UNIVERSITY OF WARWICK

THIRD YEAR PROJECT

NEW DEVELOPMENTS IN GRAPH EDITING ALGORITHMS

SAM MOON

supervised by:

Ramanujan Maadapuzhi Sridharan

May 2023

Abstract

An investigation into New Developments in Graph Editing Algorithms, studying the practicality of a recently proposed polynomial time algorithm for determining the uniqueness of the minimum vertex cover in König-Egerváry graphs. Included within this report is a summary of the algorithm's ideas and all background research required to understand and implement the algorithm. An implementation of the algorithm was created in C++ and the testing was done via the construction of a test dataset of König-Egerváry graphs using a characterisation of this class of graphs as a general graph imposed on a bipartite graph with a perfect matching.

Keywords:

Graph Editing Algorithms, König-Egerváry Graphs, Vertex Cover, Maximum Matching

Acknowledgements

I would like to thank my Supervisor Ramanujan Maadapuzhi Sridharan for his continuous support and input throughout this project, for introducing me to this interesting topic and directing me towards valuable resources necessary for this project.

My thanks also extend to my partner Robin for their moral support throughout the project.

Contents

1 Introduction

1.1 Motivation

This project aims to provide an understanding of the practicality and real-world performance of currently theoretical algorithms, which, while proven to be correct and to have polynomial runtime, have not been concretely tested. The theoretical problems which these algorithms solve often have close relations to real-world problems – as is the purpose of studying a lot of graph theory - and so determining whether they could be used in real-world solutions is important and this project will aim to answer this question as well as provide a high-quality implementation of the algorithms for benchmarking against and/or reusing.

The project builds upon content studied previously during the Discrete Maths degree course, most notably Algorithmic Graph Theory, and so is a suitable challenge for a third-year project.

1.2 Project Goals

The goals for the project were to produce an implementation of the following theoretical algorithms to showcase them and assess their performance in a practical scenario, as opposed to analysing their asymptotic performance. The specific algorithms which were planned for this were:

a) A poly-time algorithm for Testing König-Egerváry graphs for Unique Minimum Vertex Covers [1]
b) An FPT algorithm for Local Search Vertex Cover problem on Planar graphs [5]
c) A poly-time algorithm for König-Egerváry Vertex Deletion problem [6]

In addition, for each algorithm:

- The program should run smoothly and efficiently, assuming the theorised algorithm is efficient.
- The project will produce a summary of the algorithm and its process including what inputs it takes, what intermediate calculations are made and the output.

As will be discussed further later, the project only achieved these goals for algorithm (a) and so all discussed content of graph theory and the algorithms is relating to this algorithm.

2 Background

2.1 Vertex Cover Problem

The Vertex Cover problem is a well-known and common problem in Algorithmic Graph Theory with many applications in other fields of Computer Science and Mathematics such as Combinatorics but also in fields such as Computational Biology.

The problem is defined as follows: Given a graph G as input, a vertex cover of G is a subset S of vertices of G such that for all edges (u, v) in G, one or both of the following hold: u is in S or v is in S. The minimum vertex cover problem is a natural extension of this which requires that the returned vertex set is both a vertex cover and that no other vertex cover of G contains fewer vertices.

It is known that the Minimum Vertex Cover problem is NP-Complete for general graphs and so it is strongly believed that no polynomial time algorithm exists to solve it.

The algorithm we are concerned with in this project extends this Minimum Vertex Cover problem even further and asks us to determine whether the minimum vertex cover of our input graph is unique or whether there exists another distinct vertex cover also of minimal size.

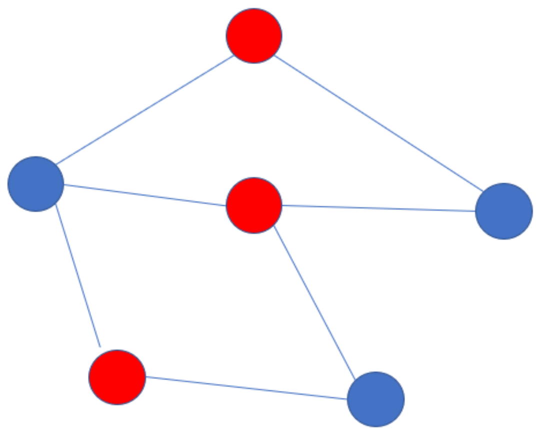


Figure 2 - A minimum vertex cover of a graph.

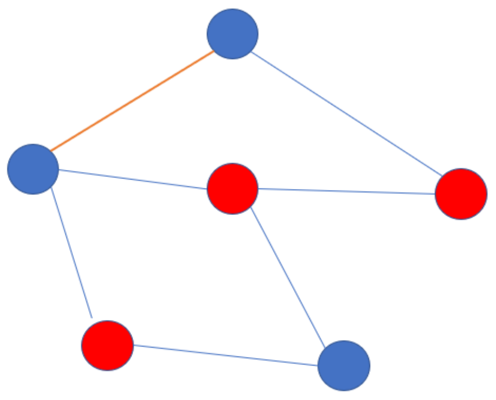


Figure 1 – A vertex set which is not a vertex cover, as there is a leftover edge.

2.2 Maximum Matching Problem

The Maximum Matching problem is another well-known problem in this field. This problem has a strong tie to the Vertex Cover problem and is defined as follows – Given a graph G as input, a matching M of G is a subset of the edges of G such that no two edges in M share an endpoint. The maximum matching problem is a natural extension of this requiring that the returned edge set is both a matching and that no other matchings of G contain more edges than M.

Unlike the Minimum Vertex Cover problem, there are known polynomial time algorithms for general graphs to solve the Maximum Matching problem. The best-known algorithms for this problem have time complexity $O(mn^{1/2})$ [7] where m is the number of edges and n the number of vertices in G.

Another algorithm for this problem is the Blossom Algorithm [2], it has time complexity $O(mn^2)$ but is simpler to implement. We will use an implementation of this algorithm during this project.

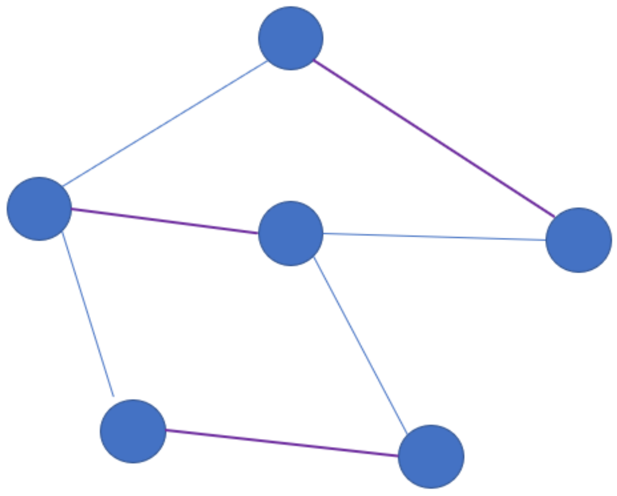


Figure 4 - A maximum matching of a graph.

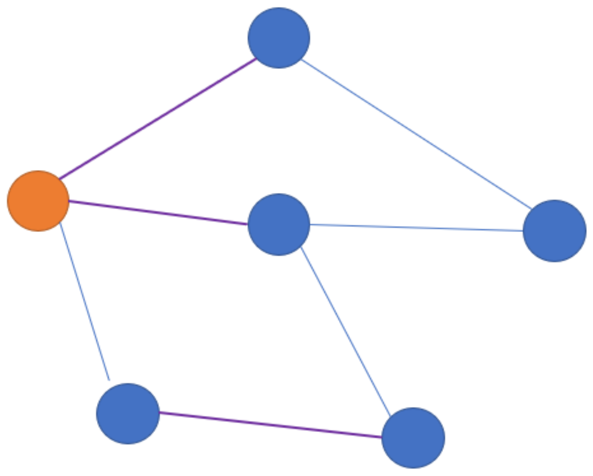


Figure 3 - An edge set which is not a matching as two edges share a vertex.

2.3 König-Egerváry Theorem

As the Vertex Cover problem is NP-Complete on general graphs, it is unlikely that we could learn much more about the problem whilst maintaining the requirement that our algorithms run in polynomial time. However, we can instead look at graphs with certain structures that can help us find the minimum vertex cover. An example of this is the König-Egerváry Theorem [8] which proves vital when reasoning with the problem on bipartite graphs – that is, graphs whose vertex set can be partitioned into two sets A and B such that all edges in the graph have exactly one endpoint in A and one endpoint in B.

The König-Egerváry theorem states that, given a bipartite graph G, the minimum vertex cover of G is the same size as the maximum matching of G. This leads to a polynomial time algorithm for the Minimum Vertex Cover problem for bipartite graphs.

The next natural step from this theorem is to find other graphs which have this duality property that the Minimum Vertex Cover size is equal to the Maximum Matching size.

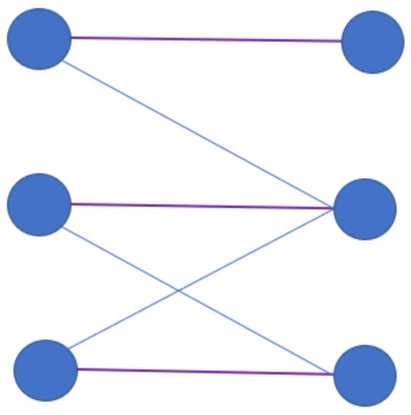


Figure 6 - A maximum matching of a bipartite graph. Matching Number = 3.

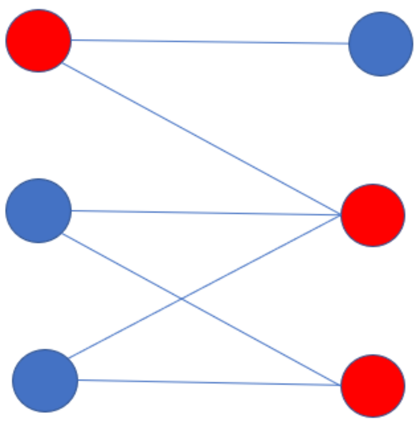


Figure 5 - A minimum vertex cover of a bipartite graph. Vertex Cover Number = 3.

2.4 König-Egerváry Graphs

This class of graphs is defined exactly as expected – they are the graphs which have the duality property from the König-Egerváry theorem. It is of course immediately clear that all bipartite graphs are König-Egerváry, but it turns out there are other graphs beyond bipartite graphs with the property as well. Another name for this class of graphs is Semipartite.

A characterisation of the graph class is given as follows – A graph G is König-Egerváry if its vertex set can be partitioned into two sets A and B such that A is a minimum vertex cover of G and there is a maximum matching M of G which is contained in the cut-set of the A/B cut and saturates A, that is, every vertex of A is the endpoint of some matching edge of M.

This characterisation implies a further fact about the graph; if A is a vertex cover of G, then B must be an Independent Set, that is, there are no edges between two vertices of B in G.

It is this structure which we can use to efficiently reason about the minimum vertex cover of any graph which we know to be König-Egerváry. So, given a graph which we know to be in this class, we can compute the maximum matching M and the minimum vertex cover A in polynomial time.

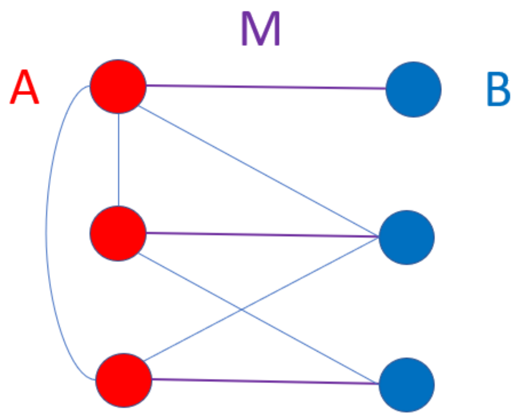


Figure 7 - A partition of a König-Egerváry graph into its minimum vertex cover A and corresponding maximum independent set B, with maximum matching M crossing the cut and saturating A.

2.5 Blossom Maximum Matching Algorithm

This algorithm [2] returns the maximum matching of the input graph in $O(mn^2)$ time. The idea is to iteratively improve an estimate for the maximum matching by finding a path which "augments" the current estimate - that is, a path beginning and ending at free vertices alternating between unmatched edges and matched edges. These augmenting paths can have all their edges flipped in or out of the matching to increase the size of the matching, this can be done by taking the symmetric difference between the matching M_i and this path P_i to obtain a larger matching M_{i+1}.

