Master of Science in Advanced Mathematics and Mathematical Engineering

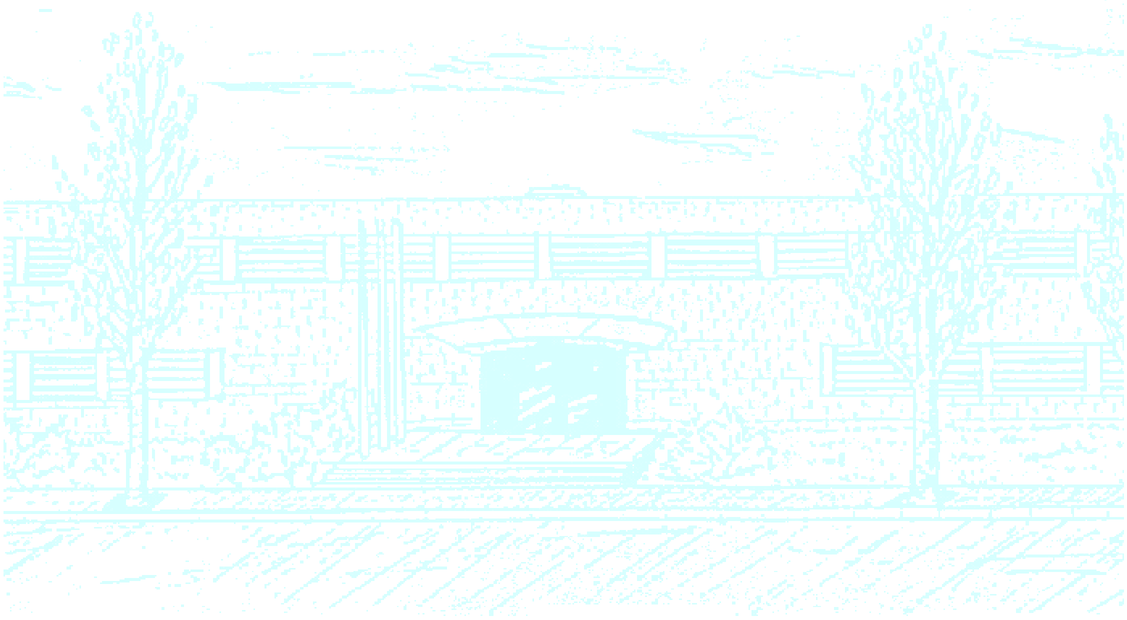
Title: Non-linear model for chip floorplanning

Author: Marta Arriaza Bosch

Advisor: Dr. Jordi Cortadella Fortuny

Department: Computer Science

Academic year: 2022-2023



UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Facultat de Matemàtiques i Estadística

Universitat Politècnica de Catalunya
Facultat de Matemàtiques i Estadística

Master in Advanced Mathematics and Mathematical Engineering
Master's thesis

Non-linear model for chip floorplanning

Marta Arriaza Bosch

Supervised by Dr. Jordi Cortadella Fortuny

June, 2023

I would like to thank my advisor, Jordi Cortadella, for his thesis proposal and all his guidance and supervision during the project. I would also like to show my gratitude to all my family and friends for always being by my side and cheering me up. A special thank you to Oscar for encouraging me throughout my studies.

Abstract

The design of the floor plan is an important step in the design process of integrated circuits. In this step, the modules are distributed along the die. A good floor plan can significantly increase the efficiency of the device.

Floorplanning is a non-convex optimisation problem. It can be solved at different levels of abstraction. The first level would be to represent the modules as points. Higher levels would be those in which we distribute the modules as circles or orthogonal polygons with a given area. In this thesis we try to solve the first step of this problem by finding an algorithm that returns good distributions of the modules by representing them as points. This algorithm will be based on two steps: the initial distribution of the points using existing methods and the reorganization of the modules to improve the uniformity of the distribution.

Keywords

Floorplanning, non-convex optimization, die(integrated circuit), spectral layout, wirelength

Contents

1. Introduction

Floorplanning is an optimisation problem in which a set of elements are distributed taking into account a specific criterion and restrictions. One of the areas in which this problem appears is in the first step of the process of integrated circuit design. In this, we define wirelength as the total length of the wires that are used to connect the components. A floor plan that distributes the modules in a way that minimises the wirelength will bring many benefits to improve the efficiency of the circuit. To begin with, taking into account that the wirelength is equivalent to the distance the signal has to travel, minimising the wirelength will minimise the delay of the signal. That is, the signal will take less time to reach its destination, so we get a more efficient communication between components. In addition, with shorter wirelength, we minimise the likelihood of signal degradation and noise. Finally, taking into account that the resistance in the wires is proportional to their length, if we reduce the wirelength, we reduce the power dissipation. Therefore, the total power consumption of the circuit will be lower the shorter the wirelength. Besides reducing the wirelength, we have to take into account another feature when designing an integrated circuit. A uniform distribution of the components along the die is also necessary for good performance. In an integrated circuit, power is consumed and resistance is produced, so heat is generated. If the modules are not evenly distributed, hotspots can occur, which are areas where the temperature is much higher than in the surrounding regions. As a result, the efficiency of the circuit can be negatively influenced. An optimal solution to the floorplanning problem, where wirelength and die blank spaces are minimised, can significantly increase efficiency.

Floorplanning is a complicated problem that we cannot solve with simple algorithms. It can be seen as a problem with different levels that we can solve step by step. As we can see in figure 1, first we can design the floor plan by representing the modules as points. Then, taking into account this solution, we can distribute the modules using areas with simple geometrical shapes like squares or circles. The final goal would be to find the arrangement of modules with different shapes in order to obtain a uniform distribution within the die while minimising the wirelength and the overlap between modules. In each of these steps, we need a mathematical algorithm that allows us to solve each of the specific optimisation problems.

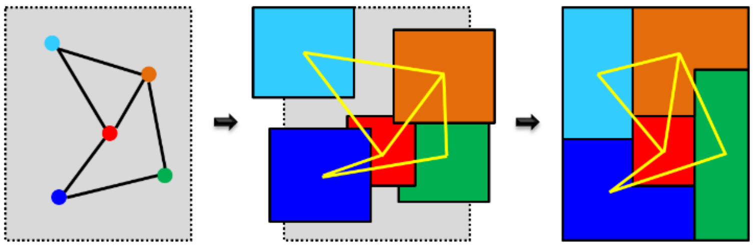


Figure 1: Levels of abstraction of the floorplanning process.

Many approaches have been made to solve the floorplanning problem. Naylor et al. described an approach to this problem in their patent (U.S.Pat.6301693) [1] in which they distributed a set of modules in a die while minimising the wirelength. Specifically, they present a model where they minimise a weighted sum in which they study different metrics, including wirelength and density. The density calculates how well distributed the modules are in the die. To compute it, a grid of control points both inside and outside the die was constructed. Using a physical analogy, they calculate a potential between each module and each point in the grid. Then, the variance of the total potential in each control point is added to the density term of the objective funciton. Similar models based on the idea of having density control points in a grid have been studied in other articles, such as [2] and [3] ,which we have also taken as a reference for this project.

In this thesis we rely on this idea not to solve the floorplanning problem in its totality but to find good solutions to the first level of the floorplanning process. That is, we will consider the modules as points and our aim will be to find the best possible solution within the die. In the following, we show a non-linear non-convex optimisation problem that, starting from good initial solutions, obtains a new solution with a more uniform distribution. Our model minimises a weigthed sum objective function that includes three terms: the wirelength, the uniformity of distribution of the modules in the area of the die and a summation of all the repulsive forces between modules that prevent them from accumulating. The distribution term is calculated in a very similar way to that used by Naylor et al., by constructing a grid of control points and calculating the variance of a potential using a bell-shaped function. It is important to remark that as the optimisation problem is non-convex, the initial configuration will influence the result. In our case, we will start with an initial configuration with good relative positions of the modules obtained from spectral methods, based on the computation of eigenvalues and eigenvectors, and force-direction based methods.

The project will be organised as follows: In section two, we present the non-convex floorplanning optimisation problem by simplifying the modules as points. In section three, we show the first step of our algorithm, where we use different methods to draw the initial graphs. In section four, we show the second step, where we evenly distribute our points along the die. In section five, we describe the metrics used to compare the results. In section six, we show the results obtained and their analysis. Finally, in section seven, we present our conclusions and future work on this topic.

2. Problem formulation

The problem consists of, given an initial distribution of the modules on the die obtained using different methods of graph drawing, finding a more uniform arrangement while minimising the wirelength. The methods we will use to obtain the initial configuration minimise the wirelength subject to certain restrictions. It is possible that when looking for a new solution with a more uniform distribution of the modules along the die, the wirelength will increase. However, we have to keep in mind that for many applications uniformity of the distribution in exchange for increased wirelength may be desirable. Therefore, our goal is to find a solution that optimises both characteristics: wirelength and a uniform distribution.

To simplify the problem, we will represent the integrated circuit with a graph. The nodes symbolise the modules while the connections through the wire are represented by the edges. Our objective is to generate an algorithm that distributes the modules along the die. Firstly, we will obtain an initial graph using already existing methods. Then, we will create a function that has as input the initial positions of the nodes and the edges between them, providing as output the optimal final location of the nodes. To do so, we generate a model that solves the following optimisation problem: given a fixed-die of dimensions L_x and L_y find the location of the modules such that it minimises the length of the wirelength maintaining a uniform distribution.

Consider x_i and y_i as our optimisation variables. These will represent the coordinates of the positions of the two-dimensional graph for each node i. Then, we can formulate the problem as follows:

$$\begin{aligned} \min_{x_i, y_i} \quad & \beta \cdot WL + P \\ \text{s.t.} \quad & 0 \leq x_i \leq L_x \\ & 0 \leq y_i \leq L_y \\ \text{for} \quad & i = 1, 2, ..., n \end{aligned} \tag{1}$$

where n is the number of modules and β is a parameter to determine the weight we give to the wirelength (WL) term with respect to the penalty distribution term (P). P increases when the uniformity in the distribution decreases. By minimising its value, we find a more uniform distribution. On the other hand, the constraints show that all modules must be within the die area.

To give more flexibility to the program, we have decided to add another term (F_R), which will be the minimisation of repulsive forces. These help to avoid configurations in which some modules are clustered at the same point. The final problem we are trying to minimise is formulated as follows:

$$\begin{aligned} \min_{x_i, y_i} \quad & \beta \cdot WL + P + \gamma \cdot F_R \\ \text{s.t.} \quad & 0 \leq x_i \leq L_x \\ & 0 \leq y_i \leq L_y \\ \text{for} \quad & i = 1, 2, ..., n \end{aligned} \tag{2}$$

where β and γ are the parameters representing the weights in the objective function of the total wirelength and the sum of repulsion forces respectively.

This optimisation problem is non-convex, i.e. its objective function is non-convex. This means that the function may contain several local minima and maxima. This makes these problems more difficult to solve compared to convex problems which have a single global minimum or maximum. For non-convex problems in general we need specialised algorithms to handle the complexity of the objective function. The most commonly used algorithms for these are gradient-based algorithms, heuristic methods, branch and bound, etc.

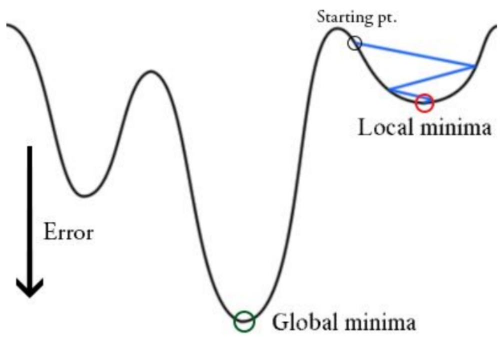


Figure 2: Representation of the performance of a non-convex optimisation problem.

In our case we will use the first group of algorithms, those based on the gradient. These methods use the gradient of the objective function to guide the solution of the problem. The gradient represents the direction of steepest ascent at each point. Specifically, in our optimisation problem, since we want to find the minimum of the function, we will use the negative gradient to find the direction of the local minimum. These methods are iterative methods in which they update the solution using the gradient calculated at each step. It is for this reason that in this project it is also important to select well the initial configuration of the modules, as this will guide us towards different final distributions.

3. Algorithms for drawing graphs

The first step in our algorithm is to find a good initial distribution of the nodes in the module's graph such that it can lead us to find the best possible final distribution when solving the non-convex optimisation problem. In this project, we have studied different methods to draw our initial graph. In particular we have used three methods which are implemented in the NetworkX package, dedicated to generate graphs. These methods are the following ones:

- Spectral layout: it calculates the eigenvectors and eigenvalues of the graph's Laplacian matrix to obtain the final coordinates of the graph's nodes.
- Spring layout: based on the calculation of attractive forces between adjacent nodes and repulsive forces between pairs of nodes in order to find an equilibrium state.
- Kamada-Kawai layout: it considers the edges of the graph as strings with their proper length and tries to minimise the energy of the system in such a way that these strings are lengthened or contracted as little as possible.

In the following, we explain in more detail how each of these methods works to obtain the final drawing of the graph. In addition, once all the algorithms have been discussed, we give an example where it can be seen the comparison between the three methods applied to two random graphs.

3.1 Spectral layout

The spectral method [4] uses the properties of the Laplacian matrix to draw the graph. Consider a graph G=(V,E) with m vertices. For each vertex $v_i \in V$ we assign a point in the plane $\rho(v_i) \in \mathbb{R}^n$. To draw the graph, we trace a segment between two points ($\rho(v_i)$ and $\rho(v_j)$) if and only if there is an edge between v_i and v_j in E. We define the matrix of the graph drawing R as the matrix that has in the i-th row the row vector $\rho(v_i)$.

Consider now a physical analogy, regarding the edges as identical strings. What we want to achieve with this model is to minimise the length of these strings. We can describe this as the drawing energy of G:

$$\mathcal{E}(R) = \sum_{v_i, v_j \in E} ||\rho(v_i) - \rho(v_j)||^2 \tag{3}$$

In other words, we minimise the square of the distance between adjacent vertices of the graph G.

Definition 3.1. Given a graph G=(V,E) with n nodes, its Laplacian matrix $L := (l_{i,j})_{nxn}$ is described element by element as follows:

$$l_{i,j} = \begin{cases} \text{degree}(i) & \text{if} \quad i = j \\ -1 & \text{if} \quad i \neq j \text{ and } v_i \text{ is adjacent to } v_j \\ 0 & \text{otherwise} \end{cases} \tag{4}$$

Equivalently, the Laplacian matrix can also be given by the expression $L = D - A$ where D is the diagonal degree matrix and A is the adjacency matrix.

The energy of the graph drawing can be reformulated as a function of the Laplacian matrix using proposition 3.2.

