faculty of science
and engineering

mathematics and applied
mathematics

Drawing the Kontsevich Graphs in LaTeX

Bachelor's Project Mathematics

June 2020

Student: S.S. Kerkhove

First supervisor: Dr. A.V. Kiselev

Second assessor: Dr. A.E. Sterk

Abstract

Kontsevich graphs are a class of oriented graphs on two sinks, build of wedges. Such graphs are naturally encoded by a list of pairs of target vertices of each wedge. This project is about semi-automatic drawing of Kontsevich graphs in the LaTeX picture environment, that is, finding the Cartesian coordinates of the top vertex of each wedge such that the result can be drawn in the picture environment while satisfying several requirements.

An algorithm to achieve this was proposed by A.V. Kiselev. This algorithm first calculates the aforementioned coordinates based on randomly assigned inclines of the edges. It then filters out the bad drawings and finally uses a penalty function to determine the most beautiful drawing. The goal of this project is to tune and modify this penalty function so that its output guarantees aesthetically pleasing drawings. I have implemented this algorithm in SageMath (with the help of R. Buring).

In this paper I contribute several modifications to the existing algorithm and I analyse which part of the penalty function is most important to draw nice graphs. This whole project is presently applied to drawing – for the first time! – a significant number of the 247 four-wedge graphs which have been discovered by Buring and Kiselev in [arXiv:1702.00681] at the fourth order of the parameter in the expansion of noncommutative associative star-product by Kontsevich.

Preface

I will start with giving a background to Kontsevich Graphs in Section 1. The purpose and approach of this project will then be discussed in Section 2. In Section 3 I will explain the steps of the algorithm proposed by A.V. Kiselev. The implementation of the algorithm in SageMath will be discussed in Section 4, the resulting programmes and their history can be found in the repository at: `https://github.com/SKerkhove/Bachelors-Project`. In these two previous sections the target function, some kind of penalty function, is mentioned, but as the (possibly) most important part of the algorithm, it will be discussed in depth in Section 5. I will then look at how the resulting drawings from the target function react to changes in its parameters in Section 6. Section 7 contains a discussion about how to change the algorithm in order to draw 2-cycles in Subsection 7.1 and a look at how

the resulting drawings change when changing the parameters of the target function in Subsection 7.2. Conclusions will then be drawn and discussed in Section 9. Finally, in Section 10, future developments of this topic will be discussed.

Contents

1 Introduction 5

2 Research Question and Approach 8

3 The Algorithm 10

4 Programming the Algorithm 12
- 4.1 The Linear System 13
- 4.2 The Algorithm 15

5 The Target Function 20
- 5.1 The Short and Long Edges Score 20
- 5.2 Intersections Score 23
- 5.3 Distance Between Unrelated Vertices Score 24
- 5.4 Overshoot Score 26
- 5.5 Height y-coordinate Score 26
- 5.6 Points on Line Score 27
- 5.7 Vertical Lines Score 27
- 5.8 Concluding Remarks and Summary 28

6 Tuning the Target Function 30
- 6.1 The Wedge 30
- 6.2 The Double Wedge, The Graph With Encoding 2 2 1 01 01 . 36
- 6.3 The Graph With Encoding 2 3 1 01 12 12 46
  - 6.3.1 Some Probability Theory 46
  - 6.3.2 Tuning the Target Function 48
- 6.4 All Graphs up to Order 3 54

7 Graphs with 2-cycles 57
- 7.1 Changing the Programme 58
- 7.2 Tuning the Target Function 59

8 Graphs of order 4 62

9 Conclusions 67

10 Discussion 73

A Appendix 81

A.1 Matlab code 81
A.1.1 The function graphfunc 81
A.2 Sage Code 81
A.2.1 The Function to generate the vector b 82
A.2.2 The Function to generate the matrix A 83
A.2.3 the Function Inclines to Coordinates 84
A.2.4 The Function DrawGraph algorithm 85
A.2.5 The Function Positive Test 87
A.2.6 The Function Same Vertices Test 88
A.2.7 The Function Length Test 89
A.2.8 The Function Overlapping Edges Test 91
A.2.9 The Function DrawGraph Filter 93
A.2.10 The Function DrawGraph Compute and Draw 96
A.2.11 The Function DrawGraph Draw 97
A.2.12 The Target Function 98
A.2.13 The Function Intersection Test 103
A.2.14 The Function Inclines to coordinates 104
A.2.15 The Function Generate List of Coordinates 105
A.2.16 The Function Choosing Best Five Pictures 108
A.2.17 Code to Sort All Possible Coordinates for the Wedge . 110
A.2.18 The Mirroring Function 112

1 Introduction

One of the early problems in the study of quantum mechanics was that the rules of classical mechanics don't hold anymore, because quantum mechanics deals with phenomena at nanoscopic scales. There is a difference in the mathematical formulation of the observable phenomena. In classical mechanics, observations are functions over the phase space (a Poisson manifold). On the other hand, in quantum mechanics, observations are functions over Hilbert spaces of wave functions. This called for a search for deformations of algebras of functions on Poisson manifolds. Deformation quantization seeks to describe the non-commutative structure of functions on the Hilbert space algebraically on the phase space [14]. A major breakthrough was made by the Russian mathematician Maxim Kontsevich in the 1990s when he proved his formality theorem, implying the existence and classification of star products on Poisson Manifolds (see Sections 1, 2.2 and 3.3 of [5] for details). To prove this, he came with the revolutionary idea of using graphs in Poisson Geometry calculus [12] (see [9, 10, 11] for his work leading up to this result).

Figure 1: The Kontsevich graphs determine polydifferential operators in the Kontsevich star product.

This project will focus on the graphs associated with the Kontsevich star product. As we can see in Figure 1, Kontsevich graphs are a class of directed graphs. Therefore, we will now first look into some basic graph

theory terminology.

We define directed graphs as follows (see Section 1.2 of [1] and 1.10 of [4]):

Definition 1. A **directed graph** (also called **digraph**) D consists of a finite set $V(D) \neq \emptyset$ of elements called **vertices** and another finite set $E(D)$ of ordered pairs of distinct vertices called edges. Such a graph D is often written as $D = (V, E)$.

Let's look at the following example for an illustration of this:

Example 1. The graph D in Figure 2 is denoted by

$$V(D) = \{x_1, x_2, x_3, x_4, x_5, x_6\}$$

and

$$E(D) = \{\langle x_3, x_1\rangle, \langle x_3, x_2\rangle, \langle x_4, x_1\rangle, \langle x_4, x_3\rangle, \langle x_5, x_2\rangle, \langle x_5, x_3\rangle, \langle x_6, x_2\rangle, \langle x_6, x_3\rangle\}.$$

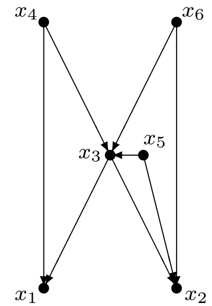


Figure 2: The graph D.

Given an edge $\langle u, v\rangle$ I will call the first vertex u its **tail**, and its latter vertex v its head. This, and the upcoming definition, have also been defined in Section 1.2 of [1].

Definition 2. Given a digraph $D = (V, E)$ and a vertex $v \in V$ the **out-degree** of v is the number of edges that have their tail in v.

A vertex with out-degree 0 will be called a **sink**. A vertex v of out-degree larger than 0 is a **source** for the edges that have their tails in v.

In the article by Buring and Kiselev [2] Kontsevich graphs of type (m, n) (first defined in [8, 9]) are described as:

Definition 3. Consider a class of directed graphs on $m+n$ vertices, labelled from 0 to $m+n-1$ such that the vertices $0,...,m-1$ are sinks and the other vertices are internal. Each of the internal vertices is a source for two edges, the two edges are ordered $L \prec R$: the preceding edge is labelled L and the other edge is labelled R. A **Kontsevich graph** of type (m,n) is an oriented graph on m sinks and n edges.

With *preceding* in the above definition, we mean that for vertex $k \geq m$, the label of the edge $L(k)$ is a lower number than the label of the edge $R(k)$.

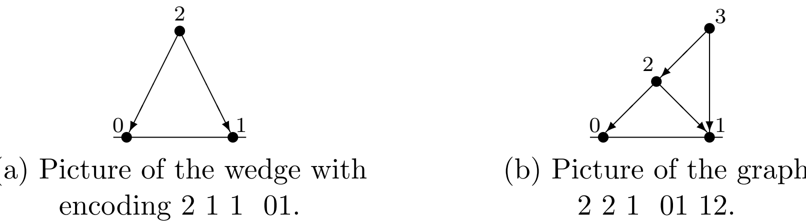
In the article by Buring, Kiselev and Rutten [3] the notion of oriented Kontsevich graphs is slightly differently formulated: The graphs are built over m ordered sinks from n wedges $\overset{L}{\leftarrow} \bullet \overset{R}{\rightarrow}$: each top $\bullet$ of such a wedge is the source of exactly two arrows. A wedge is a graph existing of 3 nodes and two vertices, those two vertices with a tail at the same node and both ending at a different node (see Figure 3a).

Every Kontsevich graph is uniquely determined by the numbers m and n, and a list of ordered pairs of nodes to which the vertices should connect.[2] We store the graphs in the format:

$$m \; n \; s \; \langle \text{list of ordered pairs} \rangle \tag{1}$$

as introduced in [2].

Example 2. The encoding 2 1 1 01 corresponds to the wedge, seen in Figure 3a. The encoding 2 2 1 01 12 corresponds to the graph in Figure 3b.



(a) Picture of the wedge with encoding 2 1 1 01.

(b) Picture of the graph 2 2 1 01 12.

Figure 3: The pictures of Example 2.

The number s is the *sign* of the encoding, it equals 0, 1 or -1. Each Kontsevich graph can be associated with a formula, a differential operator. To a Kontsevich graph with sign one can associate a differential operator

multiplied with the sign. For example, the three encodings 2 3 1 01 12 12, 2 3 -1 01 21 12 and 2 3 1 10 21 12 all correspond to the same graph. If the formula corresponding to the graph is zero, the sign is set to be 0 too. However, the sign part of the encoding will be mainly ignored in the rest of the project.

The pictures in Figure 3 are made in the LaTeX picture environment. This environment has a lot of restrictions on drawing (see Section 2). However, it is worth trying to draw the Kontsevich graphs in the picture environment, because for one, drawings within the picture environment do not take up a lot of space. Secondly, no external packages are needed when using the picture environment to draw the Kontsevich graphs, this makes it easier to use in manuscripts sent to journals.

2 Research Question and Approach

We would like to be able to properly draw the Kontsevich graphs arising from the expansion $\star \mod \bar{o}(\hbar^4)$ in LaTeX, therefore the research question of my project will be:

How can one nicely semi-automatically draw the given class of Kontsevich's directed graphs up to order 4, specifically with two sinks, within the LaTeX picture environment?

The picture environment allows one to program pictures directly into LaTeX. Although the picture environment has severe restrictions, it produces documents that are small in size and no external packages are needed, a reason for us to study how we can nicely draw graphs with these restrictions in place. To draw a digraph, in order to show the direction of the edges, we need to be able to draw arrows. As written in section 7.1 of Lamport's User Guide to LaTeX [13], arrows are drawn with the command:

```
\put(x,y){\vector(x_0,y_0){len}}.
```

Here, `(x,y)` should be two numerical values, giving the starting point of the vector, the part without the arrowhead. The values that should be put on the place of `(x_0,y_0)` give the slope of the vector, the values are restricted to integers between -4 and 4, inclusive. Moreover, they should have no common

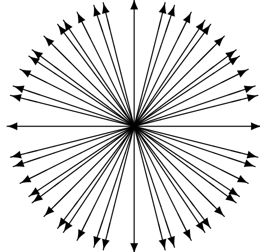


Figure 4: All the possible vector inclines in the LaTeX picture environment.

divisor, i.e. `(-4,2)` is not allowed, because -4 and 2 have common divisor 2. A drawing of all possible vector inclines is given in Figure 4.

The value at the place of `len` gives the length of the projection of the arrow in the horizontal direction, unless the arrow is purely vertical, i.e. if `x_0` $= 0$, then the value `len` does give the vertical length of the arrow. I will use `\unitlength=1pt` for this entire project, 1 point is equal to 1/72 inch, which is approximately 0.35 mm.

Example 3. With Pythagoras's theorem, we can calculate that the command `\put(0,0){\vector(3,4){24}}` gives a vector of length 40. The length is 24 in the horizontal direction and $\frac{24}{3} \cdot 4 = 32$ in the vertical direction. On the other hand, the command `\put(0,0){\vector(4,-3){24}}` gives a vector of length 30, with length 24 in the horizontal direction and 18 in the vertical direction. You can see the result of the commands in Figure 5. As you can see, the up going arrow is longer than the down going arrow, even though they have the same value on the place of `len`. Therefore, the value for `len` does not directly correspond to the length of the vector.

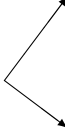


Figure 5: Two vectors with the same horizontal length value.

Another restriction of the picture environment is that because of the way lines are drawn, that for lines that aren't horizontal or vertical, there is a smallest line you can draw in the picture environment. The minimum length

is about 10 points which is 10/72 inch, and about 3.5 mm (see section 7.1 of [13]).

To get an answer to the research question I will implement an algorithm provided by A.V. Kiselev [7], that will generate the coordinates for the edges of the Kontsevich graphs. This algorithm first defines a system of equations with the help of chosen inclines for the edges (see Equation 2 below). I will start with trying to write this system in matrix form, i.e. in the form $Ax = b$ where x is a $2n$ vector of the form $[x_1, y_1, ..., x_n, y_n]^T$ where each pair (x_i, y_i) gives the coordinates of the ith inner vertex of the graph.

After this I will write a function in SageMath that can solve the system for an inputted encoding and value $\delta > 0$ prescribing the distance between the sinks. I will then write a function that filters the solutions to the system, so that we get 'nice' solutions, i.e. no overlapping vertices, no overlapping edges, etc. When this works I will write a target function with easily changed parameters. The target function will assign to each solution of the system as value. The lower the value the 'more beautiful' a drawing of the particular solution is. Finally, I will train the target function. I will try to find the best parameters for it, such that the five best, according to the target function, drawings of a particular graph all look alike.
I will start by excluding graphs with 2-cycles from my considerations. Once I think I have a good idea of the workings of the algorithm, I will extend it to include 2-cycles.

The results of this project will be beneficial to those who want to use Kontsevich graphs in their papers formatted in LaTeX. Because graphs drawn in the picture environment don't take up a lot of memory, the compiling time of the document will be lower, just like the size of the resulting paper. There is not a lot of research on this topic, making this project extra interesting to the scientific community.

3 The Algorithm

In the Syllabus Propadeutic Project (Applied) Mathematics 2017-2018, A.V. Kiselev has written down an algorithm to generate the coordinates of the Kontsevich graphs in such a way that the graphs can be drawn in the LaTeX picture environment [7]. The algorithm consists of 3 main steps, each having several sub steps. The unknown coordinates for the ith internal vertex are

denoted by (x_i, y_i). The algorithm has as input an encoding of a graph and a scalar δ that indicates the distance between the sinks of the graph.

Step 1: For i from 1 to n, take a pair of unequal inclines $(a_i^L : b_i^L)$ and $(a_i^R : b_i^R)$ from the set of choices. As mentioned in Section 2, $-4 \leq a_i^{\{L,R\}} \leq 4$ and $-4 \leq b_i^{\{L,R\}} \leq 4$, moreover $a_i^{\{L,R\}}$ and $b_i^{\{L,R\}}$ can not have a common divisor. It is also not allowed for $a_i^{\{L,R\}}$ and $b_i^{\{L,R\}}$ to both be 0, because that will not actually define an incline of the line. This will give us a total of 24 choices, if we discount opposite vectors in the same direction, e.g. $(1, 1)$ and $(-1, -1)$[1]. Define the equations

$$\frac{x_{L(i)} - x_i}{y_{L(i)} - y_i} = \frac{a_i^L}{b_i^L} \quad \text{and} \quad \frac{x_{R(i)} - x_i}{y_{R(i)} - y_i} = \frac{a_i^R}{b_i^R}. \tag{2}$$

It may seem like there will be a problem when $b_i^{\{L,R\}} = 0$, however when the system will be worked out in Section 4, there will never actually be divided by $b_i^{\{L,R\}}$ in the resulting system. So for now we can just say that $b_i^{\{L,R\}} = 0$ implies that $y_{\{L,R\}(i)} - y_i = 0$ as well.
Because the m sinks have fixed coordinates, the linear system is inhomogeneous. This system can be written in the form $A\mathbf{x} = \mathbf{b}$ where $\mathbf{x} = [x_1, y_1, ..., x_n, y_n]^T$. If this system has a solution go to step 2, if not, try another choice of inclines. For a system with n internal vertices there are $(24 \cdot 23)^n = 552^n$ possible choices of inclines. For every i, there are 24 choices for $(a_i^L : b_i^L)$ and since $(a_i^R : b_i^R)$ must be different, there are 23 choices for $(a_i^R : b_i^R)$.

Step 2: (filter)

- *2.1* Reject all solutions where:
  - $y_i \leq 0$ for some i;
  - at least two vertices overlap;
  - at least two edges are (partly) overlapping;
  - at least one edge is shorter than the minimal length for which LaTeX can draw vectors.

[1] The possible inclines are $(0, 1)$, $(1, 0)$, $(1, 1)$, $(1, 2)$, $(1, 3)$, $(1, 4)$, $(2, 1)$, $(2, 3)$, $(3, 1)$, $(3, 2)$, $(3, 4)$, $(4, 1)$, $(4, 3)$, $(-1, 1)$, $(-1, 2)$, $(-1, 3)$, $(-1, 4)$, $(-2, 1)$, $(-2, 3)$, $(-3, 1)$, $(-3, 2)$, $(-3, 4)$, $(-4, 1)$ and $(-4, 3)$.

- *2.2* Store δ and the list of acceptable coordinates for the internal vertices. Repeat the previous steps for sufficiently many randomly chosen admissible values of inclines.

Step 3: (optimisation)

- *3.1* Choose a target function whose input is a string of coordinates from 2.2. The target function should increase as long as:
  - Some edges are too short, add $a \cdot (l_0/l)^{\alpha}$;
  - Some edges are too long, add $b \cdot (l/l_0)^{\beta}$;
  - There are too many intersections of drawn edges, add $N^{\gamma} \cdot (\log N)^{\epsilon}$.

  Where $a,\ b,\ \alpha,\ \beta,\ \gamma$ and ϵ are all parameters.

- *3.2* Train the target function. Vary the parameters in such a way that for each encoding, the drawings of loop-less graphs are almost identically reproduced among the five best approximations of the minima.

- *3.3* From the set of outputs from *2.2* choose five lists of vertex coordinates such that each list is almost the best with respect to the minimisation of the trained target function.

- *3.4* Draw these 5 pictures in LaTeX and choose the favourite picture encoding.

Extend the algorithm to loop-full graphs. Do this by giving the target function an argument that will ensure that two vertices that point edges at each other are reasonably close to each other.

4 Programming the Algorithm

I will write the system of equations into the form

$$A\mathbf{x} = \mathbf{b} \tag{3}$$

where $\mathbf{x} = \left[x_1, y_1, \ldots, x_n, y_n\right]^T$. I will have to figure out what the matrix A and the vector $\mathbf{b}$ look like. After that, I will have to figure out how to write the algorithm.

4.1 The Linear System

To find out how the matrix A and vector $\mathbf{b}$ should be constructed, I started with easy graphs. In particular, I wrote down the system 3 in case of the wedge.

Example 4. The wedge is encoded by 211 01. This yields the equations

$$\frac{x_{L(1)} - x_1}{y_{L(1)} - y_1} = \frac{a_1^L}{b_1^L} \quad \text{and} \quad \frac{x_{R(1)} - x_1}{y_{R(1)} - y_1} = \frac{a_1^R}{b_1^R}.$$

When we fill this in ($x_{L(1)} = y_{L(1)} = y_{R(1)} = 0$ and $x_{R(1)} = \delta$) and with crosswise multiplying, we can get the equivalent equations

$$b_1^L x_1 - a_1^L y_1 = 0 \quad \text{and} \quad b_1^R x_1 - a_1^R y_1 = b_1^R \delta.$$

This gives us the matrix A and the vector $\mathbf{b}$

$$A = \begin{bmatrix} b_1^L & -a_1^L \\ b_1^R & -a_1^R \end{bmatrix} \quad \text{and} \quad \mathbf{b} = \begin{bmatrix} 0 \\ b_1^R \delta \end{bmatrix}.$$

Let us now look at the more complicated graph with encoding 221 01 12. Using this encoding in our Matlab function graphfunc (given in the Appendix, see Section A.1.1) gives the plot in Figure 6 We see that this

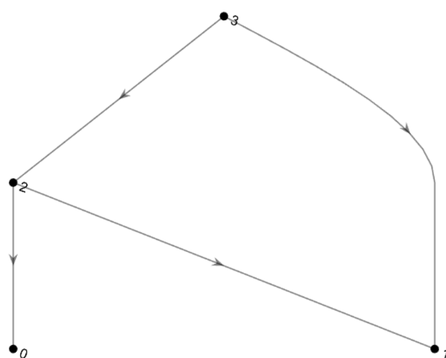


Figure 6: A plot of the graph with encoding 2 2 1 01 12.

graph has no 2-cycles, so we can use our algorithm.

Example 5. For the encoding 2 2 1 01 12 get the equations:

$$\frac{x_{L(1)} - x_1}{y_{L(1)} - y_1} = \frac{a_1^L}{b_1^L}, \qquad \frac{x_{R(1)} - x_1}{y_{R(1)} - y_1} = \frac{a_1^R}{b_1^R},$$

$$\frac{x_{L(2)} - x_2}{y_{L(2)} - y_2} = \frac{a_2^L}{b_2^L} \quad \text{and} \quad \frac{x_{R(2)} - x_2}{y_{R(2)} - y_2} = \frac{a_2^R}{b_2^R}.$$

We can now fill it in, from the encoding we can conclude that $x_{L(1)} = y_{L(1)} = y_{R(1)} = y_{L(2)} = 0$, $x_{R(1)} = x_{L(2)} = \delta$ and $x_{R(2)} = x_1$, $y_{R(2)} = y_1$. With some multiplication and addition, we get the equations:

$$\begin{aligned} b_1^L x_1 - a_1^L y_1 &= 0 \\ b_1^R x_1 - a_1^R y_1 &= b_1^R \delta \\ b_2^L x_2 - a_2^L y_2 &= b_2^L \delta \\ -b_2^L(x_1 - x_2) + a_2^L(y_1 - y_2) &= 0. \end{aligned}$$

These equations correspond to a matrix A and vector $\mathbf{b}$ of the form:

$$A = \begin{bmatrix} b_1^L & -a_1^L & 0 & 0 \\ b_1^R & -a_1^R & 0 & 0 \\ 0 & 0 & b_2^L & -a_2^L \\ -b_2^R & a_2^R & b_2^R & -a_2^R \end{bmatrix} \quad \text{and} \quad \mathbf{b} = \begin{bmatrix} 0 \\ b_1^R\delta \\ b_2^L\delta \\ 0 \end{bmatrix}.$$

We are beginning to see a pattern here. It appears that for every two rows of A we have a block in the corresponding columns of the form

$$\begin{matrix} b_i^L & -a_i^L \\ b_i^R & -a_i^R. \end{matrix}$$

Then, the other entries depend on the encoding. The ith row corresponds to the ith number in the encoding (ignoring the m, n and s slot). If this ith number is 0 or 1, all other entries in the row are zero, because then the corresponding edge goes to one of the sinks, so is not dependent on a variable. However, if this ith number is higher than 1, let's call it e, than the $(e-1)$th set of 2 columns (so the columns $2(e-1)-1$ and $2(e-1)$) have the entry

$$-b_j^{L,R} \quad a_j^{L,R}.$$

A function that gives the matrix A is given in the Appendix, Section A.2.2.

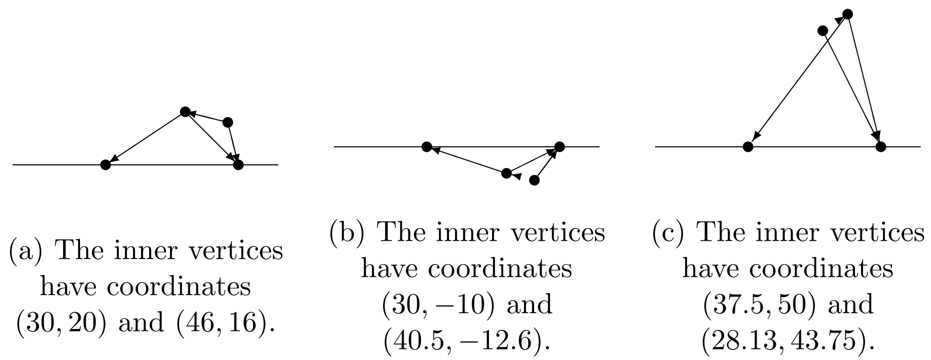
On the other hand, the ith entry in the vector $\mathbf{b}$ is 0 unless the ith entry in the encoding is 1, then the corresponding entry in the vector is of the form

$$b_j^{L,R}\delta.$$

A function that gives the vector b is given in the Appendix, Section A.2.1.

4.2 The Algorithm

I have chosen to program the algorithm in SageMath. After having programmed the algorithms for the matrix A and the vector b it is fairly easy to calculate a solution. I built in a provision that the matrix A is not singular (but for all possible systems of order 1 and 2 without 2-cycles, none of the matrices are singular), if this is the case, can x be calculated with the command `x=A\b`. The code for the algorithm can be found in the Appendix, Section A.2.3.
This basically solves Step 1 of the algorithm, but to automate the process of choosing the inclines I made the function `DrawGraph_algorithm` (see Section A.2.4) which chooses a random set of inclines for as long as it takes to get a non-singular matrix. This algorithm can henceforth give 552^n results, a drawing of three possible results of the algorithm for the encoding 2 2 1 01 12 is given in Figure 7. As we can see, while the pictures do give an idea of what



(a) The inner vertices have coordinates $(30, 20)$ and $(46, 16)$.

(b) The inner vertices have coordinates $(30, -10)$ and $(40.5, -12.6)$.

(c) The inner vertices have coordinates $(37.5, 50)$ and $(28.13, 43.75)$.

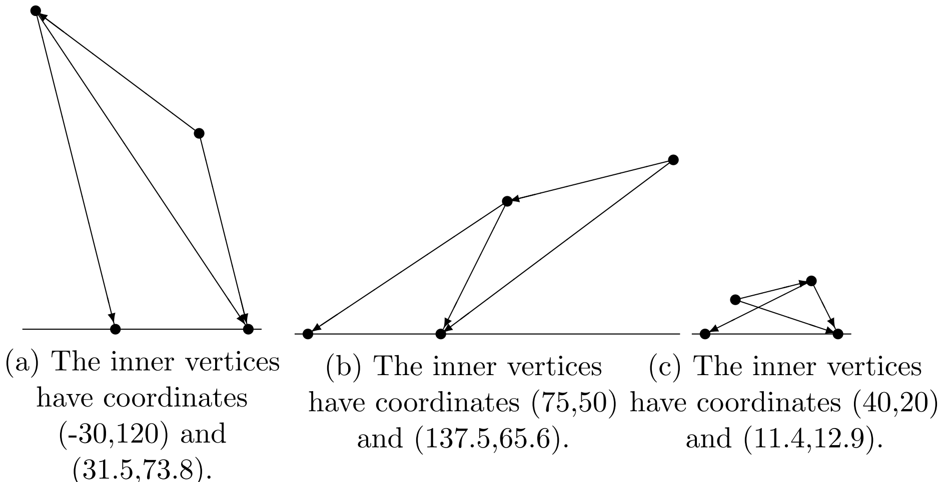
Figure 7: Three figures produced with the function `DrawGraph_algorithm`.

the graph looks like, but especially Figures 7b and 7c are far from aesthetically pleasing. Figure 7b is drawn completely below the line $y = 0$ and the edge from vertex 3 to 2 is too small for LaTeX to draw. Similarly, in Figure

7c the edge from vertex 3 to 1 and from 2 to 0 intersect. Moreover, the same edge as in Figure 7b is again too small to be drawn.

For Step 2 of the algorithm I split the different checks (so to check whether the solution has vertices below $y = 0$, too short edges, two overlapping vertices or overlapping edges) into functions, you can find them in Sections A.2.5 to A.2.8, each outputting a value True or False, indicating whether the condition is satisfied or not. I have then written the function `DrawGraph_filter` (see Section A.2.9) which computes a solution and then checks, using the above mentioned functions, whether it satisfies the conditions. If the solution doesn't satisfy the conditions, it recomputes a different solution for as long as it takes to satisfy all the conditions.

The function `DrawGraph_filter` gives us already more acceptable coordinates for the graphs. Three possible drawings of the graph with encoding 2 2 1 01 12 are given in 8.



(a) The inner vertices have coordinates (-30,120) and (31.5,73.8).

(b) The inner vertices have coordinates (75,50) and (137.5,65.6).

(c) The inner vertices have coordinates (40,20) and (11.4,12.9).

Figure 8: Three drawings produced with `DrawGraph_filter`.

While these drawings already look a lot better than the ones in Figure 7, they are for one all above the line $y = 0$ and there are no edges that are too short, we can't yet say they look very pretty. Subfigure 8a is too high, Subfigure 8b is leaning too far to the right and Subfigure 8c has an unnecessary intersection of lines.

After this I wrote the functions `DrawGraph_compute_and_draw` and `DrawGraph_draw` (Sections A.2.10 and A.2.11). These functions are not entirely necessary for

the task, but they do help me get an idea of what the graphs would look like when drawn, without needing to put the entire picture in LaTeX. The first function has as input the encoding of a graph, the distance δ between the sinks and a value `min_len` which gives the minimum length for a line to be drawn in LaTeX. This function computes a set of coordinates for the graph satisfying the conditions of Step 2 of the algorithm, and then outputs a plot of the graph. The function `DrawGraph_draw` has as input the encoding of a graph and a set of acceptable coordinates. It outputs a plot of the graph. In Figure 9 a graph computed with `DrawGraph_compute_and_draw` is shown. Graphs plotted with `DrawGraph_draw` look similar.

