

**T.R.**  
**GEBZE TECHNICAL UNIVERSITY**  
**FACULTY OF ENGINEERING**  
**DEPARTMENT OF COMPUTER ENGINEERING**

**MINIMUM CROSSING NUMBER PROBLEM**

**ESRA NUR ARICAN**

**SUPERVISOR**  
**PROF. DIDEM GÖZÜPEK**

**GEBZE**  
**2022**

**T.R.**  
**GEBZE TECHNICAL UNIVERSITY**  
**FACULTY OF ENGINEERING**  
**COMPUTER ENGINEERING DEPARTMENT**

**MINIMUM CROSSING NUMBER  
PROBLEM**

**ESRA NUR ARICAN**

**SUPERVISOR**  
**PROF. DIDEM GÖZÜPEK**

**2022**  
**GEBZE**

 <p><b>GEBZE</b> TECHNICAL UNIVERSITY</p>	<p>GRADUATION PROJECT JURY APPROVAL FORM</p>
--	--

This study has been accepted as an Undergraduate Graduation Project in the Department of Computer Engineering on 16/06/2022 by the following jury.

**JURY**

Member

(Supervisor) : Prof. Didem Gözüpek

Member : Assist.Prof.Dr. Zafeirakis Zafeirakopoulos

# ABSTRACT

The crossing number of a graph  $G$ , denoted by  $cr(G)$ , is defined as the minimum number of pairwise edge intersections of any drawing of  $G$  in the plane. Crossing numbers have been the source of many challenging problems since their introduction by Tur'an in the 1950s.

Determining the crossing number of a graph is NP-hard and remains NP-hard even for restricted versions of the problem, such as finding  $cr(G)$  for a planar graph with the addition of a single edge. In recent years, an integer linear programming approach for finding  $cr(G)$  has been developed in [5], [8] and [10]. This is successful for sparse graphs of moderate size, however ILPs and other exact methods are very limited when it comes to dense graphs. For example, even  $cr(K_{13})$  remains unknown [30].

Therefore, heuristic methods are currently of interest and are the subject of this present work.

This report includes the studies carried out during the semester on the minimum crossing number problem within the scope of the graduation project. With this project, the results of a large literature research on the minimum crossing number problem, on which many studies have been made in graph theory, will be shared. In addition, some codes presented in the studies, experimental findings, the heuristic time algorithm I developed and a comparison of an existing solution are included. The solution of the minimum crossing number problem of two different algorithms is examined with examples.

# ÖZET

Bir  $G$  grafiğinin  $cr(G)$  ile gösterilen kesişme sayısı,  $G$ 'nin düzlemdeki herhangi bir çiziminin minimum ikili kenar kesişim sayısı olarak tanımlanır. Geçiş sayıları 1950'lerde Tur'an tarafından tanıtıldığından beri birçok zorlu sorunun kaynağı olmuştur.

Bir grafiğin geçiş sayısını belirlemek NP-hard olarak sınıflandırılır ve tek bir kenarın eklenmesiyle düzlemsel bir grafik için  $cr(G)$  bulmak gibi problemin kısıtlı versiyonları için bile NP-hard olarak kalır. Son yıllarda, [5], [8] ve [10]'da  $cr(G)$ 'yi bulmak için bir tamsayılı doğrusal programlama yaklaşımı geliştirilmiştir. Bu, orta büyüklükteki seyrek grafikler için başarılıdır, ancak yoğun grafikler söz konusu olduğunda ILP'ler ve diğer kesin yöntemler çok sınırlıdır.

Bu nedenle, crossing number problemi ile ilgili buluşsal yöntemler hala ilgi çekicidir ve bu çalışmanın konusudur.

Bu rapor, bitirme projesi kapsamında minimum geçiş sayısı problemi ile ilgili yarıyıl içinde yapılan çalışmaları içerir. Bu proje ile graph teorisi alanında birçok çalışma yapılmış olan minimum geçiş sayısı problemi üzerine geniş bir literatür araştırmasının sonuçları paylaşılacaktır. Ayrıca çalışmalarda sunulan bazı kodlar, deneysel bulgular, geliştirdiğim sezgisel zaman algoritması ve mevcut bir çözümün karşılaştırması yer almaktadır. İki farklı algoritmanın minimum geçiş sayısı probleminin çözümü örneklerle incelenmiştir.

# **ACKNOWLEDGEMENT**

I would like to thank my supervisor Prof. Dr. Didem Gözüpek for supporting me in this project, Assist.Prof.Dr. Zafeirakis Zafeirakopoulos, who listened me at every follow-up meeting and Gebze Technical University.

In addition, I would like to thank to my family, friends and teachers who had supported me in every way during my education.

**Esra Nur Arıcan**

# LIST OF SYMBOLS AND ABBREVIATIONS

## Symbol or

## Abbreviation : Explanation

NP	: Non-deterministic polynomial time
NP-hard	: Non-deterministic polynomial time-hard
ILP	: Integer Linear Programming
$K_n$	: Complete Graph with n vertices
$K_{(n, m)}$	: Complete Bipartite Graph
PSO	: Particle Swarm Optimization
IRO	: Improved Ray Optimization
CBO	: Colliding Bodies Optimization
ECBO	: Enhanced Colliding Bodies Optimization

# CONTENTS

<b>Abstract</b>	<b>iv</b>
<b>Özet</b>	<b>v</b>
<b>Acknowledgement</b>	<b>vi</b>
<b>List of Symbols and Abbreviations</b>	<b>vii</b>
<b>Contents</b>	<b>ix</b>
<b>List of Figures</b>	<b>xi</b>
<b>1 Introduction</b>	<b>1</b>
<b>2 Prerequisites</b>	<b>2</b>
2.1 Basic Graph Theory Concepts . . . . .	2
2.1.1 Subdivision, Planarity and Thickness . . . . .	2
2.2 Graph Types . . . . .	4
2.2.1 Null Graph . . . . .	4
2.2.2 Trivial Graph . . . . .	4
2.2.3 Non-Directed Graph . . . . .	4
2.2.4 Directed Graph . . . . .	4
2.2.5 Complete Graph . . . . .	4
2.2.6 Cycle Graph . . . . .	5
2.2.7 Bipartite Graph . . . . .	5
2.2.8 Complete Bipartite Graph . . . . .	5
<b>3 Literature Research</b>	<b>6</b>
3.1 Minimum Crossing Number Problem . . . . .	6
3.1.1 Drawings and Embeddings . . . . .	6
3.1.1.1 Book Drawings . . . . .	7
3.1.1.2 Circular Drawings . . . . .	8
3.1.1.3 Embeddings . . . . .	8
3.1.2 Defining the Crossing Number of a Graph . . . . .	9
3.1.3 Parameters Related to the Crossing Number . . . . .	9
3.1.3.1 The maximum induced planar subgraph problem . .	9



3.1.3.2	The skewness of a graph . . . . .	10
3.1.3.3	The vertex splitting number . . . . .	10
3.1.3.4	The page number of a graph . . . . .	10
3.2	Computational Methods . . . . .	10
3.2.1	Brute Force (exact) Algorithms . . . . .	10
3.2.1.1	The Garey-Johnson Algorithm . . . . .	11
3.2.1.2	The Harris-Harris Algorithm . . . . .	11
<b>4</b>	<b>Examined Approaches</b>	<b>12</b>
4.1	Garey-Johnson Algorithm . . . . .	12
4.2	Metaheuristic Algorithms for Minimum Crossing Number . . . . .	14
4.2.1	Particle swarm optimization (PSO) . . . . .	14
4.2.2	Improved ray optimization (IRO) . . . . .	15
4.2.3	Colliding bodies optimization (CBO) and its enhanced version (ECBO) . . . . .	16
4.3	Algorithms Used in the Project . . . . .	18
4.3.1	Quick Cross Algorithm . . . . .	18
4.3.2	Proposed Algorithm . . . . .	20
<b>5</b>	<b>Runtime and Implementation</b>	<b>21</b>
<b>6</b>	<b>Experimental Results</b>	<b>23</b>
6.1	Generating Random Graphs and Running Algorithm . . . . .	23
6.2	Numerical Examples for Metaheuristic Algorithms . . . . .	25
6.3	Comparisons for Quick Cross Algorithm and Proposed Algorithm . . . . .	26
6.3.1	Quick Cross Algorithm Complete Graph Results . . . . .	27
6.3.2	Proposed Algorithm Complete Graph Results . . . . .	27
6.3.3	Quick Cross Algorithm Complete Bipartite Graph Results . . . . .	28
6.3.4	Proposed Algorithm Complete Bipartite Graph Results . . . . .	28
6.4	Proposed Algorithm Results for Simple Graph . . . . .	30
6.5	Visual Display of Simplified Graphs . . . . .	31
<b>7</b>	<b>Conclusions</b>	<b>34</b>
	<b>Bibliography</b>	<b>36</b>

# LIST OF FIGURES

1.1	A layout for the brick factory problem. . . . .	1
2.1	Illustration of graph subdivision . . . . .	3
2.2	A planar graph permits plane drawings although such a graph can still be drawn with crossings. . . . .	3
2.3	An outerplanar graph permits the construction of a plane drawing in which all vertices are present in the same region. . . . .	3
2.4	Complete graph examples. . . . .	4
2.5	Cycle graph examples. . . . .	5
2.6	An example of complete bipartite graph. . . . .	5
3.1	Crossings in self-crossing edges can always be removed. The graph in the left contains an edge $e$ crosses itself. This self-crossing has been removed in right. . . . .	6
3.2	A topological and combinatorial book embedding. . . . .	7
3.3	The possible configurations of two edges in a book. . . . .	7
3.4	A book layout on two pages is equivalent to a circular layout. . . . .	8
4.1	Garey-Johnson . . . . .	12
4.2	Test Planar Algorithm . . . . .	13
4.3	Construct Graph Algorithm . . . . .	14
4.4	Pseudocode for PSO . . . . .	15
4.5	Pseudocode for IRO . . . . .	16
4.6	Pseudocode for CBO . . . . .	16
4.7	Pseudocode for ECBO . . . . .	17
4.8	Embedding of the complete graph $K_8$ obtained by ECBO . . . . .	17
4.9	Main Loop Pseudocode . . . . .	19
4.10	Remove Procedure Pseudocode . . . . .	19
4.11	Star Insertion Pseudocode . . . . .	19
5.1	Main loop in the program . . . . .	21
5.2	For loop in the function that determines edges failed to embed. . . . .	21
5.3	Part of get crossing number function. . . . .	22
6.1	Generating random graphs by given edge and vertex number. . . . .	23
6.2	Directories created for test graph files according to their sizes. . . . .	24

