Maximum Weighted Matching VS Auction Algorithm

Ca' Foscari University of Venice
Department of Environmental Sciences, Informatics and Statistics

ADVANCED ALGORITHMS AND PROGRAMMING METHODS - 2 [CM0470-2]
Academic Year 2021 - 2022

Student Zuliani Riccardo 875532

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

A key problem that managers have to face with is how to allocate scarce resources among various activities or projects. Linear programming, or LP, is a method of allocating resources in an optimal way. In the term **linear programming**, *programming* refers to mathematical programming, or in other words refers to a planning process that allocates resources (like materials, machines, capital etc.) in the best possible way so that costs are minimized or profits are maximized. In this context we can find three entities that characterize our world:

- **Resources** are viewed as *decision variables*.
- **The criterion** for selecting the best decision variables in order to minimize the costs or maximize the profits as *objective function*.
- **Limitations** on resource availability as *constraint set*.

One of the most important and successful applications of quantitative analysis for solving business problems has been in the physical distribution of products, commonly referred to as **transportation problems**. Where the purpose is to minimize the cost of shipping goods from one location to another so that the needs of each arrival area are met and every shipping location operates within its capacity. Moreover, quantitative analysis has been used for many problems other than the physical distribution of goods, like to efficiently place employees at certain jobs within an organization. This last example sometimes is identified as **assignment problem** because, as the name says, we have to assign at each person only a single job likewise each task have to be assigned to only a single employ. [10]

In this report we analyse these two cited problems with a complete focus on the mathematical aspects. Further we describe two algorithms that aim to solve the assignment problem in the least possible time with the highest possible profit, **Maximum Weighted Matching** (MWM) and **Auction Algorithm** (AU). The following chapter explain the main key choice and goals of the application benchmark developed to study the difference in performance of both algorithms. And in the last chapter we discuss the obtained result explaining which of the two is the best, if there is a better one.

Chapter 2

T&A Problem

In this chapter we analyse the main problem that Maximum Weighted Matching and Auction Algorithm aim to solve. We are talking about the **Assignment Problem**, but before explaining the structure of this problem we have to specify where it comes from. It is a special case of the **Transportation Problem** which can be expressed by the formulation of a linear model, and can be solved using the **Simplex Algorithm**[1][2]. But because of its special structure it can be solved in a more efficient method. We inform the reader that this chapter is mostly taken by the following paper [14].

2.1 Transportation Problem

The transportation problem, how the name tell us, it deals with the *transportation* of any product from m origins, $O_1, ..., O_m$, to n destination, $D_1, ..., D_n$, with the goal of **minimize the total cost of distribution, or maximize the total profit**. Moreover we can note:

- **The origin** O_i has *supply* of a_i units, $i = 1, ..., m$.
- **The destination** D_j has *demand* for b_j units to be delivered from the origins, $j = 1, ..., n$.
- c_{ij} is the **cost per distribution** from the origin O_i to the destination D_j, $i = 1, ..., m$, $j = 1, ..., n$.

In mathematical terms the problem can be viewed as finding a set of x_{ij}'s, $i = 1, ..., m$, $j = 1, ..., n$, to meet **supply** and **demand** requisites at the minimum distribution cost. So the corresponding linear programming problem is the following one:

[1] https://ocw.ehu.eus/pluginfile.php/40932/mod_resource/content/1/2_Simplex.pdf
[2] https://optimization.cbe.cornell.edu/index.php?title=Simplex_algorithm

$$\begin{aligned} \min \quad & z = \sum_{i=1}^{m}\sum_{j=1}^{n} c_{ij}x_{ij} \\ \text{s.t.} \quad & \sum_{j=1}^{n} x_{ij} \leq a_i, \quad i = 1, ..., m \\ & \sum_{i=1}^{m} x_{ij} \geq b_j, \quad j = 1, ..., n \\ & x_{ij} \geq 0, \quad i = 1, ..., m, \quad j = 1, ..., n \end{aligned} \tag{2.1}$$

The problem is to determine x_{ij}, the number of units to be transported from O_i to D_j, so that supplies will be consumed and demands satisfied at an overall minimum cost.

- The first $\boldsymbol{m}$ **constraints** correspond to the supply limits and they express that the supply of commodity units available at each origin must not be exceeded.
- The second $\boldsymbol{n}$ **constraints** ensure that the commodity unit requirements at destination will be satisfied.
- Moreover the **decision variables** are defined positive, since they represent the number of commodity units transported.