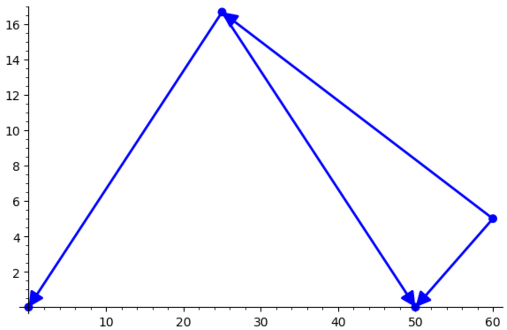


Figure 9: A graph plotted with `DrawGraph_compute_and_draw`.

This wrapped up Step 2 of the algorithm plus some extra things. I started working on the target function as defined in the equations 4 to 7 below.

$$
\begin{aligned}
TF(\ell_e, \ell_v, v_x, v_y, N, M, K) = {} & SLS(\ell_e) + IS(N) + UVS(\ell_v) \\
& + HYS(v_y) + OS(v_x) + PLS(M) + VLS(K)
\end{aligned}
\tag{4}
$$

where TF stands for target function, SLS for short and long edges score:

$$
SLS(\ell_e) = \sum_{i=1}^{2n} c_{se} \cdot \left(\frac{\ell_{e_i}}{\ell_0}\right)^{\alpha} + c_{le} \cdot \left(\frac{\ell_0}{\ell_{e_i}}\right)^{\beta}, \tag{5}
$$

IS for intersections score:

$$
IS(N) = N^{\gamma} \cdot (\log N)^{\epsilon}, \tag{6}
$$

UVS for unrelated vertices score:

$$UVS(\ell_v) = \sum_{i=1}^{n} \sum_{\substack{j=i \\ \text{no edge from } i \text{ to } j}}^{n} c_{dv} \cdot \left(\frac{\ell_1}{\ell_{v_{i,j}}}\right)^{\zeta}, \tag{7}$$

HYS for height y-coordinate score:

$$HYS(v_y) = \sum_{i=1}^{n} c_{hy} \cdot \left(\frac{\ell_2}{v_{y,i}}\right)^{\theta}, \tag{8}$$

OS for overshoot score:

$$OS(v_x) = \sum_{i=1}^{n} c_{os} \cdot \begin{cases} |v_{x,i}|^{\iota} & \text{if } v_{x,i} < 0 \\ (v_{x,i} - \delta)^{\iota} & \text{if } v_{x,i} > \delta \\ 0 & \text{if } 0 \leq v_{x,i} \leq \delta \end{cases}, \tag{9}$$

PLS for points on lines score:

$$PLS(M) = -M^{\kappa} \cdot (\log M)^{\lambda} \tag{10}$$

and VLS for vertical lines score:

$$VLS(K) = -K^{\mu} \cdot (\log K)^{\nu}. \tag{11}$$

In the equations above, ℓ_{e_i} gives the length of edge i, $\ell_{v_{i,j}}$ gives the distance from vertex i to vertex j, $v_{x,i}$ and $v_{y,i}$ give respectively the x and y-coordinate of the $i^t h$ vertex, N gives the number of intersections of edges (plus one, see Section 5.2 for details) in the drawing of a graph, M gives the number of times three vertices lie on one line (also plus 1, see Section 5.6 for details) and finally K gives the number of times 2 vertices are positioned right above each other (again plus 1, see Section 5.7 for details).

The programmed target function (see Section A.2.14) takes as input the encoding of a graph and a set of acceptable coordinates. It outputs a numerical value denoting how 'beautiful' the drawing of the graph is. The lower the value the more beautiful the drawing is. In the above, $\ell_0, c_{se}, c_{le}, \alpha, \beta, \gamma$ and ϵ are all parameters.

To check whether two lines of a graph intersect I made the function `intersection_test` (see Section A.2.13). The function uses a 2-dimensional

form of the method for finding whether two line segments intersect in a 3-dimensional space by Ronald Goldman in [6] (the actual method I used was explained on [15]). For this method the endpoints of the line segments are written in vector form as $\mathbf{p}$ and $\mathbf{p} + \mathbf{r}$, and $\mathbf{q}$ and $\mathbf{q} + \mathbf{s}$. Every point on the first line segment can now be written as $\mathbf{p} + t\mathbf{r}$ and every point on the second line segment can be written as $\mathbf{q} + u\mathbf{s}$, where t and u are scalars. Define a cross product $\mathbf{v} \times \mathbf{w} = v_x w_y - v_y w_x$. If the line segments intersect, then $\mathbf{p} + t\mathbf{r} = \mathbf{q} + u\mathbf{s}$ for some t and u. Crossing both sides with $\mathbf{s}$ gives $t(\mathbf{r} \times \mathbf{s}) = (\mathbf{q} - \mathbf{p}) \times \mathbf{s}$ (since $\mathbf{s} \times \mathbf{s} = 0$). It follows that:

$$t = \frac{(\mathbf{q} - \mathbf{p}) \times \mathbf{s}}{\mathbf{r} \times \mathbf{s}}.$$

Similarly, we can find

$$u = \frac{(\mathbf{q} - \mathbf{p}) \times \mathbf{r}}{\mathbf{r} \times \mathbf{s}}.$$

Since we do not need to worry about overlapping edges, because those have been filtered out already, we are only considered in intersections of lines that are not parallel/co-linear. So, if $\mathbf{r} \times \mathbf{s} \neq 0$ (because the line segments are parallel if and only if $\mathbf{r} \times \mathbf{s} = 0$), $0 \leq t \leq 1$ and $0 \leq u \leq 1$, then the line segments intersect at $\mathbf{p} + t\mathbf{r} = \mathbf{q} + u\mathbf{s}$.

After I wrote the target function, I wrote 2 more functions, `generate_list_of_coordinates` and `choosing_best_five_pictures` (see Sections A.2.15 and A.2.16). The function `generate_list_of_coordinates` has as input the encoding of a graph, the distance δ between the sinks, the value `min_len` that gives the minimum length for a line to be drawn in LaTeX and `num_of_iterations` and integer that indicates how long the outputted list of coordinates must be. It outputs a list of length `num_of_iterations` containing different solutions to the system.

The function `choosing_best_five_pictures` has the same four inputs, but uses `generate_list_of_coordinates` to compute a list of coordinates, deletes all duplicates in the list and it then uses the target function on the remaining solutions to determine the five sets of coordinates that have the lowest score, i.e. the five 'most beautiful' drawings.

I will use these functions to see what kind of drawings get a low score from the target function.

5 The Target Function

In this section I will take a look at the different components of the target function and how the parameters influence them. The Sections 5.1 to 5.3 consider the parts of the target function that were mentioned in the algorithm, Sections 5.4 to 5.7 contain parts added to the target function at a later stage.

5.1 The Short and Long Edges Score

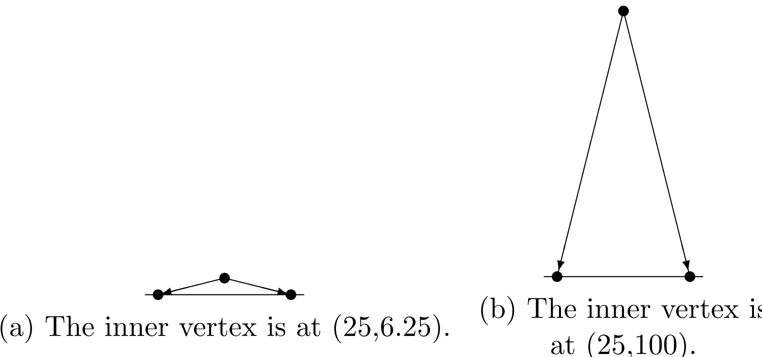
For each edge with length ℓ in the graph, the function adds a value $c_{se} \cdot (\ell/\ell_0)^\alpha$ and a value $c_{le} \cdot (\ell_0/\ell)^\beta$. I will only consider values of α and β larger than 0. It seems to be pretty clear that increasing the value of c_{se}, respectively c_{le} will make the component $(\ell/\ell_0)^\alpha$, respectively $(\ell_0/\ell)^\beta$ more important. In other words, if c_{se} increases, drawings of the graph with long edges, i.e. edges of length $\ell > \ell_0$, will get a much higher score than drawings of the graph with edges of a length ℓ much smaller than ℓ_0. Consequently, graphs that get 'good' scores will have short edges. Similarly, if c_{le} increases, drawings of the graph with short edges, i.e. edges of length $\ell < \ell_0$, will get a much higher score than drawing of the graph with edges of a length ℓ much larger than ℓ_0. So in this case, graphs with long edges will get low scores. I will try to illustrate this in the example below.

Example 6. Let's first start with taking $\ell_0 = 50$ and $\alpha = \beta = 1$. Let's take a look at the wedge (the graph with encoding 2 2 1 01). Two possible drawings of the graph are shown in Figure 10. The drawing in Subfigure 10a has two edges, both of length $\ell_s = \sqrt{25^2 + 6.25^2} \approx 25.76$. The edges of the drawing in Subfigure 10b are both of length $\ell_l = \sqrt{25^2 + 100^2} \approx 103.08$. In these cases, the target function score for the first drawing will be

$$2c_{se} \cdot \left(\frac{25.76}{50}\right) + 2c_{le} \cdot \left(\frac{50}{25.76}\right) \approx 1.03c_{se} + 3.88c_{le}.$$

Since c_{le} has a greater 'weight' than c_{se}, the drawing will get a much higher score when c_{le} is high than when c_{se} is high. For the second drawing, the opposite is true:

$$2c_{se} \cdot \left(\frac{103.08}{50}\right) + 2c_{le} \cdot \left(\frac{50}{103.08}\right) \approx 2.06c_{se} + 0.49c_{le}.$$



(a) The inner vertex is at (25,6.25). (b) The inner vertex is at (25,100).

Figure 10: Two drawings of the wedge, the coordinates for the sinks are at $(0,0)$ and $(50,0)$.

In this case the first term has the greater 'weight'. So the drawing will in this case get a higher score when c_{se} is high.

Let's now look at how α and β influence the score of the target function. If we increase α, then $(\ell/\ell_0)^\alpha \to 0$ if and only if $\ell < \ell_0$. But we also have the term $(\ell_0/\ell)^\beta$, which is minimised if $\ell > \ell_0$. If, however, α is much larger than β, then taking a value $\ell > \ell_0$ will make the score very large. So, to get a low score we need $\ell < \ell_0$. However, it doesn't matter a lot how much smaller ℓ is than ℓ_0, because a number smaller than 1 to a positive power will always remain smaller than 1. It will hence not have a big effect on the score. However a number larger than 1 to a power larger than 1 can blow up very quickly. Hence, the lowest scores will be reached when ℓ is just a little less than ℓ_0. Let us see this in an example:

Example 7. Take $\alpha = 10$ and $\beta = 2$, also set $\ell_0 = 50$ again. (We set $c_{se} = c_{le} = 1$ for now.) Consider 3 values of ℓ, $\ell_1 = 10$, $\ell_2 = 49$ and $\ell_3 = 50$. The score that the first edge (with length ℓ_1) gets in the target function is:

$$\left(\frac{10}{50}\right)^{10} + \left(\frac{50}{10}\right)^{2} = 0.2^{10} + 5^2 = 1.024 \cdot 10^{-7} + 25 \approx 25.$$

The second edge (with length ℓ_2) gets:

$$\left(\frac{49}{50}\right)^{10} + \left(\frac{50}{49}\right)^{2} \approx 0.98^{10} + 1.02^2 = 0.82 + 1.04 = 1.86.$$

And the third edge (with length ℓ_3) gets:

$$\left(\frac{50}{50}\right)^{10} + \left(\frac{50}{50}\right)^{2} = 1^{10} + 1^2 = 2.$$

It is clear that lengths larger than 50 lead to greater values of the target function, but the first calculation shows that lengths a lot less than 50 also lead to greater values.

I start to wonder what the optimal value for the length of an edge is in order to minimise the target function. The function $TF(\ell) = c_{se} \cdot (\ell/\ell_0)^{\alpha} + c_{le} \cdot (\ell_0/\ell)^{\beta}$ is differentiable for $\ell \neq 0$ (and we only consider positive values of ℓ anyway):

$$\frac{dTF}{d\ell}(\ell) = \frac{c_{se}}{\ell_0^{\alpha}} \cdot \alpha \cdot \ell^{\alpha-1} - c_{le} \cdot \ell_0^{\beta} \cdot \beta \cdot \ell^{-\beta-1}.$$

The minimum occurs if the derivative is zero. In this case, it equals:

$$\frac{c_{se}}{\ell_0^{\alpha}} \cdot \alpha \cdot \ell^{\alpha-1} = c_{le} \cdot \ell_0^{\beta} \cdot \beta \cdot \ell^{-\beta-1}.$$

If we move the terms with ℓ to the left and all the rest to the right, we get:

$$\ell^{\alpha+\beta} = \frac{\beta c_{le}}{\alpha c_{se}} \cdot \ell_0^{\alpha+\beta}.$$

So we get as solution for ℓ:

$$\ell = \left(\frac{\beta c_{le}}{\alpha c_{se}}\right)^{\frac{1}{\alpha+\beta}} \cdot \ell_0. \tag{12}$$

So if we take the values for the parameters from Example 7 we get:

$$\ell = \left(\frac{2}{10}\right)^{\frac{1}{12}} \cdot 50 \approx 43.72.$$

And indeed, if we fill this ℓ into TF, we get $TF(\ell) = 1.57$ which is lower than any of the numbers we calculated in Example 7. As we can seen in Equation 12, the optimal value of ℓ is higher or lower than ℓ_0 dependent on the values of the other parameters.
From Equation 12 we can also conclude that when $\alpha + \beta > 1$ the optimal value of ℓ will be relatively close to ℓ_0. Because in that case, the fraction will

be raised to a power less than 1, which means that the resulting number will be closer to 1 than the original fraction. Similarly, if $\alpha + \beta < 1$, the optimal value will be relatively farther away from ℓ_0, since then, the fraction will be raised to a power greater than 1, which means that fractions less than 1 will get smaller and fractions larger than 1 bigger.

Another thing we should notice is that when α and β are less than one, there will be a lesser difference between the values of the two fractions. Indeed, if $\alpha < 1$ and $\beta < 1$ and the fraction $x := (\ell/\ell_0) < 1$ is also less than 1, then $x < x^\alpha < 1$. So x^α goes to 1. However, in this case the fraction for β is $(\ell/\ell_0) = 1/x > 1$. Which means that $1 < (1/x)^\beta < 1/x$, so $(1/x)^\beta$ also gets closer to 1. It follows that the difference between the fractions gets smaller. This will lead to less strict bounds on the length of the edges of the graph.

5.2 Intersections Score

The part of the target function that increases if a drawing of a graph has intersections is given by $N^\gamma \cdot (\log N)^\epsilon$, where N is the number of intersections. Because $\log 0$ is undefined, N starts counting at 1, meaning that $N = 1$ if there are no intersections, $N = 2$ when there is one intersection, etc. It is clear that the function is minimised if there are no intersections, but how do γ and ϵ influence the value of the function?

An increase of γ or ϵ above 1 will, if there are intersections, always increase the value of the function. However $\log N < N$ for all N. This means that increasing ϵ will have a smaller effect on the value of the target function than increasing γ.

Example 8. Take $N = 3$, $\log 3 \approx 1.1$. So, $3 \cdot \log 3 \approx 3.3$. If γ is set at 2, then the value will be $3^2 \cdot \log 3 \approx 9.9$. However, if ϵ is 2, the value is $3 \cdot (\log 3)^2 \approx 3.6$. It is clear that increasing ϵ has a lesser effect on the value of the target function than increasing γ.

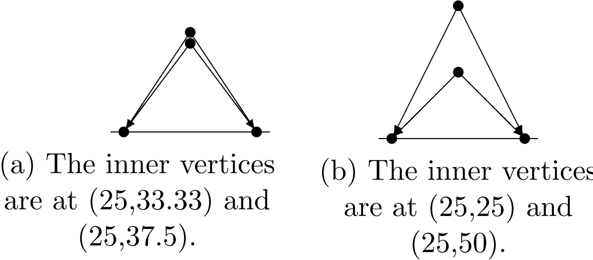
It is interesting to look at what happens when γ or ϵ gets decreased to below 1 (I do not consider $\gamma, \epsilon \leq 0$). The decrease of the value will be bigger if γ is smaller than 1 then when $\epsilon < 1$.

Example 9. Take again $N = 3$. If we take $\gamma = 1/2$, then the value will be $3^{1/2} \cdot \log 3 \approx 1.9$. If instead $\epsilon = 1/2$, the value will be $3 \cdot (\log 3)^{1/2} \approx 3.1$. That's only a small decrease from the value 3.3 when $\gamma = \epsilon = 1$!

So the value of γ has a larger influence on the value of the target function than the value of ϵ.

5.3 Distance Between Unrelated Vertices Score

The target function increases with $c_{dv} \cdot (\ell_1/\ell)^\zeta$ for any two vertices with no edge between them, where ℓ is the distance between them. c_{dv}, ℓ_1 and ζ are parameters. The goal of this term is to make sure unrelated vertices will be drawn at a reasonable distance from each other, and not close together. For an illustration, using the graph with encoding 2 2 1 01 01 as an example, see Figure 11. It is clear that the drawing in Subfigure 11a doesn't look as nice



(a) The inner vertices are at (25,33.33) and (25,37.5).

(b) The inner vertices are at (25,25) and (25,50).

Figure 11: Two drawings of the graph with encoding 2 2 1 01 01.

as the one in Subfigure 11b, because the inner vertices of the former are too close together. Also notice that the function will only seriously increase if the vertices are too close together, there is no upper bound for the distance between unrelated vertices.

The value of c_{dv} increases the importance of this term of the target function, e.g. if $c_{dv} \gg c_{se}$, then the target function might assign low scores to drawings where unrelated vertices are far apart, even if it means that the edges are shorter than ℓ_0.

Example 10. Let $\ell_0 = 50$, $\ell_1 = 10$ and $\alpha = \zeta = 1$. Compare the drawings from Figure 11. The graph has 4 edges, in the first drawing, 2 edges are of length $\sqrt{25^2 + 33.33^2} \approx 41.67$ and 2 are of length $\sqrt{25^2 + 27.5^2} \approx 45.07$, the distance between the unrelated inner vertices is 4.17.
In the second drawing, 2 edges are of length $\sqrt{2 \cdot 25^2} \approx 35.36$ and 2 of length $\sqrt{25^2 + 50^2} \approx 55.90$, the distance between the unrelated inner vertices is 25. The the value of

$$\sum_{i=1}^{n} c_{se} \cdot \left(\frac{\ell_{e_i}}{50}\right) + \sum_{i=1}^{n} \sum_{\substack{j=i \\ \text{no edge from } i \text{ to } j}}^{n} c_{dv} \cdot \left(\frac{10}{\ell_{v_{i,j}}}\right)$$

for the first drawing will be:

$$c_{se} \cdot \left(\frac{2 \cdot 41.67 + 2 \cdot 45.07}{50} \right) + c_{dv} \cdot \left(\frac{10}{4.17} \right) = c_{se} \cdot 3.47 + c_{dv} \cdot 2.40.$$

For the second drawing it will be:

$$c_{se} \cdot \left(\frac{2 \cdot 35.36 + 2 \cdot 55.90}{50} \right) + c_{dv} \cdot \left(\frac{10}{25} \right) = c_{se} \cdot 3.65 + c_{dv} \cdot 0.4.$$

Now suppose $c_{se} = 15$ and $c_{dv} = 1$, then the first value will be $52.05 + 2.40 = 54.05$ and the second value will be $36.5 + 0.4 = 55.15$. The first value is lower and would be considered better, but if the difference between c_{se} and c_{dv} would be less, the second value would be lower, and hence the second drawing would be considered better.

The value of ζ determines mainly how much ℓ can be below ℓ_1. If $\zeta > 1$, and $\ell < \ell_1$, then $(\ell_1/\ell)^\zeta > (\ell_1/\ell) > 1$. However, if $\ell > \ell_1$, then $(\ell_1/\ell)^\zeta < (\ell_1/\ell) < 1$. So, if ζ get larger, values of ℓ larger than ℓ_1 will make sure the value gets small.
If $\zeta < 1$, and $\ell < \ell_1$, then $1 < (\ell_1/\ell)^\zeta < (\ell_1/\ell)$. Which means that the value of the function will not be very high, especially if ℓ is not much less than ℓ_1.

Example 11. Consider again the drawings from Figure 11. For the first drawing the distance between the unrelated vertices is 4.17, for the second it is 25. Set again at $\ell_1 = 10$ and suppose $c_{dv} = 1$. The value for the first drawing will be:

$$\left(\frac{10}{4.17} \right)^\zeta = 2.40^\zeta.$$

The value for the second drawing will be:

$$\left(\frac{10}{25} \right)^\zeta = 0.4^\zeta$$

Now it's clear that if $\zeta > 1$, the first value will become even larger, but the second value will go to 0. However, if $\zeta < 1$, for example, $\zeta = 0.25$, then the first value will be 1.25 and the second will be 0.80. The difference between the two values will be less.

5.4 Overshoot Score

For every inner vertex that has an x-coordinate smaller than 0 or larger than δ the value $c_{os} \cdot |v_x|^{\iota}$, respectively $c_{os} \cdot (v_x - \delta)^{\iota}$. This score will make sure that the vertices will not be drawn very far outside the lines $x = 0$ and $x = \delta$. Like in most of the previous terms of the target function, c_{os} determines the importance of the overshoot score with respect to the other terms. If $\iota > 1$, any value of v_x more than 1 point away from the bounds will get a relatively high score. At the same time, if v_x is less than one point away from the bounds, a value of ι larger than 1 will make the score become relatively small. If $\iota < 1$, any value of v_x will (especially if $\iota \ll 1$) lead to a score close to 1 (times c_{os}).

5.5 Height y-coordinate Score

For every vertex the target function will increase with $c_{hy} \cdot (\ell_2 / v_y)^{\theta}$ where v_y is the y-coordinate of the vertex and c_{hy}, ℓ_2 and θ the parameters. ℓ_2 will be set at about $0.4 \cdot \ell_0$. This term will make sure that the graphs will not be drawn too flat. The height y-coordinate score looks a lot like the distance between unrelated vertices score discussed in the previous section. The parameter c_{hy} determines the importance of the term compared to the other terms of the target function. The parameter θ determines how bad it is if the drawing has vertices lower than ℓ_2. Finally ℓ_2 determines what is considered low, the higher the value of ℓ_2, the higher the vertices will have to be in order for the drawing to get a good score. Let's look at a short example.

Example 12. Let's consider the two drawings of Example 6 of the wedge with inner vertex at $(15, 6.25)$ and $(25, 100)$ (See Figure 10 for an illustration) and set $\ell_2 = 20$. The first drawing will get the score

$$c_{hy} \cdot \left(\frac{20}{6.25}\right)^{\theta} = c_{hy} \cdot 3.2^{\theta}.$$

The second drawing will get the score

$$c_{hy} \cdot \left(\frac{20}{100}\right)^{\theta} = c_{hy} \cdot 0.2^{\theta}$$

It is clear that the second drawing will get the better score.

5.6 Points on Line Score

The points on line score *deducts* a value $M^\kappa \cdot (\log M)^\lambda$, where κ and λ are parameters and M is the number of times three points lie on one line in the drawing of a graph plus 1 (because if M starts counting at 0, the function would not be defined in all cases). Let us look at an example:

Example 13. Lets look at the graph with encoding 2 2 1 01 12 as drawn in Figure 12. In this drawing, the vertices with number 0, 2 and 3 are lying

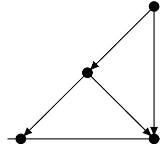


Figure 12: A drawing of the graph with encoding 2 2 1 01 12.

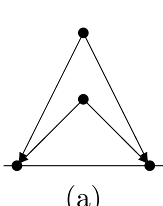
on one line. Hence, $M = 2$ and the points on line score is $-2^\kappa \cdot (\log 2)^\lambda$.

The goal of the points on line score is to give drawings that have multiple points lying on one line a lower score. This will make sure that drawings that, for example, might have slightly too long edges, but have at least 3 points on one line will nevertheless get a good score.

5.7 Vertical Lines Score

The vertical lines score deducts a value $K^\mu \cdot (\log K)^\nu$, where μ and ν are parameters and K is the number of times two points are positioned right above each other plus one (for the same reason as we start counting the N from the intersections score and the M from the points on line score at 1, otherwise $\log K$ would not be defined for all possible values of K). Let's look at an example of what I mean with vertical lines:

Example 14. Let's look at the graph with encoding 2 2 1 01 01 and its two drawings in Figure 13. Both these drawings have 2 vertices with a vertical 'line' between them. In Figure 13a, the vertices that are straight above each other are vertex 2 and 3. In Figure 13b vertices 0 and 3 are straight above each other. It is not necessary for the vertical line between vertices to be actually drawn in order to count for the vertical lines score. Both drawings in Figure 13 have vertical lines score $-2^\mu \cdot (\log 2)^\nu$.


(a)

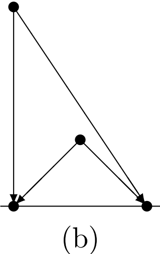

(b)

Figure 13: Two drawings of the graph with encoding 2 2 1 01 01.

5.8 Concluding Remarks and Summary

It is important to keep in mind that all the different terms influence each other. We could for example increase c_{le} in the hope of getting shorter edges, but if c_{hy} is very large, the effectiveness of c_{le} might not be what one hoped for.

In summary:

- **Short and Long Edges Score**
  - c_{se}: Increasing leads to graphs with long edges, decreasing leads to graphs with shorter edges.
  - c_{le}: Increasing leads to graphs with short edges, decreasing lead to graphs with longer edges.
  - α: Increasing above 1 leads to graphs with edges of length $\ell < \ell_0$. The larger the value of α, the closer the length of the edges gets to ℓ_0. Decreasing below 1 leads to less strict bounds on the lengths of the edges.
  - β: Increasing above 1 leads to graphs with edges of length $\ell > \ell_0$. Decreasing below 1 leads to less strict bounds on the lengths of the edges.
- **Intersections Score**
  - Increasing ϵ has a lesser effect than increasing γ. But increasing either above 1 leads to fewer intersections in the graphs.
  - Decreasing γ below 1 can lead to more intersections.
- **Distance Between Unrelated Vertices Score**

- Increasing c_{dv} increases the importance of the term, it leads to graphs with unrelated vertices being situated further apart. Decreasing c_{dv} leads to unrelated vertices being situated closer together.
- Increasing ζ above 1 leads to unrelated vertices being further apart than a length ℓ_1. Decreasing ζ below 1 will lead to unrelated vertices that are situated closer together.

- **Overshoot Score**
  - Increasing c_{os} increases the importance of the term, it leads to vertices only within the interval $[0, \delta]$. Decreasing c_{os} can lead to vertices outside the mentioned interval.
  - Increasing ι above 1 will lead to vertices being situated within the interval. Decreasing ι below 1 will lead to vertices being more likely to be situated outside of the interval.

- **Height y-Coordinate Score**
  - Increasing c_{hy} leads to higher vertices, decreasing to lower.
  - Increasing θ above 1 leads to vertices with an y-coordinate above ℓ_2. Decreasing θ below 1 gives a greater likelihood of vertices having a y-coordinate below ℓ_2.

- **Points on Line Score**
  - Increasing κ or λ above 1 both lead to more points lying on one line. Although the effect of κ is larger than the effect of λ. Decreasing them below 1 leads to less points lying on one line.

- **Vertical Lines Score**
  - Increasing μ or ν above 1 both lead to more vertices lying right above each other. Although the effect of μ is larger than the effect of ν. Decreasing them below 1 leads to less points lying right above each other.

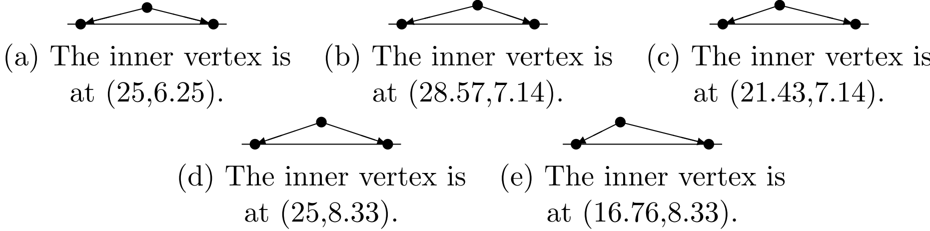
6 Tuning the Target Function

Before starting to tune the target function, recall that we always have 2 sinks, the left one always being in $(0, 0)$ and the right one being at $(\delta, 0)$.

I started out with the target function having parameters: $\ell_0 = \ell_1 = 10, c_{se} = c_{le} = c_{dv} = 1, \alpha = \beta = \gamma = \epsilon = \zeta = 1$. If you read Section 5, you might notice that I omitted some parameters. This is because I will only add the other scores later in this section. Therefore, the unmentioned parameters are not used yet. Please recall that `\unitlength=1pt`, so that the value of ℓ is in points. (1 point is approximately 0.35 mm.)

6.1 The Wedge

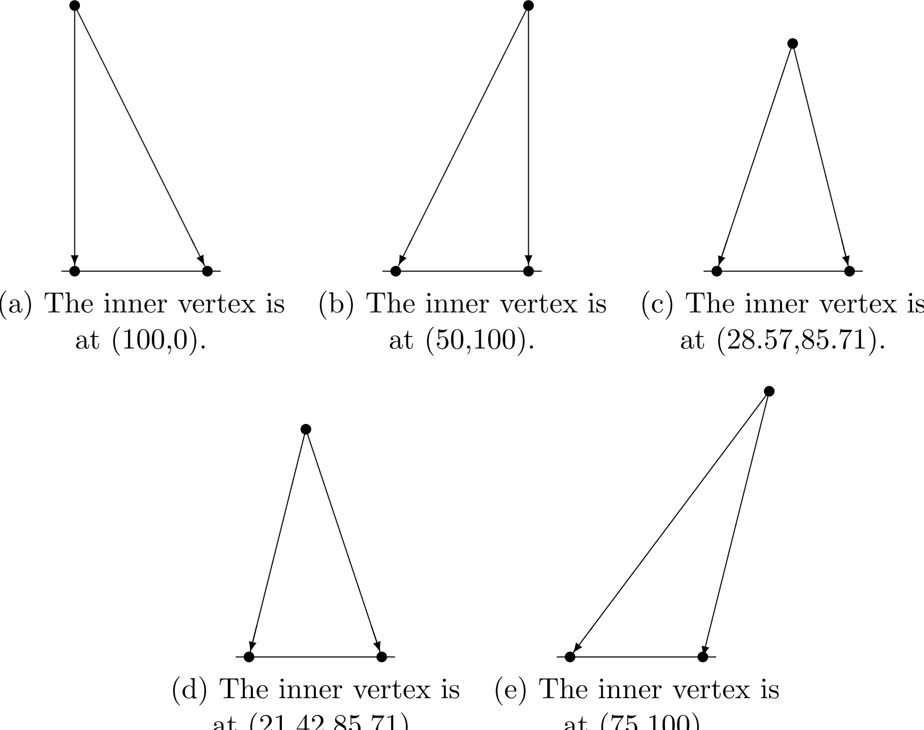
I started with tuning the function for the easiest graph, the wedge. I set the value of delta at 50 and with the starting parameters this gave me the pictures in Figure 14.



