Auction Algorithm for Bipartite Graph Matching Problem

Analysis and Improvement

By

Yu(Ivan) Qin

Department of Computer Science
University of Bristol

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

May 2022

Abstract

`Matching in bipartite graph` is the problem of finding the largest set of edges selected in a way such that no two edges share the same node. While this problem is well-known and is a central problem in graph theory algorithms.

However, the most advanced algorithms for this matching problem cannot handle today's explosive data growth. Recently, a theoretically optimised streaming algorithm, called the auction algorithm, can cope with large amounts of data. Our project aims to research this auction matching algorithm and give a more in-depth analysis and engineering optimisation.

We give worst-case theoretical analyses and validate these analyses through our experiments and visualisations. Based on the original auction algorithm and analyses, we propose several points that can be optimised from an engineering perspective.

Finally, through experiments on 15 different datasets, we compare the following three algorithms in terms of the number of iterations and run-time in practical.

- The current state-of-the-art Edmond's matching algorithm
- The new auction matching algorithm
- The improved auction matching algorithm

Acknowledgements

Upon completing this thesis, I am grateful to those who have offered me encouragement and support during my study in Bristol.

Thank you to my supervisor, Dr Christian Konrad, for providing guidance and feedback throughout this project.

Particular thanks to all the lecturers who have taught me and my personal tutor Dr Steven Ramsay generous support during these four years in Bristol.

I would like to express my most sincere thanks to my friends and my families, who have provided great help and support.

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 Research Degree Programmes and that it has not been submitted for any other academic award. Except where indicated by specific reference in the text, the work is the candidate's own work. Work done in collaboration with, or with the assistance of, others is indicated as such. Any views expressed in the dissertation are those of the author.

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

SIGNED: ....Yu Qin................ DATE: ....04/05/2022......................

Table of Contents

List of Tables

Table **Page**

List of Figures

Figure **Page**

List of Algorithms

Chapter 1

Introduction

Data has played a central role in our daily life, and algorithms have been helping us solve the problem of data explosion. Many Big Data problems require more than the capacity of traditional computing devices. We urgently need to build new algorithms for handling such huge information, especially the algorithm should focus on the efficiency of the space and the time.

One of the algorithms that are well-suited is called data streaming algorithm. As its name implies, it processes its input in streams and uses very limited space while reducing the iteration of viewing the entire graph. The Data Streaming algorithm addressing the fundamental problem is that the RAM of modern computers is much smaller (e.g., a few Gigabytes) than the size of the massive modern dataset (Terabytes or even Petabytes).

In academia, The bipartite graph matching problem has always been the central problem of graph theory algorithms. Our project will focus on this problem since it can migrate to other more complicated problems, such as minimum vertex cover, maximum independent set, maximum flow etc. As the data around us grows, streaming algorithm is becoming increasingly important in large scale graph computing.

On the industrial side, it is actively studied in large companies. For example, social networks like Twitter and Facebook are billions of monthly active users interconnected by trillions of friendships; Takeaway companies like Uber eat matches orders and delivery man. The instantaneous amount of data for this problem is enormous, often exceeding 10,000 requests per second; google's search engine indexes billions of web pages every day and answers more than 100,000 search queries per second on average, which amounts to more than a trillion queries per year.

Use Auction Algorithm to solve matching problem was firstly introduced by Dimitri 30 years ago [[3],[13]]. However, a recent paper [2] introduced a new multi-pass streaming algorithm in 2021, called New Auction Algorithm. The new algorithm is a data streaming algorithm that offers strong theoretical guarantees in space and time. Compared with the origin Auction Algorithm, New Auction Algorithm reduced the number of iterations from $O(\frac{\log\log(1/\epsilon)}{\epsilon^2})$ to

$O(\frac{1}{\epsilon^2})$; also, it reduced the space complexity from $O(n/\epsilon)$ to $O(n)$. Note that ϵ is a decimal number between 0 and 1, so the algorithm is improved from the previous algorithm.

1.1 Objectives and plans

We see an urgent need to research this auction matching algorithm. However, many problems have arisen, such as performance and any further improvement on Auction Algorithm. Hence, we can propose the goal of our research project, for instance:

- Understand, implement the algorithm and give a straightforward way to visualise the matching.
- Research the algorithm, particularly analyses on worst-case graph.
- Find feasible solutions to improve from the engineering perspective.
- Measure algorithm performance compared with the state-of-art algorithm in large real-world datasets.

We approach the Bipartite matching problem from algorithm engineering perspectives. With analyses from worst-case graph, we will be able to design an improved auction matching algorithm. In practice, the improved algorithm should produce better solutions than the New Auction Algorithm.

In Chapter 2, we formally introduce the Greedy algorithm and definitions relating to matching problems. We also mention the state-of-the-art (Edmond's) matching algorithm, as it will be one of the benchmarks for comparison in the experimental section.

We detail and explain the Auction Algorithm. The overview analysis of the algorithm will be shown in Chapter 3. More specifically, we will characterise the time, space, and iteration complexity and the proof of correctness of the algorithm.

In Chapter 4, we present some of the challenges we faced in creating an efficient C++ implementation of both the origin matching algorithm and our improved matching algorithm. In addition, some methods of pre-processing the data and visualisation are also presented in this Chapter.

Chapter 5 contains analyses of the worst-case graph. We will analyse the properties of algorithm in the beginning rounds of iterations. Based on analyses, we then propose optimisation solutions in next Chapter.

Chapter 6 will compare the performance of our optimised Auction Algorithm, Auction Algorithm, and build-in (Edmond's) algorithm across 15 different data sets. In addition, we will show test results in practice to discuss the advantages and disadvantages of optimised algorithm.

Lastly, the implementation code and visualisation of the Auction Algorithm are available [here]. In the appendix, we show results of the algorithm applied to two different graphs, as well as the two corresponding public video links.

Chapter 2

Preliminaries

Definition 2.1 (Matching)**.** A *matching* $M \in E$ in a graph $G = (V, E)$ is a subset of vertex-disjoint edges, i.e. for every $v \in V : |\{ab \in M : a = v \text{ or } b = v\}| \leq 1$.

It is *perfect* if every vertex is contained in matching M.

It is *maximum* if matching achieve its largest size.

Note that: We only focus on the unweighted graph.

Definition 2.2 (Bipartite)**.** A graph $G = (V, E)$ is *bipartite* if V can be partitioned into disjoint sets A and B which contain no edges.

Here is an example of bipartite matching. The red lines show the Maximum Matching. In the Auction Algorithm, the nodes in Set A also called bidders, and the nodes in Set B are called items.

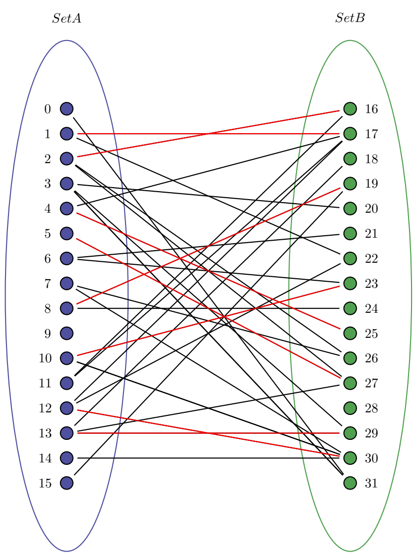


Figure 2.1: An example of bipartite matching

Lemma 2.1. *A graph is bipartite if and only if it has no odd-length cycle.*

Definition 2.3 (Approximation function)**.** Let M^* be a Maximum Matching and let M be an arbitrary matching in G. Then, for $0 \leq c \leq 1$, M is a *c-approximate* matching if:

$$|M| \geq c \cdot |M^*|$$

Greedy Algorithm [Algorithm 1] plays an important role in matching, since it is the best one-pass streaming algorithm known for Maximum Matching. There are many improvement works based on this algorithm in the matching problem. Further, this algorithm constitutes a semi-streaming algorithm with an approximation factor of 1/2.

Algorithm 1: Greedy Algorithm for matching

```
Data: $G$, Graph, a set of edges
Result: $M$, Matching, a set of edges
$M \leftarrow \varnothing$;
while there has an upcoming edge $E = \{u, v\} \in G$. do
    if $u$ and $v$ have not yet been matched then
        $M = M + \{u, v\}$;
    end
    Remove the edge $E$ from $G$;
    Find next edge $E$;
end
```

Definition 2.4 (Graph Stream)**.** A *Graph stream* is an input of edges in sequence, $e_{i_1}, e_{i_2}, ..., e_{i_m}$, where $e_{i_j} \in E$ and $i_1, i_2, ..., i_m$ is an arbitrary permutation of $[m] = \{1, 2, ..., m\}$.

Definition 2.5 (Semi-Streaming Algorithms)**.** A *Semi-Streaming Graph Algorithms* computes over a graph stream using less than $O(n \cdot ploy \log(n))$ bits of space.

Through our research, we now know that multi-pass algorithms can improve greedy algorithms [9] [8]. For example, 2-pass algorithm was proved by Konrad in 2018. It increases the approximation ratio from 0.5 to $2 - \sqrt{2} \approx 0.5857$; 3-pass achieved the ratio to 0.6067.

Definition 2.6 (Augmenting path)**.** Given a matching M in a bipartite graph G, an *augmenting path* for M is a path $P = v_0 \dots v_k$ such that:

- $\{v_i, v_{i+1}\} \in M$ for all odd i;
- $\{v_i, v_{i+1}\} \notin M$ for all even i;
- $v_0, v_k \notin \bigcup_{e \in M} e$.

In C++, the boost graph library[14] contains the state-of-art algorithm called Edmond's maximum cardinality matching, introduced by Hopcroft and Karp [7]. We will also call it built-in algorithm in later chapters. The built-in algorithm's general idea is to use greedy (breadth-first search) algorithm to find the augmenting path. An implementation by Micali and Vazirani [12] has running time to $O(\sqrt{|V|}|E|)$. We do not expand on this algorithm here, as it is only a base benchmark. See the reference for further details.

Algorithm 2: Edmond's Algorithm for matching

```
Data: $G$, Graph, a set of edges
Result: $M$, Matching, a set of edges
while $G$ is not empty do
    pick $t$ in $G$;
    queue.push($t$);
    $M \leftarrow \varnothing$;
    $M$.add($t$);
    while queue is not empty do
        $v \leftarrow$ queue.pop;
        for Every neighbours $w$ of $v$ do
            if $w$ not in $M$ and $w$ matched then
                $M$.add($w$);
                $M$.add(mate($w$));
                queue.push(mate($w$));
            else if $w$ in $M$ and even-length cycle detected then
                continue ;
            else if $w$ in $M$ and odd-length cycle detected then
                contract cycle ;
            else if $w$ in $G$ then
                expand all contracted nodes ;
                reconstruct augmenting path ;
                invert augmenting path;
        end
    end
end
```

Finally, we define the performance of a graph streaming algorithm in this project.

Lemma 2.2. *The Performance of algorithm is measured by its space usage, number of iterations complexity.*

2.1 Summary of Chapter

This chapter introduces the preliminaries associated with the auction matching algorithm, and we also present the best current matching algorithms.

The definitions and theories in this chapter are sourced from [[11], [6], [4], [12]]. For reasons of space, we will not discuss other detailed definitions here, e.g. O notation, but if necessary, check the reference.

Chapter 3

Explanation of Algorithm

Auction algorithms are a worth-thinking method of bipartite matching algorithms that interpret the challenge as choosing one item in an auction that maximizes happiness and assigning it to bidders. It was firstly introduced by Demange, etc. [5] in 1986 in the area of economics. After that, more auction algorithms were studied in area of optimization then [[13],[1]].

In 2021, Sepehr Assadi and S. Cliff Liu improved the previous best auction matching algorithm, reduced space complexity and the number of iterations [2]. It states that suppose that in each iteration of the new auction algorithm, select a maximal match by a greedy algorithm in the subgraph consisting of unassigned bidders and all of their lowest-priced items; after that, this algorithm stops it passes with $O(1/\epsilon^2)$ iterations and $(1-\epsilon)$ approximation. More details show below. [Algorithm 3]

3.1 Explanation of Auction Algorithm

- The input data is a set of vertices from Graph G, The input data is also divided into set A (bidders) and set B (items).
- In line 3, k is the number of iteration, it requires $\frac{2}{\epsilon^2}$ iterations.
- In line 4, D is the union of bidders' demanding set, initially, D and D_{bidder} set to $\varnothing$.
- In line 7, D_{bidder} is the demanding set of *bidder*, contains bidder's every neighbour items with minimal price and $price[item] < 1$.
- In line 8, **any maximum matching** can be derived by the greedy algorithm, 2-pass algorithm, or any other optimised algorithm.
- In line 11, check if there is an edge in M before. If the edge exists, the previous owner is the bidder in that edge connected to *item*.

Algorithm 3: New Auction Algorithm for bipartite matching

```
   Data: $G = (A \sqcap B, E)$
   Result: $M$, Matching, a set of edges
1  For each $bidder \in A$, let $a[bidder] = \perp$;
2  For each $item \in B$, let $price[item] = 0$;
3  for $k \leftarrow 1$ to $\lceil \frac{2}{\epsilon^2} \rceil$ do
4      $D = \varnothing = \bigcup D_{bidder}$;
5      for $bidder$ in $bidders$ do
6          if $a[bidder] = \perp$ then
7              $D_{bidder} = \arg\min_{item \in N(bidder), price_{item} < 1} P_j$;
8      $M_k$ be any maximal matching of $D$;
9      for $\{bidder, item\} \in M_k$ do
10         $a[bidder] = item$;
11         if $item$ has previous owner then
12             $bidder' =$ previous owner of $item$;
13             $a[bidder'] = \perp$;
14             Remove $\{bidder', item\}$ from $M$;
15         $price[item] = price[item] + \epsilon$;
16         Add $\{bidder, item\}$ in $M$;
17 Return the $M$;
```

3.2 Example of Algorithm

See Appendixes [A]. Here is a visualization of this algorithm, showing how the algorithm works on 16 nodes, 64 edges, random graph.

