DEPARTMENT OF COMPUTER SCIENCE

# Edge-Degree-Constraint-Subgraphs And Their Applications

A dissertation submitted to the University of Bristol in accordance with the requirements of the degree of Bachelor of Science in the Faculty of Engineering.

Friday 6th May, 2022

# Abstract

A maximal matching is the largest possible matching for a graph. There are many algorithms that can calculate maximal matchings for an input graph, but with the rise in big data, research has focused on constructing algorithms that can compute large matching approximations that use as little storage space as possible. This project studies and implements an algorithm that in theory can compute a $\frac{2}{3}$ - approximate matching using $O(nlog(n))$ space for any graph. I experiment with real world data sets to see what approximations the algorithm actually produces.

- I spent time learning about the algorithm and how it uses Edge-degree-constraint-subgraphs to achieve a $\frac{2}{3}$ - approximate matching in theory
- I implemented the algorithm in python and tested it on real world data sets.
- I ran three different experiments on the parameters of the algorithm to see if we actually achieve a $\frac{2}{3}$ - approximate matching with the algorithm in practice.

# Declaration

I declare that the work in this dissertation was carried out in accordance with the requirements of the University's Regulations and Code of Practice for Taught Programmes and that it has not been submitted for any other academic award. Except where indicated by specific reference in the text, this work is my own work. Work done in collaboration with, or with the assistance of others, is indicated as such. I have identified all material in this dissertation which is not my own work through appropriate referencing and acknowledgement. Where I have quoted or otherwise incorporated material which is the work of others, I have included the source in the references. Any views expressed in the dissertation, other than referenced material, are those of the author.

 Friday 6th May, 2022

# Contents

# List of Figures

# List of Tables

# Ethics Statement

This project did not require ethical review, as determined by my supervisor, Christian Konrad.

# Notation and Acronyms

| | | |
|---|---|---|
| G | : | The input graph |
| V(G) | : | The set of vertices of G |
| E(G) | : | The set of edges of G |
| H | : | A subgraph of G |
| S | : | The stream of edges for the input graph |
| $S_{[i,j]}$ | : | The substream $\langle e_i, ..., e_j \rangle$ |
| EDCS | : | Edge-Degree-Constrained-Subgraph |
| n | : | \|V\| |
| m | : | \|E\| |
| $M_G$ | : | A matching in G |
| $deg_H(v)$ | : | The degree of v in H |
| $\mu(H)$ | : | The size of the maximum matching in H |
| $G_{>i} \subseteq G$ | : | The subgraph of G containing all edges in $\{e_{i+1}, ..., e_m\}$ |

# Chapter 1

# Introduction

There are many known algorithms that can compute a maximum cardinality matching for a graph. The rise of big data has motivated research into developing algorithms that can compute large matching approximations, whilst only storing a fraction of the graph. A particular algorithm was developed by Aaron Bernstein [[2]]. This algorithm exploits properties of Edge-degree-constraint-subgraphs, a notion defined by Bernstein and Stein [[3]], which are subgraphs with specific degree properties. It has been shown that Edge-degree-constraint-subgraphs always achieve a $\frac{2}{3}$ - approximate matching, however it is currently unknown if it is possible to construct an EDCS using less than $O(n^{1.5})$ space. The algorithm developed by Aaron Bernstein [[2]], uses the specific degree properties of EDCS to construct an algorithm that in theory can compute a $\frac{2}{3}$ - approximate matching using $O(nlog(n))$ space for any graph. The algorithms result builds on previous work in this area of research; an algorithm constructed by Assadi et al. [[1]] also computes a $\frac{2}{3}$ - approximate matching for any input graph, however, the algorithm uses $O(n^{1.5}polylog(n))$, a far greater space requirement compared to this algorithm.

This paper focuses on implementing the algorithm developed by Aaron Bernstein [[2]] on real world data sets. The main aim is to see if we do in fact see $\frac{2}{3}$ - approximate matchings on different graphs, by varying the parameters of the algorithm.

The main aims of this project are:

- To understand the algorithm
- To implement the algorithm using real world data sets
- To conduct different experiments with the different parameters of the algorithm, to push it to its limits, and to see what we can learn from the results

We will first understand what it means for a matching to be a maximal cardinality matching, before introducing matching approximations. I cover what it means for a subgraph to be an Edge-Degree constrained subgraph, with some examples, and then look at the algorithm that exploits properties of EDCS in detail. I then explain the purpose of each parameter of the algorithm, and why in theory the value of parameters may not the most optimal when looking at real world data sets. I then discuss the experiments I have chosen to conduct to test the algorithm, and to see if we do in fact achieve this $\frac{2}{3}$ - maximal cardinality matching approximation, before analysing these results.

# Chapter 2

# Preliminaries

The main goal of this chapter is to understand what a maximum cardinality matching approximation is. This approximation is what the algorithm ultimately computes. To understand this fully, I first introduce the concept of a matching, and what it means for a matching to be a maximum cardinality matching.

## 2.1 Definitions

### 2.1.1 Matchings

A matching in a graph is a set of edges, such that every vertex in the graph is incident to at most one of these edges. This can be formalised in the following definition:

**Definition 2.1** Let G = (V,E) be a graph. A matching $M \subseteq E$, is a set of edges, such that every vertex of V is incident to at most one edge of $M$.

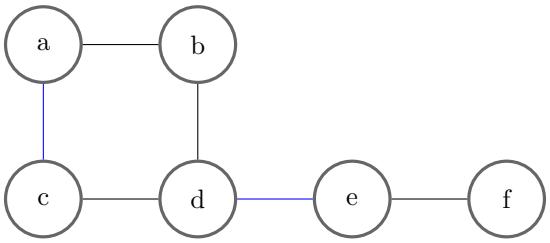



Figure 2.1: Graph G showing a Matching

Figure 2.1 shows an example of a matching for the graph G. The matching $M = \{ac, de\}$ is shown by the blue edges. Note that this matching is not unique; most graphs have many different matchings. The matching of maximum possible size, that is $|M|$ is as large as possible, is known as a maximum cardinality matching. This is formalised in the definition below:

**Definition 2.2** Let G = (V,E) be a graph. A matching $M \subseteq E$, is a maximum cardinality matching if $M$ is of maximum size.

Figure 2.2 shows an example of a Maximal Cardinality Matching for the graph G. This matching $M = \{ac, bd, ef\}$ is shown by the green edges. It can be seen that no other edge can be added to $M$ without being incident to a vertex that is already incident to an edge in $M$. This matching is a Maximal Cardinality Matching for G as there does not exist a matching $M$ with $|M| > 3$. Note that a Maximal Cardinality Matching isn't necessarily unique.

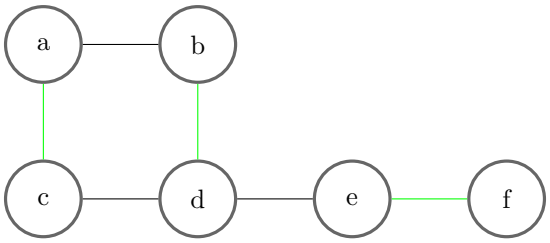



Figure 2.2: A Maximal Cardinality Matching for the graph G

### 2.1.2 Matching Approximations

Now that we have defined what it means for a matching to be a maximum cardinality matching, we can introduce the notion of a Matching Approximation. The algorithm outputs the maximum matching of the subgraph it constructs. We can find the size of this matching and compare it to the true size of the maximum cardinality matching of the original graph to calculate the approximation. This can be formalised with the below definition.

**Definition 2.3** Let $G$ be a graph, $H$ a subgraph of $G$. The Maximal Cardinality Matching Approximation for the subgraph $H$ of $G$ is the value $\frac{\mu(H)}{\mu(G)}$.