Proposition 3.2. *Given a graph G=(V,E), R its matrix of the graph drawing and L the Laplacian matrix of G, the energy of the graph can be calculated with the following formula:*

$$\mathcal{E}(R) = tr(R^T LR) \tag{5}$$

Proof: We denote R^k as the kth column of R. Then:

$$\begin{aligned}
\mathcal{E}(R) &= \sum_{\{v_i,v_j\}\in E} ||(v_i) - (v_j)||^2 \\
&= \sum_{k=1}^{n} \sum_{\{v_i,v_j\}\in E} (R_{ik} - R_{jk})^2 \\
&= \sum_{k=1}^{n} \frac{1}{2} \sum_{i,j=1}^{m} (R_{ik} - R_{jk})^2 \\
\sum_{k=1}^{n} (R^k)^T LR^k &= tr(R^T LR)
\end{aligned}$$

□

Obviously, there is a trivial solution to this problem which would correspond to locating all nodes at the same coordinates. Therefore, we have to impose at least one restriction in order to avoid the trivial solution. We suppose that the columns of R are pairwise orthogonal and have unit length, so we include the constraint $R^T R = I$ which avoids obtaining the trivial solution.

Theorem 3.3. *Given a graph G=(V,E) and its Laplacian matrix L with eigenvalues $0=\lambda_1 < \lambda_2 \leq ... \leq \lambda_m$, the minimum energy $\mathcal{E}$ corresponding to the drawing of the graph in $\mathbb{R}^n$ is equal to $\lambda_2 + ... + \lambda_{n+1}$, satisfying the condition $R^T R = I$.*

The proof of this theorem was given by Godsil and Royle and can be found in reference [5] (Section 13.4, Theorem 13.4.1).

Applying this theorem, and assuming that $\lambda_2 > 0$, the spectral method for drawing two-dimensional graphs is constructed as follows: First we compute the two smallest eigenvalues with value greater than 0 of the Laplacian of the graph G. Then, we compute the eigenvectors associated to these eigenvalues and construct the matrix R $=[u_2, u_3]$ with the two eigevectors in its columns. Finally the vertex v_i is placed in the coordinates given by the i-th row of R. This configuration achieves a minimum drawing energy where the length of the edges is minimised by satisfying that $R^T R = I$.

3.2 Spring layout

The Spring algorithm [6] is a force-directed graph drawing method that uses a physical analogy to distribute the nodes. It represents the edges as strings that tend to pull adjacent nodes together. On the other hand, the nodes repel each other with an electric force as if they were equally charged bodies. The aim is to find an equilibrium position in the system.

Consider that we have a graph G=(V,E). For each pair of vertices $v_i, v_j \in V$ with position in the 2-dimensional space represented by the vectors $\vec{v_i}, \vec{v_j} \in \mathbb{R}^2$ we can calculate the Euclidean distance between them:

$$||\vec{v_i} - \vec{v_j}|| = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2} \tag{6}$$

where (x_i, y_i) and (x_j, y_j) are the components of the vectors $\vec{v_i}$ and $\vec{v_j}$ respectively.

If the pair of vertices is adjacent, we can calculate the force generated by the string (the edge) following the form of Hooke's law:

$$\vec{F}_{attraction}(v_i, v_j) = -k_a(||\vec{v_j} - \vec{v_i}|| - l) \cdot \vec{e_{ij}} \tag{7}$$

where k_a and l are proper constants of the string and $\vec{e_{ij}}$ is the unit vector obtained from the formula $\vec{e_{ij}} = \frac{\vec{v_j} - \vec{v_i}}{||\vec{v_j} - \vec{v_i}||}$.

On the other hand, we can calculate the repulsive force using the analogy of Coulomb's law:

$$\vec{F}_{repulsion}(v_i, v_j) = \frac{k_r}{||\vec{v_j} - \vec{v_i}||^2} \cdot \vec{e_{ij}} \tag{8}$$

where k_r is a constant that determines the strength of the repulsive force.

Finally, we can calculate the total force acting on each node by adding up all the force contributions at that node:

$$\vec{F}(v_i) = \sum_{(v_i, v_j) \in E} \vec{F}_{attraction}(v_i, v_j) + \sum_{v_j \in V} \vec{F}_{repulsion}(v_i, v_j) \tag{9}$$

The algorithm calculates the force applied to each node and iteratively updates its position in the direction of the force until an equilibrium configuration is obtained. That is, until the vertices experience no force or the force is smaller than the tolerance. In practice, the algorithm is often stopped after a maximum number of iterations.

3.3 Kamada-Kawai layout

The algorithm proposed by Tomihisa Kamada and Satoru Kawai [7] has a similar basis to the Springs method, which uses a physical analogy to distribute the nodes of a graph. As in the springs method, we consider the edges of the graph as strings attached to particles that are represented by the nodes. The objective in the Kamada-Kawai method is to find a configuration in which the energy is minimised. That is, we want to find the state in which the strings have the original length, so we want to minimise the compression or elongation of the string from its relaxed state. That is to say, we want to minimise the following energy:

$$\mathcal{E} = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{1}{2} k_{ij} (|v_i - v_j| - l_{ij})^2 \tag{10}$$

where n is the number of nodes in the graph, $|v_i - v_j|$ corresponds to the Euclidean distance between vertices v_i and v_j, l_{ij} is the original length of the string and k_{ij} is an inherent constant of the string.

Using a non-linear optimisation method such as gradient descent we can find the values of v_i for which the energy is minimum. In this way we obtain the distribution of the nodes of the graph using the Kamada-Kawai method.

3.4 Comparision between Spectral, Spring and Kamada-Kawai layouts

Consider two Erdős-Rényi graphs: the first one with 10 nodes and probability 0.3 and the second one with 15 nodes and probability 0.2. Using the respective NetworkX graph drawing functions (spectral_layout, spring_layout and kamada_kawai_layout) we have determined the positions of the nodes. The results in 3, show different drawings for the same graph depending on the method used. Although some of these methods manage to distribute the nodes more or less evenly, especially the Kamada-Kawai method, in all three cases the uniformity of the spreading can still be improved. To obtain a uniform distribution from these initial solutions, will be the goal of the next step in our algorithm.

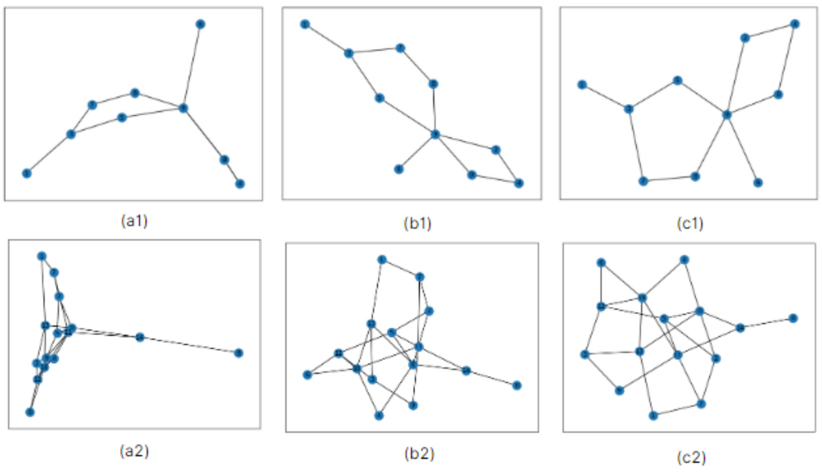


Figure 3: Comparison between the Spectral ((a1), (a2)), Spring ((b1), (b2)) and Kamada-Kawai ((c1),(c2)) methods using two Erdős-Rényi graphs.

4. Optimisation of distribution uniformity

Once we have obtained an initial configuration for our problem, the second step in our algorithm is to solve an optimisation problem. Starting from our initial solution, it will distribute the nodes inside the die in a uniform way, trying to keep the wirelength as short as possible. To do this, we have defined the optimisation problem in 2, where our optimisation variables are the coordinates of the module positions. In this problem the only restrictions we have are on the range of our variables, given by the dimensions of the die. The objective function is a weighted sum that evaluates three elements: the wirelength (WL), the distribution of the modules with respect to points on a control grid (P) and the repulsion forces between modules (F_R). Let us recall the formulation of the optimisation problem:

$$
\begin{aligned}
\min_{x_i, y_i} \quad & \beta \cdot WL + P + \gamma \cdot F_R \\
\text{s.t.} \quad & 0 \leq x_i \leq L_x \\
& 0 \leq y_i \leq L_y \\
\text{for} \quad & i = 1, 2, \ldots, n
\end{aligned}
$$

where β and γ are the weights of the total wirelength and repulsion forces respectively. We will now discuss in detail how each of the terms that appear in this function are described and calculated.

4.1 Wirelength

The term that minimises the wirelength is calculated from the square of the Euclidean distance. In this way we use a quadratic formula that is easy to derive. To do so, we first enumerate all the modules. Then, we calculate the square of the length of the edges connecting each module with modules of larger indices. Finally, we sum the results obtained for all the modules into a single term WL.

$$
wl_{ij} = \begin{cases} (x_i - x_j)^2 + (y_i - y_j)^2 & \text{if} \quad i, j \in E \\ 0 & \quad \text{otherwise} \end{cases} \tag{11}
$$

$$
WL = \sum_i \sum_{j>i} wl_{ij} \tag{12}
$$

If we do not include any other term or constraint in the objective function, the model will tend to place all the modules clustered at the same point in the die, which would make routing impossible. Therefore, we have to include a term that distributes the points along the die.

4.2 Penalty on non-uniform distributions

The second term is the penalty on non-uniform distributions. For its calculation we use a method similar to the one used in [2]. The main idea is to consider the nodes as physical particles. Then, we distribute a set of control points on the die and calculate the potential that each node produces on each of these points. The goal would be to ensure that this potential is uniform throughout the die.

To do this, we provide the nodes with an area of influence. In our case we have used an equal square area of influence for all the modules. This area is not the actual area of the modules, it will only be used to distribute the nodes. Remember that we are only studying the first step of the floorplanning problem, thus our objetive is to distribute the modules as points. Therefore, the actual area of the modules does not have any influence in our problem. These influence areas define the area where the potential created by each node has a substantial contribution. In order to control the uniformity, we construct a grid of control points that are placed equi-spaced along the die, so that any module within the die will be controlled by at least one of these points.

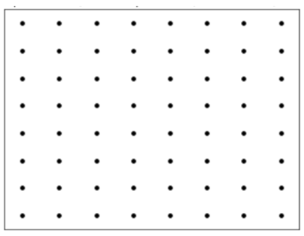


Figure 4: Grid of control points within the die.

We will then calculate the total contribution of the modules at each control point and ideally look for a distribution where all control points have an equal value. The penalty term will then be calculated as a sum of variances:

$$P = \sum_{i=1}^{m} (Potential_i - Potential_{uniform})^2 \tag{13}$$

where m is the number of control points in the die. $Potential_{uniform}$ would be the ideal value of potential, which should be equal in all points in the grid assuming a uniform distribution. The formulation of the potential that each module produces to the control point i ($Potential_i$) is not trivial. We have to keep in mind that to solve our optimisation problem we are using gradient-based methods which means that we need our objective function to be smooth. Therefore, we need to find an expression for this term such that the resulting function is differentiable at each point and the gradient can then be calculated.

The potential at each control point ($Potential_i$) will be computed as the sum of all module's contributions. The contribution of each module to each control point, will be calculated as a function based on the distance between these two points. We describe this potential as a continuous bell-shaped function that is differentiable at all points, so that we have no problems when applying gradient methods. Specifically, we use a normal distribution with maximum value when the dsitance between the centre of the module and the grid point in zero. The potential will have a value for all points within the die and its value will decrease as the distance between module and control point increases.

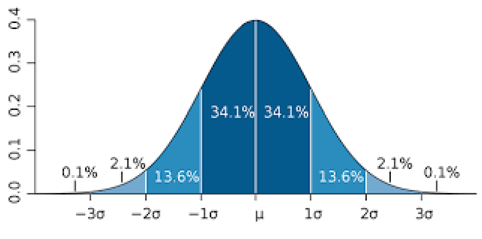


Figure 5: Gaussian function

In practice, we divide the potential in two, Px_j and Py_j, which are respectively the potentials depending on the coordinates x and y for the module j. The total potential for module j (Pot_j) is obtained from the product of these two potentials, as we can see below.

$$Pot_j(x, y) = \xi \cdot Px_j \cdot Py_j \tag{14}$$

$$Px_j(x) = e^{-\frac{(x-x_j)^2}{2\sigma^2}} \tag{15}$$

$$Py_j(y) = e^{-\frac{(y-x_j)^2}{2\sigma^2}} \tag{16}$$

where ξ is a constant that allows the magnitude of the total module contribution in the die to be preserved. That is, the area under the Gaussian function will remain the same as the influence area assigned to the module. We have considered the mean of the Gaussian function equal to zero, since we want the largest contribution made by the module to be at its centre. And last, we have chosen $\sigma = w_j/3 = h_j/3$. As we can see in the Figure 5, for values of the distance greater than 3σ, the value of the Gaussian is very small. That is why we choose this value for σ, because that way we impose that the potential it creates outside its influence area is negligible.

Potential functions accumulate inside the die. To determine the total potential falling on each control point, we sum the potential contributions of all the modules at that location. If the modules are in a good distribution, the sum of the potentials at each control point will be the same or very close. Using expressions 14, 15 and 16 we can compute the potential controlled by the grid point i as follows:

$$Potential_i = \sum_{j=1}^{n} Pot_j(x_i, y_i) \tag{17}$$

Consequently, we can conclude that in the best case, where the distribution is perfectly uniform, the potential will have the value $Potential_{uniform}$ = *sum of the total influence areas of the modules/number of control points.*

4.3 Repulsion forces

The last term in the objective function helps to maintain a good distribution of the modules by avoiding clusters of modules in the same region. It represents repulsion forces between all pairs of modules. This force follows the idea of the Spring method. It simulates that the nodes are particles with the same charge and between them there is a force that we can calculate with Coloumb's formula. Therefore, the total sum of repulsion forces between modules can be calculated using the following formula:

$$F_R(x_i, y_i, x_j, y_j) = \sum_i \sum_{j>i} \frac{k}{(x_i - x_j)^2 + (y_i - y_j)^2} \tag{18}$$

where k is a normalisation constant. By incorporating this term into the objective function, we try to minimise the repulsion forces, so we try to keep the nodes a certain distance apart to avoid clustering.

By changing the values of the weights in the objective function we obtain different distributions. Depending on the values we choose, we tend to minimise the edges better, to improve the distribution or to avoid clusters. Our aim is to find values for these weights such that we obtain satisfactory solutions for both the length of the edges and the distribution. To make the process of choosing the weights easier, we normalise all the terms in the objective function using the values of these terms in the initial configuration. This step is not necessary, but it is helpful for the implementation.

4.4 Implementation

To solve the optimisation problem we have used the GEKKO Python package which allows us to solve non-linear optimisation problems. Specifically, we used the APOPT solver which uses a gradient-based method to find the solution. To define the initial solution we used Spectral, Spring and Kamada-Kawai methods implemented in NetworkX package. The nonlinear optimisation process finds an assignment of values to the function variables, i.e. the module positions, such that the objective function is minimised. The solver uses this objective function and its gradient to determine the position of each module in the next iteration. The program is executed and stops when it obtains a solution with the desired tolerance or the number of iterations exceeds a given number of maximum iterations. If the program finds the minimum of the objective function, this solution must correspond to a good distribution of the modules within the die.

In appendix A you can find the Pyhton program where we implemented our model and from where we obtained the results.

5. Performance metrics

Once the model has been designed, we need to determine some metrics in order to compare the results obtained with the other methods of drawing graphs described in section 3. For this purpose, we have defined three metrics that measure different characteristics of the resulting graphs.