(a) The inner vertex is at (25,6.25).
(b) The inner vertex is at (28.57,7.14).
(c) The inner vertex is at (21.43,7.14).
(d) The inner vertex is at (25,8.33).
(e) The inner vertex is at (16.76,8.33).

Figure 14: The five drawings of the wedge with the initial parameters $\ell_0 = \ell_1 = 10$, $c_{se} = c_{le} = c_{dv} = 1$ and $\alpha = \beta = \gamma = \epsilon = \zeta = 1$.

For me, two things stand out in these drawings. Firstly, not all drawings are symmetrical, in fact, only Sub-figures 14a and 14d are symmetrical. (Also note that 14b and 14c are mirror images of each other.) While I would definitely like all figures to be symmetrical, a bigger issue at the moment is that the inner vertex is positioned very low in all pictures, making the graphs very flat. I think that that is because of the value of ℓ_0. Right now $\ell_0 = 10$, however 10 points is also the minimal length for any line that LaTeX can draw, making all possible solutions to the system having 'too long' lines according to the target function. To get a good view of what will happen if the value of ℓ_0 gets bigger I will first make ℓ_0 a lot larger. I set the value of ℓ_0 at 100. The results can be seen in Figure 15.



(a) The inner vertex is at (100,0).

(b) The inner vertex is at (50,100).

(c) The inner vertex is at (28.57,85.71).

(d) The inner vertex is at (21.42,85.71).

(e) The inner vertex is at (75,100).

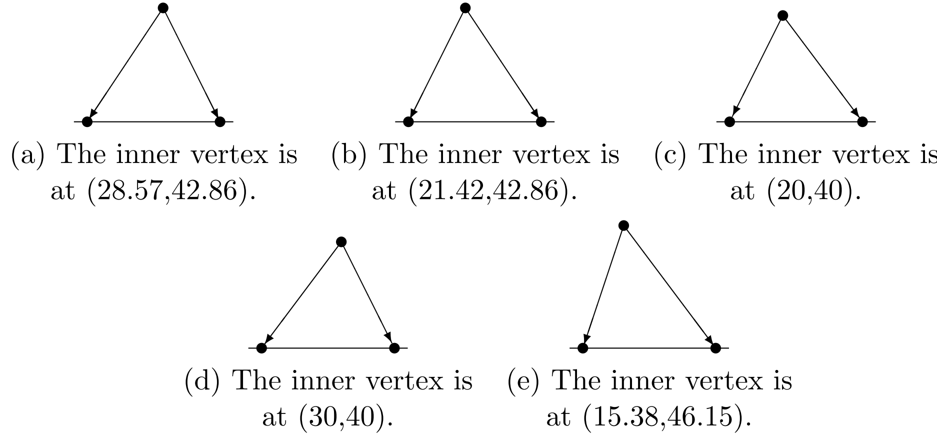
Figure 15: The five drawings of the wedge with $\ell_0 = 100$.

Now one can see that, for one, a value of 100 makes the lines way too long, and secondly, there is even less symmetry here, with none of the drawing being completely symmetrical. (But note that 15a and 15b are mirror images of each other and so are 15c and 15d.)

I think the drawings will be a lot prettier if they could be drawn in a square. Since the sinks are on a distance 50, we might like our edges to have a size of 50 too. The resulting drawings can be seen in Figure 16.

These drawings already look a lot better. I think that the optimal value of ℓ_0 might be a bit lower than 50, but for now this is good enough.

I will now look at what difference the other parameters make. To see this, I will change them one by one into a much larger number I will keep ℓ_0 at 50 for the moment, but I will now change c_{se} from 1 to 10. This gave as coordinates for the inner vertex: $(50, 150)$, $(0, 150)$, $(-50, 150)$, $(100, 150)$ and $150, 100$. Because $(50, 150)$ and $(0, 150)$ give the same drawing but mirrored and $(-50, 150)$ and $(100, 150)$ too, I will only draw the graph for

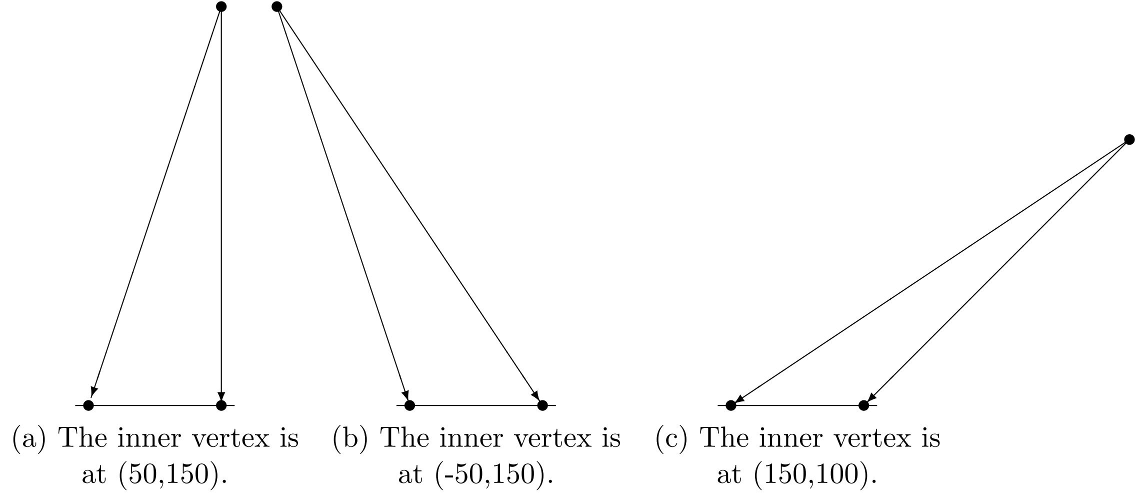


(a) The inner vertex is at (28.57,42.86). (b) The inner vertex is at (21.42,42.86). (c) The inner vertex is at (20,40).

(d) The inner vertex is at (30,40). (e) The inner vertex is at (15.38,46.15).

Figure 16: The five drawings of the wedge with $\ell_0 = 50$.

$(50, 150), (-50, 150)$ and $(150, 100)$. This gives the drawings in Figure 17.

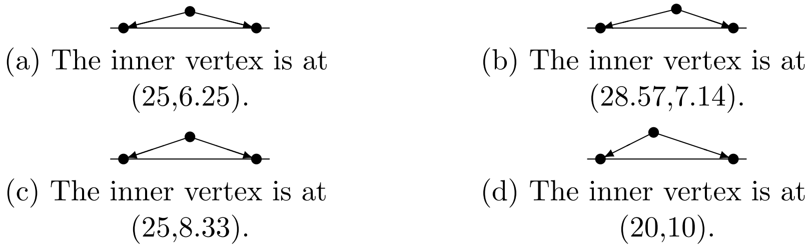


(a) The inner vertex is at (50,150). (b) The inner vertex is at (-50,150). (c) The inner vertex is at (150,100).

Figure 17: The five drawings of the wedge with $l_0 = 50$ and $c_{se} = 10$.

In this figure, we can see that a larger value of c_{se} gives graphs with longer edges. This is quite logical, since if c_{se} gets larger, in order for the score to stay low, the length of the edges will have to increase. Because if ℓ is large (larger than ℓ_0), $(\ell_0/\ell)^\alpha$ will get smaller, which means that the effect of the large c_{se} gets slightly negated.

Keeping this in mind I anticipate that increasing the value of c_{le} will lead to shorter edges, i.e. 'flat graphs'. I will set the value of c_{se} back at 1, but I will now increase the value of c_{le} to 10. This gives as values for the inner coordinates $(25, 6.25)$, $(28.57, 7.14)$, $(21.43, 7.14)$, $(25, 8.33)$ and $(20, 10)$. Since $(28.57, 7.14)$ and $(21.43, 7.14)$ are again mirror images I will only draw the first. The drawings can be seen in Figure 18.



(a) The inner vertex is at (25,6.25).

(b) The inner vertex is at (28.57,7.14).

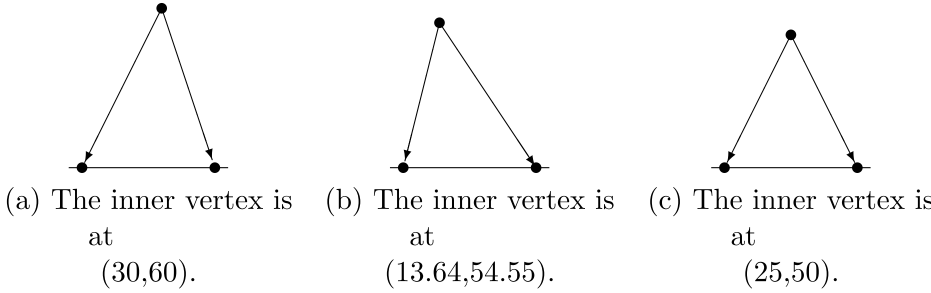
(c) The inner vertex is at (25,8.33).

(d) The inner vertex is at (20,10).

Figure 18: The five drawings of the wedge with $l_0 = 50$ and $c_{le} = 10$.

We can see that my prediction was correct, the drawings look a lot like the ones in Figure 14. (In fact they're largely the same.) Hence larger value of c_{le} gives drawings with short edges.

Let's now take a look at how changing α influences the drawings. I return the value of c_{le} to 1, but now I set $\alpha = 10$. This gives $(30, 60)$, $(20, 60)$, $(13.64, 54.55)$, $(36.36, 54.55)$ and $(25, 50)$ as coordinates for the inner vertices. The first coordinates are again in mirrored pairs, so I will only draw $(30, 60)$, $(13.64, 54.55)$ and $(25, 50)$. The drawings can be seen in Figure 19.



(a) The inner vertex is at (30,60).

(b) The inner vertex is at (13.64,54.55).

(c) The inner vertex is at (25,50).

Figure 19: The five drawings of the wedge with $l_0 = 50$ and $\alpha = 10$.

It appears that increasing α also leads to longer edges, though the increase is not as large as when c_{se} is increased. Let's increase α some more, let us set

$\alpha = 100$. This gives as coordinates for the inner vertex $(25, 50)$, $(16.67, 50)$, $(33.33, 50)$, $(37.5, 50)$ and $(12.5, 50)$. It is interesting to see that the in all cases the y-coordinate is 50. In fact, increasing α even more keeps giving the same coordinates. So, why is this? Let's take a look at the value of the target function for the drawing with inner vertex at $(25, 50)$ and the drawing with inner vertex at $(20, 40)$.

Example 15. The drawing with inner vertex at $(25, 50)$ is symmetrical and has hence two edges of equal length, $\ell = \sqrt{25^2 + 50^2} \approx 55.90$. If we fill this into the target function, we get the equation:

$$2 \cdot \left(\frac{50}{55.90}\right)^{\alpha} + 2 \cdot \left(\frac{55.90}{50}\right)^{\beta} \approx 2 \cdot 0.89^{\alpha} + 2 \cdot 1.12^{\beta}$$

As α increases, $2 \cdot 0.89^{\alpha} \to 0$ (0.89^{100} is already $8.7 \cdot 10^{-6}$), because $0.89 < 1$. So if $\beta = 1$ and α is large, the score will be around $2 \cdot 1.12 = 2.24$.

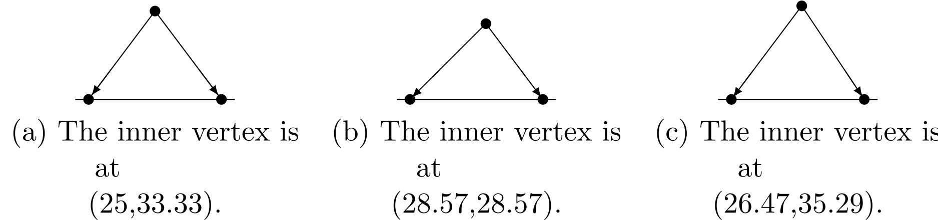
The drawing with inner vertex at $(20, 40)$ is not symmetrical and has edges of length $\sqrt{20^2 + 40^2} \approx 44.72$ and $\sqrt{(50-20)^2 + 40^2} = 50$. If we fill this into the target function, we get the equation:

$$\left(\frac{50}{44.72}\right)^{\alpha} + \left(\frac{50}{50}\right)^{\alpha} + \left(\frac{44.72}{50}\right)^{\beta} + \left(\frac{50}{50}\right)^{\beta} \approx 1.12^{\alpha} + 0.89^{\beta} + 2.$$

If α gets very large, the $0.89^{\beta} + 2$ part is not very significant anymore. The whole value will be big already ($1.12^{100} \approx 84 \cdot 10^3$), because $1.12 > 1$.

What can be concluded from these calculation is that, if α is large and the length of the edges is larger than 50, we can almost completely discount the first term in the target function and only focus on the term with β. That would mean that drawings with edges with a length close to 50 will get the lowest score from the target function. (Also note that if the length of the edges are 50, the score will be 4.) Drawings that have edges with a length that is less than 50 will automatically get a very large score from the target function and will hence not be among the three best drawings. In fact, if we make a list of all sets of coordinates that have only edges with a length longer than 50, we can find that that the 5 sets of coordinates that are given by the function `choosing_best_five_pictures`, are the five sets with the shortest edges. This can be found by changing line 40 in the code in Section A.2.17 into checking whether `len_L` or `len_R` are *smaller* than 50.

Because of these considerations, I anticipate that for β very large, the best five coordinates for the inner vertex are $(25, 37.5)$, $(26.47, 35.29)$, $(23.53, 35.29)$, $(33.33, 33.33)$ and $(16.67, 16.67)$. This was found by running the code in Section A.2.17.
But let us first look at what happens if $\alpha = 1$ and $\beta = 10$. In this case, we get $(25, 33.33)$, $(28.57, 28.57)$, $(21.43, 28.57)$, $(26.47, 35.29)$ and $(23.53, 35.29)$ as coordinates for the inner vertex. Since the last 4 are again two pairs of mirror images, I will only draw the graph for $(25, 33.33)$, $(28.57, 28.57)$ and $(26.47, 35.29)$ in Figure 20.



(a) The inner vertex is at (25,33.33).

(b) The inner vertex is at (28.57,28.57).

(c) The inner vertex is at (26.47,35.29).

Figure 20: Three of the best drawings of the wedge with $l_0 = 50$ and $\beta = 10$.

Similar as to how the resulting drawings have slightly longer edges when $\alpha = 10$, so do these drawings have slightly shorter edges than the ones in Figure 16. Another interesting feature is that the drawings seem to become more symmetric if we take measures to shorten the edges.

Let us now look at what happens if $\beta = 100$. For the inner vertex, we get coordinates $(25, 37.5)$, $(23.53, 35.29)$, $(26.47, 35.29)$, $(25, 33.33)$ and $(33.33, 33.33)$. With the exception of $(25, 33.33)$, these are indeed the coordinates that I expected. If I take a look at the rest of the list that came out of running the code in Section A.2.17 it appears that $(25, 33.33)$ is on the 6th place in the list. I assume that the difference in placement comes from the fact that while the average length of the edges of the drawing corresponding to $(33.33, 33.33)$ might be shorter than the length of the edges of the drawing corresponding to $(25, 33.33)$, but that one of the edges of $(33.33, 33.33)$ is longer than the edges of $(25, 33.33)$.

Now I would like to know what happens to the drawings if $\alpha < 1$. If I set $\alpha = 0.1$, the best five drawings have inner coordinates $(25, 6.25)$, $(21.43, 7.14)$, $(28.57, 7.14)$, $(25, 8.33)$ and $(16.67, 8.33)$. The two best drawings are given in Figure 21. As can be seen, setting α below 1 leads again to flat graphs. Similarly, setting β below 1 leads to very tall graphs.

(a) The inner vertex is at (25,6.25).

(b) The inner vertex is at (21.43,7.14).

Figure 21: The two best drawings of the wedge with $l_0 = 50$ and $\alpha = 0.1$.

The example of the wedge has given me a first idea of what the parameters in the target function do. Because the wedge has no possibilities for intersecting edges, or to change the distance between 2 unrelated vertices, I will now look at the graph with encoding 2 2 1 01 01, which, as seen in Figure 22, can have intersections and has two unrelated edges.

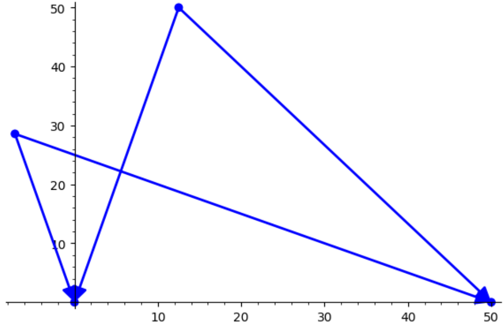
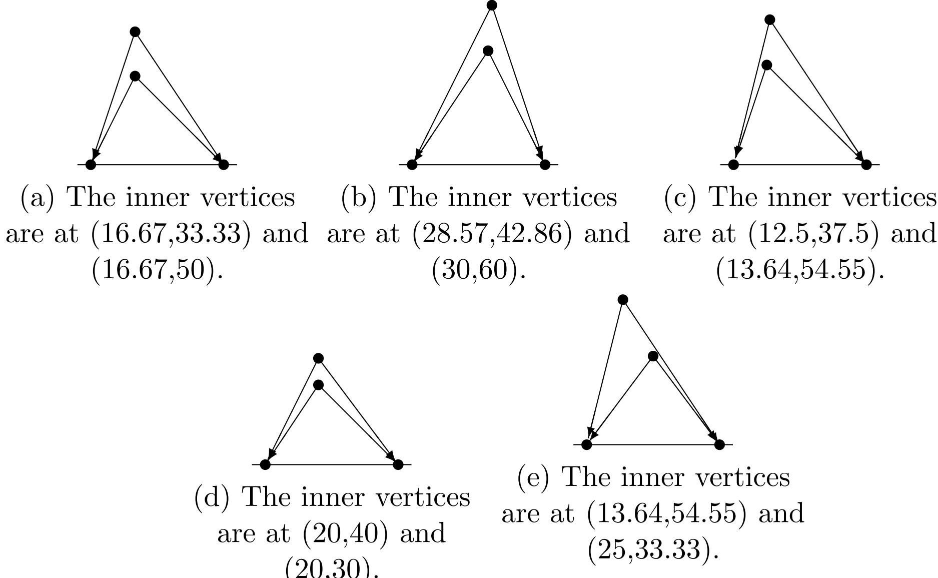


Figure 22: A drawing of the graph with encoding 2 2 1 01 01.

6.2 The Double Wedge, The Graph With Encoding 2 2 1 01 01

I will use this graph to start looking at the effects of γ and ϵ on the pictures. Recall that γ and ϵ influence the number of intersections in the pictures. To do this I will put all parameters back to 1, except from ℓ_0 which I will keep at 50 and I will set c_{dv} at 0, because I will ignore that term of the target function at first. The starting parameters will hence be: $\ell_0 = 50$, $\alpha = \beta = 1$, $c_{le} = c_{se} = 1$, $\epsilon = \gamma = 1$ and all other parameters are 0. Because there are more possible solutions to the system with this more complicated graph, I increase `num_of_iterations` to 2000. Note that even with this increase, the resulting pictures will differ a little each time, because there is no guarantee

that all possible options of inclines will be tried.[2] The 5 best drawings using these initial parameters are given in Figure 23.



(a) The inner vertices are at (16.67,33.33) and (16.67,50).

(b) The inner vertices are at (28.57,42.86) and (30,60).

(c) The inner vertices are at (12.5,37.5) and (13.64,54.55).

(d) The inner vertices are at (20,40) and (20,30).

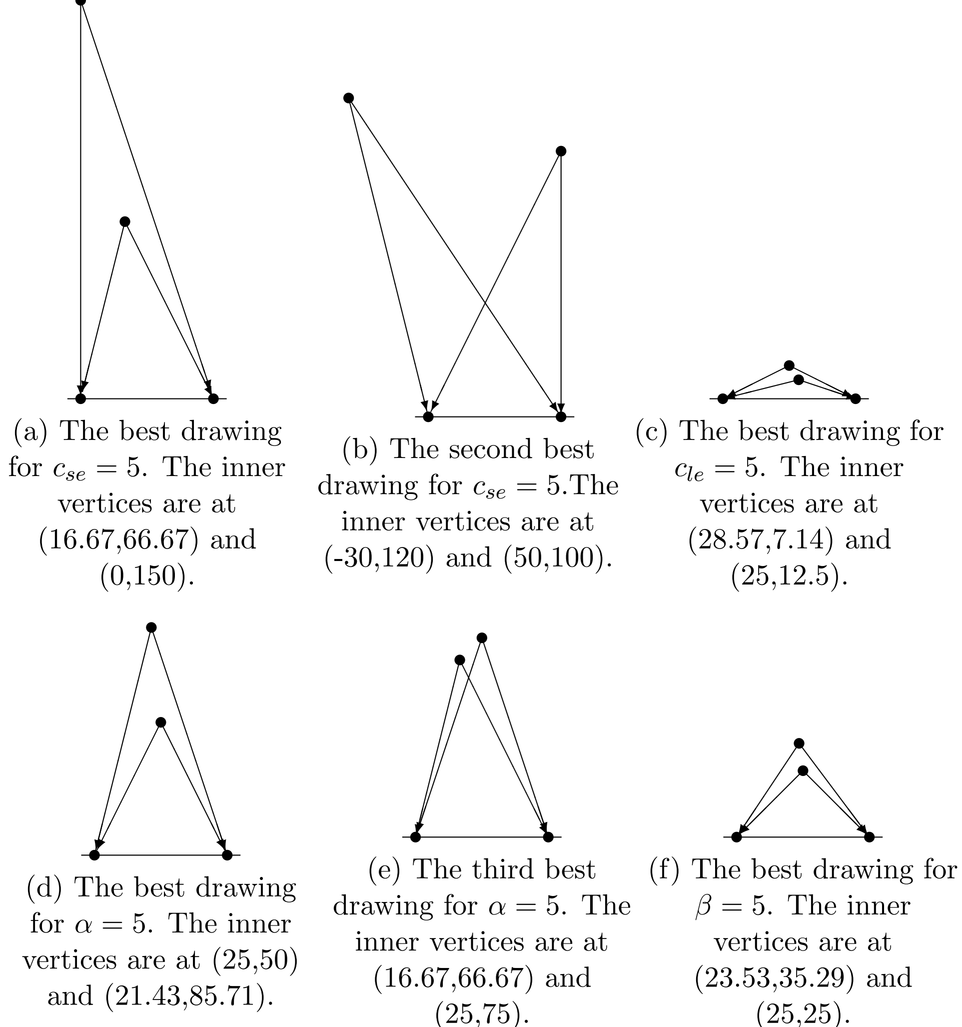
(e) The inner vertices are at (13.64,54.55) and (25,33.33).

Figure 23: The five drawings of the graph with encoding 2 2 1 01 01 with the initial parameters.

None of these drawings have an intersection. However, because I want to look at the effects of γ and ϵ, I will change the other parameters to see if I can get pictures with intersections. I set all parameters one by one to 5. The best drawing for all the parameters can be seen in Figure 24, in each drawing all parameters are 1, except the one explicitly mentioned in the caption. For $c_{se} = 5$ and $\alpha = 5$ I got an intersection for some of the 5 pictures given in the top 5. In those cases I also drew the best drawing with intersection. (See Subfigures 24b and 24e.)

Since I now first want to get some drawings with intersections, it seems like I have to use very tall drawings to get an drawings with an intersection. However, what if I combine a higher value for c_{se} and β or for c_{le} and α? Results of these can be seen in Figure 25, again all parameters are 1 unless otherwise indicated in the caption. Remember that while I, for ex-

[2]Recall that there are $552^2 = 304,704$ possible combinations of inclines. Later in this paper I will run through all possible inclines, but for now I will take 2000 random combinations for each run.



(a) The best drawing for $c_{se} = 5$. The inner vertices are at (16.67,66.67) and (0,150).

(b) The second best drawing for $c_{se} = 5$.The inner vertices are at (-30,120) and (50,100).

(c) The best drawing for $c_{le} = 5$. The inner vertices are at (28.57,7.14) and (25,12.5).

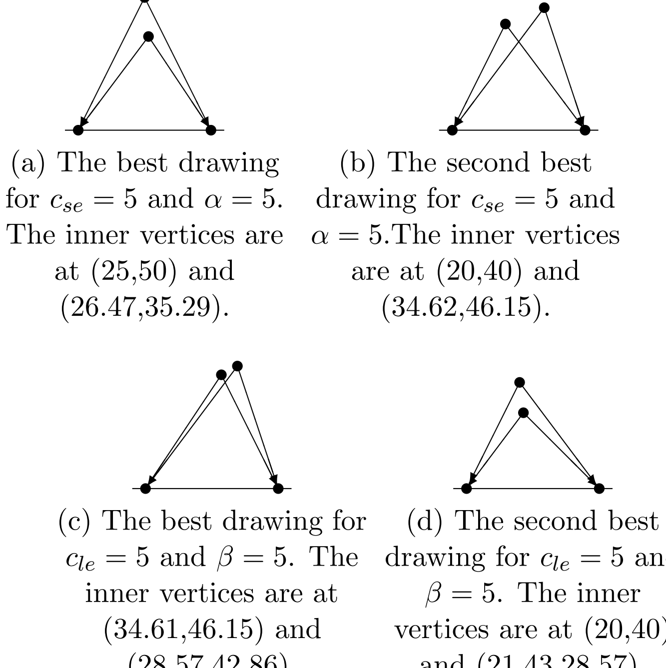
(d) The best drawing for $\alpha = 5$. The inner vertices are at (25,50) and (21.43,85.71).

(e) The third best drawing for $\alpha = 5$. The inner vertices are at (16.67,66.67) and (25,75).

(f) The best drawing for $\beta = 5$. The inner vertices are at (23.53,35.29) and (25,25).

Figure 24: Six drawings of the graph with encoding 2 2 1 01 01 with varying parameters.

ample, denote Subfigure 25a as the 'best' drawing for these parameters, this is simply the drawing that got the lowest score from the target function on this run. Moreover, because the given drawings change with each time I use `choosing_best_five_pictures`, it is highly likely that the drawing in Subfigure 25a is not the actual best drawing of this graph with these parameters. So, when I run the function again for $c_{se} = 5$ and $\alpha = 5$, I can get a different



(a) The best drawing for $c_{se} = 5$ and $\alpha = 5$. The inner vertices are at (25,50) and (26.47,35.29).

(b) The second best drawing for $c_{se} = 5$ and $\alpha = 5$.The inner vertices are at (20,40) and (34.62,46.15).

(c) The best drawing for $c_{le} = 5$ and $\beta = 5$. The inner vertices are at (34.61,46.15) and (28.57,42.86).

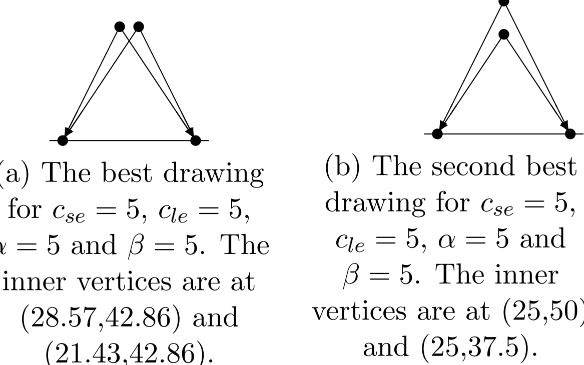
(d) The second best drawing for $c_{le} = 5$ and $\beta = 5$. The inner vertices are at (20,40) and (21.43,28.57).

Figure 25: Four drawings of the graph with encoding 2 2 1 01 01 with a combination of different parameters.

value for the inner vertices of the 'best' drawing.

We can see that when using a combination of a higher value of α and c_{se} or β and c_{le} we get some drawings with intersections and they are not terribly tall. It actually makes me curious as to what happens when I set both c_{se}, c_{le}, α and β to 5. The drawings for the combinations $\alpha = 5$ and $c_{se} = 5$, and $\beta = 5$ and $c_{le} = 5$ look a lot alike, save for the first combination seeming to produce slightly taller graphs than the second. So let's look at the drawings of the graph with $\alpha = \beta = 5$ and $c_{se} = c_{le} = 5$ (the other parameters are again 1). The best two drawings are given in Figure 26. (To be complete, the coordinates given for the inner vertices of the other 3 'good' drawings are: $(23.53, 35.29)$ and $(16.67, 50)$, $(26.47, 35.29)$ and $(40, 40)$, and $(12.5, 50)$ and $(26.47, 35.29)$. The last two graphs also have an intersection.)

From Figure 26 we can see that there are again intersections with these parameters and that the length of the graphs is more in between the lengths of the graphs of Subfigures 25a and 25b, and 25c and 25d. So let us continue with all these parameters at 5 and let us now look at what happens if we

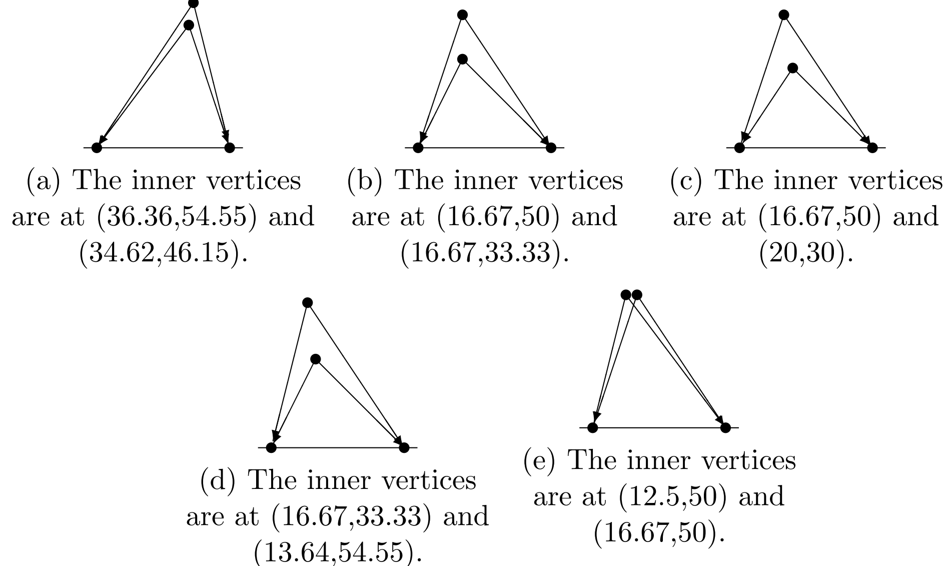


(a) The best drawing for $c_{se} = 5$, $c_{le} = 5$, $\alpha = 5$ and $\beta = 5$. The inner vertices are at (28.57,42.86) and (21.43,42.86).