# Chapter 3

# The Algorithm

The main goal of this chapter is to introduce the algorithm that exploits properties of EDCS a notion first introduced in [3]. I introduce the notion of an EDCS with some examples, and then explain how each stage of the algorithm works.

## 3.1 EDCS

### 3.1.1 Definition And Lemma

Given any undirected graph $G$, a subgraph $H$ of $G$, is said to be an EDCS if it has specific properties. To obtain the $(\frac{2}{3}-\epsilon)$-approximate matching in G, the algorithm exploits these properties. I now introduce what an EDCS is, and some notations that directly follow from the definition with some examples.

**Definition 3.1** Let G = (V,E) be a graph, and H = (V(G),E) be a subgraph of G. Given any parameters $\beta \geq 2$ and $\lambda < 1$, H is a ($\beta$, $\lambda$) - EDCS of G if H satisfies:

- **Property P1:** For any edge $(u, v) \in H$, $deg_H(u) + deg_H(v) \leq \beta$
- **Property P2:** For any edge $(u, v) \in G \setminus H$, $deg_H(u) + deg_H(v) \geq \beta(1 - \lambda)$

It is these two properties that the algorithm exploits to obtain the $(\frac{2}{3} - \epsilon)$-approximate matching in G. This notation can be summarised with the following lemma proved in [[2]] below.

**Lemma 3.2** Let G be any graph and $\epsilon < \frac{1}{2}$ be some parameter. Let $\lambda$ and $\beta$ be parameters such that $\lambda \leq \frac{\epsilon}{64}$, $\beta \geq 8\lambda^{-2}log(\frac{1}{\lambda})$. Then, for any $(\beta, \lambda)$ - EDCS H of G, we have that $\mu(H) \geq (\frac{2}{3} - \epsilon)\mu(G)$.

The lemma states that an EDCS always contains close to a $\frac{2}{3}$-approximate matching. However, the algorithm doesn't actually construct an EDCS; this is due to the fact that there is not a known way of constructing an EDCS for any graph, in less than $O(n^{1.5})$ space. Instead, the algorithm exploits these properties that define whether a subgraph is an EDCS to obtain the $(\frac{2}{3} - \epsilon)$-approximate matching. I now introduce two further definitions that help describe and analyse the algorithm; what it means for any graph to have bounded edge-degree $\beta$, and for an edge $(u, v) \in G \setminus H$ to be (G, H, $\beta$, $\lambda$)-underfull. These follow immediately from the properties that make a subgraph an EDCS.

**Definition 3.3** A graph H has bounded edge-degree $\beta$, if for every edge $(u, v) \in H$, $deg_H(u)+deg_H(v) \leq \beta$.

We see that the graph in 3.1 has bounded edge degree 6. This is because the edge (2,5) highlighted in red is connected to vertices 2 and 5 with $deg_G(2) = deg_G(5) = 3$. Hence $deg_G(2) + deg_G(5) = 6$. It can be seen from inspection that this value for every other edge in this graph is less than 6. Thus this graph has bounded edge-degree 6.

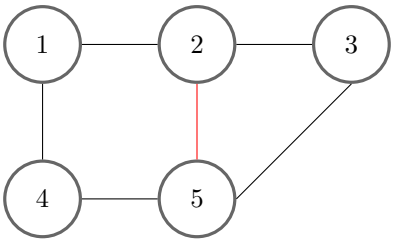



Figure 3.1: Graph G with bounded edge degree 6

**Definition 3.4** Let G be any graph, and let H be a subgraph of G with bounded edge-degree $\beta$. For any parameter $\lambda < 1$, we say that an edge $(u, v) \in G \setminus H$ is (G, H, $\beta$, $\lambda$)-underfull if $deg_H(u) + deg_H(v) \geq \beta(1 - \lambda)$

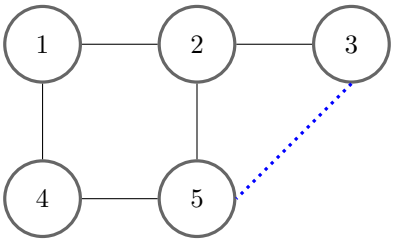



Figure 3.2: Graph H. Edge (3,5) is (G, H, 5, $\frac{1}{2}$)-underfull

We see that the edge $(3, 5) \in G \setminus H$ in 3.2 is (G, H, 5, $\frac{1}{2}$)-underfull. Suppose that the edge $(3, 5) \notin H$ the subgraph, is an edge of $G$. From inspection, it is clear that the subgraph $H$ has bounded edge-degree 5. Thus as $deg_H(3) + deg_H(5) = 1 + 2 = 3$, we have that the edge $(3, 5) \in G \setminus H$ is (G, H, 5, $\frac{1}{2}$)-underfull.

It is these two properties used by the algorithm that ensures you can construct a $\frac{2}{3}$ - approximate matching. This can be summarised and proved from [[2]], but I will also state it here.

**Lemma 3.5** Let $\epsilon < \frac{1}{2}$ be any parameter, and let $\lambda$, $\beta$ be parameters with $\lambda \leq \frac{\epsilon}{128}$, $\beta \geq 16\lambda^{-2}log(1/\lambda)$. Consider any graph G, and any subgraph H with bounded edge-degree $\beta$. Let $X$ contain all edges in $G \setminus H$ that are (G, H, $\beta$, $\lambda$)-underfull. Then $\mu(X \cup H) \geq (2/3 - \epsilon)\mu(G)$.

## 3.2 The Algorithm

In this section, I present a description of the algorithm as described in [[2]].

### 3.2.1 Algorithm description

The Algorithm is made up of three procedures, Initialisation, Phase I and RemoveOverfullEdges, as well as a function, Phase II. The algorithm begins in the initialisation procedure, where the parameters $\epsilon$, $\lambda$, $\beta$ and $\alpha$ are defined. The graph H which will become a subgraph for the input graph G is initially set to the null graph, with vertex set equal to the graph G. The approximation parameter $\epsilon$ is set as a positive real number of less than a half. In theory as we have seen, the algorithm gives a $O(\frac{2}{3} - \epsilon)$ maximum matching approximation, so for the best approximations, this should be set to as small as possible. Once this has been determined, $\lambda$, $\beta$ and $\alpha$ are then automatically defined as their values only rely on $\epsilon$, $m$, and $n$. The algorithm considers the edges of the input graph in turn uniformly randomly, thus creating a random edge stream. The algorithm then continues to Phase I.

In phase I, the algorithm looks at epochs in turn from the random edge stream of size $\alpha$. This means that when the algorithm considers epoch i, the algorithm looks at the stream of edges $S_{[(i-1)\alpha+1,i\alpha]}$ in turn. The boolean FoundUnderfull is set to False every time the algorithm looks at a new epoch. The algorithm then checks to see for the edge $(u, v)$ if $deg_H(u) + deg_H(v) < \beta(1 - \lambda)$, or in other words, if this

particular edge is $(G, H, \beta, \lambda)$-underfull. If this is the case, the edge $(u, v)$ is then added to the subgraph $H$, the boolean FoundUnderfull is set to True for this epoch, and the procedure RemoveOverfullEdges is called. The procedure RemoveOverfullEdges ensures that the subgraph $H$, has bounded edge degree $\beta$. The algorithm does this by looking at all pairs of edges connected to vertices $u$ and $v$ and checks to see if $deg_H(u) + deg_H(v) > \beta$. If this is the case, we remove this edge from the subgraph $H$, and repeat this process until no such edge remains. This ensures that by adding the edge $(u, v)$ to the subgraph, the subgraph still has bounded edge degree $\beta$. Once the algorithm has looked at all edges in epoch $i$, if the boolean FoundUnderfull has been set to True, the algorithm will then move on to look at epcoh $i + 1$. If, however FoundUnderfull is False, i.e. epoch i contained no edges that are $(G, H, \beta, \lambda)$-underfull, then the algorithm terminates phase I, and moves to the function Phase II.