6.3	An example of generated graphs,figure shows big graphs. . . . .	24
6.4	An example of generated output file with informations. . . . .	24
6.5	A terminal output for graph $K_{3,15}$ with commandline options. . . . .	25
6.6	Results for complete graphs compared with metaheuristic algorithms. . . . .	25
6.7	Results for complete bipartite graphs compared with metaheuristic algorithms. . . . .	25
6.8	Results for complete graphs with Quick Cross algorithm. . . . .	27
6.9	Results for complete graphs with proposed algorithm. . . . .	27
6.10	Examining effect of iteration number on small size complete graphs. . . . .	28
6.11	Minimum crossing numbers for Quick Cross algorithm for complete bipartite graphs. . . . .	29
6.12	Minimum crossing numbers for proposed algorithm for complete bipartite graphs. . . . .	29
6.13	Complete bipartite graph results for both algorithms. . . . .	30
6.14	Test results for graphs grouped as small (with vertex number $20 < n < 100$ ). . . . .	31
6.15	Test results for graphs grouped as medium (with vertex numbers between 20 and 100 and edge numbers for 100 and 200. Also see that different runs with same graphs can give different crossing numbers . . . . .	31
6.16	Complete graph with $n=7$ after minimizing crossing numbers to 9. . . . .	32
6.17	Complete graph with $n=7$ example. . . . .	32
6.18	Complete graph with $n=6$ after minimization of crossing numbers. . . . .	33

# 1. INTRODUCTION

In recent years, various kinds of metaheuristic algorithms are proposed to solve instances of problems like minimum crossing number that are believed to be hard in general. These algorithms achieve this by reducing the effective size of the search and exploring that space efficiently. The metaheuristic algorithms are easy to implement and the basic idea behind these methods is usually natural phenomena.

The study of the crossing number problem had its genesis in a brick factory in Budapest during the Second World War. In the factory, there were a number of kilns that were connected to several storage yards by means of railway tracks, and bricks were transported via small trucks that ran on the rails. Paul Turan, who worked in there noticed that the trucks were often derailed where two rails crossed one another. This led him consider the problem of minimizing the number of rail crossings in the system. In the brick factory, every kiln was joined by rail to every storage yard. A bipartite graph may used to model this configuration, and Turan's problem is then equivalent to asking what the minimum number of pairwise edge crossings is over all drawings of the graph.

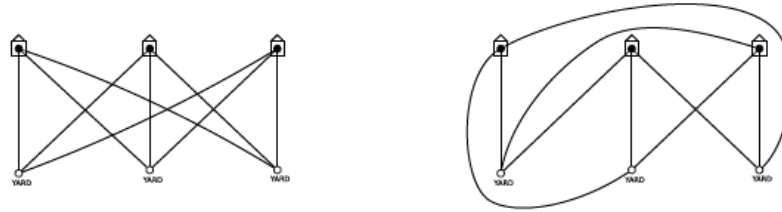


Figure 1.1: A layout for the brick factory problem.

In this paper, the approaches presented on the minimum crossing number problem and examined throughout the semester will be reviewed. The solution I developed, inspired by the Quick Cross algorithm and Planarization approaches, will be explained. In addition to these, the basic issues to be known will be explained in a summary, the operation of the mentioned codes will be detailed, and some comparisons will be included.

With this study, I planned to present a catchy narrative to the readers with an overview of the minimum crossing number problem solutions, many of which have been implemented or designed.

## 2. PREREQUISITES

The purpose of this chapter is to introduce basic notions from graph theory and graph types.

### 2.1. Basic Graph Theory Concepts

In mathematics, graph theory is the study of graphs, which are mathematical structures used to model pairwise relations between objects. A graph in this context is made up of vertices (also called nodes or points) which are connected by edges (also called links or lines). A distinction is made between undirected graphs, where edges link two vertices symmetrically, and directed graphs, where edges link two vertices asymmetrically. Graphs are one of the principal objects of study in discrete mathematics.[1]

An undirected graph, or simply a graph  $G = (V, E)$  is a finite, nonempty set  $V(G)$ , together a set  $E(G)$  of unordered two-element subsets of  $V(G)$ . The elements of  $V$  are called vertices and  $E$  are called edges. The number of vertices in a graph  $G$  is called the size of  $G$ , denoted by  $|V(G)|$ . If the unordered pair  $e = \{u, v\}$  is an edge of the graph  $G$ , it is said that the vertices  $u$  and  $v$  are adjacent in  $G$  and that the edge  $e$  joins  $u$  and  $v$  in  $G$ . The edge  $e$  is said to be incident with the vertices  $u$  and  $v$ . The vertex  $u$  is also said to be a neighbour of  $v$  and vice versa.[2]

#### 2.1.1. Subdivision, Planarity and Thickness

A subdivision of a graph  $G$  is a graph  $H$  that is obtained by replacing the edges  $G$  with paths. The vertices in the path which are not end-vertices, are called subdivision vertices. Each subdivision vertex has a degree of 2. The graph  $G_7$ , a drawing of which is shown in Figure 2.1(b) is a subdivision of the graph  $K_5$  which is shown in Figure 2.1(a).

A graph  $G$  is said to be planar if it is possible to draw  $G$  in the plane, so that no curves representing edges  $G$  in the drawing intersect or cross one another, and such that no curves intersect the points representing the vertices of  $G$ . An intersection of a pair of edges is called crossing. A drawing of a planar graph  $G$  achieves no edge crossings is called plane drawing of  $G$ .

A graph  $G$  is called to be outerplanar if it is possible to construct a plane drawing of  $G$  in which all vertices of  $G$  are present in a single region of the drawing.

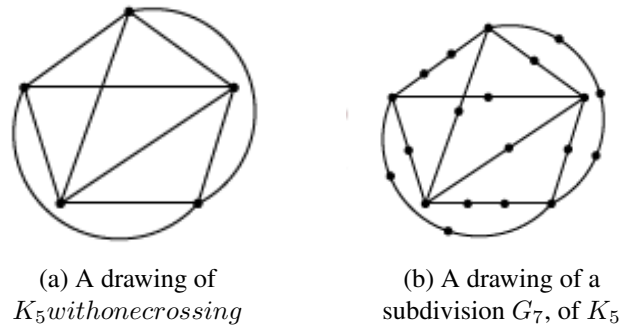


Figure 2.1: Illustration of graph subdivision

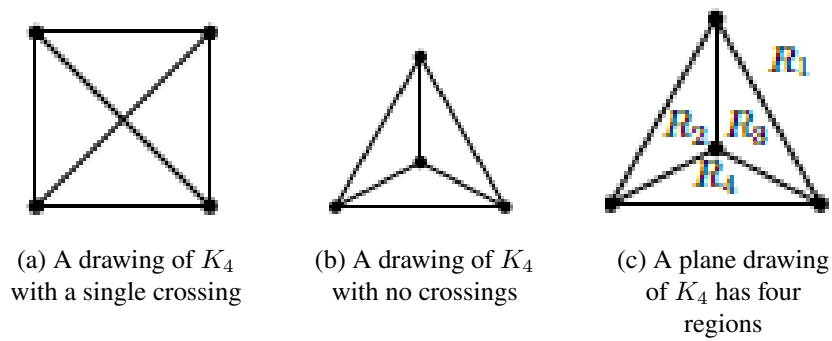


Figure 2.2: A planar graph permits plane drawings although such a graph can still be drawn with crossings.

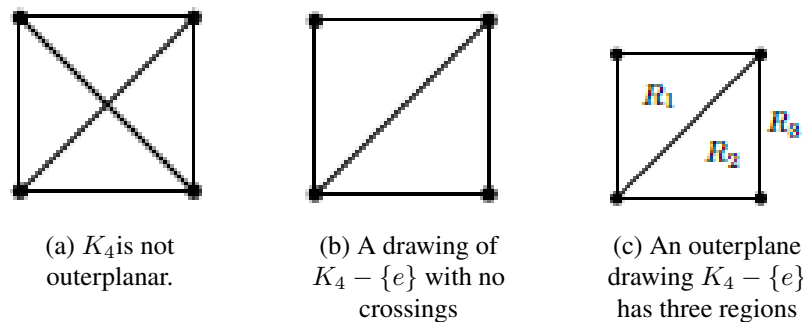


Figure 2.3: An outerplanar graph permits the construction of a plane drawing in which all vertices are present in the same region.

## 2.2. Graph Types

There are various types of graphs depending upon the number of vertices, number of edges, interconnectivity, and their overall structure. In this section, some graph types that I work on codes and that I frequently encounter during literature research will be explained.

### 2.2.1. Null Graph

A graph having no edges is called a Null Graph.

### 2.2.2. Trivial Graph

A graph with only one vertex is called a Trivial Graph.

### 2.2.3. Non-Directed Graph

A non-directed graph contains edges but the edges are not directed ones.

### 2.2.4. Directed Graph

In a directed graph, each edge has a direction.

### 2.2.5. Complete Graph

A simple graph with 'n' mutual vertices is called a complete graph and it is denoted by ' $K_n$ '. In the graph, a vertex should have edges with all other vertices, then it called a complete graph.

In other words, if a vertex is connected to all other vertices in a graph, then it is called a complete graph.

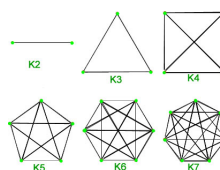


Figure 2.4: Complete graph examples.

### 2.2.6. Cycle Graph

A simple graph with 'n' vertices ( $n \geq 3$ ) and 'n' edges is called a cycle graph if all its edges form a cycle of length 'n'.

If the degree of each vertex in the graph is two, then it is called a Cycle Graph.

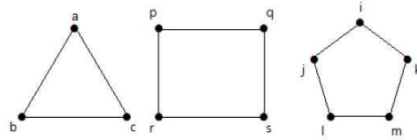


Figure 2.5: Cycle graph examples.