5.1 Total wirelength

The first metric (M_{WL}) is simply the sum of the length of the graph edges. That is, we calculate the Euclidean distance between adjacent nodes. We can derive the formula from equation 11:

$$wl_{ij} = \begin{cases} (x_i - x_j)^2 + (y_i - y_j)^2 & \text{if} \quad \{i, j\} \in E \\ 0 & \text{otherwise} \end{cases}$$

$$M_{WL} = \sum_i \sum_{j>i} \sqrt{wl_{ij}} \tag{19}$$

Our objective is to minimise the wirelength, so the smaller this value is, the better the result will be considering this metric. But as we have already explained, minimising the wirelength is not our only objective, since in that case the best solution we could find would be the trivial solution of placing all the modules at the same point. A good distribution will be one that minimises the wirelength but at the same time distributes the modules evenly along the die. Therefore, we need to define other metrics to calculate how good our distribution is.

5.2 Distribution uniformity metric

The second metric (M_D) calculates how uniformly the modules are placed in the die. To do this, we set up control regions of equal area. We decide that the number of regions should be similar to the number of modules. Therefore, we create a function with input the number of modules that returns two integers whose product is as similar as possible to the number of modules. These two numbers will give us the number of regions per row and column within the die. Once this grid of control regions is established, we count the number of modules that fall within each region (n_R). Ideally, this value should be equal across all regions and, assuming that the number of modules is divisible by the number of regions, it would have a value of $n_{uniform}$=*number of modules/number of regions*. The metric value is equal to the sum in each region of variances of n_R with respect to the ideal value $n_{uniform}$.

$$M_D = \sum_R (n_R - n_{uniform})^2 \tag{20}$$

The smaller the value of M_D, the better distributed the modules will be within the die.

5.3 Crossing edges

The last metric (M_{CE}) is given by the number of crossing edges in the graph. Ideally, in a good distribution, we would have no crossing edges, if possible, or as few as possible. Even so, it is quite likely that when trying to minimise the length of the edges and, at the same time, achieving a uniform distribution, crossing edges will appear. To calculate the number of crossing edges in the final configuration, we use geometric properties.

First it is necessary to define the orientation of a set of three points. Consider the triplet of points (a,b,c). Their orientation can be calculated from the slope of the line segments (a,b) and (b,c). We can define the slope of these segments with the following formulas:

$$\theta = (b_y - a_y)/(b_x - a_x) \tag{21}$$
$$\tau = (c_y - b_y)/(c_x - b_x) \tag{22}$$

where the sub-indices x and y indicate the two-dimensional coordinates of each point.

Using this, it is easy to see that if the two slopes are equal, the points are collinear. If the slope of the first segment is greater than the slope of the second, the points will have a clockwise orientation, whereas if the slope of the first segment is less than the slope of the second, the orientation will be counterclockwise.

Using the expressions for θ and τ, in 21 and 22 respectively, it is trivial to see that the orientation of the three points will depend on the sign of the following expression:

$$\eta = (b_y - a_y) \cdot (c_x - b_x) - (c_y - b_y) \cdot (b_x - a_x) \tag{23}$$

Then, if $\eta = 0$ the points are collinear, if $\eta > 0$ the points have a clockwise orientation and if $\eta < 0$ the points have a counterclockwise orientation.

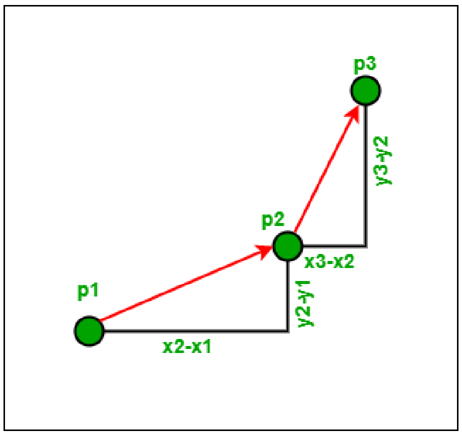
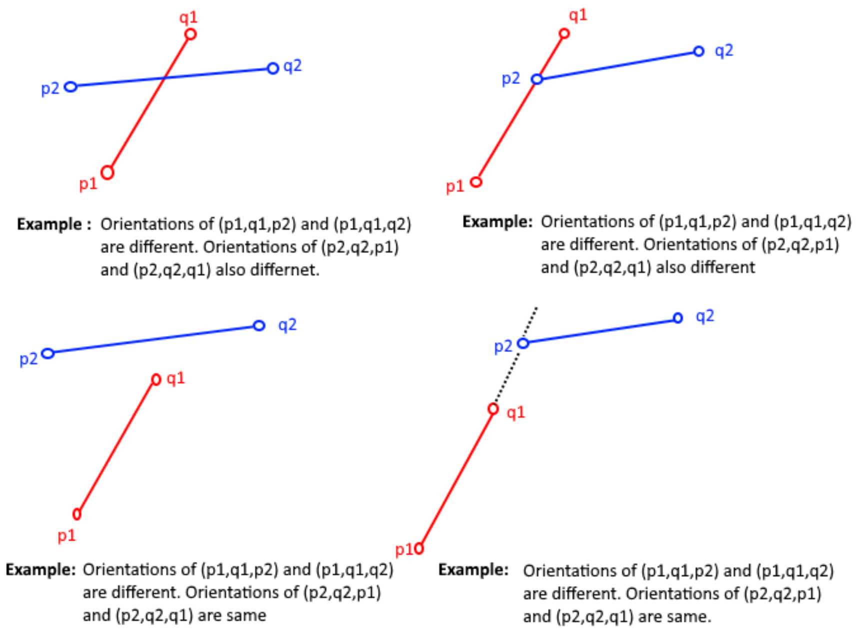


Figure 6: Determination of the orientation of three ordered points using the slope.

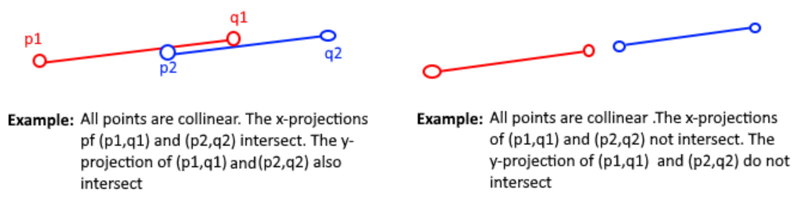
Let us now consider that we want to calculate the crossing edges of a graph G=(V,E). For each pair of edges $u, v \in E$ we determine the adjacent vertices corresponding to each of them: $u = (p_1, q_1)$ and $v = (p_2, q_2)$. To determine whether u and v intersect we need to calculate the orientations of the triplets: $o_1 = (p_1, q_1, p_2)$, $o_2 = (p_1, q_1, q_2)$, $o_3 = (p_2, q_2, p_1)$ and $o_4 = (p_2, q_2, q_1)$. We can say that if $o_1 \neq o_2$ and $o_3 \neq o_4$, then the two edges cross.



q1
p2
q2
p1
Example : Orientations of (p1,q1,p2) and (p1,q1,q2) are different. Orientations of (p2,q2,p1) and (p2,q2,q1) also differnet.
q1
p2
q2
p1
Example: Orientations of (p1,q1,p2) and (p1,q1,q2) are different. Orientations of (p2,q2,p1) and (p2,q2,q1) also different
p2
q2
q1
p1
Example: Orientations of (p1,q1,p2) and (p1,q1,q2) are different. Orientations of (p2,q2,p1) and (p2,q2,q1) are same
p2
q2
q1
p1
Example: Orientations of (p1,q1,p2) and (p1,q1,q2) are different. Orientations of (p2,q2,p1) and (p2,q2,q1) are same.

Figure 7: Determination of whether two edges cross from the orientation of three vertices. General case.

In the case that this condition is not fulfilled, there can be an instance in which the edges do cross. This case is if the points are collinear but one of the points of a segment lies on the other segment. That is why, we have to study the coordinates of the vertices for those cases where the points are collinear. In any other case, the edges will not cross each other.



Example: All points are collinear. The x-projections pf (p1,q1) and (p2,q2) intersect. The y-projection of (p1,q1) and(p2,q2) also intersect

Example: All points are collinear .The x-projections of (p1,q1) and (p2,q2) not intersect. The y-projection of (p1,q1) and (p2,q2) do not intersect

Figure 8: Determination of whether two edges cross from the orientation of three vertices. Special case.

To calculate the metric M_{CE} we look in a loop for each pair of edges if they cross each other and at each iteration we update a counter that adds one by one the number of crossing edges. That is, M_{CE} will have a value equal to the number of crossing edges in the final distribution. A better arrangement will be the one with M_{CE} as small as possible.

Therefore, as we have seen, to compare the final configurations with the initial distributions obtained from the Spectral, Spring and Kamada-Kawai methods, we use the three metrics M_{WL}, M_D and M_{CE}. Our aim is to find configurations for which the value of the three metrics is the smallest possible. It is very likely that for different configurations when we decrease the value in one metric, we will increase the value in another. We want to find distributions that find a balance between the three metrics.

6. Experimental results

To test the performance of our model, we have implemented a Python program using NetworkX and Gekko libraries. NetwrokX has been used for the graphical representation of the graphs and to obtain the initial distributions using the methods explained in section 3. Gekko has been used to solve the optimisation problem described in 4. We have run the program using different types of graphs. We have visualised the results graphically and we have also computed the values of the metrics described in 5. The results have been analysed in terms of the weights of the objective function β and γ.

6.1 Path graph

The first study case was a path graph with 20 modules. For this we fixed some parameters of the problem: the number of control points to 64, the area of influence of the modules to 1x1 and the dimension of the die to 4x4. Remember that the area of influence is not the same as the actual area of the module, since we only work with points. Therefore, it is not incoherent to distribute 20 modules with 1x1 area of influence inside a 4x4 die, because these areas can overlap without any problem.

The first step of the algorithm was to draw the graph using each of the existing methods explained in section 3. For this, we used the following functions from the NetworkX package: *spectral_layout*, *spring_layout* and *kamada_kawai_layout*. The obtained results have been given as input to our optimisation model. In it, we use GEKKO's APOPT solver, based on gradients, to solve the optimisation problem described above in section 4. As output we obtain the coordinates of the points in the new distribution which minimises the objective function. Figure 9 shows the initial configuration (in blue) given by the three different methods used to draw the graph initially. We also show some of the solutions (in red) found by initialising our model with the corresponding graph and using different values of β and γ to solve the optimisation problem.

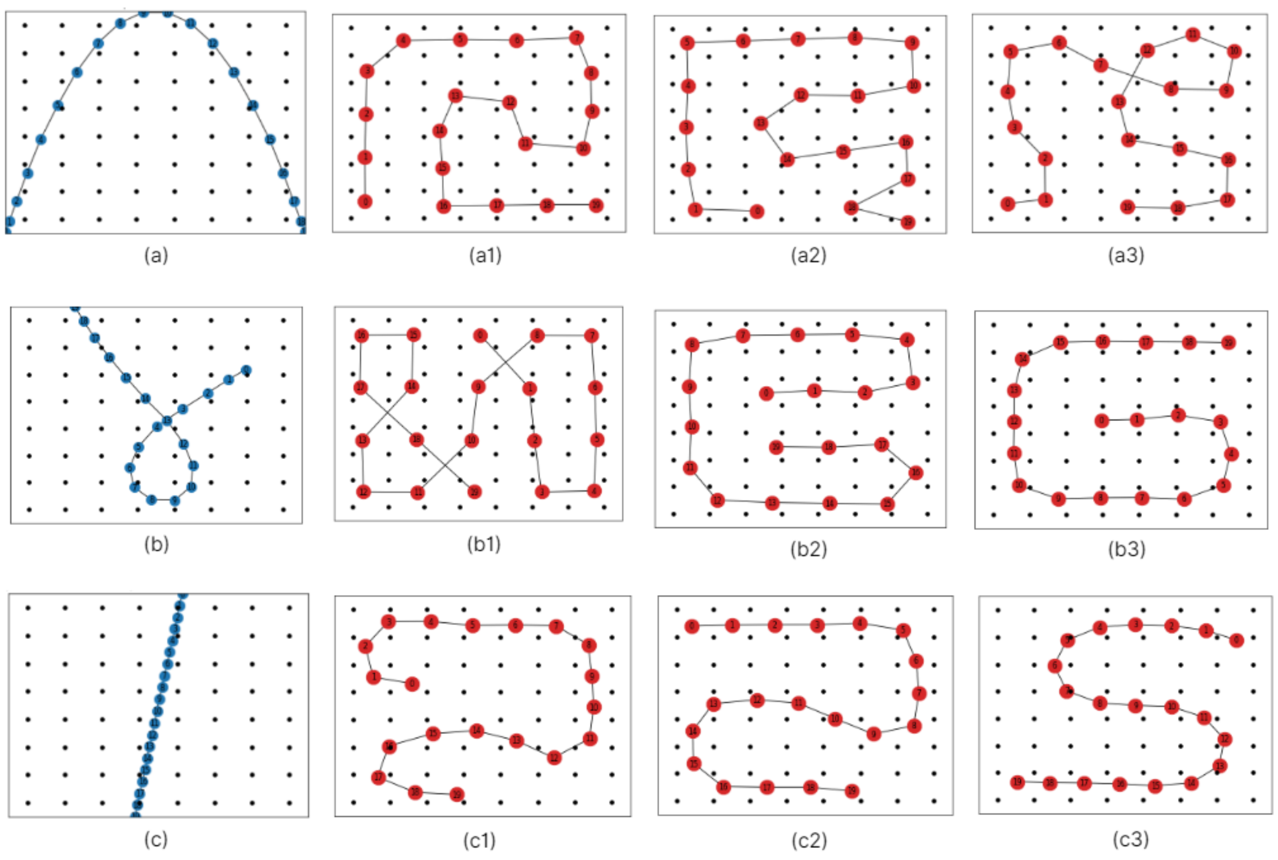


Figure 9: Results for a path graph of 20 modules and 64 control points. (a),(b),(c): Initial configurations obtained from Spectral, Spring and Kamada-Kawai methods respectively. The distributions obtained using our model are represented in red. First row corresponds to solutions that used Spectral configuration as initial configuration, second row used Spring and third one, Kamada-Kawai. The parameters used are the following: (a1) $\beta = 0.1$, $\gamma = 0.2$, (a2) $\beta = 0.1$, $\gamma = 0.5$, (a3) $\beta = 0.2$, $\gamma = 0.4$, (b1) $\beta = 0.05$, $\gamma = 0.4$, (b2) $\beta = 0.1$, $\gamma = 0.3$, (b3) $\beta = 0.2$, $\gamma = 0.1$, (c1) $\beta = 0.05$, $\gamma = 0.2$, (c2) $\beta = 0.05$, $\gamma = 0.4$, (c3) $\beta = 0.1$, $\gamma = 0.4$.

At a glance, we can see how the distribution is more uniform in the solutions obtained from our model (red graphs) as opposed to the initial solutions (blue graphs). However, in order to measure these changes more rigorously and to study how the parameters β and γ affect the final result, we calculate the metrics described in section 5. Below, we show table 1 with the values of the metrics for the results found for different values of the weights of the objective function. In the upper part of the table we show the values of the metrics for the initial configurations. In the table, the results that have been represented in figure 9 are highlighted in grey.