Suppose that the algorithm terminates phase I after the edge $e_j$. In phase II, the algorithm firsts initialises a new graph X, to be the null graph of vertex set equal to the graph G. Then, for every edge left in the stream that wasn't looked at by the algorithm in Phase 1 (i.e. $S_{j+1,m}$), the algorithm checks if $deg_H(u) + deg_H(v) < \beta(1 - \lambda)$ (if $(u, v)$ is $(G, H, \beta, \lambda)$-underfull). If this is the case, the edge $(u, v)$ is added to $X$. Once every edge in the stream has been checked, then the function returns the maximum matching in $H \cup X$. This matching will be a maximal cardinality matching approximation of the original input graph.

### 3.2.2 Algorithm Pseudo code

**Algorithm 1** The algorithm computes a Maximum Cardinality Matching approximation from a random-order stream.

**procedure** *Initialisation*
    H = $\emptyset$
    Let $\epsilon < 1/2$
    Set $\lambda = \frac{\epsilon}{128}$, $\beta = 16\lambda^{-2}log(\frac{1}{\lambda})$, $\alpha = \frac{\epsilon m}{n\beta^2+1}$
    **Go To** Phase I
**end procedure**
**procedure** *Phase I*
    **Do Until Termination**
    $FoundUnderfull \leftarrow FALSE$
    **for** $\alpha$ *Iterations:* **do**
        *Let (u,v) be the next edge in the stream*
        **if** $deg_H(u) + deg_H(v) < \beta(1-\lambda)$ **then**
            *Add edge (u,v) to H*
            $FoundUnderfull \leftarrow TRUE$
            *RemoveOverfullEdges(H)*
    **end**
**end**
**if** *FoundUnderfull = FALSE* **then**
        ***Go To*** *Phase II* ▷ *Else, move on to the next epoch of Phase I*
**end**
***end procedure***
***procedure*** *RemoveOverfullEdges(H)*
    **while** *there exists* $(u, v) \in H$ *such that* $deg_H(u) + deg_H(v) > \beta$ **do**
        *Remove* $(u, v)$ *from H*
**end**
***end procedure***
***function*** *Phase II*
    *Initialise* $X \leftarrow \emptyset$
    **foreach** *remaining edge* $(u, v)$ *in the stream* **do**
        **if** $deg_H(u) + deg_H(v) < \beta(1-\lambda)$ **then**
            *Add edge* $(u, v)$ *to X*
    **end**
**end**
    **return** *the maximum matching in* $H \cup X$
***end function***

# Chapter 4

# Implementation Details

In this section, I first describe the setup approach taken to implement the algorithm, including the reasoning behind the parameters chosen. I then explain the methodology of how I conducted the experiments to test different aspects of the algorithm. Finally, I discuss initial challenges I encountered when developing the code, and how this improved my understanding of the algorithm.

## 4.1 Setup

To be able to implement and analyse the algorithm on real world data sets, I first need to code a working version of the algorithm. I am focusing on implementing and analysing the algorithm, rather than a more engineering approach that focuses on creating an optimised software package, for example. I decided to code the algorithm in python for a few reasons. Firstly, python is the language that I have used the most; it is a language that I know relatively well and am comfortable with. Moreover, I need access to packages that compute the Maximal Cardinality Matching for a graph, and again, this is something that python offers through the Network X package. Other languages such as C++ do offer a similar package such as the Boost Graph Library, however I have not used C++ before, and with packages available in a language that I am comfortable with, it would be tedious to learn the syntax of a new language.

I then coded an initial version of the algorithm using the exact $\lambda$, $\beta$ and $\alpha$ parameters as originally described. This was done using the Snap Stanford python package. I wrote the code with the intention of varying the approximation parameter $\epsilon$, and seeing the effect that had on an input graph, and the other parameters. However, I quickly ran into a problem.

## 4.2 The Parameters

Recall the values of the parameters used by the algorithm. Let $\epsilon < 1/2$, $\lambda = \frac{\epsilon}{128}$, $\beta = 16\lambda^{-2}log(\frac{1}{\lambda})$, $\alpha = \frac{\epsilon m}{n\beta^2+1}$. With the approximation-parameter $\epsilon < 1/2$, this forces the value of $\lambda$ to be small. This in turn forces $\beta$ to be very large, as $16\lambda^{-2}$ will be large. The value of $\beta$ does not depend on $m$, nor $n$ for an input graph, and is thus fixed depending on the approximation parameter. So we see that for any graph we must have, $\epsilon < \frac{1}{2} \Rightarrow \beta > \approx 2.525 \times 10^6$ and also $\beta(1-\lambda) > \approx 2.515 \times 10^6$. This can be seen in Figure 4.1 which shows the value of the approximation parameter $\epsilon$, against the value of the parameter $\beta$.

Now consider the Squirrel graph from Snap Stanford. We have $m = 198,493$ and $n = 5,201$. Assume that the for loop in Phase I of the algorithm continues looping until an Underfull edge is found in the edge stream. In the case of squirrel, when the algorithm checks $deg_H(u) + deg_H(v) < \beta(1-\lambda)$ for a given edge $(u, v)$ in the edge stream. From what I calculated above, we can re-write this as $deg_H(u) + deg_H(v) < \approx 2.515 \times 10^6$. This becomes a problem. As the number of edges in the graph $m = 198,493$ is much smaller than $\approx 2.515 \times 10^6$, every edge in the edge stream will be added to the subgraph $H$. Additionally, the RemoveOverfullEdges procedure is called after edge insertion, but this does not remove any edges from the subgraph, as we will never have the case where there exists $(u, v) \in$ H such that $deg_H(u) + deg_H(v) > \beta = \approx 2.525 \times 10^6$, purely due to the huge differences between the values of the parameters and $m = 198,493$. This then results in $G = H$ after the final edge in the edge stream has been examined, and so the algorithm never leaves Phase I and breaks.