### 2.2.7. Bipartite Graph

A simple graph  $G = (V, E)$  with vertex partition  $V = \{V1, V2\}$  is called a bipartite graph if every edge of E joins a vertex in V1 to a vertex in V2.

In general, a Bipartite graph has two sets of vertices, let us say, V1 and V2, and if an edge is drawn, it should connect any vertex in set V1 to any vertex in set V2.

### 2.2.8. Complete Bipartite Graph

A bipartite graph  $G, G = (V, E)$  with partition  $V = \{V1, V2\}$  is said to be a complete bipartite graph if every vertex in V1 is connected to every vertex of V2.

In general, a complete bipartite graph connects each vertex from set V1 to each vertex from set V2.

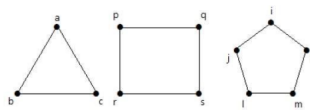


Figure 2.6: An example of complete bipartite graph.



### 3. LITERATURE RESEARCH

In this section, the general algorithms worked on within the scope of the project will be examined. First, the definition of the minimum crossing number problem will be made, and then the heuristic approaches that I researched throughout the semester will be presented.

#### 3.1. Minimum Crossing Number Problem

The study of crossing number of a graph in the plane forms part of larger field of study known as topological graph theory. This field primarily concerned with the representation of graphs in spaces that are locally 2-dimensional.

The basic operation of interest, is a formalization of the notion of a drawing of a graph, which is a mapping of its vertices to points in  $R^2$ , and of its edges to continuous curves in  $R^2$ , so that images of edges intersect a finite number of times and they are nowhere tangential.

Commonly used notions, derived from the notion of a graph drawing, such as embeddings (drawings of graphs without intersecting edges) and book drawings (drawings of graphs where vertices fall on the spine of a book and each edge is drawn on only a single page ) are also considered, as they are important in the modern theory of the crossing number of a graph.

##### 3.1.1. Drawings and Embeddings

The concept of graph drawing must be made precise if one is to investigate the crossing number problem. It cannot be overstated just how important it is to be completely specific and unambiguous about the definition of a graph drawing, since various researchers have apparently considered a number of subtly different interpretations, due to a lack of what exactly constitutes a valid graph drawing.



Figure 3.1: Crossings in self-crossing edges can always be removed. The graph in the left contains an edge  $e$  crosses itself. This self-crossing has been removed in right.

The simplest way to map a vertex into the plane is to associate it with a single point in  $R^2$ . Associating a vertex with a set in  $R^2$  gives no advantage over the single point definition, and certainly places restrictions over the exterior bounds into which the vertex may be embedded.

Besides drawings in normal form and in single-cross normal form, which still permit a large degree of freedom with respect to the placement of vertices and formation of edges, a number of other, restricted types of drawings occur in the literature.

### 3.1.1.1. Book Drawings

An  $n$ -page book is the union of  $n$ - half planes. The half planes intersect exactly on their finite boundaries, and this line is called the spine of the book. The half planes are the pages of the book.

There are two variants of book drawings; combinatorial and topological book drawings. In both, the vertices of a graph are placed on the spine of a book. In a topological book drawing, edges are not allowed to cross the spine, but in combinatorial book drawings, each edge must be drawn wholly on a single page of the book. Both drawings are in normal form.



Figure 3.2: A topological and combinatorial book embedding.

#### Computation of the crossing number for a given drawing

For a graph  $G$ , and a vertex  $v_i \in V(G)$ , let  $i$  be the index of  $v_i$ , and let the vertices be placed in increasing order of their indices on the spine of a book. Any four vertices  $v_i, v_l, v_k$  and  $v_l$  are said to be alternating on the spine if  $i < j < k < l$ .

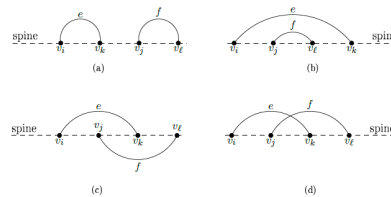


Figure 3.3: The possible configurations of two edges in a book.

For two edges  $e = \{v_i, v_k\}, f = \{v_j, v_l\} \in E(G)$  where  $i < k, j < l, i < j$ , the crossing possibilities are shown in Figure 3.3. In parts a and b of the figure, the four vertices are not alternating, and it is clear that the edges  $e$  and  $f$  can never cross. In part c, although they alternate, the edges  $e$  and  $f$  are on different pages, they can not cross. However, if the vertices are alternating, and edges  $e$  and  $f$  are on the same page, then a crossing must occur, this is shown on figure in part d.

#### Enumeration of different book drawings for a given graph

Clearly, the only factors that have an impact on whether two edges cross, are:

- the relative orderings of their end vertices
- the pages on which the edges are drawn

Thus, by permuting the vertices on the spine, and by choosing different pages for edges, different drawings may be generated.

#### 3.1.1.2. Circular Drawings

It is seen that circular drawings are sometimes used in the literature, and it is important to note that such drawings are equivalent to two-page book layouts. To illustrate the equivalence, vertices are drawn around the length of the circumference of a circle in the order given by the spine of a book layout, and edges are drawn either wholly inside the circle or wholly outside of it.

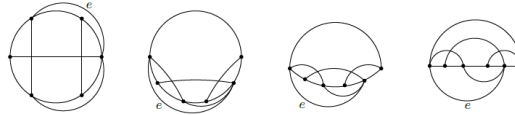


Figure 3.4: A book layout on two pages is equivalent to a circular layout.

#### 3.1.1.3. Embeddings

An embedding is drawing of a graph in which no pair of edges contains any mutual intersections. The simplest example is a plane drawing of a planar graph. It is seen that theory of embeddings provides a vocabulary within which to describe certain types of drawings or qualities of drawings.

### 3.1.2. Defining the Crossing Number of a Graph

The crossing number of a graph in  $R^2$  is the smallest number of crossings with which the graph can be drawn, when considering all drawings in single-cross normal form of the graph in  $R^2$ . Formally, this is defined as;

For any drawing  $\phi$  in single-cross normal form, of a graph  $G$ , define an indicator function

$$x(e, f, \phi) = \begin{cases} 1 & \text{if } |\phi((\tilde{e}) \cap \phi(\tilde{f}))| = 1 \text{ and } e \neq f \\ 0 & \text{otherwise} \end{cases} \quad (3.1)$$

The crossing number of  $G$  is realized by the drawing  $\phi$ , is

$$v_\phi(G) = \frac{1}{2} \sum_{e, f \in E(G)} x(e, f, \phi), \quad (3.2)$$

whilst the crossing number of  $G$  in  $R^2$  is defined as

$$v_{R^2}(G) = \min_{\phi(G)} \{v_\phi(G)\} \quad (3.3)$$

The factor of 1/2 in (3.1) is due to fact that the sum is taken over all ordered pairs of edges, and therefore each is counted twice. [3] [4]

If interested readers would like to read the definitions for crossing numbers based on types of drawings such as book drawing, circle drawing and other types of drawings:[5] [6]

And if you want to know about variants of crossing number parameters, please see here: [7] [5]

### 3.1.3. Parameters Related to the Crossing Number

Since the crossing number of a graph is a measure of its non-planarity, one may ask whether other non-planarity measures might be of aid when studying crossing number problem. I will try to define a number of planarity related parameters.

#### 3.1.3.1. The maximum induced planar subgraph problem

Given a graph  $G = (V, E)$  and a positive integer  $k \leq |V|$ , the maximum induced planar subgraph problem may be formulated as : "Is there a subset  $V' \subseteq V$  so that  $|V'| \geq k$ , with the property that  $\langle V' \rangle$  is planar?" For a lack of agreed upon notation, the largest value of  $k$  which the maximum induced planar subgraph problem may be

answered in the affirmative for the graph  $G$  is denoted  $m_v(G)$ .

### **3.1.3.2. The skewness of a graph**

The skewness of a graph  $G$  gives a measure of the minimum number of edges that would have to be removed from  $G$  in order to obtain a planar graph.

The maximum planar subgraph problem is related to the crossing number problem in the sense that, by removing appropriate edges from a drawing realizing the crossing number for a graph necessarily contain a maximum planar subgraph at the removal of the appropriate edges .

### **3.1.3.3. The vertex splitting number**

The operation of vertex splitting is a dual of the operations of edge contraction and edge deletion, which are performed when constructing a minor of a graph.

Applications of the correct splitting operations to a non-planar graph may be used to obtain a new planar graph.

### **3.1.3.4. The page number of a graph**

The page number problem, which is the problem of determining the minimum number of book pages required for a crossing-free book drawing of a graph, has applications in field of electronic circuit design [8].

## **3.2. Computational Methods**

As with most NP-complete graph parameters, algorithms for crossing number problem come in two flavours: brute force exact algorithms and heuristic approximation algorithms. The former is typically of theoretical importance, and for application to small graphs, whilst the latter is the only viable solution for all but the smallest graphs.

### **3.2.1. Brute Force (exact) Algorithms**

During the literature research for minimum crossing number problem, only two brute force algorithms founded. Both algorithms, finding the asymptotic running time has proven to be very difficult indeed - it is likely that this running time depends not only on the number of vertices and edges of the input graph, but on the structure of

graph itself.

### 3.2.1.1. The Garey-Johnson Algorithm

The algorithm derives its name from Garey and Johnson who demonstrated that determining the crossing number for a bipartite graph in the plane is an NP-complete problem.[9] It follows that since most graphs contain relatively large bipartite subgraphs, determining their plane crossing numbers in general, an NP-complete problem.

Garey and Johnson provided no details on how to implement this algorithm but later a pseudocode will be given in this report. Given a set of edge pairs  $P$ , where the edge permutations are given by  $X_e^1, X_e^2, \dots, X_e^l$ , the time required to enumerate all such permutations is  $O(X_e^1 x X_e^2 x \dots x X_e^l)$ . However, the total running time of the algorithm is difficult to estimate, since there seems to be no simple way of determining an expression that will represent the cumulative running time of all edge sets  $P$ . In the worst case, all  $k$  crossings occur on a single edge, resulting in a running time of  $O(k!)$ . Now,  $k$  pairs of edges are chosen for each set  $P$  the running time of the algorithm may then be expressed as  $O(k! \binom{q}{k})$ .

