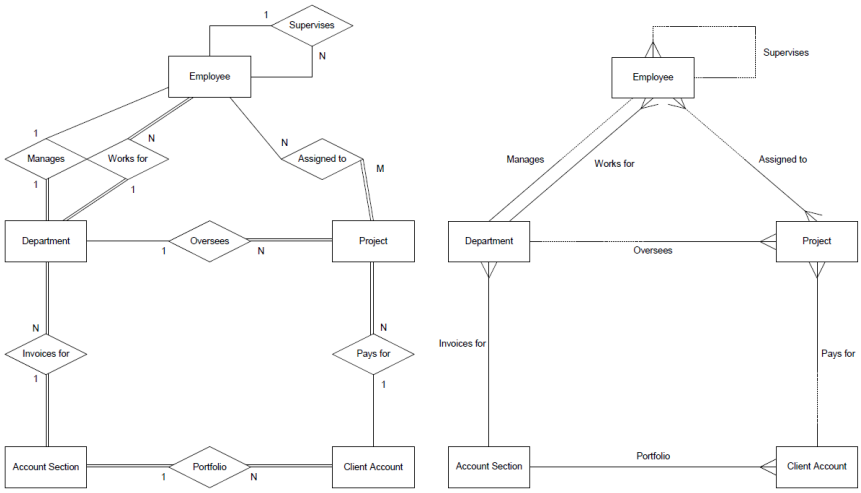

Figure 3.2: left: Chen notation, right: SSADM notation

Chapter 4

Measuring criteria

Aesthetic criteria are a common theme when discussing graph drawing algorithms. However, newly introduced algorithms often do not include methods to measure how well a certain drawing has incorporated a criterion. The criteria tend to be mentioned informally and there is no standard way to measure them. This problem has been identified and addressed by Purchase [19]. She introduces metric formulae for seven aesthetic criteria. These formal, objective metrics are scaled to return a value between 0 and 1 for each of the criteria.

We have selected four out of the seven formulae and have implemented them. Three were left out because they don't apply to E/R diagrams. We have added one formula that was not included in the original work. We will now describe the formulae (as presented by Purchase) and how we implemented them.

4.1 Bends promotion

For some of the calculations, an auxiliary graph drawing needs to be defined. A graph drawing $D(G)$ with polyline edges is transformed into a drawing $D'(G)$ with only straight-line edges. This is done by turning all bend points in $D(G)$ into nodes in $D'(G)$ and is therefore called *bends promotion*. Figure 4.1 shows the code used to implement bends promotion. We iterate over all bend points and use edge subdivision to introduce a new node and split the edge into pieces. When we want to keep our original graph, we call this function on a copy of our graph.

```
void BendPromotion(Graph& G, GraphAttributes& GA) {
    List<edge> edges;
    G.allEdges(edges);

        while (!edges.empty()) {
        edge e = edges.popFrontRet();
        DPolyline bends_e = GA.bends(e);
        node s = e->source();
        node t = e->target();

        while (!bends_e.empty()) {
                DPoint p = bends_e.front();
                node n = G.newNode();
                GA.x(n) = p.m_x;
                GA.y(n) = p.m_y;
                edge e_ = G.newEdge(s, n);
                s = n;
                bends_e.popFront();
        }
        edge e_ = G.newEdge(s, t);
        G.delEdge(e);
    }
}
```

Figure 4.1: Bends promotion

4.2 Bends ($\mathcal{N}_b$)

The *bends* aesthetic criterion tries to minimize the number of bends in edges. The metric for bends is based on the total number of edges before (m) and after (m') bend promotion. The total number of bends can also be calculated from these values: $b = m' - m$. The number of edges is scaled against the total number of edge segments after bend promotion, so we calculate the bend criterion as follows: $\mathcal{N}_b = \frac{m}{m'}$.

```
double BendCriterion(Graph& G, GraphAttributes& GA) {
        double m = G.edges.size();
        BendPromotion(G, GA);
        double m2 = G.edges.size();

        return m / m2;
}
```

Figure 4.2: Implementation of bend metric

4.3 Crossings ($\mathcal{N}_c$)

The goal of the *crossings* aesthetic criterion is to minimize edges crossing paths with each other. The metric for crossings is based on the total number of edge crossings, this number is scaled against an upper bound, (an approximation) of the number of possible crossings. The calculations are based on a straight-line drawing, so for this metric we use the auxiliary graph generated from bends promotion.

We first calculate the number of theoretically possible crossings (c_{all}). This means the number of crossings if every edge were to cross every other edge. However, there are edge crossings that are known to be impossible. In a straight-line drawing of a connected graph with no more than one edge between nodes, adjacent edges cannot cross. The number of these impossible crossings is:

$$c_{impossible} = \frac{1}{2}\sum_{j=1}^{n'} degree(u_j)(degree(u_j) - 1)$$

where n' is the number of nodes in $D'(G)$. This number is subtracted from the total number of crossings we calculated. The number of crossings we have after this subtraction (c) is scaled against this upper bound so that 1 represents the maximum *crosslessness*. The implementation of this metric is showed in Figure 4.3

```
double CrossingCriterion(Graph& G, double c) {
    double m = G.edges.size();
    double c_all = (m * (m - 1)) / 2;

    double c_impossible = 0;
    for (node n : G.nodes) {
        double sum = n->degree() * (n->degree() - 1);
        c_impossible += sum;
    }

    c_impossible = c_impossible / 2;
    double c_max = c_all - c_impossible;

    double Nc = 1;
    if (c_max > 0) {
        Nc -= (c / c_max);
    } else Nc = 0;

    return Nc;
}
```

Figure 4.3: Implementation of crossings metric

4.4 Edge orthogonality ($\mathcal{N}_{eo}$)

Purchase splits the notion of orthogonality in a graph drawing into two seperate measurements. The first one is *edge orthogonality*. The extent to which edges and edge segments follow the lines of an imaginary grid.