Path graph: 20 modules and 64 control points										
Initial distribution										
		Spectral			Spring			Kamada-Kawai		
		M_{WL}	M_D	M_{CE}	M_{WL}	M_D	M_{CE}	M_{WL}	M_D	M_{CE}
		9.3	22	0	7.6	42	1	4.1	80	0
Results using our algorithm										
Parameters		Initialising w/ Spectral			Initialising w/ Spring			Initialising w/ Kamada-Kawai		
β	γ	M_{WL}	M_D	M_{CE}	M_{WL}	M_D	M_{CE}	M_{WL}	M_D	M_{CE}
0.05	0.1	17.2	8	2	17.1	4	2	9.5	14	0
0.05	0.2	16.2	6	1	19.3	8	5	9.8	12	0
0.05	0.3	17.1	4	1	20.2	4	5	11.6	8	1
0.05	0.4	15.7	6	0	19.2	0	3	10.9	8	0
0.05	0.5	17.7	4	2	20.7	4	5	11.1	8	0
0.1	0.1	15.0	8	2	15.5	10	2	8.1	14	0
0.1	0.2	14.1	6	0	13.9	6	1	8.5	22	0
0.1	0.3	14.5	4	0	14.0	6	0	9.6	30	1
0.1	0.4	17.1	4	3	16.0	10	2	9.3	12	0
0.1	0.5	15.0	4	0	14.6	10	1	9.2	14	0
0.2	0.1	12.9	10	1	11.0	8	0	6.3	28	0
0.2	0.2	11.9	10	0	11.1	12	0	6.8	26	0
0.2	0.3	12.1	10	0	11.5	8	0	7.2	26	0
0.2	0.4	13.8	2	1	11.9	10	0	7.5	24	0
0.2	0.5	12.8	6	0	11.8	8	0	8.3	14	1
0.3	0.1	10.3	12	0	10.9	18	1	5.6	22	0
0.3	0.2	10.9	12	0	11.1	16	1	5.9	36	0
0.3	0.3	12.2	10	1	10.2	12	0	6.4	22	0
0.3	0.4	12.0	14	1	10.7	12	0	6.8	26	0
0.3	0.5	11.6	10	0	12.6	8	2	6.9	28	0
0.4	0.1	9.5	16	0	8.8	18	0	5.1	44	0
0.4	0.2	11.0	14	1	9.5	14	0	5.6	34	0
0.4	0.3	10.1	12	0	10.5	14	1	5.9	40	0
0.4	0.4	10.4	12	0	10.7	20	1	6.6	44	1
0.4	0.5	10.8	14	0	10.1	12	0	6.5	26	0
0.5	0.1	9.8	18	1	8.3	14	0	4.8	44	0
0.5	0.2	9.2	16	0	8.6	16	0	5.2	44	0
0.5	0.3	9.6	14	0	9.8	16	1	5.5	40	0
0.5	0.4	9.6	12	0	10.1	22	1	5.8	34	0
0.5	0.5	10.4	12	0	9.5	16	0	6.1	26	0

Table 1: Values of the metrics M_{WL}, M_D and M_{CE} for the initial configurations and the results obtained using our model in a path graph of 20 modules, for different values of parameters β and γ.

We can observe that all the metric values of the distributions improve after applying our model with respect to the initial configuration. On the other hand, the wirelength increases in all the results, except in the one highlighted in blue. We observe that this result, although it does not have the most optimal value for the distribution, manage to improve both the distribution metric and the wirelength metric. Moreover, with respect to the crossing edges metric, its value remains the same. Therefore, we conclude that this distribution is definitely better in all aspects with respect to the configuration given by the Spectral method. In figure 10, we can observe the representation of the initial configuration given by the Spectral method and the discussed distribution.

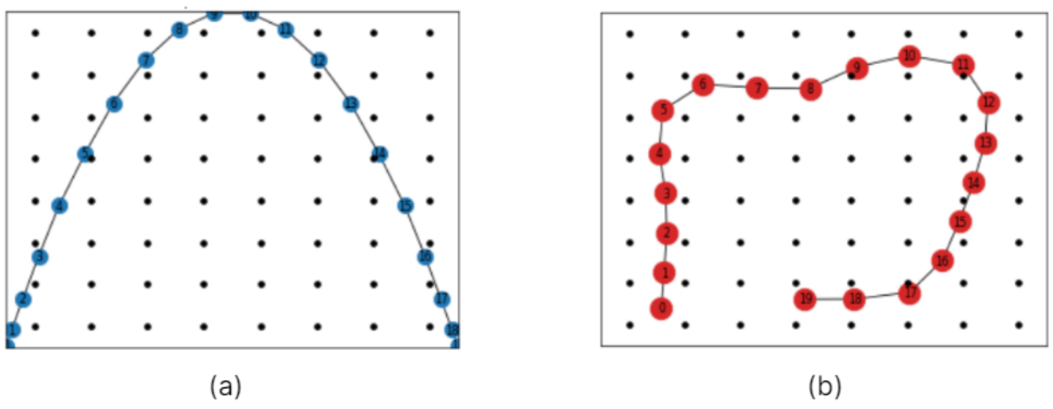


Figure 10: (a): Initial configurations obtained from Spectral method for a path graph of 20 nodes. (b): Result obtained using our model with $\beta = 0.5$, $\gamma = 0.2$.

Regarding the crossing edges metric, for the cases in which we have initialised the code with the spectral configuration, this is usually either worsened by adding one or two crossings or maintained without crossing edges. For the Kamada-Kawai case, most cases remain without crossing edges, although in some cases a crossing does appear. Finally, what is more interesting, is that we find that in the case of Spring, which starts with a pair of crossing edges, for same values of β and γ, the metric tends to improve by undoing this crossing in the resulting distribution. So after applying our method, crossings may disappear from the initial configuration.

Let's compare the results in more detail by distinguishing between the initial configuration we have chosen. We start with the Spectral method. We observe that regardless of the value we have given to β and γ within the interval [0, 0.5], the distribution metric decreases its value noticeably in all cases. We can state that in the smallest case, we can obtain a value of $M_D = 2$ (with $\beta = 0.2$ and $\gamma = 0.4$). Considering that initially this metric had a value of $M_D = 22$, the improvement in uniformity is considerable. However, it is important to see that to obtain this improvement we have had to increase a little the wirelength. Moreover, in that case, we also get one pair of crossing edges. Even so, we find other good solutions where we do not have crossing edges and we have a metric of $M_D = 4$ that still improves a lot the initial distribution uniformity.

In the case of considering the Spring method as the initial configuration, we observe similar results to those obtained starting with the spectral distribution. We can observe that for $\beta = 0.05$ and $\gamma = 0.4$ we can obtain distributions with metrics up to $M_D = 0$. However, we need to allow three crossings and increase the initial wirelength by more than double.

Finally, for the case of initialising the programme with Kamada-Kawai's method, we do not get as good distributions as in the other two cases. By initially having a configuration with a rather non-uniform distribution, the program finds solutions that are not so good. Even so, the distribution metric is initially $M_D = 80$ and we can obtain values of up to $M_D = 8$. That is to say, our programme really improves a lot on the initial configuration. But at the same time, as in the other cases, the wirelength also increases, from 4.1 to 10.9. On the other hand, we observe that initially we did not have crossing edges but in most of the cases our model gives results in which there are no crossing edges either, so in this metric we do not obtain worse results.

In order to find out if there is any dependence between the weights of the objective function and the values of the metrics, we made some plots where the values of two of these metrics are plotted for the different results. We have observed that γ does not have a clear dependence with any of the metrics, unlike β. Although for the crossing edges metric, we have not been able to determine any dependence either, β has a marked dependence for the distribution metric and for the wirelength metric. In figure 11 are the graphs in which the wirelength metric is plotted against the distribution metric for different values of β.

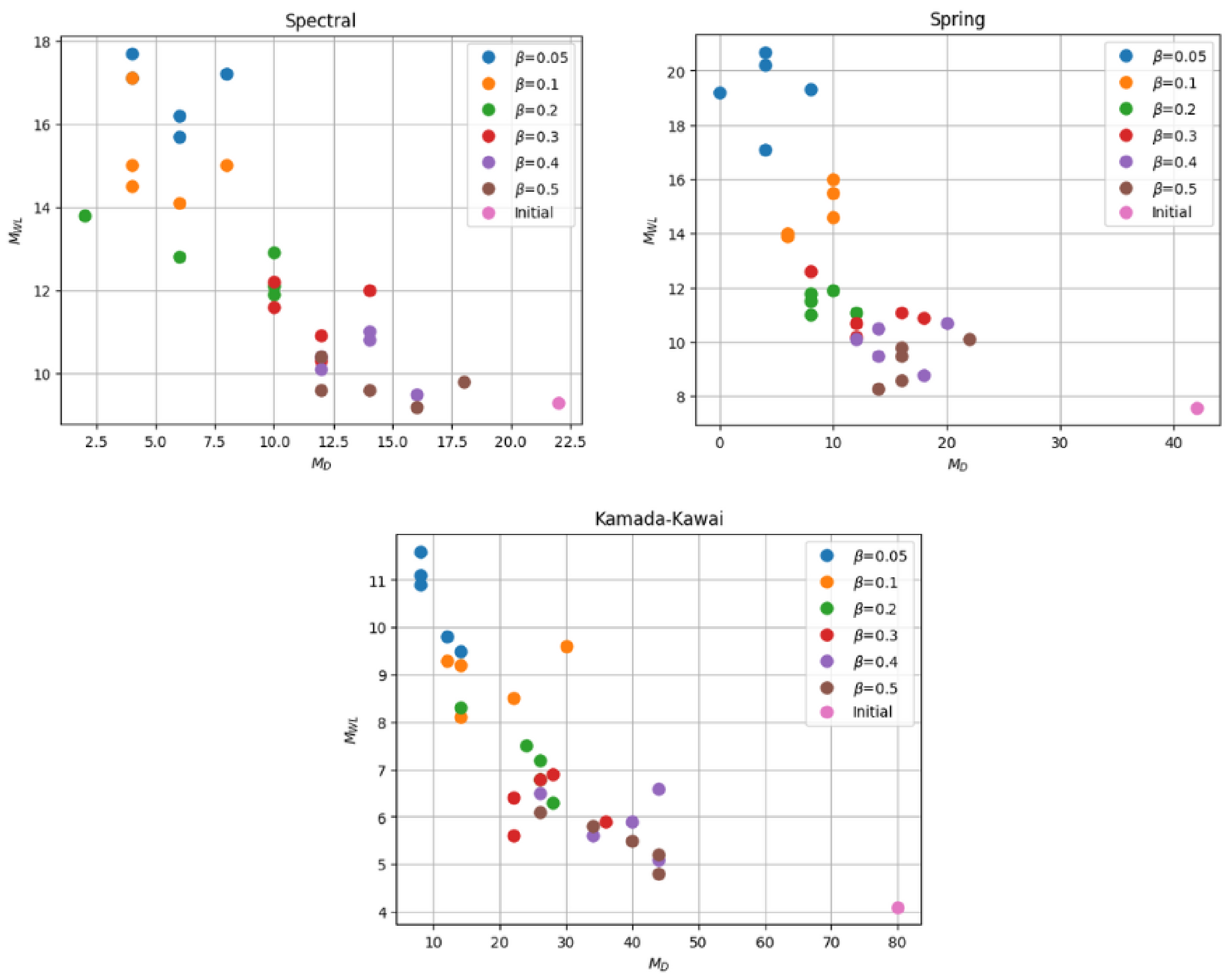


Figure 11: Wirelength metric respect distribution metric for different values of β in a path graph of 20 nodes

We observe that as we increase the value of β, the value of the distribution metric increases (we have a less uniform distribution), while the value of the wirelength decreases. This is consistent with the definition of β, which was the weight given to the wirelength in the objective function. Therefore, by analysing these graphs we can conclude that in general, as we improve the uniformity of the distribution, the wirelength increases. Furthermore, we can also compare the results obtained with the initial configuration (pink point), where we can observe the clear improvement in distribution, but an increase in the wirelength metric.

6.2 Cycle graph

The next case we have studied is a circular graph with 20 modules. The same parameters have been set in this case: number of control points, 64, area of influence of the modules, 1x1 and dimension of the die, 4x4. As in the previous example, we have studied the problem by changing the values of the parameters β and γ within the range [0,0.5].

In the same way, we started the program with the Spectral, Spring and Kamada-Kawai methods. After running our program, we have obtained, as in the previous example, configurations with more uniform distributions along the die. We show below, in figure 12, some examples for different values of β and γ:

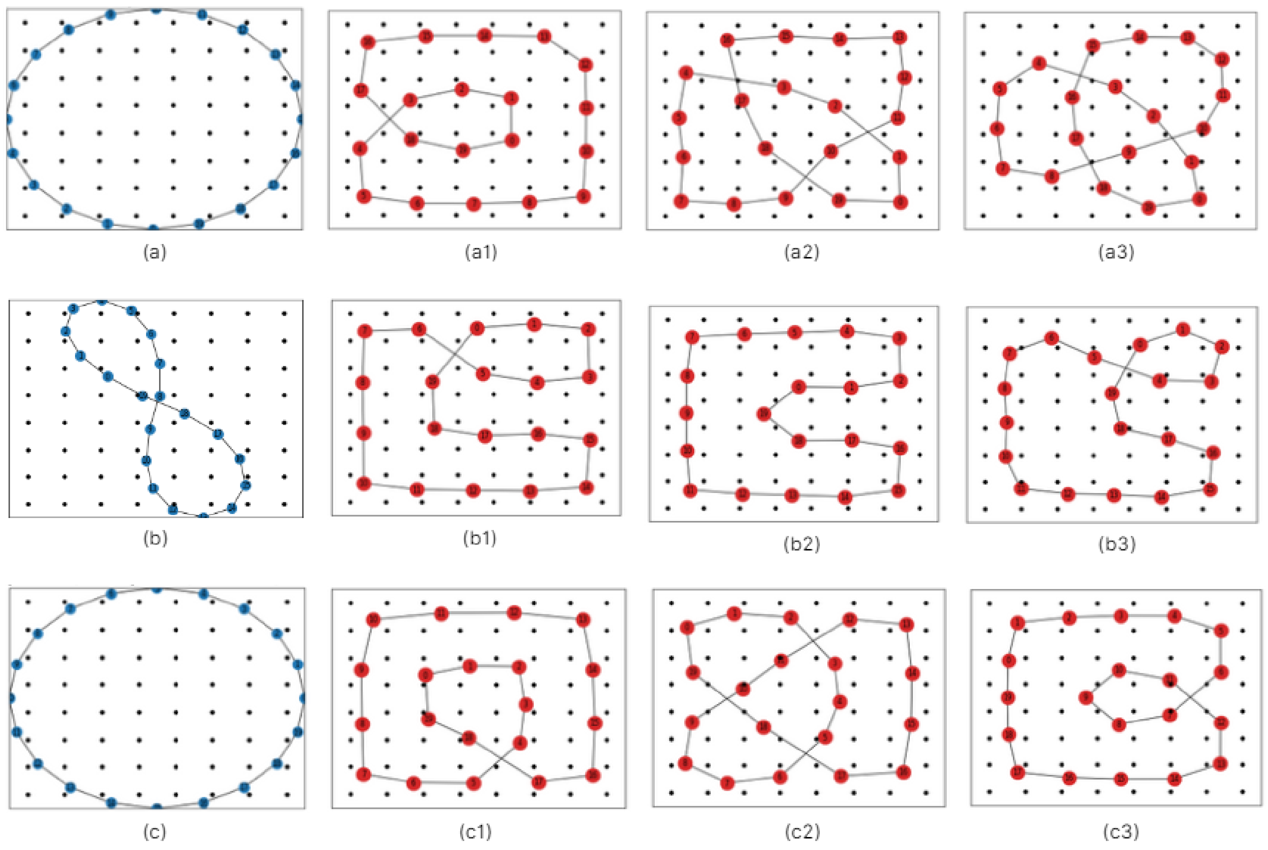


Figure 12: Results for a cycle graph of 20 modules and 64 control points. (a),(b),(c): Initial configurations obtained from Spectral, Spring and Kamada-Kawai methods respectively. The distributions obtained using our model are represented in red. First row corresponds to solutions that used Spectral configuration as initial configuration, second row used Spring and third one, Kamada-Kawai. The parameters used are the following: (a1) $\beta = 0.3$, $\gamma = 0.3$, (a2) $\beta = 0.4$, $\gamma = 0.4$, (a3) $\beta = 0.5$, $\gamma = 0.3$, (b1) $\beta = 0.1$, $\gamma = 0.1$, (b2) $\beta = 0.2$, $\gamma = 0.1$, (b3) $\beta = 0.5$, $\gamma = 0.5$, (c1) $\beta = 0.2$, $\gamma = 0.1$, (c2) $\beta = 0.3$, $\gamma = 0.3$, (c3) $\beta = 0.5$, $\gamma = 0.2$.