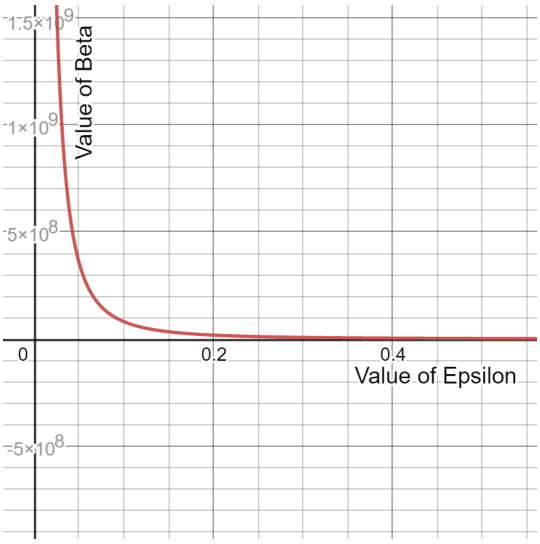



Figure 4.1: Graph showing value of Epsilon vs Beta for the Squirrel data set

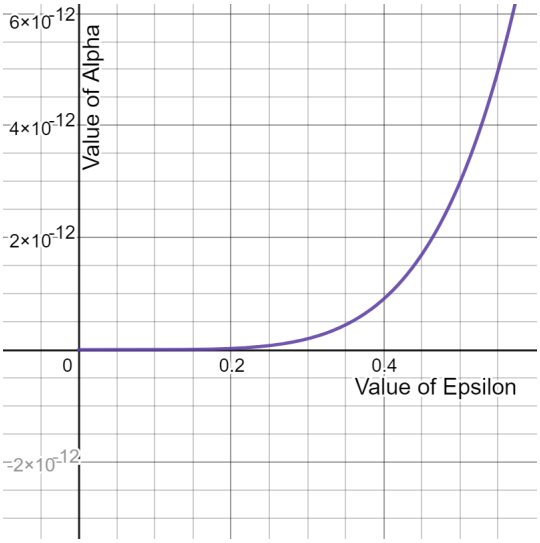



Figure 4.2: Graph showing value of Epsilon vs Alpha for the Squirrel data set

Yes, this specific case is as a result of assuming that the for loop continues looping until an underfull edge is found in the stream, rather than for $\alpha$ iterations as in the original algorithm. However, if we use this true value of $\alpha$, the algorithm breaks before even looking at the first edge of the edge stream. Figure 4.2 shows how the value of $\alpha$ varies against the approximation parameter $\epsilon$. Recall that $\epsilon < 1/2$. The graph shows that in theory, we see that the value of $\alpha$ for all values of $\epsilon$ is less than one. We cannot run a for loop for less than one iteration. Even if we use the ceiling function here, to obtain $\alpha = 1$, we still have the same values for $\beta$ and $\beta(1 - \lambda)$, and we have the same case as above; the algorithm never terminates phase I.

It is now clear. I cannot use the exact values for these parameters as given in theory when implementing this algorithm. I must modify and experiment with them to try and obtain maximum cardinality matchings approximations for a given input graph, close to the $\frac{2}{3}$ - approximate result as proven in the theory.

## 4.3 Development Challenges

### 4.3.1 RemoveOverfullEdges Procedure

One challenge I came across when initially testing the algorithm was the runtime of the RemoveOverfullEdges procedure. Recall the procedure:

**Algorithm 2** The RemoveOverfullEdges Procedure

**procedure** *RemoveOverfullEdges*(H)
  **while** *there exists* $(u, v) \in H$ *such that* $deg_H(u) + deg_H(v) > \beta$ **do**
    Remove $(u, v)$ from H
**end**
**end procedure**

Initially, I developed this function to look at every edge of $H$ in turn and check if $deg_H(u)+deg_H(v) > \beta$. This would ensure that the subgraph $H$ has bounded edge-degree $\beta$, when the function terminates, however, for graphs with tens of thousands of edges, this quickly led to large runtimes. The runtime became infeasible to experiment with; I had to come up with another solution.

The RemoveOverfullEdges procedure is called every time an edge is inserted into the subgraph $H$. This is necessary, as it is essential that by adding the edge to the subgraph, the subgraph still has bounded edge-degree $\beta$. However, the subgraph in phase I, immediately before an edge is added to the subgraph, will always have bounded edge-degree $\beta$. This is because the RemoveOverfullEdges procedure would have been executed after the most recent edge insertion to the subgraph $H$, ensuring that $H$ has bounded edge-degree $\beta$.

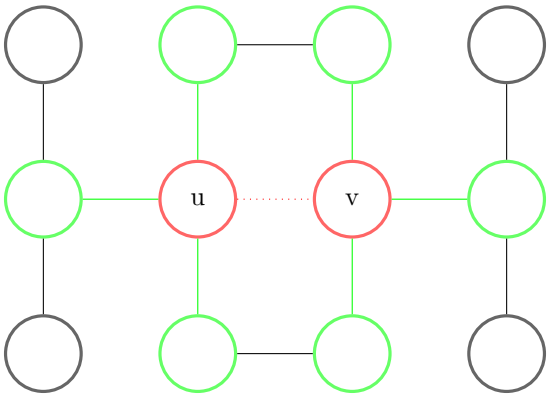



Figure 4.3: An arbitrary graph H showing the edges that could prevent the graph from having bounded edge-degree $\beta$

Figure 4.3 shows an arbitrary graph G, and the impact of adding an edge $(u, v)$ has on the vertices $u$, $v$ and their neighbours. Adding the edge $(u, v)$ to $H$, increases the degree of $u$ and $v$ by 1. Thus, the only edges that need to be checked by the RemoveOverfullEdges function are the edges incident to $u$ or incident to $v$ highlighted in green. The function does not need to check any other edges say $(i, j)$ as the value $deg_H(i) + deg_H(j)$ would not have changed since the last call to this function. Thus this method drastically improves the runtime of this function as we check less edges, whilst maintaining the purpose of the function, ensuring that the subgraph H has bounded edge degree $\beta$.

### 4.3.2 Data Sets

To calculate the Maximum matching approximation that the algorithm computes for a given input graph, I must also calculate the true size of the Maximum Matching for the graph. To do this I used the [[5]] Network X python package. The function max_weight_matching returns a maximal matching for

an input graph. The function computes the maximal matching with a method based on the "blossom" method for finding augmenting paths for a graph invented by Jack Edmonds. [[4]] Note that this function take $O(|V|^3)$ time.

The runtime of this function highlighted another limitation for the experiments. For example, the max_weight_matching function took $\approx 45$ minutes to compute the maximal matching for the "Twitch_Gamers" graph. This is due to the fact that $|V| = 168,114$ for the "Twitch_Gamers" graph. Due to this large runtime and time frame I had for this project, I became limited to experimenting with graphs with $|V|$ being no larger than a few thousand.

# Chapter 5

# Experiments

The main aim of this chapter is to experiment with the Algorithm with real-world data sets. I run tests on the input parameters of the algorithm to push the algorithm to its limits, to understand the algorithm further, and to see if we do indeed achieve $\frac{2}{3}$ - maximum matching approximations on the graphs I look at.

## 5.1 Data Sets Used

The following table presents a list of graphs used and referenced in this experimental chapter. The table also includes values of $|E|$ and $|V|$ for each graph, as well as it's density which is defined as $2|E|/(|V|(|V|-1))$.

| Name | $\|E\|$ | $\|V\|$ | Density |
|---|---|---|---|
| Squirrel | 198,493 | 5,201 | 0.015 |
| Chameleon | 31,421 | 2,277 | 0.012 |
| Crocodile | 170,918 | 11,631 | 0.03 |
| Twitch_Gamers | 6,797,557 | 168,114 | 0.0004 |
| Power | 6594 | 4941 | 0.0005 |

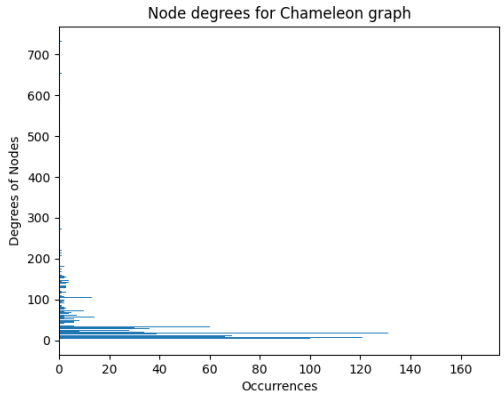
Table 5.1: Graphs used in experiments and throughout the report

### 5.1.1 Descriptions of graphs

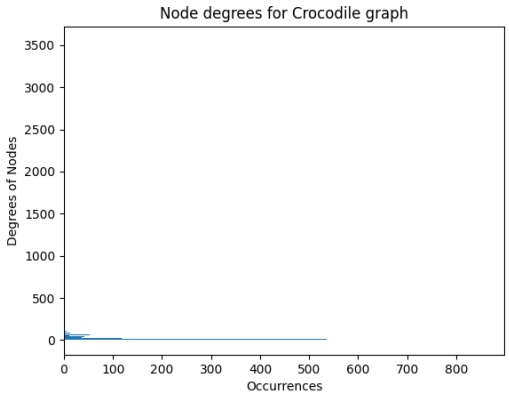
I have chosen a small range of graphs that I use in the experiments and in other parts of the report. The Squirrel, Chameleon and Crocodile graphs are from the Snap Stanford Graph Library [[6]]. These graphs represent data collected from the English Wikipedia in 2018, and represent page to page networks on the topics of squirrels, chameleons and crocodiles. The nodes in these graph represent articles with the edges representing mutual links between these pages. The Twitch Gamers graph is also from the Snap Stanford Graph Library [[6]]. The nodes in the graph represent Twitch users and edges mutual follower relationships between them. The Power graph represents a network representing the topology of the Western United States Power Grid, with the data compiled from [[7]].