3.3 Theoretical Analysis on General Bipartite Graph

In this section, we study the analyses of New Auction Algorithm working on the general bipartite graph [2]. There is a latter section that includes the analysis of New Auction Algorithm working on Semi-complete (worst-case) graph. The theoretical analysis of this algorithm will be discussed in four directions, namely Space Complexity, Number of Iterations, Dependency of ϵ, and Proof of correctness. We will start with some definitions and observations.

In this section, we study the analyses of New Auction Algorithm working on general bipartite graph [2]. The theoretical analysis of this algorithm will be discussed in four directions, namely Space Complexity, Number of Iterations, Dependency of ϵ, Proof of correctness. We will start with some definitions and observations.

Definition 3.1 (value of a bidder)**.** For any $bidder \in bidders$, define the value of $bidder$ for items as function $v_{bidder} : R \to 0$ or 1 where $v_{bidder}(i) = 1$, if $i \in N(bidder)$ and $v_{bidder}(i) = 0$ otherwise.

Definition 3.2 (utility of a bidder)**.** Define the utility of a *bidder* when given item a_{bidder} as $u_{bidder} = v_{bidder}(a_{bidder}) - price_{a_{bidder}}$, namely, the value of allocated item a_{bidder} for the bidder, minus the price that *bidder* has to pay for the item; the utility of an unallocated bidder *bidder* is $u_{bidder} = 0$

Definition 3.3 ($\epsilon - happy$)**.** We define a $bidder \in bidders$ is $\epsilon - happy$, if and only if $u_{bidder} \geq v_{bidder}(i) - price_i - \epsilon$ for all $i \in N(bidder)$, i.e., changing the allocation of *bidder* to any other item does not increase the utility of *bidder* by more than ϵ.

As we defined, in each pass of the auction algorithm, both allocated bidders and unallocated bidders with empty demanding sets are $\epsilon - happy$. Each allocated bidder picks the minimum price item in its neighbourhood and increases its price only by ϵ. Thus, the price of items is monotonically increasing. Every item in the neighbourhoods of an unallocated bidder with an empty demanding set has price 1. Finally, picking that item cannot increase the bidder's utility by more than ϵ. We use $\mu(G)$ to denote the size of maximum matching G.

3.3.1 Space Complexity

The previous best algorithm [1] achieved the space of $O(n/\epsilon)$. This algorithm has space complexity $O(n)$, only needs to process and store the following properties, which clearly show this algorithm has space of $O(n)$.

- The *prices* for each item is $O(n)$.
- The allocation of bidder (in array of a) is $O(n)$.
- Demanding set is $O(n)$.
- Using the greedy algorithm for finding the maximum matching in $O(n)$ and One-Pass.

3.3.2 Number of Iterations

From Algorithm 3, it clearly shows it has $\frac{2}{\epsilon^2}$ iteration, which is $O(\frac{1}{\epsilon^2})$.

The method of proof considers the sum of price of all items, and the sum of price of demanding set of bidder. Thus, we define

$$\Phi_{\text{bidders}} := \sum_{bidder \in Maximum\ Matching} \min_{j \in N(bidder)} price_j, \quad \Phi_{\text{items}} := \sum_{j \in items} price_j.$$

Since the price is between 0 and 1, our value of sum functions will both be larger than 0 and smaller than $\mu(G)$. In each iteration, the new matching found will update price and allocation; every new matching will increase ϵ to the functions.

Consider any iteration of the auction wherein at least $\epsilon \times \mu(G)$ bidders in Maximum Matching are not $\epsilon - happy$. As such, $\Phi_{\text{bidders}} + \Phi_{\text{items}}$ increases by $\epsilon^2 \times \mu(G)$ in this iteration. The maximum value of two function is $2 \times \mu(G)$, thus, we have only $\frac{2}{\epsilon^2}$ such iterations.

However, we will show that this number is reduced in the later chapters, but we can provide the following lemma.

Lemma 3.1. *There exists an iteration that at least $(1-\epsilon) \times \mu(G)$ bidders in maximum matching are $\epsilon - happy$.*

3.3.3 Dependency of Epsilon

The space complexity of our algorithm has no dependence on ϵ. While the ϵ does impact on two things, first is the percentage of answers, as well as the number of iteration. The former impact satisfies feature of semi-streaming algorithm. The effect of the latter on number of iterations is due to ϵ controlling the increased price. The smaller the ϵ we set, the more matching we can get, but the more iterations we need to process.

General speaking, the dependency of ϵ is low, it is also a upper/lower bound for these properties, for example to get $1 - \epsilon$ of maximum matching, we can choose a slightly bigger ϵ, in next few chapters we will show more information about these, based on our experiments and analyses.

3.3.4 Proof of Correctness

Consider if bidders in Maximum Matching become $\epsilon - happy$ in the auction, then the final matching M achieves a $(1 - \epsilon)$-approximation. We provide another lemma to show the proof of correctness. By the previous definition of $\epsilon - happy$, and summing all over the utilities of $\epsilon - happy$ bidders.

Note that, the o function mentioned below represents the allocation to the certain *bidder*.

We have the following equation:

$$\begin{aligned}
\sum_{bidder \in \epsilon-happy\ bidders} u_{bidder} &\geq \sum_{bidder \in \epsilon-happy \cap Maximum\ Matching} u_{bidder} \\
&\geq \sum_{bidder \in \epsilon-happy \cap Maximum\ Matching} (v_{bidder}(o(bidder)) - price_{o(bidder)} - \epsilon) \\
&\geq (1 - 2\epsilon) \cdot \mu(G) - \sum_{bidder \in \epsilon-happy \cap Maximum\ Matching} (price_{o(bidder)})
\end{aligned}$$

Meanwhile, we also have following equation on the other hand:

$$\begin{aligned}\sum_{bidder \in \epsilon-happy} u_{bidder} &= \sum_{bidder \in \epsilon-happy \wedge a_{bidder} \neq \perp} \left(v_{bidder}\left(a_{bidder}\right) - price_{a_{bidder}}\right) \\ &\leq |Maximum\ Matching| - \sum_{bidder \in \epsilon-happy \wedge a_{bidder} \neq \perp} price_{a_{bidder}} \\ &= |Maximum\ Matching| - \sum_{j \in items} price_j\end{aligned}$$

By combining the previous two equations, we can get the number of the maximum matching:

$$\begin{aligned}|Maximum\ Matching| &\geq (1-2\epsilon)\cdot\mu(G) + \left(\sum_{j \in items} price_j - \sum_{bidder \in \epsilon-happy \cap Maximum\ Matching} price_{o(bidder)}\right) \\ &\geq (1-2\epsilon)\cdot\mu(G)\end{aligned}$$

Lemma 3.2. *Suppose at the end of iteration of the auction algorithm, $(1-\epsilon)\cdot\mu(G)$ bidders in maximum matching are $\epsilon - happy$. Then, the final matching M has size at least $(1-2\epsilon)\cdot\mu(G)$.*

3.4 Summary of Chapter

In this chapter, we introduce the auction algorithm and give an example. In addition, we learn several properties of the algorithm, such as space complexity and epsilon dependence.

Also, based on the two lemmas, we show proof of correctness. The new auction algorithm can find a $(1-\epsilon)$-approximate matching in $O(\frac{1}{n^2})$ iterations.

However, most of our explanations are based on the ordinary bipartite graph, and in the subsequent chapter, we give a more in-depth discussion of the worst-case graphs.

Chapter 4

Implementation Details

In the last chapter, we introduced the algorithm and its theory. To have a better understanding of this algorithm, we programmed this algorithm. Our implementation environment is `C++`.

We used the `Boost Graph Library` [14] for the baseline matching algorithm and base graph structure (`adjacency-list`). Our real-world dataset originated from the `Snap dataset` [10]. We created a preprocessing script that changed the node ID to be ordered.

Moreover, some optimisation options are also proposed here. We only include essential code here, to see the complete code, please check the [link here].

4.1 Preparation of the dataset

Most datasets are sorted by the ID of two nodes (starting from 0) in the text file. However, there are some datasets that are not; for example, ego-Gplus is a Google plus social network with 107,614 nodes and 13,673,453 edges.

A more critical issue is that the google plus dataset store node IDs as 21 digits, which would be too large for integer (even long long structure) wasteful when processing the data.

Thus, we have implemented a `Python` script that changes the order of node IDs in the dataset by simply iterating graph and using hash table.

```
dic = dict()
while readline():
    a,b = line.split()
    if a not in dic:
        dic[a] = num
    if b not in dic:
        dic[b] = num
    num += 1
    writeline()
```

Also, both the real-world dataset from the SNAP library and most random generated datasets are not bipartite graphs. After creating graphs, we applied the deletion process to make them bipartite graphs. The source code is available following. We created a list called *bipartite* to store whether it is in set A or set B, then disconnected the edge from the same set.

```
//Set the bipartite for every nodes.
for ( node_ID in every nodes ) {
    bipartite[node_ID] = int(rand()%2);
}
//Delete the edge from same bipartite.
for ( edge_ID in every edges) {
    if (bipartite[edge_ID.start] == bipartite[edge_ID.end] ) {
        delete_edge()
    }
}
```

4.2 Data Structure

During completing the code, the optimization of the data structure is one of the critical steps that affect the algorithm's performance. More precisely, we need to pay attention to how to find the corresponding allocation and the price quickly.

In designing this structure, three versions have been completed. The first is the most basic array structure. The running time to find the mapped value is exponential time with iteration. Even though we added the early interruption, it still behaves slowly.

Thus, it is necessary to improve the original data structure. Our first improvement was changing the array to Hash Map. Hash Map can theoretically achieve the $O(1)$ in time complexity. In `C++`, `Map` is a sorted associative container that contains key-value pairs with unique keys. It is able to store and combine a pair of data. We set allocation and price in two maps to store their values with their node IDs.

In the final version of mapping value, we inserted two maps into our graph. This process reduces the space for storage.

In addition, it needs to deal with big data as a streaming algorithm. In order to provide enough space for data, we have expanded all data types. For example, increasing 4 bytes *signed integer* types to 8 bytes unsigned *long long int* types. Now, we are able to process bipartite graphs with a size less than 1.8×10^{19} if we have enough time. For a larger graph, we can break the number into *string* structure, but we do not discuss it here, as it is not relevant to our project.

While processing the graph, we often need two iterators, edge iterator and vertex iterator, respectively. Both are built from the forward (bidirectional) iterator, and we use a for loop with two iterators to control the ending. We use the following code to iterate the graph.

```
//Iterate the vertices
graph_traits<vector_graph_t >:: vertex_iterator vi,vi_end;
for (boost::tie(vi, vi_end) = vertices(Graph); vi != vi_end; ++vi)

//Iterate the edges
graph_traits<vector_graph_t >::edge_iterator ei, ei_end;
for (boost::tie(ei, ei_end) = edges(Graph); ei != ei_end; ++ei){
    long long s = source(*ei, Graph);
    long long t = target(*ei, Graph);  \\s, t are two node_ID of edge ei
}
```

4.3 Iteration Number and Epsilon

Finding the earlier break is another significant improvement. The algorithm will repeat $\frac{2}{\epsilon^2}$ times. We can stop the algorithm directly when the demanding set is empty from an engineering point of view. We can apply a terminate step to this algorithm because when the algorithm has no demanding set, we cannot find any new matches, then the image will not have any updates, and there is no demanding set in the next round.

The ϵ is a key variable to control the number of iterations as well as the price of increase. In this auction algorithm, the number of matching grows very slowly, especially in later iterations. Thus, we have arranged an experiment using adjustable epsilon. The primary strategy is dividing the epsilon into four parameters as follows.

- *Based epsilon* gives a small enough epsilon which can get every answer, say $\frac{1}{n/2}$.
- *Iteration threshold* is a parameter that controls increment frequency, here we set it to 3. But if there was increment, we set Price back to the based epsilon. For example, if the threshold is reached without three iterations growing in our code, the procedure for aggressive epsilon is triggered afterwards.
- *Aggressive factor* indicates the increase method and the increase factor. To be more specific, if this aggressive factor smaller than 1, add it to the current *Price epsilon*; otherwise, multiple it to current *Price epsilon.*
- *Price epsilon* controls the increase of the price of the item each time.

```
price_epsilon = based_epsilon;
for (every iteration){
    if (price_epsilon < 1 and aggressive_factor <1){
        if (no changes in last 3 iteration){
            price_epsilon += aggressive_factor;
        }
        else{
            price_epsilon = based_epsilon;
        }
    }
    else if (price_epsilon < 1 and aggressive_factor >=1){
        if (no changes in last 3 iteration){
            price_epsilon *= aggressive_factor;
        }
        else{
            price_epsilon = based_epsilon;
        }
    }
    ...
    ...
    p[item] += price_epsilon;
    ...
    ...
}
```

The code above shows how we program the aggressive epsilon, some of them using multiplication and others using more aggressive addition. Overall, this method combined with a suitable parameter improves the program's running time, though the effect is not apparent and it has a crucial limit. More specific results are shown in Chapter 6.

4.4 Maximal Matching Improvement

In Chapter 3, we introduced the new auction algorithm, and in its line 8, we select the M_k from the demanding set as the maximal matching. We need to consider how to choose a suitable maximal matching algorithm. We firstly apply the edge first greedy algorithm. More specific, iterating every edge if neither point has been visited before. We add the edge into the $final\_matching$, and mark both vertices as visited.

```
bool visited[size of graph];
fill(visited[every element], false);
for (every edge(ei) in M_k){  //Iteration of edges
    long long s = source(*ei, M_k);
    long long t = target(*ei, M_k);
    if (visited[s]==false and visited[t]==false and s!=t){
      visited[s]=true;
      visited[t]=true;
      add_edge(s,t, final_matching);
    }
  }
}
```

Furthermore, when dealing with the worst case, we simulate selecting the reversed order of edge, which means we aim to find the least number of maximum matching each iteration. This operation does not affect the correctness of the algorithm, but it will influence the run-time in practice.

```
bool visited[size of graph];
fill(visited[every element], false);
for (every vertice(vi) in M_k){
    long long s = *vi;
    vector<long long> vector_t;
    vector_t.clear();
    for (every out_edges(ei) from vertice(vi) in M_k){
        long long t = target(*ei, M_k);
        vector_t.push_back(t);
    }
    for (visit every t from vector_t in reversed order){}
        if (visited[s]==false and visited[t]==false and s!=t){
          visited[s]=true;
          visited[t]=true;
          add_edge(s,t, final_matching);
        }
    }
  }
}
```

4.5 Visualization

We use `Gnuplot` to draw graphs, and we import this extension package into our program to directly display and analyze the performance of our algorithm.