Further we can analyse the transportation problem in its standard form:

$$\begin{aligned} \min \quad & z = \sum_{i=1}^{m}\sum_{j=1}^{n} c_{ij}x_{ij} \\ \text{s.t.} \quad & \sum_{j=1}^{n} x_{ij} = a_i, \quad i = 1, ..., m \\ & \sum_{i=1}^{m} x_{ij} = b_j, \quad j = 1, ..., n \\ & x_{ij} \geq 0, \quad i = 1, ..., m, \quad j = 1, ..., n \end{aligned} \tag{2.2}$$

Example. Now we have a look on a clear example in order to make sure we acquire the structure of the problem. We take into account two pasty shop, P_1 and P_2, that daily make fresh cookies. These one are delivered to three supermarket: S_1, S_2 and S_3. The *supplies of pasty shop*, the *demands of supermarket* and the *per unit transportation cost* are displayed in the following graph:

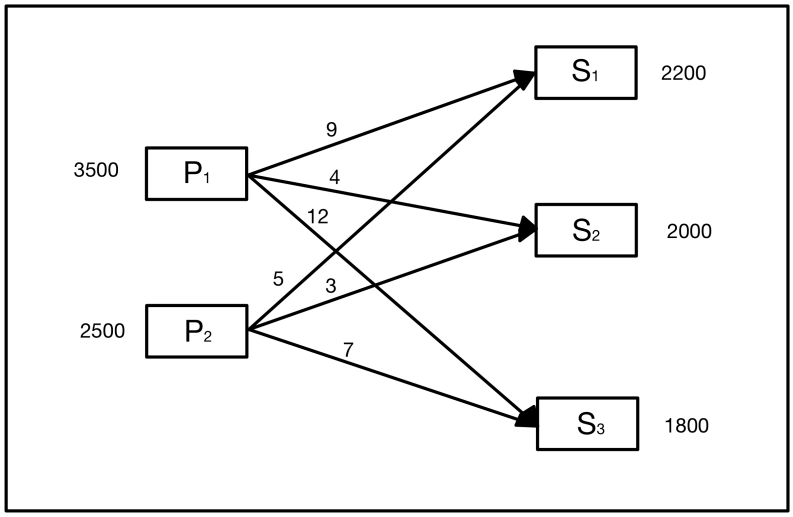

Figure 2.1: Graph representation of the transportation problem

Moreover we define the following decision variable:

- x_{ij}: the number of cakes to be distributed from the pasty shop P_i to the supermarket S_j, where $i = 1, 2$ and $j = 1, 2, 3$.

The corresponding linear model is the following one:

$$\begin{aligned} \min \quad & z = 9x_{11} + 4x_{12} + 12x_{13} + 5x_{21} + 3x_{23} + 7x_{23} \\ \text{s.t.} \quad & x_{11} + x_{12} + x_{13} = 3500 \\ & x_{21} + x_{22} + x_{23} = 2500 \\ & x_{11} + x_{21} = 2200 \\ & x_{12} + x_{22} = 200 \\ & x_{13} + x_{23} = 1800 \\ & x_{11}, x_{12}, x_{13}, x_{21}, x_{22}, x_{23} \geq 0 \end{aligned} \tag{2.3}$$

We can write the constraints in equation from because the total supply is equal to the demand. Moreover we rephrase the given example problem in matrix form.

$$\begin{aligned} \min \quad & z = (9, 4, 12, 5, 3, 7) \begin{pmatrix} x_{11} \\ x_{12} \\ x_{13} \\ x_{21} \\ x_{22} \\ x_{23} \end{pmatrix} \\ \text{s.t.} \quad & \begin{pmatrix} 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x_{11} \\ x_{12} \\ x_{13} \\ x_{21} \\ x_{22} \\ x_{23} \end{pmatrix} = \begin{pmatrix} 3500 \\ 2500 \\ 2200 \\ 2000 \\ 1800 \end{pmatrix} \\ & x_{11}, x_{12}, x_{13}, x_{21}, x_{22}, x_{23} \geq 0 \end{aligned} \tag{2.4}$$

In any transportation problem the structure of its matrix depends on the number of origins and number of destination of the problem. In fact any transportation

problem having m origins and n destinations has the same matrix. The previous one has $m + n$ rows and $m \times n$ columns, and we can note its rank is $m + n - 1$, so any basis consists of $m + n - 1$ vectors. There are only two 1's in each matrix column vector a_{ij} and they are located in rows i and $m + j$, being 0 the rest of the values of the matrix column vector.