We observe that in this case, the Spectral and Kamada-Kawai methods give us very similar, although not identical, configurations. We can differentiate that unlike the Spring method, with these two we also obtain similar solutions after applying our method. Therefore, we can see how the initial solution is connected to the final result.

Simply looking at the examples, we see that most of these results improve the distribution but in contrast increase the number of crossing edges. This is a difference with the path example, where although we could find solutions where edges crossed, we obtained a high number of results where we did not have crossing edges. Below, we show table 2 with the values of the metrics for all the results obtained through our programme. In grey, the examples represented in the figure 12 are highlighted.

Cycle graph: 20 modules and 64 control points										
Initial distribution										
		Spectral			Spring			Kamada-Kawai		
		M_{WL}	M_D	M_{CE}	M_{WL}	M_D	M_{CE}	M_{WL}	M_D	M_{CE}
		12.5	14	0	10.4	24	1	12.5	14	0
Results using our algorithm										
Parameters		Initialising w/ Spectral			Initialising w/ Spring			Initialising w/ Kamada-Kawai		
β	γ	M_{WL}	M_D	M_{CE}	M_{WL}	M_D	M_{CE}	M_{WL}	M_D	M_{CE}
0.05	0.1	25.3	4	6	22.5	4	9	28.4	0	7
0.05	0.2	26.0	4	8	20.8	2	10	29.7	0	9
0.05	0.3	26.3	0	8	20.5	2	3	30.4	6	11
0.05	0.4	25.6	6	6	21.5	2	3	27.8	2	9
0.05	0.5	25.1	6	6	21.8	6	4	30.0	6	11
0.1	0.1	19.2	4	3	16.8	0	1	19.8	4	2
0.1	0.2	21.8	4	3	17.0	0	1	24.7	6	9
0.1	0.3	22.0	6	3	18.1	4	3	23.8	4	6
0.1	0.4	22.7	4	4	16.4	4	1	23.6	4	6
0.1	0.5	22.9	6	3	17.7	4	2	22.5	6	3
0.2	0.1	18.4	4	3	14.5	2	0	17.2	2	1
0.2	0.2	18.4	4	3	14.4	4	0	18.7	4	3
0.2	0.3	18.7	4	3	14.4	4	0	19.0	4	3
0.2	0.4	19.4	4	3	15.7	6	1	19.5	4	3
0.2	0.5	19.5	4	4	17.6	2	3	19.3	4	3
0.3	0.1	17.6	4	3	14.1	6	1	17.5	6	3
0.3	0.2	18.2	6	5	14.3	6	1	17.9	0	3
0.3	0.3	16.3	0	1	14.2	4	1	16.9	8	1
0.3	0.4	17.8	2	3	15.0	4	1	19.0	8	4
0.3	0.5	17.1	4	1	13.6	8	0	19.3	6	4
0.4	0.1	16.4	8	3	13.2	8	1	15.3	4	1
0.4	0.2	15.2	4	1	13.2	10	1	17.2	2	3
0.4	0.3	16.9	8	3	13.4	10	1	17.4	6	3
0.4	0.4	17.4	2	3	13.9	4	1	16.0	2	1
0.4	0.5	16.6	8	2	13.1	8	1	16.0	6	1
0.5	0.1	15.9	6	3	11.1	14	0	14.8	4	1
0.5	0.2	14.7	6	1	12.6	12	1	15.0	2	1
0.5	0.3	16.2	4	3	14.9	12	1	11.4	8	1
0.5	0.4	15.5	6	2	13.0	10	1	15.3	6	1
0.5	0.5	16.5	10	3	13.3	4	1	15.7	8	1

Table 2: Values of the metrics M_{WL}, M_D and M_{CE} for the initial configurations and the results obtained using our model in a cycle graph of 20 modules, for different values of parameters β and γ.

Observing the values in the table, we can affirm that in all cases, our model improves the distribution of the modules within the die, since we obtain lower values for the metric M_D with respect to the initial values. On the other hand, we see that the metric M_{WL} is improved in just one example (initialising with Spring method and using $\beta = 0.5$ and $\gamma = 0.3$), where M_D is also improved but we need to allow one pair of crossing edges. In general, the value of M_{WL} is greater in our model, but we can greatly improve the distribution by increasing the wirelength less than twice. Moreover, we see that in practically all cases, the metric M_{CE} gets worse. Nevertheless, starting with the Spring method, we can distinguish some examples where the initial crossing is undone.

It is also interesting to note that in some of the cases highlighted in grey we can obtain a distribution with metric $M_D = 0$. That is, the modules are perfectly uniformly distributed within the die. However, to achieve these cases we need to increase the wirelength and allow at least one crossing between edges.

Finally, to check the dependencies between the metrics and the parameters, we have again represented the different metrics in a plot using several values for the parameters. In those plots, we have been able to observe that, as in the previous example, γ does not have a clear dependence with any of the metrics.

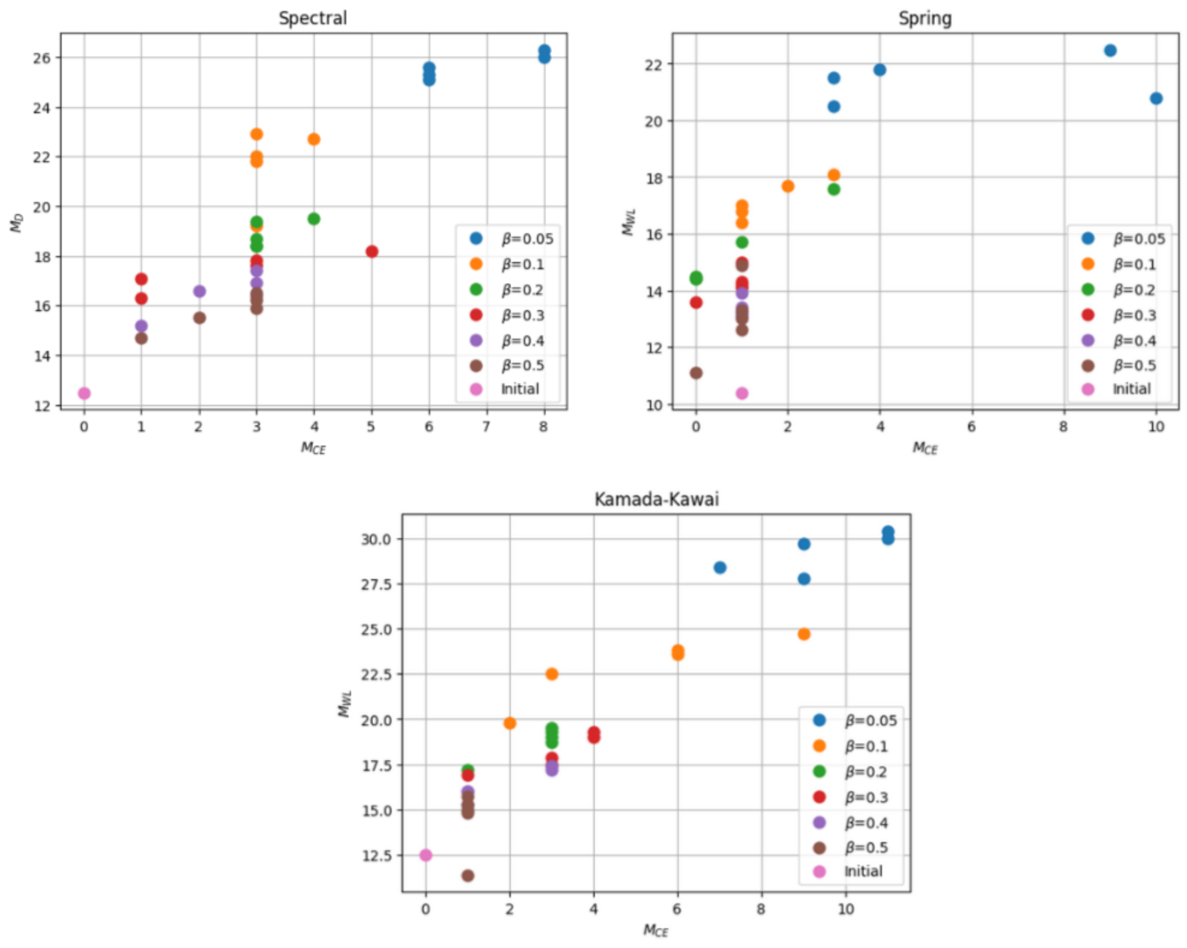


Figure 13: Wirelength metric respect crossing edges metric for different values of β in a cycle graph of 20 nodes.

Unlike in the example with the path graph, we have noticed that in this case, the parameter β does not have a clear dependence with the distribution metric M_D. However, it would seem that in this case, the β parameter does have a more direct influence on the number of crossing edges. In figure 13 we have plotted the metric M_{WL} with respect to M_{CE}. We can observe that in general, for larger values of β, the number of crossings decreases. Besides, we can also observe the expected behaviour where M_{WL} decreases when β increases.

It is also represented in these graphs how the number of crossing edges increases with respect to the initial configuration in all cases except in three where we use the spring method initially. Furthermore, the wire length is greater in all cases compared to the initial positions of the modules. With this we can conclude that through our programme we can obtain uniformly distributed circular graphs in exchange for slightly increasing the wirelength and allowing some crossing between edges.

6.3 Grid graph

In the following we have considered a two-dimensional grid graph. In this case, we have set the number of modules to 5x5, the number of control points to 64, the dimensions of the die to 4x4 and the areas of influence of the modules to 1x1. In this case, using the Spectral, Spring and Kamada-Kawai methods to draw the graph yields to very similar results. For this reason, we have decided to show only the results obtained from the spectral configuration, as they are very similar to the other results and there is no interest in comparing them. In figure 14, we show the initial module configuration used and some of the results obtained for certain values of β and γ.

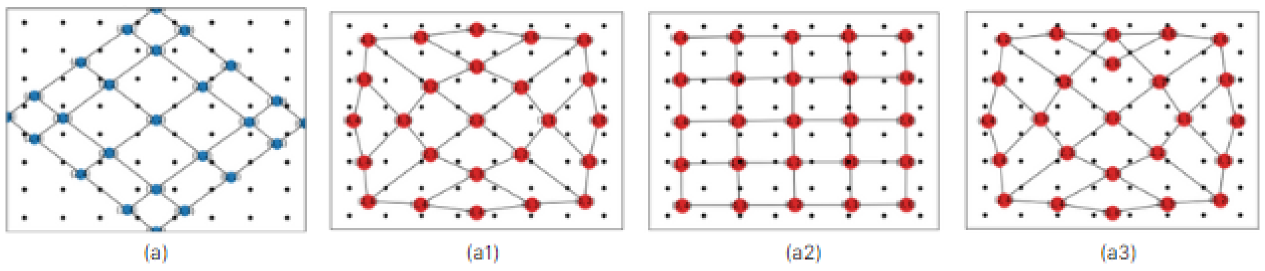


Figure 14: Results for a grid graph of 5x5 nodes and 64 control points. (a) Initial configurations obtained from Spectral method. In red we have the solutions obtained using our model. The parameters used are the following: (a1) $\beta = 0.4$, $\gamma = 0.1$, (a2) $\beta = 0.5$, $\gamma = 0.1$, (a3) $\beta = 0.5$, $\gamma = 0.4$.

We can see how the modules are evenly distributed throughout the die area. The following table 3 shows the different metrics evaluated in the case of the 25-module grid. In this case we have not used values of $\beta < 0.2$ because with these we obtained results where the wirelength and the number of crossings increased a lot. Although the metric distribution in many of these cases was minimal, we considered that due to the large increase in the other two metrics we could not consider these results as good distributions. That is why in the table we only found values of β within the range [0.2, 0.5]. In grey, the three results shown in the figure 14 are highlighted.

In all the results, after using our model we managed to improve the uniformity of the initial distribution in exchange for increasing the wirelength. In some cases, it is also necessary to allow crossing between some edges to solve the problem. We remark that for this example, we can find cases where, by slightly increasing the wirelength, we can get a distribution as uniform as possible, where $M_D = 0$. Moreover, this can be achieved in some cases without obtaining crossing edges (examples underlined in grey with ($\beta = 0.4$,$\gamma = 0.1$) and ($\beta = 0.5$,$\gamma = 0.1$)).

Grid graph: 25 modules and 64 control points				
Initial distribution				
		Spectral		
		M_{WL}	M_D	M_{CE}
		28.3	44	0
Results using our algorithm				
Parameters		Initialising w/ Spectral		
β	γ	M_{WL}	M_D	M_{CE}
0.2	0.1	37.3	2	3
0.2	0.2	43.6	6	17
0.2	0.3	37.6	10	4
0.2	0.4	45.2	4	21
0.2	0.5	42.6	4	16
0.3	0.1	35.5	2	6
0.3	0.2	37.3	2	5
0.3	0.3	37.1	2	6
0.3	0.4	38.7	6	10
0.3	0.5	40.6	4	13
0.4	0.1	33.8	0	0
0.4	0.2	31.5	0	0
0.4	0.3	34.9	0	4
0.4	0.4	38.3	8	8
0.4	0.5	40.1	6	8
0.5	0.1	31.1	0	0
0.5	0.2	35.8	0	6
0.5	0.3	31.3	0	0
0.5	0.4	34.4	0	2
0.5	0.5	37.7	8	13

Table 3: Values of the metrics M_{WL}, M_D and M_{CE} for the initial configurations and the results obtained using our model in a grid graph of 25 modules, for different values of parameters β and γ.