We also imported the LaTeX `Tikzpicture` package to illustrate the progress of the algorithm in each iteration. Finally, we use `FFmpeg` to convert `PDF` files to video and publish the result on the public website.

4.6 Summary of Chapter

This chapter shows the challenges we encountered when writing the code and presents many feasible methods to improve the code. We will experiment with these improvements and discuss them in the last chapter.

Chapter 5

Analysis on Semi-Complete Graphs

Apart from the research already done in Chapter 3, random graphs and real-world data are not suitable for theoretical analysis. We also performed a worst-case analysis of the algorithm in this Chapter.

Definition 5.1 (Semi-Complete graph)**.** Semi-Complete graph also named the half graph. A bipartite graph (X, Y, E) is a Semi-Complete graph (half graph) if its vertices can be numbered $x_0, .., x_{n-1}$ and $y_0, .., y_{n-1}$ such that x_i is adjacent to y_j iff $i \leq j$. see the following picture of an example with 32 nodes.

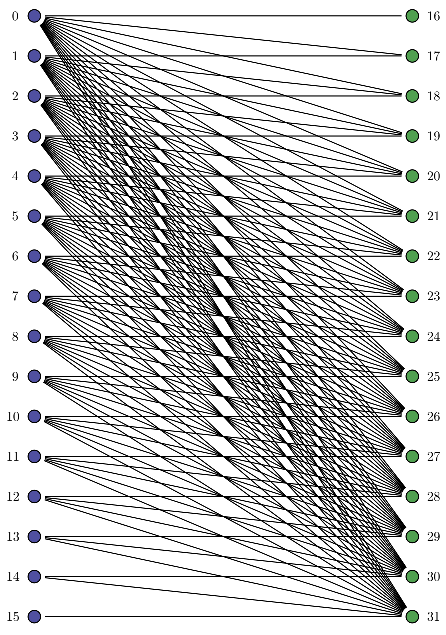


Figure 5.1: An example of 32 nodes Semi-Complete graph

Semi-Complete graphs is the most difficult scenario for Auction Algorithm. Because, thinking about the worst case of the greedy algorithm, it will increase the size of the current image by half each time, while the Semi-Complete graph can only increase the size of the current image by half of the current image size each time.

The Semi-Complete graph has a unique solution, which is the perfect matching. It is easy to check with induction: every node x_n connects to the node y_n, and the remaining vertices form another half graph. Moreover, every bipartite graph with a perfect matching is a subgraph of a Semi-Complete graph.

5.1 Analysis in First Three Iteration

The Auction Algorithm has a particular similar pattern for every Semi-Complete graph. To avoid the rounding problem, considering a large number of nodes equals the power of two. The analyses of these iterations illustrates how the algorithm works. We start with the beginning iteration, as the example of n nodes below. To better understand the algorithm, we also present images on 32 nodes Semi-Complete graph.

- **First Iteration** In the first iteration, the state of matching has following properties:

  1. The number of matching in first iteration is $\frac{n}{4}$, which is also **50%** of maximum matching.
  2. Node i matches to $n-1-i$, where $i = 0, 1, 2, .., \frac{n}{4} - 1$.
  3. Price of every matched item are ϵ, rest unmatched item are 0. The sum of price is $\frac{n\epsilon}{4}$.

- **Second Iteration** In the second iteration, the state of matching has following properties:

  1. The number of matching in second iteration is $\frac{n}{4}$, which is also **50%** of maximum matching. This number has not increased from last iteration.
  2. From the last iteration, the nodes in set A from $\frac{n}{4}$ to $\frac{n}{2}$ are unallocated bidders. By searching their demanding set, minimum price item and using the greedy algorithm, the maximum matching is $\frac{n}{8}$ edges, they are nodes i in set A connected to $n+\frac{n}{4}-1-i$ in set B, where $i = \frac{n}{4}, \frac{n}{4}+1, ..., \frac{n}{4}+\frac{n}{8}-1$.
  3. Edges i to $n-1-i$ will disconnected, where $i = 0, 1, 2, .., \frac{n}{8} - 1$. In the other words, half of matching from last iteration will be disconnected.
  4. There are three types of price: 0 for nodes $\frac{n}{2}$ to $\frac{n}{2}+\frac{n}{4}-1$; ϵ for nodes $\frac{n}{2}+\frac{n}{4}$ to $\frac{n}{2}+\frac{n}{4}+\frac{n}{8}-1$; 2ϵ for nodes $\frac{n}{2}+\frac{n}{4}+\frac{n}{8}$ to $n-1$. The sum of price is $\frac{3n\epsilon}{8}$.

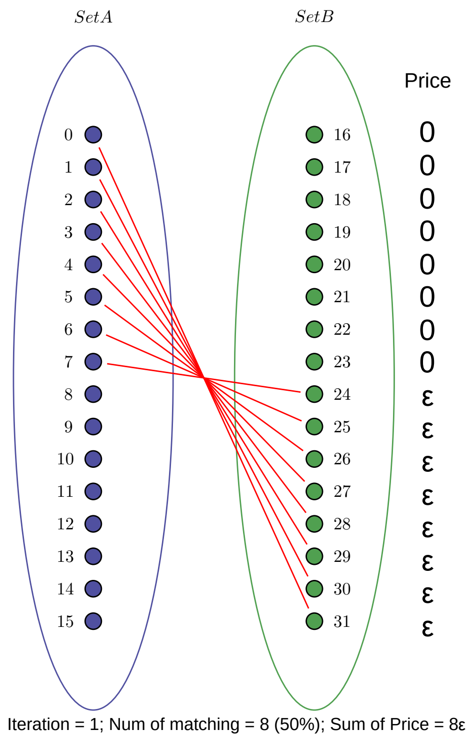


Figure 5.2: Matching state and price in the first iteration

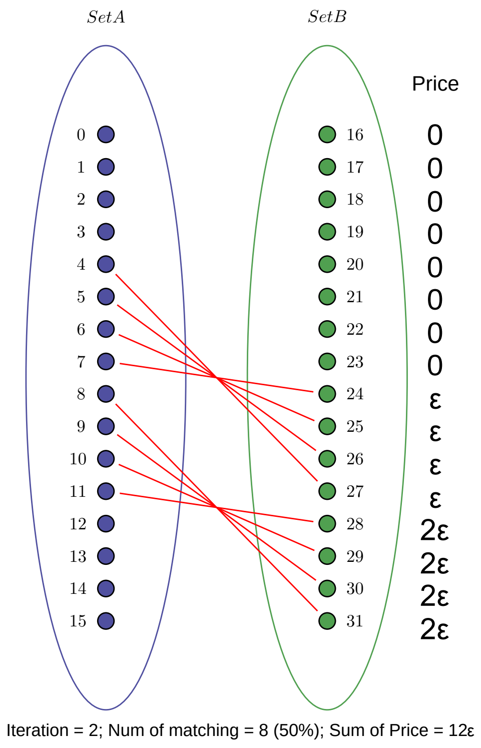


Figure 5.3: Matching state and price in the second iteration

- **Third Iteration:** In the third iteration, the state of the matching has following properties:

  1. The number of matching in third iteration is $\frac{n}{4} + \frac{n}{8}$, which is also **75%** of maximum matching.
  2. From the last iteration, the nodes in set A from 0 to $\frac{n}{8} - 1$ and from $\frac{n}{4} + \frac{n}{8}$ to $\frac{n}{2} - 1$ are unallocated bidders.

     By greedy algorithm, the maximum matching are $\frac{n}{8} + \frac{n}{16}$ edges.

     $\frac{n}{8}$ for nodes i in set A connected to $n - \frac{n}{4} - 1 - i$ in set B, where $i = 0, 1, ..., \frac{n}{8} - 1$; and $\frac{n}{16}$ for nodes i to $n + \frac{n}{4} + \frac{n}{8} - 1 - i$, where $i = \frac{n}{4} + \frac{n}{8}, \frac{n}{4} + \frac{n}{8} + 1, ..., \frac{n}{2} - \frac{n}{16} - 1$.
  3. Edges with nodes i to nodes $n + \frac{n}{4} - 1 - i$ will disconnected, where $i = \frac{n}{4}, \frac{n}{4} + 1, ..., \frac{n}{4} + \frac{n}{16}$.
  4. There are four types of price: 0 for nodes $\frac{n}{2}$ to $\frac{n}{2} + \frac{n}{4} - 1$; ϵ for nodes $\frac{n}{2} + \frac{n}{4}$ to $n - \frac{n}{8} - 1$; 2ϵ for nodes $n - \frac{n}{8}$ to $n - \frac{n}{16} - 1$; 3ϵ for nodes $n - \frac{n}{16}$ to $n - 1$. The sum of price is $\frac{9n\epsilon}{16}$.

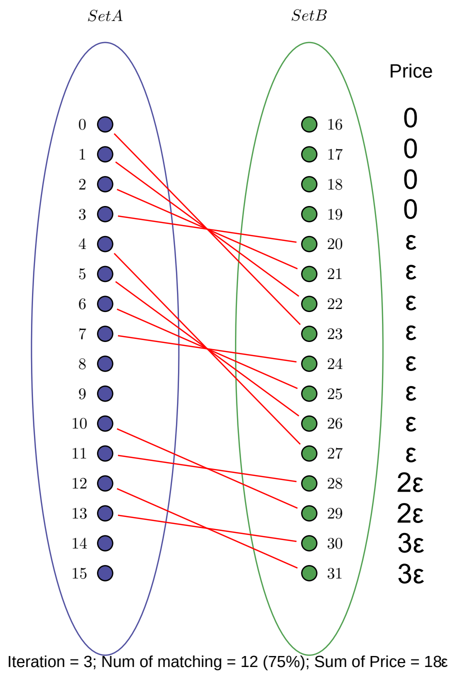


Figure 5.4: Matching state and price in the third iteration

- **Rest of Iterations**

  Figure 5.5 shows how the matching changes in subsequent iterations. In addition, we added information to the graph, such as the percentage completion and the price of each item.

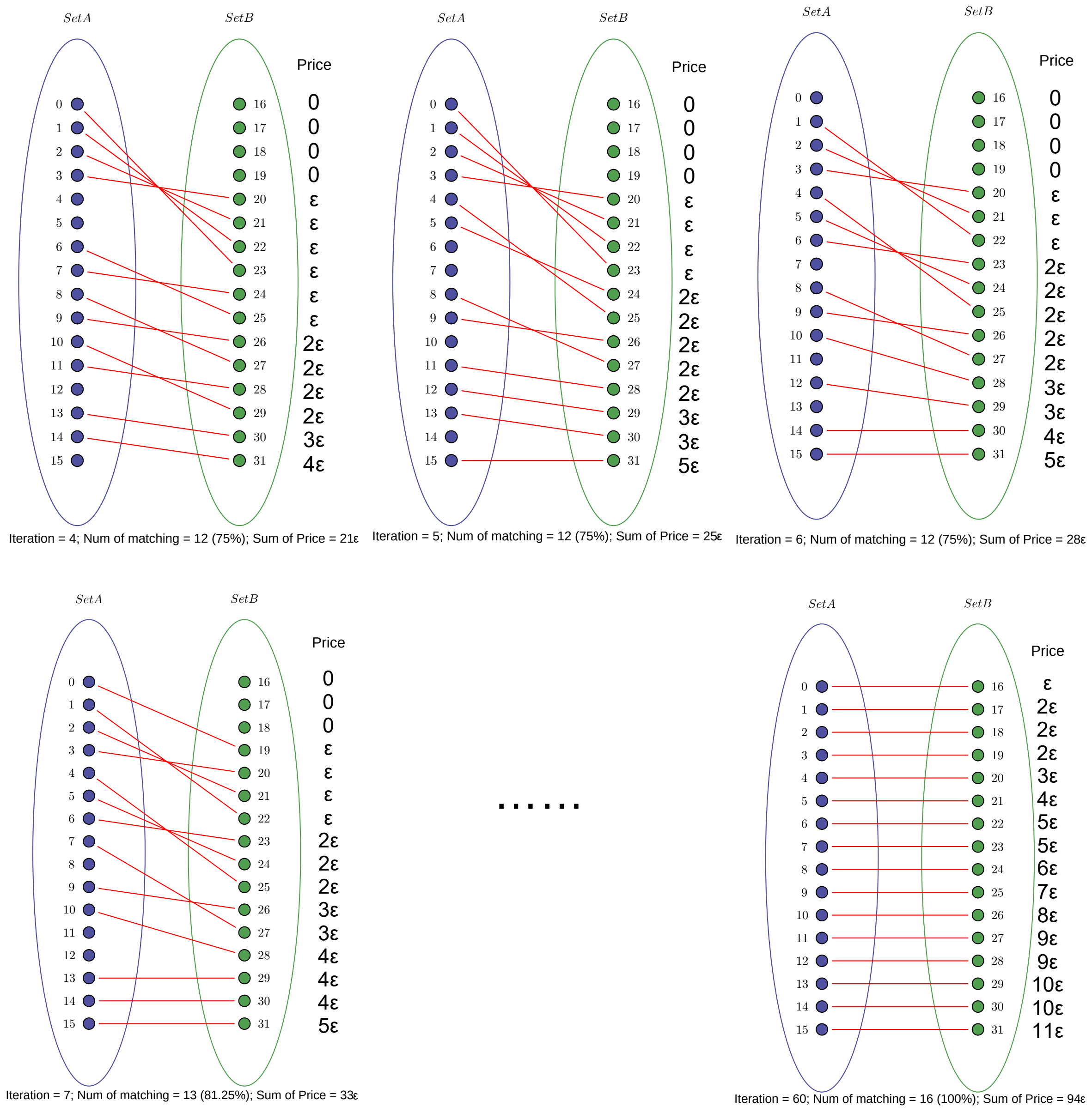

SetA
SetB
Price
Iteration = 4; Num of matching = 12 (75%); Sum of Price = 21ε
Iteration = 5; Num of matching = 12 (75%); Sum of Price = 25ε
Iteration = 6; Num of matching = 12 (75%); Sum of Price = 28ε
Iteration = 7; Num of matching = 13 (81.25%); Sum of Price = 33ε
Iteration = 60; Num of matching = 16 (100%); Sum of Price = 94ε

Figure 5.5: Matching state and price in the rest iteration

As the iterations increase, it is not easy to go any further in terms of specific values of n. However, through the theoretical analysis of previous iterations and the presentation of images, it is easy to see that the algorithm is constantly turning a large image into multiple small images. Thus, we presume a pattern for subsequent iterations and give the following observations.