We calculate the *edge deviation factor* (δ_i) for each edge segment i. (δ_i) is the angular deviation of edge segment i from the horizontal or vertical grid lines and is calculated as follows:

$$\delta_i = \frac{min(\theta_i, |90° - \theta_i|, 180° - \theta_i)}{45°}$$

where θ_i is the positive angle between edge segment i and the x-axis. As we are using edge segments, we will use the auxiliary graph drawing $D'(G)$. The average edge deviation factor of all edge segments is subtracted from 1 so that 1 means a drawing with maximum edge orthogonality.

```
double EdgeOrthoCriterion(Graph& G, GraphAttributes& GA) {
    double sum = 0;

    for (edge e : G.edges) {
        node s = e->source();
        node t = e->target();

        double s_x = GA.x(s);
        double s_y = GA.y(s);

        double t_x = GA.x(t);
        double t_y = GA.y(t);

        double angle =
        abs(atan2(s_y - t_y, t_x - s_x) * 180 / PI);

        Array<double> values =
        {angle, abs(90.0 - angle), abs(180.0 - angle)};

        double *deviation = min_element(begin(values), end(
            values));
        *deviation = *deviation / 45;

        sum += *deviation;
    }

    double avg = ((1.0 / G.edges.size()) * sum);
    return  1.0 - avg;
}
```

Figure 4.4: Implementation of edge orthogonality metric

4.5 Node orthogonality ($\mathcal{N}_{no}$)

The second part of orthogonality as defined by Purchase is the extent to which nodes (and bend points) make use of the grid points in an imaginary grid. To measure this, we calculate the total available grid points and what portion of them is in use. To find the size of our imaginary grid, we calculate the *greatest common divisors* of the set of vertical and the set of horizontal differences between all geometrically adjacent nodes. Because the vertical distance between grid points is not necessarily the same as the horizontal, we calculate two separate *gcd* values. When we divide the coordinates of the position of a node by these *gcd's*, we get the position of the node in our grid.

```
List<int> differences;
List<int>::iterator it;

for (node n : G.nodes) {
    for (node m : G.nodes) {
        if (n != m) {
            if (GA.x(n) == GA.x(m)) {
                int dif = abs(GA.y(n) - GA.y(m));
                if (dif != 0)
                    differencesY.pushBack(dif);
            }
            else if (GA.y(n) == GA.y(m)) {
                int dif = abs(GA.x(n) - GA.x(m));
                if (dif != 0)
                    differencesX.pushBack(dif);
            }
        }
    }
}

int gcdResultX = 1;
if (differencesX.size() > 0) {
    itX = differencesX.get(0);
    gcdResultX = *itX;
    for (int i = 1; i<countX; i++) {
        itX = differencesX.get(i);
        gcdResultX = Math::gcd(gcdResultX, *itX);
    }
}

// The same is done for the greatest common divisor
// of al y-differences.
```

Figure 4.5: Finding the greatest common divisor

We align the drawing so that the node with the least value coordinates is at the origin. We now find the nodes with the largest x-coordinate and with the largest y-coordinate and divide these values by the *gcd's* we found. These are the *width* and *length* of the imaginary grid. Multiplying them will give us the total number of available grid points. The final step is to divide the number of nodes, so the number of grid points in use, by the total amount of grid points available.

```
double A = (width)*(height);
double Nno = G.nodes.size() / A;
```

Figure 4.6: Calculating node orthogonality

4.6 Uniform edge lengths ($\mathcal{N}_{ue}$)

The idea behind the *uniform edge lengths* aesthetic is that when edges have the same length, we get a better graph drawing. Purchase did not introduce a metric for this aesthetic, so we will now introduce our own. The formula is designed in the same way as the ones by Purchase, returning a value between zero and one. Where one means that all edges have the same length.

First we calculate the average edge length of our graph drawing. Because we want to get the lengths of full edges, we do not use an auxiliary graph drawing here.

$$L_{avg} = \frac{L_{total}}{m}$$

Here L_{avg} is the average edge length, L_{total} is the combined length of all edges and m is the number of edges. Based on this average edge length, we then calculate the deviation from this length for each edge, and the average deviation.

$$D_i = |L_i - L_{avg}|$$
$$D_{avg} = \frac{1}{m}\sum_{i=1}^{m} D_i$$

Where D_i is the deviation from the average length for edge i and L_i is the length of edge i. As was the case with the *bends* metric, there is no upper bound to scale against. Instead we scale the average deviation against the average edge length. See Figure 4.7 for the implementation.

$$\begin{cases} N_{ue} = 1 - \frac{D_{avg}}{L_{avg}} \text{ if } D_{avg} < L_{avg} \\ N_{ue} = 0 \text{ otherwise} \end{cases}$$

```
double UniformEdgeCriterion(Graph& G, GraphAttributes& GA) {
double total_length = 0;
for (edge e : G.edges) {
    total_length += edgeLength(e, GA);
}
double avg_length = total_length / G.edges.size();

double total_deviation = 0;
for (edge e : G.edges) {
    total_deviation += abs(edgeLength(e, GA) - avg_length);
}
double avg_deviation = total_deviation / G.edges.size();

if (avg_deviation < avg_length)
    return 1 - (avg_deviation / avg_length);
else return 0;
}
```

Figure 4.7: Implementation for uniform edge length metric

Chapter 5

Algorithm

In this chapter, we will describe the algorithm we used when creating our test results. We initially planned on designing and implementing our own, new algorithm. However, as we were researching existing literature, it became clear that this would take a lot more time than we had available. Because of this we chose to use an existing algorithm and implementation.

As we have shown in Section 3.1.3 there do not exist many algorithms specifically designed for drawing E/R diagrams. We decided to use the algorithm introduced by Tamassia et al. [24] as a starting point because this algorithm is catered towards drawing diagrams. This algorithm is particularly useful as it describes a more general approach to drawing diagrams that we can use. The approach consists of three steps:

1. planarization,
2. orthogonalization,
3. compaction.

Following this outline, we now have to choose specific algorithms and implementations that perform these steps. This is described in more detail below. All of the implementations for the algorithms we have chosen are taken from the Open Graph Drawing Framework [6]. As a result of this, we do differ from the original algorithm a bit. The largest difference is that we do not actually have a separate compaction stage.

While describing the algorithm we will refer to E/R models as a *graph*, where the nodes of the graph represent entity sets and the edges represent relationships. We will explain the general approach of these algorithms, but will not go into all technical details, as this is not the goal of this thesis.