(b) The second best drawing for $c_{se} = 5$, $c_{le} = 5$, $\alpha = 5$ and $\beta = 5$. The inner vertices are at (25,50) and (25,37.5).

Figure 26: Two drawings of the graph with encoding 2 2 1 01 01 with c_{se}, c_{le}, α and β all 5.

increase γ to 5 too. The results are shown in Figure 27.



(a) The inner vertices are at (36.36,54.55) and (34.62,46.15).

(b) The inner vertices are at (16.67,50) and (16.67,33.33).

(c) The inner vertices are at (16.67,50) and (20,30).

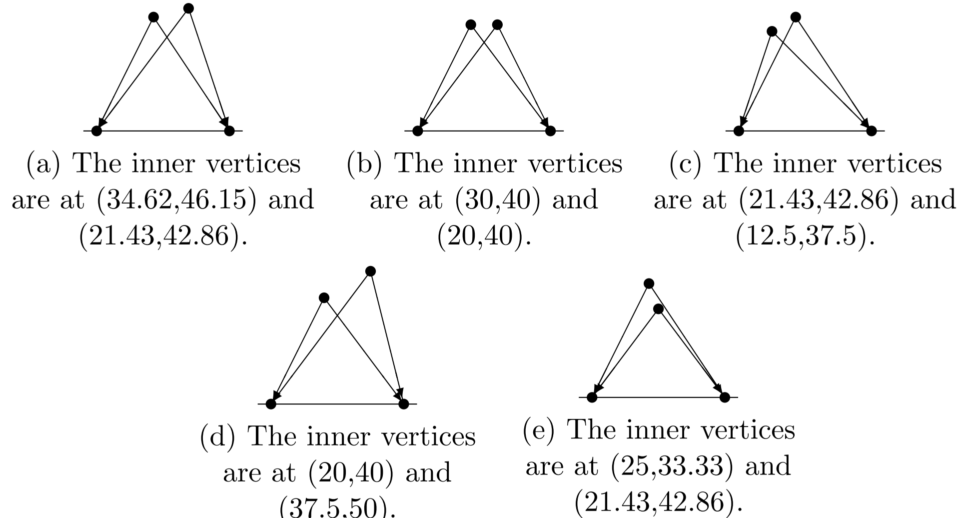
(d) The inner vertices are at (16.67,33.33) and (13.64,54.55).

(e) The inner vertices are at (12.5,50) and (16.67,50).

Figure 27: The five drawings of the graph with encoding 2 2 1 01 01 with $\gamma = 5$.

Figure 27 shows that when we raise γ to 5, we already get less drawings with an intersection. The four best drawings (Subfigures 27a, 27b, 27c and 27d) have no intersection.

Let's now compare this too when we instead increase ϵ to 5. The drawings for when $\alpha = \beta = 5$, $c_{se} = c_{le} = 5$, $\gamma = 1$ and $\epsilon = 5$ can be seen in Figure 28.



(a) The inner vertices are at (34.62,46.15) and (21.43,42.86).

(b) The inner vertices are at (30,40) and (20,40).

(c) The inner vertices are at (21.43,42.86) and (12.5,37.5).

(d) The inner vertices are at (20,40) and (37.5,50).

(e) The inner vertices are at (25,33.33) and (21.43,42.86).

Figure 28: The five drawings of the graph with encoding 2 2 1 01 01 with $\epsilon = 5$.

We see that for $\epsilon = 5$ we get more often intersections in our list of best five drawings than for $\gamma = 5$. This is not so strange if we remember how the function works. The intersections score consists of a product:

$$N^{\gamma} \cdot (\log N)^{\epsilon} .$$

In the case we're studying now there's maximally one intersection, so $N = 2$ and $\log 2 < 1$. Hence increasing ϵ will have more of an opposite effect. Permitting functions to have an intersection, because if they have one intersection and ϵ is high, then the intersections score will be close to zero.

Let us now look at what happens if we increase γ to 10, and return ϵ to 1. Since we already had little intersections for $\gamma = 5$, I'm expecting the resulting five drawings to have no intersections at all. The results are given in Figure 29. It can be seen that these drawings 29 do indeed not have any intersections.

I have two questions left about this graph.

Questions: What happens if I increase ϵ to 5 again?
And, can I make the graphs more symmetric?

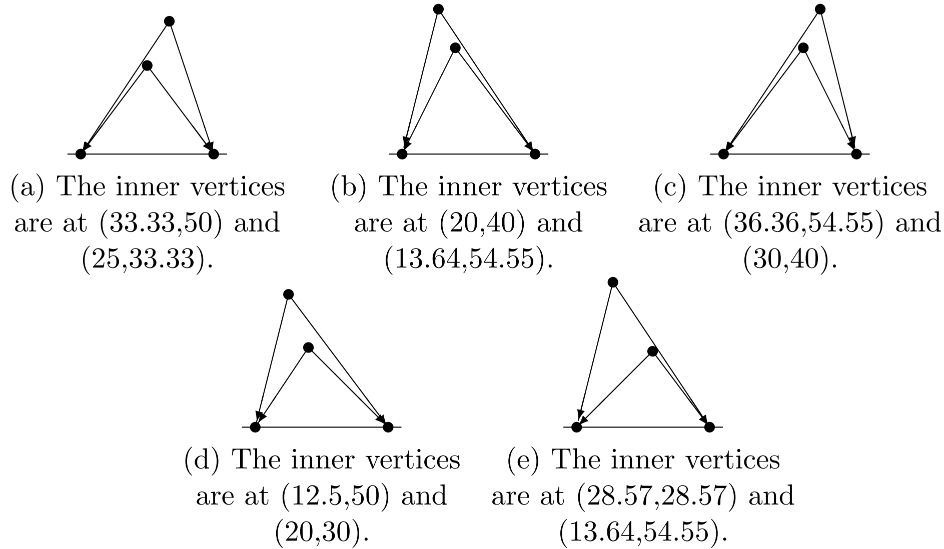


Figure 29: The five drawings of the graph with encoding 2 2 1 01 01 with $\gamma = 10$.

To answer the first question, see the drawings in Figure 30.

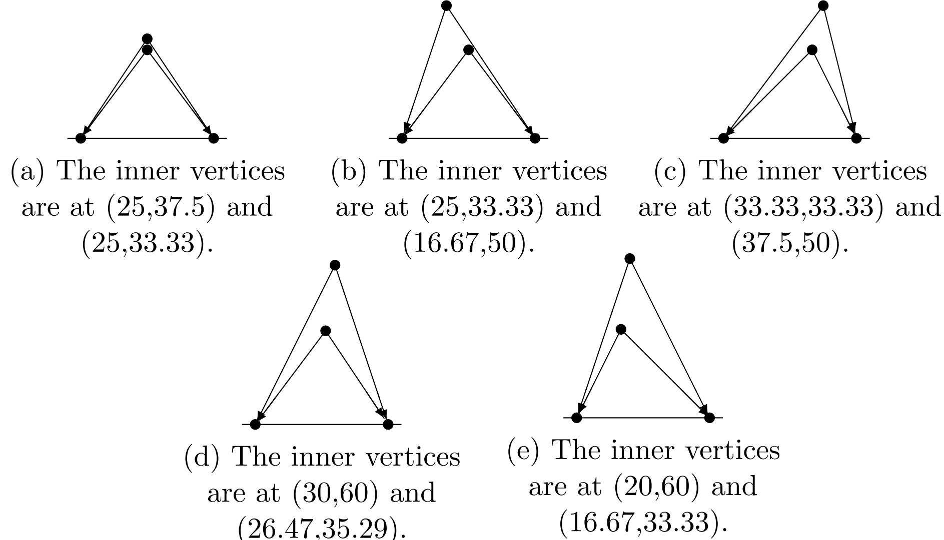
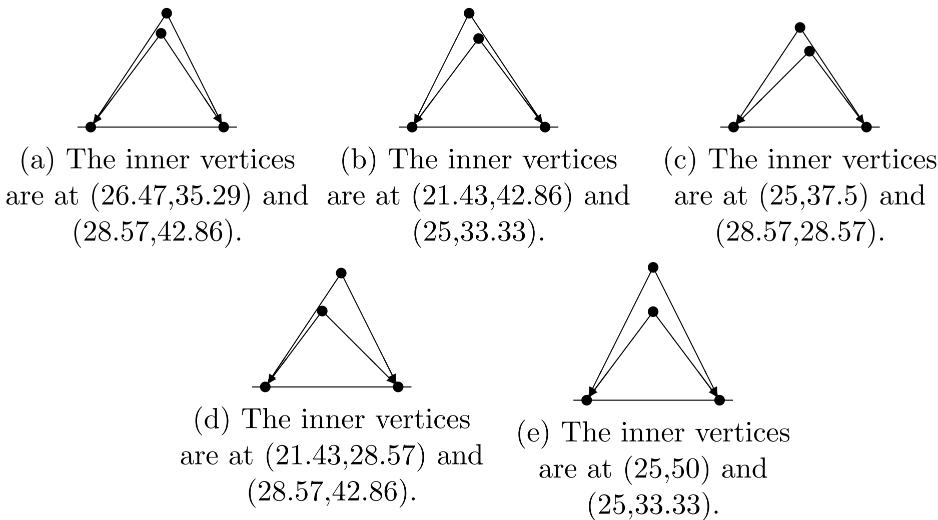


Figure 30: The five drawings of the graph with encoding 2 2 1 01 01 with $\gamma = 10$ and $\epsilon = 5$.

As we can see in Figure 30, there is not a lot of difference when we increase ϵ to 5 compared to when ϵ is 1 as in Figure 29. The best drawings still have no intersections.

Let us now look at if we can make the drawings more symmetric. As we've noticed in Section 6.1, the drawings seem to become more symmetric if we take measures to shorten the edges. I can do that in 4 ways, I can increase β or c_{le}, or I can decrease α or c_{se}. As concluded in the section on the wedge, higher values of β or α make the lengths of the vertices converge to l_0. In this case I suspect I will get drawings like in Subfigure 30a. I don't want my vertices to be so close together, so I will try to increase c_{le} and decrease c_{se}. The best five results for $\alpha = \beta = 5$, $c_{se} = 3$, $c_{le} = 7$, $\gamma = 10$ and $\epsilon = 5$ are given in Figure 31.



(a) The inner vertices are at (26.47,35.29) and (28.57,42.86).

(b) The inner vertices are at (21.43,42.86) and (25,33.33).

(c) The inner vertices are at (25,37.5) and (28.57,28.57).

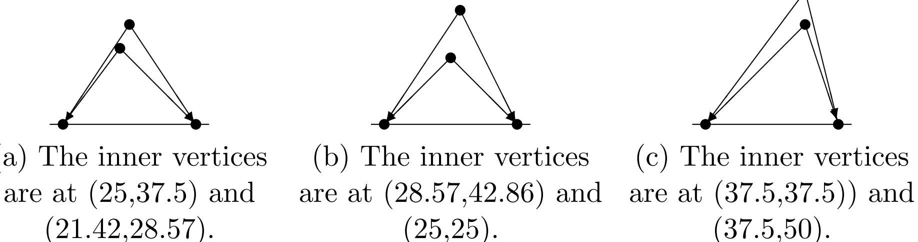
(d) The inner vertices are at (21.43,28.57) and (28.57,42.86).

(e) The inner vertices are at (25,50) and (25,33.33).

Figure 31: The five drawings of the graph with encoding 2 2 1 01 01 with $c_{le} = 7$ and $c_{se} = 3$.

The best three drawings do unfortunately still have their vertices very close to each other.

This is why the unrelated vertices score was introduced, so I will hence now set $\zeta = 1$, $c_{dv} = 1$ and $\ell_1 = 10$. The three best drawing are given in Figure 32. Even though the vertices in Subfigure 32c are 12.5pt apart, I still find them too close to each other. I will therefore increase ℓ_1 to 20. The three best resulting drawings are given in Figure 33. This still does not seem

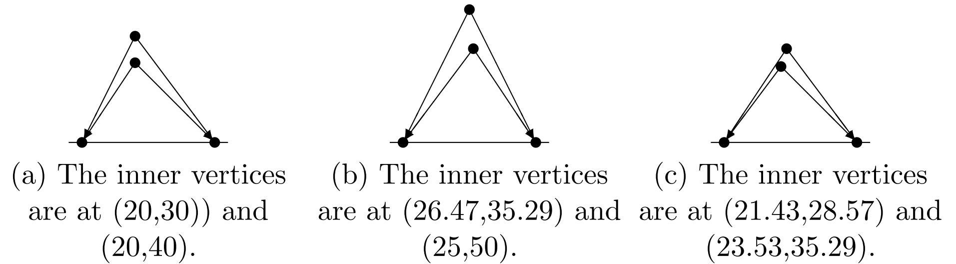


(a) The inner vertices are at (25,37.5) and (21.42,28.57).

(b) The inner vertices are at (28.57,42.86) and (25,25).

(c) The inner vertices are at (37.5,37.5)) and (37.5,50).

Figure 32: The three drawings of the graph with encoding 2 2 1 01 01 with $c_{dv} = 1$, $c_{le} = 7$ and $c_{se} = 3$.



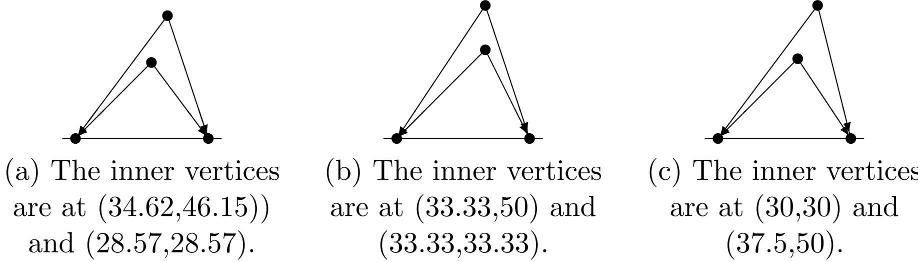
(a) The inner vertices are at (20,30)) and (20,40).

(b) The inner vertices are at (26.47,35.29) and (25,50).

(c) The inner vertices are at (21.43,28.57) and (23.53,35.29).

Figure 33: The three drawings of the graph with encoding 2 2 1 01 01 with $\ell_1 = 20$.

to work a great deal, so I will increase ζ to 5, so that for a drawing to be good, the distance between the unrelated vertices must be larger. The best 3 drawing for this are given in Figure 34. These drawings look quite a lot



(a) The inner vertices are at (34.62,46.15)) and (28.57,28.57).

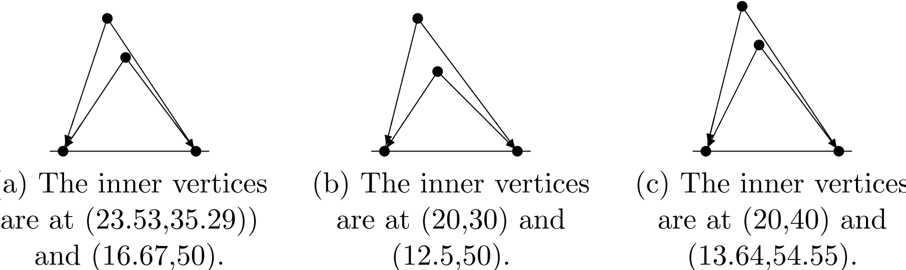
(b) The inner vertices are at (33.33,50) and (33.33,33.33).

(c) The inner vertices are at (30,30) and (37.5,50).

Figure 34: The three drawings of the graph with encoding 2 2 1 01 01 with $\zeta = 5$.

more acceptable, but what would have happened if instead of ζ, c_{dv} had been increased to 5? The resulting drawings with $\alpha = \beta = 5$, $c_{le} = 7$, $c_{se} = 3$,

$\zeta = 1$ and $c_{dv} = 5$ (and $\ell_2 = 20$) are shown in Figure 35. There doesn't seem



(a) The inner vertices are at (23.53,35.29)) and (16.67,50).

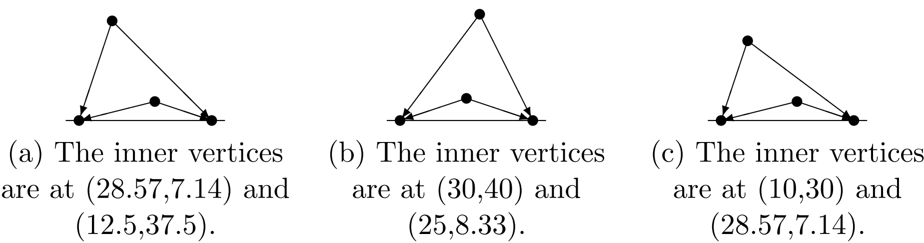
(b) The inner vertices are at (20,30) and (12.5,50).

(c) The inner vertices are at (20,40) and (13.64,54.55).

Figure 35: The three drawings of the graph with encoding 2 2 1 01 01 with $c_{dv} = 5$.

to be a big difference, but the vertices in Subfigure 35a, are closer together than in Subfigure 34a. So setting $c_{dv} = 5$ seems to be less effective than setting $\zeta = 5$. Secondly, the distance between the vertices seems to be more constant when $\zeta = 5$ compared to when $c_{dv} = 5$.

It might also be interesting to look at what happens if α and β are less than 1. As discussed in Subsection 5.1, decreasing the value of α and β below 1, will lead to less strict bounds on the length of the edges. I would suspect that α and β less than 1 and $\zeta = 5$ and $c_{dv} = 5$ might lead to the inner vertices being even further apart. The three best resulting drawings with $\alpha = \beta = 0.9$ are shown in Figure 36. These drawings start to look like what



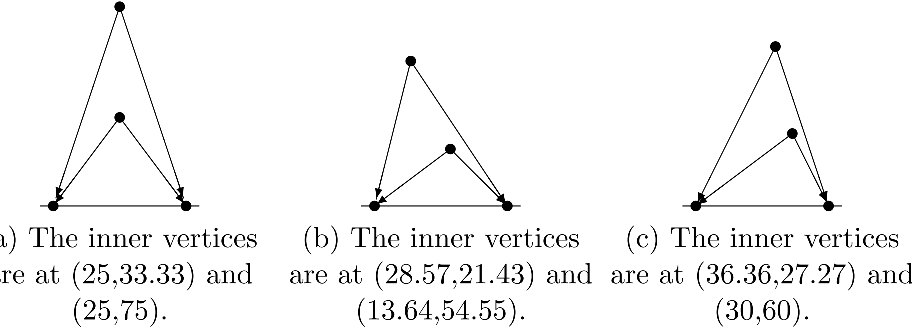
(a) The inner vertices are at (28.57,7.14) and (12.5,37.5).

(b) The inner vertices are at (30,40) and (25,8.33).

(c) The inner vertices are at (10,30) and (28.57,7.14).

Figure 36: The three drawings of the graph with encoding 2 2 1 01 01 with $c_{dv} = \zeta = 5$ and $\alpha = \beta = 0.9$.

I was hoping for, however I think the inner wedges are quite a bit too flat. This is because c_{se} is still 3 and c_{le} is still 7, so let us run the code again with $c_{se} = c_{le} = 5$. The three best drawings are shown in Figure 37. I find that these drawings look quite acceptable now. The only problem might be that they are too tall. But I will not do anything about that for now.



(a) The inner vertices are at (25,33.33) and (25,75).

(b) The inner vertices are at (28.57,21.43) and (13.64,54.55).

(c) The inner vertices are at (36.36,27.27) and (30,60).

Figure 37: The three drawings of the graph with encoding 2 2 1 01 01 with $c_{se} = c_{le} = 5$.

Since I now managed to make an acceptable drawing of the graph with encoding 2 2 1 01 01, with the parameters $\alpha = \beta = 0.9$, $c_{se} = c_{le} = 5$, $c_{dv} = 5$ and $\zeta = 5$ (and $\ell_0 = 50$ and $\ell_1 = 20$).

I will now move on to the more difficult graph with 3 internal vertices with encoding 2 3 1 01 12 12. In Figure 38 we see a first attempt at drawing the graph with the function `DrawGraph_compute_and_draw`.

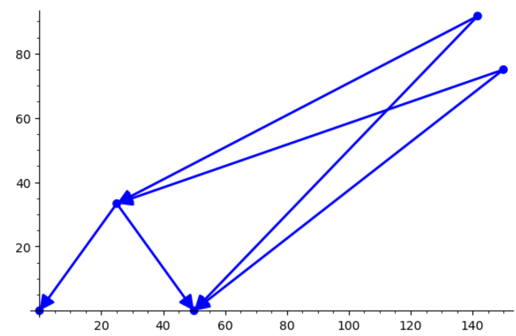


Figure 38: A drawing of the graph with encoding 2 2 1 01 12 12.

6.3 The Graph With Encoding 2 3 1 01 12 12

6.3.1 Some Probability Theory

To run the function for this graph, I estimated what I had to fill in for `num_of_iterations` so that I had probability P of picking at least 1 of the

best 5 elements. For a graph of order n the probability of picking one of the best five elements in one try is $5/552^n$ if using `DrawGraph_algorithm`. However, I'm using `DrawGraph_filter`, which means that not all 552^n options of possible inclines can also be an output. I ran all possible inclines for the wedge, and found that only 253 of the total 552 options gave an acceptable output. For the graph with encoding 2 2 1 01 01 only 56,672 of the $552^2 = 304,702$ options gave an acceptable output and for the graph with encoding 2 2 1 01 12 only 65,457 options gave an acceptable output.[3]

Idea. *The number of acceptable outputs for a graph of order n is less than $(552/2)^n$.*

Hence the change of picking one of the 5 best sets of coordinates of a graph of order n using `DrawGraph_filter`, would be around $5 \cdot (2/552)^n$. If we know this, we can use the cumulative distribution function of the **geometric distribution** as used in probability theory:

$$P(X \leq k) = 1 - (1-p)^k. \tag{13}$$

Where $P(X \leq k)$ is the probability that there occurs a success (picking one of the best 5 sets of coordinates in this case) in k tries. The change of success in each individual try is denoted by p, so that would be $5 \cdot (2/552)^n$ in this case. We can now calculate from Equation 13 that if we want a change P of success, k needs to be:

$$k = \frac{\log(1-P)}{\log(1-p)}.$$

For $n = 3$ and $P = 0.8$, this leads to a value of k of 6,767,549. For $P = 0.5$, $k = 2,914,624$. However, running the programme for all possible inclines for order 2, so for $552^2 = 304,704$ iterations took already more than 10 minutes. That would mean that to run the programme for order 3 and to have a certainty of more than 50 percent that we pick one of the best 5 sets of coordinates, I would probably have to wait more than 100 minutes, in which I am not even taking into account the fact that higher orders take longer to

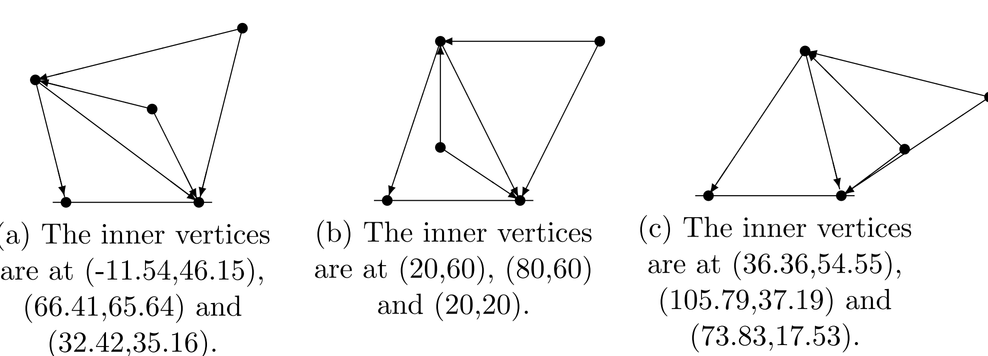
[3] I also tried to run the graph with encoding 2 3 1 01 12 12 for all possible inclines, but it wasn't done after 2 hours of running. (I measured that the time needed to run through all possible inclines for order 1 is just less than a second, but for order 2 it is already around 800s. If we assume that order 3 would be 800 times that, it would mean that running through all inclines for order 3 would cost my programme more than 177 hours!) I hence decided against running the programme through for all inclines for order 3.

run anyway. I do not have the time for that, so instead I calculated that if I want to take at least one of the best 0.1 percent[4] of the possible acceptable solutions. I would have to have $k = 3000$.[5] So, I set `num_of_iterations` at 3000 for this section.

The point of the above is to give a foundation for the number of iterations I choose to run the programme with.

6.3.2 Tuning the Target Function

I first ran the graph with the parameters as I ended up with them in Section 6.2. That is, $\ell_0 = 50$, $\ell_1 = 20$, $c_{se} = c_{le} = 5$, $c_{dv} = 5$, $\alpha = \beta = 0.9$, $\gamma = 10$, $\epsilon = 5$ and $\zeta = 5$. This gave the drawings in Figure 39 as the three best solutions. Let us first notice that the length of the edges seems to be



(a) The inner vertices are at (-11.54,46.15), (66.41,65.64) and (32.42,35.16).

(b) The inner vertices are at (20,60), (80,60) and (20,20).

(c) The inner vertices are at (36.36,54.55), (105.79,37.19) and (73.83,17.53).

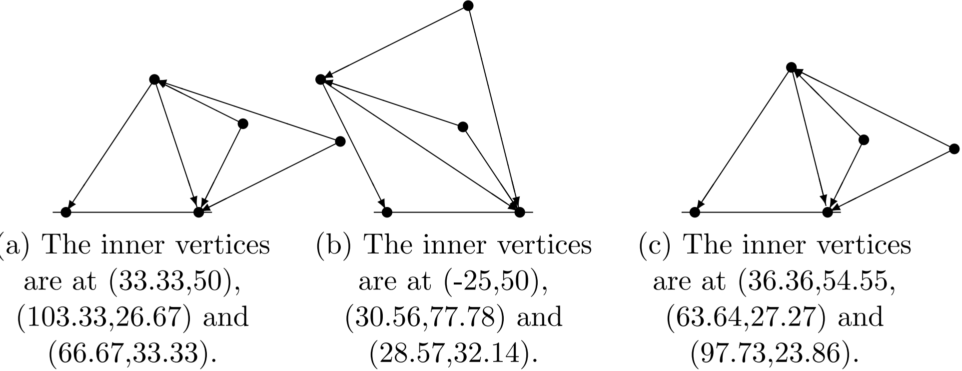
Figure 39: The three best drawings of the graph with encoding 2 2 1 01 12 12 with the parameters as mentioned in the beginning of the section.

fine, moreover, there are no intersections. The biggest problem with these drawings is that one of the wedges on the outside is inside the original wedge (one of the wedges encoded by 12 is inside the wedge encoded by 01). Since drawing the third vertex inside the first wedge inevitably leads to vertices that are close together, I will try to solve this by increasing c_{dv} to 10. As I mentioned in Section 5 the values of the c's determine the importance of

[4]However, the best 0.1 percent for order 3 still contains $(552/2)^3 \cdot 0.001 \approx 21,025$ different elements, so the solutions will change on each iteration.

[5]Also note that this calculation is the same for all other orders. If one takes $k = 3000$, one always has 95% possibility of picking one of the best 0.1 procent of the solutions. However for order 1, the best 0.1 percent is the best solution and for order 2 it is one of the 76 best solutions.

that term in the target function. The three best drawings with $c_{dv} = 10$ are given in Figure 40. These drawings all do not have a vertex inside the



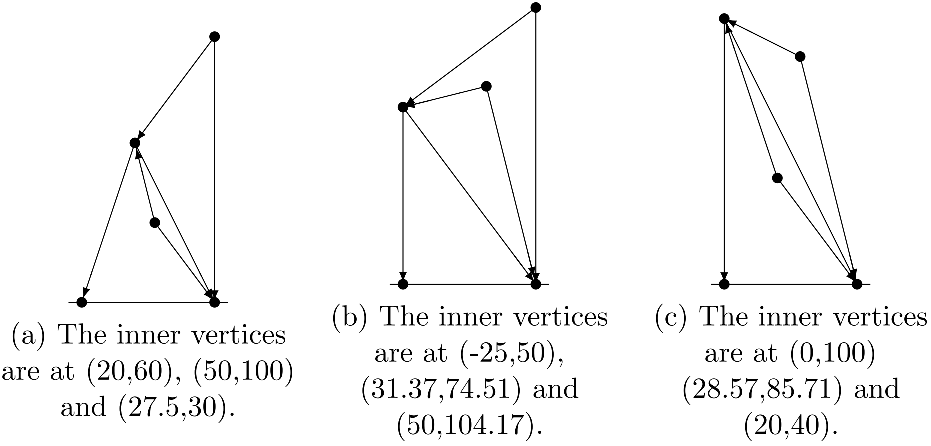
(a) The inner vertices are at (33.33,50), (103.33,26.67) and (66.67,33.33).

(b) The inner vertices are at (-25,50), (30.56,77.78) and (28.57,32.14).

(c) The inner vertices are at (36.36,54.55, (63.64,27.27) and (97.73,23.86).

Figure 40: The three best drawings of the graph with encoding 2 2 1 01 12 12 with $c_{dv} = 10$.

original wedge, so that is an improvement. However, the drawings are very much overshooting the line $x = 50$, which is not convenient if one tries to draw multiple different drawings on one line (as we can see in how Figure 40a and 40b almost intersect). I will try to solve this problem by adding an *overshoot score* to the target function (see Section 5.4). I will start by setting c_{os} and ι at 1. The three best resulting drawings are given in Figure 41.