Finding these augmenting paths is simple, except for one specific case, as we can start at any free vertex and explore in a tree structure the rest of the graph, ensuring that we alternate between matched and unmatched edges. The case we are most concerned with is when we explore a matched edge to find an "even" vertex and then from this vertex find an edge to another "even" vertex as in this case we have found a Blossom which we could traverse either clockwise or anticlockwise and we must consider both to ensure we find any augmenting path which might exist.

Edmond's Blossom algorithm resolves these blossom structures by creating a new graph such that the blossom from the original is contracted into a single vertex with all edges kept in. The algorithm then continues as normal on this new graph and if we find an augmenting path which includes the contracted vertex, we then expand the blossom back out and find which direction around the blossom we needed to travel for the augmenting path.

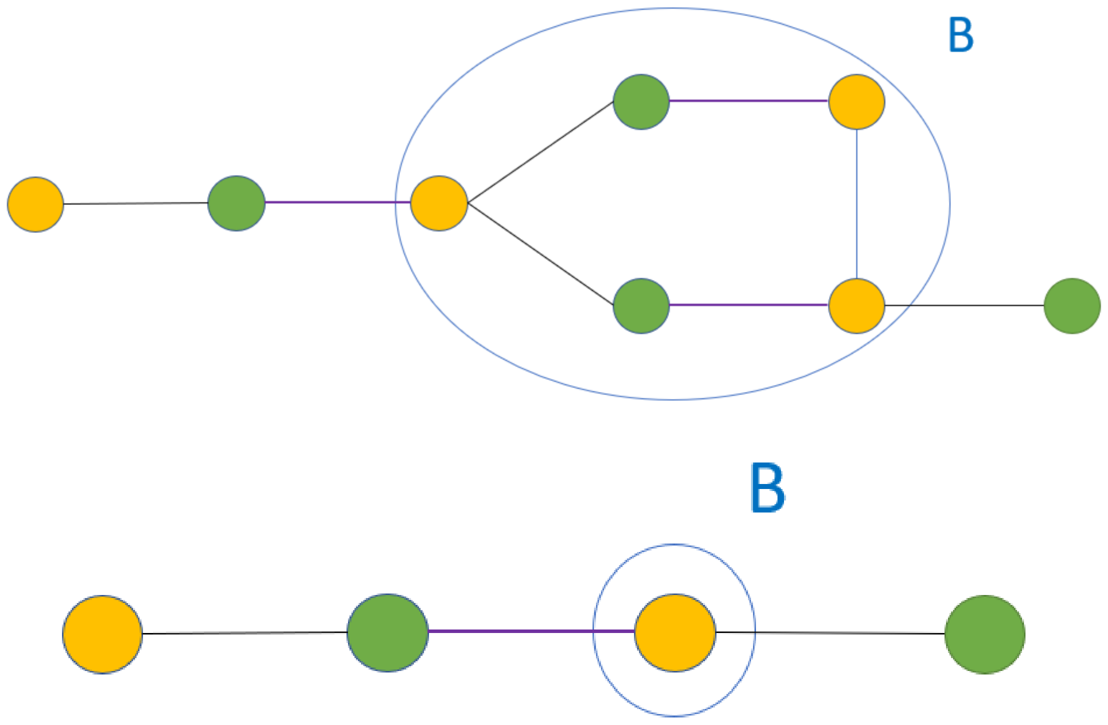


Figure 8 - An alternating tree with a blossom and the same graph with its blossom contracted into a single vertex to show the alternating path between two free vertices.

2.6 Semipartite Testing Algorithm

To find the minimum vertex cover of a König-Egerváry given its maximum matching, we can use the semipartite testing algorithm [3] - a constructive algorithm for testing whether the minimum vertex cover of any graph fits together with its maximum matching. We of course already know that the input graph is semipartite, but the algorithm will provide us with the vertex cover in addition to determining this.

The algorithm again uses an alternating tree structure to explore the graph swapping between matched and unmatched edges and labelling vertices "even" and "odd". A more concrete description is the alternating application of the following two rules:

- Rule 1: Mark "even" all unmarked vertices adjacent to an "odd" vertex. If two newly marked vertices share a matched edge, return "fail".
- Rule 2: Mark "odd" all unmarked vertices which are matched to an "even" vertex. If two newly marked vertices are adjacent, return "fail".

The algorithm also needs an initial arbitrary starting point and so we label all unmatched vertices "odd" and proceed to apply the alternating rules. If the algorithm comes to a halt without having labelled all vertices, we choose an edge and guess the labels of its vertices and continue as before – if this results in a "fail" being returned, we swap our guess around and then continue again; the final case is that the minimum vertex cover we are looking for is returned, in which case we are done. If the input graph was a general graph, then it is possible that the third option never occurs and instead, all possible branching combinations are exhausted and "fail" is returned by all, in this case, we would conclude that the graph is not semipartite, however, we are not concerned about this case here.

2.7 Computing Minimum Vertex Cover containing a specified Vertex Set

In a König-Egerváry graph G, given a minimum vertex cover A and maximum matching M in its cut and saturating A, we can determine constructively whether there exists another minimum vertex cover of G which contains a specified subset of vertices Z [1]. Of course, if Z is in A, then we are already done, so the interesting case that we will consider, is when Z is not contained in A and so intersects with B.

We construct a new graph G_{bip}, a directed bipartite graph with its two vertex sets A and B, from G by removing all edges in A, orienting all matched edges of M from A to B and orienting all unmatched edges from B to A. We explore this graph starting at the set U of unsaturated vertices, that is, any vertex not touched by an edge in M; this gives us the set R(U) of all reachable vertices from U. Here we make a check that $Z \cap R(U) - A \cap R(U) = \emptyset$, if this is not the case, we immediately conclude that no minimum vertex cover of G exists which contains Z.

Returning to graph G, we remove all vertices in R(U) from G to create a graph G/U; this graph is guaranteed to have a perfect matching M/U. Then two sets are computed in this graph; S_1 is the set of vertices to which there exists and even length M-alternating path from a vertex in $Z \cap B/U$ and S_2 is the set of vertices in A such that neither the vertex nor its matched partner are in S_1. The union of these sets $S = S_1 \cup S_2$ is the candidate for being the vertex cover containing Z, but we now check that that is the case; we must check that S is in fact a vertex cover, the size of S is equal to the size of A and that S contains Z. We then conclude as appropriate whether this is such a vertex cover or that none exist.

2.8 Determining Uniqueness of the Minimum Vertex Cover

Using our algorithm to extend the minimum vertex cover A to find other minimum vertex covers we can check if there exists a distinct minimum vertex cover by setting Z to some set not contained in A and if there exists a minimum vertex cover containing Z, we know this set cannot equal A, and so A is not unique. Our choice of sets Z to test rely on the characterisation of König-Egerváry graphs once again, we know that a minimum vertex cover S must be saturated by our maximum matching M, and so any vertex in B which is not matched by M can be ignored. Therefore, we wish to determine the existence of minimum vertex covers in G containing $Z = \{v\}$ for v some vertex in B such that v is matched in M. We know the size of M is bounded by half the number of vertices in the graph, n/2, and so we need only run our algorithm O(n) times.

3 Project Management

3.1 Methodology

Before any development could be started, it was necessary to research the algorithm thoroughly and explore all required extra subroutines. The majority of this research was done into the paper in which the algorithm was originally proposed [1] and was split into two parts; first a full read through the paper to get a general understanding of the flow of the algorithm, what subroutines are required, what exactly are the expected inputs and outputs – making notes on these. This was then followed by a more in depth study into each subroutine individually, making further notes of pseudocode and what lower-level operations would be required to implement, such as edge insertion and deletion and vertex set union and intersection. This also involved finding other papers such as for the Blossom maximum [2] matching algorithm and the Semipartite testing algorithm [3], where a similar strategy was then employed.

During development, each major subroutine was implemented and tested for correctness individually to reduce the number of problems encountered later on when implementing a later routine which depends heavily on the result of a previous routine being correct such as Semipartite testing requiring the maximum matching of the input graph.

3.2 Programming Language

Initially, to decide what programming language to produce the implementation in there were a few factors necessary to consider.

1. How fast does the language generally run?
2. How much experience do I have with using the language or similar languages?
3. Are there relevant and useful libraries which could be helpful?

Additionally, both due to my previous experience and preference, and the nature of graph editing algorithms, it was decided that a very object oriented approached would be used for the implementation and so the language used would need to be a good OOP language.

The three major considerations were therefore: Python, C++ and Java.

Firstly, we discuss factor (1); of the three, C++ is highly likely to be the fastest due to it being a lower-level style language with less bloat, more direct control over memory management and the program being fully compiled for a target machine. This is in direct opposition to both Python and Java which are designed for easy use with a wide range of machines and operating systems – which is not of use for this project as we are not looking to compare the same program on multiple machines, just testing the algorithm implementation. So, C++ would be the best option with respect to this factor.

Factor (2); my personal experience in these three languages is quite similar across the range. I had directly used both Python and Java for a few years over a range of projects. I had used C++ directly the least, just learning it by reading through other projects and examples rather than my own projects, however I also had a lot more experience in the C family of languages than either Java or Python. So, for this factor, Python and Java would be the best options however, C++ is still very viable.

Factor (3); all three languages have a range of graphing libraries such as JGraphT for Java, igraph and NetworkXX for Python and BGL and SNAP for C++. I had no experience with any of these libraries when deciding which to use but from general research into these libraries it seemed that all are very large and cover a very wide range of methods and graph representation methods. This makes it very difficult to use just a select few parts we might actually want whilst understanding how they work and/or if there are any unnecessary resources being used. Therefore, this factor might in fact not be required as it may be easier and more representative of the algorithm's efficiency if bespoke structures and methods are used. So overall, any of the three languages is viable with respect to this aspect.

In the end, it was decided that C++ would be used, as it is certainly the best option when considering speed and the lack of direct experience is a smaller consideration which can be overcome simply with a little extra work.

3.3 Time Management

As it was clear from the start that achieving all of the goals laid out was going to be difficult, the decision was made to complete each section before moving on to starting any other – that is, fully research implement and test algorithm (a) before moving on to the next. This was to ensure that if timings were misjudged, there would still be a full process and set of results to report on.

The following Gantt Chart shows the original plan for how the project would be split regarding time.

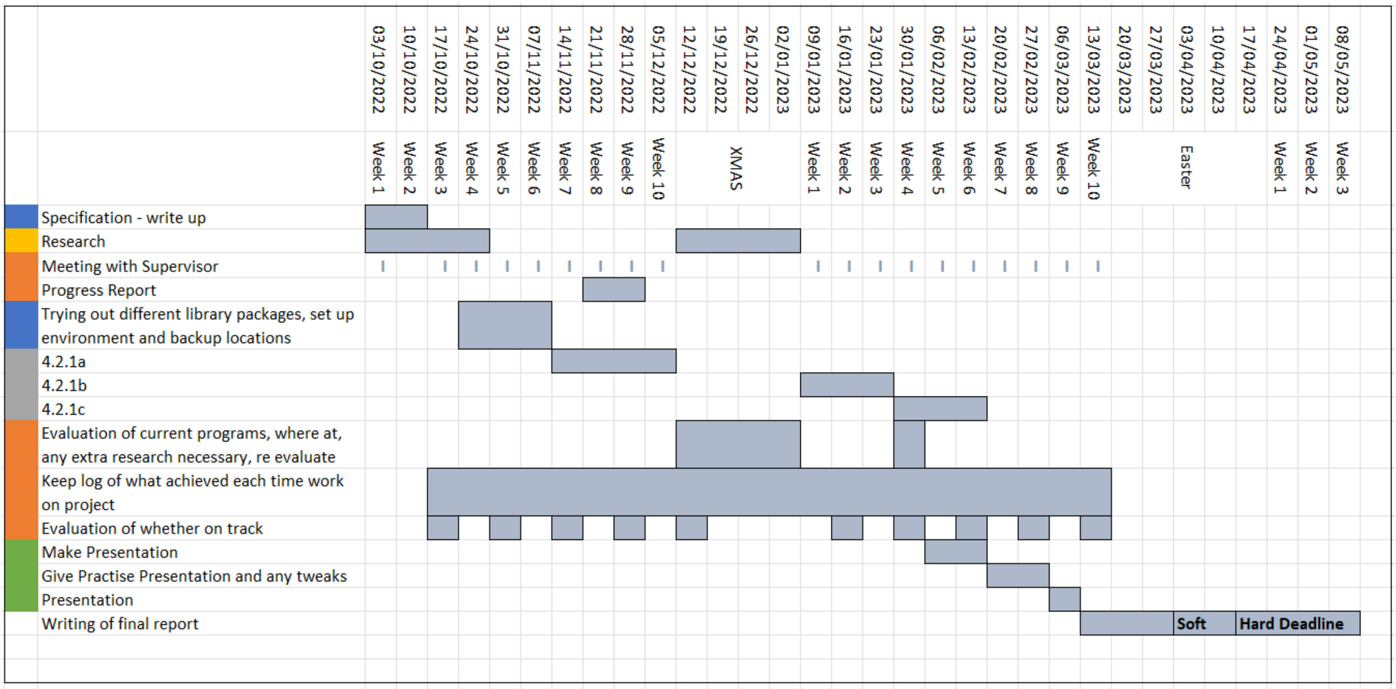


Figure 9 - Project's original time management Gantt chart.

