

Tool for Small Induced Subgraph Identification

Chigozie Emmanuel Ekwonu  
MSc Software Development - 2228371

School of Computing Science

Sir Alwyn Williams Building

University of Glasgow

G12 8RZ

A dissertation presented in part fulfillment of the requirements of the Degree of MSc in Software Development at the University of Glasgow

September 2016

**Abstract**

This report discusses the development of a tool which is used to list simplicial vertices in a graph, and also detect induced subgraphs in a graph that are triangles, diamonds, claws and other complete subgraphs. It makes use of algorithms which are discussed in a published paper.

As these algorithms are highly dependent on fast matrix multiplication algorithms, this report involves discussions of the factors which make implementing the fast matrix multiplication algorithms in practice not particularly useful.

Furthermore, the detection algorithms were extended to list all the induced subgraphs of a desired type, i.e. triangles, claws, diamonds and other complete subgraphs present in a given graph. Algorithms to generate random graphs of different sizes that do not contain a desired induced subgraph type were designed and implemented.

The implemented algorithms are evaluated to compare their performance against brute force algorithms developed to detect and list similar subgraph types. The results of the evaluations are discussed.

Education Use Consent

I hereby give my permission for this project to be shown to other University of Glasgow students and to be distributed in an electronic form.

Name: Chigozie Ekwonu Signature:

Acknowledgements

I thank God who always provided a way for me and seeing me throughout the duration of my programme. I would like to thank my parents for their constant moral support.

I would also like to thank my supervisor, Dr Alice Miller for her guidance and valuable advice which she provided during the course of this project. Many thanks also go to Craig Reilly for making out time to assist, especially while running the evaluations.

I also thank the Federal Government of Nigeria for sponsoring my place in this programme.

Contents

Chapter 1 Introduction 1

1.1 Problem Definition 1

1.2 Purpose of the Tool 1

1.3 Dissertation Organization 2

Chapter 2 Background Survey 3

2.1 Graph Theory Applications over the Years 3

2.2 Graph Terminology 3

2.3 Graph Representations 6

2.3.1 Edge Set Representation 6

2.3.2 Adjacency List Representation 6

2.3.3 Adjacency Matrix Representation 7

2.4 Matrix Multiplication Algorithms 7

2.4.1 Strassen Multiplication Algorithm 8

2.4.2 Coppersmith-Winograd Algorithm 8

2.5 Triangle Detection and Listing 8

2.6 *K*4 Detection 9

2.7 Complete Subgraph Listing 10

2.8 Diamond Detection 10

2.9 Claw Detection 11

2.10 Simplicial Vertices Listing 11

2.11 Related Software Products 11

2.11.1 NetworkX 11

2.11.2 Graph-tool 12

2.11.3 Nauty and Traces 12

2.12 Conclusion 12

Chapter 3 Requirements 13

3.1 Requirements Capture 13

3.2 Functional requirements 13

3.3 Non-functional requirements 14

3.4 Users 14

Chapter 4 Design 15

4.1 System Architecture 15

4.2 GUI Design 16

4.3 Overview of Algorithms 16

4.3.1 Matrix Multiplication Algorithm 16

4.3.2 Triangle Detection Algorithm 17

4.3.3 *P*3 Detection Algorithm 18

4.3.4 *P*3 Listing Algorithm 18

4.3.5 Triangle Listing Algorithm 18

4.3.6 Simplicial Vertices Listing Algorithm 19

4.3.7 Simplicial Vertex Detection Algorithm 20

4.3.8 K4 Detection Algorithm 20

4.3.9 K4 Listing Algorithm 20

4.3.10 Claw Detection Algorithm 20

4.3.11 Claw Listing Algorithm 21

4.3.12 Diamond Detection Algorithm 21

4.3.13 Diamond Listing Algorithm 21

4.3.14 Complete Subgraph Listing Algorithm 22

4.3.15 Complete Subgraph Detection Algorithm 23

4.3.16 Depth First Search Algorithm 23

4.3.17 Random Pattern-free Graph Generation Algorithms 24

4.3.18 Brute force Detection and Listing Algorithms 24

4.4 Conclusion 25

Chapter 5 Implementation 26

5.1 Development Methodology 26

5.2 Language of Implementation 26

5.3 Graph Abstract Data Type (ADT) Implementation 26

5.4 Matrix Multiplication Algorithm Implementation 28

5.5 Implementing the Subgraph Detection and Listing Algorithms 29

5.6 Implementing the Random Pattern-free Graph Generation Algorithms 30

5.7 Implementing the Command Line Interface 30

5.8 Graphical User Interface (GUI) Implementation 31

5.9 Implementation Challenges 31

5.9.1 Fast Matrix Multiplication Algorithm Implementation 31

5.9.2 Subgraph Replication 32

5.9.3 Multiple Edges Addition in the Complete Subgraph Listing Algorithm 32

5.9.4 Implementing Algorithms to Generate Random Pattern-free Graphs 33

Chapter 6 Testing and Evaluation 34

6.1 Testing Strategy 34

6.2 Unit Testing 34

6.3 System Testing 34

6.4 Evaluation Strategy 34

6.4.1 Evaluation of the Efficient Detection Algorithms 35

6.4.2 Evaluation of the Efficient Listing Algorithms 36

6.4.3 Evaluation of the Random Pattern-free Graph Generation Algorithms 37

6.4.4 Evaluation of the Usability of the Tool 38

Chapter 7 Conclusion 39

7.1 Status of the Product 39

7.2 What Has Been Achieved 39

7.3 Personal Reflection 39

7.4 Future Work 39

References 41

Appendix A Algorithms Pseudo-code 1

Appendix B Evaluation Result Charts 5

Appendix C UML Class Diagrams 10

Appendix D User Manual 19

Appendix E System Testing Results 20

# Introduction

This report discusses the development of a tool for subgraph identification. It details the research which was done, as well as the design, implementation, testing, and evaluation which were required to develop the project.

Graph theory is a branch of mathematics which is concerned with the study of graphs, which are network of points connected by lines. Graph theory has applications in numerous fields including chemistry, social sciences, computer science, etc. [3].

The main focus of this project is to implement efficient algorithms to check a given simple graph for the presence of subgraphs of specific types, and produce the vertices which induce such subgraphs, if they exist. These efficient algorithms have been described in a published paper [11] and the tool aims to implement them to analyze their performance in practice.

## Problem Definition

Many combinatorial problems e.g. the detection of a minimum cycle in a graph, finding the approximate Hamiltonian walk in maximal planar graphs and checking planar graphs to determine the appropriate minimum vector cover or maximum independent set rely on the detection and listing of triangles [4]. The ability to quickly identify triangles would make it more efficient to analyze and solve these problems.

Simplicial vertices (i.e. vertices whose neighbour vertices form a complete subgraph) when quickly found and removed makes tackling the problem of graph colouring easier [11]. Graph colouring involves assigning colours to the vertices of a graph such that adjacent vertices do not have the same colour.

Graphs that are claw-free are useful in finding a polynomial time algorithm for the computation of the independence number of a graph, finding Hamiltonian properties, characterization of line graphs, and observation of matching properties of such graphs [11]. The quick recognition of such claw-free graphs helps in solving these problems.

## Purpose of the Tool

The tool developed in the course of this project will have the ability to efficiently identify the presence of simplicial vertices and special classes of graphs e.g. claws, diamonds and complete graphs. If any of these graphs is found, the tool outputs the vertices which induce such a graph. The tool makes use of efficient algorithms to perform the operations. Figure 1.1 gives a quick idea of the purpose of the tool. It shows a graph which contains a diamond as an induced subgraph (indicated in dotted lines) and also a simplicial vertex (indicated by the large dot).



Figure 1.1: Graph showing a diamond subgraph whose edges in dotted lines. A simplicial vertex is indicated by the large dot

The tool also contains implementations of algorithms which are used to generate random graphs that do not contain specified types of graphs as induced subgraphs.

## Dissertation Organization

In chapter 2, we give a brief history of graph theory, introduce relevant terminology and theory, and discuss related software.

In chapter 3, we discuss the requirement capture, prioritizing the various requirements using the *MoSCoW* system.

In chapter 4, we discuss the system architecture, design decision, the algorithms that were used and their complexities.

In chapter 5, we discuss the implementation of the algorithms, the challenges encountered and decisions that had to be made.

In chapter 6, we discuss the testing and evaluation procedure used to ascertain the correctness of the tool.

Chapter 7 concludes the dissertation and discusses the state of the project, what has been achieved, personal reflection and future works that can be carried out.

# Background Survey

This chapter mentions some applications of graph theory and provides definitions of common graph terminology, especially those that were used in the course of the project. Comparisons of existing subgraph identification algorithms will be discussed together with their strengths and weaknesses. We conclude by analyzing related software products, including those that are related to subgraph identification as well as more general problems.

## Graph Theory Applications over the Years

Graph theory has many real world applications today due to the way in which graphs can be used to model complex relationships between data objects, for example in finding the shortest path between two places as used by GPS navigation systems, where numerous paths exist through which one place can be reached from another. It is also applicable in molecule mining in chemistry and genomics in biochemistry. Applications where connectivity information exists such as in social networks, computer networks, puzzles, etc. can be modeled using graphs [7].

## Graph Terminology

Most of the definitions used here are obtained from [7] and [22].

A graph G = (*V*, *E*), is a set, *V*, of vertices and a collection, *E*, of pairs of vertices from V, called edges. Some texts refer to the vertices as node. Graphs may be directed or undirected. In a directed graph, an edge (*u, v*) is directed from *u* to *v* if the pair (*u, v*) is ordered, with *u* preceding *v*. In an undirected graph, the ordering of the pair (*u, v*) does not matter. Graph edges here are grouped as a collection, rather than a set, meaning that a graph may have multiple edges (called parallel edges) between two vertices. An edge could also exist which connects a vertex to itself. Such an edge is referred to as a self-loop. The graphs which are dealt with in this project are **simple** graphs i.e. graphs which do not contain parallel edges or self-loop [7].

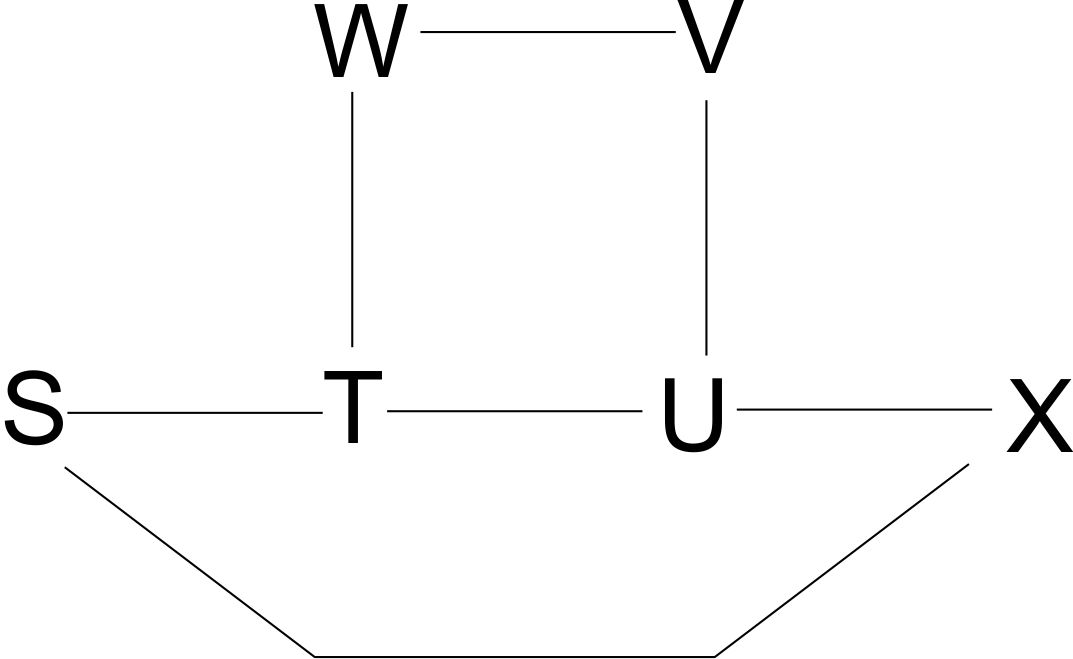


Figure 2.1: Undirected Graph (adapted from [25])

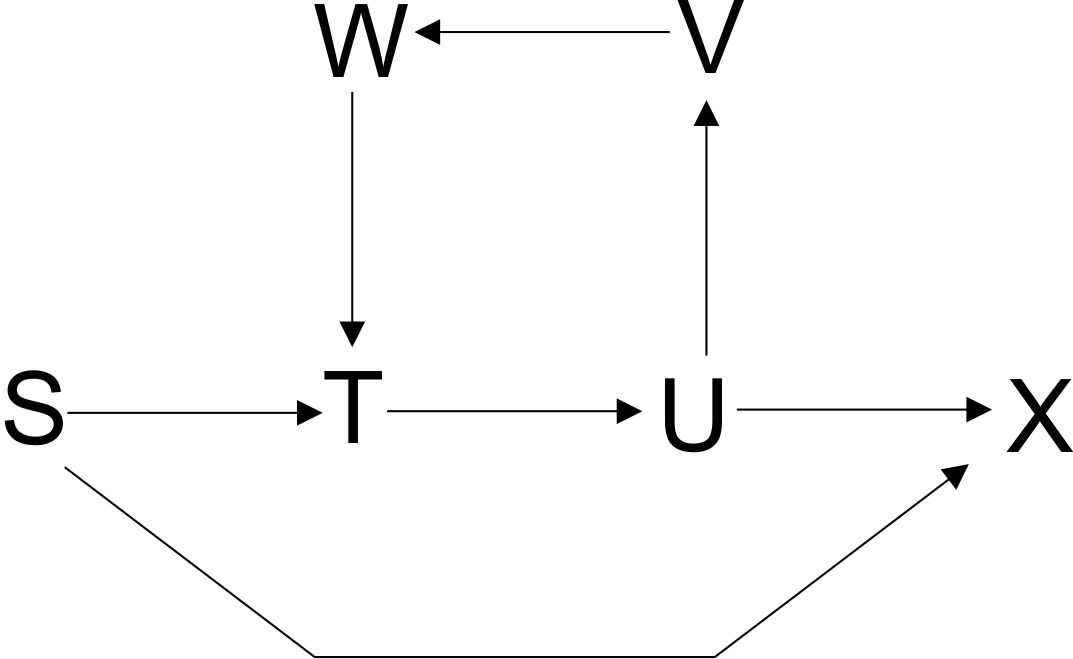


Figure 2.2: Directed Graph (taken from [25])

The **size** of a graph is the number of vertices in the graph. The **degree** of a vertex *v* is the number of incident edges of *v.* A **subgraph** *H* = (*VH, EH*) of a graph, *G* = (*VG, EG*), is a graph where *VH* ⊆ *VG* and *EH* ⊆ *EG*. An **induced subgraph** of a graph, *G*, is a subgraph graph, *H*, of *G* such that *E*H = *E*G ∩ *E* (*V*H) [22].

A **cycle** is a sequence of vertices that start and end in the same vertex, such that each two consecutive vertices in the sequence are adjacent to each other in the graph [6].

A **path** in a graph is a list of vertices such that each vertex in the list and its successor in the list are connected by an edge in the graph [25]. A path is normally represented as a *Pn*, where *n* is the number of vertices in the path. The vertices in a path should be distinct. The length of a path is the number of edges in the path. Hence a *P*n has length .