### 3.2.1.2. The Harris-Harris Algorithm

This algorithm is named after F.C. Harris Jr. and C.R. Harris, for their proposal [10] of a brute force method for finding the crossing number of a graph. The idea was to choose a clockwise ordering  $\Pi(G) = (\pi_1, \pi_2, \dots, \pi_v(G))$  of the edges around the vertices of a graph  $G$ , where  $\pi_i = (e_a, e_b, \dots, e_t)$  corresponds to the ordering of the edges incident to  $v_i \in V(G)$

Harris-Harris algorithm procedure has to repeated for every possible ordering  $\Pi$ . The total number of these permutations is of the order  $O(|G(V)||V(G)!|)$  for complete and bipartite graphs, since the degree of each vertex in such a graph is of order  $O(|V(G)|)$ , meaning that there are  $O(|V(G)!|)$  orderings of each edge around each vertex.

Thus, the Harris-Harris algorithm has the problem of worst case complexity explosions at two levels, and it seems, at least at a superficial level, that the Garey-Johnson algorithm does less work.

## 4. EXAMINED APPROACHES

The aim of this section is to give coding explanations of the optimal and heuristic approaches examined throughout the term and to share the pseudocodes of some of them. First you can find the review of the Garey-Johnson algorithm. Afterwards, you will briefly see some metaheuristic approaches. Finally, I will make an introduction to two algorithms that I have been working on in my project. We will examine these algorithms, which were developed and tested in code, in more detail in the next chapter.

### 4.1. Garey-Johnson Algorithm

As mentioned in chapter 3.2.1.1 Garey Johnson algorithm used to determine whether a crossing number of a graph is smaller or equal to given number. I will explain the code flow for this algorithm in this section.

Algorithm itself is divided into 3 major parts. The implementation actually contains Garey Johnson, which is the starting point, enumerates all  $k$  pairs of crossing edges ( $k$  is the number of crossings test for), Test Planar enumerates all permutations over each edge for the given set of  $k$  crossings. In its turn, for each set of permutations, it calls Construct Graph algorithm which constructs a new graph from the input graph where the crossings that are specified by the permutations of edges pairs are represented as artificial vertices. The algorithm Test Planar then determines whether this graph is planar.

#### Pseudocode for Garey-Johnson Algorithm

The code takes a graph  $G$  as input and gives the output as: The crossing number  $cn(G)$  of  $G$ , and a drawing realising  $cn(G)$  where crossings are modeled as artificial vertices of degree 4.

```
1: for all  $C \in \{(p_1, p_2, \dots, p_k) : (p_1, p_2, \dots, p_k) \text{ is a set of } k \text{ pairs of edges of } G\}$  do
2:    $D \leftarrow \emptyset$ 
3:   for all  $e \in E(G)$  do
4:      $L_e \leftarrow \emptyset$ 
5:   end for
6:   for all  $(e, f) \in C$  do
7:      $L_e \leftarrow L_e \cup \{f\}$ 
8:      $L_f \leftarrow L_f \cup \{e\}$ 
9:      $D \leftarrow D \cup \{e\}$ 
10:     $D \leftarrow D \cup \{f\}$ 
11:   end for
12:   if TestPlanar( $L, D, \text{FirstElement}(D), G$ ) then
13:     return TRUE
14:   end if
15: end for
16: return FALSE
```

Figure 4.1: Garey-Johnson

There are two data structures:  $L$  is a list of all lists, indexed by an edge  $e$ , records all edges that cross  $e$ , and a list  $D$  contains the edges that are to be crossed given crossing configuration. First for loop enumerates all crossing configurations with  $k$  crossings. The list  $D$  is cleared at line 2, so that maybe it filled with crossed edges for the current iteration of the loop. Same thing is done for every list in  $L$ . Then  $D$  and  $L$  are filled with crossing information needed later in the algorithm. All permutations of edge crossing for the given crossing configuration are enumerated by algorithm Test Planar, it is called in line 12. The function FirstElement( $D$ ), returns the first element in the list  $D$ . If TestPlanar algorithm finds a set of crossing permutations for graph  $G$  has  $k$  crossings, it returns boolean value TRUE. If after enumerating all possible configurations of  $k$  crossings, it is not possible to draw  $G$  with  $k$  crossings then return value will be FALSE.

### Pseudocode for Test-Planar Algorithm

**Input:** A vector of edge sets  $L$  which encodes the edge crossing set, a list of crossed edges  $D$ , the graph  $G$  of which  $L$  is the crossing set, and the current edge  $e$  of which the order of edge crossings have to be permuted.

**Output:** TRUE if a set of permutations for the current edge crossings set was found, FALSE otherwise.

```

1:  $L_e \leftarrow \text{FirstPermutation}(L_e)$ 
2: loop
3:   if HasNextElement( $D, e$ ) then
4:     if TestPlanar( $L, D, \text{NextElement}(D, e), G$ ) = TRUE then
5:       return TRUE
6:     end if
7:   else
8:      $G' \leftarrow \text{ConstructGraph}(L, D, G)$ 
9:     if IsPlanar( $G'$ ) then
10:      return TRUE
11:    end if
12:   end if
13:   if  $L_e \neq \text{LastPermutation}(L_e)$  then
14:      $L_e \leftarrow \text{NextPermutation}(L_e)$ 
15:   else
16:     return FALSE
17:   end if
18: end loop

```




Figure 4.2: Test Planar Algorithm

With  $L$  and  $D$  TestPlanar generates every possible permutation of crossings for every edge, and determines whether such configuration can manifest a drawing with  $k$  crossings, by constructing a new graph, and by testing this graph for planarity. The distinct sets of permutations are generated recursively, which has effect of producing a lexicographical ordering of the sets of edges in  $L$ . Every permutation of the list  $L_e$  for an edge  $e$  in the list of crossed edges  $D$  is generated, and for each such step all possible permutations of crossings for all edges after  $e$  in  $D$  are generated.(recursively) When no more edges remain, the feasibility of the crossing configuration is determined for the set of permutations.



## Pseudocode for ConstructGraph Algorithm

**Input:** A graph  $\mathcal{G}$ , a list of crossed edges  $\mathbf{D}$ , and the crossing configurations for the edges  $\mathbf{L}$ .  
**Output:** The graph  $\mathcal{G}$ , modified so that the crossings from  $\mathbf{D}$  are present as artificial vertices of degree four, in the order given for the edges in  $\mathbf{L}$ .

```

1:  $\phi \leftarrow$  a bijection of the edge pairs in  $\mathbf{D}$  to  $k$  new vertices in  $\mathcal{G}$ 
2: for all  $e = \{u, v\} \in \mathbf{D}$  do
3:    $x \leftarrow u$ 
4:   for all  $\{e, f\} \in \mathbf{L}_e$  do
5:      $w \leftarrow \phi(\{e, f\})$ 
6:      $E(\mathcal{G}) \leftarrow E(\mathcal{G}) \cup \{\{x, w\}\}$ 
7:      $x \leftarrow w$ 
8:   end for
9:    $E(\mathcal{G}) \leftarrow E(\mathcal{G}) \cup \{\{x, v\}\}$ 
10:   $E(\mathcal{G}) \leftarrow E(\mathcal{G}) \setminus \{e\}$ 
11: end for
12: return  $\mathcal{G}$ 

```

Figure 4.3: Construct Graph Algorithm

In order to test whether a given crossing is feasible, a graph must be constructed from  $G$  such that crossings are substituted for artificial vertices of degree four. If this graph is planar, then a planar layout of this graph is a drawing of  $G$  with  $k$  crossings.

## 4.2. Metaheuristic Algorithms for Minimum Crossing Number

The crossing numbers for basic graphs such as the complete graphs and complete bipartite graphs is still unknown. This problem has important applications such as printed circuit board layout, VLSI circuit routing, automated graph drawing and finding the degree of statical indeterminacy in structure, the problem involves placing the vertices of the graph along a horizontal “node line” in the plane and then adding edges as specified by the interconnection pattern. The objective of this problem is to embed the edges so that the total number of crossings is minimized.

I found four optimization algorithms during the studies on minimum crossing number problem. These are PSO, IRO, CBO, and ECBO. A brief overview of four optimization algorithms will be provided in the subsequent subsections. All of these methods are population-based algorithms that start with a set of randomly selected candidate solutions. According to a series of rules, mainly inspired from natural phenomena, the existing solutions are perturbed iteratively in order to improve their objective function values[11].

### 4.2.1. Particle swarm optimization (PSO)

The particle swarm optimization (PSO), as one of the successful stochastic optimization algorithms, is based on simulation of the social behavior of bird flocking and fish schooling that was introduced by Eberhart and Kennedy [12]. The PSO is a pop-

ulation based technique that involves a number of particles which represent the swarm being initialized randomly in the search space. Each particle represents a candidate solution of the optimum design problem and iteratively moves across the search space. During each generation, each particle updates its velocity and position by learning from the best position achieved so far by the particle itself (*pBest*) and the location of the best fitness achieved so far across the whole population (*gBest*). The general structure of a PSO algorithm is as follows:

```

procedure Particle Swarm Optimization (PSO)
  Initialize algorithm parameters
  for each particle
    Initial position is created randomly
    Fitness value is evaluated
  end for
  pBest and gBest are updated
  While maximum iterations is not fulfilled
    for each particle
      Velocity is updated
       $new\_position_i = current\_position_i + velocity_i$ 
    end for
    pBest and gBest are updated
  end while
end procedure

```

Figure 4.4: Pseudocode for PSO

#### 4.2.2. Improved ray optimization (IRO)

The ray optimization (RO) algorithm proposed by Kaveh and Khayatizad [13] [14] was conceptualized using the relationship between the angles of incidence and fraction based on Snell's law. In this method, each agent is modeled as a ray of light that moves in the search space in order to find the global or near-global optimum solution. Improved ray optimization (IRO), employed a new approach for generating new solution vectors which has no limitation on the number of variables, so in the process of algorithm there is no need to divide the variables into groups like RO. In order to generate new solution vectors in the IRO, dynamic parameters are utilized that make a better balance between exploration and exploitation. The procedure which returns the violated agents into feasible search space is also modified in IRO. Instead of changing all the components of violating agents, only the components that violate the boundary are refunded. As a pseudo code, the IRO method has the following form:

```

procedure Improved Ray Optimization (IRO)
  Initialize algorithm parameters
  for each agent
    Initial position and movement vector are created randomly
    Fitness value is evaluated
  end for
  /* LBM and GB are local best memory and global best, respectively*/
  LBM and GB are updated
  While maximum iterations is not fulfilled
    for each agent
       $new\_position_i = current\_position_i + movement\_vector_i$ 
      Violated components are regenerated
    end for
    LBM and GB are updated
    for each agent
      The direction of movement vector is calculated
      The magnitude of movement vector is calculated
    end for
  end while
end procedure