2.1.1 The matrix format of the transportation problem

The relevant data for any transportation problem can be summarized in a matrix format using a tableau called *the transportation costs tableau*, it displays:

- The origins with their supply.
- The destinations with their demand.
- The transportation per unit costs.

	D_1	D_1	$\cdots$	D_n	Supply
O_1	c_{11}	c_{12}	$\cdots$	c_{1n}	a_1
O_2	c_{21}	c_{22}	$\cdots$	c_{2n}	a_2
$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$	$\vdots$
O_m	c_{m1}	c_{m2}	$\cdots$	c_{mn}	a_m
Demand	c_1	b_2	$\cdots$	b_n	

Table 2.1: The general transportation costs tableau

The transportation costs tableau for our previous example is the following one:

	D_1	D_1	D_3	Supply
O_1	9	4	12	3500
O_2	5	3	7	2500
Demand	2200	200	1800	

Table 2.2: Example of transportation costs tableau

2.1.2 Theorems and definitions

As we previously said, the transportation problem is just a special type of linear programming problem. We can take advantage of its special structure to adapt the simplex algorithm to have a more efficient solution procedure.

Theorem. The necessary and sufficient condition for a transportation problem to have a solution is that the total demand equals the total supply.

Proof. According to the standard form of the transportation problem:

- Each of the supplies a_i satisfies the following constraints:

$$\sum_{j=1}^{n} x_{ij} = a_i \quad i = 1, ..., m$$

  And if we sum all the supplies we get the total amount of supply, which is:

$$\sum_{i=1}^{m}\sum_{j=1}^{n} x_{ij} = \sum_{i=1}^{m} a_i \tag{2.5}$$

- The demands b_j satisfy the following constraints:

$$\sum_{i=1}^{m} x_{ij} = b_j \quad j = 1, ..., n$$

  With total demand:

$$\sum_{j=1}^{n}\sum_{i=1}^{m} x_{ij} = \sum_{j=1}^{n} b_j \tag{2.6}$$

- We denote that the left hand side of 2.5 and 2.6 are equal, thus the mentioned equations are satisfied if and only if:

$$\sum_{i=1}^{m} a_i = \sum_{j=1}^{n} b_j$$

The theorem above states that, under the assumption that the *total supply* **equals** the *total demand*, the transportation problem always has a **feasible solution**.

But this is not always true, in fact we have to care the case when the total supply and the total demand do not match, in this case we have to adapt the problem before being solved.

Definition of Balanced Problem. A transportation problem is said to be balanced if:

$$\sum_{i=1}^{m} a_i = \sum_{j=1}^{n} b_j$$

As previously mentioned there is the the problem if the total demand and supply are not equal, this is called **Unbalanced Problem** and there are two possible cases:

1. **The demand exceeds the supply:** $\sum_{i=1}^{m} a_i < \sum_{j=1}^{n} b_j$

   In this case a dummy source or origin O_{m+1} is added to balance the model, thus its corresponding supply and unit transportation cost are the following:

   $$a_{m+1} = \sum_{j=1}^{n} b_j - \sum_{i=1}^{m} a_i$$

   $$c_{m+1,j} = 0, \quad j = 1, ..., n$$

   As the supply a_{m+1} coming from the dummy origin O_{m+1} does not exist, any destination receiving transportation units from the dummy origin will experience shortage, which means that its demand will not be correctly satisfied.

   Example. Considering the following unbalanced transportation problem:

	1	2	3	Supply
1	1	5	7	10
2	2	8	4	20
Demand	20	20	30	$70 \neq 30$

   Will be converted into the following balanced transportation problem:

	1	2	3	Supply
1	1	5	7	10
2	2	8	4	20
3	0	0	0	(70 - 30) = 40
Demand	20	20	30	70 = 70

2. **The supply exceeds the demand:** $\sum_{i=1}^{m} a_i > \sum_{j=1}^{n} b_j$ As in the previous step we add a dummy destination D_{n+1} to the problem such that its demand and unit transportation costs are:

   $$b_{n+1} = \sum_{i=1}^{m} a_i - \sum_{j=1}^{n} b_j$$

   $$c_{i,n+1} = 0 \quad i = 1, ..., m$$

   The demand of the dummy destination is the difference between the total supply and the total demand. Unit transportation costs from any origin to the dummy destination are considered to be zero, because they do not correspond to real product transportations.