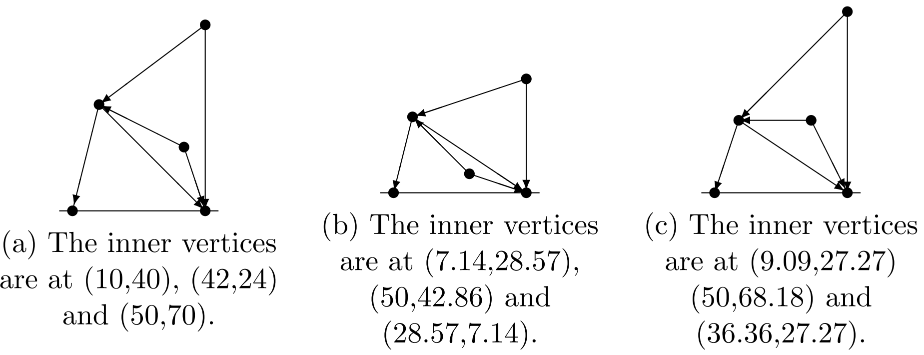
(a) The inner vertices are at (20,60), (50,100) and (27.5,30).

(b) The inner vertices are at (-25,50), (31.37,74.51) and (50,104.17).

(c) The inner vertices are at (0,100) (28.57,85.71) and (20,40).

Figure 41: The three best drawings of the graph with encoding 2 2 1 01 12 12 with $c_{0s} = 1$ and $\iota = 1$.

The overshoot score works, none of the best three drawings have vertices with an x-coordinate outside of the interval $[0, 50]$. However, the graphs are all very tall. I will try to get them a bit flatter by increasing c_{le} to 7, the resulting drawings are given in Figure 42. While I think that these drawings



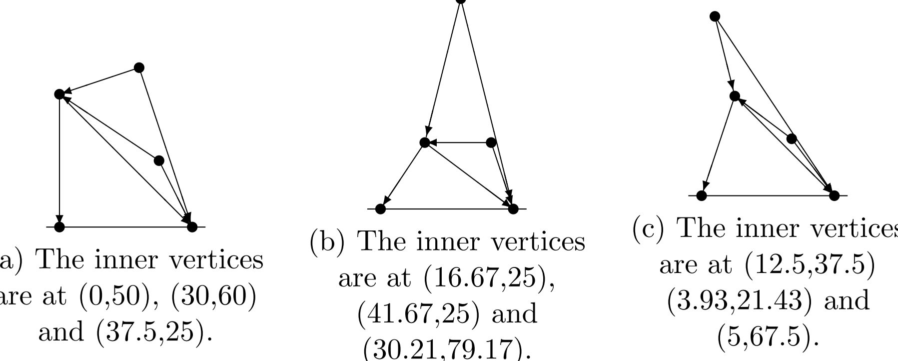
(a) The inner vertices are at (10,40), (42,24) and (50,70).

(b) The inner vertices are at (7.14,28.57), (50,42.86) and (28.57,7.14).

(c) The inner vertices are at (9.09,27.27) (50,68.18) and (36.36,27.27).

Figure 42: The three best drawings of the graph with encoding 2 2 1 01 12 12 with $c_{le} = 7$.

look better than the ones in Figure 41, the drawing in Subfigure 42b still has a vertex inside the original wedge. Because the unrelated vertices are actually on a proper distance from each other, it seems to be more efficient to 'punish' graphs for drawing vertices with a low height. For this purpose I added the *height y-coordinate score* to the target function (see Subsection 5.5). I set c_{hy} and θ at 10, and ℓ_2 at $0.4 \cdot \ell_0$. This did however result at inner vertices again being very high. With the highest inner vertex in the best three drawings having y-coordinate 100. Recalling from our tuning of α in Subsection 6.1 that increasing α did not necessary lead to longer edges, but in fact to edges with length converting to ℓ_0. Keeping this in mind, I increased θ even more to 20, the best three resulting drawings are given in Figure 43. This seems to work, these figures are however still a far cry from the example drawings in Section 1 (see Figure 1). While because of the overlapping edges test, I will not be able to exactly draw the figure there, one can look at the drawings to conclude a few things about how a nice drawing must look. Apart from everything I have discussed before, we can see that the drawings have a preference for vertical lines, and in fact not even only for edges, but also for unrelated vertices to be positioned right above each other. Secondly, the drawings often have three or more vertices positioned along one (not necessarily drawn) line. I hence added two extra scores to the target



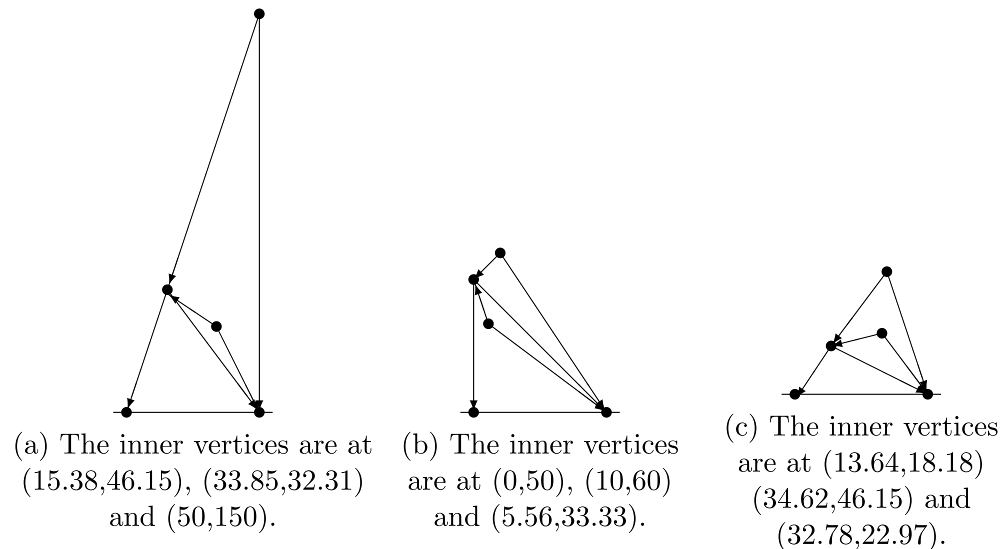
(a) The inner vertices are at (0,50), (30,60) and (37.5,25).

(b) The inner vertices are at (16.67,25), (41.67,25) and (30.21,79.17).

(c) The inner vertices are at (12.5,37.5) (3.93,21.43) and (5,67.5).

Figure 43: The three best drawings of the graph with encoding 2 2 1 01 12 12 with $\theta = 20$.

function: the *points on line score* and the *vertical lines score* (See Sections 5.6 respectively 5.7). Other than all the other scores in the target function, both these scores are negative, only decreasing the target function if there are vertical lines or more than two points positioned on one line. I started with looking at the effect of the points on line score. Because the score works basically opposite from the intersections score, I set the parameters at the same value as the intersections score, i.e. $\kappa = 10$ and $\lambda = 5$. The results



(a) The inner vertices are at (15.38,46.15), (33.85,32.31) and (50,150).

(b) The inner vertices are at (0,50), (10,60) and (5.56,33.33).

(c) The inner vertices are at (13.64,18.18) (34.62,46.15) and (32.78,22.97).

Figure 44: The three best drawings of the graph with encoding 2 2 1 01 12 12 with $\kappa = 10$ and $\lambda = 5$.

are given in Figure 44. Only the first two graphs have more than 2 vertices lying on one line (if you calculate the inclines in Figure 44c you will see that the points are not lying on one line, even if it might look like they do). The drawings also still do not have intersections, however if κ and λ are 10 times as large as the γ and ϵ of the intersections score, we will get points lying on one line, but the number of intersections will barely be considered anymore, see Figure 45.

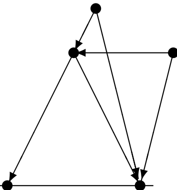
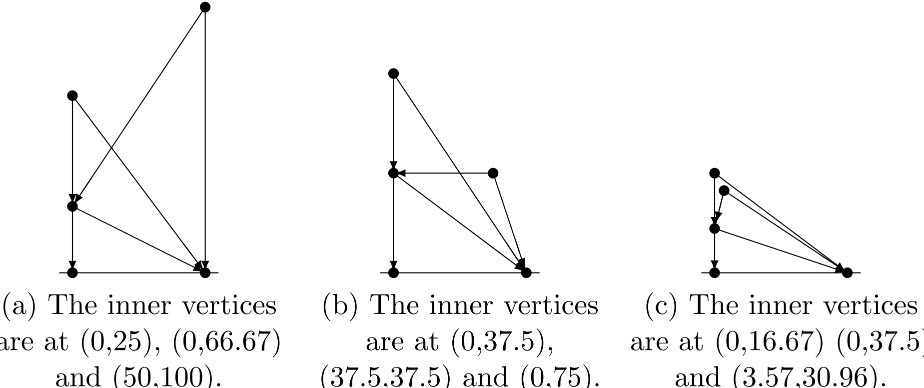


Figure 45: The best drawing of the graph with encoding 2 2 1 01 12 12 with $\kappa = 100$ and $\lambda = 50$. The inner vertices are at (12.5,37.5), (0,25) and (0,75).

Let's now look at the vertical lines score. If I set $\mu = 10$ and $\nu = 5$, similar to the points on line score, I get the drawings in Figure 46 as the best three drawings. All these drawings have at least 2 vertical lines, they do also



(a) The inner vertices are at (0,25), (0,66.67) and (50,100).

(b) The inner vertices are at (0,37.5), (37.5,37.5) and (0,75).

(c) The inner vertices are at (0,16.67) (0,37.5) and (3.57,30.96).

Figure 46: The three best drawings of the graph with encoding 2 2 1 01 12 12 with $\mu = 10$ and $\nu = 5$.

have intersections and in the last case, unrelated vertices on a close distance. We can find that if we decrease μ to 4 and ν to 2, we still have at least 2 vertical lines in the best three drawings, but no intersections anymore. The best drawing for these values is given in Figure 47.

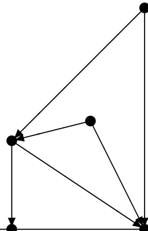
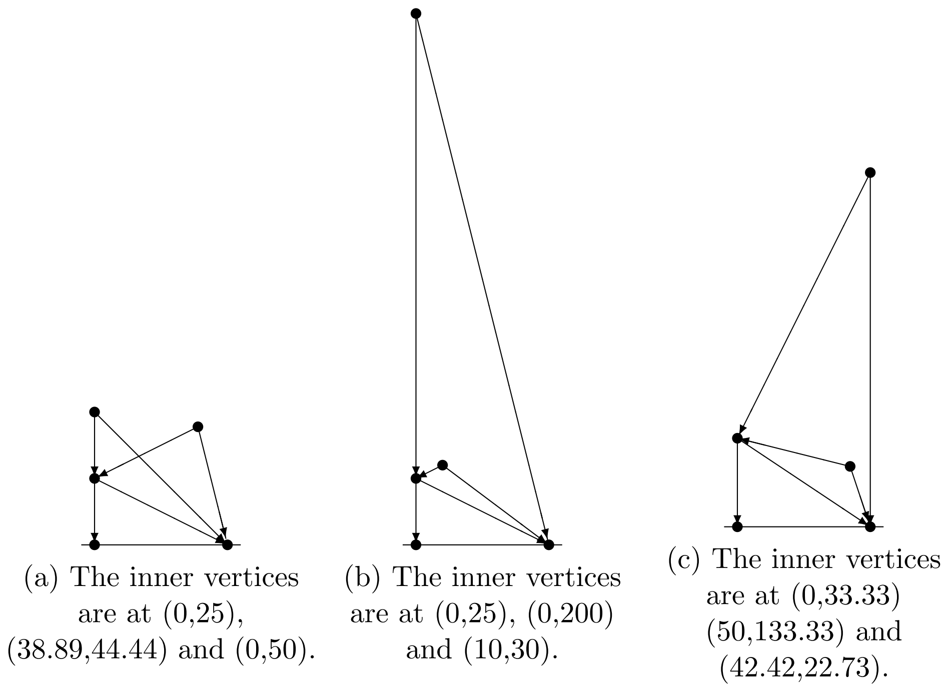


Figure 47: The best drawing of the graph with encoding 2 2 1 01 12 12 with $\mu = 4$ and $\nu = 2$. The inner vertices are at (0,33.33), (50,83.33) and (29.63,40.74).

This drawing looks quite acceptable, so let's now look at what happens if we use both the points on line score and the vertical lines score. Using the parameters as determined before, we get the drawings in Figure 48 as the best three results[6].



(a) The inner vertices are at (0,25), (38.89,44.44) and (0,50).

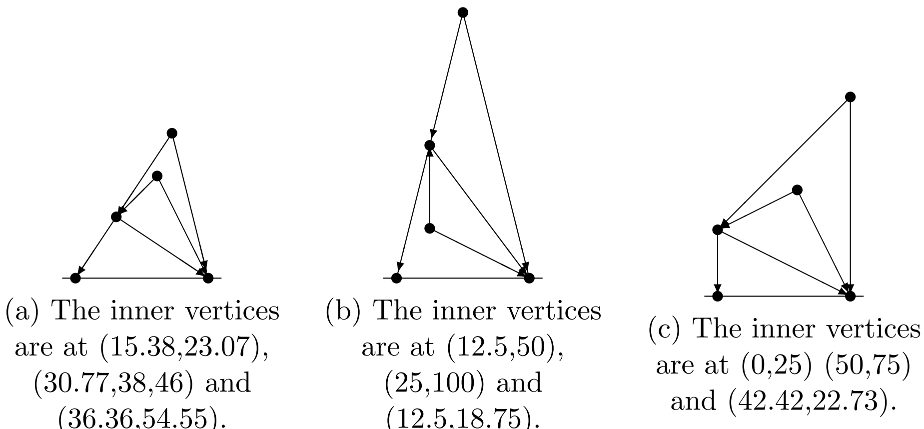
(b) The inner vertices are at (0,25), (0,200) and (10,30).

(c) The inner vertices are at (0,33.33) (50,133.33) and (42.42,22.73).

Figure 48: The three best drawings of the graph with encoding 2 2 1 01 12 12 with $\kappa = 10$, $\lambda = 5$, $\mu = 4$ and $\nu = 2$.

[6] In order to get more accurate results from now on I increase the number of iterations to 5000.

The first drawing here has as problem that there is an intersection again, the other two drawings have as problem that the edges are too long. I hence increase γ to 15 and c_{le} also to 15. The best three drawings are given in Figure 49. These drawings look quite acceptable now. In the following section I will



(a) The inner vertices are at (15.38,23.07), (30.77,38,46) and (36.36,54.55).

(b) The inner vertices are at (12.5,50), (25,100) and (12.5,18.75).

(c) The inner vertices are at (0,25) (50,75) and (42.42,22.73).

Figure 49: The three best drawings of the graph with encoding 2 2 1 01 12 12 with $\gamma = 15$ and $c_{le} = 15$.

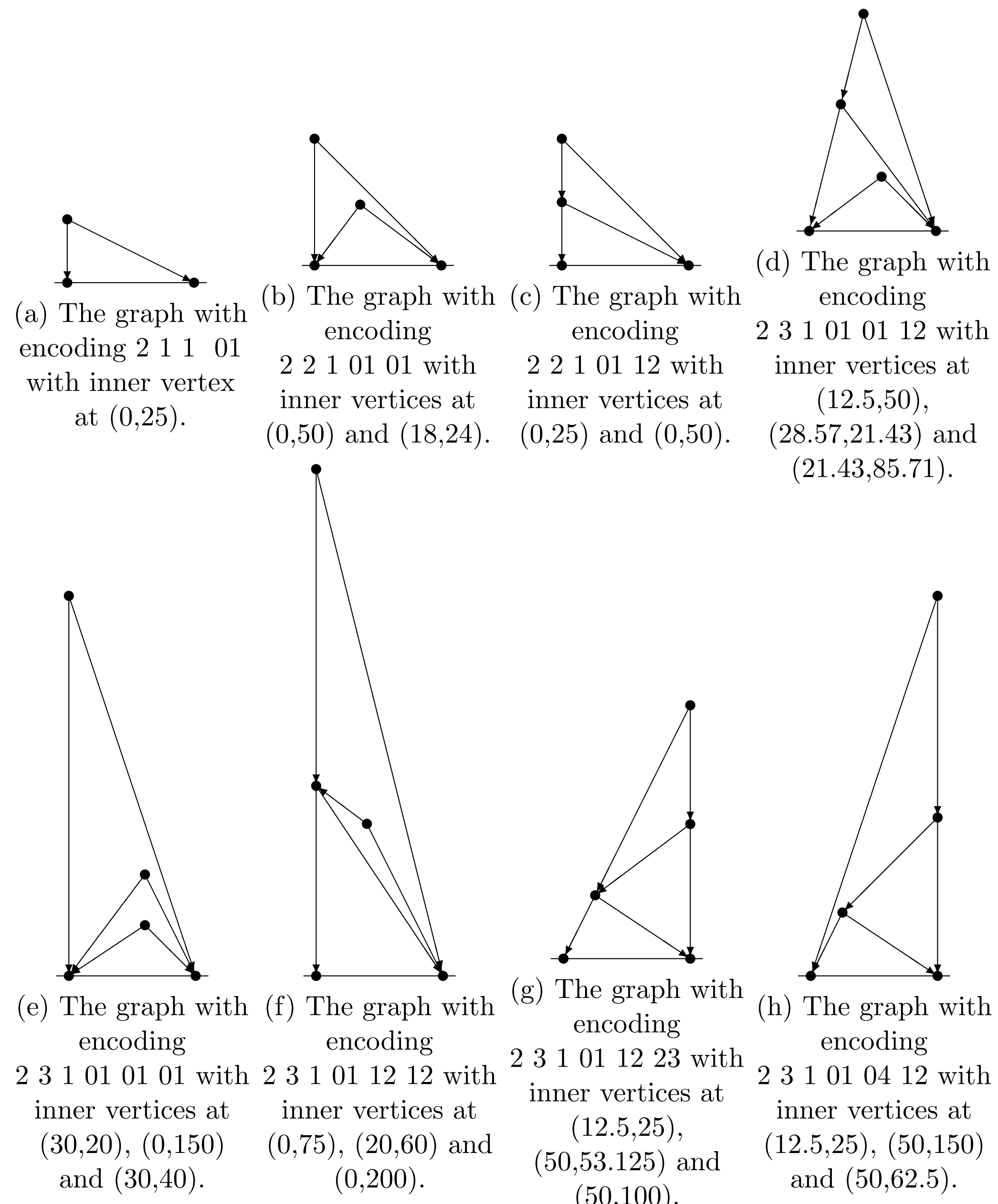
look at what all other graphs up till order three, without 2-cycles, look like with these parameters.

6.4 All Graphs up to Order 3

In this section, I will not look at any mirroring graphs. For example the graphs with encoding 2 2 1 01 12 and 2 2 1 01 02 are mirror images of each other, so I will only look at the graph with the first encoding. The graphs I will hence look at have encodings:

- *Order 1:* 2 1 1 01;
- *Order 2:* 2 2 1 01 12 and 2 2 1 01 01;
- *Order 3:* 2 3 1 01 12 12, 2 3 1 01 01 12, 2 3 1 01 01 01, 2 3 1 01 12 23 and 2 3 1 01 04 12.

Recall that I used the parameters $\ell_0 = 50$, $\ell_1 = 20$, $\ell_2 = 0.4 \cdot \ell_0$, $c_{se} = 5$, $c_{le} = 15$, $c_{dv} = 10$, $c_{hy} = 10$, $c_{os} = 1$, $\alpha = 0.9$, $\beta = 0.9$, $\gamma = 15$, $\epsilon = 5$, $\zeta = 5$, $\iota = 1$, $\theta = 20$, $\kappa = 10$, $\lambda = 5$, $\mu = 4$ and $\nu = 2$.



(a) The graph with encoding 2 1 1 01 with inner vertex at (0,25).

(b) The graph with encoding 2 2 1 01 01 with inner vertices at (0,50) and (18,24).

(c) The graph with encoding 2 2 1 01 12 with inner vertices at (0,25) and (0,50).

(d) The graph with encoding 2 3 1 01 01 12 with inner vertices at (12.5,50), (28.57,21.43) and (21.43,85.71).

(e) The graph with encoding 2 3 1 01 01 01 with inner vertices at (30,20), (0,150) and (30,40).

(f) The graph with encoding 2 3 1 01 12 12 with inner vertices at (0,75), (20,60) and (0,200).

(g) The graph with encoding 2 3 1 01 12 23 with inner vertices at (12.5,25), (50,53.125) and (50,100).

(h) The graph with encoding 2 3 1 01 04 12 with inner vertices at (12.5,25), (50,150) and (50,62.5).

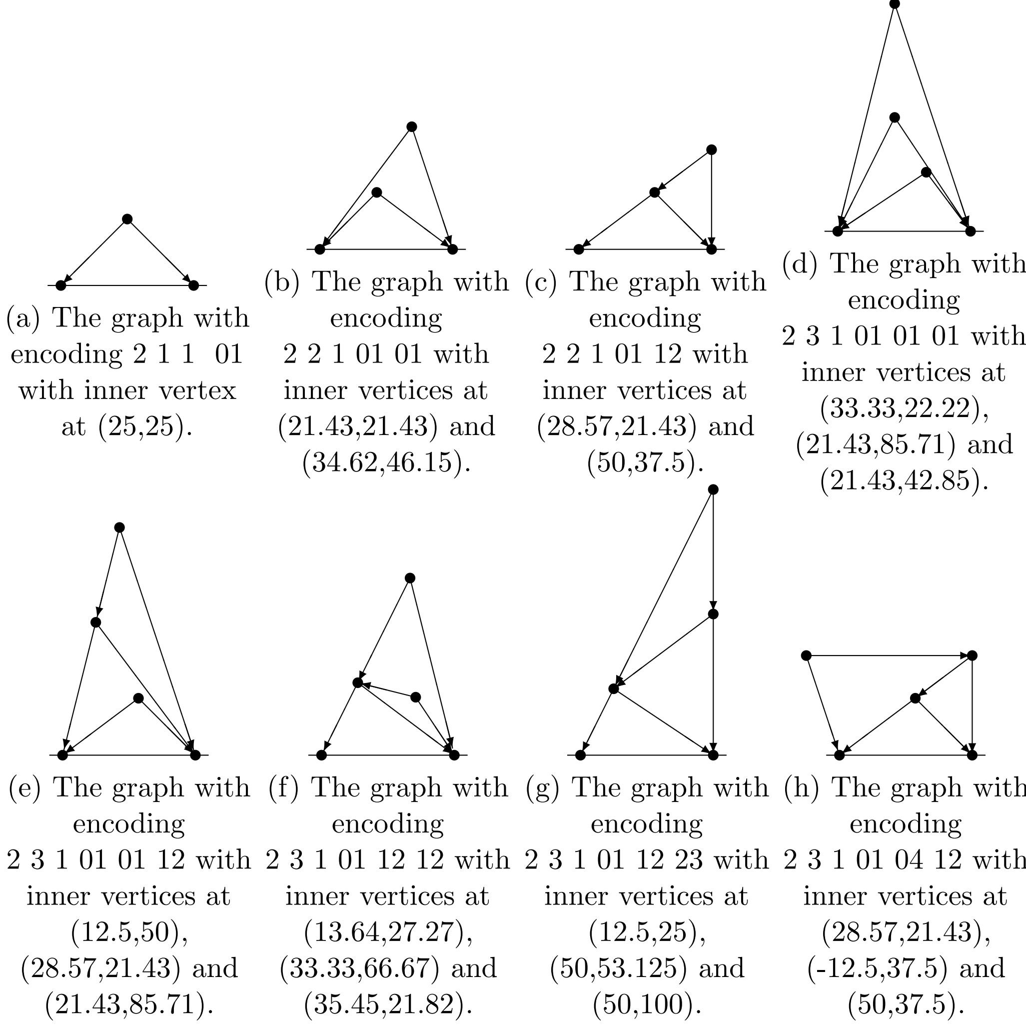
Figure 50: The best drawings of all graphs up to order 3, save for mirroring images, with the parameters as mentioned in the beginning of the section.

When I run these graphs through the target function[7], I get the drawings

[7]The graphs up to order 2 have been run through all possible inclines, the graphs of

in Figure 50 as the best. These drawings do not at all look ugly! There are no intersections and all vertices are drawn on a reasonable distance from each other.

The main problem in these drawings is that a couple of them are very tall, I hence increase c_{le} to 20 and β to 1.5. It also appears that the horizontal lines has gone a bit too far, so I will look at what happens if I decrease μ to 2 and ν to 1. The best resulting drawings are given in Figure 51.



(a) The graph with encoding 2 1 1 01 with inner vertex at (25,25).

(b) The graph with encoding 2 2 1 01 01 with inner vertices at (21.43,21.43) and (34.62,46.15).

(c) The graph with encoding 2 2 1 01 12 with inner vertices at (28.57,21.43) and (50,37.5).

(d) The graph with encoding 2 3 1 01 01 01 with inner vertices at (33.33,22.22), (21.43,85.71) and (21.43,42.85).

(e) The graph with encoding 2 3 1 01 01 12 with inner vertices at (12.5,50), (28.57,21.43) and (21.43,85.71).

(f) The graph with encoding 2 3 1 01 12 12 with inner vertices at (13.64,27.27), (33.33,66.67) and (35.45,21.82).

(g) The graph with encoding 2 3 1 01 12 23 with inner vertices at (12.5,25), (50,53.125) and (50,100).

(h) The graph with encoding 2 3 1 01 04 12 with inner vertices at (28.57,21.43), (-12.5,37.5) and (50,37.5).

Figure 51: The best drawings of all graphs without 2-cycles up to order 3 with $c_{le} = 20$, $\beta = 1.5$, $\mu = 2$ and $\nu = 1$.

order 3 are run for 10,000 iterations.

These drawings look already very good, they are more symmetric and less high. Notably, the drawing of the wedge in Figure 51a is completely symmetric and the drawing in Figure 51h now has 3 points on one line and a horizontal line.

One should of course keep in mind that we have only used 10,000 iterations for the graphs of order 3. It is hence possible that the best graphs of order 3 are not in our sample.

I tried to get slightly more horizontal lines, by increasing ν to 2, in the hope that the vertices in Figure 51b would be situated straight above each other. However, this only gave a different drawing for the graph with encoding 2 2 1 01 12 (see Figure 52) which I actually thought less beautiful than the drawing in Figure 51b.

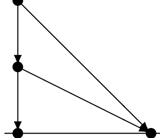


Figure 52: The best drawing of the graphs with encoding 2 2 1 01 12 with $\nu = 2$.

Because one could also argue that the first three and last drawing of Figure 51 might actually be too short, I also looked at what would happen if I decreased c_{le} again to 15, while at the same time also decreasing c_{hy} to 5 to make sure not all graphs would become very tall again. This did however give the exact same effect as increasing μ to 2, as did the combination of the two.

I set the parameters back to how they were in Figure 51, because this is likely the best possible combination of parameters.

7 Graphs with 2-cycles

I also extended the algorithm to deal with 2-cycles. At first, I thought that I could simply change the function `overlapping_edges_test` to accept overlapping edges if they were pointing to each other, however it wasn't that simple.

7.1 Changing the Programme

Like I said above, while changing the function `overlapping_edges_test` was absolutely necessary, it didn't solve all problems. I did get no solutions using `DrawGraph_filter` and when using `DrawGraph_algorithm` I could only get solutions where the vertices that were supposed to be in a two cycle sat at the same point. Why did this happen? To see this, let us look at an example using the graph with encoding 2 2 1 03 12. A drawing of this graph is given in Figure 53.

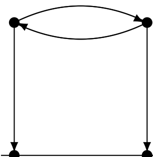


Figure 53: A drawing of the graph with encoding 2 2 1 03 12.

Example 16. The graph with encoding 2 2 1 03 12 corresponds to a matrix A of the form:

$$A = \begin{bmatrix} b_1^L & -a_1^L & 0 & 0 \\ b_1^R & -a_1^R & -b_1^R & a_1^R \\ 0 & 0 & b_2^L & -a_2^l \\ -b_2^R & a_2^R & b_2^R & -a_2^R \end{bmatrix}.$$

Recall that the corresponding vector $\mathbf{b}$ would be of the form: $[0, 0, \delta \cdot b_2^L, 0]^T$ and that $A\mathbf{x} = \mathbf{b}$. From the second and fourth row of the equations, the following equations can always be made:

$$x_1 - x_2 = 0$$
$$y_1 - y_2 = 0.$$

(Just add (a_1^R/a_2^L) times the fourth row to the second row and divide the resulting second row by $-b_1^R$. After that the second equation is simply gotten by subtracting a_2^R times the resulting second row from the fourth row.) This implies that x_1 would always have to be x_2 and y_1 needs to be y_2. Hence the two vertices of the 2-cycle will end up in the same point.

As the example shows, our current way of building the matrix A is not sufficient anymore. R. Buring suggested that I could maybe change the graph

into one without 2-cycles, by adding an extra vertex inside the 2-cycle and 2 more extra vertices on top and on the bottom of the 2-cycle. See Figure 54 for an illustration. However this graph had the same problem where all the

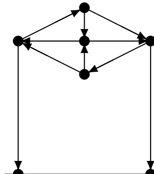


Figure 54: A drawing of the graph with encoding 2 2 1 03 12, but with extra vertices, essentially making it a graph with encoding 2 5 1 06 25 13 24 45.

vertices would end up in the same point.

I hence had to come up with a different solution, which would permit me to draw graphs with 2-cycles. I decided that in the case of a 2-cycle, I would set the height of the first vertex of the 2-cycle at δ (recall that δ is the distance between the sinks). This will make sure that the matrix is not always singular nor that the points of the inner vertices will always be drawn at the same point.

Example 17. Let us return to Example 16. With my changes to the way the matrix A is build, the matrix A will be of the form:

$$A = \begin{bmatrix} b_1^L & -a_1^L & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & b_2^L & -a_2^l \\ -b_2^R & a_2^R & b_2^R & -a_2^R \end{bmatrix}.$$

The corresponding vector $\mathbf{b}$ will be of the form $[0, \delta, \delta \cdot b_2^L, 0]^T$. The coordinate y_1 will always be δ, but all other coordinates can be chosen in a variety of ways.

An example of what a graph drawn with this underlying algorithm looks like is given in Figure 55.