```

Figure 4.5: Pseudocode for IRO

### 4.2.3. Colliding bodies optimization (CBO) and its enhanced version (ECBO)

As a newly developed type of meta-heuristic algorithm, CBO is a population-based stochastic optimization algorithm based on the governing laws of one dimensional collision between two bodies from the physics. Each agent is modeled as a colliding body (CB) with a specified mass and velocity. One object collides with other object and they move toward minimum energy level.

```

procedure Colliding Bodies Optimization (CBO)
  Initialize algorithm parameters
  for each CB
    Initial position is created randomly
  end for
  While maximum iterations is not fulfilled
    for each CB
      Fitness value is evaluated
      The value of mass is calculated
    end for
    Stationary and moving groups are created
    for each CB
      The velocity before collision is calculated
      The velocity after collision is calculated
      Position is updated
    end for
  end while
end procedure

```

Figure 4.6: Pseudocode for CBO

The enhanced colliding bodies optimization (ECBO) is introduced in order to improve the exploration capabilities of the CBO and to prevent premature convergence, a stochastic approach is employed in ECBO that changes some components of CBs randomly. Colliding memory (CM) is also considered to save some historically best CB vectors and their related mass and objective function values to improve the performance of the CBO and reduce the computational cost.

```

procedure Enhanced Colliding Bodies Optimization (ECBO)
  Initialize algorithm parameters
  for each CB
    Initial position is created randomly
  end for
  While maximum iterations is not fulfilled
    for each CB
      Fitness value is evaluated
      The value of mass is calculated
    end for
    CM is updated
    Population is updated
    Stationary and moving groups are created
    for each CB
      The velocity before collision is calculated
      The velocity after collision is calculated
      Position is updated
    end for
  end while
end procedure

```

Figure 4.7: Pseudocode for ECBO

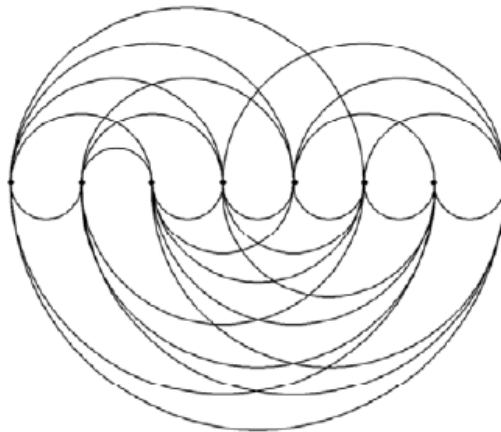


Figure 4.8: Embedding of the complete graph  $K_8$  obtained by ECBO

## 4.3. Algorithms Used in the Project

In this section, I will briefly introduce the algorithms I use in my project to compare results.

The first of these is the Quick Cross algorithm, and the second is the heuristic algorithm that I have created with the inferences I have obtained from all the methods mentioned in this chapter and the literature research section.

In the Quick Cross, the method is based upon repeatedly solving the so-called star insertion problem in the setting where the combinatorial embedding is fixed, and has several desirable characteristics for practical use.

In the heuristic algorithm I wrote, I tried to obtain a smaller crossing number by trying to reconstruct the edges of the graph given by the planarization method, to summarize simply. This approach tries to add as much edges as it can while preserving planarity. At the same time, in order to observe the change of the crossing number depending on the number of iterations, I tried to observe the effect on the result by leaving the number of iterations dependent on the user input.

### 4.3.1. Quick Cross Algorithm

While the philosophy of the planarisation method is to start with a planar subgraph and increase the number of crossings at each iteration as the full graph is rebuilt, Quick Cross approach works in the opposite direction; it starts with a combinatorial embedding corresponding to a, presumably suboptimal, drawing of the full graph and at each iteration, attempts to find a combinatorial embedding corresponding to a drawing with fewer crossings. Unlike the planarisation method, Quick Cross does not require a planar subgraph to be computed. Instead it relies upon iteratively solving the star insertion problem in a combinatorial embedding which corresponds to the current (non-planar) drawing of  $G$ .

The Quick Cross focus on a fixed embedding at each iteration, so the initial combinatorial embedding of  $G$  plays a large role in the performance of the algorithm. Algorithm computes initial embeddings with three possibilities. The first method produces an embedding quickly, however the initial number of crossings can be as large as  $\binom{n}{4}$ . The second method is slower to compute but the initial number of crossings is usually much smaller for the case of sparse graphs. The third method is an implementation of a force-directed graph drawing algorithm. These three initial embedding schemes are referred as circle, planar and spring.

The algorithm runs in  $O((k + n)m)$  time where  $k$  is the number of crossings in the drawing associated with the current embedding of  $G$ .

Quick Cross algorithm consists of three main procedures. First one is the main loop its inputs are a combinatorial embedding  $\Pi$  corresponding to some initial drawing  $D$  of  $G$ . Second one is vertex deletion procedure. Given a dual graph  $\Pi^*$  and a vertex  $v$  of  $G$ , this performs edge contractions. Third one is star insertion problem solver. Given a dual graph  $\Pi^*$  along with a vertex  $v$  of  $G$ , this performs edge contractions in the dual graph. Then the fixed embedding version of the star insertion problem is solved for  $v$ .

---

```

1: procedure MAIN LOOP
2:   current cr  $\leftarrow cr_D(G)$ 
3:   while true do
4:     improvement found  $\leftarrow false$ 
5:     Find the faces of  $\Pi$ .
6:      $\Pi_0^* \leftarrow$  dual graph of  $\Pi$ 
7:     for  $v \in V(G)$  do
8:        $\Pi^* \leftarrow \Pi_0^*$  (make a copy of  $\Pi_0^*$ )
9:        $\Pi^* \leftarrow REMOVE(G, \Pi^*, v)$ 
10:      (new cr, newface, shortest paths)  $\leftarrow SIP(G, \Pi^*, v)$ 
11:      if new cr < current cr then
12:        improvement found  $\leftarrow true$ 
13:        break
14:      end if
15:    end for
16:    if improvement found then
17:      Update  $\Pi$  to reflect new placement using newface and shortest paths.
18:      Check if any subdivisions are needed.
19:      Check if any previous subdivisions can be removed.
20:      current cr  $\leftarrow$  new cr
21:      continue
22:    else
23:      break
24:    end if
25:  end while
26:  return (current cr,  $\Pi$ )
27: end procedure

```

Figure 4.9: Main Loop Pseudocode

```

1: procedure REMOVE( $G, \Pi^*, v$ )
2:   for  $e^* \in E(\Pi^*)$  do
3:     if  $e^*$  corresponds to an edge of  $G$  which is incident to  $v$  then
4:       Contract  $e^*$ .
5:     end if
6:   end for
7:   return ( $\Pi^*$ )
8: end procedure

```

Figure 4.10: Remove Procedure Pseudocode

```

1: procedure SIP( $G, \Pi^*, v$ )
2:   for  $w \in N_G(v)$  do
3:      $\Pi^{**} \leftarrow \Pi^*$  (make a copy of  $\Pi^*$ )
4:     In  $\Pi^{**}$ , contract the cycle formed by dual edges corresponding to
       edges incident to  $w$  in  $\Pi$ , call the contracted vertex  $w_d$ .
5:      $dist_w \leftarrow$  Shortest path algorithm( $\Pi^{**}, w_d$ ).
6:     Set the dist of vertices contracted to form  $w_d$  to zero.
7:   end for
8:   newface  $\leftarrow \operatorname{argmin}_k (\sum_{w \in N(v)} dist_w(k))$ 
9:   shortest paths  $\leftarrow$  Shortest path algorithm( $\Pi^*, newface$ )
10:  return (new cr, newface, shortest paths)
11: end procedure

```

Figure 4.11: Star Insertion Pseudocode

### 4.3.2. Proposed Algorithm

In the proposed heuristic algorithm, like the other heuristic algorithms examined, the graph planarization method was used with given input graph  $G$ . Simply, the edges of the given graph are simplified by the embedding method and a new graph is created. Depending on the number of iterations, the numbers of edges failed to embed and edges succeeded to embed into a list are kept here. With these variables, the number of crossing numbers was reached by using the method that calculates the crossing number in the main procedure. I will explain the methods mentioned here in the following sections.

The program that runs proposed algorithm consists of three programs that written in C++ language.

The main file named main.cpp takes a filename as input and assigns file contents into a edges list. Main file then, uses two general for loop to decide minimum crossing number. The first loop iterates through edges and takes record to edges that failed to embed list. The second for loop iterates as iteration number taken from the user, calls getCrossingNumber method and calculates the minimum crossing number.

The file named greedy.cpp includes helper functions for embedding edges, this program uses greedy approach. tryToAddEdge helper method adds if edge doesn't affect on graph planarity, otherwise it won't. Returns wheter failed or not to add an edge. tryToEmbed method tries to add as much edges as it can while preserving planarity. Returns the number of edges which were unable to add. Another helper method finds smallest the number of edges failed to embed. The method named get crossing number uses all variables that comes from above helper functions, and tries to find minimum crossing number for given graph. First, it subtracts the number of failed embedding from the vertex number of the input graph and creates a graph with the help of the add edge method. It then loops through the edgesFailedToEmbed list and traverses the faces of this graph. It adjusts the number of crossing numbers when it finds conflict during roaming.

## 5. RUNTIME AND IMPLEMENTATION

In this chapter we will look details about proposed algorithm implementation and runtime by looking at written code blocks.