### 5.1.2 Degree distributions

Figures 5.1 and 5.2 show the degree distributions for the graphs that I have used during the experiments. These will be helpful when analysing the data collected in the experiments, as we know the subgraph constructed by the algorithm will have bounded edge-degree $\beta$. So, these graphs can help visualise how many edges in the graph will immediately break this property for a set $\beta$. The Chameleon, Crocodile and Squirrel graphs all have nodes with degrees mostly less than 100, but they all have small numbers of nodes that have large degrees in comparison. For example, the Crocodile graph has one node with a degree over 3000. The nodes of the Power graph on the other hand all have degrees less than 20, with a mode of 2.

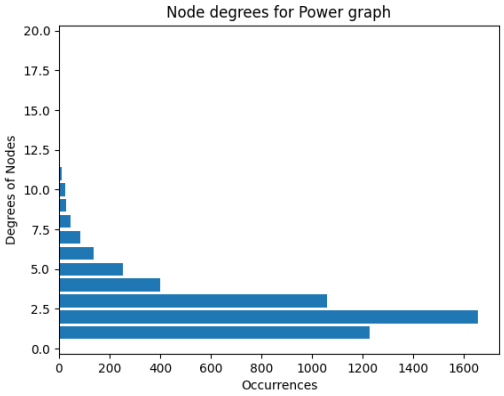



(a) Chameleon Graph

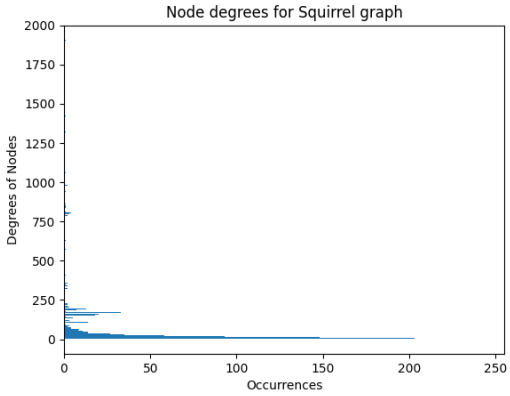



(b) Crocodile Graph

Figure 5.1: Node degree distributions for Chameleon and Crocodile graphs




(a) Power Graph




(b) Squirrel Graph

Figure 5.2: Node degree distributions for Chameleon and Crocodile graphs

There are many different experiments that I could run to test out and push the algorithm to its limits with these data sets. These could include testing how long the algorithm spends in a certain Phase of the algorithm varied against a parameter such as $\beta$, or seeing how often the algorithm actually removes an edge from the subgraph when the RemoveOverfull Edges procedure is called. Both of these experiments would provide further insight into how the algorithm performs, but I am firstly interested in seeing if we can achieve $\frac{2}{3}$ - maximum matching approximations. To see if we can indeed achieve this desired approximation, I would have to see how varying the input parameters affects the output approximation.

## 5.2 The affect of $\beta$ on the approximation

I am particularly interested in first seeing how varying the $\beta$ parameter affects the approximation. This is because out of all the parameters, $\beta$ is used in each phase of the algorithm; determining if an edge is $(G, H, \beta, \lambda)$-underfull in the first phase, ensuring that the subgraph $H$ has bounded edge-degree $\beta$ in the RemoveOverfullEdges procedure, and in Phase II, checking for any edge remaining in the edge stream if the edge is $(G, H, \beta, \lambda)$-underfull. As I am only focused on the effect of the $\beta$ parameter, I will fix the values of $\alpha$, and $\lambda$. Note that the approximation parameter $\epsilon$ only impacts the value of $\epsilon$, and is not used in the algorithm itself.

I have chosen to fix the value of $\alpha$ to be equal to 6 in these initial experiments. Recall that in Phase I, the algorithm looks at $\alpha$ edges in each epoch, and the algorithm moves to Phase II if every edge in the current epoch is not $(G, H, \beta, \lambda)$-underfull. If it is the case where we don't see good enough

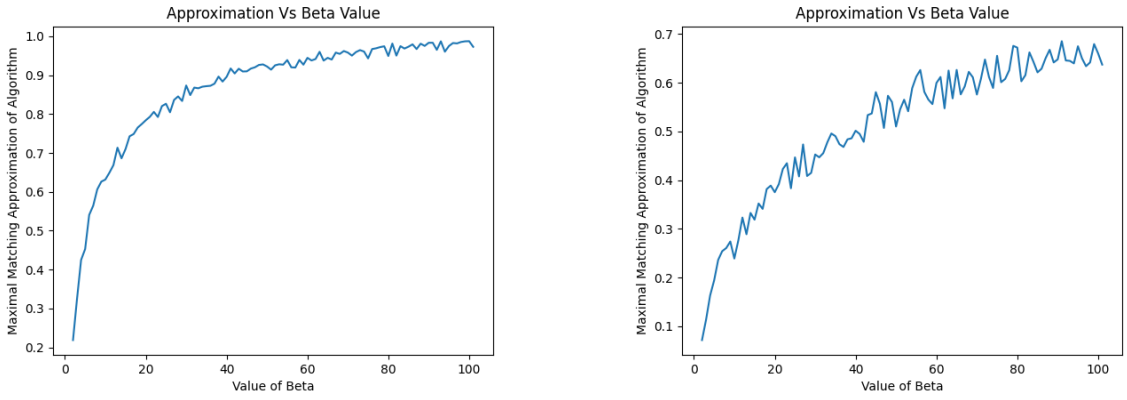
approximations for this value, this can be easily changed, and I will be looking at the affect of varying the $\alpha$ has on the algorithm in a different experiment. As in the theory, the algorithm computes a $\frac{2}{3} - \epsilon$ - approximation, I will aim to keep the value of the $\lambda$ parameter to be as small as possible to reflect this. This in turn then fixes the value of $\beta(1-\lambda)$ which is used in both phases. We have $\beta > \beta(1-\lambda)$, and as $\lambda$ needs to be as small as possible, I have fixed $\beta(1-\lambda)$ to be one less than $\beta$. It is important that these values are as different in practice as they are in theory; an interesting experiment could be seeing the affect of having the same values for these parameters has on the algorithm.

### 5.2.1 Methodology

I conducted this experiment by running four iterations for each $\beta$ value, with each iteration using a randomised edge-stream for the input graph. I would have tested each $\beta$ value more than four times, however, as the algorithm uses the Network X max weight matching method [[5]] to calculate the maximum cardinality matching of the subgraph, and as we saw in 4.3.2, the runtime of this is very large. This would result in the experiment to take a considerable amount of time. Thus, I limited each value to run four times, as it is a good trade off between the time it takes to complete the experiment, whilst still preserving some statistical significance in the data.

As this edge-stream was randomised each time, the subgraph constructed by the algorithm is different, resulting in different approximation values. I then took the mean approximation and used this value to plot the graphs.