Definition 5.2 (First Stage iteration)**.** We define the first stage iteration as the first correct matching to the final state of matching, in other words, the first time node $\frac{n}{2} - 1$ connected to node $n - 1$. Also, if a graph is of size 2 to the n-th power, we will reach the First Stage of iteration at the n-th iteration.

For example, in 32 nodes Semi-Complete graph, the First Stage iteration is iteration 5 since it is the first time node 15 matches node 31.

Lemma 5.1. *The prices of the item are monotonically increasing. The smallest price (first item) is always 0 except the last iteration; the largest price (last item) is strictly less than the* $\frac{n\epsilon}{2}$.

Lemma 5.2. *From the algorithm beginning to the First Stage iteration, the price of last item (nodes* $n - 1$*) equals its iteration times* ϵ.

The price of the last item does not follow the pattern after the First Stage iteration. The algorithm looks for the smallest item in the unallocated bidder. The last item not only has the largest price, but it is relatively stable as soon as the First Stage iteration is reached, after then it grows intermittently, rising to less than $\frac{n\epsilon}{2}$.

In the table below, we calculated and listed the theoretical performance of the algorithm, including, for each iteration, the unallocated bidder, the number of new matching, the number of matching to be disconnected, the total price, and the total number of matching. Note that the data here do not consider the rounding problem. In other words, this table is only true for very large graphs with a number of points be the power of 2.

No. Iteration	**No. unallocated bidders**	**No. new matching (size of M_k)**	**Matching needs to be disconnected**	**Total price**	**No. Total matching**
1	n/2	n/4	0	nϵ/4	n/4
2	n/4	n/8	n/8	3nϵ/8	n/4
3	n/4	3n/16	n/16	9nϵ/16	3n/8
4	n/8	3n/32	3n/32	21nϵ/32	3n/8
5	n/8	7n/64	7n/64	49nϵ/64	3n/8
6	n/8	11n/128	11n/128	109nϵ/128	3n/8
7	n/8	23n/256	15n/256	241nϵ/256	13n/32
8	3n/32	39n/512	39n/512	521nϵ/512	13n/32
9	3n/32	55n/1024	55n/1024	1097nϵ/1024	13n/32
10	3n/32	151n/2048	151n/2048	2345nϵ/2048	13n/32
Trend analysis	Decreasing	Decreasing unsteadily	Decreasing unsteadily	Increasing	Increasing

Table 5.1: Variable of algorithm changes in first 10 iteration

Finding new matching is getting more and more difficult in latter iteration, also matching increments will also be less and less in later iterations. The above table gives us a description of the trend in terms of numerical increments, in the longitudinal direction. And horizontally it also reveals some of following correlations.

Lemma 5.3.

- *The number of unallocated bidders equals to the number of all items minus the number of total matching from last iteration.*

  `In the table, column 5 equals` $n/2$ `minus the next row of column 1.`

- *The number of unallocated bidders is always bigger or equal than the number of new matching, and both of them are always bigger or equal than the number of matching needs to be disconnected.*

  `In the table, column 1` $\geq$ `column 2` $\geq$ `column 3.`

- *The number of new matching (size of M_k) is equal to the number of matching that needs to be disconnected, except the iteration has increased the total matching, such as iteration 1, iteration 3 and iteration 7.*

- *The total price equals the price of the last iteration plus ϵ times the number of new matching.*

  `In the table, every unit in column 4 = its last line + column 2` $\times\epsilon$.

5.2 Analysis in Percentage of Matching

As a data streaming algorithm, the Auction algotirhm has many advantages in some cases over the traditional matching algorithm (Edmond's matching Algorithm). For example, traditional algorithms cannot process data if we only need partial answers. To be more specific, we can give an initial summary toward iteration 7 (toward 80% of matching) based on the analysis and calculation in the last section.

Iteration	Percentage of total Matching	Increment
1	50%	1/2
3	75%	1/4
7	81.25%	1/16

Table 5.2: The relation between percentage of matching and number of iteration

Thinking about the final iteration, we will only add the last edge (node 0 to node $n/2$, horizontal matching). There is a huge number of iterations that will be waiting before it. The Table 5.2 includes every iteration before 7, which has the increment of matching.

We only calculate toward iteration 7. Further iteration performance is shown in the experiment Chapter 6. Using our analysis of the Table 5.2, we tentatively believe that this growth will be $O(\frac{1}{c^{d \times iteration}})$ slowing down the increment; also, the increment happened with $O(\frac{1}{c^{d \times iteration}})$ frequency, the c, d here denote two constant.

Moreover, measuring this exponential relation can be actually done by applying a few numerical analysis methods. We set an experiment in the next Chapter.

In general, new matching is becoming increasingly tricky to outcome, in terms of the number of increases and the frequency of increases. Since it will be extremely low efficiency at later iterations, thus, this algorithm is not particularly suitable for perfect matching.

5.3 Analysis of Pattern

When performing greedy matching, we consider the worst-case scenario, i.e. the least number of matching. This number is 1/2 of the original graph. We set the matching to connect to its achievable largest price nodes. Then, there will always get some regular patterns. We define them as following.

Definition 5.3 (Crossover sub-matching)**.** The Crossover(G_c) is a sub-graph and sub-matching of G. In G_c, every node $i + s$ is connected to node $k - i - s$.

The sum of two points is denoted as k, a constant between $n/2$ to $(3n - 2)/2$;

s is denoted as the size (number of matching) of this crossover.

Lemma 5.4. *If the pattern has value k, but node ID in set B minus node ID in set A not equal to $n/2$ or $n/2 + 1$, then the we call this crossover shift.*

The following Figure 5.6 shows an example of 2 crossovers. During the algorithm process, the algorithm has only one crossover at the first iteration, while at the completion, there are n/2 of such shapes with size 1. It is very similar to the idea of merging. We think it is because the core of the algorithm for finding matching uses the greedy algorithm, and the matching can grow by 1/2 the size of the subgraph each time.

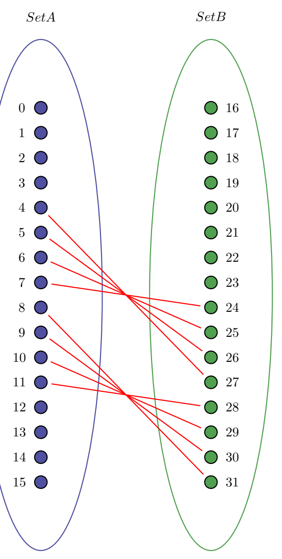


Figure 5.6: Two Crossovers in graph

See the first three iterations from previous graphs, the crossovers have no shift. However, this is not always the case in subsequent iterations. For example, in the fifth iteration, see the Figure 5.7, the second crossover has a shift of $n/32$, but in subsequent iterations, it calibrates itself. When all the offsets have been calibrated, a new matching will appear in the next iteration.

Furthermore, the graph will converge to a perfect matching as the iterations increase. The first to finish must be the matching at the bottom edges. It is because we default to finding the smallest item in the demanding set first.

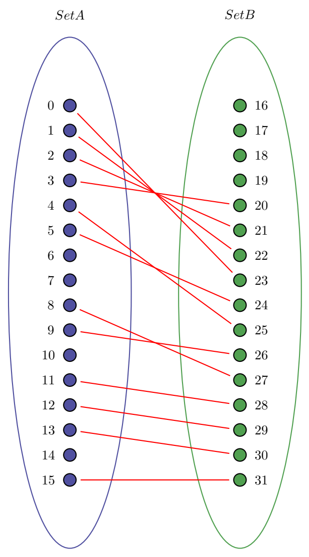


Figure 5.7: Second Crossover shifted

5.4 Analysis of Epsilon

We are now starting our research on epsilon. In this algorithm, epsilon is also set to the value of item increase, and if the total value exceeds 1, it no longer goes into the demanding set. We will know how to choose a suitable epsilon through this part analyses.

From analyses in Chapter 3, we introduced that Auction Algorithm will terminate in a finite number $O(1/\epsilon^2)$ of steps. The number of iterations required increases significantly as ϵ decreases. Conversely, to obtain the most answer at termination, the value of epsilon must be below a certain threshold.

As Figure 5.5 shows, the prices of items are ascending. The price of the last item will be $n\epsilon/2$ at most, and this value should not exceed 1. The algorithm terminates otherwise. Thus we can get the following lemma.

Lemma 5.5. *The ϵ has a lower bound (0). Note that, 0 is not included.*

Lemma 5.6. *The ϵ below than $\frac{1}{n/2}$ can find every matching. But it is not the largest upper bound for ϵ.*

In chapter 3, it was presumed that last item's price is $n\epsilon/2$. However, in figure 5.5, this value is smaller than the $n\epsilon/2$. Thus, $\frac{1}{n/2}$ is not the largest upper bound for ϵ. In another instance, in the Semi-Complete graph with 10 nodes, we can find every matching if ϵ can be any value in $(0, 1/4)$, instead of $(0, 1/5)$.

5.5 Summary of Chapter

The Semi-Complete graph is the most challenging dataset to deal with in the Auction Algorithm. We address the theoretical analysis of this algorithm from several perspectives, including the start of the algorithm, completion, shape, and choice of epsilon. In the next section, we follow this chapter's conclusions and show more of our findings.

Chapter 6

Experiments and Results

In this Chapter, we have designed several experiments to test our hypothesis, as well as prove the theoretical analysis from previous Chapter. We will also use our Auction matching Algorithm to produce exact solutions for some instances and provide insight into the improvement explored.

Our test environment is `arm64 ARCHS` CPU with frequency 3.2 GHz.

6.1 Datasets

We tested our algorithm on fifteen different datasets of bipartite matching graphs. Among them, seven were Semi-Complete graphs, four automatically generated random graphs, and the other four instances generated from social networks in a SNAP database.

- We test algorithms on the Semi-Complete graph to prove and discuss our conclusions in Chapter 5 Analyses.
- We used Random graphs to test whether our prepossessing step can small random graphs.
- We finally experimented with the real-world dataset, which contains several large examples from actual Internet companies. They were used to validate practical uses. In particular, they were tested to see if they could be used for large amounts of data.

6.1.1 Semi-Complete Graph

The following table illustrates the information of the instances converted from the graph, along with the various data associated with them, such as Number of Vertices, Number of Edges, and Number of Maximum Matching. First, we show seven sets of Semi-Complete graphs sorted from small to large by a factor of 2. See Table 6.1.

By looking at Table 6.1, it basically meets the expectations of our analysis in Chapter 5. Our Auction Algorithm is a $(1 - \epsilon)$ data streaming algorithm. As discussed in the implementation chapter, our improvement allows the algorithm to exit earlier if the demanding set is empty.

ID	**Type**	**Nodes**	**Edges**	**Number of Maximum Matching**	**80% of Matching**	**90% of Matching**	**95% of Matching**	**Number of Iteration to find all matching**
1	Semi complete	32	136	16	7	14	60	60
2	Semi complete	64	528	32	7	18	47	210
3	Semi complete	128	2080	64	7	26	44	612
4	Semi complete	256	8256	128	7	29	64	2966
5	Semi complete	512	32896	256	7	30	101	9758
6	Semi complete	1024	131328	512	7	30	90	44645
7	Semi complete	2048	524800	1024	7	30	94	178296

Table 6.1: The result of seven Semi-Complete graphs

Thus, we can compare the iteration it needs with the theoretical iteration. We will discuss our results in two cases, one where we get a partial answer and one where we get the full answer.

6.1.1.1 Partial matching

When $\epsilon = 0.2$, we want 80% of the maximum matching. As Figure 6.1 shows, no matter the size of the graph, we only need around seven iterations; Meanwhile, when $\epsilon = 0.1$, we hope to achieve 90% of the maximum matching. We need around 30 iterations, as Figure 6.2 shows.

The two figures below both have a converge to a specific value. However, we find oscillation in the beginning part of 80% figure and in the 90% figure. This oscillation is because the number of points in our graph is an integer and the percentage increase is often fractional. Nevertheless, as long as the epsilon is greater than 0, it must eventually converge to a certain number. For the last example, looking back at the previous table, for a larger Semi-Complete graph of 512 to 2048 points, around 100 iterations are sufficient to complete 95% of the matching.

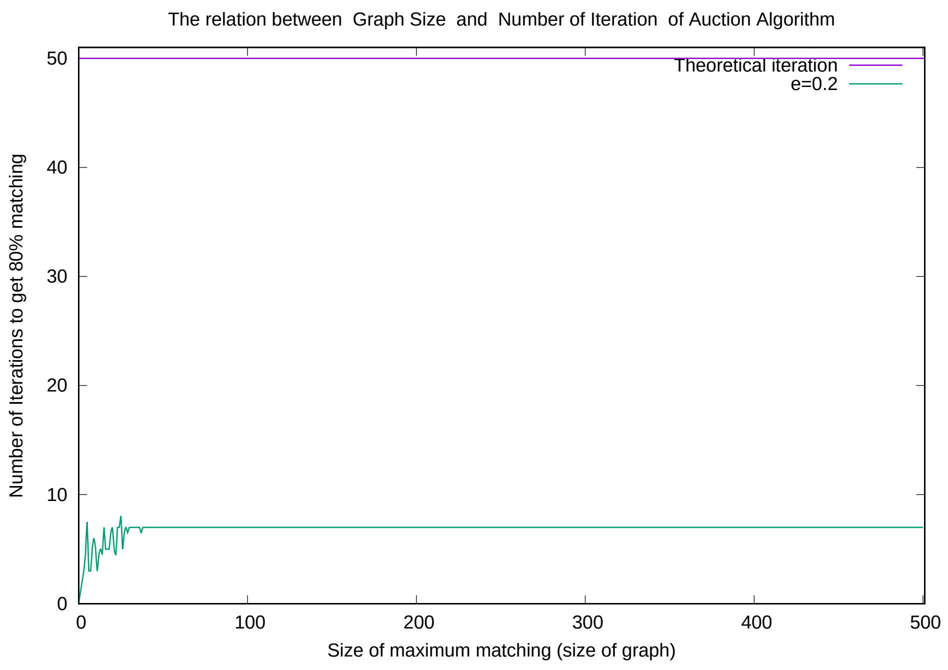


Figure 6.1: The relation between Size and Number of iteration to get 80% matching.