5.1 Planarization

The first step in our algorithm is to make sure we have a planar representation. If our graph is planar, this means finding an *embedding* for the graph. However in many cases the original graph may not be planar. When this is the case, a planar representation has to be created. This process can be separated into two parts:

1. Finding a planar subgraph.
2. Reinserting edges into our graph.

5.1.1 Finding a planar subgraph

When creating a planar subgraph, some edges of the original graph have to be removed. We would like the number of edges that must be removed to be as low as possible. This is because reinserting these edges will result in edge crossings. We want the number of edge crossings to stay low.

The algorithm we have used for finding a planar subgraph is one introduced by Jayakumar et al. [15]. Their algorithm uses PQ-trees and has a time complexity of $\mathcal{O}(n^2)$. The algorithm is based on the planarity test by Lempel, Even and Cederbaum [16] and follows the approach for graph planarization as described by Ozawa and Takahashi [17].

In broad terms, the algorithm builds a subgraph by adding nodes and their outgoing edges to it. The order in which the nodes are added to the subgraph is determined by a so-called *st-numbering*. The algorithm starts with the first node and follows the st-numbering. When a node is added, the new subgraph is tested for planarity. The planarity testing algorithm is implemented using a data structure called PQ-trees. If the subgraph is planar, the algorithm can continue adding nodes. When the subgraph is not planar, some edges have to be removed in order to regain planarity. Determining which edges should be removed is also done using PQ-trees.

5.1.2 Reinserting edges

We now have a planar subgraph, but we do not want edges to be discarded. The next step in planarizing our graph is to reinsert the removed edges. When reinserting edges we try to keep edge crossings to a minimum. Reinserting the deleted edges into the graph is done using an algorithm by Gutwenger et al. [13]. To make sure we have a planar representation of our graph, edge crossings are (temporarily) replaced with *dummy nodes*.

Embedding

The algorithm we use for inserting edges also gives us an *embedding* of our planarized graph. An embedding defines the order of incident edges

around each node. When two graph drawings have the same order they are considered to be equivalent. A graph can have multiple embeddings, an example of this is shown in Figure 5.1. This figure also shows that choosing a certain embedding can have great impact on the amount of crossings in the drawing.

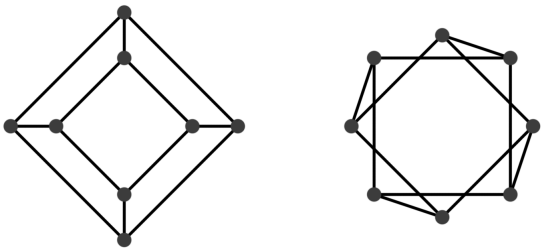

Figure 5.1: Two embeddings of the same graph

Gutwenger et al. [13] takes into account that choosing the right embedding can have great impact on the quality of a drawing. They define the *edge insertion problem* as follows: Given a planar graph $G = (V, E)$ and a pair of vertices (v_1, v_2) in G, find an embedding Π of G such that we can add the edge $e = (v_1, v_2)$ to Π with the minimum possible number of crossings among all embeddings of G. After the insertion of the previously deleted edges we now have an embedding of the planarized graph.

5.2 Orthogonalization

Now that we have a planar embedding of the graph, the next step is to create an orthogonal representation. Recall from Section 2.1.1 that an orthogonal drawing maps each edge into a chain of horizontal and vertical segments. Figure 5.2 shows an example of two different drawings, where the right one is an orthogonal drawing of the same graph.

This orthogonalization step is done using a variation of the bend minimizing algorithm by Tamassia [23]. This algorithm works by constructing a flow network from the graph. Then a minimum cost flow in this network is calculated and is used to embed the graph in the grid. Originally the algorithm requires as input a planar graph containing nodes with a maximum degree of four and an embedding of the graph. The algorithm as we use it is adapted to work with nodes of arbitrary node degrees.

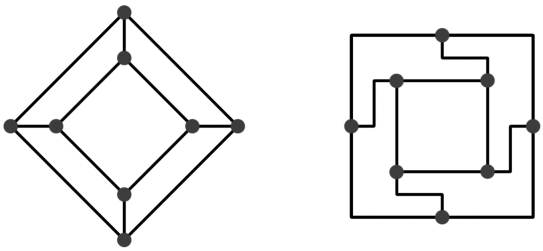

Figure 5.2: Straight-line and orthogonal drawing of the same graph

5.3 Compaction

The final step of the algorithm determines the actual size and lengths of the nodes and edges and by doing so fixes the coordinates of the drawing. With the implementations we use from the OGDF compaction is not a real distinct step, as it is in the original approach by Tamassia.

The *module* for the orthogonalization step as implemented in the OGDF is called `OrthoLayout`. With this module some settings for spacing can be specified. For example the minimum distance between nodes and edges, or how close an edge can attach at the corner of a node. With these settings we can have some influence on the compaction.

Chapter 6

Results and discussion

6.1 Results

For testing the algorithm and scoring the drawings we have used real database models. We did not use (randomly) generated graphs, because they would not be representative for real life E/R diagrams.

We will now show some of the drawings that the algorithm has created. Drawings of all models can be found in Appendix A. Figure 6.1 is a model containing information regarding students and their activities. It is a medium sized E/R diagram, consisting of 29 nodes and 38 edges.

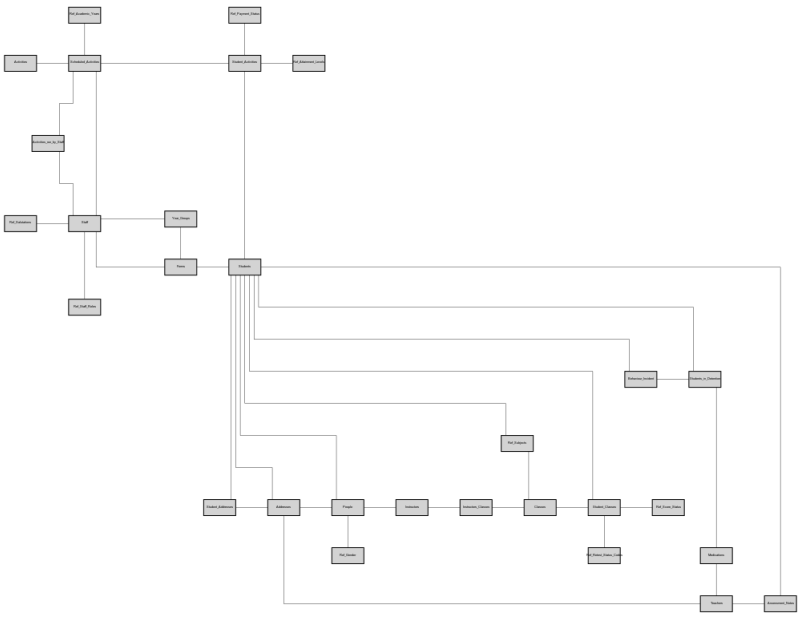

Figure 6.1: Student model

The next example is a diagram of a *school management system* model. It is a small E/R diagram, consisting of 15 nodes and 18 edges. The drawing in Figure 6.3 is our largest one, this graph has 84 nodes and 140 edges.