7.2 Tuning the Target Function

I will start with looking at all graphs with 2-cycles up to order 3, i.e. the graphs with encodings: 2 2 1 03 12, 2 3 1 03 12 23, 2 3 1 03 12 12 and

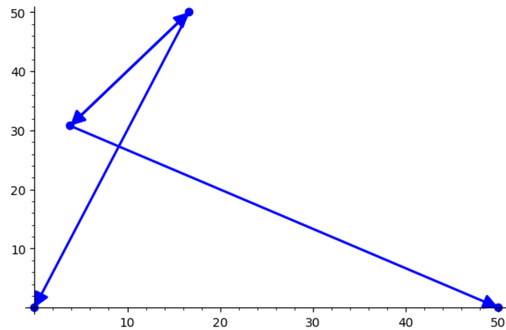
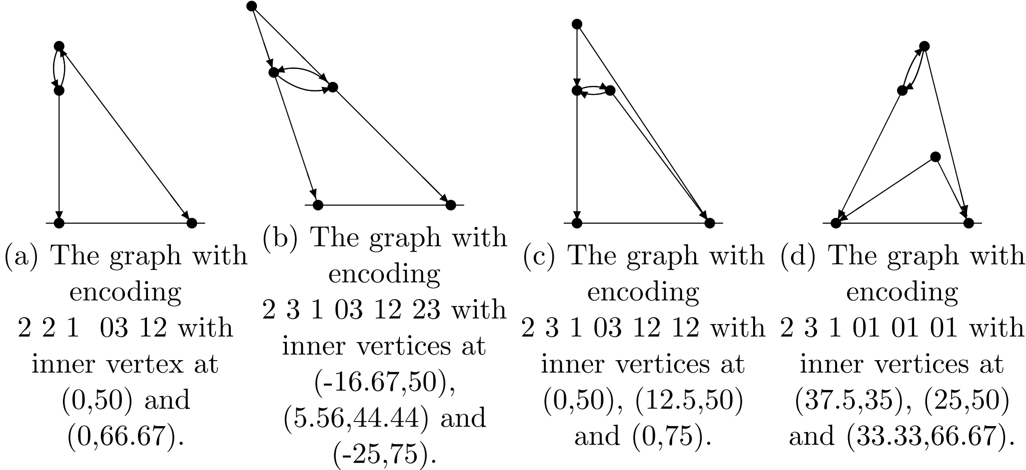


Figure 55: The graph with encoding 2 2 1 03 12 drawn with the function `DrawGraph_compute_and_draw`.

2 3 1 01 04 13. The order 2 graph will be run over all possible inclines, the order 3 graphs will be run over only 10,000 iterations. The drawing of all graphs with the parameters as established at the end of section 6.4 is given in Figure 56. The main problem with these drawings seems to be that the



(a) The graph with encoding 2 2 1 03 12 with inner vertex at (0,50) and (0,66.67).

(b) The graph with encoding 2 3 1 03 12 23 with inner vertices at (-16.67,50), (5.56,44.44) and (-25,75).

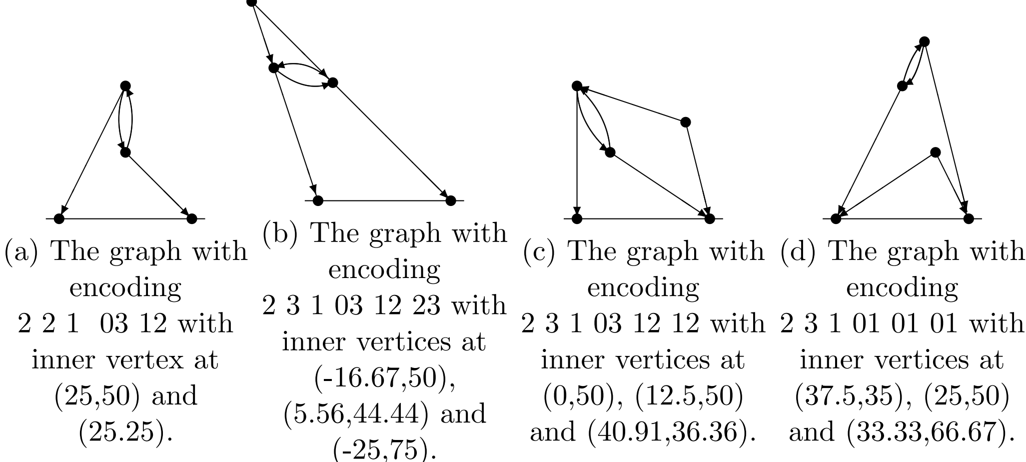
(c) The graph with encoding 2 3 1 03 12 12 with inner vertices at (0,50), (12.5,50) and (0,75).

(d) The graph with encoding 2 3 1 01 01 01 with inner vertices at (37.5,35), (25,50) and (33.33,66.67).

Figure 56: The best drawings of all graphs with 2-cycles up to order 3 with the parameters as in the end of Section 6.4.

graphs are too focused on getting points lying on one line. I hence decrease κ to 5. The resulting drawings are given in Figure 57.

It is also important to look at what such a change of parameter does with the drawings of the graphs without 2-cycles. Only the graphs with encoding



(a) The graph with encoding 2 2 1 03 12 with inner vertex at (25,50) and (25.25).

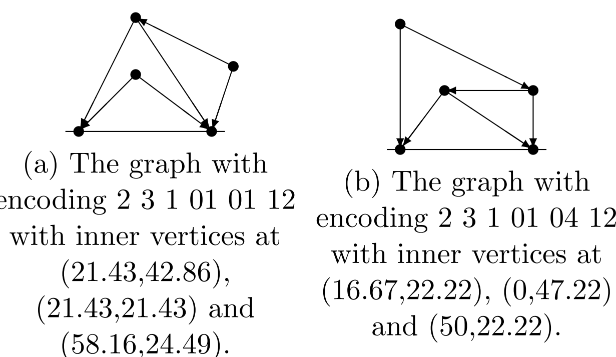
(b) The graph with encoding 2 3 1 03 12 23 with inner vertices at (-16.67,50), (5.56,44.44) and (-25,75).

(c) The graph with encoding 2 3 1 03 12 12 with inner vertices at (0,50), (12.5,50) and (40.91,36.36).

(d) The graph with encoding 2 3 1 01 01 01 with inner vertices at (37.5,35), (25,50) and (33.33,66.67).

Figure 57: The best drawings of all graphs with 2-cycles up to order 3 with $\kappa = 5$.

2 3 1 01 01 12 and 2 3 01 04 12 are affected, their drawings can be seen in Figure 58. Luckily the resulting drawings are not a lot uglier than the



(a) The graph with encoding 2 3 1 01 01 12 with inner vertices at (21.43,42.86), (21.43,21.43) and (58.16,24.49).

(b) The graph with encoding 2 3 1 01 04 12 with inner vertices at (16.67,22.22), (0,47.22) and (50,22.22).

Figure 58: The best drawings of the two graphs without 2-cycles up to order 3 that change with $\kappa = 5$.

drawings we had. However, the drawing in Figure 58a does overshoot the bound of $x = \delta$.

The combination of the drawings in Figure 57 and 58 suggests to me that I should decrease the importance of the vertical lines score even more by setting $\mu = 1$ and $\nu = 2$ and by increasing the importance of the overshoot score by setting $c_{os} = 2$. The only graph with 2-cycle whose best drawing

changed is the graph with encoding 2 2 1 03 12, the result is given in Figure 59. The graphs without 2-cycles can of course also change because of this.

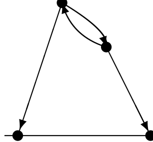


Figure 59: The best drawing of the graph 2 2 1 03 12 $\mu = 1$, $\nu = 2$ and $c_{os} = 2$. The inner vertices are at (16.67,50) and (33.33,33.33).

However, only the drawing of the graph with encoding 2 3 1 01 01 12 did change, the result is given in Figure 60.

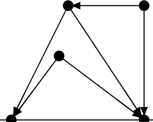


Figure 60: The best drawing of the graph with encoding 2 3 1 01 01 12 with $\kappa = 5$. The inner vertices are at (21.43,42.86), (18,24) and (50,42.86).

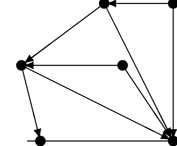
I will hence set the 'best' parameters to be: $\ell_0 = 50$, $\ell_1 = 20$, $\ell_2 = 20$, $c_{se} = 5$, $c_{le} = 20$, $c_{dv} = 10$, $c_{hy} = 10$, $c_{os} = 2$, $\alpha = 0.9$, $\beta = 1.5$, $\gamma = 15$, $\epsilon = 5$, $\zeta = 5$, $\theta = 20$, $\iota = 1$, $\kappa = 5$, $\lambda = 5$, $\mu = 1$ and $\nu = 2$. Keep in mind that this might not be the actual best parameters, but because I haven't done a full run through of all possible inclines for the order 3 graphs, it is difficult to draw definitive conclusions.

8 Graphs of order 4

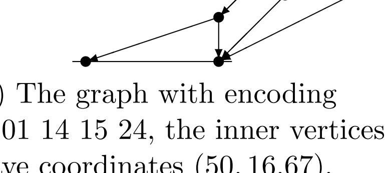
I didn't have time to run all the graphs of order 4, but I ran the graphs from line 41 to 155 excluding symmetries (which would together give drawings of more than 100 graphs of order 4) of the list of all encodings of Kontsevich graphs at `https://github.com/rburing/kontsevich_graph_series-cpp/blob/master/data/star4.txt` each with 10000 iterations. Recall that for graphs of order 4 there are 552^4 different combinations of inclines, this means that 10000 iterations is by no means a large (or even significant) sample, but

there were some decent pictures in there.[8]

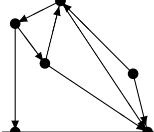
I also didn't have time to actually look at all drawings of these graphs while determining the best combination of parameters, so I picked a sample of 12 to see the influence of the target function on. Graphs that are on lines close together often look like, so I picked graphs more or less evenly distributed over my sample. The graphs I chose have encodings: 2 4 1 01 01 01 01, 2 4 1 01 12 12 13, 2 4 1 01 01 02 13, 2 4 1 01 02 12 12, 2 4 1 01 02 12 13, 2 4 1 01 02 03 12, 2 4 1 01 12 13 34, 2 4 1 01 14 15 24, 2 4 1 03 14 12 14, 2 4 1 03 12 12 12, 2 4 1 03 24 12 24 and 2 4 1 03 14 35 24. I will look at the drawings of these graphs using the function `DrawGraph_draw`, but I will only draw the graphs with encodings 2 4 1 01 12 12 13, 2 4 1 01 14 15 24 and 2 4 1 03 14 12 14 in LaTeX.



(a) The graph with encoding 2 4 1 01 12 12 13, the inner vertices have coordinates $(-7.14, 28.57)$, $(24.03, 51.95)$, $(30.95, 28.57)$ and $(50, 51.95)$.



(b) The graph with encoding 2 4 1 01 14 15 24, the inner vertices have coordinates $(50, 16.67)$, $(75, 25)$, $(150, 50)$ and $(183.33, 150)$.

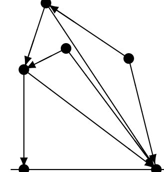


(c) The graph with encoding 2 4 1 03 14 12 14, the inner vertices have coordinates $(0, 40.81)$, $(11.22, 25.85)$, $(17.09, 49.36)$ and $(44.52, 21.94)$.

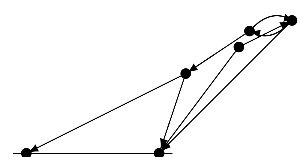
Figure 61: Order 4 graphs with the parameters as the end of Section 7.2.

[8] The list of all coordinates I considered can be found at https://github.com/SKerkhove/Bachelors-Project/blob/master/coordinates_order4_part1.txt and https://github.com/SKerkhove/Bachelors-Project/blob/master/coordinates_order4_part2.txt

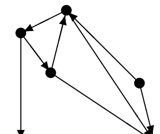
I will start with the parameters as I determined them in the end of Section 7.2. The resulting drawings are shown in Figure 61. Looking at the drawings, it seems that the biggest problem is too high graphs and overshooting the bounds $x = 0$ and $x = 50$. I try to solve this by increasing c_{le} from 20 to 25, β from 1.5 to 2, c_{os} from 2 to 5 and ι from 1 to 2. The results are shown in Figure 62. We still see overshoot in Figure 62b, more-



(a) The graph with encoding 2 4 1 01 12 12 13, the inner vertices have coordinates $(0, 37.5)$, $(8.33, 62.5)$, $(15.91, 45.45)$ and $(39.58, 41.67)$.



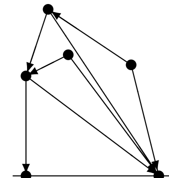
(b) The graph with encoding 2 4 1 01 14 15 24, the inner vertices have coordinates $(60, 30)$, $(80, 40)$, $(100, 50)$ and $(84, 46)$.



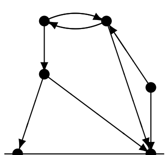
(c) The graph with encoding 2 4 1 03 14 12 14, the inner vertices have coordinates $(0, 40.81)$, $(11.22, 25.85)$, $(17.09, 49.36)$ and $(44.52, 21.94)$.

Figure 62: Order 4 graphs with the parameters as the end of Section 7.2, but $c_{le} = 25$, $c_{os} = 5$ $\beta = 2$ and $\iota = 2$.

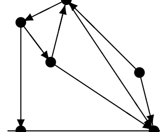
over unrelated vertices are situated close together. I will try to solve this by increasing c_{dv} from 10 to 15 and c_{os} from 5 to 10. The results are given in Figure 63. Please keep in mind that I'm only looking at a small sample of the total number of possible drawings, so there is a chance that I would actually choose the drawings I have now as the best ones in the sample. These drawings actually look very good. (Note that the drawings in Subfigures 63a and 63c are the same as the ones in Subfigures 62a and 62c.) While these drawings do not really look like my ideal drawing of these graphs, they are all



(a) The graph with encoding 2 4 1 01 12 12 13, the inner vertices have coordinates $(0, 37.5)$, $(8.33, 62.5)$, $(15.91, 45.45)$ and $(39.58, 41.67)$.



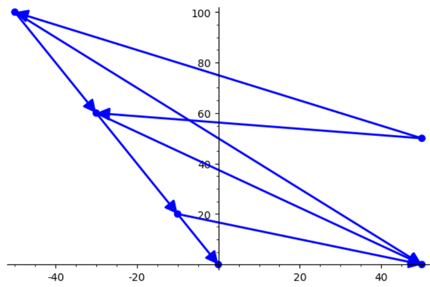
(b) The graph with encoding 2 4 1 01 14 15 24, the inner vertices have coordinates $(10, 30)$, $(50, 25)$, $(33.33, 50)$ and $(10, 50)$.



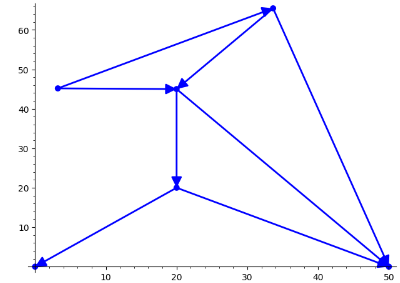
(c) The graph with encoding 2 4 1 03 14 12 14, the inner vertices have coordinates $(0, 40.81)$, $(11.22, 25.85)$, $(17.09, 49.36)$ and $(44.52, 21.94)$.

Figure 63: Order 4 graphs with the parameters as the end of Section 7.2, but $c_{os} = 10$ and $c_{dv} = 15$.

clear to look at. The other drawings that I only drew using `DrawGraph_draw` also look good now. For example, the graph with encoding 2 4 1 01 12 13 34 looked terrible in the first two runs (it mainly had an intersection), but it looks very reasonable under these parameters (see Figure 64).



(a) The 'best' drawing of the graph with the parameters as in Figure 61.



(b) The 'best' drawing of the graph with the parameters as in Figure 63.

Figure 64: The 'best' drawings of the graph with encoding 2 4 1 01 12 13 34 with different parameters.

Because of the size of my sample I find it unlikely that there is a combination of parameters that would give me a more satisfying result. I will hence now look at whether these parameters also give a nice result for the graphs up to order 3. Most graphs up to order 3 get more or less the same result as at the end of Section 7, but the graph with encoding 2 2 1 03 12 23 does look different. The drawing can be seen in Figure 65. However I think

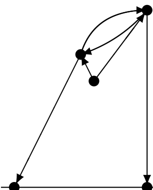
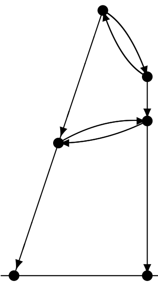


Figure 65: The graph with encoding 2 2 1 03 12 23, the inner vertices have coordinates $(25, 50)$, $(50, 66.67)$ and $(30, 40)$. The parameters are as in Figure 63.

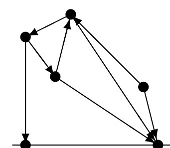
that this drawing is in some ways better than the one we got at the end of Section 7 (it's not overshooting the line $y = 0$). So while this is not my ideal drawing, I do think this might be one of the best drawings in the sample.

Therefore, I will conclude that the best drawings of graphs up to order 4 are given by the parameters: $\ell_0 = 50$, $\ell_1 = 20$. $\ell_2 = 20$, $c_{se} = 5$, $c_{le} = 25$, $c_{dv} = 15$, $c_{hy} = 10$, $c_{os} = 10$, $\alpha = 0.9$, $\beta = 2$, $\gamma = 15$, $\epsilon = 5$, $\zeta = 5$, $\theta = 20$, $\iota = 2$, $\kappa = 5$, $\lambda = 5$, $\mu = 1$ and $\nu = 2$.

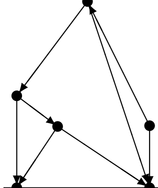
It is of course also interesting to see what other graphs or order 4 look like with these parameters, are they also reasonable? So let's look at 3 randomly chosen graphs, they are given in Figure 66. As we can see, these graphs also look very reasonable. In fact I even quite like the graphs in Subfigures 66a and 66c. So I think we can conclude that the parameters defined above are indeed good parameters for graphs up to order 4.



(a) The graph with encoding 2 4 1 03 12 25 34, the inner vertices have coordinates $(16.67, 50)$, $(50, 58.33)$, $(33.33, 100)$ and $(50, 75)$.



(b) The graph with encoding 2 4 1 03 14 12 14, the inner vertices have coordinates $(0, 40.81)$, $(11.22, 25.85)$, $(17.09, 49.36)$ and $(44.52, 21.94)$.



(c) The graph with encoding 2 4 1 01 02 13 14, the inner vertices have coordinates $(15.38, 23.08)$, $(0, 34.62)$, $(26.63, 70.12)$ and $(50, 23.37)$.

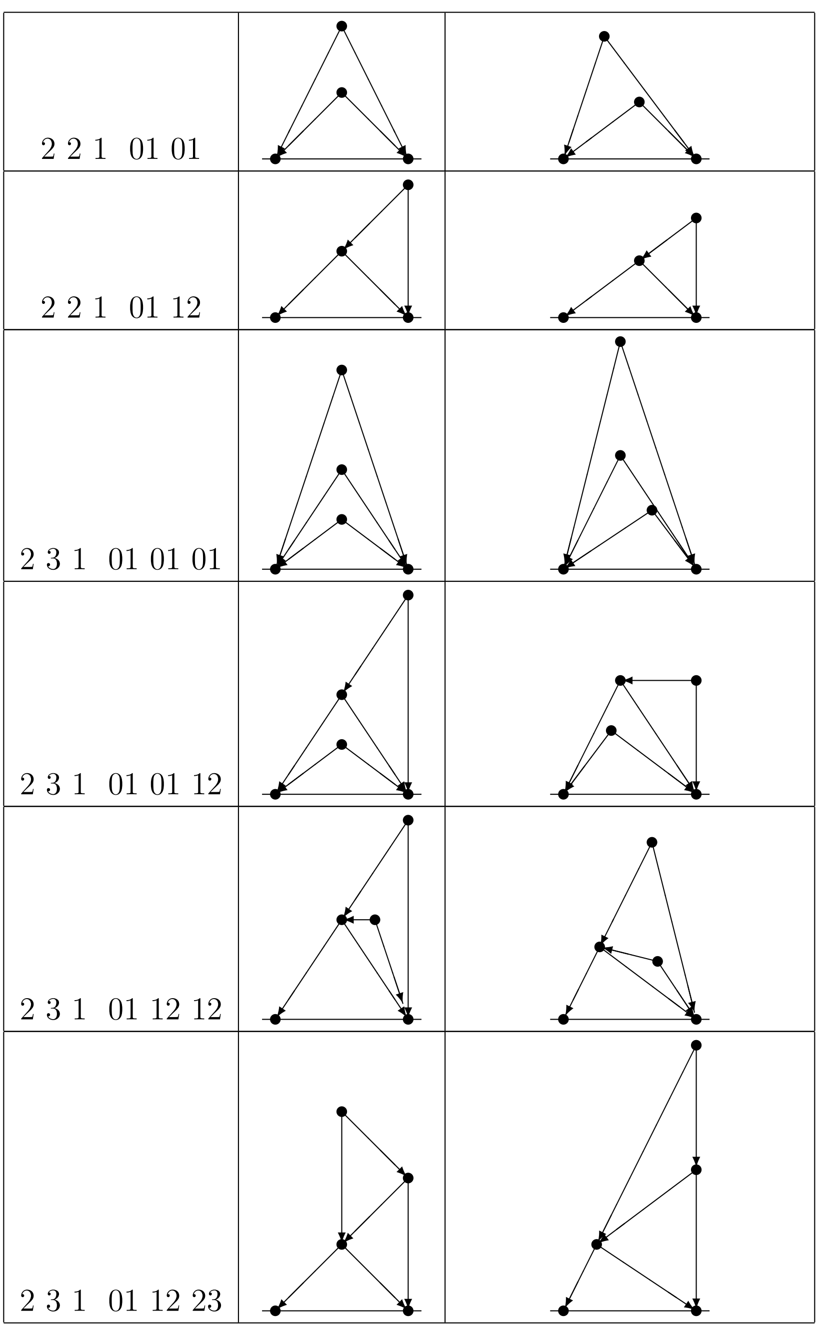
Figure 66: Three randomly chosen order 4 graphs with the parameters as the end of Section 8.

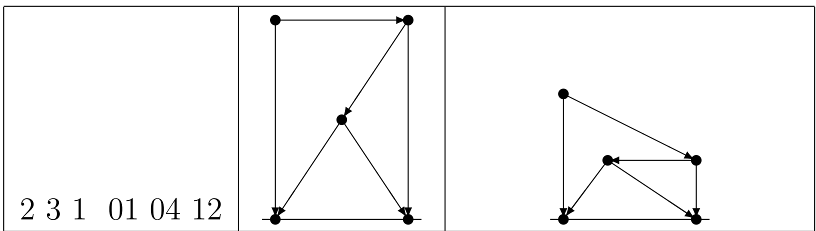
9 Conclusions

I have given all Kontsevich graphs without 2-cycles up to order 3 that I have drawn with the algorithm in Table 1.

Table 1: All graphs without 2-cycles up to order 3.

Encoding	Hand-Drawn	Drawn by the Algorithm
2 2 1 01		


2 2 1 01 01
2 2 1 01 12
2 3 1 01 01 01
2 3 1 01 01 12
2 3 1 01 12 12
2 3 1 01 12 23

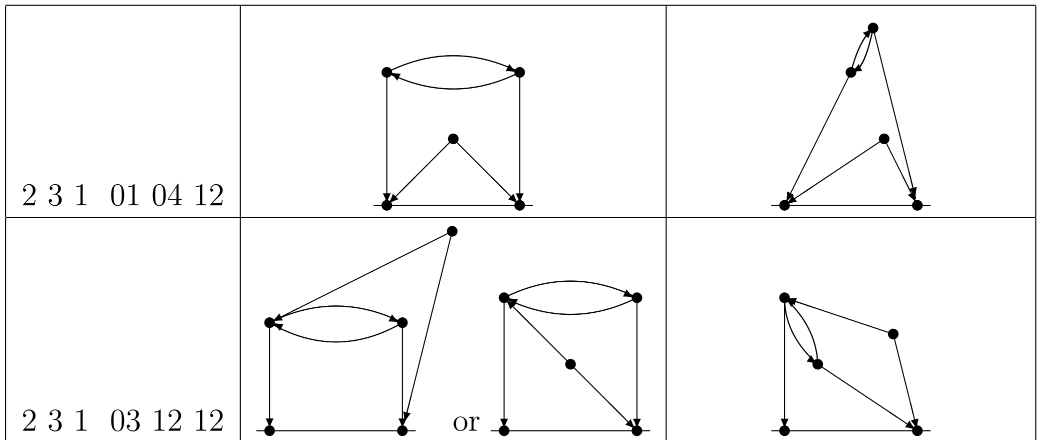
2 3 1 01 04 12		

From the comparison between the columns can we conclude that the algorithm works pretty well in drawing the Kontsevich graphs in LaTeX. Most graphs drawn by the algorithm look quite like their hand-drawn counterparts. I find that especially the graphs with encoding 2 2 1 01 and 2 2 1 01 12 are drawn very well by the algorithm. Furthermore, the drawings of the graphs with encoding 2 2 1 01 01, 2 3 1 01 01 01, 2 3 1 01 12 12 and 2 3 1 01 12 23 also look quite good. While the remaining two graphs, with encoding 2 3 1 01 01 12 and 2 3 1 01 04 12, don't look as much like their hand-drawn counterpart, I still don't think they are ugly. One should keep in mind that I only took 10,000 iterations for the graphs of order 3 instead of running through all possible inclines. It is hence very likely that there might actually be a better drawing for the graphs of order 3.

In Table 2 I have drawn the Kontsevich graphs with 2-cycles up to order 3.

Table 2: All graphs with 2-cycles up to order 3.

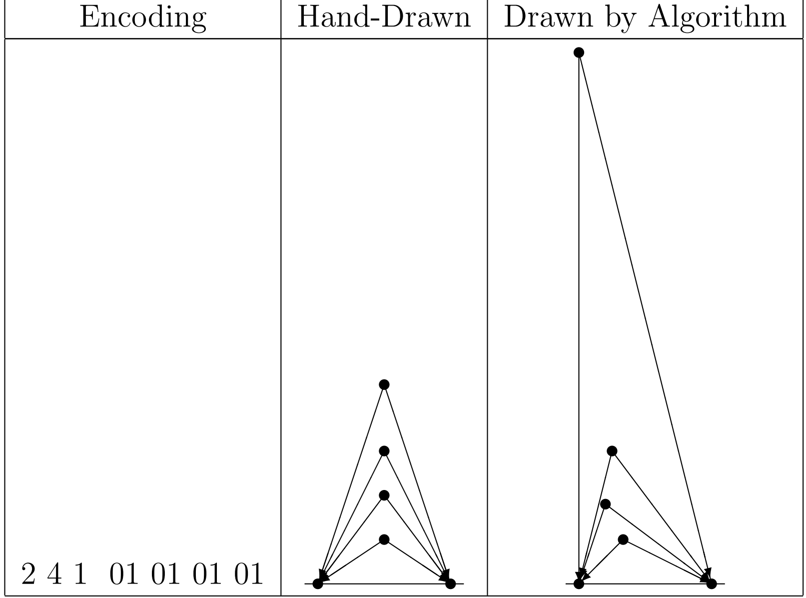
Encoding	Hand-Drawn	Drawn by the Algorithm
2 2 1 03 12		
2 3 1 03 12 23		

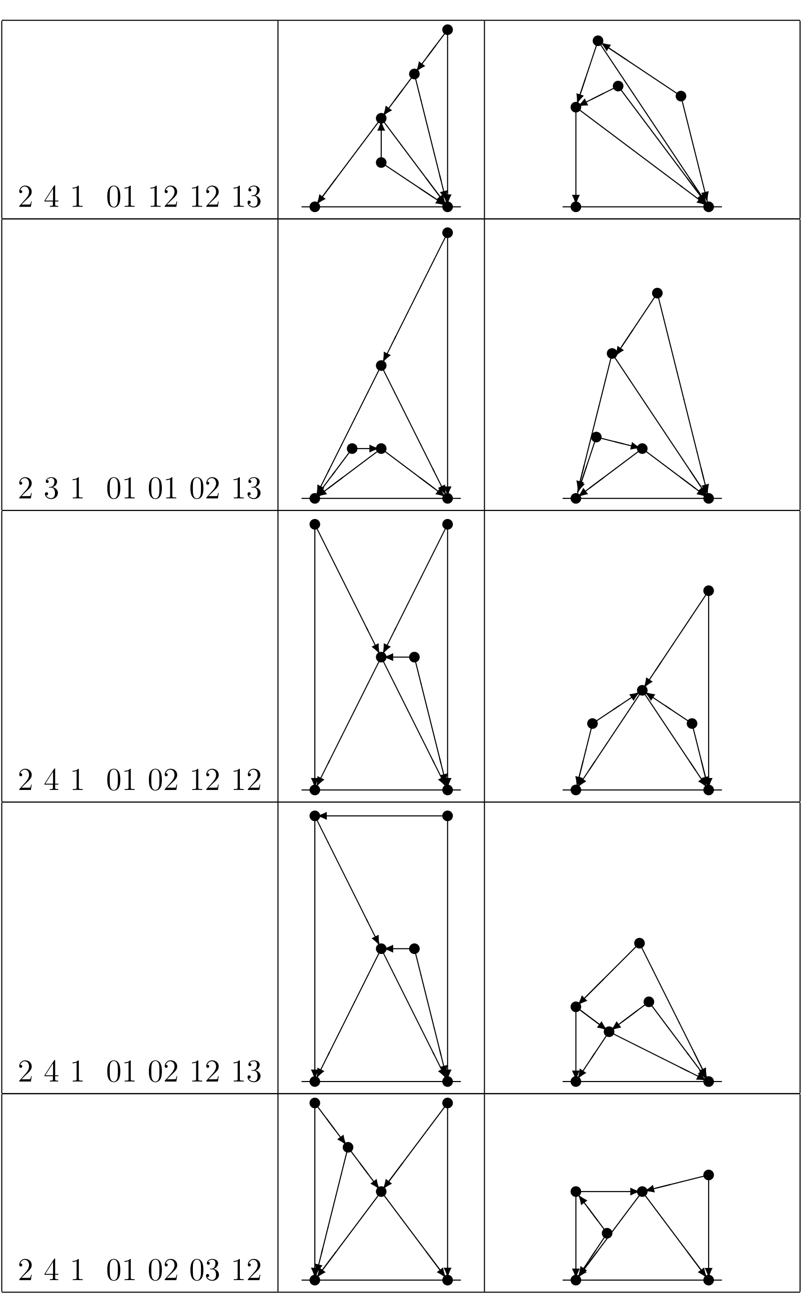


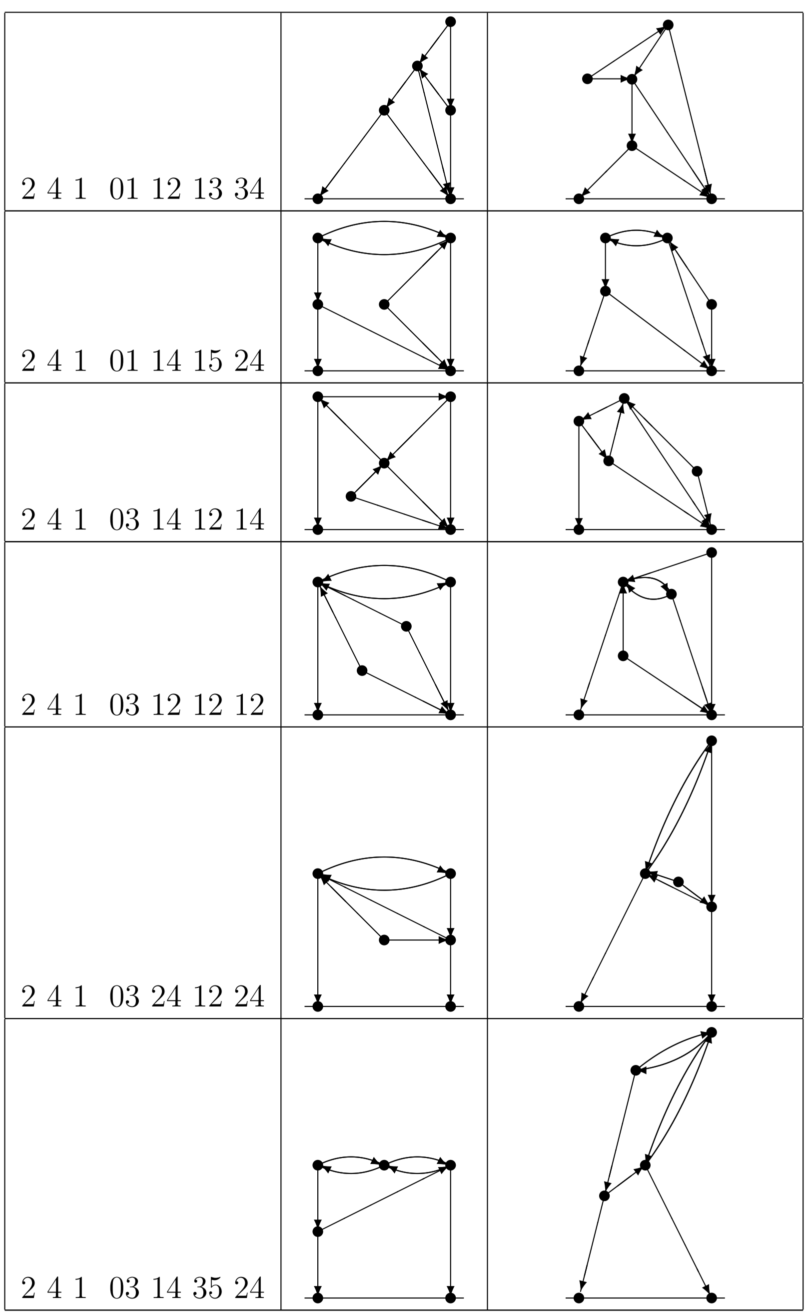
While I think the graphs with 2-cycles do not look as good as the graphs without 2-cycles I think they are capable of showing what the graph should look like. It should also be remembered that most graphs with 2-cycles that I considered are of order 3. As mentioned above, it is very likely that the graphs of order 3 actually have a better drawing that simply didn't end up in the sample I took.