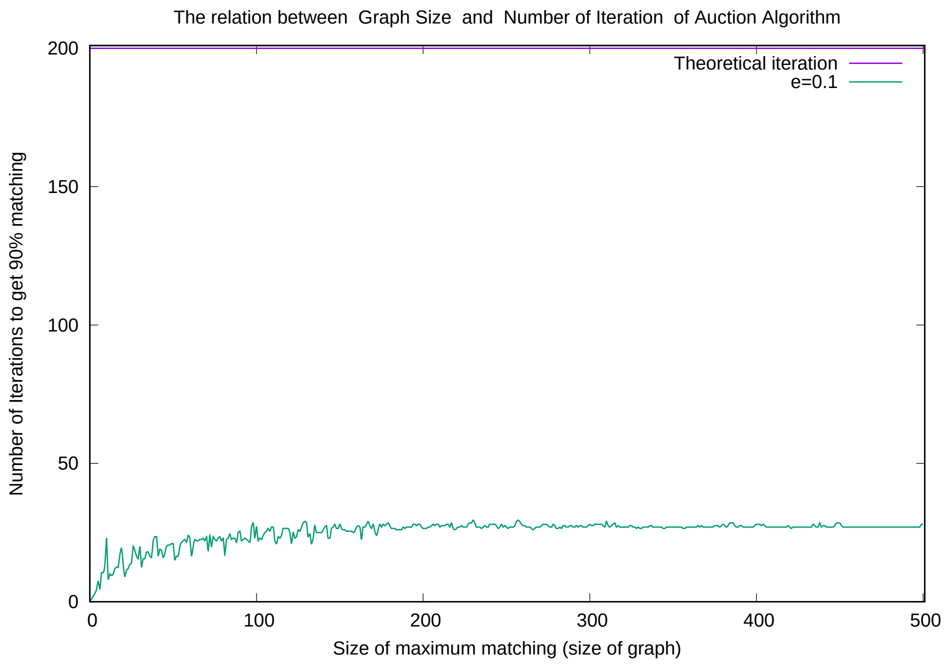


Figure 6.2: The relation between Size and Number of iteration to get 90% matching

Moreover, these two graphs show an essential property of the streaming algorithm, which works perfectly for a large dataset. The algorithm has the approximation guarantee, the larger size of the graph, the more stable number of iterations it requires. We find the number of iterations it really required is much less than the theoretical iteration from the original algorithm $\frac{2}{\epsilon^2}$. Here, 7 and 30 less than 50 and 200, respectively.

With this finding, we have determined that the Auction Algorithm is applicable when partial answers are desired. We build large graphs and compute the following Table 6.2 shows the algorithm performance. Note that it will be more accurate on the large graph with the size of power 2.

Iteration	Percentage of Maximum Matching Finding
1	50%
3	75%
7	81.25%
11	84.38%
13	87.50%
18	89.06%
28	89.98%
30	91.41%
34	92.38%

Table 6.2: The relation between Percentage of maximum matching and Iteration

6.1.1.2 Perfect matching

In the two graphs below, the x-axis represents the size of the Semi-Complete graph; the y-axis represents the number of iterations required. The number of iterations required grows very quickly when we need to find a perfect matching.

We have plotted its upper bound $\frac{2}{\epsilon^2}$ on the first graph. By our previous analysis, we can set ϵ here to be $\frac{1}{n/2}$ to achieve every matching. See the Figure 6.3.

In Figure 6.4, we applied one exponential regression and one polynomial regression function, $y = 87.8 * 1.02^x$ and $y = 0.31 * x^{1.87}$, respectively. These two functions are fitted based on the results of size of 1 to 250 Semi-Complete graphs from small to large. By looking at these two functions, it is clearly to verify our previous theoretical analysis of $O(n^2)$. See both functions, the exponential function grows faster than the actual result, while function $O(n^{1.87})$ clearly grows slower than the actual result.

In general, this algorithm does not perform very well on Semi-Complete graphs. But this is reasonable, since the Semi-Complete graph was designed separately for this algorithm. The worst case is tougher, if we need close to the maximum number of matches, requiring a large number of iterations. This data only requires $O(n^2)$. These two graphs also verify that this algorithm does not have an advantage in solving exact matches.

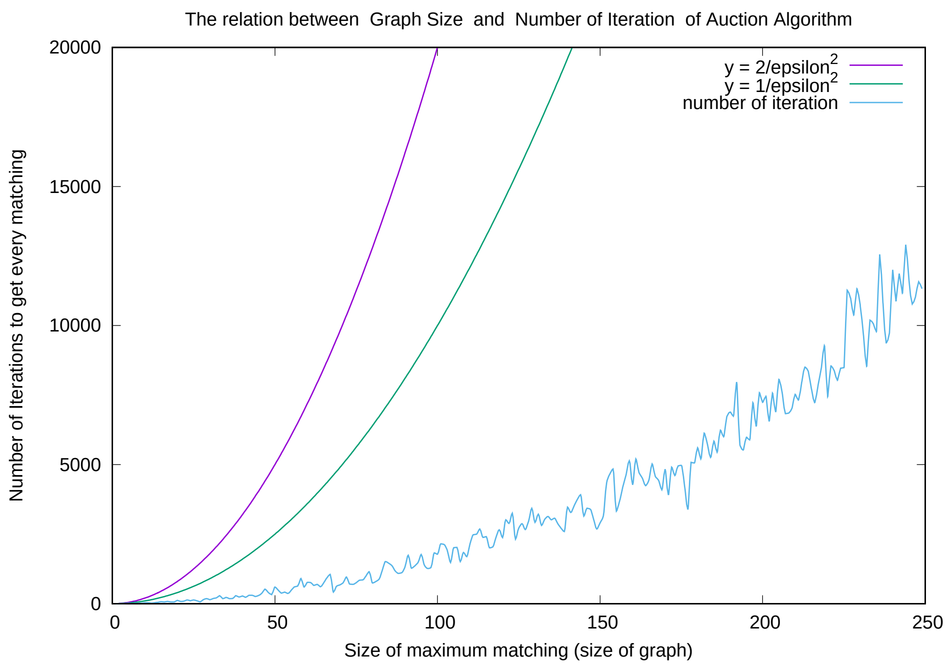


Figure 6.3: The upper bound and relation between Size of graph and number of iteration to get perfect matching

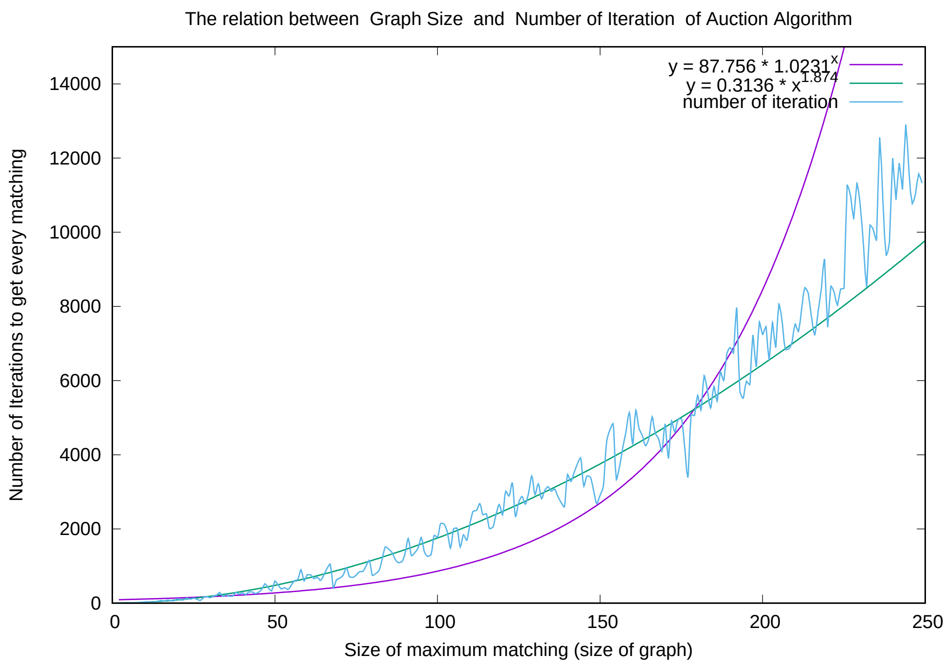


Figure 6.4: Two regression for the relation between Size and Iteration required

6.1.2 Random Graph

We also generated random graphs to simulate small samples of real-life data to test the success of the program. See the following Table 6.3. The degree of nodes affects the required iteration. The fully connected graph has an easy solution, same as the graph with only a few edges. By contrast, the Semi-Complete graph is much more difficult. However, the algorithm generally works very well on random graphs, and many cases only require a few iterations.

ID	Type	Nodes	Edges	Number of Maximum Matching	90% of Matching	95% of Matching	Number of Iteration to find all matching
8	Random	128	106	44	1	3	4
9	Random	128	236	61	3	4	16
10	Random	128	681	64	1	1	5
11	Random	128	1853	64	1	1	6

Table 6.3: Results for random graph

6.1.3 Real-world Graph

Finally, we tested several graphs of the real-world dataset. Our code can be applied quickly to most real-world graphs. When the graph is extremely large, see test ID 15 for an example, even the build-in method cannot be completed in a few seconds. Our method can use a few iterations to find a partial matching.

ID	Type	Description	Nodes	Edges	Number of Maximum Matching	90% of Matching	95% of Matching	99% of Matching	Number of Iteration to find all matching
12	Real world	Twitter	1293	8175	554	1	3	6	30
13	Real world	Facebook	4039	44008	1839	1	3	23	146
14	Real world	Google+	107614	6121041	42567	4	7	37	920
15	Real world	LiveJournal	4847571	21425445	unknown (approx. 1730000)	4 iteration find over 1600000	11 iteration find over 1700000	113 iteration find over 1728000	unknown

Table 6.4: Results for real-world graph

6.1.4 Summary for dataset result

The Semi-Complete graph is a well-designed complex graph for Auction Algorithm. The number of iterations to find perfect matching increases as the graph grows, while the number of iterations to find partial matching converges as the size of the graph grows.

The Auction Algorithm is perfectly working on the random graph and real-world graph. Our method does not find all the results directly in an extremely large graph, but we can provide part of the results first and optimise the result as the iteration grows.

6.2 Performance of improvements

After testing the dataset, our algorithm can be applied to the most non-extremely large graph. We will continue to validate the theories and improvements we presented earlier.

6.2.1 Epsilon bound

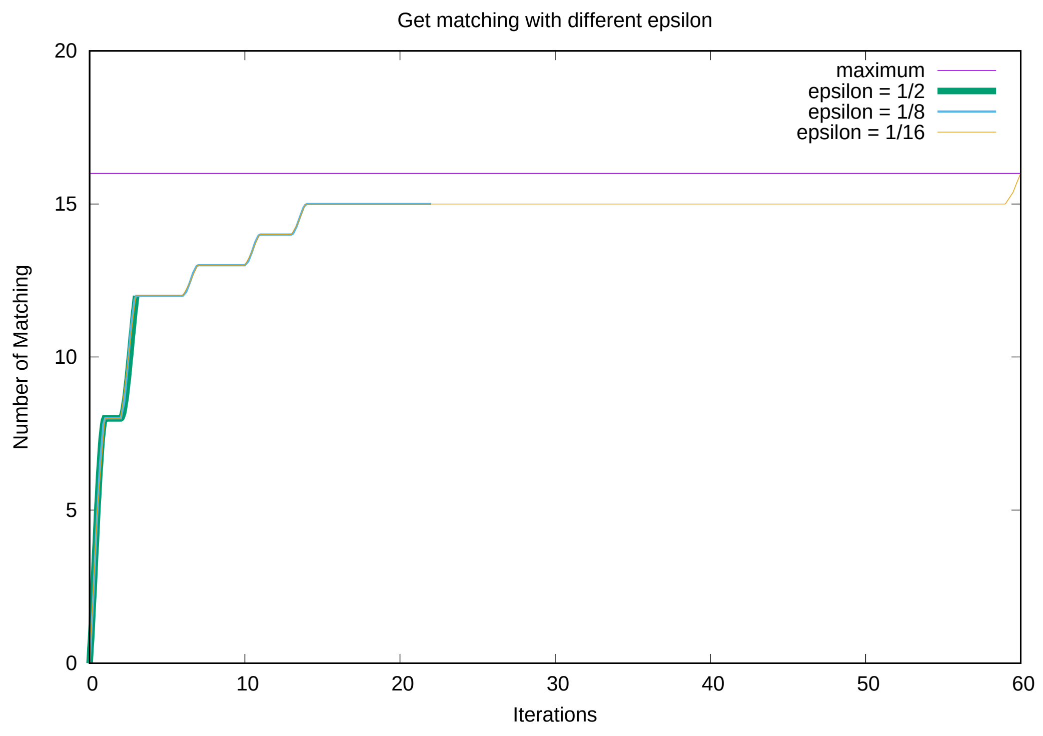
As we mentioned in our previous study, ϵ plays a role as a parameter in the approximation function of $1 - \epsilon$. In the perfect matching problems, we could theoretically set ϵ to any number greater than 0 and less than $\frac{1}{n/2}$. ϵ should ideally be 0, but is also a value for price increment, 0 is not appropriate. Too small ϵ will lead to a reduction in the speed of the calculation and may cause accuracy problems in Floating-point arithmetic; ϵ can be slightly bigger than the $\frac{1}{n/2}$, but too large ϵ is also inappropriate and may not find every matching.