In this case the dependence between parameters and metrics is not so clear. From the graphs used in the previous examples, we cannot extract clear information about the dependence for each parameter with the metrics. Obviously, if we fix the value of γ and increase β we see that the metric for the wirelength decreases. As for the other two metrics, there is no clear behaviour. However, we have observed that for small values of β the number of crossing edges is very large.

6.4 Random graph

Finally, we want to see how our program works with random graphs. To do so, we will consider the same graphs used in section 3.4 and run our program with the following parameters: 64 control points, die size of 4x4 and module influence area of 1x1. The parameters β and γ have been varied within the range [0.1,0.5]. The results are shown in the following sections:

6.4.1 Random graph with 10 nodes

In the figure 15 we can see some of the results obtained for a 10-node random graph. As in the previous examples, the initial configurations correspond to the results obtained from the Spectral, Spring and Kamada-Kawia methods. We can see how for the selected parameters, the uniformity in the distribution is improved after applying our model.

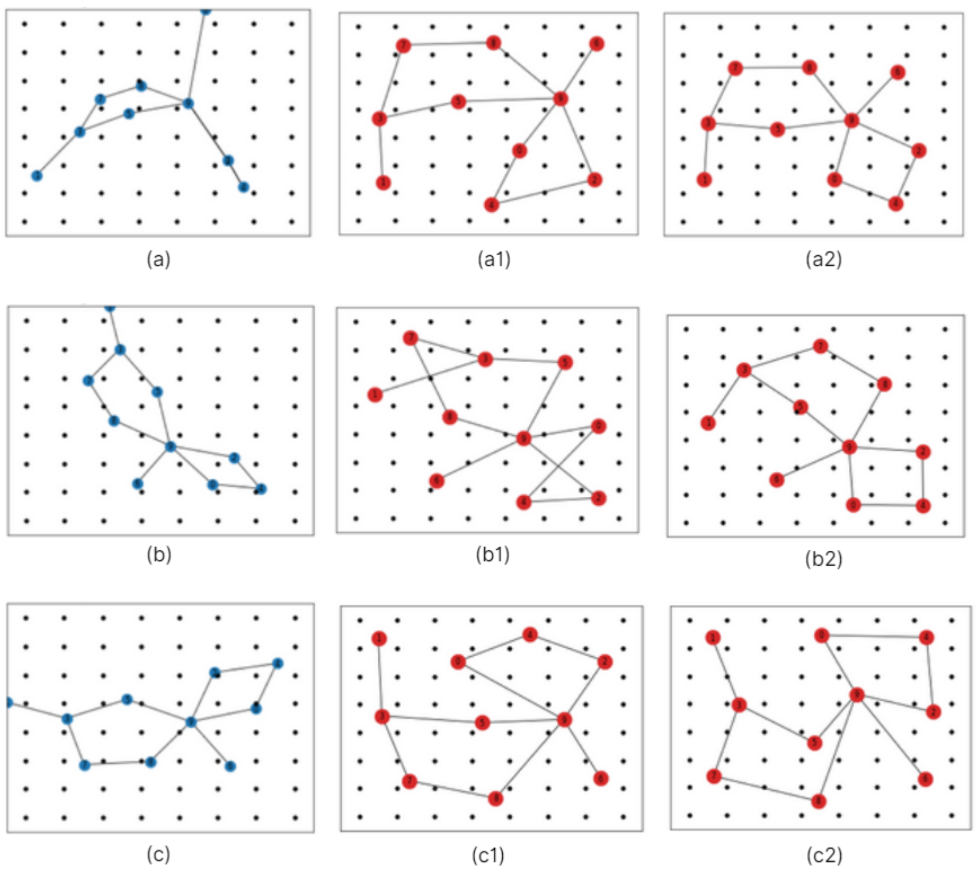


Figure 15: Results for a random graph of 10 modules and 64 control points. (a),(b),(c): Initial configurations obtained from Spectral, Spring and Kamada-Kawai methods respectively. The distributions obtained using our model are represented in red. First row corresponds to solutions that used Spectral configuration as initial configuration, second row used Spring and third one, Kamada-Kawai. The parameters used are the following: (a1) $\beta = 0.1$, $\gamma = 0.1$, (a2) $\beta = 0.5$, $\gamma = 0.5$, (b1) $\beta = 0.1$, $\gamma = 0.1$, (b2) $\beta = 0.3$, $\gamma = 0.1$, (c1) $\beta = 0.1$, $\gamma = 0.3$, (c2) $\beta = 0.1$, $\gamma = 0.5$.

The values of the metrics are shown in the table 4. In grey we show those corresponding to the examples represented in 15. We can see that after applying our model to the random graph, the distribution improves in all cases with respect to the initial configuration except in those marked in red. For those, the distribution metric has either the same or higher values. In these cases, we observe that none of the three metrics is improved, so the initial distribution would be better than the one obtained after using our model. This indicates the importance of consciously choosing the values of β and γ. With a good choice of these parameters we observe that for the three initial configurations, we are able to reduce our distribution metric significantly.

Random graph: 10 modules and 64 control points										
Initial distribution										
		Spectral			Spring			Kamada-Kawai		
		M_{WL}	M_D	M_{CE}	M_{WL}	M_D	M_{CE}	M_{WL}	M_D	M_{CE}
		9.4	12.9	0	8.7	8.9	0	9.6	4.9	0
Results using our algorithm										
Parameters		Initialising w/ Spectral			Initialising w/ Spring			Initialising w/ Kamada-Kawai		
β	γ	M_{WL}	M_D	M_{CE}	M_{WL}	M_D	M_{CE}	M_{WL}	M_D	M_{CE}
0.1	0.1	13.8	0.9	0	14.3	0.9	2	20.4	2.9	1
0.1	0.3	13.5	4.9	0	15.0	0.9	2	14.1	0.9	0
0.1	0.5	14.7	0.9	0	16.2	2.9	3	15.0	0.9	0
0.3	0.1	11.6	10.9	0	11.6	2.9	0	13.0	4.9	1
0.3	0.3	11.9	2.9	0	12.3	8.9	2	13.3	4.9	1
0.3	0.5	12.1	4.9	0	12.0ç	2.9	0	12.6	2.9	0
0.5	0.1	10.6	8.9	0	11.0	8.9	1	11.3	6.9	0
0.5	0.3	10.9	6.9	0	10.9	6.9	1	12.2	6.9	1
0.5	0.5	11.2	0.9	0	11.2	6.9	1	12.3	6.9	1

Table 4: Values of the metrics M_{WL}, M_D and M_{CE} for the initial configurations and the results obtained using our model in a random graph of 10 modules, for different values of parameters β and γ.

We observe that we are able to obtain distributions where the value of the metric M_{CE} remains at zero. However, it is necessary to increase the metric M_{WL} in order to improve the uniformity of the distribution. Even so, in grey we can see distributions in which the wirelength increases not much, but the uniformity improves greatly.

In conclusion, for a good choice of parameters β and γ we can obtain more uniform distributions after applying our model than not using only the Spectral, Spring or Kamada-Kawai methods. In these cases, we will obtain distributions that by increasing the wirelength a little bit the uniformity is greatly improved, so in practice, these results can be more useful for floorplanning.

6.4.2 Random graph with 15 nodes

Let us now consider the binary random graph with 15 nodes and probability 0.2. This case is more complicated than the previous one, since by including more nodes in the random graph we will have more probabilities of having crossing edges. In figure 16, we can observe the initial configurations obtained from the Spectral, Spring and Kamada-Kawai methods as well as the final results after applying our algorithm.

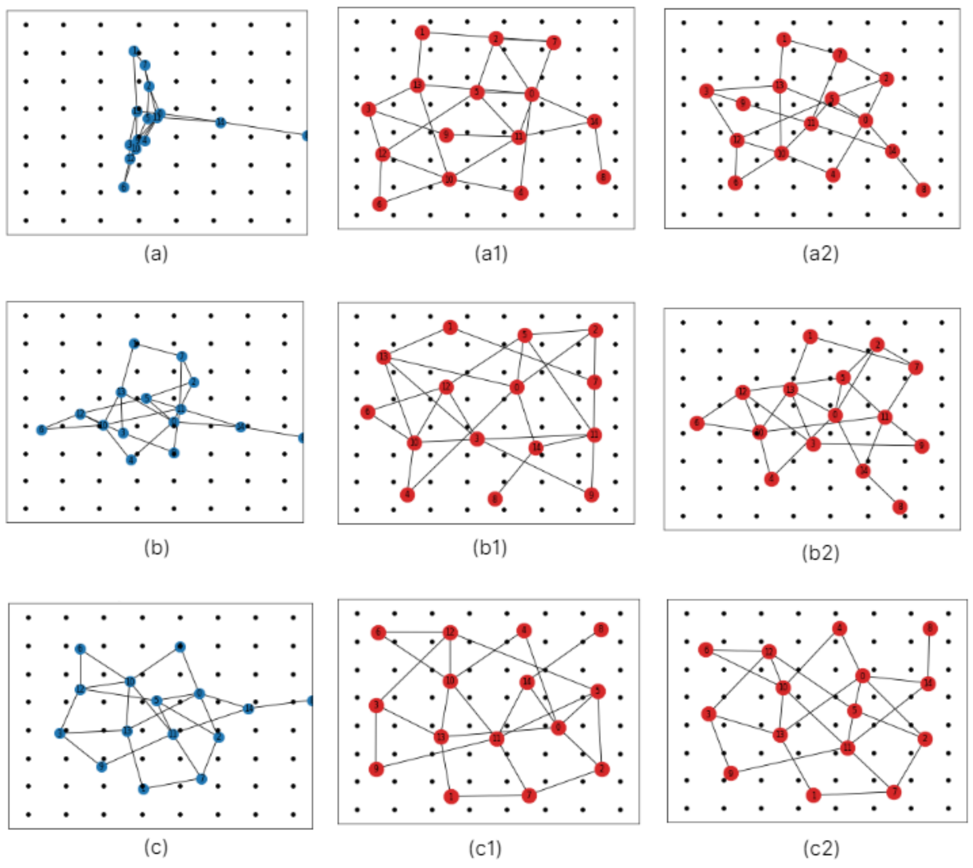


Figure 16: Results for a random graph of 15 modules and 64 control points. (a),(b),(c): Initial configurations obtained from Spectral, Spring and Kamada-Kawai methods respectively. The distributions obtained using our model are represented in red. First row corresponds to solutions that used Spectral configuration as initial configuration, second row used Spring and third one, Kamada-Kawai. The parameters used are the following: (a1) $\beta = 0.1$, $\gamma = 0.5$, (a2) $\beta = 0.3$, $\gamma = 0.5$, (b1) $\beta = 0.1$, $\gamma = 0.3$, (b2) $\beta = 0.5$, $\gamma = 0.5$, (c1) $\beta = 0.3$, $\gamma = 0.1$, (c2) $\beta = 0.5$, $\gamma = 0.3$.

We observe that the distribution obtained by the Spectral method contains a large number of nodes grouped in the same region. For this reason, in this case, this method would not be a good algorithm to solve the floorplanning problem. Both the Spring method and the Kamada-Kawai method give better distributed layouts, but the uniformity can still be improved. In contrast, the results obtained with our method are observed to be more uniform. In the following table 5 we can see which metrics have improved after applying our algorithm.

Random graph: 15 modules and 64 control points										
Initial distribution										
		Spectral			Spring			Kamada-Kawai		
		M_{WL}	M_D	M_{CE}	M_{WL}	M_D	M_{CE}	M_{WL}	M_D	M_{CE}
		12.9	40.1	8	18.2	16.9	14	19.0	8.9	8
Results using our algorithm										
Parameters		Initialising w/ Spectral			Initialising w/ Spring			Initialising w/ Kamada-Kawai		
β	γ	M_{WL}	M_D	M_{CE}	M_{WL}	M_D	M_{CE}	M_{WL}	M_D	M_{CE}
0.1	0.1	25.4	8.9	7	35.1	2.9	19	69.2	0.9	20
0.1	0.3	26.1	8.9	9	33.3	2.9	15	77.7	0.9	20
0.1	0.5	28.0	4.9	7	33.8	2.9	15	68.6	0.9	20
0.3	0.1	19.1	14.9	7	26.2	12.9	13	41.2	2.9	11
0.3	0.3	20.6	10.9	7	27.0	10.9	15	43.0	2.9	12
0.3	0.5	21.6	4.9	8	27.6	6.9	14	41.6	2.9	13
0.5	0.1	16.6	22.9	8	26.4	12.9	14	35.1	6.9	10
0.5	0.3	18.0	16.9	8	24.1	8.9	14	35.2	4.9	10
0.5	0.5	19.0	16.9	8	24.9	10.9	13	38.9	4.9	12

Table 5: Values of the metrics M_{WL}, M_D and M_{CE} for the initial configurations and the results obtained using our model in a random graph of 15 modules, for different values of parameters β and γ.

We observe that in all cases the distribution metric M_D decreases its value with respect to its initial configuration after using our algorithm. We observe that after using our algorithm, in the best cases, we can get values for the distribution metric between $M_D = 0.9$ and $M_D = 4.9$. To achieve this, in all cases the wirelength is increased quite a lot. On the other hand, the metric M_{CE} in some cases improves and in others worsens with respect to its initial configuration. But for a good choice of parameters we can obtain uniform distributions where in the worst case the value of M_{CE} does not increase by more than three units.

In conclusion, for this random graph, we can get uniformly distributed configurations of modules by increasing the wirelength. Moreover, it is likely that in the result, more crossing edges appear. However, the number of crossings we need to obtain good configurations is not much higher than what we get with the Spectral, Spring and Kamada-Kawai methods. In general, we would have to study the particular problem. If uniformity in the distribution is important, we could use our algorithm in exchange for sacrificing the minimisation of the wirelength. On the other hand, if in our particular problem, the wirelength has to be reduced to the maximum, it might be better to use another of the methods we have seen for drawing graphs taking into account that the modules will not be so well distributed in the die.

6.5 Changing the number of control points

So far we have analysed results for different types of graphs by varying the values of the weights in the objective function, β and γ. For these results other program parameters have been fixed, such as the number of control points, the dimensions of the die, etc. These parameters can also influence the result. In particular, we are interested in studying how the number of control points affect the results. In order to analyse that, we have used the simplest of graphs, the path graph. Through our program we have distributed a set of path graphs for fixed values of the weights in the objective function. Specifically, for $\beta = 0.1$ and $\gamma = 0.2$. We used the Spectral method in the first step of our code to obtain the intial configuration from where we start the optimization.

We have decided to study path graphs with different numbers of modules because it is reasonable to think that this may influence the choice of the number of control points. In particular, we have distributed path graphs of 10,20,30 and 40 nodes.

The table 6 below shows the results obtained for the different metrics according to the number of control points chosen. It should be noted that, to be coherent with the square dimensions of the die, we distributed the control points in a square grid. Therefore, the number of control points is square.