A **connected graph** is a graph in which there exists a path between any two vertices of the graph. If a graph *G* is not connected, then the maximal connected subgraphs are referred to as the connected components (or simply components) of *G*. A connected graph has one component. The graph shown in Figure 2.3 below is a disconnected graph

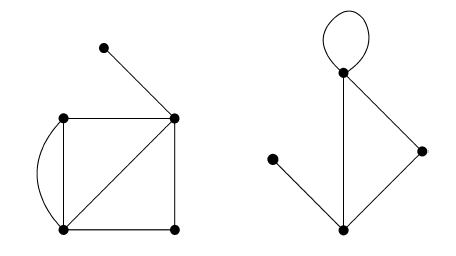


Figure 2.3: A disconnected graph with two components (taken from [22])

A graph may have an edge whose end vertices are identical. Such an edge is called a **self-loop** [22]. A graph may also have more than one edge being associated with a given pair of vertices. Such edges are known as **parallel** edges. A simple graph is a graph that does not have self-loops or parallel edges [22].

A **complete graph** is a simple graph in which each pair of distinct vertices has an edge between them [22]. In a complete graph with *n* vertices, the number of edges *m* = *n\*(n-1)/2.* A **clique** of a graph, *G,* is a complete subgraph of *G*. A *Kl*graph is a complete graph with *l* vertices.

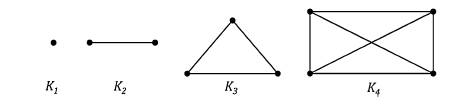


Figure 2.4: Complete graphs *K1, K2, K3, and K4 (taken from [22]*

The **neighbourhood** of a vertex *v* of graph *G* is a graph *N*(*v*)induced by the vertices adjacent to *v*. The closed neighbourhood of vertex *v* is a graph *N*[*v*]such that *N*[*v*] = {*v*} ∪ *N*(*v*) [11].

The **complement graph**, ­, of a graph, is a graph with the same vertex set as *G* such that two vertices *u* and *v* are adjacent in ­ precisely when they are not adjacent in *G* [22].

The **arboricity**, of a graph *G* is the minimum number of edge-disjoint spanning forests whose union is *G.* The term “edge-disjoint spanning forests” means that each of the forests into which *G* can be decomposed has exactly the same vertex set as *G* but the edge sets differ among the forests. An acyclic graph has an arboricity of 1. The maximum arboricity of a planar graph is 3. A planar graph is a graph which can be drawn such that there is no intersection between any two of its edges [22]. A complete graph with *n* vertices has an arboricity equal to . A graph *G* with *n* vertices where and m edges and *m*p being the maximum number of edges in any subgraph of *G* with *p* vertices has an arboricity given in Equation 2.1 [9]:

(2.1)

For a graph *G* with *n* vertices, the adjacency matrix of *G* is an *n* by *n* symmetric binary matrix *A* defined over the ring of integers such that *A*[*i, j*] = 1 if (*i, j*) is an edge in *G*, and 0 otherwise [7][22]. Figure 2.5 shows an adjacency matrix of a graph.

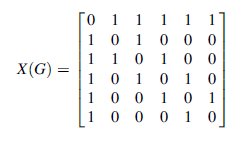


Figure 2.5: Adjacency matrix of a graph (taken from [22])

## Graph Representations

A graph is represented in a computer programming language as an abstract data type (ADT), i.e. its internal data representation is private, but provides a set of methods through which the internal data can be accessed or modified [17]. Some of the ways in which the internal representation of a graph’s data can be done in are discussed next.

### Edge Set Representation

Graphs can be represented as a vertex set with an edge set, with each edge object having pointers to the two vertices that it connects. By representing both sets with doubly linked lists, vertex and edge addition as well as edge deletion operations can be done in constant time [25]. The size of the graph can be obtained also in constant time if the graph’s size is stored as an additional variable. Checking if two vertices are adjacent, getting a list of a vertex’s neighbours and deleting a vertex (which also requires deletion of any connecting edges) would require a time complexity of *O(m)*, where *m* is the number of edges in the graph.

This *O(m)* time complexity is not suitable for the algorithms which are to be used, as the algorithms involve accessing the neighbours of a vertex on a regular basis.

### Adjacency List Representation

An alternate representation of graphs is with the use of adjacency lists, where each vertex maintains a list of its neighbours. The adjacency list representation requires (as suggested by [7]):

* A collection, *V*, of *n* vertices
* A collection, *E*, of *m* edges
* For each vertex, *v,* in *V*, an adjacency list for *v* which represents the edges incident on *v*. The adjacency list could be implemented as a list of references to each vertex, *w,* so that edge (*v, w)* is contained in *E*, or as a list of references to each edge that is incident on *v*

A constant time operation to retrieve the neighbours of vertex *v* is then possible. Checking if vertices *u* and *v* are adjacent can be done by checking the adjacency list for *u* or *v*. Checking the smaller of the two lists will resultin *O*(min{*du,dv*}) running time [7], where *dv* is the degree of *v* and *d*u is the degree of *u*.

Deletion of vertex *v* will take *O(dv)* time, because even though removing the vertex from the collection (e.g. a doubly linked list or a hash map) will take constant time, the deleted vertex would still have edges that link to it present in the edges collection, *E*, and its neighbours’ adjacency lists, and traversing all the adjacency lists to find and delete these edges would require *dv* steps. Traversing *E* would also require *dv* steps, giving a total of 2\**dv* steps and a time complexity of *O*(*dv*).

### Adjacency Matrix Representation

Adjacency matrix representation of graphs involves numbering of the vertices 1, 2, 3,…,n, with the edges being viewed as pairs of the integers[7]. Representations may make use of an *n* by *n* array, *A* to represent a graph, *G*, such that *A*[*i*, *j*] stores a reference to an edge object, *e*, if an edge, (*i, j*) exists in *G*, otherwise *A*[*i*, *j*] is null [7]. In some cases in which a custom edge object is not be required, if an edge, exists in *G*, otherwise If the vertices or edges have data associated with them, then some sort of mapping of vertex numbers to vertex data, and vertex pairs to edge data, must be present.

With the use of an adjacency matrix to represent a graph, to check if two vertices are adjacent takes constant time. This is done by accessing the matrix indices *i* *and j,* which correspond to vertices *v* and *w* respectively, and then checking if *A*[*i, j*] contain a reference to an edge object, or if a custom edge object is not required for the graph. In addition, edge addition and edge deletion operations are constant time operations.

However, adjacency matrix representation of graphs consumes space as it has a space complexity of *O*(*n2*). Also, it is unsuitable for graphs in which parallel edges are possible. The representation is not suitable for use in situations where vertices are added or removed after the matrix has been created. It is for this reason that the adjacency matrix implementation was not used in the development of the tool.

A useful property of adjacency matrices is that when the matrix is raised to the *kth* power, it is easy to see at a glance the number of paths of length *k* that exist between any pair of vertices. This is useful in the algorithm to detect the presence of a triangle in a graph.

## Matrix Multiplication Algorithms

Matrix multiplication algorithms are discussed because the algorithms implemented rely on a fast matrix multiplication algorithm to arrive at the specified time complexities.

The standard operation of multiplying two *n* x *n* matrices together is commonly known to require a cubic number of operations, i.e. running in *O*(*n3*) time. Strassen in 1969 came up with an algorithm which reduced the time complexity of matrix multiplication to . Over the years, different researchers have made breakthroughs aimed at reducing the exponent of matrix multiplication, ω, even further. Note that ω = 3 for standard matrix multiplication and ω = 2.808 for Strassen’s matrix multiplication. Coppersmith and Winograd in 1989 came up with an algorithm that reduced ω to less than 2.376 and this remained unchanged for more than twenty years [26]. Williams in 2014 improved upon Coppersmith-Winograd’s algorithm, by reducing ω to less than 2.373 [26].

However among the fast matrix multiplication algorithms which have been discovered, – Coppersmith-Winograd algorithm, Strassen algorithm, Stother’s algorithm, William’s algorithm – only Strassen’s algorithm has been implemented practically, due to the high degree of complexity surrounding the other algorithms.

### Strassen Multiplication Algorithm

As found in [5], Strassen’s multiplication approach follows an idea in which the number of multiplication operations that will be performed during matrix multiplication is reduced. Strassen used a recursive approach, in which two *n* x *n* matrices are multiplied by making use of 7 recursive multiplications of *n*/2 x *n*/2 matrices. The base condition of the recursion occurs when the resulting matrix is of dimension *1* x *1,* where only one multiplication operation occurs and that branch of the recursion tree terminates*.* To ensure that *n*/2 will always result in a whole number, the initial value of *n* must therefore be a power of 2.

Ideally, a simple divide-and-conquer algorithm for matrix multiplication described in [5] would require 8 recursive multiplication operations of *n*/2 x *n*/2 matrices at each depth of the recursion tree, except at the base condition. Strassen reduced this number by eliminating one matrix multiplication and instead adding several new *n*/2 x *n*/2 matrices, which will require a constant number of additions. The idea is that it is more efficient to perform addition operations on matrices than it is to multiply them. ­­­This trade off resulted in a lower asymptotic running time, giving a time complexity of *O*(). More details of Strassen’s implementation can be found in [5].

Due to the requirement that the initial value *n* is a power of 2, matrices which do not satisfy this requirement would require their rows and columns to be padded with zeros before Strassen’s multiplication can be applied. This means that the calculated product would have some padded entries. The correct result will have to be obtained by picking the values at the related rows and columns from the calculated result.

Strassen’s algorithm produces better performance in practice when applied to large matrices. This is due to the large value of the constant factors which are omitted in the analysis of the time complexity but have an impact on the practical performance. It would be ideal to add a check such that Strassen’s algorithm is used on matrices above a certain size (the cross-over point), while switching to the standard matrix multiplication below that size. The size is however highly system dependent [5]. The sub-matrices that are created at the recursion depths also take up some space, and could cause the program to run out of heap space when implemented in java.

### Coppersmith-Winograd Algorithm

Coppersmith-Winograd’s algorithm is unfortunately very complex to implement practically, and in addition requires very large matrix sizes to be used [26] in order to record practical advantages over standard matrix multiplication. This is mainly as a result of the large constant factors which are left out in the analysis of the time complexity but turn out to be significant in practice. Details of the algorithm can be found in [26]. No practical implementation of this algorithm in any programming language has been found.

## Triangle Detection and Listing

In [10], algorithms for finding a triangle (a cycle that contains three vertices, commonly referred to as a *C­3*) in a graph, *G* were presented*.* In one of these algorithms, a triangle can be found with a worst case time complexity of *O*(*nm*)*,* where *n* is the number of vertices in G and *m* is the number of edges.

Another algorithm presentedinvolves finding the square of *A*, where *A* is the adjacency matrix of *G.* and then checking that *A2*[*i, j*] > 0 for all *i ≠ j*. Such an occurrence indicates the presence of a path with three vertices*.* This can be done in *O(n*ω*)* running time*,* where ω = 2.808 as calculated from Strassen’s matrix multiplication algorithm. The third vertex *k* to complete the triangle can be found in *O*(*n*) running time, giving a combined running time of *O(n*ω*)*.

In [1], another algorithm for detecting the presence of a triangle in a graph was presented. This algorithm has a running time of *O*(*m1.41*), where *m* is the number of edges in *G*. This algorithm assumed that the Coppersmith-Winograd multiplication algorithm is used, and has ω = 2.376. The algorithm involves partitioning the vertices of *G* into low degree and high degree vertices. High degree vertices have degrees greater than *D* while low degree vertices have degrees less than or equal to *D*. The value of *D* is obtained from Equation 2.2:

(2.2)

All paths of length two whose intermediate vertex is of low degree can be found in *O*(*mD*) running time. The endpoints of each such path can then be checked to detect if an edge exists between them. The presence of such an edge indicates the presence of a triangle. If a triangle is not found at this stage, then a triangle, if present in *G*, has all its vertices as high degree vertices. The maximum number of high degree vertices in *G* is, hence a triangle can be found in time through matrix multiplication [1]. Adding and and substituting the value of *D* results in a time complexity of *O*(*m1.41*). For complete graphs, the stage of the algorithm that utilizes matrix multiplication will be used, which puts it at a disadvantage when compared to triangle detection algorithms that do not involve matrix multiplication.

In [4], an algorithm which lists all triangles in a graph in *O*() running time was described. This algorithm was discovered to perform better in practice than the triangle listing algorithms of [1] and [10] when the latter algorithms were modified to list all triangles found. According to [23], algorithms which are based on matrix multiplication are useful in counting triangles in a graph but not in listing them.

## *K*4 Detection

In [11], an algorithm to detect the presence of an induced *K*4subgraph in a connected graph was described. The algorithm was also extended to make it count the number of *K*4’s in the given connected graph.

This algorithm has a time complexity of *O*(*m*(ω +1)/2) = *O*(*m*1.69) if *ω* is the exponent of matrix multiplication and is equal to 2.376 as proposed by Coppersmith-Winograd’s matrix multiplication algorithm. Details of this algorithm will be discussed in chapter 4 of this dissertation.

## Complete Subgraph Listing

An algorithm to list all the complete subgraphs of size *l* was provided in [4]. The algorithm runs recursively such that to find a complete subgraph *Kl*, one can pick a vertex *v* and find a complete subgraph *Kl-1*that is induced by the neighbours of *v*. This algorithm was proven to have a running time of , and uses linear space.

An algorithm which detects the presence of a K*l* graph in *O*() time was provided in [18], where *i* here equals the remainder when *l* is divided by 3 and *n* is the size of the graph. The algorithm was designed around the time complexity of the triangle detection algorithm being *O*(). The idea was that an auxiliary graph, *H,* is created such that *H* contains a triangle if and only if *G* contains a subgraph of size *l.* For values of *l* greater than 5, such an algorithm has to operate recursively (i.e. call itself) in order to construct the auxiliary graph *H.*

The complete subgraph detection algorithm of [18] is improved in [11] by modifying the triangle recognition algorithm from [1] and the *K4* recognition algorithm described in section 2.6. This resulted in a running time of *O*() if *i* ∈ {1, 2} and *O*(), where *m* is the number of edges in the graph and if *i*= 0. *i* is as defined in the previous paragraph.

## Diamond Detection

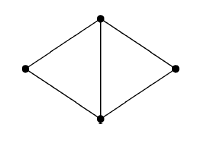


Figure 2.6: A diamond (taken from [11])

A diamond graph is a graph that is isomorphic to the graph shown in Figure 2.6 It is noteworthy that by the definition of an induced subgraph, a complete graph cannot contain an induced diamond. If a diamond is to be detected in a graph but not as an *induced* subgraph, then a graph containing a complete subgraph of size greater than or equal to 4 would contain such diamond.

An algorithm which detects the presence an induced subgraph isomorphic to a diamond in a graph *G* was presented in [11]. The algorithm once again involves partitioning the vertices of graph *G* into low and high degree vertices, such that the high degree vertices are vertices whose degrees are greater than *D,* where and *m* is the number of edges in the graph. The algorithm involved was done in three phases. Details of this are found in [11] and in chapter 4 of this dissertation.

This algorithm has a time complexity of , where *ω*  is as defined in previous sections, *m* is the number of edges in the graph and *n* is the number of vertices in *G*.

## Claw Detection

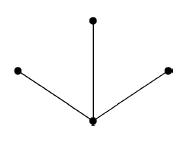


Figure 2.7: A claw (taken from [11])

A claw graph is a graph that is isomorphic to the graph shown in Figure 2.7 above. The vertex of a claw whose degree is 3 is known as the central vertex. A complete graph cannot contain a claw as an *induced* subgraph. However any graph which contains a complete subgraph of size greater than 3 will contain a claw as a subgraph.

An algorithm for checking if a given graph G is claw-free was presented in [11]. The algorithm was developed around the fact that a claw-free graph does not have any vertex with more than neighbours. Details of the algorithm can be found in [11] and in chapter 4 of this dissertation. The algorithm has a running time of . *m* and ω are as defined earlier.