The main file named main.cpp has a general loop and this loop iterates the graph as number as iteration counts. To calculate complexity of this loop, we need to find complexities of function used in the loop.

```
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
    for(int i = 0; i < iterationsCount; i++)
    {
        ebegin = clock();
        edgesFailedToEmbedCount = getEFTEC(edgesList, edgesCount,
        edgesFailedToEmbedList, edgesSucceedToEmbed, greedyIterations);
        eend = clock();

        cbegin = clock();
        int cr = getCrossingNumber(edgesSucceedToEmbed, edgesCount,
        edgesFailedToEmbedList, edgesFailedToEmbedCount, vertexCount);
        cend = clock();
        if(cr < minCr)
        {
            minCr = cr;
            minEFTEC = edgesFailedToEmbedCount;
        }

        sumCr += cr;
    }
    //END MAIN BODY
    end = clock();
```

Figure 5.1: Main loop in the program

In the main loop there is a function call to getEFTEC in line 105 and another function call to getCrossingNumber in line 110. So we first look these functions time complexity.

Get edges to failed to embed count function has a for loop inside. This loop actually determines the function's complexity. When we call  $n$  to the edges failed to embed, and  $m$  to the edge count; this function has complexity of  $O(nm)$ .

```
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
    for(int i = 0; i < iterations; i++) //figure out how many iterations you need
    {
        graph0 theGraph = graph0(0);
        shuffleEdges(edgesList, edgesCount);
        edgesFailedToEmbed = tryToEmbed(theGraph, edgesList, edgesCount, failedToEmbedIndexes);
        if(edgesFailedToEmbed < minEdgesFailedToEmbed)
        {
            for(int j = 0; j < edgesFailedToEmbed; j++)
            {
                edgesFailedToEmbedList[j][0] = edgesList[failedToEmbedIndexes[j]][0];
                edgesFailedToEmbedList[j][1] = edgesList[failedToEmbedIndexes[j]][1];
            }
            edgesSucceedToEmbed->clear();
            for(int j = 0; j < edgesCount; j++)
            {
                bool found = false;
                for(int k = 0; k < edgesFailedToEmbed; k++)
                {
                    if(failedToEmbedIndexes[k] == j)
                    {
                        found = true;
                        break;
                    }
                }
                if(!found)
                {
                    edgesSucceedToEmbed->push_back(std::pair<int, int>(edgesList[j][0], edgesList[j][1]));
                }
            }
            minEdgesFailedToEmbed = edgesFailedToEmbed;
        }
    }
```

Figure 5.2: For loop in the function that determines edges failed to embed.



```

247     for (std::vector<int>::iterator it = startFaces.begin(); it != startFaces.end(); ++it)
248     {
249         dualGraph[*it].isClosed = true;
250         deq.push_back(dualGraph[*it]);
251         map[*it] = -1;
252     }
253
254     bool found = false;
255     /deq/
256     while(!deq.empty())
257     {
258         Face element = deq.front();
259         deq.pop_front();
260
261         for (std::vector<int>::iterator it = endFaces.begin(); it != endFaces.end(); ++it)
262         {
263             if(*it == element.id)
264             {
265                 found = true;
266                 break;
267             }
268         }
269
270         if(found)
271         {
272             faces_list_path = backtrace(map, element.id, dualGraph);
273             cr = path.size() - 1;
274             planarize_path(dualGraph, edgesFailedToEmbedList[i], &path, &vertexCount);
275             break;
276         }
277     }

```

Figure 5.3: Part of get crossing number function.

Another function that calculates crossing number, named `getCrossingNumber`, has loops and if conditions inside. We need to calculate its complexity too. This function has two for loops and one while loop inside. And one of the for loops has an inner for loop. The first loop iterates as count of edge number - edges failed to embed count. Its complexity is  $O(n - m)$ . Other for loop has complexity  $O(n \cdot (n - m))$  because it loops through edges failed to embed count times with an inner for loop that iterates  $n - m$ , so the complexity of get crossing number method is  $O(n^2 - nm)$  which can be simplified to  $O(n^2)$ .

Now, we have two function calls with complexity  $O(n^2)$  and  $O(nm)$ . And the main procedure has time complexity of  $O((n^2) \cdot k)$  where  $k$  is the number of iterations.

## 6. EXPERIMENTAL RESULTS

In this chapter, some experimental results will be given in order to see the behavior of the compared algorithms in the crossing number problem. I will refer to both the results examined during the literature research and the comparisons I made, respectively.

Since I explained how the codes work in the previous chapter, I will show how the codes are run in the terminal in a section here.

First of all, we will see how the metaheuristic algorithms described in the examined approaches section perform on complete bipartite graphs. Later, I will share the results of Quick Cross and the proposed algorithm for the same graphs. I made my own comparisons using complete graphs and complete bipartite graphs.

### 6.1. Generating Random Graphs and Running Algorithm

In this section, I will give a short explanation about how I created my test graphs, and how to run the program that runs proposed algorithm.

A python script is written to construct an input file. This program takes edge number and vertice number as an input and creates a file that represents graphs. Files created by the program named as 'vertice' x 'edge'.txt. Then main program takes the file name as input. Example for generating complete bipartite graphs; to generate  $K_{n,m}$  graph user should enter  $n + m$  as vertex number and  $n.m$  for edge number.

Below, you can see created test file examples:

```
esra@ubuntu:~/Desktop/mincrossing/min_cross_num_heuristic/generators$ python Gnn.py 20 9
esra@ubuntu:~/Desktop/mincrossing/min_cross_num_heuristic/generators$ python Gnn.py 10 10
esra@ubuntu:~/Desktop/mincrossing/min_cross_num_heuristic/generators$ python Gnn.py 13 13
esra@ubuntu:~/Desktop/mincrossing/min_cross_num_heuristic/generators$ python Gnn.py 50 25
esra@ubuntu:~/Desktop/mincrossing/min_cross_num_heuristic/generators$ python Gnn.py 100 40
esra@ubuntu:~/Desktop/mincrossing/min_cross_num_heuristic/generators$ python Gnn.py 10000 200
esra@ubuntu:~/Desktop/mincrossing/min_cross_num_heuristic/generators$
```

Figure 6.1: Generating random graphs by given edge and vertex number.

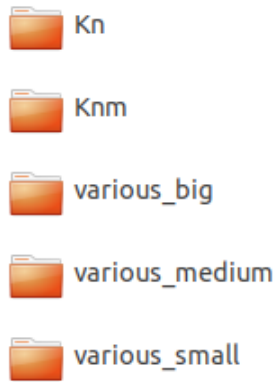


Figure 6.2: Directories created for test graph files according to their sizes.

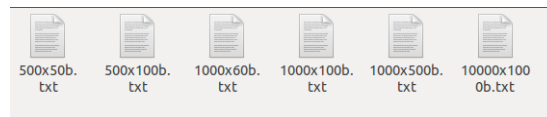


Figure 6.3: An example of generated graphs, figure shows big graphs.

The other issue I will talk about in this section is about how the program will work. I have prepared a makefile to run the program, it is possible for users to compile the program by typing make command in linux environment. After that, what is required should be the actual input file name. If the file name is not entered, the program will not run. In addition, if the user wants to write the number of iterations, time information, output results to the file as input arguments, an output file name can be entered, edges failed to embed count can be seen with entering the -e command.

```
time spent on EFTEC search - 32.04
time spent on crossings search - 0.02
time spent - 32.06
EFTEC - 1
average crossing number - 1
```

Figure 6.4: An example of generated output file with informations.

```

time spent on EFTE search - 0.000131
time spent on crossings search - 0.0021
time spent - 0.118434
EFTEC - 7
minimum crossing number - 20
average crossing number - 25.4
/home/.../Desktop/MinCrossing/Min_cross_num_heuristic/greedy$ ./greedy -f 45x10.txt -n -t -e -n 10 -d
time spent on EFTE search - 0.000731
time spent on crossings search - 0.000649
time spent - 0.0129
EFTEC - 9
minimum crossing number - 20
average crossing number - 20
/home/.../Desktop/MinCrossing/Min_cross_num_heuristic/greedy$ ./greedy -f 45x10.txt -n -t -e -d
time spent on EFTE search - 0.000013
time spent on crossings search - 0.007413
time spent - 0.015406
EFTEC - 18
minimum crossing number - 32
average crossing number - 32
/home/.../Desktop/MinCrossing/Min_cross_num_heuristic/greedy$ ./greedy -f 45x10.txt -n -t -e -l
24
/home/.../Desktop/MinCrossing/Min_cross_num_heuristic/greedy$ ./greedy -f 45x10.txt -n -t -e
time spent - 0.010209
EFTEC - 8
minimum crossing number - 22
average crossing number - 22
/home/.../Desktop/MinCrossing/Min_cross_num_heuristic/greedy$

```

Figure 6.5: A terminal output for graph  $K_{3,15}$  with commandline options.

## 6.2. Numerical Examples for Metaheuristic Algorithms

Four metaheuristic algorithms are considered for the fixed linear crossing number problem. In this problem, the vertices of a graph are placed in a fixed order along a horizontal “node line” in the plane, each edge is drawn as an arc in one of the two half-planes (pages), and the objective is to minimize the number of edge crossings. The analyzed metaheuristic algorithms (PSO, IRO, CBO and ECBO) were tested 20 times on complete and complete bipartite graphs. Six complete graphs ( $K_8, K_9, \dots, K_{13}$ ) and eight complete bipartite graphs ( $K_{3,10}, K_{3,15}, K_{4,5}, K_{4,10}, K_{4,15}, K_{5,5}, K_{5,10}, K_{5,15}$ ) are considered to verify the efficiency of the metaheuristic algorithms.

Table 1: Best results comparison on complete graphs

Graph	Optimum	Shahrokhi et al. [10]	Cimikowski and Shope [11]	Wang and Okazaki [9]	PSO	IRO	CBO	ECBO
$K_8$	18	19	18	18	18	18	18	18
$K_9$	36	36	36	36	36	36	36	36
$K_{10}$	60	62	60	60	60	60	60	60
$K_{11}$	100	100	100	100	104	100	100	100
$K_{12}$	150	154	150	150	158	153	151	150
$K_{13}$	-	265	225	225	245	227	231	225

Figure 6.6: Results for complete graphs compared with metaheuristic algorithms.