We do not show edge labels, indicators for cardinality or symbols for the type of relationship in our drawing. This is because the algorithm and the metrics we used for testing focus on the positioning of nodes and edges.

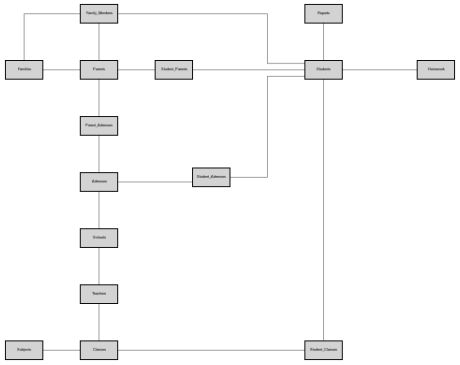

Figure 6.2: School management system

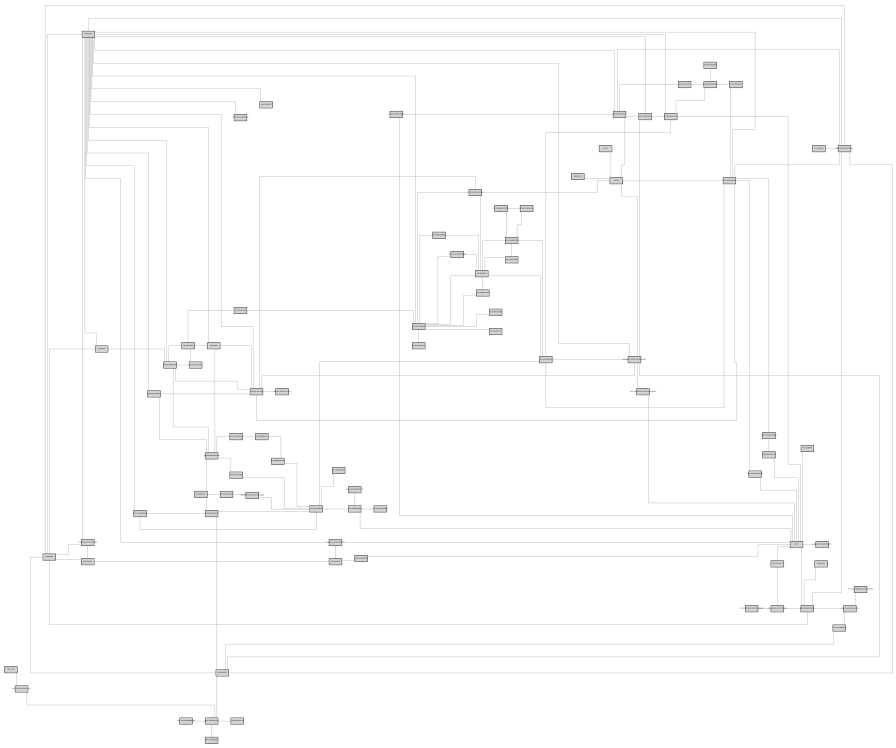

Figure 6.3: Big System

Table 6.1 shows the scores our algorithm got for each metric, tested with a variety of models. As can be read in Chapter 4, we have tested with the following metrics:

- Crossings ($\mathcal{N}_c$)
  A higher score indicates less edge crossings in the drawing.
- Bends ($\mathcal{N}_b$)
  A higher score indicates less edge bends in the drawing.
- Node orthogonality ($\mathcal{N}_{no}$)
  A higher score indicates that nodes make good use of a grid.
- Edge orthogonality ($\mathcal{N}_{eo}$)
  A higher score indicates more edges following horizontal and vertical paths.
- Uniform edge lengths ($\mathcal{N}_{ue}$)
  A higher score indicates more edges are of (roughly) the same length.

Models 1 to 7 are taken from a large collection of industry models [26]. Model 8 is provided by the company CGI. Models 9 and 10 are taken from templates available in Microsoft Access. Models 11 to 13 are taken from another collection of ready-to-use data models [9].

	Model	$\mathcal{N}_c$	$\mathcal{N}_b$	$\mathcal{N}_{no}$	$\mathcal{N}_{eo}$	$\mathcal{N}_{ue}$
1	School management system	1	0.78	0.05	1	0.18
2	Pharma & Biotech info	1	0.79	0.04	1	0.37
3	Car sales	1	0.67	0	1	0.25
4	Student behavior monitoring	1	0.82	0.58	1	0.37
5	Student Class Scheduling	1	1	0.55	1	0.58
6	Student activities	1	0.74	0	1	0.49
7	Student combined*	1	0.66	0	1	0.01
8	Big system	1	0.47	0	1	0.01
9	Access: Books	1	1	1	1	1
10	Access: Attendence tracker	1	1	0.44	1	1
11	Survey data model	1	0.51	0.04	1	0.41
12	Online shop data model	1	0.74	0	1	0.24
13	Real estate data model	1	0.71	0.01	1	0.27

Table 6.1: Results

** Model 7 is the combination of models 4, 5 and 6.*

The values in Table 6.1 have been rounded, this means that the ones for the crossings metric do not always mean that the drawing had zero crossings. For example, the *big system* model actually has a decent amount of crossings (47), which can be seen in Figure 6.3. Because the crossings metric is scaled against all possible crossings, the score was still 0.999706 which gets rounded up to 1.