## Simplicial Vertices Listing

A time algorithm for listing all the simplicial vertices of a graph was described in [11]. A simplicial vertex is a vertex whose neighbourhood forms a complete subgraph. This algorithm was implemented in phases. Details of the phases can be found in [11] and in chapter 4 of this dissertation.

## Related Software Products

### NetworkX

NetworkX is a python package for working with graphs. It provides functionalities for different graph types i.e. undirected simple graphs, directed simple graphs, undirected and directed graphs with self-loop and multigraphs. The main purpose of the library is to perform different analyses on graphs [24]. Some of these analyses as seen in [24] include:

* Working with components of graphs such as testing for graph connectivity, number of connected components, etc.
* Working with cliques for example finding the maximal clique in a graph, removing cliques from a graph, getting the clique number, etc.
* Getting an approximate maximum independent set of a graph
* Isomorphism operations such as checking if two graphs are definitely isomorphic or could be isomorphic, etc.

NetworkX can interface with another python library, *matplotlib*, to visualize graphs [24].

Since NetworkX is actually a software library, and not a ready-to-use tool, it will require coding skills before it can be put to use.

### Graph-tool

Graph-tool is another library for manipulating directed and undirected graphs. It is equipped with numerous algorithms for performing different operations on graph. It is python based, but its internals make use of C++ for performance through the Boost Graph Library [21]. Like NetworkX, it has methods for standard graph operations – vertex addition and deletion, edge addition and deletion, providing different iterators for iterating through vertices and edges, etc. It also has the ability to visualize graphs that have been created.

Graph-tool has many similarities with NetworkX in terms of the different types of analyses that both libraries can perform on graphs. As a result, they share similar drawbacks when compared with the tool which was developed in this project. The creators of Graph-tool in [8] claim that it performs faster than NetworkX due to the pure python implementation of NetworkX, which makes it slower than C++.

### Nauty and Traces

Nauty and Traces are programs which are useful in computing the automorphism groups of graphs, testing graphs for isomorphism, as well as producing the canonical labeling of graphs. The automorphism group of a graph is the group of isomorphisms of the coloured graph to itself [15]. Canonical labeling of graphs involves placing the vertex labels such that their new positions are independent on their former positions. Isomorphic graphs have the same canonical labeling. [14].

Included in the Nauty packages are tools for generating many types of graphs: bipartite, multigraphs, non-isomorphic graphs, etc., and for solving the graph isomorphism problem. This problem is known to belong to the NP computational complexity class [15]. The tool aims to tackle the problem by generating the automorphism groups of the graphs to be checked for isomorphism, and then comparing both groups. Both Nauty and Traces are written in a portable subset of C programming language, making them runnable of modern computers.

Nauty and Traces are useful programs, due to their features and functionalities. However, these make them very complex to use, as they have a lot of commands which a user would have to be familiar with.

## Conclusion

We have discussed the related literature works that were consulted as well as related tools. In the next chapter, the requirements gathering and requirement priorities will be discussed.

# Requirements

## Requirements Capture

The requirements of the tool were received after discussions with the project supervisor Dr. Alice Miller and Craig Reilly, a PhD candidate who works with graphs. The tool is required to implement algorithms described in [11].

The following requirements were developed and their priorities assigned using MoSCoW analysis. MoSCoW analysis involves assigning priorities to the requirements. These priorities include: must have, should have, could have and would like to have [20].

## Functional requirements

The functional requirements of the tool are provided in Table 3.1 below. Figure 3.1 shows the use case diagram of the system.

|  |  |  |
| --- | --- | --- |
| **S/N** | **Requirement** | **Priority** |
| 1 | Detect and list induced subgraphs that are triangles | Must have |
| 2 | Detect and list induced subgraphs that are diamonds | Must have |
| 3 | Detect and list induced subgraphs that are claws | Must have |
| 4 | Detect and list simplicial vertices | Must have |
| 5 | Detect and list induced complete subgraphs of size 4 using the K*4*detection algorithm in [11] | Must have |
| 6 | Detect and list induced complete subgraphs using the K*L*detection algorithm in [11] | Must have |
| 7 | Interact with the tool via command prompt | Should have |
| 8 | Accept graphs presented as adjacency matrix in a file | Should have |
| 10 | Generate random diamond-free graphs | Could have |
| 11 | Generate random claw-free graphs | Could have |
| 12 | Generate random simplicial vertex free graphs | Could have |
| 13 | Generate random graphs free of complete subgraphs of size greater than 1 | Could have |
| 14 | Save randomly generated graphs in files | Could have |
| 15 | Interact with tool via a graphical user interface | Could have |
| 15 | Visualize graph and any subgraph found | Would like to have |

Table 3.1: Functional Requirements and their priorities

## Non-functional requirements

* Implemented algorithms conform to the theoretical worst-case time complexities.
* Perform testing to check correctness of implemented algorithm.
* Evaluate algorithm performance when compared to naïve brute force approach in the detection of the various subgraphs and produce evaluations results.

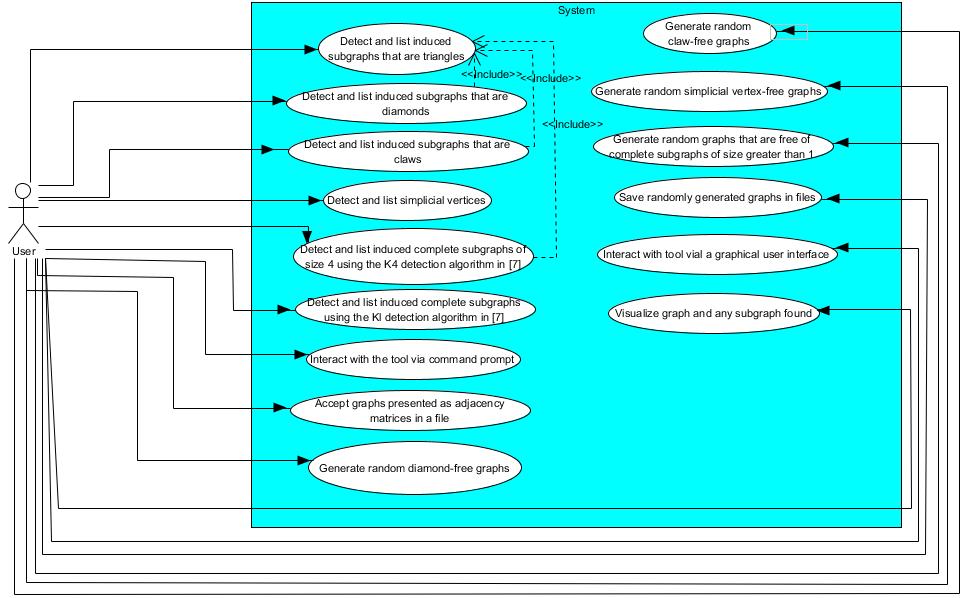


Figure 3.1: Use case diagram

## Users

The users of the tool include students and researchers that work in the field of combinatorics and have to solve the problems listed in section 1.1.

# Design

In this chapter, the architecture of the system will be discussed, together with the major algorithms which were used in the development of the tool.

## System Architecture

The tool design is split into multiple packages: efficient.detection, bruteforce.detection, efficient.listing, bruteforce.listing, exception, gui, generate, general, test and a default package.

The efficient.detection package contains the different classes which detect the presence of various types of induced subgraphs using the algorithms described in [11]. The algorithms terminate as soon as one of such subgraphs is found.

The generate package contains a class which contains methods for generating random graphs that are free from certain induced subgraph types and another class for generating random graphs used for evaluation purposes.

The test package contains unit test classes for verifying the correctness of the tool and a class for combining results of the evaluation experiments.

The gui package contains the GUI class for creating a graphical user interface for the tool, some worker classes used by the GUI class to prevent the GUI display from freezing up when it is performing a long task, and a class which serves as the controller class for handling events generated in swing.

The bruteforce.detection package contains brute force implementations of algorithms to detect the presence of the different induced subgraph types. This is useful to determine how well the efficient algorithms perform in practice.

The efficient.listing package contains modifications of the efficient.detection package classes to list all the subgraphs found.

The bruteforce.listing package contains modifications of the bruteforce.detection package classes to list all the subgraphs found.

The exception package contains custom exception classes which are useful in communicating disruptive events that occur during program execution.

The default package contains only the RunMe class which is used to provide a command line interface for interacting with the tool.

The general package contains other classes such as the Utility class which contains methods that were used repeatedly by other classes, the Graph interface which represents the contract for the graph ADT and the UndirectedGraph class which contains implementation of the graph ADT.

UML class diagrams are provided in Appendix C.

## GUI Design

Since the GUI requirement has a “could have” priority, the design was done later in the project and it presents the user with a way to interact with the tool without having to type in commands on a command line interface. The GUI offers most of the features which are accessible through the command line interface. Figure 4.1 shows a wireframe of the GUI.

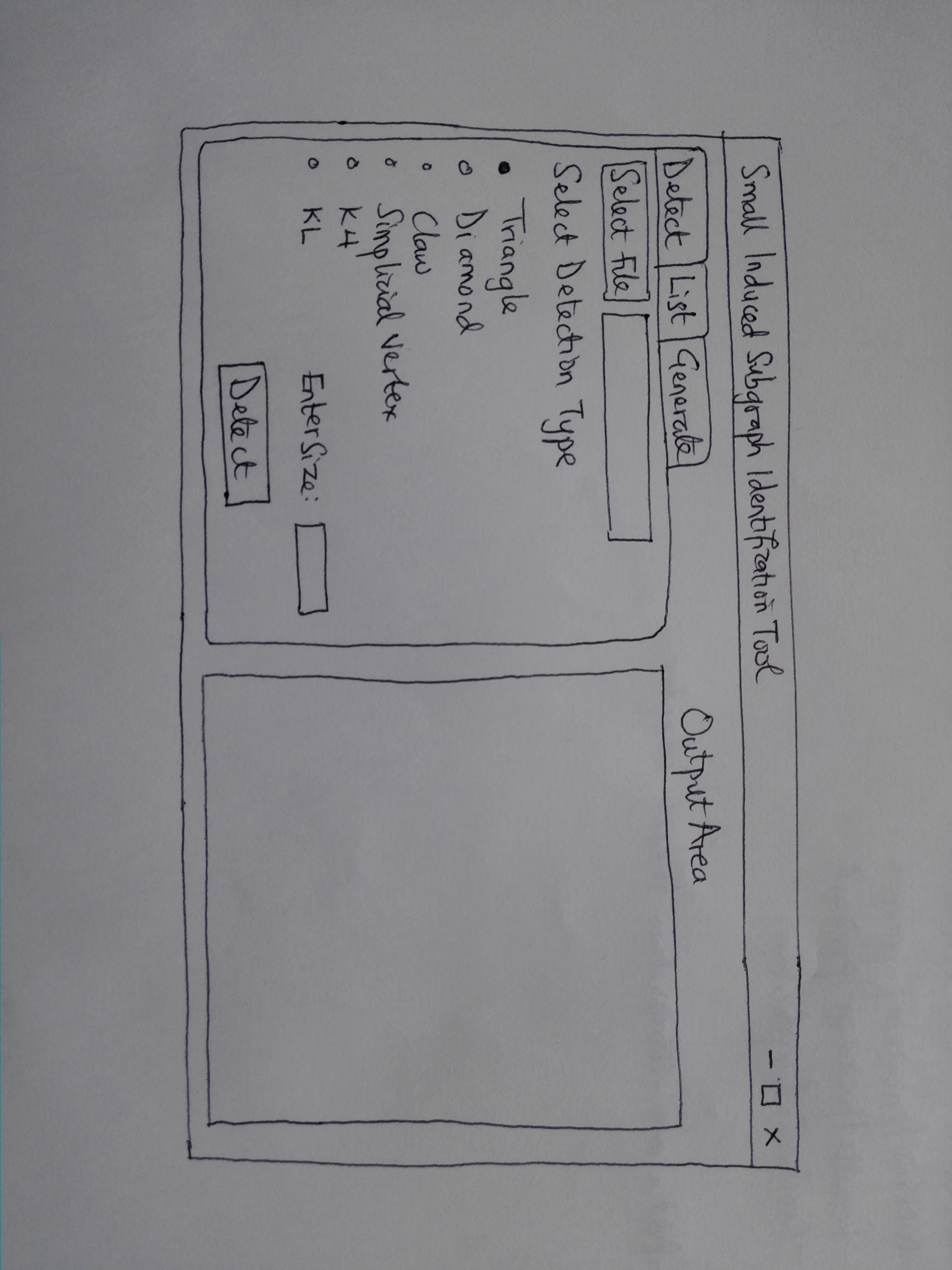


Figure 4.1: Wireframe of the graphical user interface of the tool

## Overview of Algorithms

### Matrix Multiplication Algorithm

Most of the algorithms described in [11] make use of matrix multiplication. To achieve the best performance when these algorithms are implemented, the best matrix multiplication algorithm should be used. The Coppersmith-Winograd matrix multiplication algorithm has the best theoretical time complexity, but has already been stated in section 2.4.2 to be complicated and has no known implementation in practice. Strassen‘s algorithm was also considered but the setbacks discussed in section 2.4.1 prevented it from being used. As a result of these, matrix multiplication was done using the standard matrix multiplication algorithm. The matrices which are to be multiplied are all square matrices with equal dimensions. This algorithm is given in algorithm 1 below.

Algorithm 1: Standard Matrix Multiplication Algorithm (from [13])

Input: matrices *A* and *B*, each matrix having a dimension of *n* by *n*

1. Let *C* be a new matrix with dimension *n* by *n*
2. **For** *i* = 1,…,*n*, **do**:
3. **for** *j*=1,….,*n***, do**:
4. let *sum* = 0
5. **for** *k*= 1,…,n, **do**:
6. add to *sum*
7. **end for**
8. set to *sum*
9. **end for**
10. **end for**
11. **terminate** yielding *C*

The standard matrix multiplication algorithm has a time complexity of .

### Triangle Detection Algorithm

A lot of the detection algorithms described below involve the recognition of the presence of a triangle in a given graph or subgraph. A common triangle detection algorithm relies on matrix multiplication to achieve a time complexity of as described in section 2.5 earlier. The square of the adjacency matrix of a graph reveals pairs of vertices which have one or more s between them. A is a path with 3 vertices. If such vertices are found and there exists an edge between them as well, then this indicates the presence of a triangle in the graph. The pseudo-code of this algorithm is provided below.

Algorithm 2: To detect a triangle in graph *G* with *n* vertices

**input:** the graph *G*

1. Get the adjacency matrix *A* of *G*
2. Calculate
3. **for** *i* = 0,1,…,n-1, **do**:
4. **for** *j* = *i*+1,…,n-1, do:
5. **if** *A*(*i, j*) = 1 and *A2*(*i ,j*) > 0, **do**:
6. **for** *k* = 0,1,…,*n*-1, **do**:
7. **if** *A*(*k, i*) = 1 and *A*(*k, j*) = 1:
8. **return** (*i, j, k*)
9. **end if**
10. **end for**
11. **end if**
12. **end for**
13. **end for**
14. **return** nothing

**Time complexity analysis**

Step 1 takes *O*( time. Step 2 involves matrix multiplication and will take *O*( time, where ω depends on which matrix multiplication algorithm is used. The outer loop in step 3 requires *n* iterations. The inner loop in step 4 requires iterations. The inner loop in step 6 will run only once and has *n* iterations, and it is guaranteed that *k* will be found once this algorithm processor gets to this loop, after which the algorithm terminates. As a result, the total time complexity of steps 3 to 13 is *O*(*n*2 + *n*) and therefore a combined time complexity of *O*(*n*2). The entire algorithm has a combined running time of *O*( and therefore is asymptotically *O*(

### *P*3 Detection Algorithm

*P*3 detection is useful in the algorithm given to detect an induced diamond in a graph. A *P*3 is a path of length 2. Like the triangle detection algorithm, it has a time complexity of as both algorithms are similar but with only a slight difference between them. The pseudo-code of this algorithm is given in Algorithm 3.