However, this was an optimistic estimate both on how quickly a high-quality implementation of algorithm (a) could be produced as well as how quickly I could understand the content required to implement algorithm (b) and so unfortunately only a full analysis of algorithm (a) was achieved. The timings of the project instead were the following:

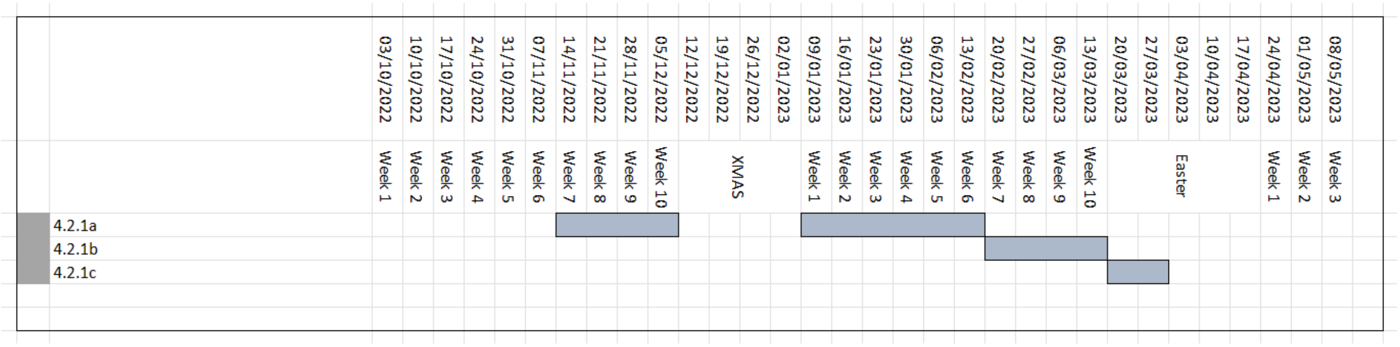


Figure 10 - Update to the Gantt chart displaying how time was split across the three objectives.

The time spent on algorithms (b) and (c) involved research into them and how I would go about implementing them but, as mentioned, for (b) it was deemed too challenging for a second part of a project and would instead require a full project dedicated to it and for (c), while it was a similar challenge as (a), unfortunately too much time had been spent on (b) and so the decision was made to dedicate the project instead just to algorithm (a).

As for week-by-week time management, the following two timetables were used throughout the two terms.

Term 1	10am	11am	12am	1am	2am	3am	4am	5am
Monday		S	x	x		x	x	
Tuesday	x							
Wednesday						x	x	x
Thursday				x	x	x		
Friday								
Saturday								
Sunday								

Term 2	10am	11am	12pm	1pm	2pm	3pm	4pm	5pm
Monday								
Tuesday		S	x			x		
Wednesday		x	x			x		
Thursday						x		
Friday								
Saturday			x			x		
Sunday			x			x		

S used to denote Supervisor Meetings

These schedules enabled me to regularly work on my project, with some weeks more dedicated to research, and some to the implementation and analysis of the project. My timetable was flexible and able to be changed when necessary, but meant I set aside regular timings to work on it.

I also met with my supervisor regularly – this enabled me to set weekly targets for myself, so as to know what to aim for prior to my meeting. This was then able to be discussed with him, and any challenges encountered could be worked through.

4 Design & Development

4.1 Input Parameters

The test dataset needed to be entirely composed of König-Egerváry graphs as this is all the algorithm is designed to work on – it would not, however, be a huge leap to add checks into the algorithm allowing it to identify which inputs are König-Egerváry and which are not, as this is the real purpose of the Semipartite testing algorithm. As no such dataset is available online it was necessary to create our own using the characterisation of this class of graphs mentioned earlier:

A graph G is König-Egerváry if its vertex set can be partitioned into two sets A and B such that A is a minimum vertex cover of G and there is a maximum matching M of G which is contained in the cut-set of the A/B cut and saturates A, that is, every vertex of A is the endpoint of some matching edge of M.

We make use of this by dividing all König-Egerváry graphs into three parts:

1. A general graph on n_1 vertices.
2. A bipartite graph with bipartition (L, R) where L has n_1 vertices and R has n_2 vertices.
3. An L-perfect matching across the bipartition.

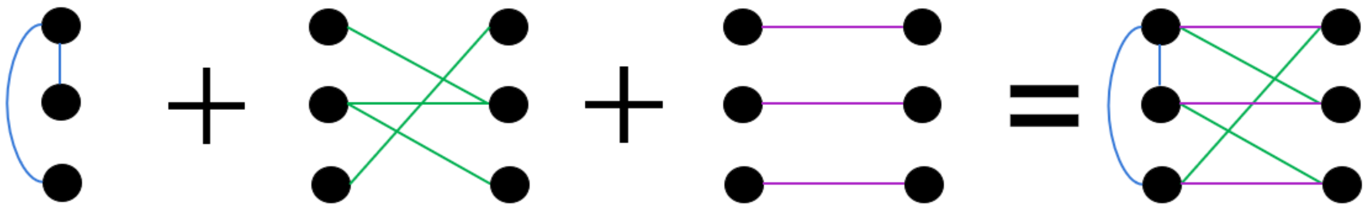


Figure 11 - Diagram displaying how the three components of a König-Egeváry graph can be combined.

These three parts are objects that we can find a dataset [4] for and so we had two sets:

- 18 bipartite graphs of order n for each n = 100, 200, …, 700
  - With exactly half on each side of the bipartition.
- 18 general graphs of order n for each n = 50, 100, …, 350

These were combined as expected by imposing all edges of a general graph onto the L set of a bipartite graph. Then to ensure the maximum matching is a perfect matching, we just add all parallel edges across the bipartition if they weren't already present - that is all edges (l_i, r_i) for $i \leq n/2$.

While the restriction of this method to just bipartite graphs with exactly a half-half split does not include all possible König-Egerváry graphs, the assumption that the matching is completely perfect (not just L-perfect) only has an effect on one part of the algorithm; that is, when we are choosing sets Z to determine whether there exists a minimum vertex cover distinct from A, we only choose vertices from B which are matched in M. So in this case, this means we will always check all vertices of B – and so, any results on graphs found this way are also quite indicative of how the algorithm would perform on any graph with the same size minimum vertex cover A (equivalently, any graph constructed from a bipartite graph with equal size L set, potentially with a larger R set) as any extra vertices added to the set B have no impact on the performance here as these vertices are ignored.

Overall, this gave us a dataset of 2268 König-Egerváry graphs to test the algorithm on. With sizes ranging from 100 vertices and 200 edges to 700 vertices and 165,000 edges.

4.2 Storing Results

The results for this algorithm require storing the size of each graph, the time taken to run the algorithm on the graph and the outcome – whether the graph has a unique minimum vertex cover or not. This form suits a simple .csv file well and so all results were saved to a line appended to a .csv file of the whole dataset. In the next section the code used to do this will be discussed but the format is as follows:

```
Graph Name, n, m, Result, Time Taken
```

This can then be imported into Excel or Google Sheets where datapoints can be easily grouped together by size and displayed visually on a graph.

5 Implementation

5.1 Libraries

As we will see next, most of the classes and methods used for storing and manipulation of the graphs were implemented bespoke for this project to remove any dependence on “hidden” library methods which might introduce bloat or increase processing time unnecessarily. That said, a few small standard C++ libraries were used for string and file manipulation and measuring processing time to obtain our results.

```
#include <iostream>
#include <list>
#include <string>
#include <fstream>
#include <chrono>
```

5.2 Data Structures

As is quite common when implementing algorithms for graphs which are not necessarily sparse, an adjacency list data structure was used here to represent all graphs. Following are details of all classes implemented and used by the algorithm.

To represent vertices, we just use integer values which works with the adjacency list implementation of graphs as we can access the neighbourhood of a vertex v by just taking the list adjList[v].

The purpose of the Graph class is to store all relevant information about a single graph and allow for some basic manipulation such as insertion and deletion of edges. In addition, the class stores labels for the vertices which are used by specific subroutines when exploring the graph and/or labelling the vertices even or odd in an alternating tree.

```
class Graph
public:
	//Adjacency list storing neighbourhoods of each vertex
	list<int>* adjList;

	//Useful details to store for easy access
	int n;
	int m;
	bool directed;

	//Labels for vertices when exploring the graph
	bool* explored;

	//Alternating tree labels
	list<int> even;
	list<int> odd;
```

One problem encountered during development was very high memory usage when given larger input graphs or multiple graphs in succession. It turned out this was due to the internal data structures of the graphs not being freed up correctly, this destructor corrects that.

```
//Destructor to ensure all memory is cleared up
~Graph() {
	delete[] adjList;
	delete[] explored;
}
```

This constructor simply initialises the class attributes. A limitation of this implementation of graphs is the fixed-vertex set due to the use of an array of lists rather than a dynamic structure containing the lists – whilst in many other settings this would cause problems, here at no point do we need to remove or add vertices from or to a graph and so this suffices.

```
//Constructor to initialise a fixed-vertex-set empty-edge-set graph
Graph(int N, bool Directed = false) {
	adjList = new list<int>[N];
	n = N;
	m = 0;
	directed = Directed;
	explored = new bool[N];
	for (int i = 0; i < N; i++)
		explored[i] = false;
}
```

A copy constructor is also used when we just want to run some checks and modifications on a graph but still need the graph intact later.

```
//Copy constructor for creating copies of graphs to manipulate without changing the original
Graph(const Graph& G) {
	n = G.n;
	m = G.m;
	directed = G.directed;

	if (!G.even.empty())
		even = list<int>(G.even);
	if (!G.odd.empty())
		odd = list<int>(G.odd);

	adjList = new list<int>[n];
	for (int i = 0; i < n; i++)
		for (int j : G.adjList[i])
			adjList[i].push_back(j);

	explored = new bool[n];
	for (int i = 0; i < n; i++)
		explored[i]= G.explored[i];
}
```

The remaining methods are quite self-explanatory – they are the basic graph manipulations used throughout the algorithm and a print method used during debugging.

```
//Add an edge between two vertices
void addEdge(int u, int v) {
	if (find(adjList[u].begin(), adjList[u].end(), v) != adjList[u].end())
		return;

	adjList[u].push_back(v);
	if (!directed) {
		adjList[v].push_back(u);
	}
	m++;
}
```

```
//Display the adjList of this graph for debugging purposes
void print(string label = "Graph:") {
	cout << label << "\n";
	for (int i = 0; i < n; i++) {
		cout << i << " --> ";
		for (int it : adjList[i]) {
			cout << it << " ";
		}
		cout << endl;
	}
	cout << endl;
}
```

```
//Remove a specified edge from the graph
void removeEdge(int v, int u) {
	list<int> t1 = adjList[v];
	list<int> t2 = adjList[u];
	t1.remove(u);
	t2.remove(v);
	adjList[v] = t1;
	adjList[u] = t2;
	m--;
}

void removeEdge(Edge e) {
	removeEdge(e.get(0), e.get(1));
}
```

```
//Remove all incident edges of a specified vertex
// - used to check if a set of vertices is a vertex cover
void removeEdgesFromVertex(int v) {
	if (directed)
		cout << "Warning this function does not handle directed graphs correctly.";

	m -= adjList[v].size();
	adjList[v].clear();

	for (int i = 0; i < n; i++) {
		adjList[i].remove(v);
	}
}
```