Changing the number of control points												
	10-node path			**20-node path**			**30-node path**			**40-node path**		
	Initial distributions											
	M_{WL}	M_D	M_{CE}	M_{WL}	M_D	M_{CE}	M_{WL}	M_D	M_{CE}	M_{WL}	M_D	M_{CE}
	9.8	4.9	0	4.9	22	0	3.3	55	0	2.5	75.7	0
	Results using our algorithm											
Control Points	M_{WL}	M_D	M_{CE}	M_{WL}	M_D	M_{CE}	M_{WL}	M_D	M_{CE}	M_{WL}	M_D	M_{CE}
16	14.7	4.9	0	7.0	10.0	0	4.9	18.0	0	70.0	83.9	57
25	14.8	2.9	0	10.0	4	1	7.8	20	1	5.8	29.9	1
36	29.9	0.9	5	9.5	8	0	7.5	14	1	5.6	33.9	0
49	15.7	4.9	0	12.0	6	1	8.0	14	1	5.6	27.9	0
64	12.6	4.9	0	10.5	6	0	7.0	12	0	5.8	25.9	0
81	15.3	2.9	0	12.1	8	1	9.0	16	1	5.8	19.9	0

Table 6: Values of the metrics M_{WL}, M_D and M_{CE} for the initial configurations and the results obtained using our model in path graphs of 10,20,30 and 40 modules, for different number of control points.

In the table, the best solutions for each of the examples are highlighted in grey, which are also represented in figure 17. We can affirm that when the number of control points is much smaller than the number of modules, the algorithm has difficulties to find a solution to the optimisation problem. For example, for the case of 16 control points, in the case of the 40-node graph, the algorithm does not find an optimal solution after the maximum number of iterations. In contrast, for the 10-node graph the algorithm can return a good distribution.

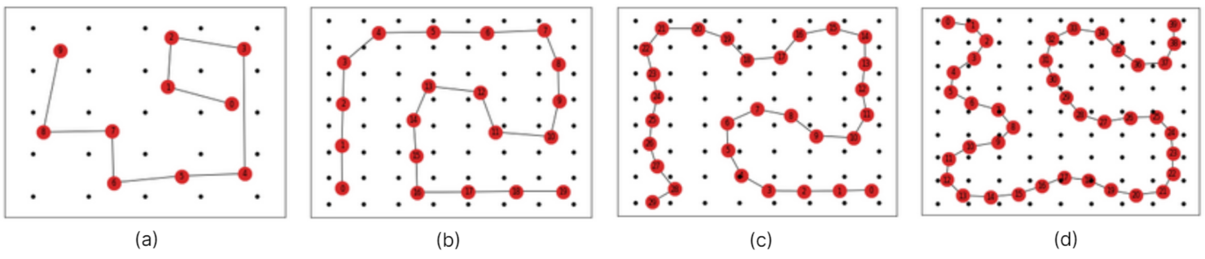


Figure 17: Results obtained initialising our model with the Spectral method. (a) 10-node path with 25 control points, (b)20-node path with 64 control points, (c) 30-node path with 64 control points, (d) 40-node path with 81 control poitns.

In general, we have observed that the algorithm tends to work well for a number of control points about approximately twice the number of modules. However, by increasing the number of control points we also find good solutions. On the other hand, if the number of control points is much lower, it is possible that the algorithm is not able to find uniform distributions.

Even so, it is important to keep in mind that the algorithm does not only depend on this parameter as we have seen in previous sections. In other words, it is very likely that by modifying the values of the weights, we can obtain better solutions with the same number of control points. This implies that for each particular problem we will have to study well possible combinations of parameters.

When choosing the number of control points, we not only have to take into account the improvement in the distribution of the modules. An increase in the number of number of control points usually influences at the same time an increase in the problem solving time. Therefore, we increase the computational cost of the program. In figure 18, we see a plot representing the time the program needed to find each of the solutions as a function of the number of control points.

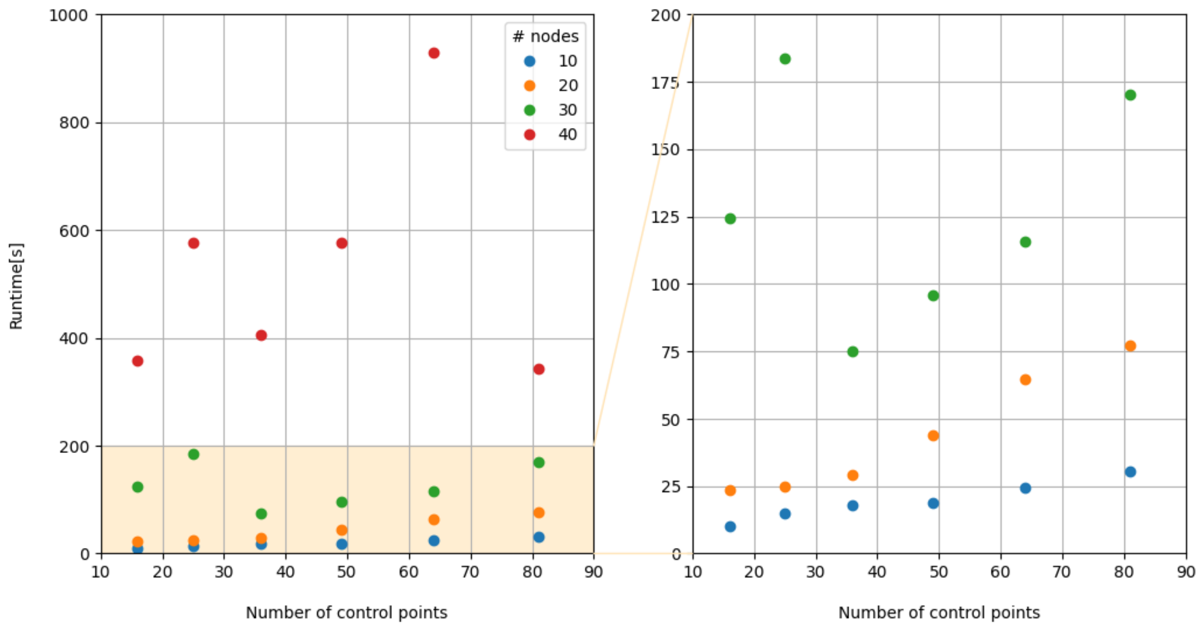


Figure 18: Runtime of the program as a function of the number of control points for 10, 20, 30 and 40-node path graph. The plot on the right shows a zoom of the left hand side plot for the path graphs with 10, 20 and 30 nodes.

We can observe that for path graphs of 10 and 20 nodes, the behaviour is as expected with an increase of the runtime as we increase the number of control points. However, we can see that for graphs with more nodes, the behaviour is different. As we have mentioned before, the number of control points influences the solution of the problem. We have observed that in general, when the number of nodes increases, the best solutions are found with more control points. In general this number is between two and three times the number of nodes. In addition, when the number of control points is much smaller than the number of nodes, we have observed solutions that are not very uniform or have many crossing edges. Therefore, we can assume that the runtime for graphs of 30 and 40 nodes is higher in cases where we have fewer control points because the program cannot find optimal distributions.

With this we can conclude, that in general, when we want to solve a particular floorplanning problem, we have to take into account that the number of control points we choose will depend on the number of modules we have. Moreover, we can assume as a first solution that this number will be around double or triple the number of nodes to obtain good distributions. On the other hand, keep in mind that for a very high number of modules, the runtime required with our program can increase significantly.

7. Conclusions and future work

The aim of this thesis was to find a mathematical algorithm to solve the first level of the floorplanning problem. That is, we wanted to distribute the modules within a die, through the representation with graphs, where the nodes represent the modules. After having worked with different models, we finally found an algorithm that achieved satisfactory results in many of the cases studied.

This algorithm is divided into two steps. The first step is the graphical representation of the graphs through one of the following techniques: Spectral method, Spring layout or Kamada-Kawai method. These methods allow us to draw the modules according to a graph. The second step of the algorithm is based on the improvement of these initial distributions.

The resolution of the first case has been automatic through the NetworkX library. This library includes some functions that receive a graph as input and return the graphical representation calculated through the Spectral, Spring and Kamada-Kawai methods.

The second step of the algorithm has been treated as a non-convex optimisation problem. Specifically, it has been solved using the APOPT solver of the GEKKO library. In the formulation of the problem we minimised a weighted sum with three terms representing: the wirelength, the uniformity of the distribution and the repulsion forces between modules. To determine the uniformity term, we created a grid of control points within the die. We then considered the modules and control points as physical particles. In this way, we could calculate the potential that the modules exerted on each control point. This potential is given the form of a Gaussian function in order to optimise the problem using graident-based methods. In the end, the distribution term in the objective function was calculated as the sum of potential variances at all control points.

As the problem is non-convex, we can find different local minima depending on the initial solution with which we start the problem. For this reason, we have studied the performance of the algorithm using the different initial configurations given by the three graph drawing methods indicated above. In the project, we have studied different types of graphs: path, circular, grid and random graphs. For each of these we have compared the results obtained with the Spectral, Spring and Kamada-Kawai methods with those obtained after applying our model. As a result, we have seen that in most of cases, our model manages to uniform the nodes of the graph along the die area. Consequently, edges generally tend to increase in length, so in order to improve the distribution we need to sacrifice the minimisation of the wirelength. In addition, we have seen that when we use our model, the number of crossing edges in the results is usually similar to that obtained with the other methods, although it can both increase and decrease depending on the example.

We have also analysed the results by varying the parameters of the algorithm. Initially, we have compared how the weights of the objective function β and γ influence the solutions. We have observed that depending on the problem we want to solve, these parameters influence the results differently. It is difficult to determine a clear dependence of the γ parameter on the metrics used to compare the results. For β we have been able to observe the expected behaviour. As it increases, the wirelength decreases. Moreover, in the case of path graphs we have seen that by increasing β the distribution metric tended to worsen. On the other hand, for circular graphs, we have seen that by increasing β, usually the number of crossing edges decreases. We can conclude that we cannot determine an exact behaviour with the increase or decrease of these two parameters, but that by studying different values of these we can vary the distribution of the result quite a lot. Therefore, for each particular problem we will have to analyse the results for different values of the weights.

On the other hand, we have studied the influence of the number of control points. For this purpose, we have worked with path graphs. We have been able to verify that this parameter has a real influence on the results. Moreover, we can assume that the choice of a good value for this number has to be influenced by the number of nodes in our graph. The more nodes we have, the better solutions we can obtain as long as we increase the number of control points. Moreover, if the number of control points is much smaller than the number of nodes, the program may not be able to find an optimal solution. However, in general, the increase in the number of control points goes hand in hand with the increase in computational cost.

In conclusion, we can state that our model achieves uniform distributions within the die while minimising the wirelength. Moreover, the number of crossing edges is comparable to that obtained from other graph drawing methods. However, in general, the wirelength is increased with respect to other methods but in contrast we achieve more uniformity. That is why, with a good choice of model parameters, we can obtain better distributions than those obtained by other methods such as Spectral, Spring or Kamada-Kawai.

These new distributions can help to increase the efficiency of integrated circuits. However, there is still a lot of work to be done in this area. First of all, we need to start working with real problems and study for each particular problem the parameters of the model with which we can obtain the desired distribution. We claim that our model will be able to provide an optimal distribution for the corresponding graph. Afterwards, we would need to proceed to the next level of the floorplanning problem. In order to solve this one, we can provide our distribution as an initial solution for a more effective solution. In this second level, we would take into account the areas of the particular modules and add other constraints to the problem such as avoiding overlapping between modules.

References

[1] Naylor et al., *Non-linear optimization system and method for wire length and delay optimization for an automatic electric circuit placer*, US006301693B1, Oct.9, 2001

[2] Yong Zhan, Yan Feng and Sachin S. Sapatnekar, *A Fixed-die Floorplanning Algorithm Using an Analytical Approach* In Proceedings of the ASP-DAC 2006: Asia and South Pacific Design Automation Conference 2006 (pp. 771-776). [1594779] https://doi.org/10.1145/1118299.1118477

[3] A. B. Kahng and Q. Wang, *Implementation and Extensibility of an Analytical Placer*, Proceedings of the ACM International Symposium on Physical Design, pp. 18-25, Apr. 2004

[4] Gallier, Jean and Quaintance, Jocelyn, *Linear Algebra and Optimization with Applications to Machine Learning: Volume I: Linear Algebra for Computer Vision, Robotics, and Machine Learning*, DOI:10.1142/11446, March. 2020

[5] Chris Godsil and Gordon Royle. *Algebraic Graph Theory*. GTM No. 207. Springer Verlag, first edition, 2001.

[6] Ulrik Brandes. *Drawing on Physical Analogies. Drawing Graphs.* LNCS 2025: 71–86, 2001.

[7] Tomihisa Kamada and Satoru Kawai. *An Algorithm for Drawing General Undirected Graphs.* Information Processing Letters 31:7-15, 1989.

A. Implementation of the model in Python

```python
import numpy as np
from gekko import GEKKO
import networkx as nx
import matplotlib.pyplot as plt
import numpy as np
import random as rd
import math

# Function to set global options of GEKKO
def global_options(m, solver, otol, rtol, maxiter):
    m.options.SOLVER = int(solver)  # Solver: 1: APOPT, 2: BPOPT, 3: IPOPT
    m.options.OTOL = otol  # Optimisation function tolerance
    m.options.RTOL = rtol  # Restriction tolerance
    m.options.MAX_ITER = int(maxiter)  # Maximum number of iterations

# When using spectral_layout, the positions of the points are within the range [0:1]. I create
     this function to obtain the positions within the range of the dimensions of the die

def range_function(x, range1, range2):
    b=(range2[1]-range2[0])/(range1[1]-range1[0])
    a=range2[1]-b*range1[1]
    return(a+b*x)

# Function such that I give as an input a natural number and it returns me a pair of natural
    numbers whose product is similar to the input. In this function I try to find the two
    numbers that have a similar ratio between them as the dimensions of the die

def product(dimension_die, number):

    ratio = round(dimension_die[0]/dimension_die[1])
    initial_guess = round(math.sqrt(number/ratio))

    product1 = initial_guess**2*ratio
    product2 = (initial_guess - 1)**2*ratio
    product3 = (initial_guess)*(initial_guess*ratio-1)
    product4 = (initial_guess + 1)**2*ratio
    product5 = (initial_guess)*(initial_guess*ratio+1)

    diference1 = abs(number - product1)
    diference2 = abs(number - product2)
    diference3 = abs(number - product3)
    diference4 = abs(number - product4)
    diference5 = abs(number - product5)

    if minimum_diference == diference1:
        print('1')
        return([initial_guess*ratio , initial_guess])
    elif minimum_diference == diference2:
        print('2')
        return([(initial_guess-1)*ratio, initial_guess - 1])
    elif minimum_diference == diference3:
        print('3')
        return([initial_guess*ratio - 1, initial_guess])
    elif minimum_diference == diference4:
        print('4')
        return([(initial_guess+1)*ratio, initial_guess + 1])
    elif minimum_diference == diference5:
        print('5')
        return([initial_guess*ratio + 1, initial_guess])