The 12 graphs of order 4 that I considered are given in Table 3.

Table 3: Twelve graphs of order 4.




2 4 1 01 12 12 13
2 3 1 01 01 02 13
2 4 1 01 02 12 12
2 4 1 01 02 12 13
2 4 1 01 02 03 12



While I don't think these graphs are all necessarily beautiful, I do think they all look quite clear. Moreover, I find it remarkable that these graphs even

look so reasonable on a sample of only 10,000 out of the 552^4 possibilities. I need to make a remark on the last drawing (of the graph with encoding 2 4 1 03 14 35 24, this drawing can never look like my ideal version, because I programmed the algorithm to put the first vertex of every second 2-cycle a distance δ above the first 2-cycle. This will be discussed more in Section 10.

Let us now look at the parameters I ended up with and see if we can conclusions from that. It is of course a bit difficult to draw strong conclu-

Table 4: The final parameters for each score of the target function.

SLS	$\ell_0 = 50$	$c_{se} = 5$	$c_{le} = 25$	$\alpha = 0.9$	$\beta = 2$
UVS	$\ell_1 = 20$	$c_{dv} = 15$		$\zeta = 5$	
HY	$\ell_2 = 20$	$c_{hy} = 10$		$\theta = 20$	
OS		$c_{os} = 10$		$\iota = 2$	
IS				$\gamma = 15$	$\epsilon = 5$
PLS				$\kappa = 5$	$\lambda = 5$
VLS				$\mu = 1$	$\nu = 2$

sions, because many scores influence each other. For example, the height y-coordinate score works against the long edges score, while it works with the short edges score. This would be why c_{le} and β are much larger than c_{se} and α.

However, I think we can conclude that, in order to get aesthetically pleasing pictures, it is in general important that the edges have a reasonable length (of around 50 points). It seems to be slightly less important, but nevertheless, it looks like it is important that unrelated vertices are not positioned too close together. Finally, drawings shouldn't have intersections in order to look pretty.

10 Discussion

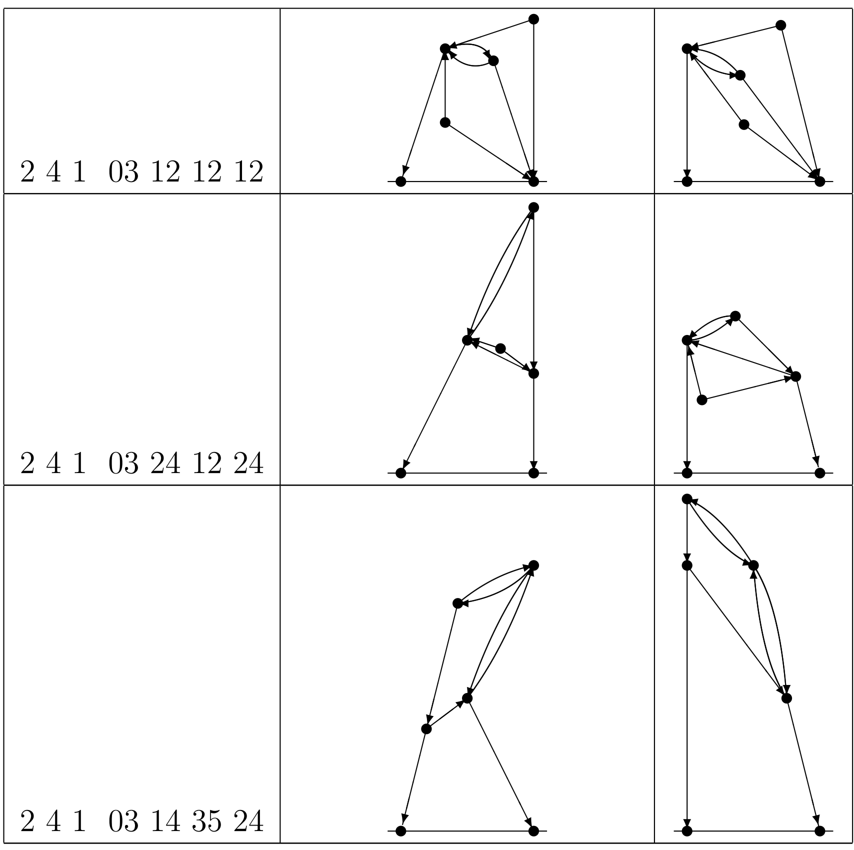
First we need to go back to Section 3. There we see in step 3.4 that we do not actually need to draw the picture that gets the lowest score from the target function, but rather the one from the 5 pictures with the lowest score that I think is the prettiest. So let's look at what drawings I will get if I follow this procedure. The results are shown in Table 5. While I believe that these other drawings look better than the ones with the lowest score, it

should be noted that all these choices are purely personal preferences. For example someone could actually prefer the first drawing of the graph with encoding 2 4 1 01 02 12 13 while I prefer the other one.

Table 5: The graphs up to order 4 with a best picture that doesn't have the lowest score from the target function.

Encoding	Picture with lowest score	Best picture
2 3 1 01 01 01		
2 3 1 01 01 12		
2 3 1 01 04 12		
2 2 1 03 12		
2 3 1 03 12 23		

2 3 1 01 04 12		
2 3 1 03 12 12		
2 4 1 01 12 12 13		
2 4 1 01 02 12 13		
2 4 1 01 02 03 12		
2 4 1 01 12 13 34		
2 4 1 03 14 12 14		



While I have gotten a long way towards a good algorithm for drawing the Kontsevich graphs in LaTeX, the problem hasn't been completely solved yet.

Firstly, one could consider adding more terms to the target function. Some ideas for other terms in the target function are a *horizontal lines score*, a score that increases if an upper-bound for the highest vertex has been crossed, an *edges almost overlapping score*, a score that increases if two edges are situated close to each other and a *symmetry score*. I actually considered adding a symmetry score of the form

$$SS(v_x) = \sum_{i=1}^{n} \begin{cases} c_{ss} \cdot (\delta/2 - v_{x,i})^{\eta} & \text{if } v_{x,i} \leq \delta/2 \\ c_{ss} \cdot (v_{x,i} - \delta/2)^{\eta} & \text{if } v_{x,i} > \delta/2 \end{cases}.$$

I did ultimately decide against it in order to be able to properly draw the asymmetric graphs. (Like the graph with encoding 2 2 1 01 12.) But maybe

there is another way to add a similar score to the target function that does take into account the asymmetric graphs.

Another thing that could be considered in future research is using the command `\line` instead of the command `\vector` to draw the edges. For the inclines of lines in the picture environment, both the numerator and the denominator can be chosen from -6 to 6 inclusive. All possible line inclines are given in Figure 67. Comparing this to Figure 4, it can be seen that there

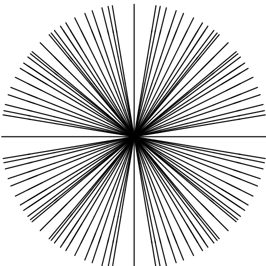


Figure 67: All possible inclines for lines in the LaTeX picture environment.

is a higher density in the lines than in the vectors. We could use the inclines of the lines to draw the graphs and attach a zero length vector of a slightly different incline to it. The difference in incline will only be small and hard to see. See Figure 68 for an illustration.

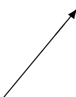


Figure 68: A line with direction `(5,6)` with a zero length vector with direction `(3,4)` attached to it.

I drew the 2-cycles in this project using the `\qbezier` command. This command takes 3 points as input, and draws a quadratic Bezier curve from them, using the first and last point as end points and the middle point as a control point. For example, the code

```
\qbezier(0,0)(40,60)(100,20)
```

draws the curve in Figure 69 (the dotted lines illustrate how the curve is determined). It would be useful to extend the algorithm to be able to compute

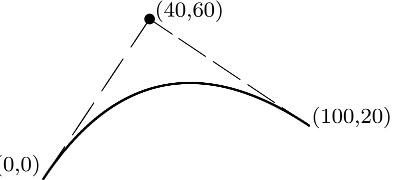


Figure 69: An example of a Bezier curve.

the control point of the Bezier curves for the 2-cycles.

Something that R.Buring suggested and that I actually did, was building a function that generates an encoding and coordinates for the mirror image of a graph. The function can be found in in Section A.2.18. Note that I say an encoding, by permutation of the labels of the vertices there are multiple encodings for the most graphs. For example, the encoding 2 2 1 01 12 defines the same graph as the encoding 2 2 1 13 01.

Acknowledgements

I would like to thank my supervisor A.V. Kiselev for giving me multiple points of feedback during the course of this project. I am also grateful to H.N. Kern for looking over my probability theory calculations and reassuring me that they are sound. But mostly, I couldn't have done a lot of things in my project without the help of R. Buring, who helped me tremendously with all the problems I had during coding the algorithm, including but not limited to explaining what the target function was supposed to do, pointing me to a technique to find intersections of lines and explaining the sign element of the encoding to me.

References

[1] Jørgen Bang-Jensen and Gregory Z Gutin. *Digraphs: theory, algorithms and applications*. Springer Science & Business Media, 2008.

[2] R Buring and AV Kiselev. The expansion $\star$ mod $\bar{o}(\hbar^4)$ and computer-assisted proof schemes in the Kontsevich deformation quantization. *Experimental Mathematics*, pages 1–54, 2019.

[3] Ricardo Buring, Arthemy V Kiselev, and Nina J Rutten. Poisson brackets symmetry from the pentagon-wheel cocycle in the graph complex. *Physics of Particles and Nuclei*, 49(5):924–928, 2018.

[4] Reinhard Diestel. *Graph Theory. Number 173 in Graduate Texts in Mathematics*. Springer, 2012.

[5] Chiara Esposito. *Formality Theory: From Poisson Structures to Deformation Quantization*, volume 2. Springer, 2014.

[6] Andrew S Glassner. *Graphics gems*. Elsevier, 2013.

[7] AV Kiselev. An algorithm to draw the Kontsevich graphs in LaTeX. in: Syllabus Propaedeutic project (Applied) Mathematics, acad. year 2017-2018. JBI RUG. -2p.

[8] Maxim Kontsevich. Formal (non)-commutative symplectic geometry. In *The Gelfand mathematical seminars, 1990–1992*, pages 173–187. Springer, 1993.

[9] Maxim Kontsevich. Feynman diagrams and low-dimensional topology. In *First European Congress of Mathematics Paris, July 6–10, 1992*, pages 97–121. Springer, 1994.

[10] Maxim Kontsevich. Homological algebra of mirror symmetry. In *Proceedings of the international congress of mathematicians*, pages 120–139. Springer, 1995.

[11] Maxim Kontsevich. Formality conjecture. *Deformation theory and symplectic geometry*, 128:139–156, 1997.

[12] Maxim Kontsevich. Deformation quantization of Poisson manifolds. *Letters in Mathematical Physics*, 66(3):157–216, 2003.

[13] Leslie Lamport. *LaTeX: a document preparation system: user's guide and reference manual*. Addison-Wesley, 1994.

[14] Erik Panzer. Multiple zeta values in deformation quantization. https://www.maths.ox.ac.uk/node/31248, 2019. Accessed on: 2020-07-11.

[15] Gareth Rees. How do you detect where two line segments intersect? https://stackoverflow.com/a/565282, 2009. Accessed on: 2020-07-11.

A Appendix

A.1 Matlab code

A.1.1 The function graphfunc

This function takes a Kontsevich graph encoding and outputs the corresponding graph.

```
function G=graphfunc(inputstring)
    %inputstring needs to be in the form of a vector
    %e.g. the encoding 221 01 01 must be put in the form
   [2,2,1,0,1,0,1]

    m = inputstring(1);
    n = inputstring(2);
    num_edges = 0.5*(length(inputstring)-3);
    edges = zeros(num_edges,2);

    %define a matrix to store the decoding of the edges
        %the +1 is neccessary because you can't have 0 as an
   index
    for i = 1:num_edges
        edges(i,:)=[inputstring(2*i+2)+1,inputstring(2*i+3)
   +1];
    end

    %making the graph
    G = digraph();
    G = addnode(G, m+n);        %number of nodes of the graph

                                %constructing the edges
    for i=1:n
    G = addedge(G, m+i, [edges(i,1),edges(i,2)]);
    end

    %the output
    G
    plot(G,'Layout','layered','Sinks',[1,2])
```

A.2 Sage Code

All code, including history, can also be found at: https://github.com/SKerkhove/Bachelors-Project.

A.2.1 The Function to generate the vector b

```
1  def b_vector(encoding,delta,inclines):
2      #saved under the name 'b_vector_algorithm'
3      r"""
4      Return the vector $b$ in the system $Ax=b$
5  
6      INPUT:
7  
8      - ''encoding'' - a vector of the form $[m,n,s,e_1,e_2
   ,...,e_2n]$
9  
10     - ''delta'' - a positive scalar
11 
12     - ''inclines'' - a matrix of 2n inclines, of the form $[
   a_1^L,b_1^L; ... ;a_n^R,b_n^R]$
13 
14     OUTPUT:
15 
16     - a vector $b$ of size 2n x 1
17 
18     EXAMPLES:
19 
20         sage: b_vector(vector([2,1,1,0,1]),1,matrix
   ([[1,1],[-1,1]]))
21         (0,1)
22         sage: b_vector(vector([2,2,1,0,1,1,2]),1,\matrix
   ([[0,-1],[1,-1],[-1,0],[0,-1]]))
23         (0,-1,0,0)
24     """
25     m=encoding[0]
26     n=encoding[1]
27 
28     b=zero_vector(2*n)
29     two_cycles_list=[]
30     for i in range(2*n):
31         e=encoding[i+3]
32         if e < m:
33             b[i]=e*delta*inclines[i,1]
34         else:
35             if encoding[2*e-1]==m+floor(i/2) or encoding[2*e
   ]==m+floor(i/2):
36                 two_cycle=sorted([e,m+floor(i/2)])
37                 if two_cycle in two_cycles_list:
38                     continue
```

```
                b[i]=delta
                two_cycles_list.append(two_cycle)

    return b
```

A.2.2 The Function to generate the matrix A

```
def A_matrix(encoding,inclines):
    #saved under the name A_matrix_algorithm
    r"""
    Return the matrix A in the system Ax=b

    INPUT:

    - ''encoding'' - a vector of the form (m,n,s,e_1,e_2
  ,...,e_2n)

    - ''inclines'' - a 2n x 2 matrix of the form [a_1^L,b_1^
  L;...;a_n^R,b_n^R]

    OUTPUT:

    - a matrix A of size 2n x 2n

    EXAMPLES:
        sage: A_matrix(vector([2,1,1,0,1]),matrix([[1 ,1] ,[
   -1 ,1]]))
        [1,-1]
        [1,1]

        sage: A_matrix(vector([2,2,1,0,1,1,2]),matrix
  ([[0,-1],[1,-1],[-1,0],[0,-1]]))
        [-1  0  0  0]
        [-1 -1  0  0]
        [ 0  0  0  1]
        [ 1  0 -1  0]

    """
    m=encoding[0]
    n=encoding[1]
    A=zero_matrix(2*n,2*n)
    two_cycles_list=[]
    for i in range(2*n):
        e=encoding[i+3]
        if e>=m:
            if encoding[2*e-1]==m+floor(i/2) or encoding[2*e
```

```
    ]==m+floor(i/2):
36                  two_cycle=sorted([e,m+floor(i/2)])
37                  if two_cycle in two_cycles_list:
38                      if i%2 == 0:
39                          A[i,i]=inclines[i,1]
40                          A[i,i+1]=-inclines[i,0]
41                          if e >= m:
42                              A[i,2*(e-m+1)-2]=-inclines[i,1]
43                              A[i,2*(e-m+1)-1]=inclines[i,0]
44                      if i%2 == 1:
45                          A[i,i]=-inclines[i,0]
46                          A[i,i-1]=inclines[i,1]
47                          if e >= m:
48                              A[i,2*(e-m+1)-2]=-inclines[i,1]
49                              A[i,2*(e-m+1)-1]=inclines[i,0]
50                      continue
51                  two_cycles_list.append(two_cycle)
52                  if i%2 == 0:
53                      A[i,i+1]=1
54                  if i%2 == 1:
55                      A[i,i]=1
56                  continue
57          if i%2 == 0:
58              A[i,i]=inclines[i,1]
59              A[i,i+1]=-inclines[i,0]
60              if e >= m:
61                  A[i,2*(e-m+1)-2]=-inclines[i,1]
62                  A[i,2*(e-m+1)-1]=inclines[i,0]
63          if i%2 == 1:
64              A[i,i]=-inclines[i,0]
65              A[i,i-1]=inclines[i,1]
66              if e >= m:
67                  A[i,2*(e-m+1)-2]=-inclines[i,1]
68                  A[i,2*(e-m+1)-1]=inclines[i,0]
69
70
71      return A
```

A.2.3 the Function Inclines to Coordinates

```
1 def inclines_to_coordinates(encoding,inclines,delta):
2     #saved under the name 'inclines_to_coordinates'
3
4     r"""
5     returns a vector x containing the coordinates of the
    system defined by the encoding, inclines and delta
```

```
6
7       INPUT:
8
9       - ''encoding'' - a vector of the form $[m,n,s,e_1,e_2
    ,...,e_2n]$
10
11      - ''inclines'' - a matrix of 2n inclines, of the form $[
    a_1^L,b_1^L; ... ;a_n^R,b_n^R]$
12
13      - ''delta'' - a positive scalar
14
15      OUTPUT:
16
17      - a 2n x 1 vector x containing the coordinates of the
    form (x_1,y_1,...,x_n,y_n)
18
19      EXAMPLES:
20          sage: inclines_to_coordinates(vector([2,1,1,0,1]),
    matrix([[1 ,1] ,[ -1 ,1]]),1)
21          (1/2,1/2)
22          sage: inclines_to_coordinates(vector
    ([2,2,1,0,1,0,1]),matrix([[0,1],[1,-1],[1,1],[0,1]]),1)
23          (0,1,1,1)
24
25      """
26      A=A_matrix(encoding,inclines)
27      b=b_vector(encoding,delta,inclines)
28
29      if det(A)==0:
30          x=0
31      if abs(det(A))>0:
32          x=A\b
33
34      return x
```

A.2.4 The Function DrawGraph algorithm

```
1 def DrawGraph_algorithm(encoding,delta):
2       #saved under the name 'DrawGraph_algorithm'
3       r"""
4       returns a matrix with coordinates of the internal nodes
5
6       INPUT:
7
8       - ''encoding'' - a vector of the form $[m,n,s,e_1,e_2
    ,...,e_2n]$
```

```
    - ''delta'' - a positive scalar

    OUTPUT:

    - an n x 2 matrix containing coordinates in the form [
  x_1,y_1;...;x_n,y_n],
        because of randomization, the coordinates can change
   with each use of the function

    EXAMPLES:
        sage: DrawGraph_algorithm(vector([2,1,1,0,1]),1)
        [ 0.818181818181818 -0.272727272727273]
        sage: DrawGraph_algorithm(vector([2,2,1,0,1,0,1]),1)
        [ 3.00000000000000  3.00000000000000]
        [0.307692307692308 0.230769230769231]

    """

    #we're first generating all possible inclines, stored in
   set_of_inclines
    positive_inclines=matrix
  ([[0,1],[1,0],[1,1],[1,2],[1,3],[1,4],\
    [2,1],[2,3],[3,1],[3,2],[3,4],[4,1],[4,3]])
    nofinclines=(positive_inclines.nrows()-1)*2
    half_nofi=nofinclines/2
    set_of_inclines=zero_matrix(nofinclines,2)
    for i in range(nofinclines):
        if i<=half_nofi:
            set_of_inclines[i,:]=positive_inclines[i,:]
        if i>half_nofi:
            set_of_inclines[i,0]=positive_inclines[i-
  half_nofi+1,0]
            set_of_inclines[i,1]=-positive_inclines[i-
  half_nofi+1,1]

    #from here on we will be computing the solution x
    #for a random set of inclines
    n=encoding[1]
    inclines=zero_matrix(2*n,2)
    x=0
    while x==0:
        for i in range(2*n):
            k=ZZ.random_element(0,nofinclines)
            inclines[i,:]=set_of_inclines[k,:]
```

```
                if i%2 == 1:
                    if inclines[i,:]==inclines[i-1,:]:
                        if k==nofinclines-1:
                            inclines[i,:]=set_of_inclines[0,:]
                        if k<nofinclines-1:
                            inclines[i,:]=set_of_inclines[k+1,:]
            x=inclines_to_coordinates(encoding,inclines,delta)


    #changing the look of the output
    coordinates=zero_matrix(RR,n,2)
    for i in range(n):
        coordinates[i,0]=x[2*i]
        coordinates[i,1]=x[2*i+1]

    return coordinates
```

A.2.5 The Function Positive Test

```
def positive_test(coordinates):

    r"""
    returns a value True or False based on whether the
    coordinates all lie above the line y=0

    INPUT:

    - ''coordinates'' - an n x 2 matrix containing
    coordinates in the form [x_1,y_1;...;x_n,y_n]

    OUTPUT:

    - a value False or True, indicating whether the input
    matrix had coordinates below or on the line y=0

    EXAMPLES:
        sage: positive_test(matrix([[1,4],[2,3]]))
        True

        sage: positive_test(matrix([[1,4],[2,-2]]))
        False

        sage: positive_test(matrix([[1,0],[2,3],[4,5]]))
        False

    """
```

```
    n=coordinates.nrows()
    value=True
    for i in range(n):
        if coordinates[i,1]<=0:
            value = False
            break

    return value
```

A.2.6 The Function Same Vertices Test

```
def same_vertices_test(encoding, delta, n_coordinates):

    r"""
    returns a value True or False indicating whether the
   coordinates all indicate unique points

    INPUT:

    - ''encoding'' - a vector of the form $[m,n,s,e_1,e_2
   ,...,e_2n]$

    - ''delta'' - a positive scalar

    - ''n_coordinates'' - an n x 2 matrix containing
   coordinates in the form [x_1,y_1;...;x_n,y_n]

    OUTPUT:

    - a value False or True, indicating whether the input
   matrix has unique coordinates

    EXAMPLES:
        sage: same_vertices_test(matrix([[1,2],[3,4],[1,2]]))
        False
        sage: same_vertices_test(matrix([[1,2],[3,4],[1,3]]))
        True

    """
    m=encoding[0]
    m_coordinates=zero_matrix(RR,m,2)
    for i in range(m):
        m_coordinates[i,0]=i*delta

    coordinates=block_matrix([[m_coordinates],[n_coordinates
   ]])
```

```
31      k=coordinates.nrows()
32      value=True
33      for i in range(k):
34          for j in range(i+1,k):
35              if coordinates[i,:]==coordinates[j,:]:
36                  value = False
37                  break
38 
39      return value
```

A.2.7 The Function Length Test

```
1  def length_test(encoding,delta,coordinates,min_len):
2      r"""
3      returns a value False or True
4      depending on whether any of the edges defined by the
   coordinates and encoding has
5      a length smaller than the minimal value min_len
6 
7      INPUT:
8 
9       - ''encoding'' - a vector of the form $[m,n,s,e_1,e_2
   ,...,e_2n]$
10 
11      - ''delta'' - a positive scalar
12 
13      - ''coordinates'' - an n x 2 matrix containing
   coordinates in the form [x_1,y_1;...;x_n,y_n]
14 
15      - ''min_len'' - a positive scalar
16 
17     OUTPUT:
18 
19     - a value False or True, denoting whether all edges of
   the defined graph are larger than the minimum length
20 
21     EXAMPLES:
22         sage: length_test(vector([2,3,1,0,1,0,4,1,2]),40,
   matrix([[160,40],[210,105],[280,0]]),10)
23         True
24         sage: length_test(vector([2,1,1,0,1]),10,matrix
   ([[1,3]]),10)
25         False
26 
27     """
28 
```

```
    m=encoding[0]
    n=encoding[1]
    value=True

    sink_coord=zero_matrix(RR,m,2)
    for i in range(m):
        sink_coord[i,0]=i*delta

    for i in range(n):
        source_vert_x=coordinates[i,0]
        source_vert_y=coordinates[i,1]


        goal_vert_L=encoding[3+2*i]
        goal_vert_R=encoding[4+2*i]
        if goal_vert_L <m:
            goal_vert_L_x=sink_coord[goal_vert_L,0]
            goal_vert_L_y=sink_coord[goal_vert_L,1]
        if goal_vert_L>=m:
            goal_vert_L_x=coordinates[goal_vert_L-m,0]
            goal_vert_L_y=coordinates[goal_vert_L-m,1]

        if goal_vert_R <m:
            goal_vert_R_x=sink_coord[goal_vert_R,0]
            goal_vert_R_y=sink_coord[goal_vert_R,1]
        if goal_vert_R>=m:
            goal_vert_R_x=coordinates[goal_vert_R-m,0]
            goal_vert_R_y=coordinates[goal_vert_R-m,1]

        delta_L_x=source_vert_x-goal_vert_L_x
        delta_L_y=source_vert_y-goal_vert_L_y
        delta_R_x=source_vert_x-goal_vert_R_x
        delta_R_y=source_vert_y-goal_vert_R_y


        edge_length_sq=delta_L_x^2+delta_L_y^2
        edge_length=sqrt(edge_length_sq)
        if edge_length<min_len:
            value=False
            break

        edge_length_sq=delta_R_x^2+delta_R_y^2
        edge_length=sqrt(edge_length_sq)
        if edge_length<min_len:
```

```
74              value=False
75              break
76
77
78     return value
```