The Edge class is a very simple tool for more easily representing an individual edge of a graph with some methods for copying and comparing edges. The endpoints are stored as a 2-length array of ints and a bool flag to indicate if the edge is directed or bi-directional.

```
class Edge {
public:
	int v[2];
	bool directed;
```

The class has a basic constructor which sorts the endpoints in ascending order for undirected edges for easier comparison and legibility when debugging.

```
//Constructor to instantiate an edge given its endpoints and directed flag
Edge(int u1, int u2, bool Directed = false) {
	directed = Directed;
	if (directed || u1 < u2) {
		v[0] = u1;
		v[1] = u2;
	}
	else {
		v[0] = u2;
		v[1] = u1;
	}
}
```

Here an override for the = operator to assign the endpoints of the parameter to this edge whilst keeping the two instances independent.

```
//Copy the values for the endpoints of the edge
void operator=(const Edge& e) {
	v[0] = e.v[0];
	v[1] = e.v[1];
}
```

In addition, an override for the == operator used to compare two edges and determine equality. This occurs for undirected edges when endpoints match up in either combination or for directed edges when they match up in order as well as value.

```
//Compare the endpoints of the edges in order and if the edge isn't directed, also when the order is
reversed
bool operator==(const Edge& f) const {
	return (f.get(0) == v[0] && f.get(1) == v[1]) ||
			(f.get(1) == v[0] && f.get(0) == v[1]) && !directed;
}
```

This is a simple get function to access the value of either of the Edge's two endpoints.

```
//Get the requested endpoint
int get(int i) const {
	return v[i];
}
```

The meet method determines whether either of the two endpoints of this Edge equals either of the two endpoints of the parameter Edge – that is, they meet at some vertex and so are adjacent.

```
//Check any of the 4 pairs of endpoints for equality to check if the two edges meet
bool meet(const Edge& e) const {
       if (e.get(0) == v[0] || e.get(0) == v[1] || e.get(1) == v[0] || e.get(1) == v[1])
              return true;
       return false;
}
```

The EdgeSet class maintains a list of Edge objects with functionality to compare EdgeSet objects, determines whether the set covers specified vertices and add and remove edges.

```
class EdgeSet {
public:
       list<Edge> E;
```

Assignment of EdgeSet objects here is defined as simply assigning the list of Edge objects to that of a copy of the passed EdgeSet – effectively, this copies the EdgeSet rather than assigning by reference.

```
void operator=(EdgeSet S) {
       E = S.E;
}
```

To determine whether a vertex is covered by the EdgeSet this method iterates through all its Edge objects and checks if any are incident to the vertex.

```
bool coversVertex(int i) const {
       for (const Edge& it : E) {
              if (it.get(0) == i || it.get(1) == i)
                     return true;
       }
       return false;
}
```

The following methods are the basic manipulations to add and remove Edge objects to and from the set. In addition, a print method for debugging.

```
void addEdges(EdgeSet s) {
	for (const Edge& it : s.E)
		E.push_back(it);
}
```

```
void addEdge(int u, int v, bool directed = false) {
	Edge e(u, v);
	E.push_back(e);
}
```

```
void addEdge(Edge e) {
	E.push_back(e);
}
```

```
void removeEdge(int u, int v) {
	for (const Edge& it : E) {
		if (it.get(0) == u && it.get(1) == v) {
			E.remove(it);
			return;
		}
	}
	cout << "EdgeSet.removeEdge(u, v) > ERROR: edge (" << u << ", " <<  v << ") to be removed not
found";
}
```

```
void removeEdge(Edge e) {
	E.remove(e);
}
```

```
void print(string label = "EdgeSet:", bool alternating = false) {
	cout << label << "   ";
	bool matched = false;
	for (const Edge& it : E) {
		if (!alternating)
			cout << "(" << it.get(0) << ", " << it.get(1) << ") ";
		else if (matched)
			cout << "+(" << it.get(0) << ", " << it.get(1) << ") ";
		else
			cout << "-(" << it.get(0) << ", " << it.get(1) << ") ";
		matched = !matched;
	}
	cout << "\n";
}
```

The augmentBy method is used by Edmond’s Blossom algorithm to find the maximum matching of our input graph. It strictly increases the size of the matching EdgeSet by taking the symmetric difference of it with the augmenting path P – that is, we want all edges which are in exactly one of the two and we discard all other edges.

```
EdgeSet augmentBy(EdgeSet P) {
	EdgeSet L;
	L.addEdges(P);
	L.addEdges(*this);

	for (const Edge& it : E) {
		for (const Edge& itP : P.E) {
			if (it == itP) {
				L.removeEdge(it);
			}
		}
	}

	return L;
}
```

```
//Return all vertices of G without an incident matched edge
list<int> UnsaturatedVertices(const Graph& G, const EdgeSet& M) {
	list<int> U;
	for (int i = 0; i < G.n; i++) {
		if (!M.coversVertex(i))
			U.push_back(i);
	}
	return U;
}
```

The sort method here was used for legibility when debugging.

```
void sort(bool alt = false) {
	if (alt) {
		list<Edge> t;
		for (int i = 0; i < E.size(); i++) {
			Edge e(-1, -1);
			for (const Edge& it : E) {
				if (find(t.begin(), t.end(), it) == t.end()) {
					if (it.get(1) > e.get(1))
						e = it;
				}
			}
			if (e.get(1) != -1)
				t.push_front(e);
		}
		E = t;
	} else {
		list<Edge> t;
		for (int i = 0; i < E.size(); i++) {
			Edge e(-1, -1);
			for (const Edge& it : E) {
				if (find(t.begin(), t.end(), it) == t.end()) {
					if (it.get(0) > e.get(0))
						e = it;
				}
			}
			if (e.get(0) != -1)
				t.push_front(e);
		}
		E = t;
	} }
```

As with these three structures, we need some basic manipulation methods for our vertex sets. These are implemented as global methods - they include a print method for debugging purposes and the set operations intersection, union and set difference.

```
void PrintVertices(const string& str, const list<int>& lst) {
       cout << str << "     {";
       for (int i : lst) {
              cout << i << ", ";
       }
       cout << "}\n";
}
```

```
list<int> IntersectVertices(const list<int>& A, const list<int>& B) {
       list<int> C;
       for (int i : A)
              if (find(B.begin(), B.end(), i) != B.end())
                     C.push_back(i);
       return C;
}
```

```
list<int> UnionVertices(const list<int>& A, list<int>& B) {
       list<int> C(A);
       C.merge(B);
       C.sort();
       C.unique();
       return C;
}
```

```
list<int> SetMinusVertices(const list<int>& A, const list<int>& B) {
       list<int> C;
       for (int i : A)
              if (find(B.begin(), B.end(), i) == B.end())
                     C.push_back(i);
       return C;
}
```

5.3 Maximum Matching

To find the maximum matching of the input graph we use Edmond's Blossom algorithm [2] which finds augmenting paths for some matching and then augments the matching by this path to produce a larger matching. This continues until no augmenting paths remain – then we have a maximum matching.

```
//Blossom contraction algorithm to find max matching
EdgeSet FindMaxMatching(Graph& G, EdgeSet M) {
	EdgeSet P = FindAugmentingPath(G, M);

	if (P.E.size() != 0) {
		EdgeSet aug = M.augmentBy(P);
		return FindMaxMatching(G, aug);
	} else
		return M;
}
```

The FindAugmentingPath method is the starting point for the construction of the alternating tree. It tries to find a path starting from each free vertex in the graph, or until any path is found. In this implementation, the tree is constructed depth-first with an incrementing depth bound. This is for a couple of reasons:

1. Simplicity of implementation.
2. Favours finding short augmenting paths
3. This procedure has a very minimal impact on the run time of the algorithm as a whole.

In a real-world scenario, it is likely a better option to either use a breadth-first approach as it has the advantage over depth-first that any short augmenting paths are found quickly while not making redundant calculations when extending the depth-bound and having to retrace the tree. However, it was found here that when testing the implementation, computing the maximum matching took less than 5% of the total runtime and so the decision was made to use a simple implementation with minimal effect on the results.

```
//Find an augmenting path for the current best matching candidate
EdgeSet FindAugmentingPath(Graph& G, const EdgeSet& M) {
	EdgeSet P;

	for (int i = 0; i < G.n; i++) {
		if (!M.coversVertex(i)) {
			int maxDepth = 1;
			while (P.E.empty()) {
				G.odd.clear();
				G.even.clear();
				P = exploreEven(G, M, i, P, maxDepth+=2);
			}
			break;
		}
	}

	return P;
}
```

This method deals with exploring an even vertex in the alternating tree. We explore its neighbours and have three cases to consider; the neighbour is unexplored – we mark it odd and explore it, the neighbour is odd – we ignore it, the neighbour is even – we have found a blossom and need to contract it and continue, then expand it back out if a path is found later.

```
//Explore an even vertex of the alternating tree
// - Contract any blossoms found and then revert if a path is found
EdgeSet exploreEven(Graph& G, const EdgeSet& M, const int v, EdgeSet path, int maxDepth) {
	if (path.E.size() > maxDepth)
		return EdgeSet();
	G.even.push_back(v);

	for (int u : G.adjList[v]) {
		if (find(G.even.begin(), G.even.end(), u) != G.even.end()) {
			//Merge the blossom into a single vertex and continue the search, then unfold 
the blossom vertex if a path is found
			EdgeSet pathToBase;
			EdgeSet M_H;
			Graph H(G);
			EdgeSet pathR = path;
			pathR.E.reverse();
			list<int> blossom;

			for (const Edge& it : pathR.E) {
				blossom.push_back(it.get(0));
				blossom.push_back(it.get(1));
				if (it.get(0) == u || it.get(1) == u)
					break;
			}

			blossom.sort();
			blossom.unique();

			for (int w : blossom) {
				for (int i : G.adjList[w])
					if (find(blossom.begin(), blossom.end(), i) == blossom.end()) {
						H.addEdge(i, u);
					}
				if (w != u) {
					H.adjList[w].clear();
					for (int j = 0; j < G.n; j++)
						H.adjList[j].remove(w);
				}
			}

			for (const Edge& it : path.E) {
				//Get the next edge in the path to check this is an in-edge not an out
				list<Edge> temp(path.E);
				temp.pop_front();
				Edge next = temp.front();

				if ((it.get(0) == u || it.get(1) == u) && (it == path.E.front() && 
next.get(0) != u && next.get(1) != u))
					break;
				pathToBase.addEdge(it);
				if (it.get(0) == u || it.get(1) == u)
					break;
			}

			for (const Edge& it : M.E) {
				if (it.get(0) == u || it.get(1) == u)
					M_H.addEdge(it);
				if (find(blossom.begin(), blossom.end(), it.get(0)) == blossom.end() &&
                                    find(blossom.begin(), blossom.end(), it.get(1)) == blossom.end())
                                    M_H.addEdge(it);
                        }

                        EdgeSet newPath = exploreEven(H, M_H, u, pathToBase, maxDepth);

                        if (!newPath.E.empty()) {
                              EdgeSet unfoldedPath;
                              bool reachedBase = false;
                              int target = -1;
                              for (const Edge& it : newPath.E) {
                                    if (target == -1) {
                                          if (it.get(0) == u) {
                                                //Get the next edge in the path to check this is an
in-edge not an out-edge
                                                list<Edge> temp(newPath.E);
                                                Edge next(-1, -1);
                                                if (!temp.empty())
                                                      temp.pop_front();
                                                if (!temp.empty())
                                                      next = temp.front();

                                                if (find(G.adjList[u].begin(), G.adjList[u].end(),
it.get(1)) == G.adjList[u].end()) {
                                                      //Find path through blossom to target
                                                      target = it.get(1);
                                                      EdgeSet p(path);
                                                      p.addEdge(u, v);

      unfoldedPath.addEdges(findPathThroughBlossom(G, M, blossom, u, target, p));
                                                      continue;
                                                }
                                          } else if (it.get(1) == u) {
                                                //Get the next edge in the path to check this is an
in-edge not an out-edge
                                                list<Edge> temp(newPath.E);
                                                Edge next(-1, -1);
                                                if (!temp.empty())
                                                      temp.pop_front();
                                                if (!temp.empty())
                                                      next = temp.front();

                                                if (find(G.adjList[u].begin(), G.adjList[u].end(),
it.get(0)) == G.adjList[u].end()) {
                                                      //Find path through blossom to target
                                                      target = it.get(0);
                                                      EdgeSet p(path);
                                                      p.addEdge(u, v);

      unfoldedPath.addEdges(findPathThroughBlossom(G, M, blossom, u, target, p));
                                                      continue;
                                                }
                                          }
                                    }
                                    unfoldedPath.addEdge(it);
                              }
                              return unfoldedPath;
                        }
                  } else if (find(G.odd.begin(), G.odd.end(), u) != G.odd.end()) {
                        //Ignore this edge
                  } else {
                        //Continue as normal
                        EdgeSet newPath = path;
                        newPath.addEdge(v, u, G.directed);
                        newPath = exploreOdd(G, M, u, newPath, maxDepth);
                        if (!newPath.E.empty())
                        return newPath;
                }
        }
        G.even.remove(v);
        return EdgeSet();
}
```