Algorithm 3: To detect a *P*3 in graph *G* with *n* nodes

**input:** the graph *G*

1. Get the adjacency matrix *A* of *G*
2. Calculate
3. **for** *i* = 0,1,…,n-1, **do**:
4. **for** *j* = *i*+1,…,n-1, do:
5. **if** *A*(*i, j*) = 0 and *A2*(i, j) > 0, **do**:
6. **for** *k* = 0,1,…,*n*-1, **do**:
7. **if** *A*(*k, i*) = 1 and *A*(*k ,j*) = 1:
8. **return** (*i ,j, k*)
9. **end if**
10. **end for**
11. **end if**
12. **end for**
13. **end for**
14. **return** nothing

**Time Complexity Analysis**

Due to the similarity with the triangle detection algorithm, the same loops are involved. The difference exists in step 5 of both algorithms. While the triangle detection algorithm checks if *A*(*i, j*) = 1, the *P*3 detection algorithm checks if *A*(*i, j*) = 0. Step 5 does not have any significant contribution to the overall time complexity hence it is still .

### *P*3 Listing Algorithm

The *P*3 detection algorithm is modified so that each *P*3 found is added to a list, rather than terminating the algorithm as soon as a *P*3 is found. The list of all the *P*3sfound is then returned at the end of the algorithm. Since a *P*3 may be found more than once, a measure is put in place to check if a *P*3 has already been found before adding it to the list.

### Triangle Listing Algorithm

Triangle listing is particularly useful in the complete subgraph detection algorithm as described in [18]. The triangle detection algorithm could easily be modified to list all triangles found. However this will lead to the listing of the same triangle more than once. Duplication can be avoided by keeping a collection of already found triangles and checking if the collection already contains a new triangle before it is added. This additional check increases the time required to list all triangles, since the time to perform the check increases as more triangles are added to the collection. A more efficient algorithm for triangle listing that does not rely on matrix multiplication was presented in [4]. This algorithm as stated in section 2.5 has a time complexity of *O*(), where is the arboricity and *m* is the number of edges. Algorithm 4 describes the process of listing all triangles in a graph.

Algorithm 4: To list all triangles in a graph (taken from [4])

**input:** the graph *G*

1. Make list *triangles* empty.
2. Sort the vertices *v1, v2,…, vn* in non-increasing order of their degrees.
3. **for** *i* = 1,2,…n-2, **do**:
4. mark all vertices adjacent to *vi*
5. **for each** marked vertex *u*, **do**:
6. **for** each vertex *w* adjacent to *u*, **do**:
7. **If** *w* is marked:
8. add triangle (*vi, u, w*) to *triangles* list
9. **end if**
10. Erase mark from *u*.
11. **end for**
12. **end for**
13. Delete vertex *vi*  from *G* and let *G* be the resulting graph
14. **end for**
15. **terminate** yielding *triangles*

### Simplicial Vertices Listing Algorithm

The algorithm to list the simplicial vertices in a graph was derived in [11]. For a vertex *x* to be simplicial, for all the neighbours *y* of *x*, the closed neighbourhood of *y* must subsume the closed neighbourhood of *x.* The vertices of the graph are partitioned into low and high degree vertices such that each low degree vertex has a degree that is less than or equal to *D* where . Vertices with degrees greater than *D* are high degree vertices. The algorithm operates in phases.

**Phase 1:** For each low degree vertex, each pair of vertices in its neighbourhood is checked to ascertain if they are adjacent. If that is the case, then that vertex is a simplicial vertex. This phase of the algorithm has a time complexity of where *m* is the edge count of the graph and *D* is as defined above.

**Phase 2: E**ach high degree vertex that has a low degree vertex as a neighbour is marked as not being simplicial because according to [11], if a simplicial vertex is of high degree, then all its neighbours will be of high degree. All the low degree vertices are then removed from the graph. We get the 0/1-adjacency matrix, *A*, of the resulting graph, put 1’s on the diagonal and compute *A2*. For each vertex *x* in the resulting graph, if *x* is not marked, we check if *A2*(*x*, *y*) = *A2*(*x,* *x*) for all *y* ∈ *N*(*x*). If the condition holds, then *x* is a simplicial vertex. This phase of the algorithm has a time complexity of .

The combined time complexity of the algorithm is [11]. Algorithm 5 in appendix A shows the pseudo-code of this algorithm.

### Simplicial Vertex Detection Algorithm

The algorithm to detect the presence of a simplicial vertex in a graph is similar to the simplicial vertices listing algorithm. The difference is that rather than adding all the simplicial vertices to the list and returning the list at the end of the algorithm, a simplicial vertex is returned as soon as it is found and the algorithm terminates.

### K4 Detection Algorithm

The *K4* (complete graph with 4 vertices) detection algorithm implemented was described in [11]. It was designed around the idea that an induced *K4* in a graph would most likely be made up of high degree vertices. Hence to find a *K4*, the neighbourhood of each of the high degree vertices is checked for the presence of an induced triangle that is made up of high degree vertices. The presence of such a triangle indicates the presence of a *K4*, and the vertices which induce the *K4* can be obtained by getting the vertices of the triangle found, and adding the high degree vertex whose neighbourhood contained the triangle.

If no *K4* is found, then a *K4*if present in the graph has at least one low degree vertex as one of its vertices. To find this *K4*, we check the neighbourhood of each of the low degree vertices for the presence of a triangle. The vertices of the triangle if found together with the low degree vertex induce the *K4*.

The algorithm has a time complexity of . Algorithm 6 in Appendix A shows the pseudo-code of this algorithm.

### K4 Listing Algorithm

To list all the complete graphs of size 4, we modify the *K*4 detection algorithm to add each K4 found to a list, and then return this list at the end of the algorithm. This means that all of the triangles in the neighbourhood of a vertex will have to be listed. We make use of Algorithm 4 to list the triangles. A similar approach used in the *P*3 listing algorithm is adopted to prevent replication.

### Claw Detection Algorithm

The claw detection algorithm was designed around the fact that a claw-free graph will have at most neighbours according to [11]. Therefore we first check the graph for a vertex that has a degree greater than . If such a vertex is found, then a claw which contains that vertex must be present in the graph. The vertex with that degree is called the central vertex of the claw. We then look for a triangle in the complement of the neighbourhood of the central vertex. A claw is thus induced by the vertices of the triangle and the central vertex.

However if there is no vertex whose degree is greater than , then for each vertex of the graph, we check if the complement of the neighbourhood contains a triangle. If we find a triangle, then the vertices of that triangle together with the vertex whose neighbourhood’s complement contains the triangle will induce a claw. The algorithm has a time complexity of as indicated in [11], where *m* is the number of edges in the graph.

Algorithm 7 in Appendix A shows the pseudo-code for the claw detection algorithm.

### Claw Listing Algorithm

To list all claws, we modify the claw detection algorithm to add each claw found to a list, and then return this list at the end of the algorithm. We check the complement of the neighbourhood of each vertex of the graph for the presence of triangles using Algorithm 4. The vertices of each found triangle together with the low degree vertex in whose neighbourhood’s complement the triangle was found induce a claw. Each claw found is added to a list, which is returned at the end of the algorithm.

### Diamond Detection Algorithm

The algorithm to detect a diamond in a graph as described in [11] works as follows. The graph’s vertices are partitioned into low degree and high vertices. The low degree vertices are those vertices whose degree is less than or equal to the square root of the edge count of the graph. The other vertices are high degree vertices.

**Phase 1:** For each low degree vertex, we find the subgraph induced by its neighbours, compute the components of the subgraph and check if each component is a clique. If a component is not a clique, then it should contain a *P*3. The vertices of the *P*3 together with the low degree vertex induce a diamond. The time complexity of this phase is where *m* is the edge count of the graph and *D* is the square root of the edge count [11].

**Phase 2:** If no diamond is found in phase 1, then for each of the low degree vertices, we have the cliques in its neighbourhood as obtained in phase 1. We check if each pair of vertices in each clique has another neighbour apart from the low degree vertex whose neighbourhood the clique belongs to. If we find such a neighbour, then a diamond is present in the graph. Such a diamond is induced by the low degree vertex, the pair of vertices in the clique found in the neighbourhood of the low degree vertex, and the neighbour of that pair of vertices found outside the low degree vertex’s neighbourhood. The time complexity of this phase is where *m* and *D* are as defined in phase 1 and *n* is the size of the graph [11].

**Phase 3:** If no diamond is found in phase 2, then we remove all the low degree vertices from the graph and apply the procedure described in phase 1 to all the vertices in the resulting graph. The phase has a time complexity of .

The combined time complexity of the algorithm is therefore [11]. The pseudo-code for this algorithm is shown in Algorithm 8 in Appendix A.

### Diamond Listing Algorithm

To list all diamonds, we modify the diamond detection algorithm such that phase one is applied to all the vertices of the graph rather than only the low degree vertices. If a component of the neighbourhood graph of a vertex is not a clique, then for each *P*3 found in the component, the vertices of the *P*3together with the vertex in whose neighbourhood the component was found induce a diamond. We add each diamond found to a list and return this list at the end of the algorithm. Since phase one is applied to all the vertices of the graph, there will be no need for the other phases to be included in the algorithm. The approach taken in the *P*3 listing algorithm is also utilized to avoid replication of diamonds.

### Complete Subgraph Listing Algorithm

The algorithm to list complete subgraphs of size *l* implemented in this project was formulated in [18]. Such a subgraph will be subsequently referred to as a *Kl.* A naïve algorithm to list complete graphs of size *l* would have a time complexity of . The *K*l listing algorithm was built around the triangle detection algorithm with the aim of improving the time requirement of *Kl* detection to where ω is the exponent of matrix multiplication and *i* is the remainder when *l* is divided by 3. The algorithm works as follows:

Given the task to find a *Kl* in a graph *G,* if *l* is a multiple of 3, then an auxiliary graph, *H*, is built, such that:

* *H* contains a vertex if and only if *G* contains a complete graph of size
* *H* contains an edge if and only if *G* contains a complete graph of size

To achieve this, *G* is searched for complete subgraphs of size . For each of such subgraphs found, a vertex is added to *H*. There should be a way to keep track of which vertices in *H* correspond to which complete subgraphs of size in *G*. Then for each complete subgraph of size found in *G*, an edge should be added between the corresponding vertices in *H*. Since the order in which the vertices of the complete subgraph of size are listed does not matter, edges between several vertices in *H* would be added. A check is put in place to ensure that only one edge is added per complete subgraph of size as this will not affect the final output and will make the algorithm run faster in practice. This check could be in the form of a list to store the vertices of *G* that correspond to an edge in *H* so that another edge in *H* that corresponds to the same vertices in *G* would not be added.For values of *l* greater than 5, the algorithm will have to operate recursively to construct *H*. After *H* is constructed, it is then searched for triangles. Each triangle found in *H* corresponds to a complete subgraph of size in *G*. The triangles can be found using Algorithm 4.

If *l* is not a multiple of 3, then two scenarios are possible.

1. **The remainder, *i*, when *l* is divided by 3 is 1**. In such cases, for each vertex in *G*, its neighbourhood is checked for the presence of complete subgraphs of size by applying the procedure described above. If such complete subgraphs are found, then adding the vertex in whose neighbourhood the complete subgraphs were found to the vertices of each subgraph will produce the complete subgraph of size *l.* For example if *l* = 7, then *i* = 1. We then check the neighbourhood of each vertex for the presence of complete subgraphs of size 6. The vertices of each of such subgraphs if found together with the vertex in whose neighbourhood the complete subgraph was found will induce a complete subgraph of size 7.
2. **The remainder, *i*, when *l* is divided by 3 is 2**. For each edge in *G,* we find the subgraph induced by the neighbours common to the vertices that are connected by the edge. We then check for the presence of complete subgraphs of size in such neighbourhood subgraph by applying the procedure described above. If such complete subgraphs are found, then adding the vertices that are connected by the edge to the vertices of each of the found complete subgraphs will produce a complete subgraph of size *l*.

The nature of the algorithm gives it the ability to list triangles and *K*4 subgraphs as well.

### Complete Subgraph Detection Algorithm

The algorithm to detect complete subgraphs of size *l* described in [11] uses the complete subgraph listing algorithm of [18]. The vertex set of graph, *G*, is partitioned into low degree and high degree vertices, where high degree vertices are those vertices with degree greater than *D*.

The algorithm has two phases. In phase one, we find the subgraphs induced by the neighbours of each low degree vertex, and check whether each of such neighbourhood subgraphs contains a complete subgraph of size by using the complete subgraph listing algorithm of [18]. We stop as soon as we find such complete subgraph in any neighbourhood. Then the *K*l-1 found together with the low degree vertex in whose neighbourhood the subgraph was found form a complete subgraph of size *l.* If no *Kl* is found in phase one, then we find the subgraph induced by the set of all high degree vertices and check if the subgraph contains a *Kl*by applying the complete subgraph listing algorithm of [18].

Combining both phases gives a time complexity of *O*(), where if *l* is a multiple of 3, and *O*() otherwise.

### Depth First Search Algorithm

Depth first search was used in the diamond detection algorithm to compute the components of a subgraph. The depth first search algorithm traverses one path as far as possible, before other paths are traversed [16]. The Depth First Search algorithm has a time complexity of , where *m* is the number of edges and *n* is the number of vertices in the graph [7]. The pseudo-code of the algorithm is given in algorithm 9.

Algorithm 9: Depth-First Traversal (from [16])

To traverse a graph *G* in depth-first order, starting at vertex *start*

1. Make *vertex-stack* contain only vertex *start*, and mark *start* as reached
2. **While** *vertex-stack* is not empty, **repeat**:
3. Remove the top element of *vertex-stack* into *v*
4. Visit vertex *v*
5. **For each** unreached successor *w* of vertex *v*, **repeat**:
6. Add vertex *w* to *vertex-stack­*, and mark *w* as reached
7. **end for**
8. **terminate**

### Random Pattern-free Graph Generation Algorithms

The algorithms to detect the presence of the different induced subgraphs earlier described were utilized to develop algorithms which generate random graphs that do not contain a selected induced subgraph type.

Initially, an algorithm to generate a random graph given a desired graph size and the edge probability (which determines how dense the graph will be) was used to generate multiple graphs which were used to run evaluations on the tool to analyze its performance. This algorithm creates an adjacency matrix to represent the graph with all its values being 0. Then a number between zero and one is randomly “guessed”. If this number is less than the edge probability, then a 1 is added at the appropriate positions of the adjacency matrix to represent an edge between two vertices. This is repeated until all pairs of vertices have been traversed. The algorithms have a time complexity of , where *n* is the size of the graph to be generated, and *C* is the time complexity of the respective detection algorithm.

Our algorithm to generate a random graph that is free from a particular type of induced subgraph, e.g. a diamond, is almost similar to the ordinary random graph generation algorithm just described. It however differs in two ways.

1. Only the desired number of vertices in the graph was provided, the edge probability was not provided.
2. After the addition of an edge, the resulting graph is then checked for the presence of a diamond. If a diamond is found, then that edge is removed. The process is repeated until all pairs of vertices have been traversed.

### Brute force Detection and Listing Algorithms

In order to perform evaluations on how the algorithms described in [11] perform in practice, brute force algorithms to detect and list simplicial vertices as well as the other types of induced subgraphs were designed. The performances of the algorithms of [11] are then compared with those of the brute force algorithms.