### 5.2.2 Results and analysis

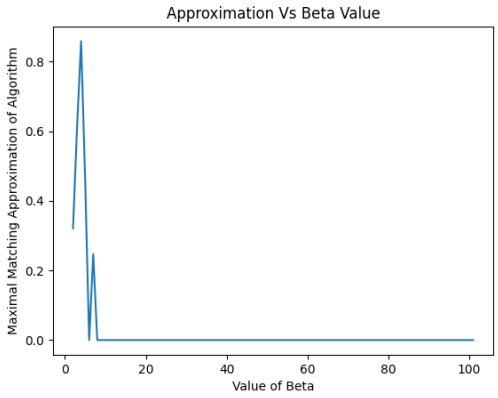



(a) Chameleon Graph

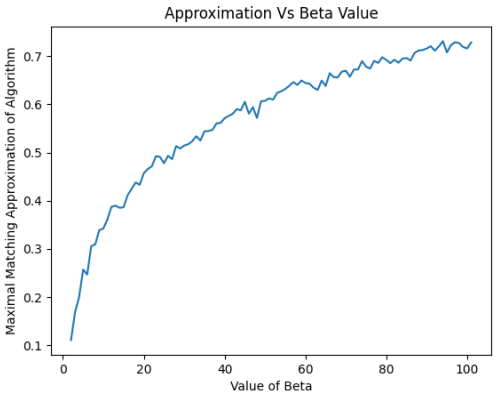
(b) Crocodile Graph

Figure 5.3: Graph showing value of $\beta$ vs the algorithms approximation for the Chameleon and Crocodile data sets

The data in figures 5.3 and 5.4 shows some important results. The graphs shows that there does in fact exist values of $\beta$ that obtain approximations that are not only close to, but exceed the theories $\frac{2}{3}$ - approximation. The results of the Chameleon, Crocodile and Squirrel data sets also seem to fit an almost logarithmic curve. This is to be expected; when experiments are run with small values of $\beta$, it restricts the degrees of the nodes of the subgraph that the algorithm constructs, and naturally, this results in smaller approximation values. One could think that this trend would continue; the greater the value of $\beta$, the greater the approximation. However this is not the case, as observed with the Power data set in figure 5.4. The graph shows we do obtain and exceed the $\frac{2}{3}$ - approximation for small values of $\beta$. In comparison to the Squirrel and Crocodile data sets, the degree distribution of the graph is concentrated towards values much lower than that of the squirrel graph. Recall that the algorithm only exits Phase I after finding an epoch where every edge is not $(G, H, \beta, \lambda)$-underfull. This means that $deg_H(u) + deg_H(v) \geq \beta(1-\lambda)$ for every edge $(u, v)$ in an epoch. And as $\beta(1-\lambda)$ becomes larger than the degrees of most or all nodes in the graph, the algorithm never exits Phase 1, thus the algorithm terminates without ever returning




(a) Power Graph




(b) Squirrel Graph

Figure 5.4: Graph showing value of $\beta$ vs the algorithms approximation for the Power and Squirrel data sets

an approximation. This is one of the outcomes I talked about in section 4.2. Now I could adjust the algorithm to move to Phase II if the input graph leads to a case like this, however this defeats the whole purpose of the algorithm. The subgraph constructed if this were the case would just be entire graph, so the algorithm would have done nothing.

## 5.3 The affect of $\beta$ on the size of the output graph

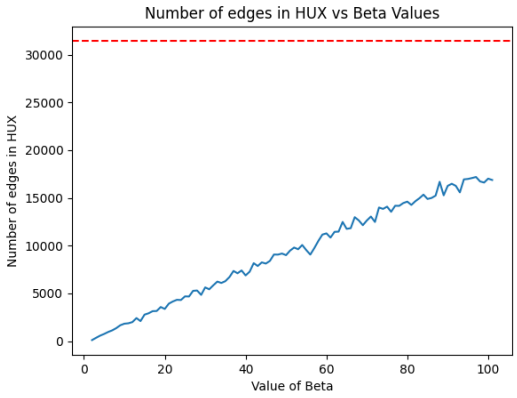
We have seen that for different values of $\beta$, the algorithm does in fact achieve a $\frac{2}{3}$ - maximum cardinality matching for all the data sets that I have looked at. In theory, the algorithm uses $O(nlog(n))$ space to compute the approximation, where $n = |V|$. An interesting experiment to look into this further is to see how changing the $\beta$ parameter impacts the size of the subgraph that the algorithm constructs, and hence what is the ratio of the number of edges in the subgraph compared to the number of edges in the input graph?

### 5.3.1 Methodology

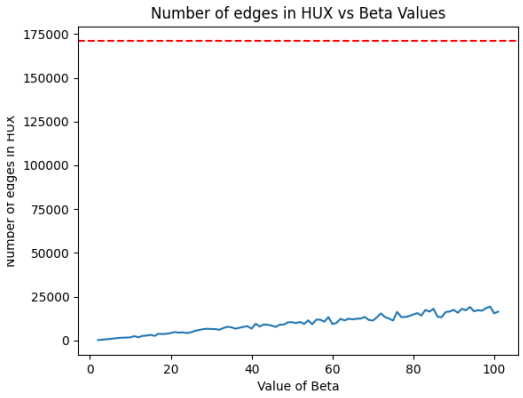
To conduct this experiment, I used a similar method that I used in the previous experiment. I ran four iterations for each $\beta$ value, with each iteration again using a randomised edge-stream for the input graph. Again, I would have tested each $\beta$ value more then four times, but for the same reasons as discussed in 5.2.1 I limited each value to run four times. I modified the output of the algorithm to return the size of the subgraph constructed instead of the size of the maximal cardinality matching of the subgraph, calculated the mean for each $\beta$ value, and plotted the results.

### 5.3.2 Results and analysis

The graphs in figures 5.5 and 5.6 show some interesting results. Note that the dashed red line on each of the graphs indicates the number of edges in the original graph. The blue line shows how the value of the $\beta$ parameter affects the number of edges in $H \cup X$. Recall that $H \cup X$ is the subgraph that the algorithm computes. We see a range of results from the graphs I experimented on. The Chameleon graph shows a linear relation, but the plotted line has a higher gradient compared to the Crocodile and Squirrel graphs. The subgraph that the algorithm constructs in this case is made up of a large proportion of the edges from the entire graph. This is likely due to the node distribution for the Chameleon graph. We saw in figure 5.1 that the most nodes in the Chameleon graph have degree less than 100, which explains why this graph has the largest proportion of edges in the subgraph for large $\beta$ compared to the other graphs where this fact is not the case. But as we saw in figure 5.3, the algorithm does achieve a $\frac{2}{3}$ - approximation for smaller $\beta$ values specifically $\beta$ values of 15 - 20. We see for these values of $\beta$ in this experiment, the algorithm stores less than 5000 edges, or less than 17% of the entire input graph, and

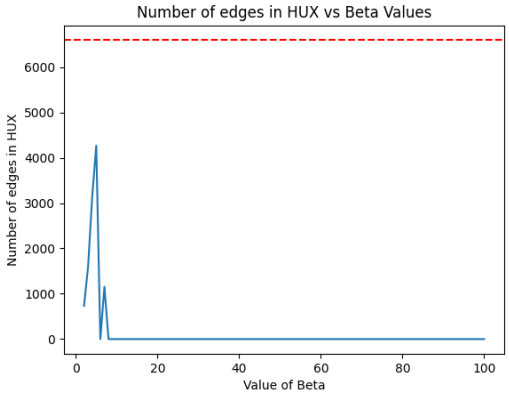



(a) Chameleon Graph

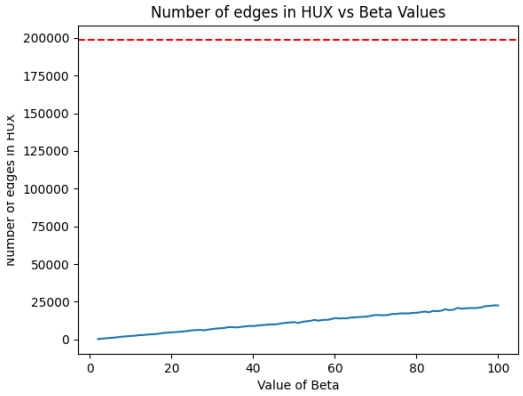



(b) Crocodile Graph

Figure 5.5: Graph showing number of edges in the constructed subgraph vs the value of $\beta$ for the Chameleon and Crocodile data sets




(a) Power Graph




(b) Squirrel Graph

Figure 5.6: Graph showing number of edges in the constructed subgraph vs the value of $\beta$ for the Power and Squirrel data sets

from this, achieves and even succeeds a $\frac{2}{3}$ - maximal cardinality matching approximation. This is a huge result, being able to calculate such a high approximation on only a small proportion of the entire graph.