Example. Considering the following unbalanced transportation problem:

	1	2	3	Supply
1	2	5	4	50
2	5	9	5	50
Demand	20	20	20	$60 \neq 100$

Will be converted into the following balanced transportation problem:

	1	2	3	4	Supply
1	2	5	4	0	50
2	5	9	5	0	50
Demand	20	20	20	(100 - 60) = 40	100 = 100

Theorem. A balanced transportation problem always has a feasible solution.

Proof. Consider the linear model in standard form of a balanced transportation problem. The previous mentioned theorem told us it has a solution, so let's find out. Consider:

$$T = \sum_{i=1}^{m} a_i = \sum_{j=1}^{n} b_j$$

$x_{ij} = \frac{a_i b_j}{T}$ verifies the constraints $i = 1, ..., m$, $j = 1, ..., n$ and is consequently a solution. Moreover it is feasible because $x_{ij} \geq 0$ holds for $i = 1, ..., m$, $j = 1, ..., n$.

2.1.3 Improvement of a basic feasible solution

The dual transportation problem is used to find an improved basic feasible solution. Consider a balanced transportation problem:

$$\begin{aligned}
\min \quad & z = \sum_{i=1}^{m} \sum_{j=1}^{n} c_{ij} x_{ij} \\
\text{s.t.} \quad & \sum_{j=1}^{n} x_{ij} = a_i, \quad i = 1, ..., m \\
& \sum_{i=1}^{m} x_{ij} = b_j, \quad j = 1, ..., n \\
& x_{ij} \geq 0, \quad i = 1, ..., m, \quad j = 1, ..., n
\end{aligned}$$

We denote $u_1, ..., u_m$ and $v_1, ..., v_n$ the dual variables, which leads to the following dual model:

$$\begin{aligned}
\max \quad & G = \sum_{i=1}^{m} a_i u_i + \sum_{j=1}^{n} b_j v_j \\
\text{s.t.} \quad & u_i + v_j \leq c_{ij} \quad i = 1, ..., m, \quad j = 1, ..., n \\
& u_i, v_j : \text{unrestricted} \quad i = 1, ..., m, \quad j = 1, ..., n
\end{aligned} \tag{2.7}$$

Example. Considering the initial transportation problem, let us calculate the corresponding dual model.

$$\begin{aligned}
\min \quad & z = 9x_{11} + 4x_{12} + 12x_{13} + 5x_{21} + 3x_{23} + 7x_{23} \\
\text{s.t.} \quad & x_{11} + x_{12} + x_{13} = 3500 \\
& x_{21} + x_{22} + x_{23} = 2500 \\
& x_{11} + x_{21} = 2200 \\
& x_{12} + x_{21} = 2000 \\
& x_{13} + x_{23} = 1800 \\
& x_{11}, x_{12}, x_{13}, x_{21}, x_{22}, x_{23} \geq 0
\end{aligned}$$

The dual variables are denoted by u_1, u_2, v_1, v_2 and v_3, which leads to the following dual model:

$$\begin{aligned}
\max \quad & G = 3500u_1 + 2500u_2 + 2200v_1 + 2000v_2 + 1800v_3 \\
\text{s.t.} \quad & u_1 + v_1 \leq 9 \\
& u_1 + v_2 \leq 4 \\
& u_1 + v_3 \leq 12 \\
& u_2 + v_1 \leq 5 \\
& u_2 + v_2 \leq 3 \\
& u_2 + v_3 \leq 7 \\
& u_i, v_j : \text{unrestricted}
\end{aligned}$$

The method used to compute the optimal solution of a transportation problem is actually a direct adaptation of the simplex method.

2.2 Assignment Problem

The assignment problem is a special case of the transportation problem. It deals with assigning n origins (workers) to n destinations (jobs or machines) with the goal of determining the minimum cost assignment. We have that each *origin* must be assigned to one and only one *destination*, and likewise each *destination* must be assigned only to one *origin*. Moreover c_{ij} represents the *cost* of assigning the origin O_i to the destination D_j, for $i, j = 1, ..., n$.

The decision variable are defined like the following system:

$$x_{ij} = \begin{cases} 1 & \text{if } O_i \text{ is assigned to } D_j \\ 0 & \text{otherwise} \end{cases}$$

A linear model in standard form for the assignment problem is given by the

following optimization problem:

$$\begin{aligned} \min \quad & z = \sum_{i=1}^{n}\sum_{j=1}^{n} c_{ij}x_{ij} \\ \text{s.t.} \quad & \sum_{j=1}^{n} x_{ij} = 1, \quad i = 1, ..., n \\ & \sum_{i=1}^{n} x_{ij} = 1, \quad j = 1, ..., n \\ & x_{ij} = 0, 1 \quad i, j = 1, ..., n \end{aligned}$$

- The first n constraints ensure that each origin is assigned to one and only one destination.
- The following n constraints ensure that each destination is assigned to one and only one origin.