**Brute Force Triangle Detection and Listing Algorithm**

For each edge, *e*, in the graph, we check each vertex, *v,* in the graph to determine if both vertices at the ends of *e* are adjacent to *v*. If they are, then a triangle is found. To list all triangles, we repeat the check for all edges in the graph. Each triangle will be found three times, so we use a set to keep track of already found triangle vertices to prevent a triangle from being added to the list of found triangles more than once. The algorithm has a time complexity of where *m* is the number of edges and *n* is the number of vertices in the graph [10].

**Brute Force Diamond Detection and Listing Algorithm**

For each vertex, *v*, in the graph, we check its neighbourhood graph for the presence of a *P*3 by applying Algorithm 3. If a *P*3 is found, then the graph contains a diamond. The neighbourhood graph may have multiple *P*3s and each of these *P*3s together with *v* form a diamond.The process is repeated for all the vertices in the graph to find all diamonds. . A set is used to store already found diamonds to prevent the addition of the same diamond to the final list of found diamonds more than once.

**Brute Force Simplicial Vertices Detection and Listing Algorithm**

For each vertex, *v*, in the graph, we check if its neighbourhood graph is a clique. If it is, then *v* is simplicial. By repeating the process for all vertices in the graph, all the simplicial vertices can be found.

***K*4 Detection and Listing Algorithm**

For each vertex, *v*, in the graph, we check if its neighbourhood graph contains a triangle. If it does, then *v*, together with the vertices of the triangle found, form a *K*4. The neighbourhood graph may have multiple triangles and each of these triangles together with *v* form a *K*4. The process is repeated for all the vertices in the graph to find all *K4*s. The replication prevention approach described in the brute force diamond listing was also applied here.

**Claw Detection and Listing Algorithm**

For each vertex, *v*, in the graph, we find the complement of its neighbourhood graph. We then check this complement for the presence of triangles. Each triangle found together with *v* form a claw. By repeating the process for all vertices in the graph, all the claws present in the graph are found.

**Complete Subgraph Detection and Listing Algorithm**

To detect complete subgraphs of size *l* in a graph, we get the neighbourhood graph of each vertex *v* of the graph and recursively check if the neighbourhood graph contains a complete subgraph of size . The base condition of the recursion is reached when *l* = 1, at which point any vertex found in the subgraph at that recursion depth is returned.

To list all the complete subgraphs of size *l*, we change the base condition to return all the vertices found in the subgraph that is passed at that depth of the recursion tree. At the other depths of the recursion tree, the algorithm checks the subgraph that is passed to it for complete subgraphs of size passed at that depth and returns them. For example to list all complete subgraphs of size 10 in a graph, at the first depth of the recursion tree, the algorithm returns all subgraphs of size 9. At the second depth, the algorithm returns all subgraphs of size 8. This process continues until it gets to the base condition.

## Conclusion

In this chapter, the major algorithms which were used in the tool were discussed. The next chapter will include discussion on how the implementation of the tool was carried out.

# Implementation

In this chapter, we will discuss how the tool was implemented, the programming language that was used, and the challenges that were encountered during the implementation.

## Development Methodology

The development was divided into three parts; the first part involved the detection of induced subgraphs, the second part involved the design of algorithms to list all the induced subgraphs of a type found in a graph, while the third part involved the generation of random graphs that were free from induced subgraphs of a desired pattern. The detection part was implemented first, as it included the requirements with “must have” priority. The listing part was done next, while the generation part was developed after, in line with it having requirements with lower priorities.

Version control was used throughout the development process, as the code is kept updated on a Github repository, allowing changes to be easily tracked. In addition, it provided a form of security and backup, ensuring that progress made in the coding does not get lost in the case of any accidents or damages.

## Language of Implementation

The tool was developed using the Java programming language. Java has a huge advantage due to its platform independence, enabling the tool to be used on most popular operating systems.

## Graph Abstract Data Type (ADT) Implementation

The outcome of the background study on the different graph representations done in section 2.3 was taken into consideration in the implementation of the graph ADT class. It is important to point out that the tool is meant to work with undirected graphs only.

Initial implementation included the use of doubly linked lists to represent the vertex and edge sets. The contract of the ADT was specified in the Graph interface, which contained inner interfaces for the vertex and edge contracts.

The UnVertex class contains a reference to the element held by the vertex object, as well as pointers to the previous and next vertex objects in the linked list. The UnEdge class contains a reference to the attribute of the edge, together with pointers to the previous and next edge objects in the linked list. The UndirectedGraph class then contained references to the first and last vertices in the list and a reference to the first edge in the list. It also had variables to keep track of the size of the graph and the number of edges in the graph. This representation had some properties which include:

1. Time complexity of *O*(*m*) in checking whether two vertices are adjacent, where *m* is the number of edges in the graph.
2. Time complexity of *O*(*m*) in constructing the adjacency list of a vertex.
3. Time complexity of *O*(*n*) to get the vertex object that contains a given element, where *n* is the number of vertices in the graph.

It is important to get the vertex object that contains a given element because the elements being unique within a graph serve as a means to identify the vertices. The algorithms implemented require that new subgraphs are constructed from a given graph, and we needed a way to identify vertices that are “the same” although they are present in different graphs. The vertex elements presented a way to achieve this.

Due to these setbacks, the representation was changed. The new representation retained the use of doubly linked lists to represent the edge set as it made it easy to access all the edges in the graph. The vertex set representation was changed to make use of a hash map. The references to the first and last vertices and the size variable were removed as they were no longer required. The size of the graph is gotten by returning the size of the hash map. The vertex class was also modified to contain a list of a vertex’s neigbours, so that a vertex object held a collection of references to the edges that were incident on it. The changes resulted in the following improvements:

1. The time complexity to check if two vertices *u* and *v* are adjacent improved to *O*(min{*du,dv*}), as described in Section 2.3.2
2. The time complexity to get the adjacency list of a vertex was improved to *O*(*d*), where *d* is the maximum degree of a vertex in the graph.
3. Time complexity of *O*(1) to get the vertex object that contains a given element.

The vertices in the graph are easily obtained by returning an iterator for the values of the vertex set hash map. The degree of a vertex is obtained by returning the size of its adjacency list. Adding an edge involves inserting the edge object at the front of the doubly linked list that represents the edge set, as well as adding the reference to the edge object to the adjacency lists of the vertices connected by the edge. Edge addition is a constant time operation. Removing an edge involves removing the edge object from the DLL representing the edge set, and removing the references to the edge from the adjacency lists of the vertices connected by the edge. The methods to perform breath first search and depth first search were implemented in the graph ADT class.

Inner private classes which represent iterators for iterating through the edges of a graph, the connecting edges of a vertex, and the neighbours of a vertex were included in the graph ADT class.

The adjacency matrix of a graph is obtained by calling the getAdjacencyMatrix method also implemented in the UndirectedGraph class. The method returns the adjacency matrix as a two-dimensional array of integers. The operation to compute the adjacency matrix has a time complexity of , where *n* is the size of the graph and *d* is the maximum degree of a vertex. This is because each pair of vertices of the graph is checked for the presence of an edge between them. The complement matrix of a graph can also be obtained in a similar way by calling the getComplementMatrix method of the UndirectedGraph class

## Matrix Multiplication Algorithm Implementation

The implementation of the matrix multiplication followed Algorithm 1 of section 4.3. However some modifications to optimize the algorithm were made. The idea for the modifications was gotten from [19]. The modifications however do not change the time complexity of the algorithm. These modifications include:

* changing the order in which the *for* loops are executed, specifically swapping the *for* loops in steps 3 and 5.
* Introducing extra variables to the algorithm. These variables are ira, irc, krb and ikA shown in the code snippet in Figure 5.1.

**int**[][] result = **new** **int**[a.length][b[0].length];

**for**(**int** i = 0; i < a.length; i++) {

**int**[] ira = a[i];

**int**[] irc = result[i];

**for**(**int** k = 0; k < a.length; k++) {

**int**[] krb = b[k];

**int** ikA = ira[k];

**for**(**int** j = 0; j < a.length; j++) {

irc[j] += ikA \* krb[j];

}

}

}

Figure 5.1: Optimized standard matrix multiplication algorithm in Java (from [19])

These modifications increase cache use by the algorithm as well as the memory access patterns [13], which in turn speed up the performance in practice. The performance improvement can be seen in the chart in Figure 5.2. The benchmarking test was run on a computer with an Intel(R) Core(TM) i5-2410M CPU.

Figure 5.2: Chart showing average matrix multiplication time before and after standard matrix multiplication algorithm optimization

## Implementing the Subgraph Detection and Listing Algorithms

Each algorithm for detecting a type of induced subgraph was placed in its own class. Also, each algorithm for listing a type of induced subgraph was placed in its own class. Classes which had algorithms that were broken down into phases used methods to represent the phases. The methods that were commonly used by these classes were placed in the Utility class to prevent code duplication across multiple classes. The Utility class contains methods for accessing the file system to load graph files as well as to save the generated graph files, thus ensuring communicational cohesion is maintained. Cohesion simply means ‘keeping things together that belong together’ [12]. Communicational cohesion is achieved if the facilities for operating on the same data are kept together. Increasing cohesion is one of the design principles in software engineering [12].

Implementing the detection algorithms were quite straight-forward, except the algorithm for detecting complete subgraphs. This is because in order to detect a complete subgraph of size *l*, **all** the complete subgraphs of size would have to be found first using the complete subgraph listing algorithm. And to list all complete subgraphs of size , all complete subgraphs of size would have to be found first. This results in a recursive algorithm which introduces additional complexities and thus increases the overall running time of the detection program. The methods in the detection and listing classes which represent the phases of the respective algorithms are left as public methods because the unit test classes require access to these methods to test that the methods function as operated.

When the detection or listing type is specified either via the command line interface or the graphical user interface, the methods in the Utility class perform validations on the input file specified. The validations include checking whether the file exists and whether the adjacency matrix contained in the file has consistent number of rows and columns. If the input file passes the validations, then an UndirectedGraph instance is created from the adjacency matrix, an instance of the appropriate detection or listing class is created, and the UndirectedGraph instance created earlier is passed to the detect method of the detection or listing class. This detect method returns the list of vertices which induce the desired type of subgraph to be detected or listed. If any of the validations fails, then a GraphFileReaderException object containing details of the failure is thrown to the caller of the method of the Utility class, which then handles the exception by notifying the user via a command line output if a user was interacting with the tool via the command line interface or a JOptionPane if the user was using the graphical user interface.

Each of the detection and listing classes includes a means of measuring the time taken to execute the operation. This time is useful in evaluating the performance of the tool when applied to graphs of different sizes and densities.

## Implementing the Random Pattern-free Graph Generation Algorithms

The algorithms to generate graphs that do not contain a selected induced subgraphs type were very similar. As a result, all the algorithms were implemented in the PatternFreeGraphGenerator class, with each algorithm implemented in a different method of the class. After the required graph is generated, it is automatically saved in a text file and the name and location of this file is returned to the user.

## Implementing the Command Line Interface

The interaction with the tool via the command line is done by running the RunMe class. When this class is run, the user is prompted to enter a couple of arguments separated by a space. The first argument represents the class of operation the user wants to perform. The second argument represents the operation class type. The third argument represents the path of the graph file that the user wants to perform an operation on if the user wants to detect or list subgraphs or simplicial vertices, or the size of the graph to be generated if the user wants to generate graphs. The fourth argument represents the size of the complete subgraph to be detected, listed or generated. Some operation classes do not require the second third and fourth arguments. Figure 5.3 shows a snapshot of the command line interface of the tool.

The command line interface uses threads to perform detection, listing and generate operations. This is to enable the user to enter other commands while an operation is executing. Specifically, this is to enable the user to cancel an operation while it is running. The use of threads was added after receiving feedback on the usability of the tool from Craig Reilly.

Details on the use of the command line interface are provided in the user manual in Appendix D.

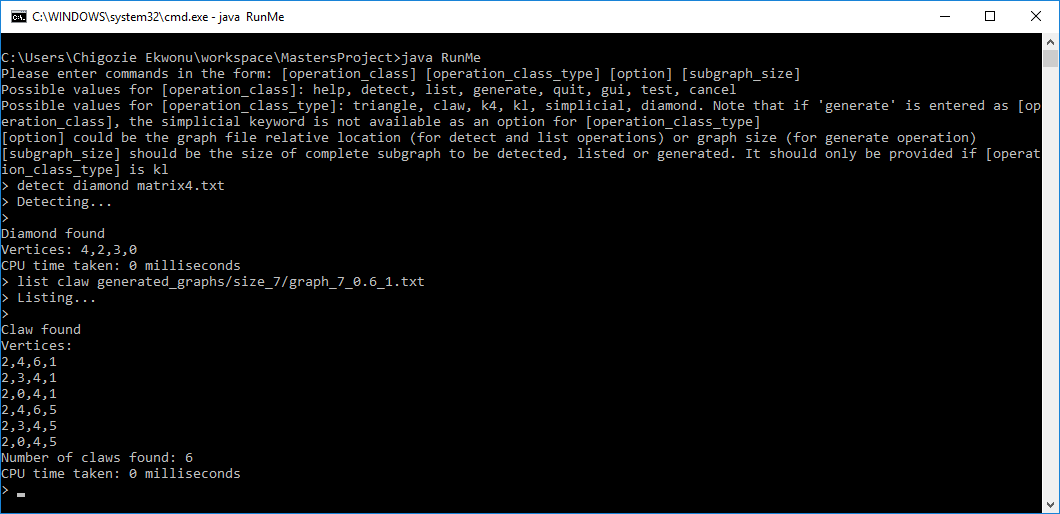


Figure 5.3: Command line interface of the tool

## Graphical User Interface (GUI) Implementation

The GUI was implemented after the command line interface was done and it follows the model-view-controller (MVC) architecture. The model part was developed as part of the command line interface of the tool. A controller class to handle events generated by the view class was written.

The GUI has an output area implemented using a JEditorPane which displays the output of the tool that would otherwise be displayed on the command line window if the command line interface was used. The GUI makes use of JRadioButton to represent the types of subgraphs to be detected or listed. The GUI uses a JFileChooser which allows the user to navigate to the location of the text file containing the adjacency matrix of the input graph.

The GUI can be displayed by entering the keyword “gui” as the first argument in the command line tool. Figure 5.4 shows a snapshot of the GUI. Depending on the size of the input graph, the detection and listing operations could take a long time to execute. To prevent the GUI from becoming unresponsive, the detection, listing and generation operations are handled by SwingWorker objects, which present a safe way to make use of threads in java’s Swing package. The Swing package is a popular java package which contains classes for building graphical user interfaces.

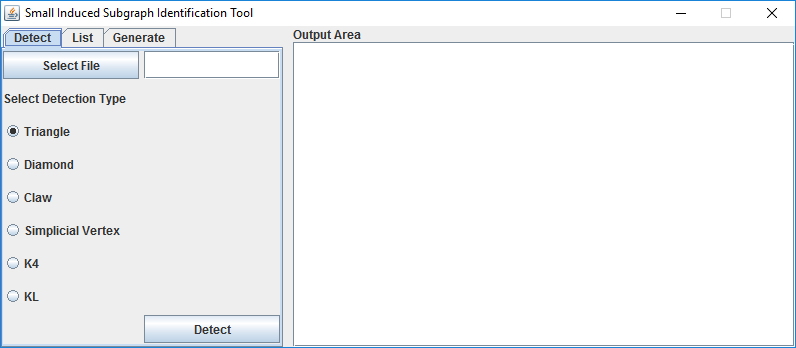


Figure 5.4: Graphical User Interface of the Tool

## Implementation Challenges

### Fast Matrix Multiplication Algorithm Implementation

The algorithms in [11] rely on the Coppersmith-Winograd fast matrix multiplication algorithm to achieve the desired efficiency. However as stated in Section 2.4.2, Coppersmith-Winograd matrix multiplication algorithm exists only in theory. Due to the complexity surrounding the algorithm, it could not be implemented in practice.