We see very similar results and conclusions of this experiments from the Crocodile and Squirrel graphs in figures 5.5 and 5.6. We saw in figure 5.3 for the crocodile graph, that the algorithm achieved the $\frac{2}{3}$ - maximal cardinality matching approximation with $\beta$ values around 80. And as we can see in figure 5.5, the algorithm stores less than 25,000 edges, or less than 15% of the entire input graph and still obtains and succeeds the $\frac{2}{3}$ - maximal cardinality matching approximation, as we saw with the Chameleon graph. For the Squirrel graph, we saw in figure 5.4 that the algorithm achieved the $\frac{2}{3}$ - maximal cardinality matching approximation that we are after with $\beta$ values around 60. Again, as we can see in 5.6, the algorithm again stores less than 25,000 edges, or less than 13% of the entire input graph, and still obtains the desired $\frac{2}{3}$ - maximal cardinality matching approximation. So these observations are being seen across multiple data sets.

The results of the Power graph for this experiment tell us a slightly different story. The first thing to note is that the algorithm stores no edges in the subgraph for $\beta$ values greater than 10. This is again due to the node distribution of the Power graph and the affect that this has on the $\beta$ parameter as I already discussed in the previous experiment for this data set in section 5.2.1. This graph is harder to analyse due to the fact that we only get subgraphs for small $\beta$ values, and only ran four trials for each

value of $\beta$. But the results still provide an insight into how the algorithm works for graphs with nodes that have smaller degrees. For $\beta = 8$ we see that the algorithm stores just over 1000 edges. However this result should be discounted. I only ran four iterations for each value of $\beta$ and for this precise value, the algorithm failed to exit Phase I and construct a subgraph in three out of the four trials. When the algorithm fails, a value of 0 is recorded for that trial. However as the algorithm was successful in just one of these trials, when the mean of the trials for this value of $\beta$ was calculated, we obtained a result that is not representative of whats happening and thus this result should be discarded.

## 5.4 The affect of $\alpha$ on the approximation

I have now experimented with the $\beta$ parameter and have observed some promising results, however I fixed the $\alpha$ parameter during these experiments. Recall that the $\alpha$ parameter is used in Phase I of the algorithm. The algorithm looks at epochs of edges from the edge stream of size $\alpha$, and moves on to Phase II of the algorithm if every edge in this epoch is not $(G, H, \beta, \lambda)$-underfull. An interesting experiment would be to see how changing this parameter affects the algorithms approximation.

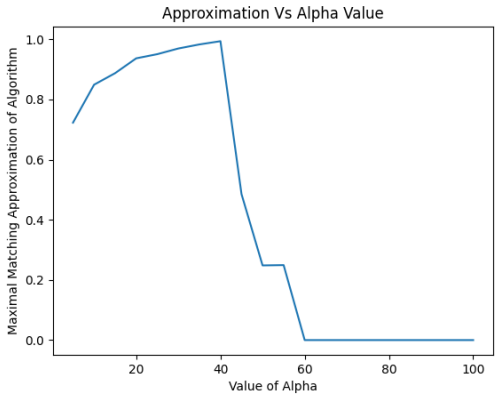
### 5.4.1 Methodology

I conducted this particular experiment slightly differently from the previous experiments. First, as I am varying the $\alpha$ parameter, I need to fix the values of $\beta$ and $\beta(1-\lambda)$ for each data set. I decided to use the $\beta$ values that achieved close to the $\frac{2}{3}$ - maximal matching approximation from the first experiment. I chose these values in particular, as I am interested in seeing what affect increasing the $\alpha$ parameter has on this approximation value. These $\beta$ values are, $\beta = 4$ for the Power data set, $\beta = 60$ for both the Crocodile and Squirrel data sets, and $\beta = 18$ for the Chameleon data sets. As discussed in section 5.2, we need to keep the value of $\beta(1-\lambda)$ as close as possible to the value of $\beta$ as possible. And so this value has been fixed to being one smaller than the value of $\beta$ used for each data sets. As for the experiments of $\beta$, I still ran four iterations for each value of $\alpha$, however, due to the time that these individual trials took, I looked at $\alpha$ values of multiples of 5. Note, as the Power data set is a graph with significantly fewer nodes compared to the other data sets, I ran this data set on every $\alpha$ value between 1 and 100. Again, I calculated the mean approximation for each trial, and plotted the results.

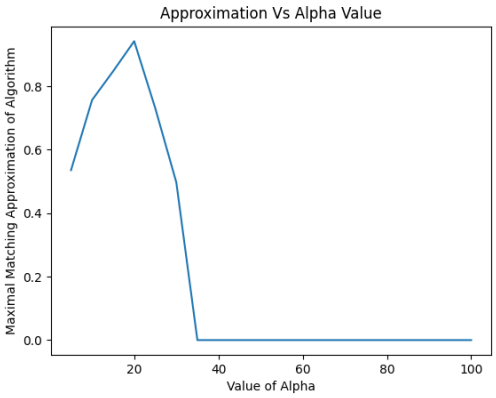
### 5.4.2 Results and analysis

We see similar results for these experiments across all the data sets as shown in figures 5.7 and 5.8. In general, as the value of $\alpha$ increases, the maximal matching approximation of the algorithm approaches 1. The graphs also show that the algorithm constructs a subgraph with maximal matching approximations close to and exceeding $\frac{2}{3}$. This is the case for every data set in the experiment.

There is a point in each graph where we get a sudden decline until the approximation hits 0 for larger values of $\alpha$ in all four data sets. The sudden decrease in the approximation to 0 is again a result of the algorithm failing a number of the four trials, before eventually failing all four trials for larger $\alpha$ values. Similarly to the other experiments, the algorithm in this case would have failed to exit the first phase. The reason why this happens for larger $\alpha$ is due to the increased size of the epoch and the randomness of the graphs edge stream. Larger $\alpha$ values results in larger epochs. We know that the algorithm can move onto Phase II if every edge in the epoch is not $(G, H, \beta, \lambda)$-underfull. But edges incident to vertices with very high degrees in comparison to the other nodes in the graph would likely violate this property and indeed be $(G, H, \beta, \lambda)$-underfull. The algorithm only needs one of the edges in the epoch to be $(G, H, \beta, \lambda)$-underfull for it to move onto the next epoch instead of Phase II after every edge in the epoch has been looked at. The result of having epochs of larger size thus increases the chance of this happening, to the point where every epoch fails and the algorithm finishes never having left the first phase. This explains why we get an approximation value of 0 for every data set with large $\alpha$ values.