This method deals with exploring an odd vertex, this is a lot simpler than for even vertices. We are looking for a matching edge which is incident to this vertex, if one is found we traverse it and label the next vertex even. If we can't find a matching edge then this is a free vertex labelled odd and so is the endpoint of an augmenting path and we can return this path.

```
//Explore an odd vertex in the alternating tree
// - Find the matched
EdgeSet exploreOdd(Graph& G, const EdgeSet& M, const int v, EdgeSet path, int maxDepth) {
        if (path.E.size() > maxDepth)
                return EdgeSet();

        G.odd.push_back(v);

        bool found = false;

        for (const Edge& it : M.E) {
                if (it.get(0) == v) {
                        found = true;
                        EdgeSet newPath = path;
                        newPath.addEdge(v, it.get(1), G.directed);
                        newPath = exploreEven(G, M, it.get(1), newPath, maxDepth);
                        if (!newPath.E.empty())
                                return newPath;
                        break;
                } else if (!it.directed && it.get(1) == v) {
                        found = true;
                        EdgeSet newPath = path;
                        newPath.addEdge(v, it.get(0), G.directed);
                        newPath = exploreEven(G, M, it.get(0), newPath, maxDepth);
                        if (!newPath.E.empty())
                                return newPath;
                        break;
                }
        }

        if (found) {
                G.odd.remove(v);
                return EdgeSet();    //following matched edge returned no result so return empty
        } else
                return path; //odd vertex is free so path found
}
```

5.4 Minimum Vertex Cover

Following is the implementation of the semipartite testing algorithm [3] which determines whether a minimum vertex cover exists such that M saturates it and cuts across the cut it defines. We already know that this exists but we still want to know what the set is and the algorithm provides this as well.

This method applies Rule 1 of the algorithm – to mark "even" all unmarked vertices adjacent to an "odd" vertex and the check if any of these newly marked vertices are matched together, if so return "fail".

```
//Apply rule 1 – mark even vertices adjacent to odd vertices
bool Rule1(Graph& G, const EdgeSet& M, list<int>& unmarked, int newEven = -1, int newOdd = -1) {
	list<int> newMarked;
	if (newEven != -1)
		newMarked.push_back(newEven);

	if (unmarked.empty())
		return true;

	for (const int v : G.odd)
		for (const int u : G.adjList[v])
			if (find(unmarked.begin(), unmarked.end(), u) != unmarked.end())
				newMarked.push_back(u);

	if (newMarked.empty())
		return true;

	newMarked.sort();
	newMarked.unique();

	for (const Edge& e : M.E) {
		if (find(newMarked.begin(), newMarked.end(), e.get(0)) != newMarked.end() &&
find(newMarked.begin(), newMarked.end(), e.get(1)) != newMarked.end())
			return false;
	}

	for (int v : newMarked) {
		G.even.push_back(v);
		unmarked.remove(v);
	}

	return Rule2(G, M, unmarked, newOdd);
}
```

This method applies Rule 2 of the algorithm – to mark "odd" all unmarked vertices matched to an "even" vertex and the check if any of these newly marked vertices are adjacent together, if so return "fail".

```
//Apply rule 2 - mark odd vertices matched to even vertices
bool Rule2(Graph& G, const EdgeSet& M, list<int>& unmarked, int newOdd = -1) {
	list<int> newMarked;
	if (newOdd != -1)
		newMarked.push_back(newOdd);

	if (unmarked.empty())
		return true;

	for (const Edge& e : M.E) {
		if (find(unmarked.begin(), unmarked.end(), e.get(1)) != unmarked.end() &&
find(G.even.begin(), G.even.end(), e.get(0)) != G.even.end())
			newMarked.push_back(e.get(1));
		else if (find(unmarked.begin(), unmarked.end(), e.get(0)) != unmarked.end() &&
find(G.even.begin(), G.even.end(), e.get(1)) != G.even.end())
			newMarked.push_back(e.get(0));
	}

	if (newMarked.empty())
		return true;

	newMarked.sort();
	newMarked.unique();

	for (int v : newMarked) {
		for (int u : G.adjList[v]) {
			if (find(newMarked.begin(), newMarked.end(), u) != newMarked.end())
				return false;
		}
	}

	for (int v : newMarked) {
		G.odd.push_back(v);
		unmarked.remove(v);
	}

	return Rule1(G, M, unmarked);
}
```

The MinVertexCover method starts the process of labelling the vertices and handles the case where no new vertices are labelled but there are still vertices to label. In this case, and to begin, an edge is chosen arbitrarily, and its two vertices are labelled distinctly and arbitrarily. The rules are then applied alternately as normal. If the choice made results in a failed attempt, we return and swap the labels before continuing.

```
//Compute the Minimum Vertex Cover of G given M and that G is Semipartite by alternating labels
//even and odd, the evens are our vertex cover.
list<int> MinVertexCover(Graph& G, EdgeSet& M, list<int>& unmarked, int newEven = -1, int newOdd = -
1) {
	if (!Rule1(G, M, unmarked, newEven, newOdd))
		return list<int>();

	if (G.odd.size() + G.even.size() == G.n)
		return G.even;
	else {
		for (int v = 0; v < G.n; v++) {
			if (find(unmarked.begin(), unmarked.end(), v) != unmarked.end()) {
				for (int u : G.adjList[v]) {
					if (find(unmarked.begin(), unmarked.end(), u) != unmarked.end())
{
						Graph H(G);
						Graph I(G);
						list<int> backupUnmarked(unmarked);
						list<int> S = MinVertexCover(I, M, unmarked, u, v);
						if (S.empty())
							return MinVertexCover(H, M, backupUnmarked, v, u);
						else
							return S;
					}
				}
			}
		}
	}

	cout << "Error: Min Vertex Cover method failed";
	return list<int>();
}
```

```
//Start the MinVertexCover method with free vertices marked odd and all other vertices unmarked
list<int> MinVertexCoverStart(const Graph& G, EdgeSet M) {
	//Assume M is maximum matching of G and G Konig-Egervary

	Graph H(G);
	H.even.clear();
	list<int> freeVertices;
	list<int> unmarked;
	for (int i = 0; i < H.n; i++)
		if (!M.coversVertex(i))
			freeVertices.push_back(i);
		else
			unmarked.push_back(i);
	H.odd = freeVertices;

	return MinVertexCover(H, M, unmarked);
}
```

5.5 Minimum Vertex Cover containing set Z

This method [1] constructs the directed bipartite G_{bip} graph which takes the input graph G with min vertex cover A and maximum matching M; it removes all edges between two vertices in A, orients all matched edges from A to B and all remaining edges from B to A.

```
//Construct the bipartite A -> B directed graph G_bip(M)
Graph ConstructG_bip(const Graph& G, const EdgeSet& M, list<int>& A) {
	Graph G_bip(G.n, true);
	for (const Edge& e : M.E) {
		if (find(A.begin(), A.end(), e.get(0)) != A.end())
			G_bip.addEdge(e.get(0), e.get(1));
		else
			G_bip.addEdge(e.get(1), e.get(0));
	}
	for (int j : A) {
		for (const Edge& it : M.E) {
			if (it.get(0) == j) {
				G_bip.addEdge(it.get(1), j);
			} else if (it.get(1) == j) {
				G_bip.addEdge(it.get(0), j);
			}
		}
	}

	return G_bip;
}
```

The ReachableFromVertices method finds the set of vertices which are reachable from a given set of vertices U in a given graph. This is found by simply following all available edges to expand the frontier of unexplored vertices and then moving to the next frontier vertex to explore similarly. This is used to find the set R(U) of reachable vertices from the set U of unsaturated vertices in G_{bip}.

```
//Explore the graph using a frontier of unexplored vertices until the frontier is empty
list<int> ReachableFromVertices(Graph& G, list<int>& U) {
	list<int> reachable;
	list<int> frontier(U);

	while (!frontier.empty()) {
		int v = frontier.front();
		frontier.pop_front();
		reachable.push_back(v);
		for (int u : G.adjList[v])
			if (find(reachable.begin(), reachable.end(), u) == reachable.end())
				frontier.push_back(u);
	}

	return reachable;
}
```

This method simply removes the specified set of vertices from the graph. In reality, the vertices are still present in the graph data structure however they have no edges incident to them and so for the purposes required they will never be found.

```
//Strip away all edges attached to the passed vertices
Graph RemoveVerticesFromGraph(Graph G, list<int>& V) {
	for (int i : V)
		G.removeEdgesFromVertex(i);
	return G;
}
```

The following methods calculate S_1, the set of vertices to which there is an even length alternating path from a vertex in B/U in the graph G/U - the subgraph of G created by removing all vertices in R(U) from G. Along the way we perform some checks that no two odd vertices are adjacent to each other and no two matched edges are both even; if either of these fails we can conclude that no minimum vertex cover containing Z exists in G. We also must construct the tree starting from each vertex in A and ensure that they all match up.

```
bool exploreOdd (Graph& G, const EdgeSet& M, const int v, list<int> firstEven, list<int> firstOdd) {
	G.odd.push_back(v);

	if (!firstOdd.empty() && find(firstOdd.begin(), firstOdd.end(), v) == firstOdd.end()) {
		//v was labelled odd in this tree but not in the first tree
		return false;
	}

	bool ans = true;

	for (int u : G.adjList[v]) {
		if (find(G.even.begin(), G.even.end(), u) != G.even.end()) {
			//Ignore this edge
		}
		else if (find(G.odd.begin(), G.odd.end(), u) != G.odd.end()) {
			//If two odd vertices are adjacent, return false
			return false;
		}
		else {
			//Continue as normal
			ans &= exploreEven(G, M, u, firstEven, firstOdd);
		}
	}

	return ans;
}
```

```
bool exploreEven(Graph& G, const EdgeSet& M, const int v, list<int> firstEven, list<int> firstOdd) {
	G.even.push_back(v);

	if (!firstEven.empty() && find(firstEven.begin(), firstEven.end(), v) == firstEven.end()) {
		//v was labelled even in this tree but not in the first tree
		return false;
	}

	bool ans = true;

	for (const Edge& it : M.E) {
		if (it.get(0) == v) {
			//Matched edge has two even endpoints
			if (find(G.even.begin(), G.even.end(), it.get(1)) != G.even.end()) {
				return false;
			}
			ans &= exploreOdd(G, M, it.get(1), firstEven, firstOdd);
		}
		else if (!it.directed && it.get(1) == v) {
			//Matched edge has two even endpoints
			if (find(G.even.begin(), G.even.end(), it.get(0)) != G.even.end()) {
				return false;
			}
			ans &= exploreOdd(G, M, it.get(0), firstEven, firstOdd);
		}
	}

	return ans;
}
```

```
list<int> CalculateS1(const Graph& G, const list<int>& R, const EdgeSet& M) {
	//Construct alternating tree from each vertex in R
	//Ensure labellings match up, if not return FAIL
	//Ensure no two odd vertices are adjacent, if not return FAIL
	//Ensure no two matched vertices are both even, if not return FAIL
	//Else return the even vertices
	bool ans = true;
	list<int> firstEven;
	list<int> firstOdd;

	for (int i : R) {
		Graph H(G);
		H.odd.clear();
		H.even.clear();
		ans &= exploreEven(H, M, i, firstEven, firstOdd);
		if (i == R.front()) {
			firstEven = H.even;
			firstOdd = H.odd;
		}
	}

	if (ans == false) {
		//One of the trees contains a matching between even vertices or two adjacent odd
vertices
		return list<int>();
	} else {
		return firstEven;
	}
}
```