Strassen’s matrix multiplication algorithm when tested in practice was slower than the standard matrix multiplication algorithm. It was stated in [5] that the crossover point, i.e. the matrix size above which Strassen’s algorithm performs better than the standard matrix multiplication algorithm was system dependent. This value was found to be as low as 8 and as high as 2150 in different systems. In some other systems, the value could not be found at all [5]. Including benchmarking code to determine the value of the crossover point to be used for the system that this tool is to be used on would require more time than is available for completion of this project. This is left as possible future work.

As a result, we had to make use of the standard matrix multiplication algorithm in this project. The standard matrix multiplication was optimized to improve the use of cache memory which led to performance speed ups.

### Subgraph Replication

Apart from the algorithms to list triangles and simplicial vertices, the other listing algorithms are modifications of the detection algorithms of their respective subgraph types. These detection algorithms were not designed by their authors to return the vertices of more than one subgraph found. When they were modified to do so, some of the subgraphs became replicated.

To solve the replication challenges, a collection was used to hold the already found subgraphs. An ArrayList object was initially used for this collection. Each subgraph found is then checked whether it has already been found before it is added to the list. Checking this list introduced additional complexity, increasing the program execution time. The program execution time was reduced by replacing the ArrayList object with a HashSet object. Adding objects to a HashSet and checking if an object is already present in a HashSet are constant time operations. This efficiently solved the problem.

### Multiple Edges Addition in the Complete Subgraph Listing Algorithm

In the complete subgraph listing algorithm, in order to detect the presence of a complete subgraph of size *l* in graph *G*, an auxiliary graph *H* was constructed. According to the algorithm in Section 4.3.14, while constructing *H*, multiple edges were added between different pairs of vertices in *H* for each complete subgraph of size found in *G*. When a triangle is then found in *H*, the vertices of *G* which are mapped to the vertices in the triangle found in *H* induce a complete subgraph of size *l* in *G*. It was noticed that due to the multiple edges, multiple triangles in *H* were mapped to the same vertices in *G*, leading to replication. Also, it took a longer time to list complete subgraphs that are present in *G*.

To overcome the replication challenge and reduce the execution time, the implementation was modified such that only one edge is added between vertices in H for each complete subgraph of size found in *G.* This was done by using a HashSet object to store the vertices of *G* that correspond to an edge in *H* once the edge is added. Before a new edge is added to *H*, this HashSet object is checked to prevent the addition of a new edge which corresponds to the same set of vertices in *G* that has already been found.

### Implementing Algorithms to Generate Random Pattern-free Graphs

After implementing the algorithms to generate random graphs that do not contain a specified subgraph type, the following were observed.

1. The graphs generated were sparse because an edge is removed from the graph if the resulting graph contains the specified induced subgraph type.
2. The algorithm when applied to generate random graphs with no simplicial vertices generated graphs with no edges. This is because once an edge exists between two vertices, a simplicial vertex will be found and thus that edge will be removed. As a result, the tool is unable to generate random graphs that do not contain any simplicial vertex.
3. The algorithm is not suitable for generating large random pattern-free graphs because of the time it would take to check for the presence of the specified induced subgraph type for each edge added.

**Conclusion**

We have discussed how we implemented the tool and the challenges which we encountered. In the next chapter, we will detail how testing was carried out and the evaluation that was done.

# Testing and Evaluation

In this chapter, we discuss how the tool was tested and evaluated. We also analyze the result of the evaluation.

## Testing Strategy

The different classes for detecting and listing the vertices of the subgraphs found were developed iteratively. Each of the classes has a main method, which enabled us to quickly test the functionality of a class and reveal any problems found. Including a main method in each class is one of the ways of designing for testability [12].

## Unit Testing

Unit tests for the ADT class were written after the graph ADT class was developed to test each of the methods of the class as it was important that the methods were functioning properly before we could develop the classes for performing detection and listing of subgraphs. The unit tests ensured that methods of the ADT class which failed to operate as expected were quickly identified. Unit tests for the detection and listing classes were written later after each of the classes had been developed. Files containing the adjacency matrices of specifically created graphs were provided as input to the unit test classes. With these input files, we know which vertices induce which type of subgraph and were thus able to specify the expected results in the unit tests. The unit tests are found in the test package of the tool’s source code.

## System Testing

Unit tests could not be done on the command line and graphical user interfaces of the tool because the inputs provided to the tool would depend on the user. However efforts were made to ensure that both the command line and the graphical user interfaces of the tool are functioning as expected for valid inputs, and notify the user of invalid inputs when detected. The results of the system testing are provided in Appendix E.

## Evaluation Strategy

The evaluation of the tool was in the form of experiments comparing the efficient algorithms described in [11] with easy to implement brute force algorithms to detect the same types of induced subgraphs. The experiments to compare the efficient algorithms and the brute force algorithms were carried out on the School of Computing’s Eglinton computer which has 20 CPU cores.

An algorithm to generate random graphs of different sizes and densities was designed and implemented. This algorithm was used to generate graphs with up to 1000 vertices. For each graph size, we generated graphs with edge probabilities of 0.2, 0.4, 0.6, 0.8 and 1.0. We generated 10 graphs for each edge probability except for probability of 1.0 which had only one graph generated for it. This is because such a graph would be a complete graph and there was no point in generating multiple complete graphs with the same size.

Since each of the detection and listing classes has a main method, Unix *Makefiles* which take each of the randomly generated graph files and passes the file as a command line argument to each class were written. The main method in each detection class prints out the time taken to run the detection as well as the result to the command line terminal, indicating whether the desired subgraph type was found. The main method in each listing class prints out the time taken to run the detection as well as the number of simplicial vertices or the number of a desired subgraph type found to the command line terminal. The *Makefile* tool then writes this output to a text file for each graph file tested. After the experiments were completed, the output text files were then combined into a single text file for each subgraph class. The code that combines the results is present in the CombineResults class in the test package.

Each of the detection and listing algorithms was run once per graph on Eglinton, but since the randomly generated graphs consisted of 10 graphs for each graph density and graph size, the average time per graph density and graph size was taken. By making use of the “-j20” option of the unix *make* command, we hoped that Eglinton would utilize its 20 CPU cores to make the experiments run faster by running the detection on 20 graphs at the same time.

### Evaluation of the Efficient Detection Algorithms

The tool was evaluated for two things: ability to correctly detect the presence of the presence of a selected induced subgraph type and the time taken to detect such a subgraph if any. The ability to correctly detect the presence of a type of induced subgraph was easy to measure manually for graphs with smaller vertex count. The unit tests written for the detection classes easily took care of that. For both small and large graphs, comparison of the results of the efficient detection algorithms and the brute force detection algorithms revealed that the tool could correctly identify the presence of a selected subgraph type. Benchmarking tests for the efficient and brute force detection algorithms were run on each of these generated graphs and the time taken for each algorithm to run as well as whether the required type of subgraph was found for each graph was recorded.

It was discovered in the initial stages of the experiments that for large graphs, it could take more than 90 minutes to run the claw detection algorithm on a complete graph with 800 vertices and finally decide that it does not contain a claw. In order to cut down on the experiment run times, the diamond detection and claw detection algorithms were excluded from running on complete graphs. This is because by the definition of induced subgraphs, complete graphs cannot contain diamonds and claws as induced subgraphs.

From the results of the experiments, it was discovered that the algorithms described in [11] when implemented did not perform better than the brute force algorithms except when used to detect simplicial vertices. This was attributed to the matrix multiplication which was done in most of the detection classes. Our brute force algorithms did not rely on matrix multiplication, and as a result performed better that the efficient detection algorithms for detection operations.

While the efficient algorithm would have to perform matrix multiplication to detect a triangle, the brute force algorithm could quickly find a triangle without multiplying matrices. The brute force triangle detection algorithm was still quite efficient with an efficiency of with *m* and *n* being as defined in previous chapters. In a complete graph, the brute force algorithm will detect a triangle after checking the first three vertices in the graph, irrespective of the size of the graph. The efficient algorithm would have to multiply matrices, and the time to perform the multiplication increases in direct proportion to the matrix size. This gives the brute force detection algorithm a huge advantage over the efficient detection algorithm. As each of the classes to detect diamonds, claws and other complete subgraphs relied on triangle detection; the delay associated with matrix multiplication was propagated among the other detection classes, making them slower than their brute force counterparts.

The performance of the efficient algorithm to detect simplicial vertices over its brute force counterpart could be attributed to the reduced number of matrix multiplications performed in the efficient simplicial vertices detection algorithm. Specifically, matrix multiplication is done only once, and this was in the second phase of the algorithm. As the size and density of the graph increases, the brute force algorithm takes a longer time to test whether the neighbours of a vertex form a clique. When implementing the brute force detection algorithms, a timeout of 30 seconds was configured, in order to prevent the algorithms from taking an extremely long time to execute. Although the detection experiments were run on graphs with sizes of up to 1000, the charts presented here show the results for graphs with a maximum size of 100 in order to provide a clearer picture. Microsoft excel files containing the raw data are submitted along with this report. Figure 6.1 shows a chart which displays the trend of the average times taken to detect simplicial vertices using efficient and brute force algorithms. Other charts showing the results for the detection experiments for other subgraph classes are shown in Appendix B.

### Evaluation of the Efficient Listing Algorithms

The output of the tool was modified for the listing experiments to show the time taken to list all the subgraphs of a selected type present in a graph and how many of such subgraphs were found. For the simplicial vertices listing, the output showed the time taken to list all simplicial vertices found and the number of such vertices.

It was anticipated that the listing algorithms would take longer times to execute than the detection algorithms, hence the listing algorithms experiments were run on graphs with maximum size of 100. The correctness of the number of simplicial vertices and number of subgraphs of a selected type found using the efficient listing algorithms was established by comparing it with the number found by the corresponding brute force listing counterparts. Both numbers were found to be the same. From the outcomes of the experiments, most of the efficient listing algorithms were discovered to run faster than their brute force counterparts. This could be attributed to the use of a triangle listing algorithm which does not rely on matrix multiplication in the triangle listing class. Other listing classes apart from the simplicial vertices listing class and the diamond listing class relied on being able to find all triangles in a subgraph and as a result encountered faster runtimes.

Figure 6.1: Chart Showing Detection Times for the Efficient and Brute force Simplicial Vertices Detection Algorithms

The diamond listing class however still had to employ matrix multiplication to find all *P*3s in a subgraph and hence did not receive much performance increase when compared with the brute force diamond listing algorithm. This discovery substantiates our suggestion that the matrix multiplication which was used in some of the efficient detection algorithms was responsible for the slowdown in their performance. Figure 6.2 shows the trend of the average time taken to detect claws when both efficient and brute force algorithms were applied. Other charts showing the results of the listing experiments are available in Appendix B.

### Evaluation of the Random Pattern-free Graph Generation Algorithms

The algorithms to generate random graphs that do not contain a given subgraph were evaluated by simply specifying the size of the graph to be generated, and then providing the generated graph as input to the tool to check whether the graph generated is free from the desired subgraph type. As mentioned in section 5.9.4, the random pattern-free graph generation algorithms are not suitable to be used to generate large graphs due the time taken to check for the presence of a selected subgraph type after each edge is added to the randomly generated graph.

When very large values are provided as the size of the graph to be generated, the java virtual machine (JVM) runs out of heap memory and an OutOfMemoryError is reported.

Figure 6.2: Chart showing claw listing times for efficient and brute force listing algorithms

This was noticed on a computer with 8Gigabytes of RAM when a graph size of 13,000 was entered while trying to generate a random graph that does not contain a triangle. Since the memory allocated to the JVM heap is dependent on the configurations of the machine that the tool is run on, we cannot predict a suitable maximum graph size to place an upper bound on. The tool has been developed to correctly report this error.

### Evaluation of the Usability of the Tool

Towards the end of the project, the tool was presented to Craig Reilly and Dr. Ruth Hoffmann in order to provide feedback on the usability of the tool. Dr. Ruth Hoffman is a research assistant at the School of Computing. Both of them agreed that the tool was easy to use.

Craig pointed out that for the command line interface, rather than having a single class which provides an interface to access the different detection, listing and generation classes, he preferred being able to access the classes directly so that the output can be manipulated using other unix tools. However since the detection and listing classes have main methods, they can also be accessed directly from the command line terminal. Craig suggested incorporating a means of cancelling the detection, listing and generation operations without exiting the program because of the long time that may be taken for an operation to execute. The tool was modified to incorporate this. Some of the suggestions which were received were incorporated into the tool. Dr. Hoffman suggested that the number of the different classes of subgraphs that can be detected are limited and suggested extending the tool to allow detection of custom graphs. This is left as future work.

Conclusion

The status of the project, what has been achieved, personal reflection and future work will be discussed in this chapter.

## Status of the Product

The tool developed satisfied all but one of the functional requirements stated in Section 3.1. The requirement that was not satisfied is the ability to generate random graphs that do not contain simplicial vertices and this requirement has a ‘could have’ priority. The reason why this requirement was not satisfied was stated in section 5.9.4.

## What Has Been Achieved

The tool developed during this project has the ability to detect the presence of triangles, claws, diamonds, and complete subgraphs present in a given graph as induced subgraphs and list all of such induced subgraphs found. It is also capable of detecting and listing all simplicial vertices found in a graph. The algorithms implemented were described in [11]. It was discovered that fast matrix multiplication algorithms which the detection algorithms were built upon are complicated to implement in practice and do not necessarily perform better than the standard matrix multiplication algorithm. As a result, algorithms to detect the presence of only one of such induced subgraphs which are not built around matrix multiplication were discovered to perform better than those which relied on matrix multiplication. However when the algorithms of [11] were modified to list **all** the induced subgraphs found, with the modification involving less matrix multiplication they were found to be run faster than the brute force listing algorithms.

## Personal Reflection

By undertaking this project, I have learnt quite a lot about graph theory. My knowledge about algorithms and their complexities has also been improved. I have learnt that there are other ways in which two matrices can be multiplied together. My knowledge of unix commands has been improved as I had to make use of such commands to set up and automate experiments on the School of Computing’s powerful Eglinton computer.

## Future Work

As it was stated earlier in Section 2.4.1 that Strassen’s matrix multiplication algorithm produces better performance over the standard matrix multiplication algorithm for matrix sizes over a crossover point, the tool can be extended to include a benchmarking tool that is capable of calculating this crossover point for the system that the tool is to be used on. The method to multiply matrices in our tool can then be modified to utilize Strassen’s algorithm such that Strassen’s algorithm is used once a matrix size above the found crossover point is found.

The algorithms to generate random graphs that do not contain certain types of induced subgraphs as mentioned in Section 4.9.4 generated sparse graphs. Further research could be done into developing algorithms to achieve the same purpose which will be able to generate dense graphs.

The tool could be extended to include the ability to detect the presence of custom subgraphs, i.e. graphs that are not constrained to a particular defined shape like triangles or diamonds.

Visualization of the input graph as well as the vertices and edges which induce an induced subgraph of a selected type could be added to the tool. This requirement was assigned a “would like to have” priority in Section 3.1.

References

[1] Alon, N., Yuster, R., & Zwick, U. *Finding and counting given length cycles*. Algorithmica, 17(3), 209-223. 1997.

[2] Bondy, J. A., & Murty, U. S. *Graph theory with applications*. New York: American Elsevier Pub. 1976.

[3] Carlson, S. C. *Graph Theory*. Retrieved July 19, 2016, from <https://www.britannica.com/graph-theory>

[4] Chiba, N., & Nishizeki, T. *Arboricity and Subgraph Listing Algorithms*. SIAM J. Comput. SIAM Journal on Computing, 14(1), 210-223. 1985.