A.2.8 The Function Overlapping Edges Test

```
1  def overlapping_edges_test(encoding,delta,n_coordinates):
2      r"""
3      returns a value False or True, depending on whether the
   defined graph has overlapping edges
4
5      INPUT:
6
7        - ''encoding'' - a vector of the form $[m,n,s,e_1,e_2
   ,...,e_2n]$
8
9        - ''delta'' - a positive scalar
10
11       - ''n_coordinates'' - an n x 2 matrix containing
   coordinates in the form [x_1,y_1;...;x_n,y_n]
12
13      OUTPUT:
14
15       - a value False or True,  depending on whether the
   defined graph has overlapping edges
16
17      EXAMPLES:
18          sage: overlapping_edges_test(vector([2,2,1,0,1,1,2])
   ,10,matrix([[0,10],[5,5]]))
19          False
20          sage: overlapping_edges_test(vector([2,2,1,0,1,1,2])
   ,10,matrix([[0,10],[10,10]]))
21          True
22
23      """
24      m=encoding[0]
25      n=encoding[1]
26      value=True
27
28      sink_coord=zero_matrix(RR,m,2)
29      for i in range(m):
30          sink_coord[i,0]=i*delta
31
32      coordinates=block_matrix([[sink_coord],[n_coordinates]])
```

```
33
34     edges_source_goal=zero_matrix(QQ,2*n,4)
35     for i in range(2*n):
36         if i%2==0:
37             source_vertex_label=i/2+m
38         else:
39             source_vertice_label=(i-1)/2+m
40         source_vertex_x=coordinates[source_vertex_label,0]
41         source_vertex_y=coordinates[source_vertex_label,1]
42
43         goal_vertex_label=encoding[3+i]
44         goal_vertex_x=coordinates[goal_vertex_label,0]
45         goal_vertex_y=coordinates[goal_vertex_label,1]
46
47         edges_source_goal[i,:]=vector([source_vertex_x,
   source_vertex_y,goal_vertex_x,goal_vertex_y])
48
49     for i in range(2*n):
50         p_0=vector([edges_source_goal[i,0],edges_source_goal[
   i,1]])
51         p_1=vector([edges_source_goal[i,2],edges_source_goal[
   i,3]])
52         if p_0==p_1:
53             continue
54         r=p_1-p_0
55
56         for j in range(i+1,2*n):
57             q_0=vector([edges_source_goal[j,0],
   edges_source_goal[j,1]])
58             q_1=vector([edges_source_goal[j,2],
   edges_source_goal[j,3]])
59             if q_0==q_1:
60                 continue
61
62             if p_0 == q_1 and p_1==q_0:
63                 continue
64
65             s=q_1-q_0
66             r_cross_s=r[0]*s[1]-r[1]*s[0]
67
68             q_min_p=q_0-p_0
69             q_min_p_cross_r=q_min_p[0]*r[1]-q_min_p[1]*r[0]
70
71             if r_cross_s == 0 and q_min_p_cross_r == 0:
72                 t_0=q_min_p*r/(r*r)
```

```
73                 t_1=t_0+s*r/(r*r)
74                 if 0<=t_0<=1 or 0<=t_1<=1:
75                     if t_0 == 0 and t_1<0:
76                         continue
77                     if t_0 == 1 and t_1>1:
78                         continue
79                     if t_1 == 0 and t_0<0:
80                         continue
81                     if t_1 == 1 and t_1>1:
82                         continue
83                     value = False
84                     break
85 
86     return value
```

A.2.9 The Function DrawGraph Filter

The first code is the original version. The second is to use when one has done a full run through of all possible inclines.

```
1 def DrawGraph_filter(encoding,delta,min_len):
2 
3     r"""
4     returns coordinates that are a proper solution to the
    system
5 
6     INPUT:
7 
8      - ''encoding'' - a vector of the form $[m,n,s,e_1,e_2
    ,...,e_2n]$
9 
10     - ''delta'' - a positive scalar
11 
12     - ''min_len'' - a positive scalar
13 
14     OUTPUT:
15 
16     - an (m+n) x 2 matrix containing coordinates in the form
     [x_1,y_1;...;x_m+n,y_m+n],
17          because of randomization, the coordinates can change
     with each use of the function
18 
19     EXAMPLES:
20          sage: DrawGraph_filter(vector([2,1,1,0,1]),40,10)
21          [  0.000000000000   0.000000000000]
22          [ 40.000000000000   0.000000000000]
```

```
        [--------------------------------]
        [120.000000000000 240.000000000000]

        sage: DrawGraph_filter(vector([2,2,1,0,1,0,1])
,40,10)
        [  0.0000000000000    0.0000000000000]
        [ 40.0000000000000    0.0000000000000]
        [------------------------------------]
        [-32.0000000000000   96.0000000000000]
        [ 48.0000000000000   16.0000000000000]

    """
    value = False
    k=0
    while value == False:
        n_coordinates=DrawGraph_algorithm(encoding,delta)
        v1=positive_test(n_coordinates)
        v2=same_vertices_test(encoding, delta, n_coordinates)
        v3=length_test(encoding,delta,n_coordinates,min_len)
        v4=overlapping_edges_test(encoding,delta,
n_coordinates)
        value = all([v1,v2,v3,v4])
        k=k+1
        if k == 10^3:
            n_coordinates=0
            break

    m=encoding[0]
    m_coordinates=zero_matrix(RR,m,2)
    for i in range(m):
        m_coordinates[i,0]=i*delta
    if n_coordinates!=0:
        coordinates=block_matrix([[m_coordinates],[
n_coordinates]])
    else:
        coordinates=0

    return coordinates
```

```
def DrawGraph_filter_incl(encoding,n_coordinates,delta,
min_len):

    r"""
    returns coordinates that are a proper solution to the
system

```

```
6     INPUT:
7
8      - ''encoding'' - a vector of the form $[m,n,s,e_1,e_2
    ,...,e_2n]$
9
10     - ''n_coordinates'' - a n x 2 matrix of the form [x_1,
    y_1;...;x_n,y_n]
11
12     - ''delta'' - a positive scalar
13
14     - ''min_len'' - a positive scalar
15
16     OUTPUT:
17
18     - an (m+n) x 2 matrix containing coordinates in the form
     [x_1,y_1;...;x_m+n,y_m+n],
19         because of randomization, the coordinates can change
     with each use of the function
20
21     EXAMPLES:
22         sage: DrawGraph_filter(vector([2,1,1,0,1]),40,10)
23         [  0.000000000000   0.000000000000]
24         [ 40.000000000000   0.000000000000]
25         [---------------------------------]
26         [120.000000000000 240.000000000000]
27
28         sage: DrawGraph_filter(vector([2,2,1,0,1,0,1])
    ,40,10)
29         [  0.0000000000000   0.0000000000000]
30         [ 40.0000000000000   0.0000000000000]
31         [-----------------------------------]
32         [-32.0000000000000  96.0000000000000]
33         [ 48.0000000000000  16.0000000000000]
34
35    """
36    if n_coordinates == 0:
37        value = False
38    else:
39        v1=positive_test(n_coordinates)
40        v2=same_vertices_test(encoding, delta, n_coordinates)
41        v3=length_test(encoding,delta,n_coordinates,min_len)
42        v4=overlapping_edges_test(encoding,delta,
    n_coordinates)
43        value = all([v1,v2,v3,v4])
44
```

```
45
46     return value
```

A.2.10 The Function DrawGraph Compute and Draw

```
1 def DrawGraph_compute_and_draw(encoding,delta,min_len):
2     r"""
3     returns a randomized plot of the graph defined by the
    encoding
4
5     INPUT:
6
7      - ''encoding'' - a vector of the form $[m,n,s,e_1,e_2
    ,...,e_2n]$
8
9      - ''delta'' - a positive scalar
10
11     - ''min_len'' - a positive scalar
12
13     OUTPUT:
14
15     - a graphics plot
16
17    """
18
19    m=encoding[0]
20    n=encoding[1]
21
22    coord=DrawGraph_filter(encoding,delta,min_len)
23    coordinates=coord.rows()
24    points=point2d(coord,size=50)
25    plot_list=[points]
26
27    for i in range(n):
28        source_vertex=coordinates[m+i]
29
30        left_vertex_label=encoding[2*i+3]
31        right_vertex_label=encoding[2*i+4]
32
33        left_vertex=coordinates[left_vertex_label]
34        right_vertex=coordinates[right_vertex_label]
35
36        left_edge=arrow2d(source_vertex,left_vertex)
37        right_edge=arrow2d(source_vertex,right_vertex)
```

```
        plot_list.append(left_edge)
        plot_list.append(right_edge)

        plot=points
    for i in range(len(plot_list)-1):
        plot+=plot_list[i+1]


    return plot
```

A.2.11 The Function DrawGraph Draw

```
def DrawGraph_draw(encoding,coord):
    r"""
    returns a randomized plot of the graph defined by the
   encoding

    INPUT:

    - ''encoding'' - a vector of the form $[m,n,s,e_1,e_2
   ,...,e_2n]$

    - ''coord'' - an (m+n) x 2 matrix containing coordinates
   in the form [x_1,y_1;...;x_m+n,y_m+n]

    OUTPUT:

    - a graphics plot

    """

    m=encoding[0]
    n=encoding[1]


    coordinates=coord.rows()
    points=point2d(coord,size=50)
    plot_list=[points]

    for i in range(n):
        source_vertice=coordinates[m+i]

        left_vertex_label=encoding[2*i+3]
        right_vertex_label=encoding[2*i+4]

```

```
        left_vertex=coordinates[left_vertex_label]
        right_vertex=coordinates[right_vertex_label]

        left_edge=arrow2d(source_vertice,left_vertex)
        right_edge=arrow2d(source_vertice,right_vertex)

        plot_list.append(left_edge)
        plot_list.append(right_edge)

        plot=points
    for i in range(len(plot_list)-1):
        plot+=plot_list[i+1]


    return plot
```

A.2.12 The Target Function

```
def target_function(encoding,delta,coordinates):
    r"""
    returns a score denoting how 'beautiful' the drawing of
    the graph is

    INPUT:

    - ''encoding'' - a vector of the form $[m,n,s,e_1,e_2
    ,...,e_2n]$

    - ''coordinates'' - an (n+m) x 2 matrix containing
    coordinates in the form [x_1,y_1;...;x_n+m,y_n+m]

    OUTPUT:

    - a scalar, the larger the value, the less 'beautiful'
    the drawing is

    EXAMPLES:

    """

    #the parameters
    L_0 =  50        #short and long edges score
    L_1 = 20         #distance between unrelated vertices
    L_2 = 0.4*L_0    #height y-coordinate
    c_se = 5         #short edges
    c_le = 16        #long edges
```

```
25    c_dv = 10          #distance between unrelaed vertices
26    c_ss = 0           #symmetry
27    c_hy = 1          #height y-coordinate
28    c_os = 1           #overshoot
29    alpha = 0.9        #short edges
30    beta = 1           #long edges
31    gamma = 15         #intersections
32    epsilon = 5        #intersections
33    zeta = 5           #unrelated vertices
34    eta = 10           #symmetry
35    theta = 20         #height y-coordinate
36    iota = 5           #overshoot
37    kappa = 10         #points on one line
38    labda = 5          #points on one line
39    mu = 2             #horizontal lines
40    nu = 1             #horizontal lines
41
42    #getting the prerequisites from the data
43    m=encoding[0]
44    n=encoding[1]
45
46    #vertices_sink is an n x 4 matrix of which the first two
     columns give the coordinates for the left vertex with\
47    #which the source is connected and the column 2 and 3
     give the coordinates for the right vertex
48    #edges_length is an n x 2 matrix of which the first
     column gives the length of the left edge of the source
     vertex\
49    #and the second column gives the length of the right edge
50    #edges_begin_end is a 2n x 4 matrix containing the begin
     and end point of each edges in the rows
51    vertices_sink = zero_matrix(RR,n,4)
52    edges_length=zero_matrix(RR,n,2)
53    edges_begin_end=zero_matrix(RR,2*n,4)
54
55    for i in range(n):
56        e_L=encoding[3+2*i]
57        e_R=encoding[4+2*i]
58        vertices_sink[i,0:2]=coordinates[e_L,:]
59        vertices_sink[i,2:4]=coordinates[e_R,:]
60
61        coord_source_x=coordinates[m+i,0]
62        coord_source_y=coordinates[m+i,1]
63        edges_begin_end[[2*i,2*i+1],0]=coord_source_x
64        edges_begin_end[[2*i,2*i+1],1]=coord_source_y
```

```
        coord_sink_L_x=coordinates[e_L,0]
        coord_sink_L_y=coordinates[e_L,1]
        edges_begin_end[2*i,2]=coord_sink_L_x
        edges_begin_end[2*i,3]=coord_sink_L_y


        coord_sink_R_x=coordinates[e_R,0]
        coord_sink_R_y=coordinates[e_R,1]
        edges_begin_end[2*i+1,2]=coord_sink_R_x
        edges_begin_end[2*i+1,3]=coord_sink_R_y

        delta_x_L=coord_source_x-coord_sink_L_x
        delta_y_L=coord_source_y-coord_sink_L_y

        delta_x_R=coord_source_x-coord_sink_R_x
        delta_y_R=coord_source_y-coord_sink_R_y

        edges_length[i,0] = sqrt(delta_x_L^2+delta_y_L^2)
        edges_length[i,1] = sqrt(delta_x_R^2+delta_y_R^2)


    # short edges
    short_edges_score=0
    #long edges
    long_edges_score = 0
    for i in range(n):
        length_L=edges_length[i,0]
        length_R=edges_length[i,1]

        left_short_edges_score = c_se*(L_0/length_L)^alpha
        right_short_edges_score =c_se*(L_0/length_R)^alpha

        left_long_edges_score = c_le*(length_L/L_0)^beta
        right_long_edges_score = c_le*(length_R/L_0)^beta

        short_edges_score += left_short_edges_score+
right_short_edges_score
        long_edges_score += left_long_edges_score+
right_long_edges_score

    #intersections score
    intersections_score = 0
    num_of_intersections=1

```

```
108     for i in range(2*n):
109         p_0=vector([edges_begin_end[i,0],edges_begin_end[i
    ,1]])
110         p_1=vector([edges_begin_end[i,2],edges_begin_end[i
    ,3]])
111 
112         for j in range(i+1,2*n):
113             q_0=vector([edges_begin_end[j,0],edges_begin_end[
    j,1]])
114             q_1=vector([edges_begin_end[j,2],edges_begin_end[
    j,3]])
115 
116             if p_0==q_0 or p_0 == q_1:
117                 continue
118             if p_1 == q_0 or p_1 == q_1:
119                 continue
120 
121             value=intersection_test(p_0,p_1,q_0,q_1)
122             if value == True:
123                 num_of_intersections+=1
124 
125     intersections_score = num_of_intersections^gamma*RDF(log(
    num_of_intersections))^epsilon
126 
127     #distance between unrelated vertices
128     distance_vertices_score=0
129     for i in range(n):
130         vertex_1_x=coordinates[i+m,0]
131         vertex_1_y=coordinates[i+m,1]
132         for j in range(i+m):
133             if j == encoding[3+2*i] or j == encoding[4+2*i]:
134                 continue
135             vertex_2_x=coordinates[j,0]
136             vertex_2_y=coordinates[j,1]
137             Length_ij_sq=(vertex_2_x-vertex_1_x)^2+(
    vertex_2_y-vertex_1_y)^2
138             Length_ij=sqrt(Length_ij_sq)
139             distance_vertices_score+=c_dv*(L_1/Length_ij)^
    zeta
140 
141         for j in range(i+m+1,n+m):
142             if j == encoding[3+2*i] or j == encoding[4+2*i]:
143                 continue
144             vertex_2_x=coordinates[j,0]
145             vertex_2_y=coordinates[j,1]
```

```
146             Length_ij_sq=(vertex_2_x-vertex_1_x)^2+(
    vertex_2_y-vertex_1_y)^2
147             Length_ij=sqrt(Length_ij_sq)
148             distance_vertices_score+=c_dv*(L_1/Length_ij)^
    zeta
149 
150      #height y-coordinate score
151      height_y_score=0
152      for i in range(n):
153          vertex_y=coordinates[m+i,1]
154          height_y_score+=c_hy*(L_2/vertex_y)^theta
155 
156      #overshoot score
157      overshoot_score = 0
158      for i in range(n):
159          vertex_x=coordinates[m+i,0]
160          if vertex_x>delta:
161              overshoot_score += c_os*(vertex_x-delta)^iota
162          if vertex_x<0:
163              overshoot_score += c_os*abs(vertex_x)^iota
164 
165      #points on line score
166      num_vert_on_line = points_on_line_test(coordinates)
167      points_on_line_score = -num_vert_on_line^kappa*RDF(log(
    num_vert_on_line))^labda
168 
169      #horizontal 'lines' score
170      num_horizontal_lines=1
171      for i in range(m+n):
172          p_x=coordinates[i,0]
173          for j in range(i+1,m+n):
174              q_x=coordinates[j,0]
175              if p_x==q_x:
176                  num_horizontal_lines+=1
177      horizontal_lines_score = -num_horizontal_lines^mu*RDF(log
    (num_horizontal_lines))^nu
178 
179      #symmetry score
180      symmetry_score=0
181      for i in range(n):
182          vertex_x=coordinates[m+i,0]
183          if vertex_x<=0.5*delta:
184              symmetry_score+=-c_ss*(0.5*delta-vertex_x)^eta
185          if vertex_x>0.5*delta:
186              symmetry_score+=c_ss*(vertex_x-0.5*delta)^eta
```

```
187
188     score=short_edges_score+long_edges_score+
    intersections_score\
189     +distance_vertices_score+height_y_score\
190     +overshoot_score+horizontal_lines_score+
    points_on_line_score+symmetry_score
191
192
193     return score
```

A.2.13 The Function Intersection Test

```
1 def intersection_test(p_1,p_2,q_1,q_2):
2     r"""
3     returns a value True or False indicating whether the line
      segments defined by the points intersect
4
5     INPUT:
6
7     - ''p_1'' - a vector indicating the source point of the
    first line segment
8
9     - ''p_2'' - a vector indicating the end point of the
    first line segment
10
11    - ''q_1'' - a vector indicating the source point of the
    second line segment
12
13    - ''q_2'' - a vector indicating the end point of the
    second line segment
14
15    OUTPUT:
16
17    - a value True or False, depending on whether the line
    segments intersect
18
19    EXAMPLES:
20        sage: intersection_test(vector([1,1]),vector([4,4]),
    vector([3,0]),vector([1,4]))
21        True
22        sage: intersection_test(vector([1,0]),vector([3,3]),
    vector([5,2]),vector([3,5]))
23        False
24    """
25
26    value=False
```

```
    #computing r
    r=zero_vector(RR,2)
    r[0]=p_2[0]-p_1[0]
    r[1]=p_2[1]-p_1[1]

    #computing s
    s=zero_vector(RR,2)
    s[0]=q_2[0]-q_1[0]
    s[1]=q_2[1]-q_1[1]

    r_x_s=r[0]*s[1]-r[1]*s[0]
    if r_x_s!=0:
        q_min_p=q_1-p_1
        q_min_p_x_s=q_min_p[0]*s[1]-q_min_p[1]*s[0]
        t=q_min_p_x_s/r_x_s

        q_min_p_x_r=q_min_p[0]*r[1]-q_min_p[1]*r[0]
        u=q_min_p_x_r/r_x_s

        if 0<=t<=1 and 0<=u<=1:
            value=True
            if t == 0:
                if u == 0 or u == 1:
                    value=False
            if t == 1:
                if u == 0 or u == 1:
                    value=False


    return value
```

A.2.14 The Function Inclines to coordinates

```
def inclines_to_coordinates(encoding,inclines,delta):
     #saved under the name 'inclines_to_coordinates'

     r"""
     returns a vector x containing the coordinates of the
    system defined by the encoding, inclines and delta

     INPUT:

     - ''encoding'' - a vector of the form $[m,n,s,e_1,e_2
    ,...,e_2n]$

     - ''inclines'' - a matrix of 2n inclines, of the form $[
```

```
    a_1^L,b_1^L; ... ;a_n^R,b_n^R]$
12
13      - ''delta'' - a positive scalar
14
15      OUTPUT:
16
17      - a 2n x 1 vector x containing the coordinates of the
    form (x_1,y_1,...,x_n,y_n)
18
19      EXAMPLES:
20          sage: inclines_to_coordinates(vector([2,1,1,0,1]),
    matrix([[1 ,1] ,[ -1 ,1]]),1)
21          (1/2,1/2)
22          sage: inclines_to_coordinates(vector
    ([2,2,1,0,1,0,1]),matrix([[0,1],[1,-1],[1,1],[0,1]]),1)
23          (0,1,1,1)
24
25      """
26      A=A_matrix(encoding,inclines)
27      b=b_vector(encoding,delta,inclines)
28
29      if det(A)==0:
30          x=0
31      if abs(det(A))>0:
32          x=A\b
33
34      return x
```

A.2.15 The Function Generate List of Coordinates

```
1 def generate_list_of_coordinates(encoding,delta,min_len,
    num_of_iterations):
2      r"""
3      returns a list containing coordinates for the system
    determined by encoding and delta
4
5      INPUT:
6
7      - ''encoding'' - a vector of the form $[m,n,s,e_1,e_2
    ,...,e_2n]$
8
9      - ''delta'' - a positive scalar
10
11     - ''min_len'' - a positive scalar
12
13     - ''num_of_iterations'' - a positive integer
```

```
14
15     OUTPUT:
16
17     - a list containing num_of_iterations times a solution to
       the system
18
19     """
20
21     n=encoding[1]
22     list_of_coordinates=[]
23
24     if n<3:
25         #we're first generating all possible inclines, stored
       in set_of_inclines
26         positive_inclines=matrix
      ([[0,1],[1,0],[1,1],[1,2],[1,3],[1,4],\
27         [2,1],[2,3],[3,1],[3,2],[3,4],[4,1],[4,3]])
28         nofinclines=(positive_inclines.nrows()-1)*2
29         half_nofi=nofinclines/2
30         set_of_inclines=zero_matrix(nofinclines,2)
31         for i in range(nofinclines):
32             if i<=half_nofi:
33                 set_of_inclines[i,:]=positive_inclines[i,:]
34             if i>half_nofi:
35                 set_of_inclines[i,0]=positive_inclines[i-
      half_nofi+1,0]
36                 set_of_inclines[i,1]=-positive_inclines[i-
      half_nofi+1,1]
37
38         m=encoding[0]
39         m_coordinates=zero_matrix(RR,m,2)
40         for i in range(m):
41             m_coordinates[i,0]=i*delta
42         counter=0
43         for i in range(nofinclines):
44             inclines=zero_matrix(2*n,2)
45             inclines[0,:]=set_of_inclines[i,:]
46             for j in range(nofinclines):
47                 incl = set_of_inclines[j,:]
48                 if incl == inclines[0,:]:
49                     continue
50                 inclines[1,:]=incl
51                 if n == 1:
52                     x=inclines_to_coordinates(encoding,
      inclines,delta)
```

```
53                     if x == 0:
54                         continue
55                     n_coordinates=zero_matrix(RR,n,2)
56                     for q in range(n):
57                         n_coordinates[q,0]=x[2*q]
58                         n_coordinates[q,1]=x[2*q+1]
59                     value=DrawGraph_filter_incl(encoding,
    n_coordinates,delta,min_len)
60                     if value == False:
61                         counter+=1
62                         continue
63                     coordinates=block_matrix([[m_coordinates
    ],[n_coordinates]])
64                     list_of_coordinates.append(coordinates)
65                     continue
66                 for k in range(nofinclines):
67                     inclines[2,:]=set_of_inclines[k,:]
68                     for l in range(nofinclines):
69                         incl=set_of_inclines[l,:]
70                         if incl==inclines[2,:]:
71                             continue
72                         inclines[3,:]=incl
73                         if n == 2:
74                             x=inclines_to_coordinates(
    encoding,inclines,delta)
75                             if x == 0:
76                                 continue
77                             n_coordinates=zero_matrix(RR,n,2)
78                             for r in range(n):
79                                 n_coordinates[r,0]=x[2*r]
80                                 n_coordinates[r,1]=x[2*r+1]
81                             value=DrawGraph_filter_incl(
    encoding,n_coordinates,delta,min_len)
82                             if value == False:
83                                 counter+=1
84                                 continue
85                             coordinates=block_matrix([[
    m_coordinates],[n_coordinates]])
86                             list_of_coordinates.append(
    coordinates)
87                             continue
88                         for o in range(nofinclines):
89                             inclines[4,:]=set_of_inclines[k
    ,:]
90                             for p in range(nofinclines):
```

```
                                        incl=set_of_inclines[p,:]
                                        if incl==inclines[4,:]:
                                            continue
                                        inclines[5,:]=incl
                                        x=inclines_to_coordinates(
encoding,inclines,delta)
                                        n_coordinates=zero_matrix(RR,
n,2)
                                        for s in range(n):
                                            n_coordinates[s,0]=x[2*s]
                                            n_coordinates[s,1]=x[2*s
+1]
                                        value=DrawGraph_filter_incl(
encoding,n_coordinates,delta,min_len)
                                        if value == False:
                                            continue
                                        coordinates=block_matrix([[
m_coordinates],[n_coordinates]])
                                        list_of_coordinates.append(
coordinates)
        print(counter)

    else:
        for i in range(num_of_iterations):
            coordinates=DrawGraph_filter(encoding,delta,
min_len)
            list_of_coordinates.append(coordinates)
    print(len(list_of_coordinates))
    return list_of_coordinates
```

A.2.16 The Function Choosing Best Five Pictures

```
def choosing_best_five_pictures(encoding,delta,min_len,
    num_of_iterations,list_of_coordinates=0,
    list_of_unique_coordinates=0):
    r"""
    returns the five best coordinates after a certain number
    of applying DrawGraph_filter

    INPUT:

    - ''encoding'' - a vector of the form $[m,n,s,e_1,e_2
    ,...,e_2n]$

    - ''delta'' - a positive scalar

```

```
11     - ''min_len'' - a positive scalar
12 
13     - ''num_of_iterations'' - a positive integer
14 
15     OUTPUT:
16 
17     - a list of 5 coordinate systems
18 
19     """
20     m=encoding[0]
21     n=encoding[1]
22 
23     e_list=[]
24     for i in range(n):
25         e_list.append([encoding[2*i+3],encoding[2*i+4]])
26 
27 
28     if list_of_coordinates == 0:
29         list_of_coordinates=generate_list_of_coordinates(
    encoding,delta,min_len,num_of_iterations)
30     if list_of_unique_coordinates == 0:
31         list_of_unique_coordinates=[]
32         for i in range(num_of_iterations):
33             list_of_unique_coordinates.append(
    list_of_coordinates[i])
34 
35         length = len(list_of_coordinates)
36         for i in range(length):
37             coord_1=list_of_coordinates[i]
38 #           stop = False
39             for j in range(i+1,length):
40 #               if stop == True:
41 #                   break
42                 coord_2=list_of_coordinates[j]
43                 if coord_1==coord_2:
44                     list_of_unique_coordinates.remove(coord_1
    )
45                     break
46 
47 #                for k in range(m,m+n):
48 #                    if stop == True:
49 #                        break
50 #                    for l in range(k+1,m+n):
51 #                        if e_list[k-m] == e_list[l-m]:
52 #                            if coord_1[k] == coord_2[l] and
```

```
    coord_1[l] == coord_2[k]:
53 #                              if coord_1[m:k] == coord_2[m
    :k] and\
54 #                              coord_1[k+1:l] == coord_2[k
    +1:l] and\
55 #                              coord_1[l+1:m+n] == coord_2[
    l+1:m+n]:
56 #
    list_of_unique_coordinates.remove(coord_1)
57 #                                  stop = True
58 #                                  break
59
60              #still not completely valid for like 010101
61              #break
62      #else:
63      #    list_of_unique_coordinates=
    generate_list_of_coordinates(encoding,delta,min_len,
    num_of_iterations)
64
65      num_unique_coordinates=len(list_of_unique_coordinates)
66
67      score_list=[]
68      for i in range(num_unique_coordinates):
69          score=target_function(encoding,delta,
    list_of_unique_coordinates[i])
70          score_list.append([score,i])
71
72      score_list=sorted(score_list)
73      best_score_list=[]
74      best_five=[]
75      if num_unique_coordinates<10:
76          best_five=list_of_unique_coordinates
77      else:
78          for i in range(10):
79              best_score_list.append(score_list[i][0])
80              k=score_list[i][1]
81              best_five.append(list_of_unique_coordinates[k])
82
83      return best_five
```

A.2.17 Code to Sort All Possible Coordinates for the Wedge

```
1 A=generate_list_of_coordinates(encoding,delta,min_len,100000)
2 #attempting to find all the sets of coordinates
3
4 unique_list_of_coordinates=[]
```

```
5 #throwing all the duplicate coordinates away
6 for i in range(num_of_iterations):
7     x=A[i][2,0]
8     y=A[i][2,1]
9     if [x,y] not in unique_list_of_coordinates:
10        unique_list_of_coordinates.append([x,y])
11
12 def pyt_compt(unique_list_of_coordinates):
13     #pythagoras computer, to compute the hypothenuse of a
    rectangular triangle
14     k=len(unique_list_of_coordinates)
15     length_list=[]
16     for i in srange(k):
17         del_x_L=w[i][0]
18         del_x_R=50-w[i][0]
19
20         del_y=w[i][1]
21
22         len_L_sq=del_x_L^2+del_y^2
23         len_R_sq=del_x_R^2+del_y^2
24
25         len_L=sqrt(len_L_sq)
26         len_R=sqrt(len_R_sq)
27         length_list.append([len_L,len_R,w[i]])
28
29     return length_list
30
31 length_list=pyt_compt(unique_list_of_coordinates)
32 #the above gives a list of the lengths of all the edges
33 #the below gives the same list, but sorted from small to
    large
34 length_list_dup=sorted(length_list)
35
36 #we now remove all edges larger than 50
37 for i in range(len(length_list)):
38     len_L=length_list[i][0]
39     len_R=length_list[i][1]
40     if len_L>=50 or len_R>=50: #the >= can be changed in <=
    to find all the edges smaller than 50
41         length_list_dup.remove(length_list[i])
42
43 #calculate the average length of each pair of coordinates
    that has edges larger than 50
44 gem_len_list=[]
45 for i in range(len(length_list_dup)):
```

```
46     gem_len=(length_list_dup[i][0]+length_list_dup[i][1])/2
47     gem_len_list.append([gem_len,length_list_dup[i][2]])
48 sorted(gem_len_list)
49 
50 #the final output is a list that gives the average length of
       the edges of one set of coordinates
51 #and the corresponding coordinates for the inner vertex
```

A.2.18 The Mirroring Function

```
1  def mirroring_function(encoding,coordinates):
2      m=encoding[0]
3      n=encoding[1]
4  
5      new_encoding=[m,n,encoding[2]]
6      print(new_encoding)
7      for i in range(2*n):
8          e=encoding[3+i]
9          if e==1:
10             if i%2==1 and encoding[2+i]==0:
11                 new_encoding.append(1)
12             else:
13                 new_encoding.append(0)
14         if e==0:
15             if i%2==0 and encoding[4+i]==1:
16                 new_encoding.append(0)
17             else:
18                 new_encoding.append(1)
19         if e not in [0,1]:
20             new_encoding.append(e)
21         print(new_encoding)
22     new_coordinates=zero_matrix(RR,m+n,2)
23     for i in range(m):
24         new_coordinates[i,:]=coordinates[i,:]
25     for i in range(n):
26         x=coordinates[2+i,0]
27         new_x=50-x
28         new_coordinates[2+i,0]=new_x
29         new_coordinates[2+i,1]=coordinates[2+i,1]
30     return new_encoding,new_coordinates
```