The practical result is shown in Figure 6.5. Both graphs are Semi-Complete with 32 nodes and 16 maximum matching in the purple line.

On the upper graph in Figure 6.5, the thick green line shows the performance of the Auction Algorithm, with ϵ set to half. This green line stops at the third iteration and gets 12 matches greater than $1 - \epsilon$ (greater than 50%). By zooming in on the upper graph, here is another blue line. The ϵ for the blue line is 1/8, but it has around 22 iterations and finds 15 matches. The blue line is able to find more matching than the green line. We can also see that it takes much iteration to process from 15 matching to 16 matching.

On the lower graph in Figure 6.5, two performances are exactly same in different ϵ. So please zoom in to see the details. In theory, we need to set the ϵ to 1/16, but here 1/11 is also correct. The same result is also seen for the 1024 nodes Semi-Complete graph, which theoretically requires 1/512. However, in practical, setting ϵ be 1/250 also be able to find all the matches. Although they both have the same progress and require the same number of iterations, the line with larger epsilon should reduce the number of floating-point errors in the computer in terms of engineering perspectives. We will give our comparison test before and after improvement in later section.

For ϵ bound in general, a smaller ϵ allows the algorithm to increase the iteration further and find more matching; the ϵ can also be slightly larger than $\frac{1}{n/2}$, which has the same performance and possible to improve speed and accuracy in practical.



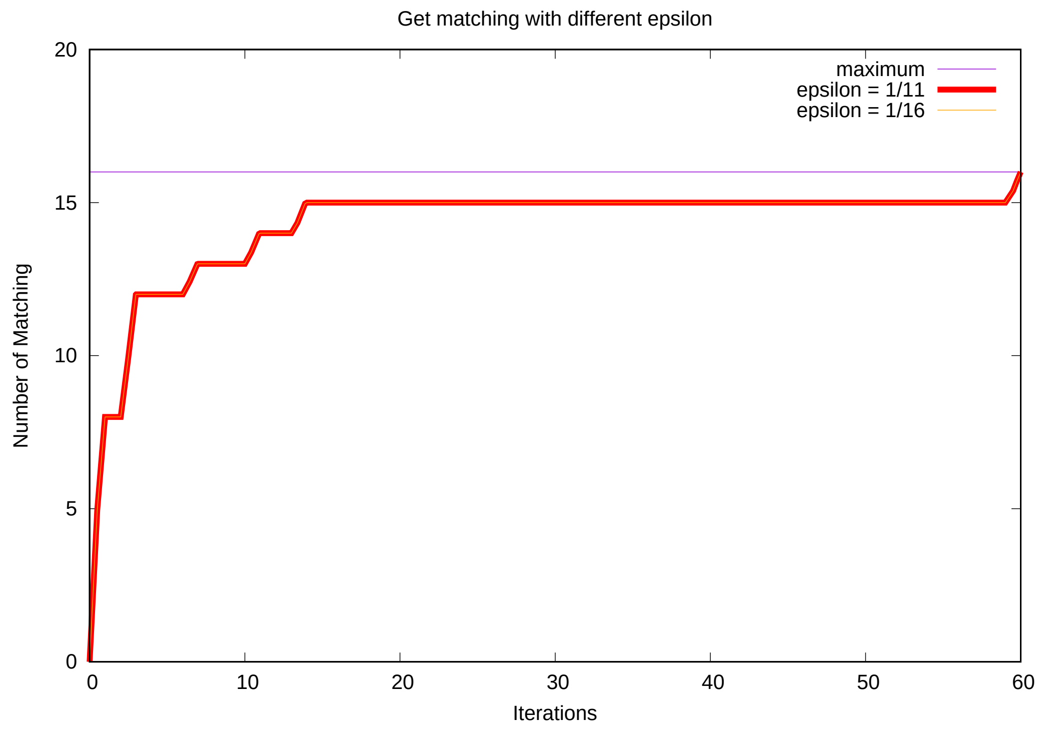


Figure 6.5: The first 60 iteration of Auction Algorithm on 32 nodes Semi-Complete graph with different epsilon

6.2.2 Aggressive Epsilon

Not only ϵ affects the completion percentage, but it also changes the increment price. It is difficult to get new matching in later iterations when encountering complicated graphs. Here we give an experiment on the feasibility and the effect of using multiple aggressive epsilons to increase their price. The detail of the aggressive strategy was discussed in the Chapter 5 implementation.

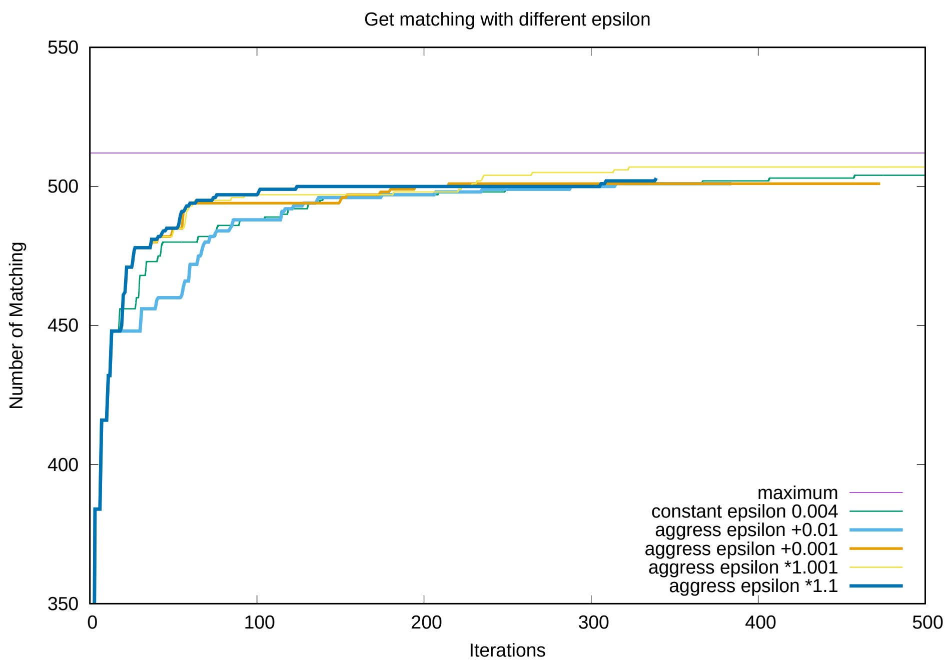


Figure 6.6: The Auction Algorithm with aggressive epsilon on 32 nodes Semi-Complete graph

The dataset ID 6 Semi-Complete graph is selected. The green line is the based epsilon (0.004), which can find every matching with 40000 iterations. The dark blue, light blue, and orange lines cannot find every matching, since they terminate earlier. The yellow line performs well not only in 90% and 95% of matching, but can achieve 90% matching. For more details, see the following table 6.5.

In the example we tested here, we let based epsilon be 0.004, higher than the theoretical upper bound (0.002). It is very close to its upper bound, any not mild epsilon will miss the solution. As we mentioned before, we think it is unnecessary to have an aggressive epsilon unless we can find a suitable strategy for aggressive.

Aggressive Method	Line color	Iteration 90% of Matching	Iteration 95% of Matching	Iteration 99% of Matching	Iteration to find all matching
constant epsilon 0.004	Green	30	90	903	44644
aggress epsilon +0.01	Light blue	55	86	NA	NA
aggress epsilon +0.001	Orange	20	56	NA	NA
aggress epsilon *1.001	Yellow	20	57	323	NA
aggress epsilon *1.1	Dark blue	20	54	NA	NA

Table 6.5: Aggressive method and their performance

6.3 Performance of algorithm in practical

Finally, we will show algorithms performance in time in this section. We will also compare the Auction Algorithm with performance of the built-in algorithm and the optimised algorithm. Each test was within a running time of 300 seconds, all units in the table 6.6 below are seconds.

Test ID	**Dataset Description**	**Built-in method**	Time 90% of Matching	Time to find all matching	**Improved Time 90% of Matching**	**Improved Time to find all matching**
6	Semi-Complete 1,024 nodes	0.028	0.019	0.243	0.018	0.239
7	Semi-Complete 2,048 nodes	0.182	0.048	3.103	0.047	1.137
12	Twitter 1,000 nodes	0.004	0.006	0.036	0.005	0.031
13	Facebook 4,000 nodes	0.010	0.007	0.034	0.005	0.032
14	Google+ 100,000 nodes	8.544	1.872	7.231	1.683	6.529
15	LiveJournal 4,000,000 nodes	Unknown more than 300 seconds	24.635	Unknown more than 300 seconds	20.172	Unknown more than 300 seconds

Table 6.6: The comparison performance of algorithm in practical (second)

For finding the perfect matching, the improved Auction Algorithm was generally better than the original Auction Algorithm. Also, the improved algorithm was better than the built-in algorithm. The Semi-Complete graph was the most complicated case we devised. It did not perform as well as the built-in algorithm, apart from in test ID 14 Auction Algorithm performing better on the large dataset.

However, if the question is only looking for partial answers, such as 90%, the Auction Algorithm is much better (18% better in Test ID 15) than the built-in algorithm.

6.4 Summary of Chapter

In this Chapter, we presented our experiments on 15 different datasets and collated their results. We tested and verified our previous theories and conjectures.

It demonstrates the ability to find partial results quickly as a streaming algorithm. By comparison, we can see that the Auction Algorithm performs better than the current state-of-the-art matching algorithm in large dataset. Finally, the test illustrates the effect of our improvement on the Auction Algorithm.

Chapter 7

Conclusion

7.1 Contributions

Overall, this project taught us a wide range of details about the Auction Algorithm. The combination of theory and practice has given us the following contributions.

Through theoretical analysis in the worst-case scenario, we simulated the first few steps and discovered this streaming algorithm's characteristics. We found that there is a larger upper bound for epsilon, and it is related to the price of items; also we present some analysis of the algorithm price and pattern.

After verifying these theoretical analyses by coding and testing, theory and practice helped us to indicate our optimisation directions. We have improved the performance of original Auction Algorithm from perspective of algorithm engineering. We managed to obtain an average improvement of 12.7% compared to original algorithm with the following improvements:

- Changing the data structure to map allocation and price in O(1); Inserting these maps into the graph to reduce space.
- Using an early exit in loop, reducing the iteration number of our algorithm.
- Using a larger epsilon to reduce computational precision and increase speed.
- Choosing the suitable aggressive epsilon can increase speed. However, note that excessive aggressive epsilon can break the approximation function, and possible no sufficient answer will be given.

7.2 Future work

Time has passed quickly. Our learning process has revealed that the project still has a lot of potential for improvement:

- **More theoretical analysis.**

  1. Predicting the number of iterations required to find perfect matching on different size of Semi-Complete graph. As well as the relation between the number of iterations required and $(1 - \epsilon)$ matching.
  2. We found that epsilon has an unknown upper bound. We can check the price of the last item when the matching ends. However, is it possible to predict its exact value before the code runs?
  3. Research a better time complexity algorithm for the matching problem, since the improvement from engineering perspectives is limited.

- **Improvements from engineering perspectives**

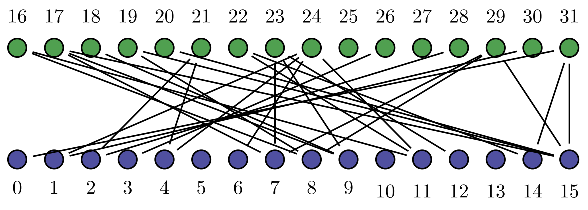
  1. For random graphs, algorithm lets us choose any maximum matching. Is there a better algorithm? Currently, we use the greedy algorithm. Can we apply two-pass matching algorithm [9] to improve the run time?
  2. Many authors have mentioned that this algorithm can be extended to parallel computing. Study how we can optimise it in parallel computing.

Appendix A

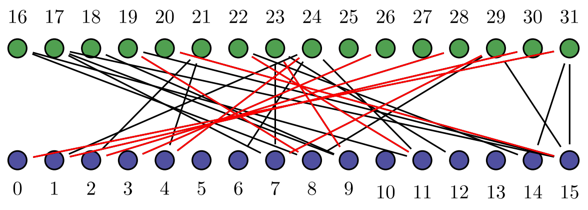
Example of Algorithm on Random Graph

In the next few pages, we show how Auction Algorithm working on a 32 nodes Random Graph in each iteration. The first graph is the neighbourhood relation between Set A and Set B. The red line in latter graph shows the matching. To see the video, please click https://www.youtube.com/shorts/Dw-TVHN3lmc.

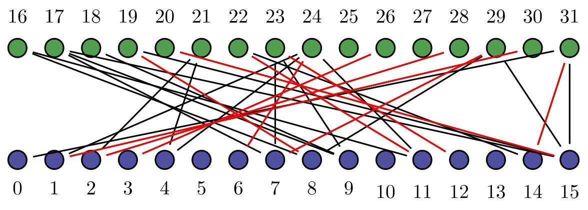
In the Iteration 5, we find 13 matching, it is the largest matching.