[5] Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. *Introduction to algorithms*. Cambridge, MA: MIT Press. 2009

[6] *Cycle (graph theory)*. Retrieved August 24, 2016, from <https://en.wikipedia.org/wiki/Cycle_(graph_theory)>

[7] Goodrich, M. T., & Tamassia, R. *Algorithm design and applications*. Wiley. 2015

[8] *Graph-tool Efficient network analysis*. Retrieved July 26, 2016, from <https://graph-tool.skewed.de/performance>

[9] Harary, F. *Graph theory*. Reading, MA: Addison-Wesley Pub. 1969

[10] Itai, A., & Rodeh, M. *Finding a Minimum Circuit in a Graph.* SIAM J. Comput. SIAM Journal on Computing, 7(4), 413-423. 1978

[11] Kloks, T., Kratsch, D., & Müller, H. *Finding and counting small induced subgraphs efficiently*. Information Processing Letters, 74(3-4), 115-121. 2000

[12] Lethbridge, T. C., & Laganière, R. *Object-oriented software engineering: Practical software development using UML and Java*. London: McGraw-Hill. 2001

[13] *Matrix multiplication algorithm*. Retrieved August 23, 2016, from <https://en.wikipedia.org/wiki/Matrix_multiplication_algorithm>

[14] Mckay, B. D., & Piperno, A. *Nauty and Traces User’s Guide (Version 2.6).* Retrieved July 28, 2016, from <http://users.cecs.anu.edu.au/~bdm/nauty/nug26.pdf>

[15] Mckay, B. D., & Piperno, A. *Practical graph isomorphism*, II. Journal of Symbolic Computation, 60, 94-112. 2014

[16] Miller, A. *Graph ADTS lecture slides*. Lecture presented in School of Computing, University of Glasgow.

[17] Miller, A. *Abstract Data Types lecture slides*. Lecture presented in School of Computing, University of Glasgow.

[18] Nesetril, J., & Poljak, S. *On the complexity of the subgraph problem*. Comment. Math. Univ. Carolinae, 14, 415-419. 1985

[19] *Optimizing Matrix Multiplication*. Retrieved August 23, 2016, from <https://www.daniweb.com/programming/software-development/code/355645/optimizing-matrix-multiplication>

[20] Poet, R. *Use Cases Software Project Management lecture slides*. Lecture presented in School of Computing, University of Glasgow.

[21] *Quick start using graph-tool*. Retrieved July 26, 2016, from <https://graph-tool.skewed.de/static/doc/quickstart.html>

[22] Ray, S. S. *Graph theory with algorithms and its applications: In applied science and technology*. New Delhi: Springer. 2013

[23] Schank, T., & Wagner, D. *Finding, Counting and Listing All Triangles in Large Graphs, an Experimental Stud*y. Experimental and Efficient Algorithms Lecture Notes in Computer Science, 606-609. 2005

[24] *NetworkX Tutorial*. Retrieved July 26, 2016, from <http://networkx.readthedocs.io/en/networkx-1.11/tutorial/>

[25] Watt, D. A., & Brown, D. F. *Java Collections: An Introduction to Abstract Data Types, Data Structures, and Algorithms*. John Wiley. 2001

[26] Williams, V. V. *Multiplying Matrices in O(n2.373) time*. Retrieved July 28, 2016, from <http://theory.stanford.edu/~virgi/matrixmult-f.pdf>

###### Algorithms Pseudo-code

This appendix contains pseudo-code of some of the algorithms utilized in this project. It serves as a continuation of the algorithms in chapter 4 which could not be included as a result of space constraints.

Algorithm 5: Listing simplicial vertices in a Graph (taken from [11])

**Input:** the graph *G*

1. Get vertex set, *V,* of *G*
2. Make lists *H* and *L* empty.
3. Set where is the exponent of matrix multiplication and m is edge count
4. **for** each vertex *v* in *V*, **do**:
5. **if** degree of *v* > *D*:
6. Add *v* to *H*
7. **else:**
8. Add *v* to *L*
9. **end if**
10. **end for**
11. Create list *simplicials*
12. **for** each *x* in *L,* **do**:
13. **if** each pair of the neighbours of *x* is adjacent:
14. add *x* to *simplicials*
15. **end** **if**
16. **end for**
17. create list *marked*
18. **for** **each** *x* **in** *H*, **do**:
19. **if** any of the neighbours of *x* is in *L*:
20. add *x* to *marked*
21. **end if**
22. **end for**
23. **for** **each** *x* **in** *L*, **do**:
24. Remove *x* from *G*. Let *G\** be the resulting graph
25. **end for**
26. Set *A* to the adjacency matrix of *G\** with 1’s on the diagonal
27. Calculate *A2*
28. **for** *x* **in** *G\** vertices, **do**:
29. **if** *x* is not in *marked*, **do**:
30. **if** *A2*(*x*, *y*) = *A2*(*x,* *x*) for all *y* adjacent to *x*:
31. add *x* to *simplicials*
32. **end if**
33. **end if**
34. **end for**
35. **terminate** yielding *simplicials*

**Time complexity analysis**

The *for* loop in step 4 will take *O*(*n*) time to execute, where *n* is the number of vertices in *G*. The *for* loop in step 12 will take steps, which according to [11] has an upper bound of . Hence this *for* loop will take *O*(*mD*) time to execute. According to [11] also, the *for* loops in steps 18 and 23 can be executed in linear time. After step 23, graph *G\** now has at most vertices of high degree [11]. As a result, step 23 will require time to complete. Step 28 will also have a running time of , since each *x* will have at most () vertices. Combining these complexities will result in a total running time of ) which yields .

Algorithm 6: To Detect a K4 in a graph

**input:** the graph G

1. Get vertex set, *V*, of *G*
2. Make lists *H* and *L* empty
3. Set *D* = where *m* is the edge count of *G*
4. **for each** vertex *v* **in** *V,* **do**:
5. **if** degree of *v* > *D*:
6. add *v* to *H* ≫ *H* is a list of high degree vertices
7. **else**
8. add *v* to *L* ≫ *L* is a list of low degree vertices
9. **end if**
10. **end for**
11. **for each** vertex *x* **in** *H*, **do**:
12. get the neighbourhood *N*(*x*) of *x*
13. compute *G*[*N*(*x*) ∩ *H*]
14. get the adjacency matrix of *G*[*N*(*x*) ∩ *H*]
15. use algorithm 1 to detect a triangle in *G*[*N*(*x*) ∩ *H*]
16. **if** a triangle is found:
17. get list *k4list* of vertices that correspond to *i*, *j*, *k*
18. add *x* to *k4list*
19. **return** *k4list*
20. **end** **if**
21. **end** **for**
22. **for each** vertex x **in** L, **do**:
23. get the neighbourhood N(x) of x
24. get the adjacency matrix of *G*[*N*(*x*)]
25. use algorithm 1 to detect a triangle in *G*[*N*(*x*)]
26. **if** a triangle is found:
27. get list *k4list* of vertices that correspond to *i*, *j*, *k*
28. add *x* to *k4list*
29. **return** *k4list*
30. **end if**
31. **end for**
32. **terminate** yielding none

**Time complexity analysis**

The vertex partitioning operation between steps 4 to 10 will take *O*(*n*) time, where *n* is the number of vertices in the graph. According to [11], steps 11 to 21 will take time while steps 22 to 31 will take time. This will result in a combined running time of , based on the value chosen for *D*, with *m* is the edge count of graph.

Algorithm 7: To detect a Claw in a Graph

input: the graph G

1. for each vertex x in G, do:
2. if degree of x is greater than :
3. get the neighbourhood graph N(x) of x
4. get the complement graph C(x) of N(x)
5. use algorithm 1 to detect a triangle in C(x)
6. if a triangle is found:
7. get list *claw* of vertices that correspond to *i*, *j*, *k*
8. add *x* to *claw*
9. **return** *claw*
10. end if
11. end if
12. end for
13. for each vertex x in G, do:
14. get the neighbourhood graph N(x) of x
15. get the complement graph C(x) of N(x)
16. use algorithm 1 to detect a triangle in C(x)
17. if a triangle is found:
18. get list *claw* of vertices that correspond to *i*, *j*, *k*
19. add *x* to *claw*
20. **return** *claw*
21. end if
22. end for
23. terminate yielding none

Algorithm 8: To detect a Diamond in a Graph

**Input:** the graph G

1. Get vertex set, *V*, of *G*
2. Make lists *H* and *L* empty
3. Set *D* = where *m* is the edge count of *G*
4. **for each** vertex *v* **in** *V,* **do**:
5. **if** degree of *v* > *D*:
6. add *v* to *H* ≫ *H* is a list of high degree vertices
7. **else**
8. add *v* to *L* ≫ *L* is a list of low degree vertices
9. **end if**
10. **end for**
11. make *cliques* map empty
12. **for each** vertex *x* **in** *L*, **do**:
13. make list *vertexcliques* empty
14. get the neighbourhood *N*(*x*) of *x*
15. get the components *comps* of *N*(*x*)
16. for each component *c* in *comps*, do:
17. if *c* is a clique:
18. add *c* to *vertexcliques*
19. else
20. get a *P*3 in component
21. make list *diamond* containing vertices of the p3 found
22. add x to *diamond*
23. return *diamond*
24. end if
25. end for
26. add *vertexcliques* to *cliques* with *x* as the key
27. end for
28. Get the adjacency matrix *A* of *G*
29. Compute *A*2
30. for each vertex *x* in *L*, do:
31. get the cliques *vertexcliques* in the neighbourhood of *x* from *cliques*
32. for each clique *C* in *vertexcliques*, do:
33. for each pair of vertices y, z in *C*, do:
34. if *A*2(*y, z*) > |*C*| - 1:
35. find *w* such that *w* is not in *N*[*x*]
36. make list *diamond* and add *x*, *y*, *z* and *w* to it
37. return diamond
38. end if
39. end for
40. end for
41. end for
42. Remove all vertices in *L* from *G* and call the resulting graph *G*\*
43. Repeat steps 11 to 27 on all vertices of G\* to check if a diamond is found.

###### Evaluation Result Charts

This appendix contains charts showing performance of the efficient algorithms implemented in the tool against brute force algorithms when the different subgraph type detections and listings are carried out on graphs of different sizes and densities.

Figure B.1: Chart Showing Average Claw Detection Times

Figure B.2: Chart Showing Average Diamond Detection Times

Figure B.3: Chart Showing Average K4 Detection Times

Figure B.4: Chart Showing Average Triangle Detection Times

Figure B.5: Chart Showing Average Diamond Listing Times

Figure B.6: Chart Showing Average K4 Listing Times

Figure B.7: Chart Showing Average Simplicial Vertices Listing Times

Figure B.8: Chart Showing Average Triangle Listing Times

Figure B9: Chart showing average time taken to generate pattern-free graphs

###### UML Class Diagrams

**C1** **efficient.detection package**

Figure C1-C6 show the class diagrams for classes in the efficient.detection package

**C2** **efficient.listing package**

Figure C7-C12 show the class diagrams for classes in the efficient.listing package