This method calculates S_2, the set of vertices from A such that neither the vertex nor its matching partner from B are in S_1. The method here adds all vertices such that neither matching vertex is included and then takes any vertex from B from this set, this is due to the matching data structure not providing as easy way to determine which vertex of each edge is in A and which is in B.

```
//Find all matched vertices such that neither it nor its partner are in S1, then set minus B
list<int> CalculateS2(const Graph& G, const EdgeSet& M, const list<int>& B, const list<int>& S1) {
	list<int> S2;

	for (const Edge& e : M.E) {
		if (find(S1.begin(), S1.end(), e.get(0)) == S1.end() && find(S1.begin(), S1.end(),
e.get(1)) == S1.end()) {
			S2.push_back(e.get(0));
			S2.push_back(e.get(0));
		}
	}

	SetMinusVertices(S2, B);

	return S2;
}
```

This method tests the candidate vertex cover $S = S_1 \cup S_2$ to ensure it is in fact a minimum vertex cover containing Z. To do this we just check each of the three properties; that removing all edges from vertices in S leaves a graph with zero edges – that is, S is a vertex cover, that the size of S equals the size of A – that is, S is minimum, and that S intersect Z is equal to Z – that is, Z is contained in S.

```
bool TestS(const Graph& G, const list<int>& S, const int size, const list<int>& Z) {
	Graph H(G);
	for (int i : S)
		H.removeEdgesFromVertex(i);

	list<int> I = IntersectVertices(S, Z);
	I.sort();

	return H.m == 0 && S.size() == size && I == Z;
}
```

The MinimumVertexCoverContaining method is the main entry point into the algorithm, it can call all that is necessary to find a minimum vertex cover containing Z in G from nothing. However to save resources in practise when testing, some of these objects which do not depend on Z were computed just once then passed to this method – that being the maximum matching M, the minimum vertex cover A and the corresponding independence set B, and finally the set R(U) of vertices reachable from an unsaturated vertex with respect to M in the graph G_{bip} and the graph G/U, that is G with R(U) removed. Following these objects, the method calculates S_1 and S_2 and their union, then tests if this set does fulfil the required conditions.

```
list<int> MinimumVertexCoverContaining(Graph& G, list<int>& Z) {
	EdgeSet M;
	list<int> A, B, R_U;
	Graph G_U(0);
	return MinimumVertexCoverContaining(G, Z, M, A, B, R_U, G_U);
}
```

```
list<int> MinimumVertexCoverContaining(Graph& G, list<int>& Z, EdgeSet& M, list<int>& A, list<int>&
B, list<int>& R_U, Graph& G_U) {
	Z.sort();

	if (M.E.empty()) {
		//Calculate Max Matching if not already done
		M = FindMaxMatching(G, M);
	}

	if (A.empty()) {
		//Calculate (A,B) partition if not already done
		A = MinVertexCoverStart(G, M);
	}

	if (B.empty()) {
		//Calculate (A,B) partition if not already done
		for (int i = 0; i < G.n; i++)
			if (find(A.begin(), A.end(), i) == A.end())
				B.push_back(i);
	}

	//Calculate L and R sets
	list<int> R = IntersectVertices(Z, B);

	if (R_U.empty()) {
		//Calculate G_bip(A->B, M) graph
		Graph G_bip = ConstructG_bip(G, M, A);

		//Calculate U, R(U)
		list<int> U = UnsaturatedVertices(G, M);
		list<int> R_U = ReachableFromVertices(G_bip, U);
	}

	list<int> X = IntersectVertices(A, R_U);

	//Check (Z intersect R(U)) - X is empty
	if (!SetMinusVertices(IntersectVertices(Z, R_U), X).empty()) {
		return list<int>();
	}

	if (G_U.n == 0) {
		//Calculate G/U graph
		Graph G_U = RemoveVerticesFromGraph(G, R_U);
	}

	list<int> t = SetMinusVertices(R, R_U);
	list<int> S1 = CalculateS1(G_U, t, M);
	S1.sort();

	list<int> S2 = CalculateS2(G, M, B, S1);
	S2.sort();

	list<int> S = UnionVertices(S1, S2);

	if (TestS(G, S, A.size(), Z)) {
		PrintVertices("Min Vertex Cover: ", S);
		return S;
	}

	return list<int>();
}
```

5.6 Uniqueness of Minimum Vertex Cover

The final method [1] used in the algorithm is another entry point method. This is called just the once per graph, whereas the previous method is called multiple times and so here is where in practise we compute all the objects mentioned before, we then pass them to each instance of the MinimumVertexCoverContaining method. After these have been computed, we are check that for every matching edge (a, b) with a in A, if, when taking Z = {b}, there is a minimum vertex cover containing Z – if so, A is not unique, else it is.

```
list<int> UniqueMinimumVertexCover(Graph& G) {
	//Calculate Max Matching
	EdgeSet M;
	M = FindMaxMatching(G, M);
	M.print("M Length " + to_string(M.E.size()) + ":");

	//Calculate (A,B) partition
	list<int> A = MinVertexCoverStart(G, M);
	PrintVertices("Min Vertex Cover: ", A);
	list<int> B;
	for (int i = 0; i < G.n; i++)
		if (find(A.begin(), A.end(), i) == A.end())
			B.push_back(i);

	//Calculate G_bip(A->B, M) graph
	Graph G_bip = ConstructG_bip(G, M, A);

	//Calculate U, R(U), X sets
	list<int> U = UnsaturatedVertices(G, M);
	list<int> R_U = ReachableFromVertices(G_bip, U);

	//Calculate G/U graph
	Graph G_U = RemoveVerticesFromGraph(G, R_U);

	//For every matching edge mi = (ai, bi), we want to find a min vertex cover containing bi
	//If any of these return a minimum vertex cover, then we know A is not unique

	for (const Edge& e : M.E) {
		if (find(A.begin(), A.end(), e.get(0)) != A.end()) {
			Graph H(G);
			list<int> Z;
			Z.push_back(e.get(1));
			if (!MinimumVertexCoverContaining(H, Z, M, A, B, R_U, G_U).empty())
				return list<int>();
		} else {
			Graph H(G);
			list<int> Z;
			Z.push_back(e.get(0));
			if (!MinimumVertexCoverContaining(H, Z, M, A, B, R_U, G_U).empty())
				return list<int>();
		}
	}

	return A;
}
```

5.7 File Handling: Dataset and Results

In order to test the algorithm on a large dataset, it was necessary to import an externally sourced dataset [4] of both general and bipartite graphs. These graphs were represented as a text file per graph in the following form:

```
n	m
v1 u1
v2 u2
…
vm um
```

So, in order to parse this, we simply need to take the first line and split it by a tab, then loop until the file end and add an edge for each line of the file, splitting the endpoint labels of each by a space. The following method does this and returns an undirected graph in our own Graph structure.

```
Graph ReadInGraph(string fileName) {
	string text;
	ifstream file(fileName + ".txt");

	//	n	m - Graph parameters
	getline(file, text);
	int n = stoi(text.substr(2, text.find("	")));

	Graph G(n);

	//vi ui - An edge in the graph
	while (getline(file, text)) {
		int v = stoi(text.substr(0, text.find(' ')));
		int u = stoi(text.substr(text.find(' '), text.size()));
		G.addEdge(v, u);
	}

	file.close();

	return G;
}
```

Following the importing of the dataset, it was then necessary to combine all pairs of order n/2 general graphs and order n bipartite graphs into order n König-Egerváry graphs. The method for this was to simply add all the edges of the general graph onto the R set of the bipartite graph and then, to ensure there exists an R-perfect matching on this new graph, add one edge between each vertex in R and a unique vertex in L.

```
//Requires the input graphs to be such that the Bipartite is split with vertices 1,...,|L| all in L
//and the remainder |L|+1,....,|L|+|R| are in R and the General has at most |L| vertices
Graph CombineGeneralBipartite(string generalPath, string bipartitePath) {
	Graph G = ReadInGraph(bipartitePath);

	string text;
	ifstream file(generalPath + ".txt");

	//n   m - Graph parameters
	getline(file, text);
	int n = stoi(text.substr(2, text.find("   ")));

	//vi ui - An edge in the graph
	while (getline(file, text)) {
		int v = stoi(text.substr(0, text.find(' ')));
		int u = stoi(text.substr(text.find(' '), text.size()));
		G.addEdge(v - 1, u - 1);
	}

	file.close();

	//Add a perfect matching to ensure the graph is Konig
	for (int i = 0; i < n; i++)
	{
		G.addEdge(i, i + n);
	}

	return G;
}
```

These new graphs were then saved in the same file format as the dataset such that when testing with one, the ReadInGraph method can be reused.

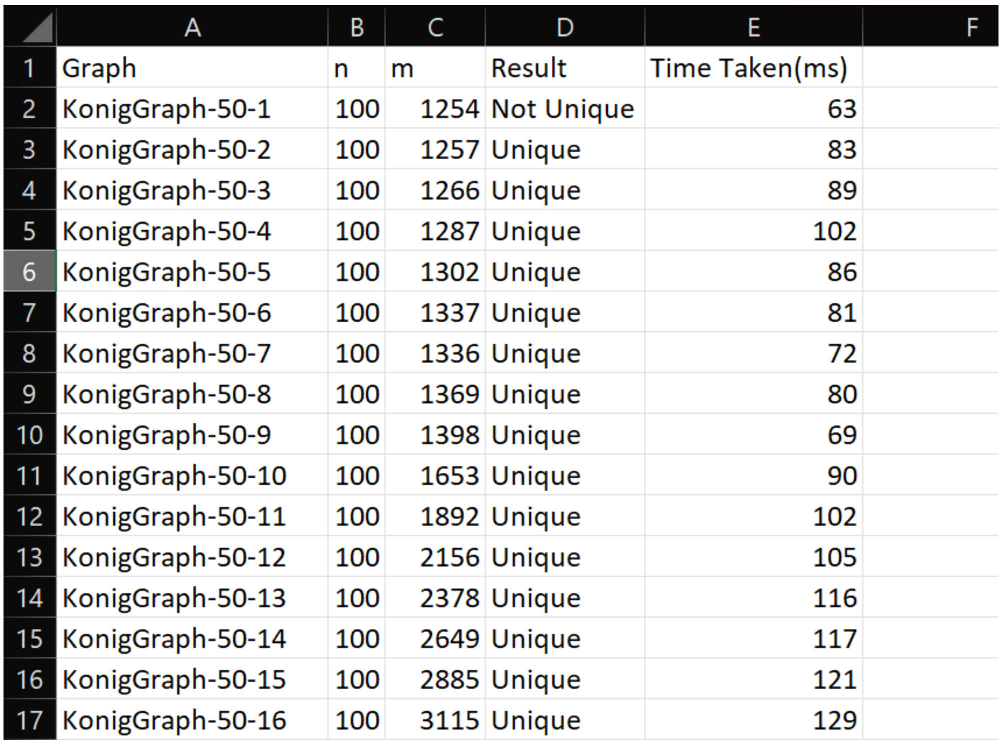
```
void WriteGraphToFile(const Graph& G, const string fileName) {
	ofstream file(fileName + ".txt");

	file << "  " << G.n << "   " << G.m << "\n";

	for (int i = 0; i < G.n; i++)
		for (int j : G.adjList[i])
			file << i << " " << j << "\n";

	file.close();
}
```

5.8 Testing

Finally, to test the algorithm and obtain runtime results, the following method uses the std::chrono library to record the start and end times of an entire run of the algorithm on one graph. This value is then added to a .csv file with the graph size, the result (unique or not unique) and the time taken in milliseconds.