	$\mathcal{N}_c$	$\mathcal{N}_b$	$\mathcal{N}_{no}$	$\mathcal{N}_{eo}$	$\mathcal{N}_{ue}$
Average	1	0.76	0.21	1	0.4

Table 6.2: Average results for metrics

6.2 Discussion

When looking at our results in Table 6.1, the scores for the *crossings* ($\mathcal{N}_c$) and *edge orthogonality* ($\mathcal{N}_{eo}$) metric stand out. The results for edge orthogonality are not surprising. The algorithm we used strictly draws horizontal and vertical edge segments. Because of this the results for this metric will always be one.

We expected the results for the crossings metric to be better than the others, as the main focus of the algorithm is to keep crossings low. However we did not think we would only get the maximum score with all models. These results show that the *planarization approach* of the algorithm is really effective at keeping the number of edge crossings low. When possible, a planar E/R diagram is created. But even with larger and non-planar models, the algorithm still scores one.

The next best scoring metric is *bends* ($\mathcal{N}_b$). With the exception of the *big system* model, the metric scored at least above 0.5. As can be seen in Figure 6.2, the average score of this metric was 0.76. This is a decent score, but when looking at the generated drawings, we can easily detect some unnecessary bends. These are mostly bends that would not have been needed if non-orthogonal, straight lines had been used. A nice example of this is Figure 6.1, where edges from the center node (*Student*) connecting to the nodes below have a lot of bends. Most of these bends are only a result of the orthogonal drawing style.

The two lowest scoring criteria are *uniform edge length* ($\mathcal{N}_{ue}$) and *node orthogonality* ($\mathcal{N}_{no}$). As already noted while discussing the bends criterion, the way edges are routed could be improved. Apart from removing unnecessary bends, we can also see whole parts of a drawing that can be moved to create more uniform edge lengths. Again we look at Figure 6.1 as an example. The bottom right part of the drawing could be moved upwards. This would reduce the lengths of a lot of edges and increase uniformity.

The results for node orthogonality turned out lower than expected. These low results are the result of the way this metric is calculated (see chapter 4). The calculation of this metric revolves around a virtual grid on which nodes are mapped. However, the algorithm does not map nodes strictly to a grid. This can cause the size of the grid as calculated for the metric to be exceedingly large, as if every pixel was a possible grid point. Because of this some models score zero for this metric. This also means that the results for this metric do not tell us much about the drawings, other than that the algorithm fails to place nodes in a neat grid. A better measurement would have been to simply calculate how much of the total drawing space is used.

Chapter 7

Conclusions and future work

7.1 Literature study

From the research we have done into existing literature we can conclude that *aesthetic criteria* play an important role when it comes to graph drawing. However, when graph drawing algorithms are introduced, no proof of how well the algorithm preforms and no way of measuring these criteria is presented. This problem has been identified and metrics for some of these aesthetic criteria have been introduced [19]. We have introduced one extra metric and implemented a total of five metrics.

Some research has been done into the effect of aesthetic criteria on the readability and understandability of a graph. These studies have shown that the *crossings* criteria has the most impact. The usefulness of the *bends* criteria is also confirmed, but the results are less clear than for crossings. Other criteria that were tested did not show significant improvement in readability and understandability of graphs. These studies focused on general graphs, not E/R diagrams. One study looked specifically at the criteria in UML diagrams. This research did not include the crossings criteria and only the bends criteria showed a little impact.

No research regarding the impact of aesthetic criteria specifically for E/R diagrams exists. There is a study that compares two notation methods for E/R diagrams and how they impact the readability and understandability. This research shows that the *SSADM* notation performs better than *Chen's* notation.

7.2 Experiment

Our own experiment shows that an algorithm is capable of drawing E/R diagrams that get good scores when applying the metrics for aesthetic criteria. The algorithm we used focuses on the crossings criterion and gets the maximum value for all models we tested with. While the algorithm performs

really well with crossings, the other criteria score significantly lower. The bends criterion gets decent scores, but upon inspecting the actual drawing, there clearly is room for improvement.

7.3 Future Work

While there already exists a lot of research in the field of graph drawing and aesthetic criteria, there is still more that can be done.

An empirical study researching the human understanding of E/R diagrams drawn with various aesthetics does not yet exist. In stead of having to answer questions about general graph theory, subjects could answer questions about the underlying E/R model. It is possible that the results of such a study are different than those of studies where general graphs were used. It would also be interesting to find out what is considered a *good* score for each criteria. It could be that for one criterion 0.5 is already pretty good while for another criterion only 0.9 or higher is considered a good score. As there is not much data available this is hard to tell as of right now.

This brings us to the following point. Our research can be used to gather more data regarding the aesthetic criteria. The implementation of the metrics enables researchers to score their own (existing or new) algorithms and compare them with each other and also with our results. We also think our own research can be improved upon by testing more models, especially larger models. Our own experiment consisted mostly of small models, with only one really large model.

Apart from researching the aesthetic criteria and their metrics, more can be done with respect to the actual algorithm. As we have seen, the current algorithm only really performs well with one criterion. New algorithms can definitely improve in a number of ways:

1. Reduce the number of bends.
2. Make edge lengths more uniform.

We also think it would be interesting to look further than the current aesthetic criteria. Perhaps an entirely different way to look at graphs and E/R diagrams can be used to generate a layout in stead of using the current aesthetics as guidelines.

Bibliography

[1] Carlo. Batini, M. Talamo, and Tamassia Roberto. Computer aided layout of entity relationship diagrams. *The Journal of Systems and Software*, 4:163–173, 1984.

[2] John M Boyer and Wendy J Myrvold. Stop minding your p's and q's: A simplified $\mathcal{O}(n)$ planar embedding algorithm. In *SODA*, volume 99, pages 140–146, 1999.

[3] Jiazhen Cai, Xiaofeng Han, and Robert E Tarjan. An $\mathcal{O}(m \log n)$-time algorithm for the maximal planar subgraph problem. *SIAM Journal on Computing*, 22(6):1142–1162, 1993.

[4] Gruia Călinescu, Cristina G Fernandes, Ulrich Finkler, and Howard Karloff. A better approximation algorithm for finding planar subgraphs. *Journal of Algorithms*, 27(2):269–302, 1998.

[5] Peter Chen. The entity-relationship model - toward a unified view of data. *ACM Transactions on Database Systems*, 1:9–36, 1976.