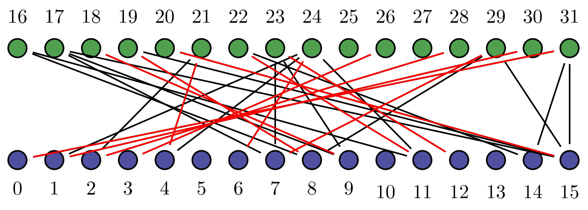
Auction Algorithm on (32 nodes random) bipartite graph



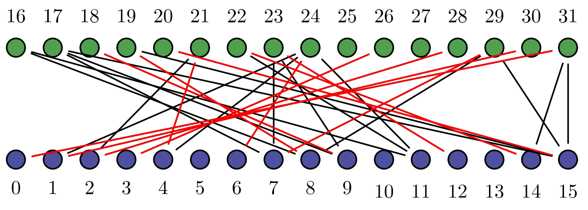
Iteration 1, 10 matching



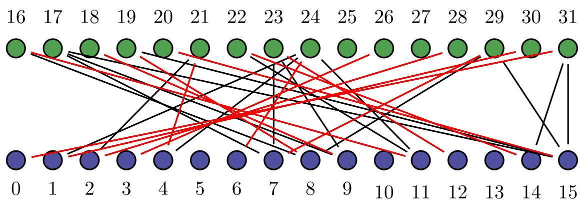
Iteration 2, 10 matching



Iteration 3, 12 matching



Iteration 4, 12 matching



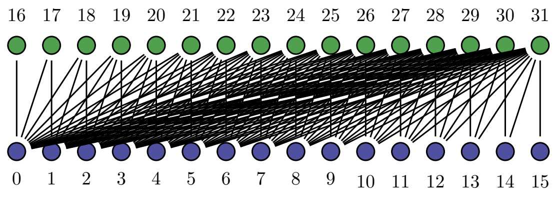
Iteration 5, 13 matching

Appendix B

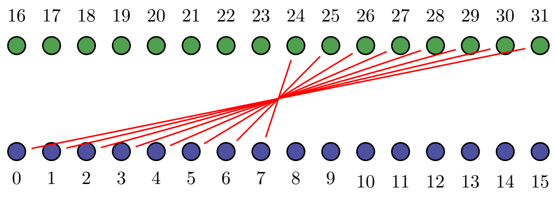
Appendix Example of Algorithm on Semi-complete Graph

In the next few pages, we show how Auction Algorithm working on a 32 nodes Semi-complete Graph in each iteration. The first graph is the neighbourhood relation between Set A and Set B. The red line in latter graph shows the matching. To see the video, please click https://youtu.be/pvaFh3txBRE.

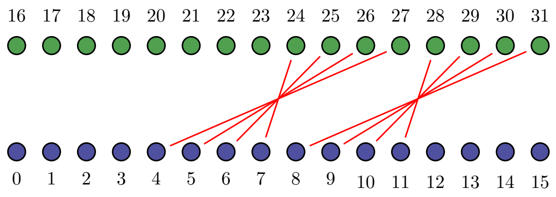
In the Iteration 60, we find 16 matching, each nodes in Set A connect to nodes in Set B, it is the largest matching.



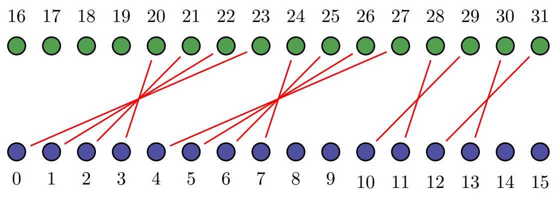
Auction Algorithm on (32 nodes random) bipartite graph