Table 3: Best results comparison on complete bipartite graphs

Graph	Optimum	PSO	IRO	CBO	ECBO
$K_{3,10}$	20	28	20	20	20
$K_{3,15}$	49	71	49	49	49
$K_{4,5}$	8	10	10	10	10
$K_{4,10}$	40	63	54	54	54
$K_{4,15}$	98	217	133	130	130
$K_{5,5}$	16	20	20	20	20
$K_{5,10}$	80	109	100	100	100
$K_{5,15}$	196	360	264	244	244

Figure 6.7: Results for complete bipartite graphs compared with metaheuristic algorithms.

The experimental results indicate that all algorithms nearly have an acceptable performance for the crossing number problem. ECBO yields near-optimal solutions and outperforms the other methods. It is superior over other methods in terms of reliability and solution accuracy. However, the performance of IRO and CBO is approximately the same and better than PSO. [15] [16]

I will examine how these results look compared to the Quick Cross algorithm and the proposed algorithm tested in the project in the next sections.

### **6.3. Comparisons for Quick Cross Algorithm and Proposed Algorithm**

In this section, the Quick Cross algorithm and the results of the proposed algorithm on complete and complete bipartite graphs will be shared with the readers.

For complete graphs, as seen in the literature,  $n$  numbers were generated 50 times, up to 15, with random graphs and the average values were recorded.

For Complete bipartite graphs,  $n_1$  and  $n_2$  numbers have been tested starting from 3. Here, too, each graph size has been tested 50 times.

### 6.3.1. Quick Cross Algorithm Complete Graph Results

Kn	Minimum Crossing Number
2	0
3	0
4	0
5	1
6	3
7	9
8	18
9	36
10	60
11	100
12	150
13	225
14	316
15	445

Figure 6.8: Results for complete graphs with Quick Cross algorithm.

### 6.3.2. Proposed Algorithm Complete Graph Results

When the output below is examined, you can see that the proposed algorithm finds more crossing numbers for some  $n$  values than Quick Cross. However, when I run the same  $n$  values again, this time giving 50 or 100 iterations, it achieves the same result as Quick Cross. And for value  $n$  11, proposed algorithm finds less crossing number than Quick Cross algorithm.

Kn	Cr in the first run	Cr with more iterations
2	0	0
3	0	0
4	0	0
5	1	1
6	3	3
7	9	9
8	18	18
9	36	36
10	63	60
11	104	98
12	154	150
13	225	225
14	316	316
15	446	445

Figure 6.9: Results for complete graphs with proposed algorithm.

I mentioned that by increasing the number of iterations in the Proposed algorithm, I could bring the minimum crossing number, which is more than Quick Cross, to the same number. The other issue I want to mention here is that for some  $n$  values, it is not necessary to give a large iteration number in the first place. As can be seen in the output below, when running 100 iterations for  $n$  9, it finds more crossing numbers than fewer iterations. According to my conclusion here using the number of iterations on graphs of larger sizes provides more efficient results than using for smaller size graphs.

```
esra@ubuntu:~/Desktop/mincrossing/min_cross_num_heuristic/greedy$ ./greedy -f k8.txt -t -m -n 10
time spent - 0.005197
minimum crossing number - 19
average crossing number - 19
esra@ubuntu:~/Desktop/mincrossing/min_cross_num_heuristic/greedy$ ./greedy -f k8.txt -t -m -n 50
time spent - 0.017604
minimum crossing number - 21
average crossing number - 21
esra@ubuntu:~/Desktop/mincrossing/min_cross_num_heuristic/greedy$ ./greedy -f k8.txt -t -m -n 100
time spent - 0.028599
minimum crossing number - 18
average crossing number - 18
esra@ubuntu:~/Desktop/mincrossing/min_cross_num_heuristic/greedy$ ./greedy -f k9.txt -t -m -n 100
time spent - 0.057498
minimum crossing number - 38
average crossing number - 38
esra@ubuntu:~/Desktop/mincrossing/min_cross_num_heuristic/greedy$ ./greedy -f k9.txt -t -m
time spent - 0.008516
minimum crossing number - 36
average crossing number - 36
```

Figure 6.10: Examining effect of iteration number on small size complete graphs.

When we compare the general result with the metaheuristic algorithms given in the previous sections, we see that the proposed algorithm for complete graphs gives better results than the metaheuristic algorithms, and gives the same result with the Quick Cross algorithm. In addition, the proposed algorithm, found the crossing numbers to be higher than Quick Cross in the first try for some  $n$  values in the complete graphs, but when the number of iterations increased, it reached a better solution for  $n$  11, and reached the same result for other values that were found bigger at the beginning.

### 6.3.3. Quick Cross Algorithm Complete Bipartite Graph Results

Testing complete bipartite graphs for both algorithms are made for various size of graphs. For a complete bipartite graph  $K_{n,m}$  I used values in a range  $3 < n < 20$  and  $3 < m < 20$  and run graphs 50 times. Below, in the figure 6.11 you can find crossing numbers for Quick Cross algorithm.

### 6.3.4. Proposed Algorithm Complete Bipartite Graph Results

Same graphs tested with proposed algorithm, and it has been seen that the proposed algorithm produces much more efficient results on complete bipartite graphs than the Quick Cross algorithm, without the need to increase the number of iterations. Figure 6.12

B	C
$K_{n,m}$	Quick Cross Cr
$K_{3,4}$	2
$K_{3,5}$	4
$K_{4,4}$	4
$K_{4,5}$	8
$K_{3,10}$	20
$K_{3,15}$	49
$K_{4,15}$	98
$K_{5,5}$	16
$K_{5,10}$	80
$K_{10,10}$	400
$K_{15,10}$	980
$K_{20,10}$	1800
$K_{20,20}$	8100

Figure 6.11: Minimum crossing numbers for Quick Cross algorithm for complete bipartite graphs.

F	G
$K_{n,m}$	Proposed Algorithm Cr
$K_{3,4}$	2
$K_{3,5}$	4
$K_{4,4}$	4
$K_{4,5}$	2
$K_{3,10}$	5
$K_{3,15}$	20
$K_{4,15}$	77
$K_{5,5}$	16
$K_{5,10}$	34
$K_{10,10}$	315
$K_{15,10}$	880
$K_{20,10}$	1380
$K_{20,20}$	7084

Figure 6.12: Minimum crossing numbers for proposed algorithm for complete bipartite graphs.



Kn,m	Quick Cross Cr	Proposed Algorithm Cr
K3,4	2	2
K3,5	4	4
K4,4	4	4
K4,5	8	2
K3,10	20	5
K3,15	49	20
K4,15	98	77
K5,5	16	16
K5,10	80	34
K10,10	400	315
K15,10	980	880
K20,10	1800	1380
K20,20	8100	7084

Figure 6.13: Complete bipartite graph results for both algorithms.

Looking at the results obtained for complete bipartite graphs, it can be seen that the metaheuristic algorithm results for small values are weak compared to Quick Cross and the proposed algorithm. The proposed algorithm achieved better results than the Quick Cross algorithm for complete bipartite graphs.

## 6.4. Proposed Algorithm Results for Simple Graph

A simple graph has no self loops, no multiple edges, is unweighted and undirected. I tested the algorithm for simple graphs too, to observe the minimum crossing numbers for various sized graphs. While generating simple test graphs, I considered the maximum number of edges that the simple graph can have according to a certain number of vertexes. Accordingly, I made sure that the number of edges in the files created should not exceed the maximum number of edges by using the formula  $(n \cdot n - 1) / 2$  for  $n$  vertex number.

Considering the rule stated here, I tested random graphs in 3 different groups in the categories small, medium and big and I will share the results.

```

esra@ubuntu:~/Desktop/mincrossing/mtn_cross_num_heuristic/greedy$ ./greedy -f 50x30.txt -t -n
time spent - 0.023523
minimum crossing number - 19
average crossing number - 19
esra@ubuntu:~/Desktop/mincrossing/mtn_cross_num_heuristic/greedy$ ./greedy -f 20x7.txt -t -n
time spent - 0.002977
minimum crossing number - 7
average crossing number - 7
esra@ubuntu:~/Desktop/mincrossing/mtn_cross_num_heuristic/greedy$ ./greedy -f 30x15.txt -t -n
time spent - 0.005141
minimum crossing number - 3
average crossing number - 3
esra@ubuntu:~/Desktop/mincrossing/mtn_cross_num_heuristic/greedy$ ./greedy -f 40x20.txt -t -n
time spent - 0.008929
minimum crossing number - 9
average crossing number - 9
esra@ubuntu:~/Desktop/mincrossing/mtn_cross_num_heuristic/greedy$ ./greedy -f 50x30.txt -t -n
time spent - 0.015462
minimum crossing number - 20
average crossing number - 20
esra@ubuntu:~/Desktop/mincrossing/mtn_cross_num_heuristic/greedy$ ./greedy -f 60x40.txt -t -n
time spent - 0.036701
minimum crossing number - 14
average crossing number - 14
esra@ubuntu:~/Desktop/mincrossing/mtn_cross_num_heuristic/greedy$ ./greedy -f 80x65.txt -t -n
time spent - 0.06021
minimum crossing number - 17
average crossing number - 17

```

Figure 6.14: Test results for graphs grouped as small (with vertex number  $20 < n < 100$ ).

```

esra@ubuntu:~/Desktop/mincrossing/mtn_cross_num_heuristic/greedy$ ./greedy -f 100x20n.txt -t -n
time spent - 0.292925
minimum crossing number - 330
average crossing number - 330
esra@ubuntu:~/Desktop/mincrossing/mtn_cross_num_heuristic/greedy$ ./greedy -f 100x30n.txt -t -n
time spent - 0.14902
minimum crossing number - 211
average crossing number - 211
esra@ubuntu:~/Desktop/mincrossing/mtn_cross_num_heuristic/greedy$ ./greedy -f 200x30n.txt -t -n
time spent - 9.20879
minimum crossing number - 1456
average crossing number - 1456
esra@ubuntu:~/Desktop/mincrossing/mtn_cross_num_heuristic/greedy$ ./greedy -f 200x200n.txt -t -n
reading from file failed
esra@ubuntu:~/Desktop/mincrossing/mtn_cross_num_heuristic/greedy$ ./greedy -f 200x100n.txt -t -n
time spent - 0.465392
minimum crossing number - 381
average crossing number - 381
esra@ubuntu:~/Desktop/mincrossing/mtn_cross_num_heuristic/greedy$ ./greedy -f 200x100n.txt -t -n
time spent - 0.419992
minimum crossing number - 355
average crossing number - 355
esra@ubuntu:~/Desktop/mincrossing/mtn_cross_num_heuristic/greedy$ ./greedy -f 200x30n.txt -t -n
time spent - 10.5066
minimum crossing number - 1536

```

Figure 6.15: Test results for graphs grouped as medium (with vertex numbers between 20 and 100 and edge numbers for 100 and 200). Also see that different runs with same graphs can give different crossing numbers

## 6.5. Visual Display of Simplified Graphs

We can understand that the graph drawing is simplified by looking at the crossing numbers, but to give the readers an idea visually, I have prepared some examples for some graphs with a file type .dia extension which supports diagram style files. I will share these examples in this section.