[6] Markus Chimani, Carsten Gutwenger, Michael Jünger, Gunnar W Klau, Karsten Klein, and Petra Mutzel. The open graph drawing framework (ogdf). *Handbook of Graph Drawing and Visualization*, 2011:543–569, 2013.

[7] Giuseppe Di Battista, Peter Eades, Roberto Tamassia, and Ioannis G Tollis. Algorithms for drawing graphs: an annotated bibliography. *Computational Geometry*, 4(5):235–282, 1994.

[8] PA Eades. A heuristic for graph drawing. *Congressus numerantium*, 42:149–160, 1984.

[9] erdiagrams.com. Erdiagrams.com. Retrieved from `http://www.erdiagrams.com/`.

[10] Shimon Even. *Graph Algorithms*. Computer Science Press, 2012.

[11] Thomas MJ Fruchterman and Edward M Reingold. Graph drawing by force-directed placement. *Software: Practice and experience*, 21(11):1129–1164, 1991.

[12] Hector Garcia-Molina. *Database systems: the complete book*. Pearson Education India, 2008.

[13] Carsten Gutwenger, Petra Mutzel, and René Weiskircher. Inserting an edge into a planar graph. *Algorithmica*, 41(4):289–308, 2005.

[14] John Hopcroft and Robert Tarjan. Efficient planarity testing. *Journal of the ACM (JACM)*, 21(4):549–568, 1974.

[15] R Jayakumar, Krishnaiyan Thulasiraman, and Madisetti NS Swamy. O (n^2) algorithms for graph planarization. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 8(3):257–267, 1989.

[16] Abraham Lempel, Even S, and Cederbaum I. An algorithm for planarity testing of graphs. *Theory of Graphs, International Syposium, Rome, July 1966*, 1967.

[17] Takao Ozawa and H Takahashi. A graph-planarization algorithm and its application to random graphs. In *Graph Theory and Algorithms*, pages 95–107. Springer, 1981.

[18] Helen Purchase. Which aesthetic has the greatest effect on human understanding? In *International Symposium on Graph Drawing*, pages 248–261. Springer, 1997.

[19] Helen C Purchase. Metrics for graph drawing aesthetics. *Journal of Visual Languages & Computing*, 13(5):501–516, 2002.

[20] Helen C Purchase, Robert F Cohen, and Murray James. Validating graph drawing aesthetics. In *International Symposium on Graph Drawing*, pages 435–446. Springer, 1995.

[21] Helen C Purchase, Matthew McGill, Linda Colpoys, and David Carrington. Graph drawing aesthetics and the comprehension of uml class diagrams: an empirical study. In *Proceedings of the 2001 Asia-Pacific symposium on Information visualisation-Volume 9*, pages 129–137. Australian Computer Society, Inc., 2001.

[22] Helen C Purchase, Ray Welland, Matthew McGill, and Linda Colpoys. Comprehension of diagram syntax: an empirical study of entity relationship notations. *International Journal of Human-Computer Studies*, 61(2):187–203, 2004.

[23] Roberto Tamassia. On embedding a graph in the grid with the minimum number of bends. *SIAM Journal on Computing*, 16(3):421–444, 1987.

[24] Roberto Tamassia, Giuseppe Di Battista, and Carlo Batini. Automatic graph drawing and readability of diagrams. *IEEE Transactions on Systems, Man and Cybernetics*, 18:61–79, 1988.

[25] Philip Weaver, Nick Lambrou, and Matthew Walkley. *Practical SSADM Version 4+: a complete tutorial guide*. Financial Times Pitman, 1998.

[26] Barry Williams. Industry data models. Retrieved from `http://www.databaseanswers.org/data_models/index.htm`.