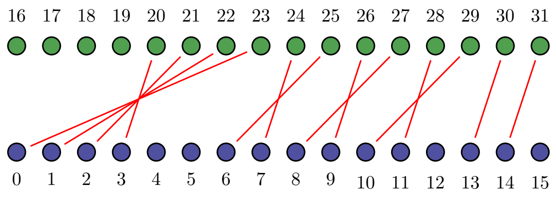
Iteration 1, 8 matching



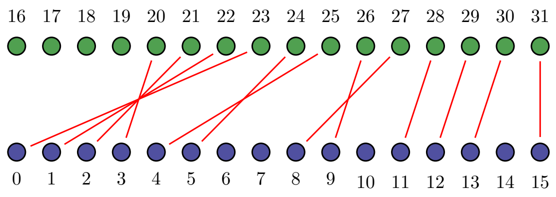
Iteration 2, 8 matching



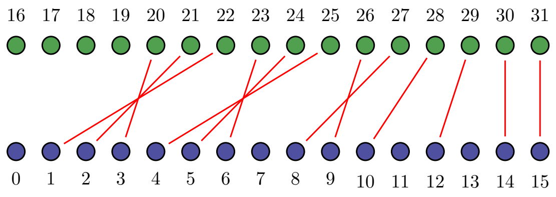
Iteration 3, 12 matching



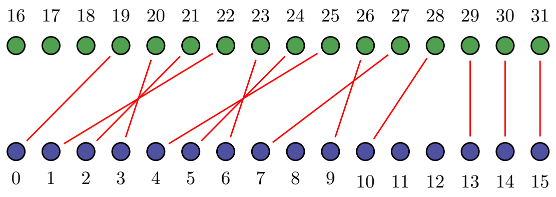
Iteration 4, 12 matching



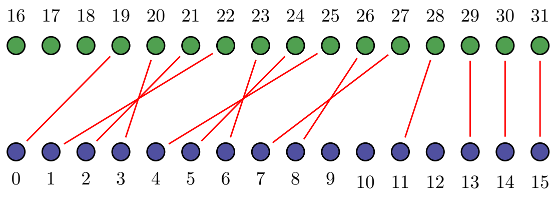
Iteration 5, 12 matching



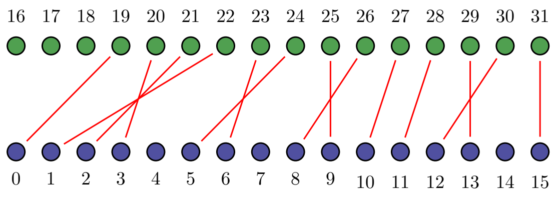
Iteration 6, 12 matching



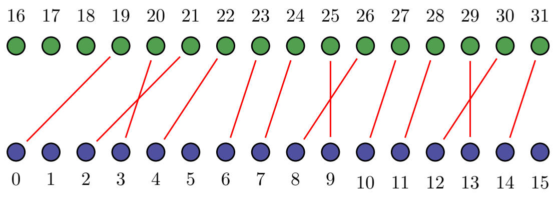
Iteration 7, 13 matching



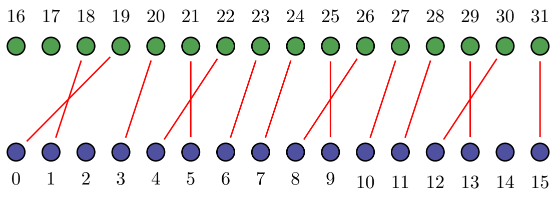
Iteration 8, 13 matching



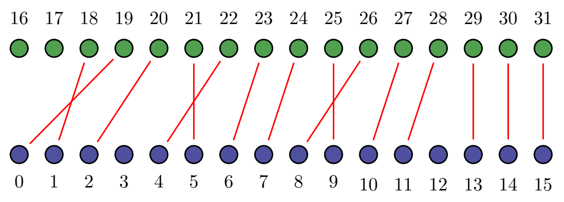
Iteration 9, 13 matching



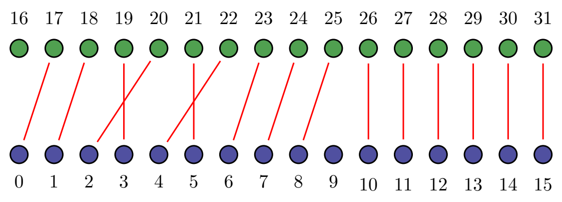
Iteration 10, 13 matching



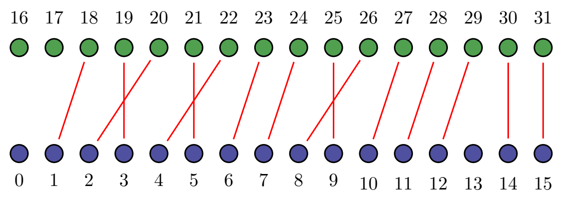
Iteration 11, 14 matching



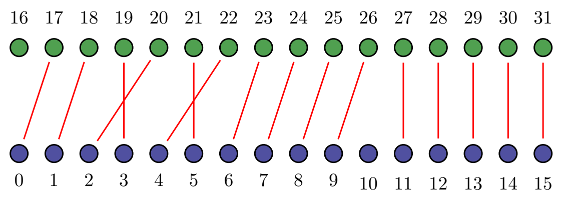
Iteration 12, 14 matching



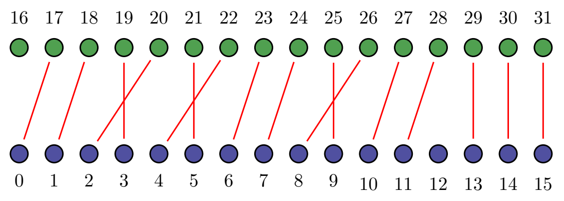
Iteration 18, 15 matching



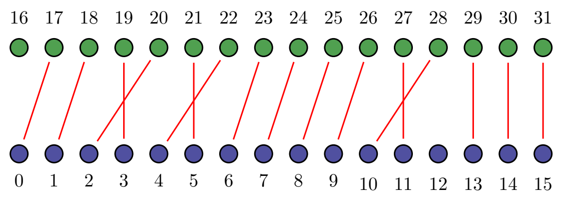
Iteration 13, 14 matching



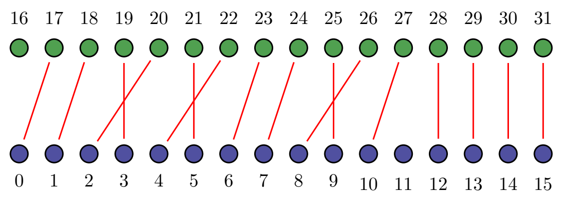
Iteration 19, 15 matching



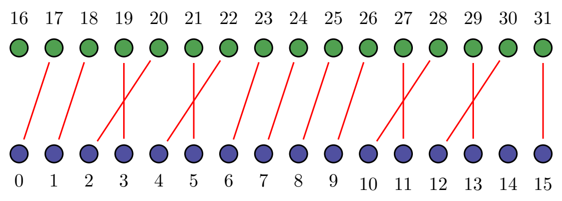
Iteration 14, 15 matching



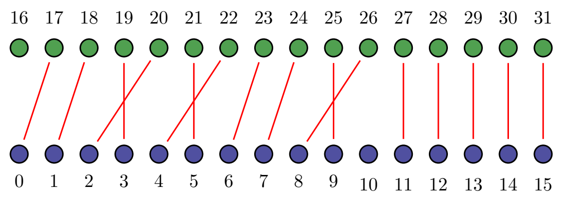
Iteration 20, 15 matching



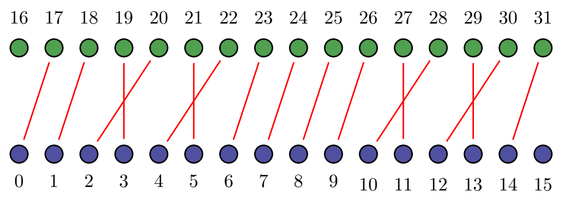
Iteration 15, 15 matching



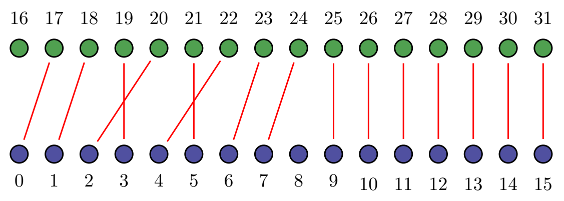
Iteration 21, 15 matching



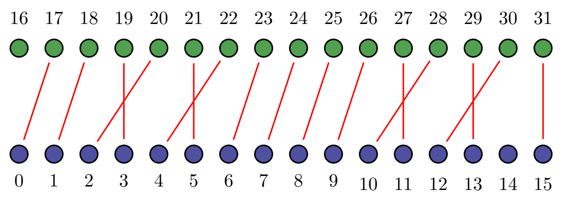
Iteration 16, 15 matching



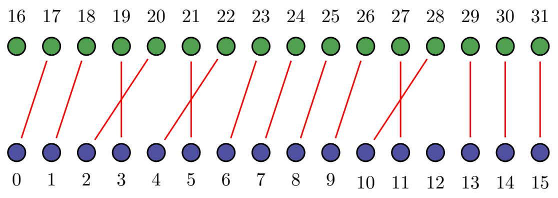
Iteration 22, 15 matching



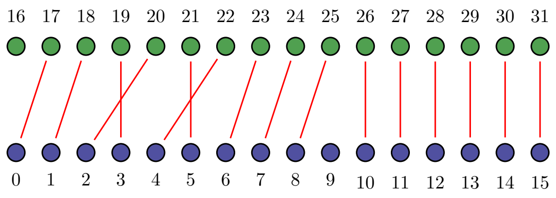
Iteration 17, 15 matching

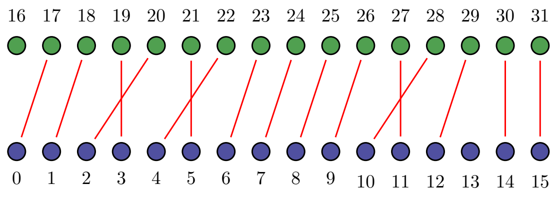


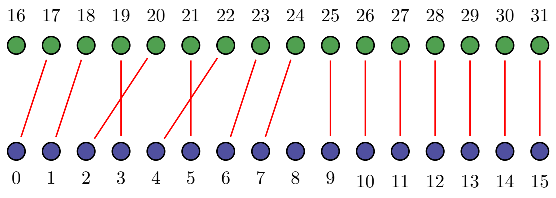
Iteration 23, 15 matching

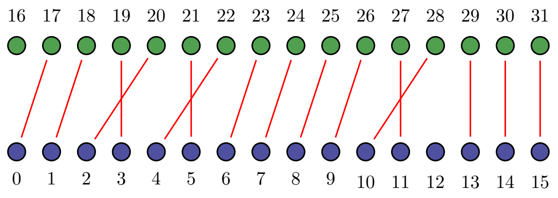


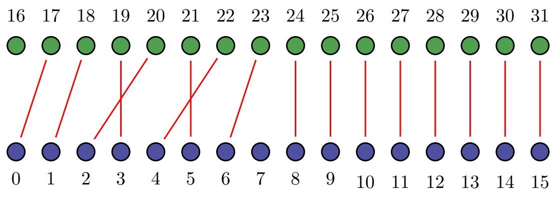
Iteration 24, 15 matching

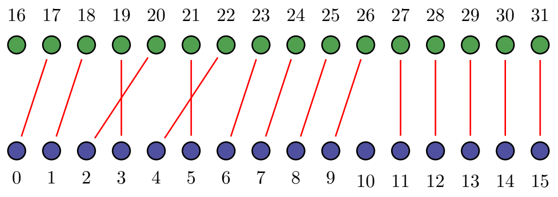


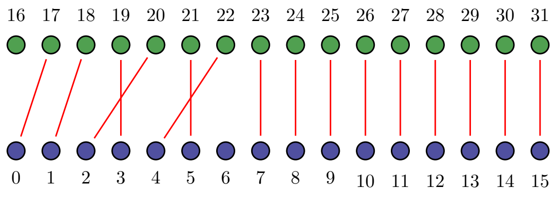


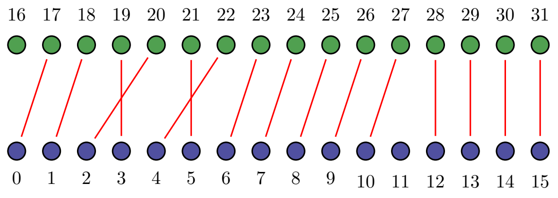


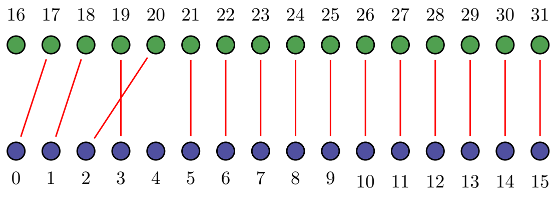


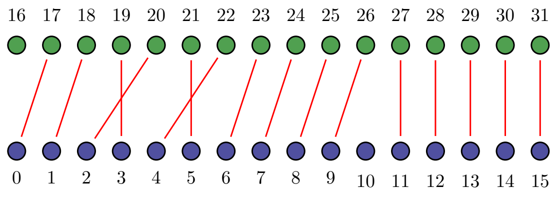


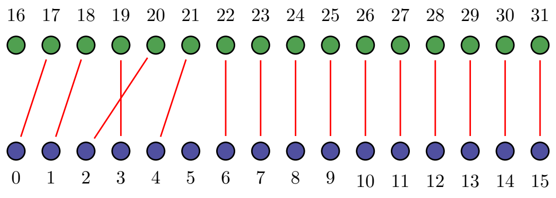


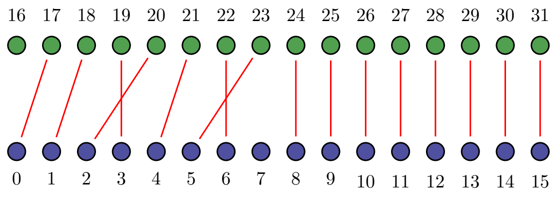
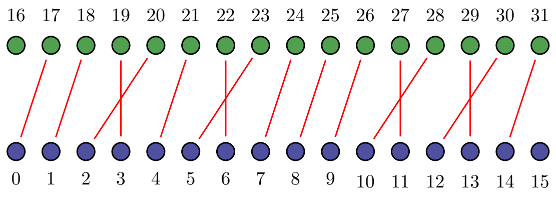
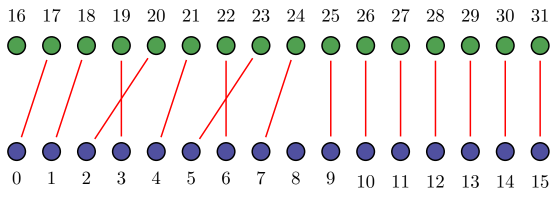
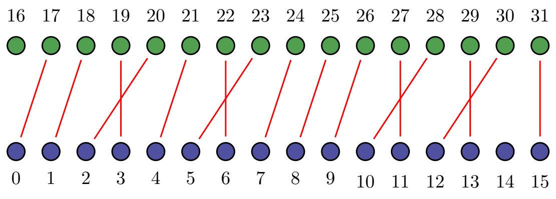
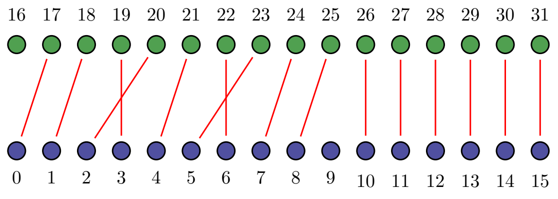
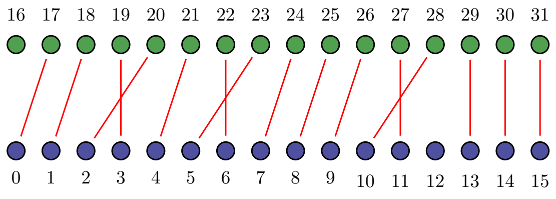
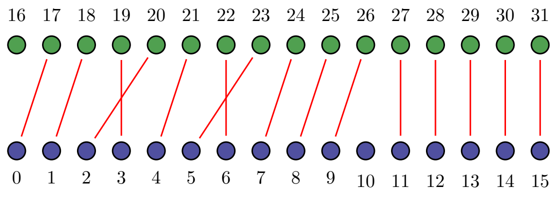
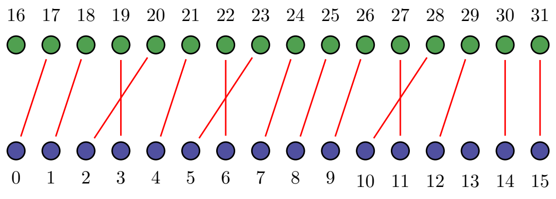
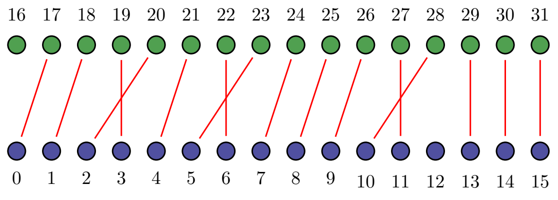
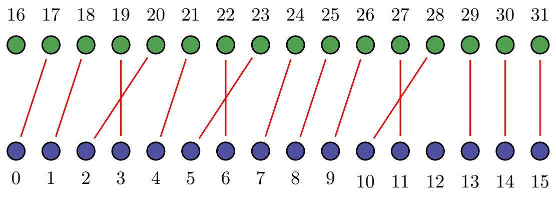
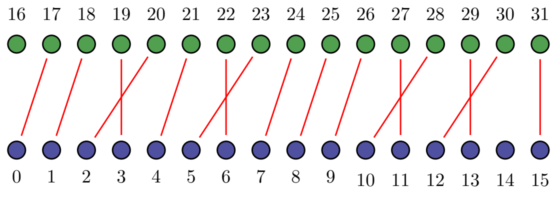
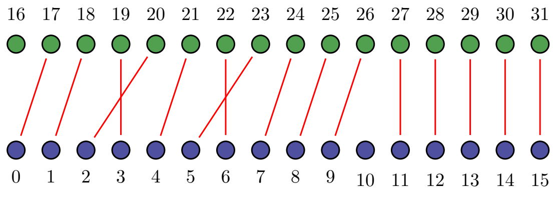


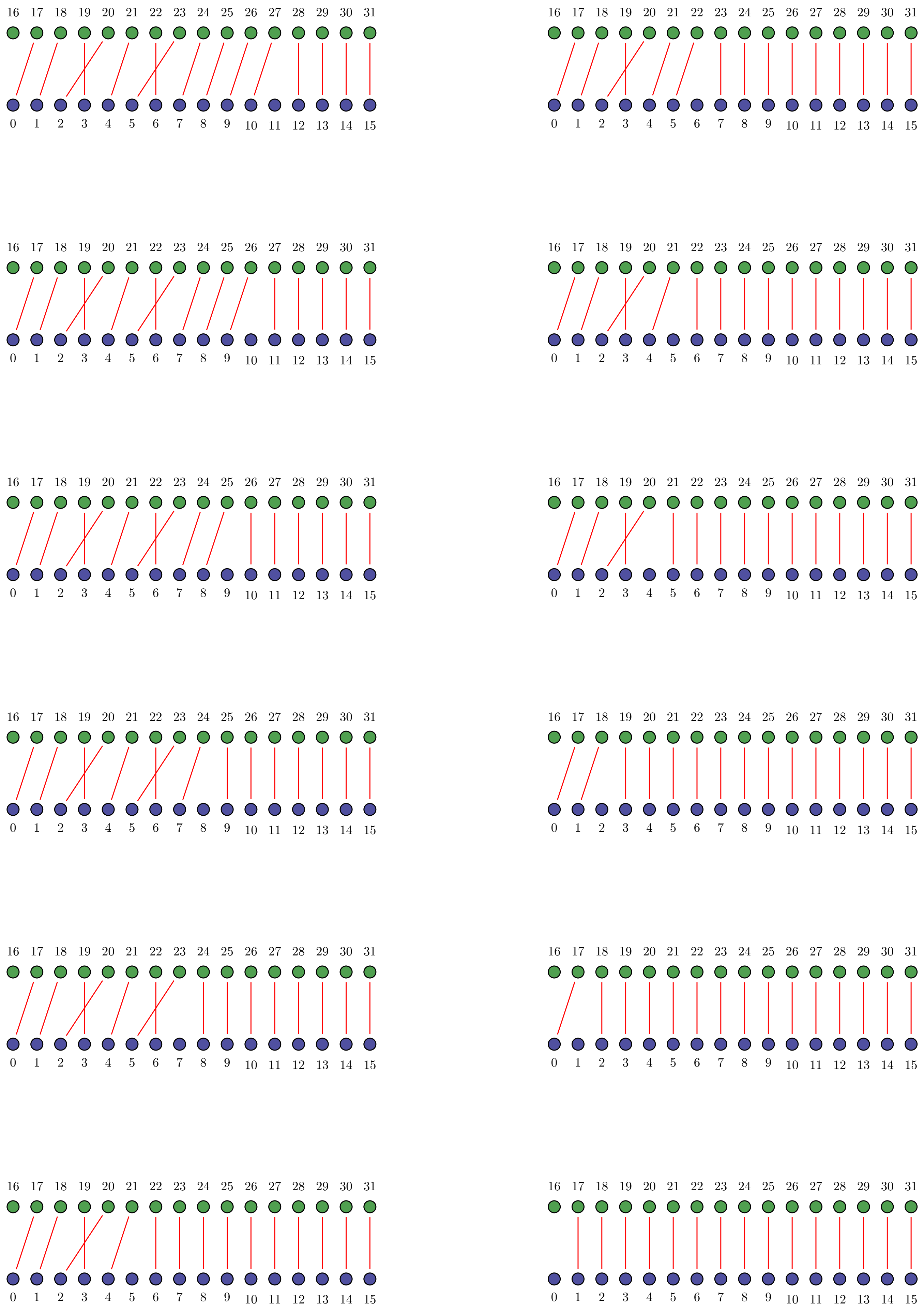




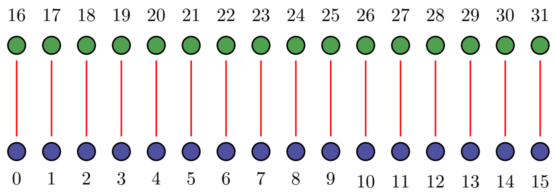





16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15

16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
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 26 27 28 29 30 31
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 26 27 28 29 30 31
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 26 27 28 29 30 31
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 26 27 28 29 30 31
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 26 27 28 29 30 31
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 26 27 28 29 30 31
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 26 27 28 29 30 31
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 26 27 28 29 30 31
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 26 27 28 29 30 31
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 26 27 28 29 30 31
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15



Iteration 59, 15 matching



Iteration 60, 16 matching

Bibliography

[1] K. J. Ahn and S. Guha, *Linear programming in the semi-streaming model with application to the maximum matching problem*, in International Colloquium on Automata, Languages, and Programming, Springer, 2011, pp. 526–538.

[2] S. Assadi, S. C. Liu, and R. E. Tarjan, *An auction algorithm for bipartite matching in streaming and massively parallel computation models*, in Symposium on Simplicity in Algorithms (SOSA), SIAM, 2021, pp. 165–171.

[3] D. P. Bertsekas, *The auction algorithm: A distributed relaxation method for the assignment problem*, Annals of operations research, 14 (1988), pp. 105–123.

[4] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, The MIT Press, 2nd ed., 2001.

[5] G. Demange, D. Gale, and M. Sotomayor, *Multi-item auctions*, Journal of political economy, 94 (1986), pp. 863–872.

[6] J. Feigenbaum, S. Kannan, A. McGregor, S. Suri, and J. Zhang, *On graph problems in a semi-streaming model*, Theoretical Computer Science, 348 (2005), pp. 207–216.

[7] J. E. Hopcroft and R. M. Karp, *An n^5/2 algorithm for maximum matchings in bipartite graphs*, SIAM Journal on computing, 2 (1973), pp. 225–231.

[8] S. Kale and S. Tirodkar, *Maximum matching in two, three, and a few more passes over graph streams*, arXiv preprint arXiv:1702.02559, (2017).

[9] C. Konrad, *A simple augmentation method for matchings with applications to streaming algorithms*, in 43rd International Symposium on Mathematical Foundations of Computer Science (MFCS 2018), Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2018.

[10] J. Leskovec and A. Krevl, *SNAP Datasets: Stanford large network dataset collection*. `http://snap.stanford.edu/data`, June 2014.

[11] L. Lovász and M. D. Plummer, *Matching theory*, vol. 367, American Mathematical Soc., 2009.

[12] S. Micali and V. V. Vazirani, *An algorithm for finding maximum matching in general graphs*, in 21st Annual Symposium on Foundations of Computer Science (sfcs 1980), 1980, pp. 17–27.

[13] D. P, *Auction algorithms for network flow problems: A tutorial introduction*, Computational optimization and applications, 1 (1992), pp. 7–66.

[14] J. Siek, L.-Q. Lee, A. Lumsdaine, et al., *The boost graph library*, vol. 243, Pearson India, 2002.