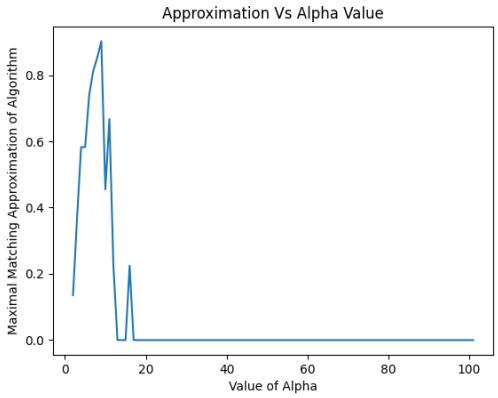



(a) Chameleon Graph

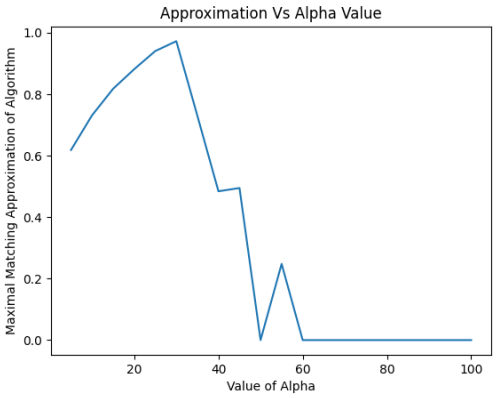



(b) Crocodile Graph

Figure 5.7: Graph showing the value of $\alpha$ vs the algorithms approximation for the Chameleon and Crocodile data sets




(a) Power Graph




(b) Squirrel Graph

Figure 5.8: Graph showing the value of $\alpha$ vs the algorithms approximation for the Power and Squirrel data sets

## 5.5 Other Experiments

There are many other experiments that I could run to test out the algorithm, and see if we can improve upon the results and observations that we have seen this far. However, due to time constraints with this project, I was unable to look into these further; these could be experiments that one could look at for further work. Some details and initial work of potential experiments is described below.

### 5.5.1 Further Experiments on $\alpha$

### 5.5.2 Remove Overfull Edges Procedure

The Remove Overfull Edges procedure is apart of the algorithm that could be experimented with. Recall that this procedure is called after every edge insertion the algorithm makes to the subgraph, and its purpose it to ensure that the subgraph has bounded edge-degree $\beta$, by removing edges that violate this property. A natural experiment that arises from this description is looking into how much time we spend in this procedure and how this changes from adding the first edge to the subgraph to when the algorithm adds the last edge in the subgraph.

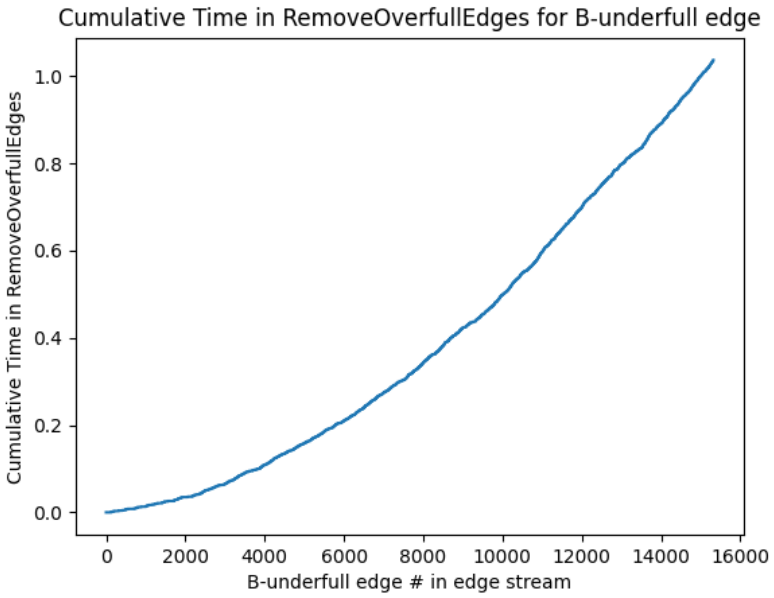



Figure 5.9: Graph showing the cumulative time spent in the RemoveOverfullEdges for a $\beta$ - underfull edge

Figure 5.9 shows an observation for such an experiment on the Squirrel data set. The parameters were fixed in this case with $\beta = 60$ and $\alpha = 6$, as these parameters provided close to the desired $\frac{2}{3}$ - maximum cardinality matching. We can see we have an almost quadratic relation which is of no surprise. As more edges are added to the subgraph, the more time the algorithm spends in the procedure as there are more edges to check. However, this is only a single observation with set parameters, so is difficult to conclude with just one observation, but this provides an insight into what results one could find if future work was done.

Another experiment that could be conducted related to the Remove Overfull Edges procedure is seeing how often we actually remove an edge from the subgraph when this procedure is run. This would depend on the value of $\beta$ being used for the algorithm as the procedure ensures that the subgraph has bounded edge-degree $\beta$. It would be natural to conclude that as the number of edges in the subgraph increases, the number of edges being removed by the procedure also increases as it is more likely that the subgraph does not have bounded edge-degree $\beta$. But to conclude such a result, further work and experiments would have to be done.

# Chapter 6

# Summary and Conclusion

## 6.1 Future Work and Limitations

This project has focused on implementing and experimenting with the algorithm to see what approximations we could obtain whilst varying the parameters, amongst other experiments. There are a few directions that any future work related to this project could take. One direction is to expand the number of real world data sets to test and push the algorithm further. A limitation of this project is the amount of data sets that I used to experiment with on the algorithms. As I only used four graphs, it is difficult to justify the results in general. However, as these graphs were chosen randomly, and the algorithm uses random order edge streams, we still obtain a good idea into how the algorithm does work with real world data sets. If future work was done with more data sets, it could confirm the observations that we have seen in the experiments that I have run.

Another direction future work could take would to focus on developing the algorithm to be as efficient as possible. This approach would take more of an engineering direction than this project. An efficient low level implementation of the algorithm could be developed for a software package in a specific programming language that could aid particular programming projects.

## 6.2 Summary of Key Findings

The main goal of this project was to analyse, and implement the algorithm from [[2]]. The algorithm in theory achieves a $\frac{2}{3}$ - maximal cardinality matching approximation using $O(nlog(n))$ space. I have shown a working implementation on some real world data sets and run experiments on the different parameters to see if we actually achieve anything close to the $\frac{2}{3}$ - maximal cardinality matching approximation that we do in theory. The project has achieved this to some extent, by obtaining approximations that not only achieve, but succeed the desired approximation on the real world data sets that I experimented with.

# Bibliography


[1] Sepehr Assadi, MohammadHossein Bateni, Aaron Bernstein, Vahab Mirrokni, and Cliff Stein. Coresets meet edcs: algorithms for matching and vertex cover on massive graphs. In *Proceedings of the Thirtieth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 1616–1635. SIAM, 2019.

[2] Aaron Bernstein. Improved bound for matching in random-order streams. *arXiv preprint arXiv:2005.00417*, 2020.

[3] Aaron Bernstein and Cliff Stein. Fully dynamic matching in bipartite graphs. In *International Colloquium on Automata, Languages, and Programming*, pages 167–179. Springer, 2015.

[4] Zvi Galil. Efficient algorithms for finding maximum matching in graphs. *ACM Computing Surveys (CSUR)*, 18(1):23–38, 1986.

[5] Aric Hagberg, Pieter Swart, and Daniel S Chult. Exploring network structure, dynamics, and function using networkx. Technical report, Los Alamos National Lab.(LANL), Los Alamos, NM (United States), 2008.

[6] Jure Leskovec and Andrej Krevl. SNAP Datasets: Stanford large network dataset collection. `http://snap.stanford.edu/data`, June 2014.

[7] Duncan J Watts and Steven H Strogatz. Collective dynamics of 'small-world'networks. *nature*, 393(6684):440–442, 1998.