Appendix A

E/R diagram drawings

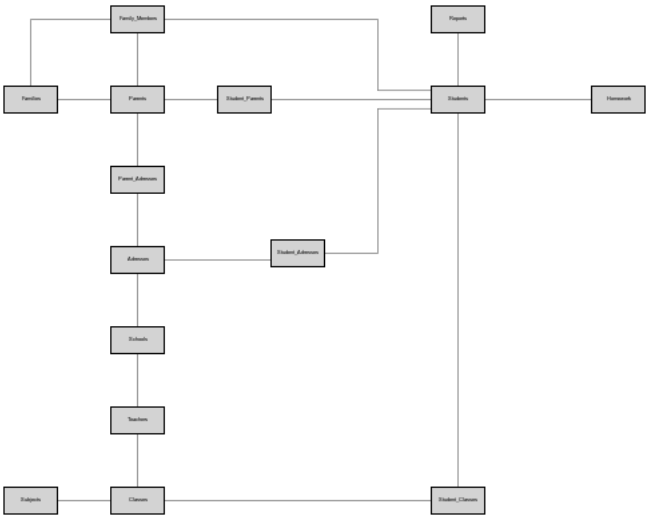

Figure A.1: School management system

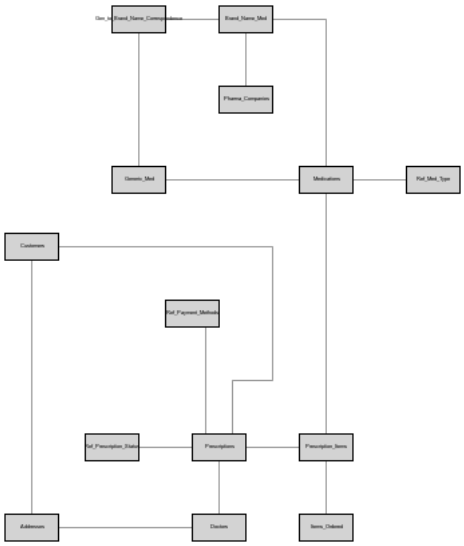

Figure A.2: Pharma & Biotech info

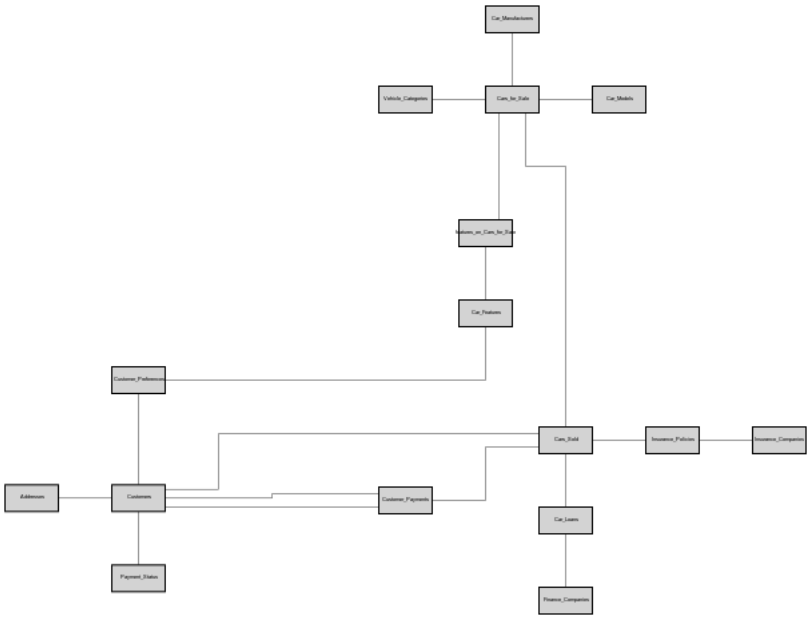

Figure A.3: Car sales

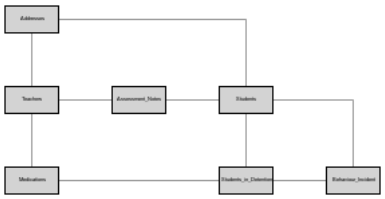

Figure A.4: Student behavior monitoring

Figure A.5: Student Class Scheduling

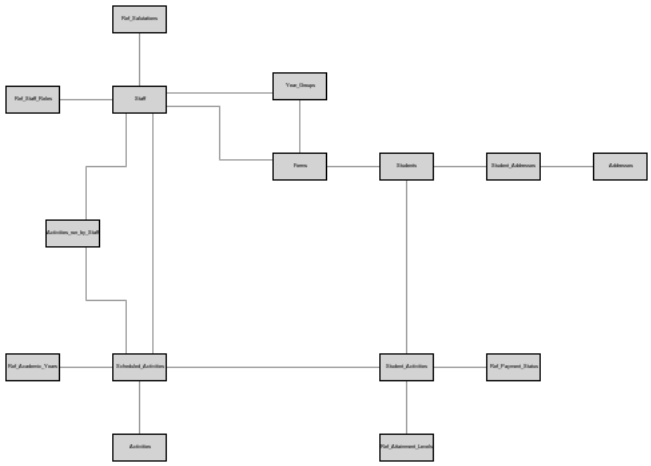

Figure A.6: Student activities