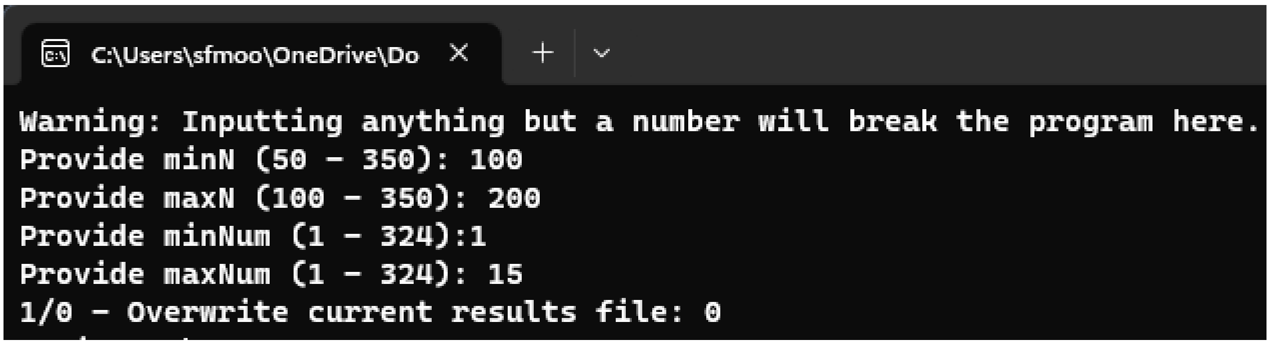
```cpp
void runAlgorithm(Graph& G, string name = "G") {
        cout << name + " - n=" << G.n << " m=" << G.m << "\n";

        unsigned __int64 start = chrono::duration_cast<chrono::milliseconds>(
                chrono::system_clock::now().time_since_epoch()).count();

        list<int> S = UniqueMinimumVertexCover(G);

        unsigned __int64 end = chrono::duration_cast<chrono::milliseconds>(
                chrono::system_clock::now().time_since_epoch()).count();

        ofstream results = ofstream("results.csv", ios_base::app);

        results << name + "," + to_string(G.n) + "," + to_string(G.m);

        if (S.empty()) {
                cout << "G does not have a unique minimum vertex cover\n";
                results << ",Not Unique,";
        } else {
                cout << "This is the unique vertex cover in G\n";
                results << ",Unique,";
        }

        cout << "Time taken: " << (end - start) << "ms\n\n";

        results << (end - start) << "\n";

        results.close();
}
```

The results recorded for each graph could then be imported into an Excel spreadsheet in the following form:



	A	B	C	D	E	F
1	Graph	n	m	Result	Time Taken(ms)	
2	KonigGraph-50-1	100	1254	Not Unique	63	
3	KonigGraph-50-2	100	1257	Unique	83	
4	KonigGraph-50-3	100	1266	Unique	89	
5	KonigGraph-50-4	100	1287	Unique	102	
6	KonigGraph-50-5	100	1302	Unique	86	
7	KonigGraph-50-6	100	1337	Unique	81	
8	KonigGraph-50-7	100	1336	Unique	72	
9	KonigGraph-50-8	100	1369	Unique	80	
10	KonigGraph-50-9	100	1398	Unique	69	
11	KonigGraph-50-10	100	1653	Unique	90	
12	KonigGraph-50-11	100	1892	Unique	102	
13	KonigGraph-50-12	100	2156	Unique	105	
14	KonigGraph-50-13	100	2378	Unique	116	
15	KonigGraph-50-14	100	2649	Unique	117	
16	KonigGraph-50-15	100	2885	Unique	121	
17	KonigGraph-50-16	100	3115	Unique	129	

To give a little control over the testing process, a simple console-based interface is implemented here to allow for specifying which graphs in the set of König-Egerváry graph files should be used and whether the .csv file should be overwritten or appended to. Below is a screenshot showing this interface:



```
Warning: Inputting anything but a number will break the program here.
Provide minN (50 - 350): 100
Provide maxN (100 - 350): 200
Provide minNum (1 - 324):1
Provide maxNum (1 - 324): 15
1/0 - Overwrite current results file: 0
```

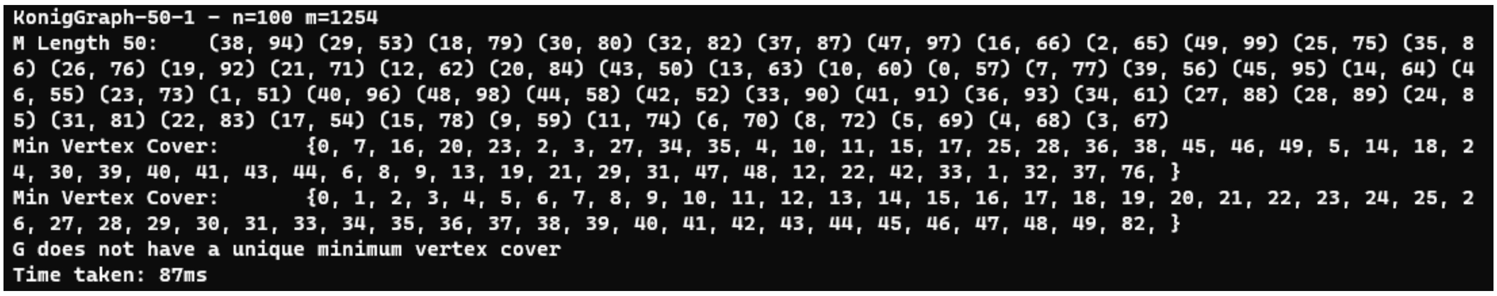
```
void TestDatasetForUniqueVertexCovers() {

	int minN = -1;		//50
	int maxN = -1;		//350
	int minNum = -1;	//1
	int maxNum = -1;	//18*18 = 324
	bool overwrite = false;

	while (!(minN >= 50 && minN <= 350 && maxN >= minN && maxN <= 350 && minNum >= 1 && minNum <=
324 && maxNum >= minNum && maxNum <= 324)) {
		cout << "Warning: Inputting anything but a number will break the program here.\n";
		cout << "Provide minN (50 - 350): ";
		cin >> minN;
		cout << "Provide maxN (" + to_string(minN) + " - 350): ";
		cin >> maxN;
		cout << "Provide minNum (1 - 324): ";
		cin >> minNum;
		cout << "Provide maxNum (" + to_string(minNum) + " - 324): ";
		cin >> maxNum;
		cout << "1/0 - Overwrite current results file: ";
		cin >> overwrite;
	}

	if (overwrite) {
		ofstream results = ofstream("results.csv");
		results << "Graph,n,m,Result,Time Taken(ms),,Total(s),=SUM(E:E)/1000\n";
		results.close();
	}

	//Load each Konig graph and run it
	for (int n = minN; n <= maxN; n += 50) {
		for (int i = minNum; i <= maxNum; i++) {
			Graph G = ReadInGraph("KonigGraph-" + to_string(n) + "-" + to_string(i));
			runAlgorithm(G, "KonigGraph-" + to_string(n) + "-" + to_string(i));
		}
	}
}
```

```
int main() {

	TestDatasetForUniqueVertexCovers();

	cout << "\n";
	system("pause");
}
```

6 Evaluation

6.1 Results

The program itself does provide an output for each graph into the console, mostly used for debugging the program during development. However, it does show that the implementation not only determines the answer ‘Yes’ or ‘No’ but computes the maximum matching, first minimum vertex cover A and then any other vertex covers it finds which are distinct. In addition, the time taken to compute this is displayed.



```
KonigGraph-50-1 - n=100 m=1254
M Length 50:    (38, 94) (29, 53) (18, 79) (30, 80) (32, 82) (37, 87) (47, 97) (16, 66) (2, 65) (49, 99) (25, 75) (35, 8
6) (26, 76) (19, 92) (21, 71) (12, 62) (20, 84) (43, 50) (13, 63) (10, 60) (0, 57) (7, 77) (39, 56) (45, 95) (14, 64) (4
6, 55) (23, 73) (1, 51) (40, 96) (48, 98) (44, 58) (42, 52) (33, 90) (41, 91) (36, 93) (34, 61) (27, 88) (28, 89) (24, 8
5) (31, 81) (22, 83) (17, 54) (15, 78) (9, 59) (11, 74) (6, 70) (8, 72) (5, 69) (4, 68) (3, 67)
Min Vertex Cover:       {0, 7, 16, 20, 23, 2, 3, 27, 34, 35, 4, 10, 11, 15, 17, 25, 28, 36, 38, 45, 46, 49, 5, 14, 18, 2
4, 30, 39, 40, 41, 43, 44, 6, 8, 9, 13, 19, 21, 29, 31, 47, 48, 12, 22, 42, 33, 1, 32, 37, 76, }
Min Vertex Cover:       {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 2
6, 27, 28, 29, 30, 31, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 82, }
G does not have a unique minimum vertex cover
Time taken: 87ms
```

The overall results of the project are displayed in Figure 9 with the colours of each point indicating the number of vertices in the graph.

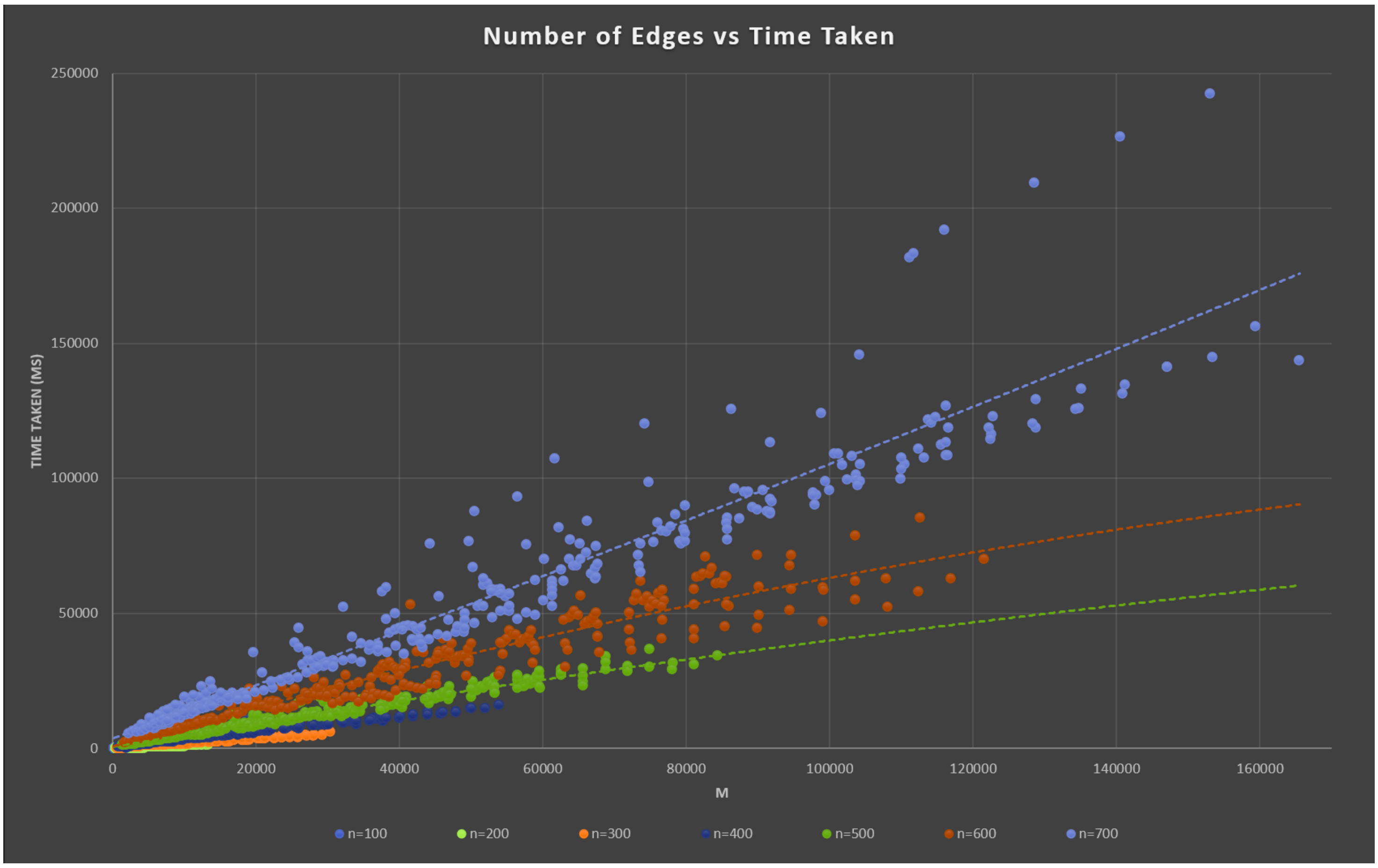


Figure 12 - Number of Edges vs Time Taken linear scale graph of results.

As expected of any algorithm which does not run in constant time, the runtime clearly increases with the size of the graphs; both when we fix the vertex set and increase the density of edges and when we increase the number of vertices but use a similar number of edges.