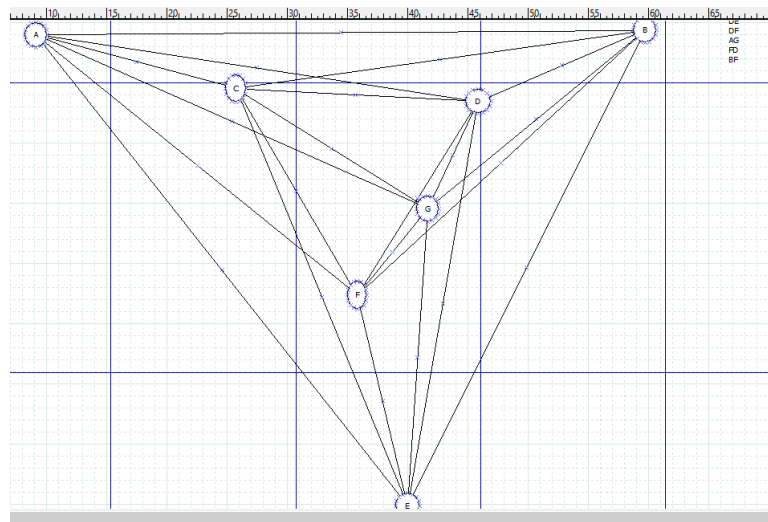


Figure 6.16: Complete graph with  $n=7$  after minimizing crossing numbers to 9.

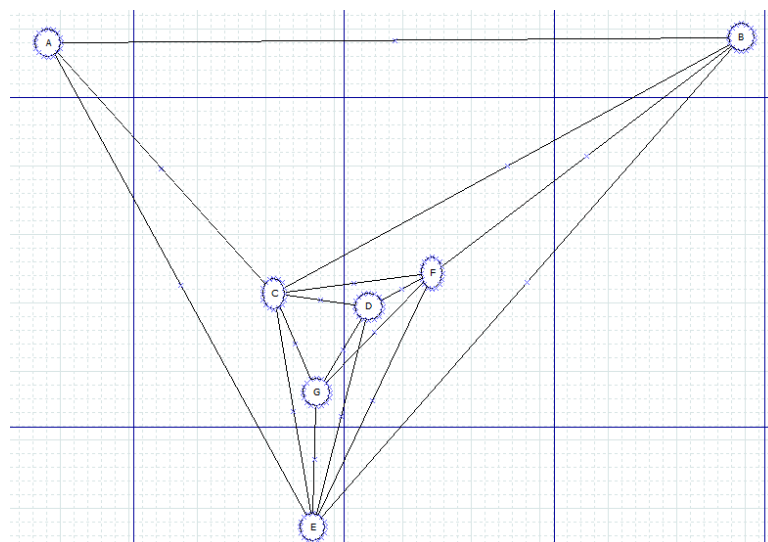


Figure 6.17: Complete graph with  $n=7$  example.

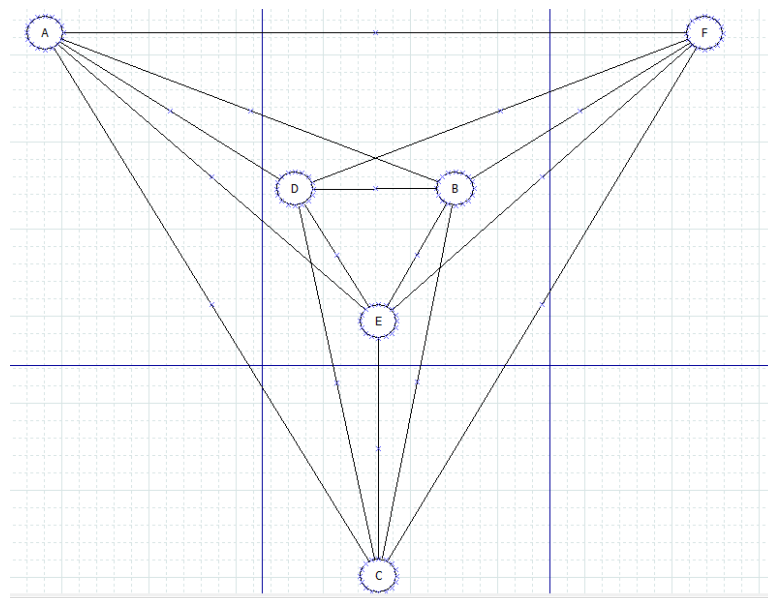


Figure 6.18: Complete graph with  $n=6$  after minimization of crossing numbers.

## 7. CONCLUSIONS

In this study, the minimum crossing number problem, which has a great place in graph theory, is examined. First of all, the definition of the problem and the points that should be known about graph theory were examined. In the next step of the study, previous solutions were investigated. Of these, brute force algorithms and some heuristic algorithms were examined. As far as I can see in the solution suggestions for the minimum crossing number problem, there is actually no compatible approach for each graph type and size. Some studies, for example, did well on complete graphs, while others focused on getting good results on small sizes, or some studies just tried their test with  $n, m$  values up to 10 for complete bipartite graphs. I also wanted to test its behavior on complete and complete bipartite graphs, along with the algorithm I will compare here.

After the coding phase was over, I tried to create a wider test set than the existing studies for complete and complete bipartite graphs. Our main goal in this study was to find the minimum number of crossing numbers lower than the compared algorithm, but I also completed the comparisons with the metaheuristic algorithms examined.

The results for Complete graphs were the same as the Quick Cross algorithm, except for one size, and we achieved more efficient results than the metaheuristic algorithms.

According to the test results made in Complete bipartite graphs, while the most tried  $n=3,4,5$   $m=5,4,10$  numbers gave the same results as the compared algorithms, when we enlarged the size of the complete bipartite graph, the solution developed gave much more effective results than the Quick Cross algorithm. and was able to make the crossing number more minimal.

The time the algorithm spent doing the calculations was less efficient for complete graphs compared to Quick Cross, while it caught the same times for complete bipartite graphs. The point I noticed here is that when we manually enter the number of iterations, the running time is even longer.

During the term, with an extensive literature search, followed by a solution proposal and tests; While the project was inefficient in some cases in terms of the time spent on success criteria, it produced much more efficient results in terms of crossing numbers than the other algorithm compared in complete bipartite graphs. In addition, while the Quick Cross algorithm cannot produce results in large graphs (for example, 50 vertex 500 edge) even in more than 50 minutes, the proposed algorithm produced results between 30-40 minutes in these graphs(The time given here should be considered as an extreme example since a very large graph is tried).

# BIBLIOGRAPHY

- [1] N. Biggs, E. K. Lloyd, and R. J. Wilson, *Graph Theory, 1736-1936*. Oxford University Press, 1986.
- [2] E. A. Bender and S. G. Williamson, *Lists, decisions and graphs*. S. Gill Williamson, 2010.
- [3] W. T. Tutte, "Toward a theory of crossing numbers," *Journal of Combinatorial Theory*, vol. 8, no. 1, pp. 45–53, 1970.
- [4] J. Pach and G. Tóth, "Which crossing number is it anyway?" *Journal of Combinatorial Theory, Series B*, vol. 80, no. 2, pp. 225–246, 2000.
- [5] J. Pach, F. Shahrokhi, and M. Szegedy, "Applications of the crossing number," *Algorithmica*, vol. 16, no. 1, pp. 111–117, 1996.
- [6] F. Shahrokhi, L. A. Székely, O. Skora, and I. Vrt'o, "Drawings of graphs on surfaces with few crossings," *Algorithmica*, vol. 16, no. 1, pp. 118–131, 1996.
- [7] F. Shahrokhi, O. Skora, L. A. Székely, and I. Vrt'o, "The crossing number of a graph on a compact 2-manifold," *advances in mathematics*, vol. 123, no. 2, pp. 105–119, 1996.
- [8] F. R. Chung, F. T. Leighton, and A. L. Rosenberg, "Embedding graphs in books: A layout problem with applications to vlsi design," *SIAM Journal on Algebraic Discrete Methods*, vol. 8, no. 1, pp. 33–58, 1987.
- [9] M. R. Garey and D. S. Johnson, "Crossing number is np-complete," *SIAM Journal on Algebraic Discrete Methods*, vol. 4, no. 3, pp. 312–316, 1983.
- [10] F. C. Harris Jr and C. R. Harris, "A proposed algorithm for calculating the minimum crossing number of a graph," in *Proceedings of the 8th Quadrennial International Conference on Graph Theory, Combinatorics, Algorithms, and Applications*, Citeseer, 1996.
- [11] A. Kaveh and A. Zolghadr, "Comparison of nine meta-heuristic algorithms for optimal design of truss structures with frequency constraints," *Advances in Engineering Software*, vol. 76, pp. 9–30, 2014.
- [12] R. Eberhart and J. Kennedy, "A new optimizer using particle swarm theory," in *MHS'95. Proceedings of the sixth international symposium on micro machine and human science*, Ieee, 1995, pp. 39–43.
- [13] A. Kaveh and M. Khayatazad, "A new meta-heuristic method: Ray optimization," *Computers & structures*, vol. 112, pp. 283–294, 2012.

- [14] —, “Ray optimization for size and shape optimization of truss structures,” *Computers & Structures*, vol. 117, pp. 82–94, 2013.
- [15] R.-L. Wang *et al.*, “Artificial neural network for minimum crossing number problem,” in *2005 International Conference on Machine Learning and Cybernetics*, IEEE, vol. 7, 2005, pp. 4201–4204.
- [16] A. Kaveh and G. M. ILCHI, “Metaheuristic algorithms for minimum crossing number problem,” 2015.