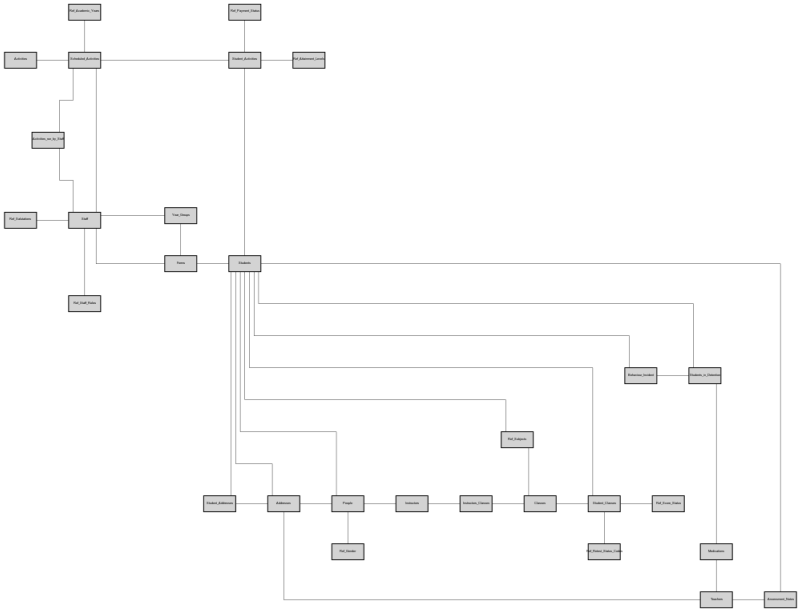

Figure A.7: Student combined

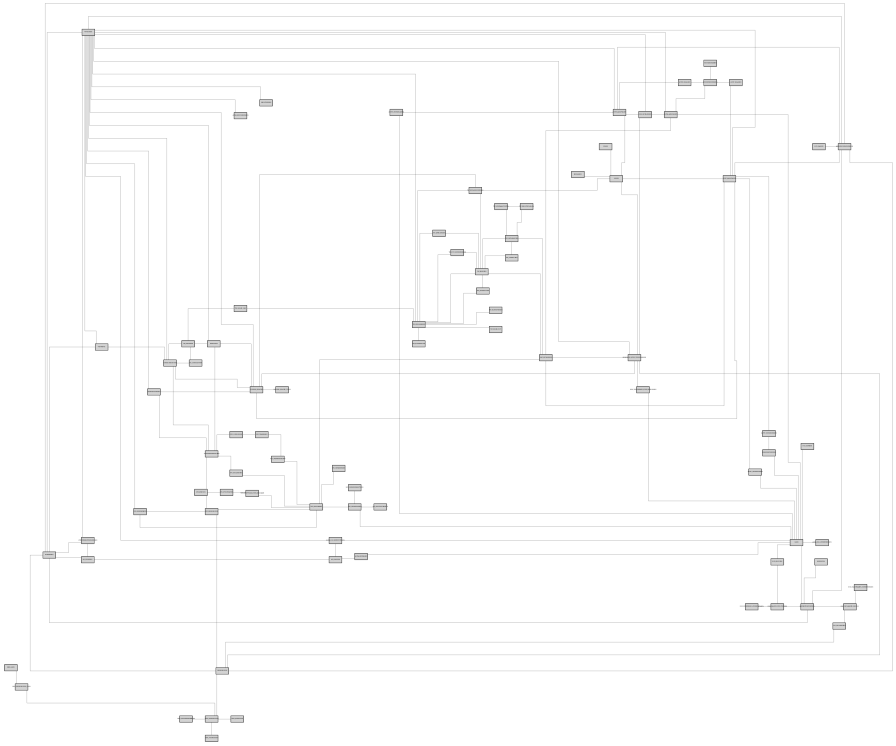

Figure A.8: Big system

Figure A.9: Access: Books

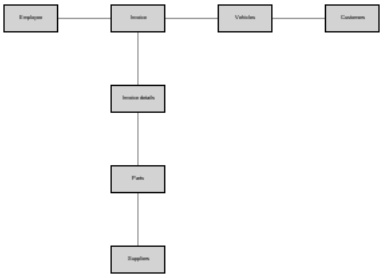

Figure A.10: Access: Attendence tracker

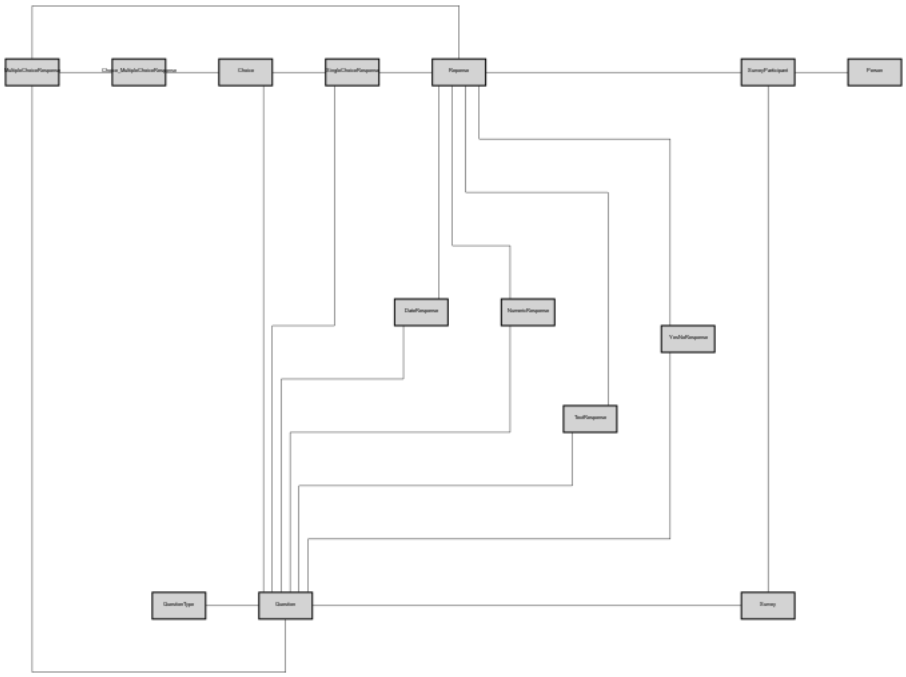

Figure A.11: Survey data model

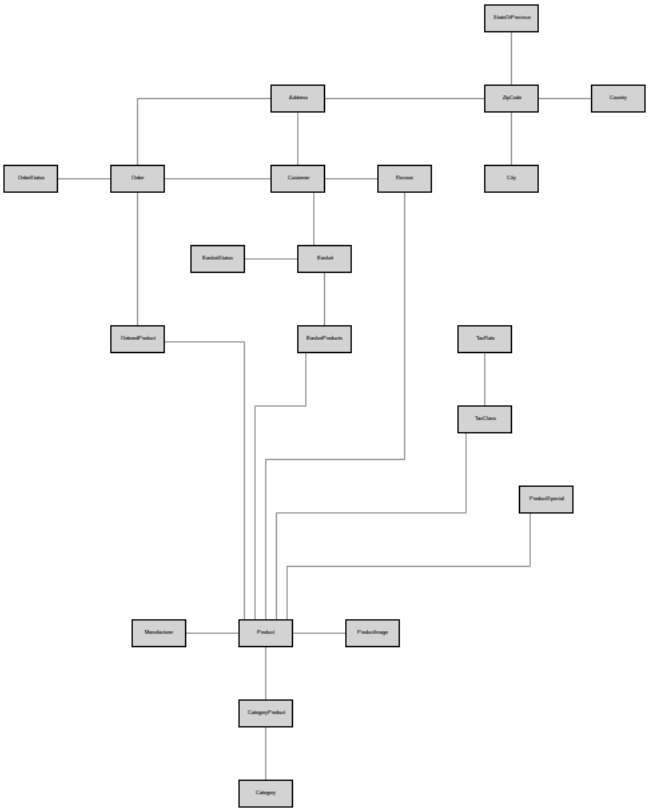

Figure A.12: Online shop data model

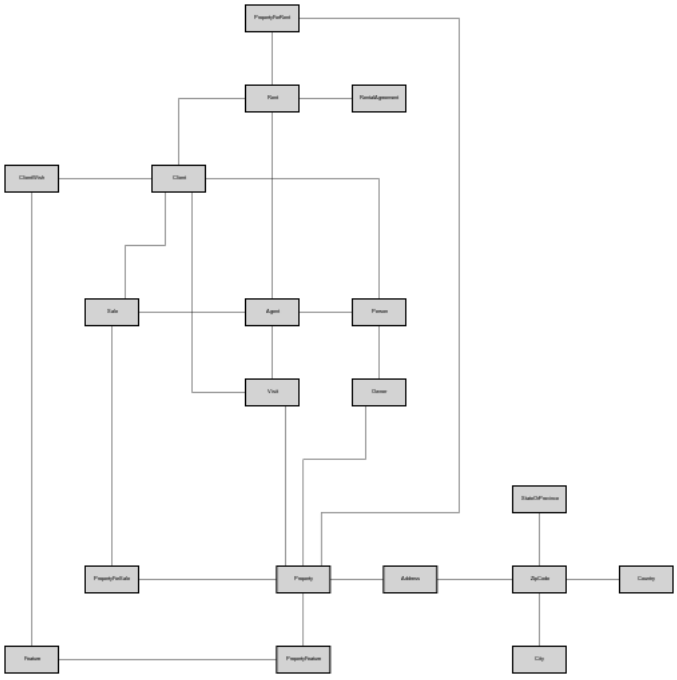

Figure A.13: Real estate data model