However, Figure 12 struggles to show the distribution of datapoints correctly, as the majority of the points are packed together tightly near the origin. To solve this, Figure 13 displays the same points on logarithmic axes to better highlight the distinction between the classes of graphs with different numbers of vertices.

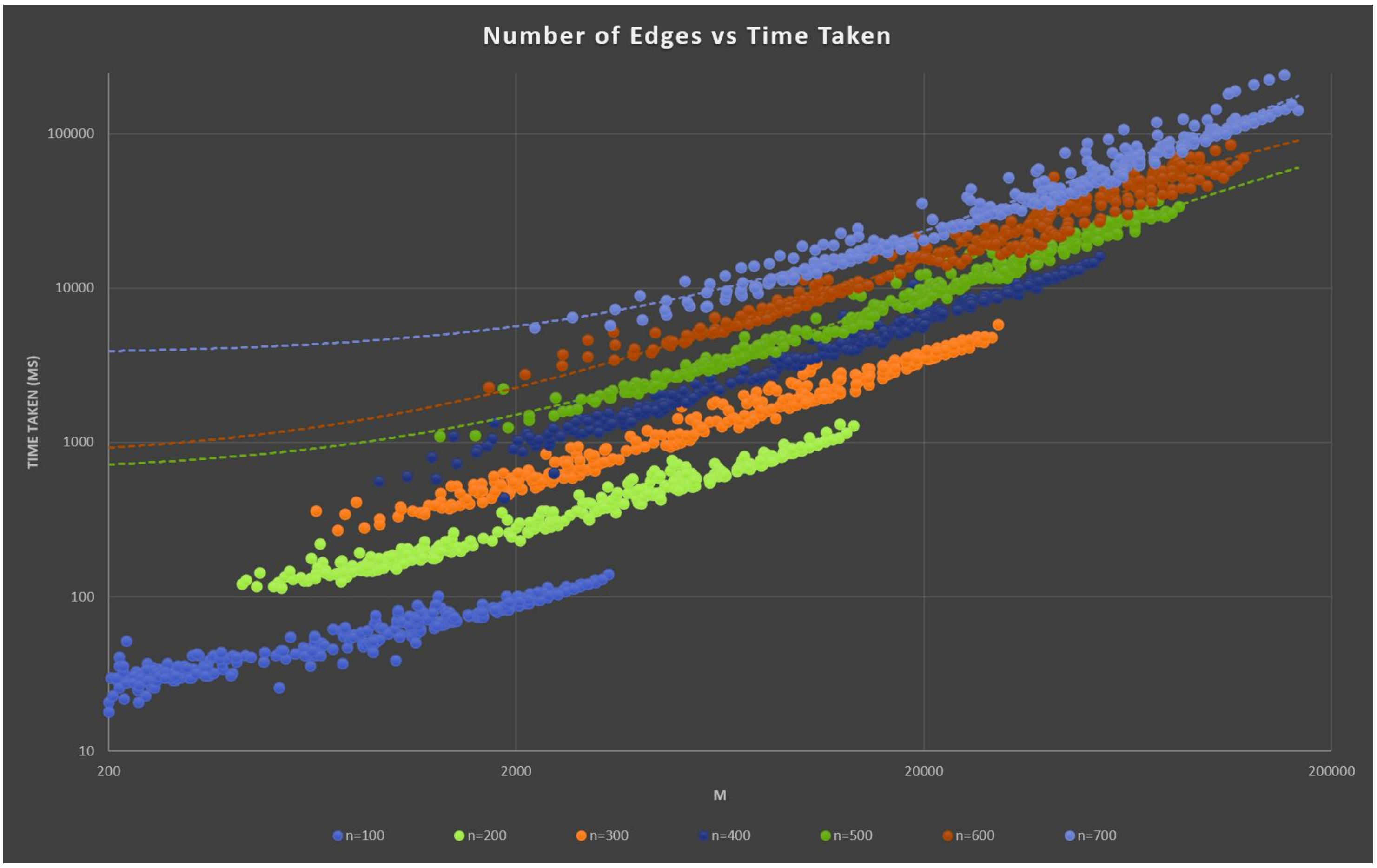


Figure 13 - Number of Edges vs Time Taken logarithmic scale graph of results.

Some further indicative statistics include:

- 56% of the dataset took less than 5 seconds to run.
- 80% of the dataset took less than 20 seconds to run.
- The largest graphs running in under 20 seconds had between 400-700 vertices and 18,000-54,000 edges.
- The maximum runtime on any of the graphs was 4 minutes on 700 vertices and 150,000 edges.
- The largest graphs requested up to 1GB of memory resources.

These results, while they show that the algorithm is certainly fast on small to medium sized graphs, show that in a real-world scenario where graphs could have millions of edges this algorithm could take hours to run and so would likely not be useful in a setting where quick results might be required. It also doesn't have an entirely insignificant effect on space resources which might have to be considered seriously on, again, any real-world graphs with millions of edges.

However, there are few additional considerations; firstly, these results were obtained on a personal laptop, not a powerful computer representative of what might be available to a large company working with these larger graphs.

Secondly, the algorithm proposed in the theoretical setting was a Corollary to the algorithm to compute a minimum vertex cover containing a specified subset of vertices and involved invoking this routine O(n) times – it is very possible that this is not the most efficient way to use the same idea to determine the uniqueness of the minimum vertex cover. For example, in this algorithm, we only ever call MinimumVertexCoverContaining on a singleton set Z = {b} for b in B but there are operations performed in the routine which are only required if Z is larger and intersects A, these could be removed if the routine was only required for this purpose. In addition to this, the majority of the runtime spent on any one graph was in this computing of the extended minimum vertex covers trying to find one distinct from A, as shown in Figure 14 as an example, and so if we only wanted to know whether this was true for one set Z, we would be reducing the most time demanding part of the process by an O(n) factor.

Thirdly, the implementation did not make use of any multi-threading or performance optimisations. There are clear ways throughout this algorithm where multi-threading could be implemented; it is a common method in Edmond’s Blossom algorithm to construct alternating trees from all free vertices simultaneously and then handle intersections later and it would also be very possible to thread all the invocations of MinimumVertexCoverContaining as they are independent queries. Regarding other optimisations, a large portion of the CPU time was spent searching through lists and removing specific vertices from lists; these could potentially be avoided by using different data structures with better average find and remove time complexities.

A final consideration regarding the space complexity of the algorithm; as is displayed in Figure 14 on a graph with n=700, m=165,000, the memory usage is very clearly significantly worse in the first part of the algorithm, specifically during the Blossom algorithm for maximum matching. This is likely due to an issue in the implementation due to how large the disparity is rather than an intrinsic issue with the algorithm and so may not be indicative of the space complexity of the algorithm overall. The maximum memory usage elsewhere during the process is around 60MB for this example and so this might be more representative.

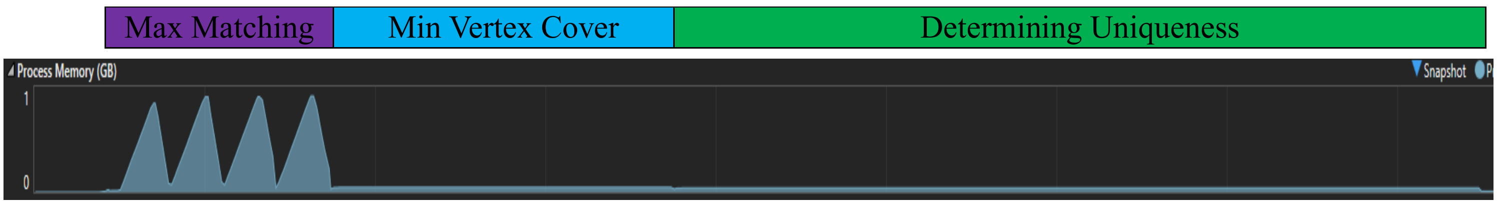


Figure 14 - Memory usage of a single run of the algorithm on a large graph. The break down of what was being computed throughout is displayed above.

6.2 Challenges

Throughout the project, there were many challenges faced, both anticipated and unanticipated. Due to the selection of C++ as the language to work with, it was expected that it would take some time to learn the very specific difference between it and other languages that I had previously worked with, which were generally higher-level languages. A major example of this stemmed from a lack of experience in memory management in C++; due to the lack of a destructor of the Graph class, whenever a Graph object was deleted – usually due to leaving the scope of a method – the entire adjList structure would remain stored. This led to the program requesting over 2GB of memory when either testing a long series of small graphs or even when testing just a single large graph and due to the default build settings for Visual Studio solutions, the Windows OS would simply refuse and force close the program.

The most impactful challenge faced was unexpected delays in developing a high-quality implementation of the algorithms. This was because it was not known exactly how many other routines the proposed algorithm relied on – for example, an early version of the implementation did not make use of the Semipartite testing algorithm to find the minimum vertex cover, and instead naively brute forced the process in exponential time which caused the program to take significantly longer to halt. Similarly, the process of contracting and uncontracting blossoms in the Blossom algorithm for maximum matching is a lot easier to describe in words than to implement.

Another unexpected challenge was the complexity of the second objective algorithm – researching and implementing this was too big of an undertaking for the scope of this project, and fulfilling this objective would likely need an entire project dedicated to it from the start. This meant a lot of time in Term 2 was spent attempting to understand a lot of new concepts without any results. The cascading effects of these two significant timing issues then led to the third objective being unattainable too due to time constraints.

7 Conclusions and Future Work

In summary, this project has validated that the algorithm to determine the uniqueness of the minimum vertex cover in König-Egerváry graphs can perform well on small and medium-sized graphs but there is a fast increase in runtime as the graphs get larger with the largest graphs tested taking 4 minutes while being orders of magnitude smaller than real-world graph could potentially be. However, this should only be taken as a guideline for these extremely large graphs as the conclusions here are due to extrapolation and only the performance on graphs on up to 700 vertices can be fairly assessed with our results.

Regarding the original project goals, the project did unfortunately fail to achieve the final two of its three goals. This was mainly due to underestimates in timings for the first objective and an underestimate of the complexity of the second objective algorithm. In hindsight, more initial research should have been completed into the three algorithms to determine their suitability for this project – if this had been done it is likely that the project goals would instead have been (a) and (c) and the project would have achieved both as less time would have been spent attempting (b).

As has been discussed throughout this report, there are a few options for extending the findings of this project; firstly it would be natural to look for results of this implementation of a dataset of König-Egerváry graphs which are more general, and not restricted to those with $|A| = |B|$ and instead allow B to be larger. It would also be preferential to test graphs even larger than those tested here, to give a better idea of how the algorithm behaves as the input size increases as is likely if we were to use real-world graphs.

It would also likely be beneficial to optimise and improve the implementation of the algorithm to include multi-threading, use a more advanced maximum matching algorithm and reduce dependencies on basic C++ list operations such as searching for and removing entries. These could provide a better idea of how practical the algorithm is and whether it would work well in a real-world solution to a problem. Similarly, an implementation used for real-world problems would still need to be tested on the exact hardware that is available for use, as this project was restricted to just the performance of the implementation on a personal laptop.

8 References

[1] V. Raman et al., A characterization of König-Egerváry graphs with extendable vertex covers, Inf Process. Lett. (2020)

[2] Edmonds, J. (1965). Paths, Trees, and Flowers. Canadian Journal of Mathematics, 17, 449-467.

[3] Fănică Gavril, Testing for equality between maximum matching and minimum node covering, Information Processing Letters, Volume 6, Issue 6, 1977, Pages 199-202, ISSN 0020-0190,

[4] Kartelj, Aleksandar (2019): General graph datasets. 4TU.ResearchData. Dataset.

[5] M. R. Fellows, F. V. Fomin, D. Lokshtanov, F. A. Rosamond, S. Saurabh, and Y. Villanger. Local search: Is brute-force avoidable? Journal of Computer and System Sciences, (2012)

[6] D. Lokshtanov, N. S. Narayanaswamy, V. Raman, M. S. Ramanujan, and S. Saurabh. Faster parameterized algorithms using linear programming. ACM Transactions on Algorithms, (2014)

[7] Micali, Silvio; Vazirani, Vijay (1980). *An $O(V^{1/2}E)$ algorithm for finding maximum matching in general graphs*. 21st Annual Symposium on Foundations of Computer Science. IEEE Computer Society Press, New York. pp. 17–27.

[8] Kőnig, Dénes *(1931), "Gráfok és mátrixok", Matematikai és Fizikai Lapok, 38: 116–119.*