**C3** **exception package**

Figure C13-C14 show the class diagrams for classes in the detection package

**C4** **general package**

Figure C15-C17 show the class diagrams for classes in the general package

**C5** **generate package**

Figure C18-C19 show the class diagrams for classes in the generate package

**C6** **gui package**

Figure C20-C24 show the class diagrams for classes in the gui package

**C6** **default package**

Figure C25-C28 show the class diagrams for classes in the default package

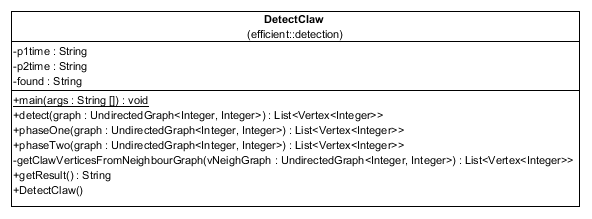


Figure C1: Class diagram for DetectClaw.java

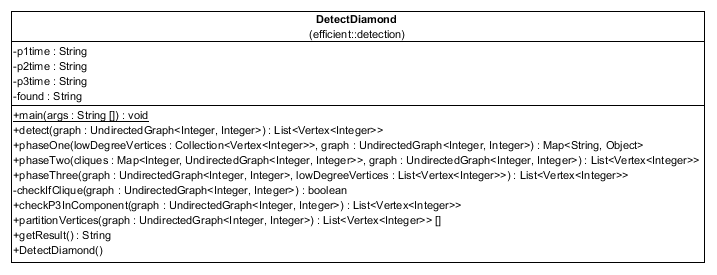


Figure C2: Class diagram for DetectDiamond.java

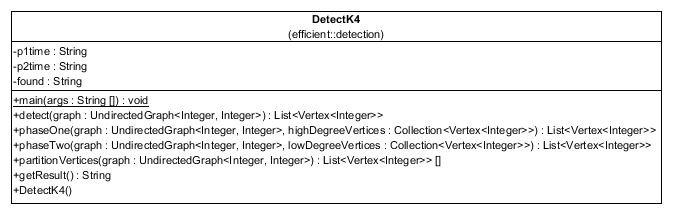


Figure C3: Class diagram for DetectK4.java

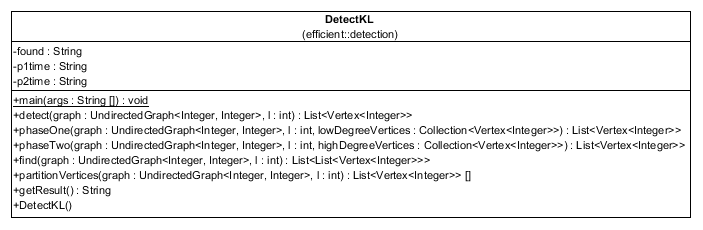


Figure C4: Class diagram for DetectKL.java

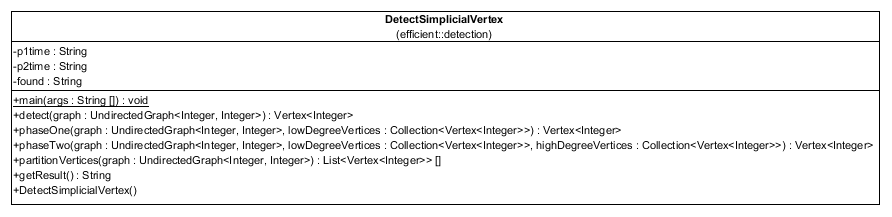


Figure C5: Class diagram for DetectSimplicialVertex.java

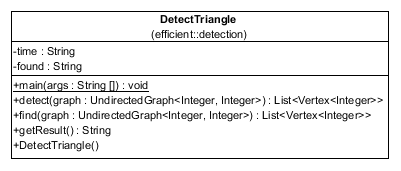


Figure C6: Class diagram for DetectSimplicialVertex.java

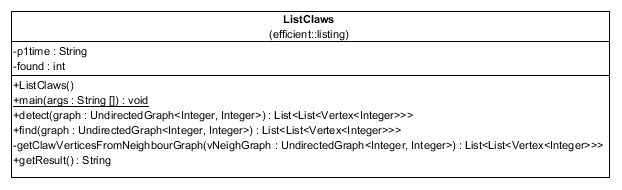


Figure C7: Class diagram for ListClaws.java

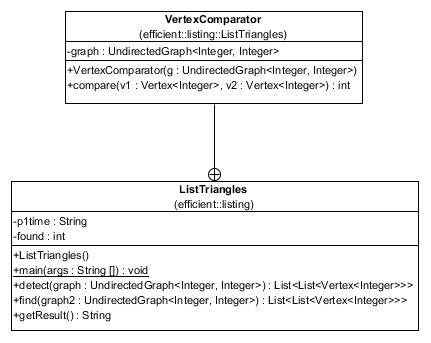


Figure C8: Class diagram for ListTriangles.java

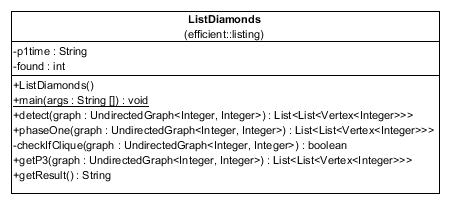


Figure C9: Class diagram for ListDiamonds.java

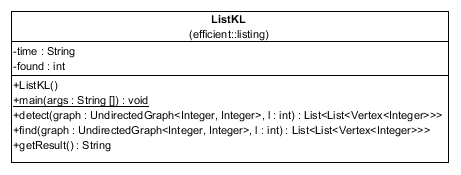


Figure C10: Class diagram for ListKL.java

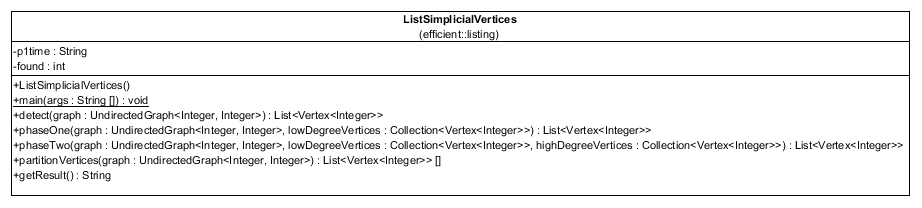


Figure C11: Class diagram for ListSimplicialVertices.java



Figure C12: Class diagram for ListK4.java

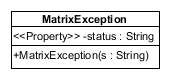


Figure C13: Class diagram for MatrixException.java

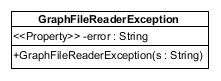


Figure C14: Class diagram for GraphFileReaderException.java

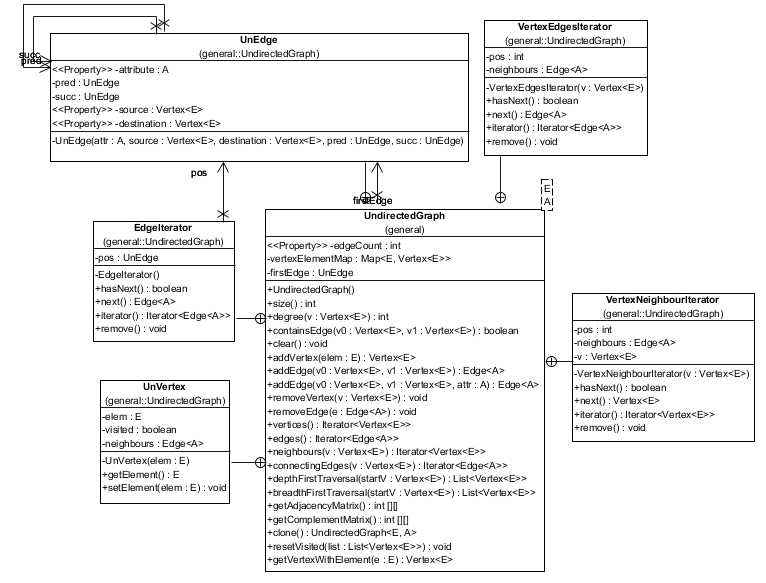


Figure C15: Class diagram for UndirectedGraph.java

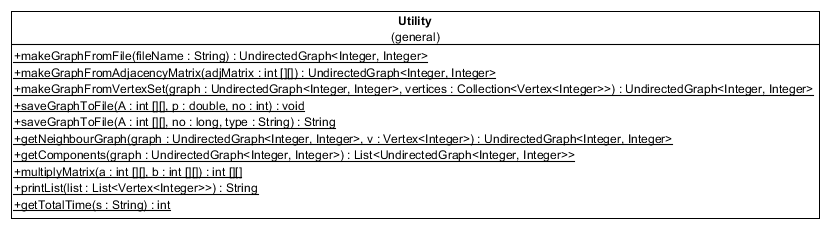


Figure C16: Class diagram for Utility.java

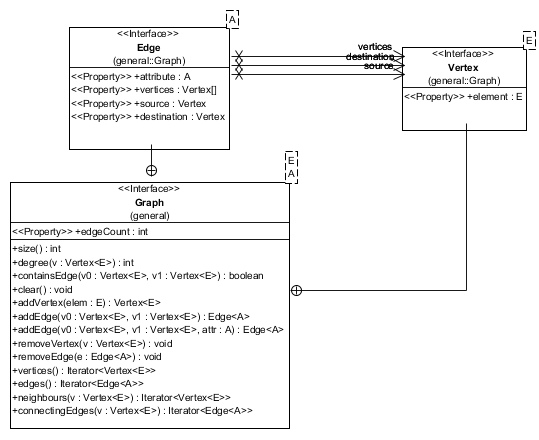


Figure C17: Class diagram for Graph.java

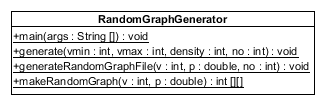


Figure C18: Class diagram for RandomGraphGenerator.java

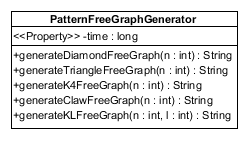


Figure C19: Class diagram for PatternFreeGraphGenerator.java

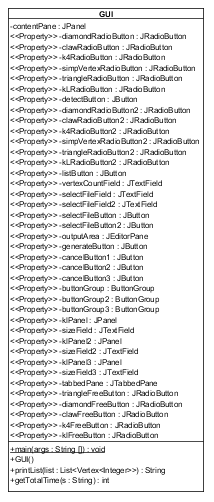


Figure C20: Class diagram for GUI.java

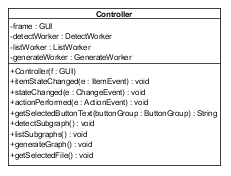


Figure C21: Class diagram for Controller.java

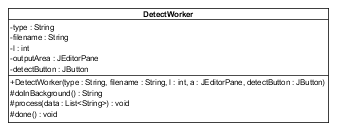


Figure C22: Class diagram for DetectWorker.java

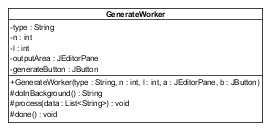


Figure C23: Class diagram for GenerateWorker.java

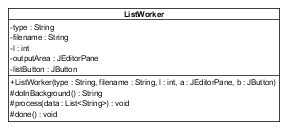


Figure C24: Class diagram for ListWorker.java

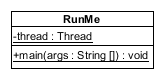


Figure C25: Class diagram for RunMe.java

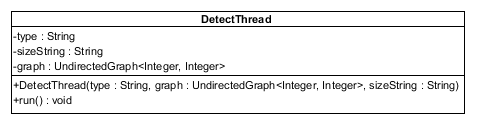


Figure C26: Class diagram for DetectThread.java



Figure C27: Class diagram for GenerateThread.java

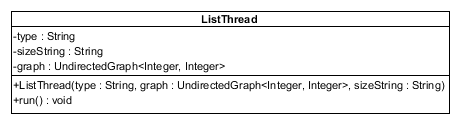


Figure C28: Class diagram for ListThread.java

###### User Manual

This appendix contains the user manual on how to set up and use the tool.

**D.1 User Manual**

This software was developed by Chigozie Ekwonu as a Masters project to fulfill the requirements for the award of a Masters in Software Development at the University of Glasgow.

The tool is used to detect the presence of specific subgraph types – triangles, claws, diamonds and other complete subgraphs as induced subgraphs in a graph and lists the vertices which induce such subgraphs. It also lists all simplicial vertices found in a graph. The software was tested on Windows 7 and Windows 10. Its command line interface was also tested on Linux.

**D.1.1 Installing the Software**

###### System Testing Results and Screenshots

This appendix contains screenshots showing the behavior of the tool for different inputs and operation types.

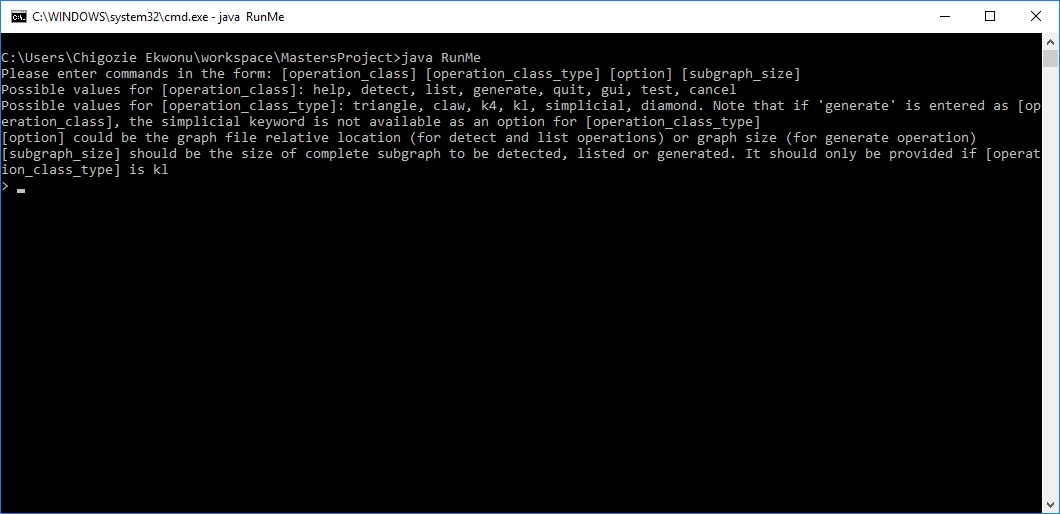


Figure E1: Command line interface after launch

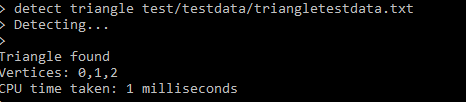


Figure E2: Command line interface after running triangle detection on valid data



Figure E3: Command line interface showing output after a non-existing input file is entered

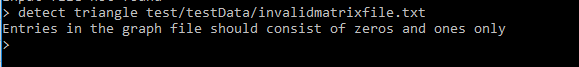


Figure E4: Command line interface showing output after an input file containing values apart from zeros and ones



Figure E5: Command line interface showing output after an input file with inconsistent number of columns in the adjacency matrix

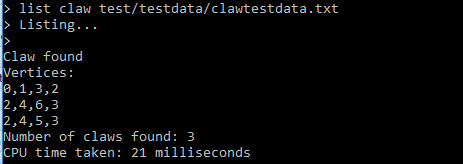


Figure E6: Command line interface showing output after running claw listing on valid data

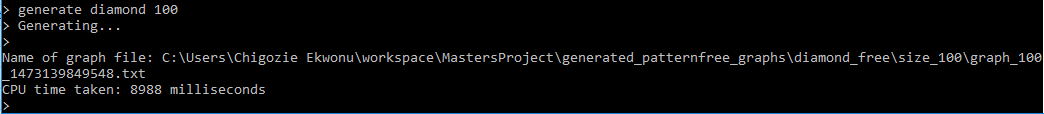


Figure E6: Command line interface showing time taken to generate a graph of size 100 that is diamond free

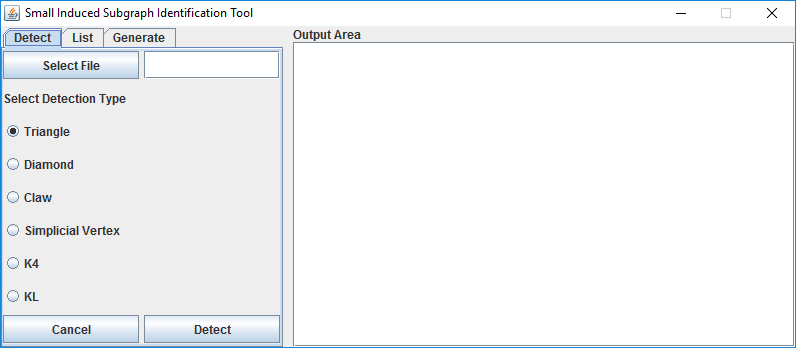


Figure E7: The Graphical user interface when launched

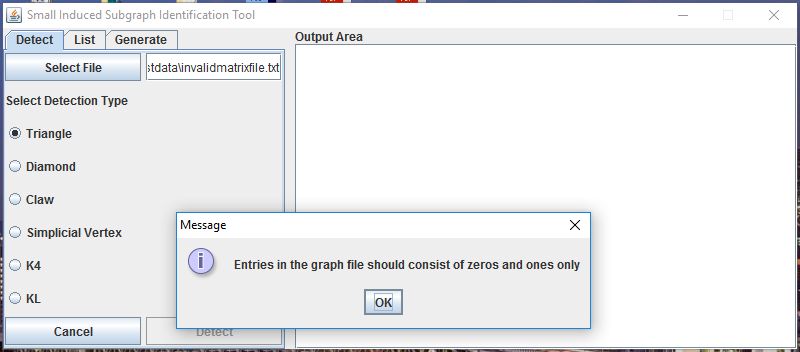


Figure E8: GUI notifying user of an invalid input file

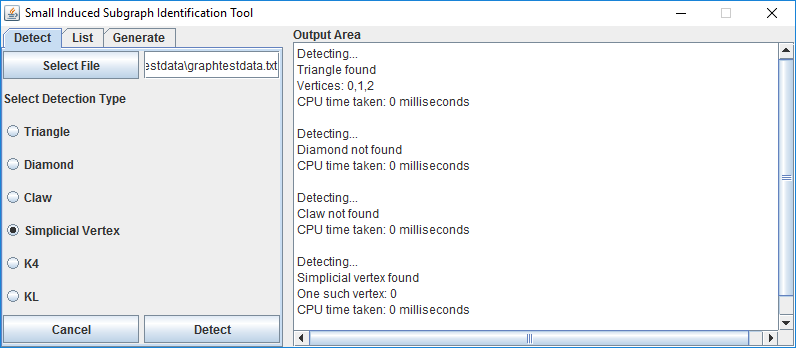


Figure E9: GUI after some detection operations have been done

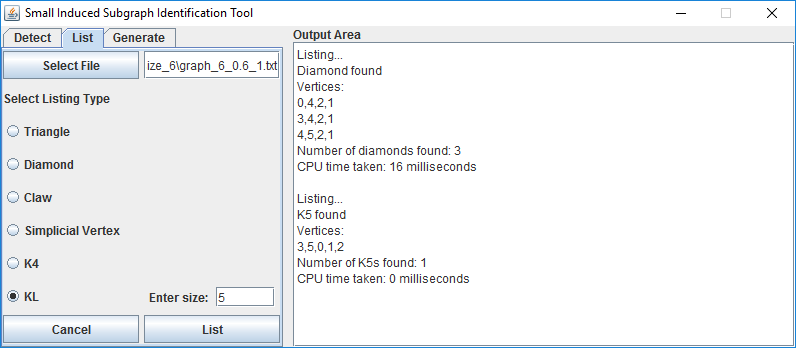


Figure E10: GUI after some listing operations have been done

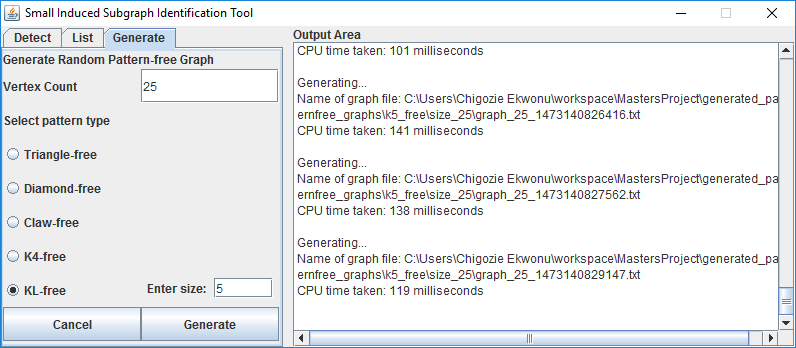


Figure E11: GUI after some generation operations have been done