```

```python
# GEKKO model

def model2(dimension_die, number_control, graph, number_modules, coord_modules_initial, dimension_modules, solver, otol, rtol, maxiter, draw, coef_edges, coef_repulsion, type_of_graph):

    m=GEKKO()

    # Create parameters and variables with the coordinates of the control points and the modules

    # Find the coordinates of the control points:

    number_control_points_row=int(number_control[0])
    area_control_x=(dimension_die[0]/number_control_points_row)
    if dimension_die[0]==0:
        coord_control_x=[0]
    else:
        coord_control_x=np.linspace(area_control_x/2,dimension_die[0]-area_control_x/2,int(number_control_points_row))

    control_range_x=range(len(coord_control_x))

    number_control_points_column=int(number_control[1])
    area_control_y=(dimension_die[1]/number_control_points_column)
    if dimension_die[1]==0:
        coord_control_y=[0]
    else:
        coord_control_y=np.linspace(area_control_y/2,dimension_die[1]-area_control_y/2,int(number_control_points_column))

    control_range_y=range(len(coord_control_y))

    number_control_points=int(number_control_points_row)*int(number_control_points_column)
    control_range=range(int(number_control_points))
    coord_control=m.Array(m.Param,(int(number_control_points),2))

    n=0
    for x in control_range_x:
        for y in control_range_y:
            coord_control[n,0].value=coord_control_x[x]
            coord_control[n,1].value=coord_control_y[y]
            n=n+1

    # Load the initial coordinates of the modules:

    number_modules=len(coord_modules_initial)
    modules_range=range(int(number_modules))
    coord_modules = m.Array(m.Var, (number_modules, 2), lb=0)
    index_coord_modules = list(coord_modules_initial)

    for p in modules_range:
        for i in range(2):
            coord_modules[p][i].value=coord_modules_initial[index_coord_modules[p]][i]
            coord_modules[p][i].upper=dimension_die[i]

    # Depending on the layout used to obtain the positions of the modules, the dictionary will have different keys. I create a dictionary that allows me to change the keys to {0, 1, 2,...} and vice versa.
    dictionary_to_numbers={}
    dictionary_to_keys={}
    i=0
    for key in list(coord_modules_initial.keys()):
        dictionary_to_numbers[key] = i
        dictionary_to_keys[i] = key
        i=i+1
    # MODEL:
    # Define some Parameters:

    alpha = m.Param(value=coef_edges)
    Pot = [[None]* number_modules for c in control_range]
    Pot_x = [[None]* number_modules for c in control_range]
    Pot_y = [[None]* number_modules for c in control_range]
    Sum_pot_control = [None]* number_control_points
    Sum_pot_modules = [None]* number_modules
    Norm_const_pot = [None]* number_modules

    # Optimise distribution

    area_die = dimension_die[0]*dimension_die[1]
    area_modules = dimension_modules**2 # I assume the same square area for all modules
    expected_pot = m.Param(value=(number_modules*area_modules)/number_control_points)

    # I compute the potentials using a gaussian funciton
    sigma = dimension_modules/3 #sigma**2 is the variance in the Gaussian function

    for p in modules_range:
        for c in control_range:

            Pot_x[c][p] = m.exp(-(1/2)*((coord_modules[p][0]-coord_control[c,0])/sigma)**2)
            Pot_y[c][p] = m.exp(-(1/2)*((coord_modules[p][1]-coord_control[c,1])/sigma)**2)
            Pot[c][p] = Pot_x[c][p]*Pot_y[c][p]

    # I compute the normalisation constant for the potential of each module

    for p in modules_range:
        Sum_pot_modules[p] = m.sum([Pot[c][p] for c in control_range]) #Sum of the potential
    generated for each module

    for p in modules_range:
        Norm_const_pot[p] = area_modules/Sum_pot_modules[p]

    # Sum of the potential at each control point
    for c in control_range:
        Sum_pot_control[c] =m.sum([Norm_const_pot[p]*Pot[c][p] for p in modules_range])

    # Variance of the potential at each control point

    Var_pot = m.sum([(Sum_pot_control[c]-expected_pot.value)**2 for c in control_range])

    # Normalisation constant of the distribution term in the objective funciton

    Pot_initial = [[None]* number_modules for c in control_range]
    Pot_x_initial = [[None]* number_modules for c in control_range]
    Pot_y_initial = [[None]* number_modules for c in control_range]
    Sum_pot_control_initial = [None]* number_control_points
    Sum_pot_modules_initial = [None]* number_modules
    Norm_const_pot_initial = [None]* number_modules

    for p in modules_range:
        for c in control_range:
            Pot_x_initial[c][p] =  m.exp(-(coord_modules_initial[index_coord_modules[p]][0]-
    coord_control[c,0])**2/(2*(sigma)**2))
            Pot_y_initial[c][p] =  m.exp(-(coord_modules_initial[index_coord_modules[p]][1]-
    coord_control[c,1])**2/(2*(sigma)**2))
            Pot_initial[c][p] = Pot_x_initial[c][p]*Pot_y_initial[c][p]

    for p in modules_range:
        Sum_pot_modules_initial[p] = m.sum([Pot_initial[c][p] for c in control_range])

    for p in modules_range:
        Norm_const_pot_initial[p] = area_modules/Sum_pot_modules_initial[p]

    for c in control_range:
        Sum_pot_control_initial[c] =m.sum([Norm_const_pot_initial[p]*Pot_initial[c][p] for p
    in modules_range])

    Var_pot_initial = m.sum([(Sum_pot_control_initial[c]-expected_pot.value)**2 for c in
    control_range])

    # Optimise minimisation edges

    sum_edges = m.Intermediate(m.sum([(coord_modules[dictionary_to_numbers[i[0]]][0]-
    coord_modules[dictionary_to_numbers[i[1]]][0])**2+(coord_modules[dictionary_to_numbers[i
    [0]]][1]-coord_modules[dictionary_to_numbers[i[1]]][1])**2 for i in list(graph.edges())]))

    # Normalisation constant of the edges term in the objective funciton

    sum_edges_initial=0
    for i in list(graph.edges()):
        sum_edges_initial += (coord_modules_initial[i[0]][0]-coord_modules_initial[i[1]][0])
    **2+(coord_modules_initial[i[0]][1]-coord_modules_initial[i[1]][1])**2
        if sum_edges_initial==0:
            sum_edges_initial=1

    # Repulsion forces

    epsilon = 1e-8
    Repulsion_force = m.sum([m.sum([1/(((coord_modules[i][0]-coord_modules[j][0])**2+(
    coord_modules[i][1]-coord_modules[j][1])**2+epsilon)) for j in range(i+1, number_modules)
    ]) for i in range(number_modules-1)])

    # Normalisation constant of the repulsion forces term in the objective funciton

    Repulsion_force_initial = sum([sum([1/(((coord_modules_initial[index_coord_modules[i]][0]-
    coord_modules_initial[index_coord_modules[j]][0])**2+(coord_modules_initial[
    index_coord_modules[i]][1]-coord_modules_initial[index_coord_modules[j]][1])**2+epsilon))
    for j in range(i+1, number_modules)]) for i in range(number_modules-1)])

    # Objective function
    m.Minimize(alpha*(1/sum_edges_initial)*sum_edges + (1/Var_pot_initial)*Var_pot +
    coef_repulsion*(1/Repulsion_force_initial)*Repulsion_force)

# Solve the problem

    global_options(m, solver, otol, rtol, maxiter)

    m.solve(debug=0, disp=True)

    # Draw the solution

    if draw=="True":
        node_labels={}
        node_labels_final={}
        node_labels_control={}

        #Initial positions:

        for i in index_coord_modules:
            for j in range(2):
                coord_modules_initial[i][j]=round(coord_modules_initial[i][j],2)

        for p in modules_range:
            node_labels[p]=coord_modules_initial[dictionary_to_keys[p]]

        #Final positions:

        final_graph=nx.Graph()
```

```python
    final_graph.add_nodes_from([p for p in modules_range])
    coord_modules_final = nx.spectral_layout(final_graph)

    for p in modules_range:
        coord_modules_final[p]=[coord_modules[p][0].value[0], coord_modules[p][1].value[0]]
        node_labels_final[p]=coord_modules_final[p]

    j=0
    for i in index_coord_modules:
        coord_modules_final[i]=coord_modules_final.pop(j)
        j=j+1

    final_graph.add_edges_from(list(graph.edges()))

    #Control positions:

    control_graph=nx.Graph()
    control_graph.add_nodes_from(list(c for c in control_range))
    nodePos_control = nx.spectral_layout(control_graph)

    for c in control_range:
        nodePos_control[c]=[coord_control[c,0].value[0], coord_control[c,1].value[0]]
        node_labels_control[i]=nodePos_control[c]

    # Performance of the model
     # Wirelength metric

    wl_initial=0
    for i in list(graph.edges()):
        wl_initial += ((coord_modules_initial[i[0]][0]-coord_modules_initial[i[1]][0])**2+(coord_modules_initial[i[0]][1]-coord_modules_initial[i[1]][1])**2)**(1/2)

    wl_final=0
    for i in list(graph.edges()):
        wl_final += ((coord_modules_final[i[0]][0]-coord_modules_final[i[1]][0])**2+(coord_modules_final[i[0]][1]-coord_modules_final[i[1]][1])**2)**(1/2)

    # Distribution metric for the initial positions:

    control_areas = product(dimension_die, number_modules)
    modules_in_control_area = np.zeros(control_areas[0]*control_areas[1])
    number_control_areas = control_areas[0]*control_areas[1]

    area_control_distribution_x=(dimension_die[0]/control_areas[0])
    area_control_x=np.linspace(area_control_distribution_x/2,dimension_die[0]-area_control_distribution_x/2,int(control_areas[0]))
    control_range_distribution_x=range(len(area_control_x))

    area_control_distribution_y=(dimension_die[1]/control_areas[1])
    area_control_y=np.linspace(area_control_distribution_y/2,dimension_die[1]-area_control_distribution_y/2,int(control_areas[1]))
    control_range_distribution_y=range(len(area_control_y))

    coord_control_area_x = np.zeros(number_control_areas)
    coord_control_area_y = np.zeros(number_control_areas)

    n=0
    for x in control_range_distribution_x:
        for y in control_range_distribution_y:
            coord_control_area_x[n]=area_control_x[x]
            coord_control_area_y[n]=area_control_y[y]
            n=n+1
    for c in range(number_control_areas):
        for p in modules_range:

            if ((coord_modules_initial[index_coord_modules[p]][0]-coord_control_area_x[c])**2 <= (area_control_distribution_x/2)**2) and ((coord_modules_initial[index_coord_modules[p]][1]-coord_control_area_y[c])**2 <= (area_control_distribution_y/2)**2) :
                modules_in_control_area[c] = modules_in_control_area[c] + 1

    distribution_metric_initial = sum([(modules_in_control_area[c] - number_modules/number_control_areas)**2 for c in range(number_control_areas)])

    # Distribution metric for the final result:

    modules_in_control_area = np.zeros(control_areas[0]*control_areas[1])

    for c in range(number_control_areas):
        for p in modules_range:

            if ((coord_modules_final[index_coord_modules[p]][0]-coord_control_area_x[c])**2 <= (area_control_distribution_x/2)**2) and ((coord_modules_final[index_coord_modules[p]][1]-coord_control_area_y[c])**2 <= (area_control_distribution_y/2)**2) :
                modules_in_control_area[c] = modules_in_control_area[c] + 1

    distribution_metric_final = sum([(modules_in_control_area[c] - number_modules/number_control_areas)**2 for c in range(number_control_areas)])

    # Worst possible distribution: All points in the same control area

    distribution_metric_worst = (number_modules/number_control_areas)**2*(number_control_areas-1) + (number_modules - number_modules/number_control_areas)**2

    print('-------------------------------------------------------------')
    print("Initial distribution metric", distribution_metric_initial)
    print("Final distribution metric", distribution_metric_final)
    print("Worst possible distribution", distribution_metric_worst)
    print("Initial length edges", wl_initial)
    print("Final length edges", wl_final)

    # Crossing edges metric

    def segments_intersect(seg1, seg2):
        def orientation(p, q, r):
            val = (q[1] - p[1]) * (r[0] - q[0]) - (q[0] - p[0]) * (r[1] - q[1])
            if val == 0:
                return 0  # Collinear
            elif val > 0:
                return 1  # Clockwise orientation
            else:
                return 2  # Counterclockwise orientation

        def on_segment(p, q, r):
            if (q[0] <= max(p[0], r[0]) and q[0] >= min(p[0], r[0]) and
                    q[1] <= max(p[1], r[1]) and q[1] >= min(p[1], r[1])):
                return True
            return False

        p1, q1 = seg1
        p2, q2 = seg2

        o1 = orientation(p1, q1, p2)
        o2 = orientation(p1, q1, q2)
        o3 = orientation(p2, q2, p1)
        o4 = orientation(p2, q2, q1)

        if (o1 != o2 and o3 != o4):
            return True  # Segments intersect



        # Special cases for collinear segments
        if (o1 == 0 and on_segment(p1, p2, q1)):
            return True
        if (o2 == 0 and on_segment(p1, q2, q1)):
            return True
        if (o3 == 0 and on_segment(p2, p1, q2)):
            return True
        if (o4 == 0 and on_segment(p2, q1, q2)):
            return True

        return False  # Segments do not intersect

    crossing_edges = 0

    for i in list(graph.edges()):
        for j in list(graph.edges()):

            point1_seg1 = [coord_modules_final[i[0]][0], coord_modules_final[i[0]][1]]
            point2_seg1 = [coord_modules_final[i[1]][0], coord_modules_final[i[1]][1]]
            point1_seg2 = [coord_modules_final[j[0]][0], coord_modules_final[j[0]][1]]
            point2_seg2 = [coord_modules_final[j[1]][0], coord_modules_final[j[1]][1]]


            if (i[0] == j[0] or i[0] == j[1] or i[1] == j[0] or i[1] == j[1]):
                a = 0

            else:
                seg1 = ((coord_modules_final[i[0]][0],coord_modules_final[i[0]][1]),(coord_modules_final[i[1]][0],coord_modules_final[i[1]][1]))
                seg2 = ((coord_modules_final[j[0]][0],coord_modules_final[j[0]][1]),(coord_modules_final[j[1]][0],coord_modules_final[j[1]][1]))

                if segments_intersect(seg1, seg2):
                    crossing_edges = crossing_edges + 1

    crossing_edges = crossing_edges/2
    print("Crossing edges:", crossing_edges)

    #Plot:
    fig, ax = plt.subplots(1, dpi=60)
    plt.xlim(0,dimension_die[0])
    plt.ylim(0,dimension_die[1])
    nx.draw_networkx(graph, pos=coord_modules_final, node_color="tab:red", with_labels=True, font_size=9, label="Final position")
    nx.draw_networkx(control_graph, pos=nodePos_control, node_color="black", with_labels=False, font_size=9, node_size=20, label="Control points")
    fig, ax = plt.subplots(1, dpi=60)
    plt.xlim(0,dimension_die[0])
    plt.ylim(0,dimension_die[1])
    nx.draw_networkx(graph, pos=coord_modules_initial, node_color="tab:blue", with_labels=True, font_size=9, node_size=150, label="Initial position")
    nx.draw_networkx(control_graph, pos=nodePos_control, node_color="black", with_labels=False, font_size=9, node_size=20, label="Control points")

    plt.title(str(type_of_graph) + ' Graph with: Edges = ' + str(coef_edges) +  ', Repulsion = ' + str(coef_repulsion) + ', # Control poitns = ' + str(number_control_points) + ', # Modules = ' + str(number_modules) + ', Distribution =' + str(distribution_metric_final) + ', Crossing edges =' + str(crossing_edges))

return(coord_control, coord_modules_final, Pot, Pot_x, Pot_y, expected_pot.value, Sum_pot_control, modules_in_control_area, distribution_metric_final, crossing_edges)
```

Listing 1: Python program of the implementation of our algorithm