As before for the general transportation problem if the number of origins and destination are not equal, the assignment problem is *unbalanced* and in order to fix it to transforming it to a *balanced* assignment problem we can add as many dummy origins or destination as necessary. The cost of the dummy destinations or origins are as before equal to zero. Finally the relevant data for any assignment problem can be summarized in a compact $n \times n$ matrix using a tableu called *the assignment tableau*.

	D_1	D_2	$\cdots$	D_n
O_1	c_{11}	c_{12}	$\cdots$	c_{1n}
O_2	c_{21}	c_{22}	$\cdots$	c_{2n}
$\vdots$	$\vdots$	$\ddots$	$\ddots$	$\vdots$
O_n	c_{n1}	c_{n2}	$\cdots$	c_{nn}

Table 2.3: The general assignment cost tableau

We can mention two type of Assignment Problem in depends on the number of possible choice that every worker could take.

- If every worker can choose the preferred machine on the full set of machines, we call this as an **Assignment Problem on a Complete Bipartite Graph**.
- If at least a worker can not choose the preferred machine of the full set but in a subset of machines, we call this as an **Assignment Problem on a Incomplete Bipartite Graph**.

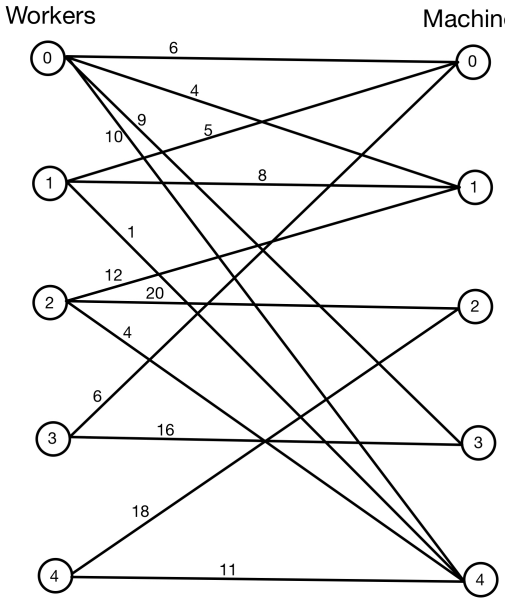

Figure 2.2: Complete bipartite graph for Assignment Problem

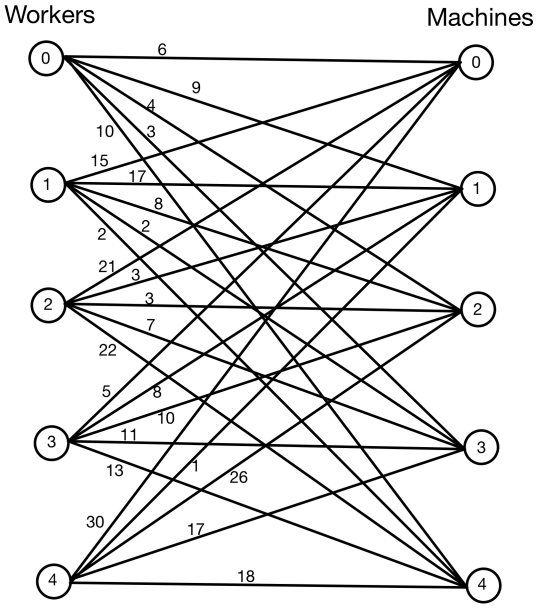

Figure 2.3: Incomplete bipartite graph for Assignment Problem

Chapter 3

The Algorithms

In this chapter we analyse the two algorithm taken into account that aim to solve the assignment problem. Firstly we discuss the theory behind Maximum Weighted Matching and then its Boost Graph Library implementation, moreover we deal with the behaviour of the Auction Algorithm and its pseudocode.

3.1 Maximum Weighted Matching

3.1.1 Primal/Dual Problem for weighted matching in Bipartite Graphs

The problem of maximum weighted matching matching in a given bipartite graph $G(A \cup B), E$ with edge weights $w_{ij} \geq 0$ is to find out a maximum weight perfect matching. Before going deeply in the discussion we have to remark some general definition that are useful to better explain the algorithm.

General definitions. [9]

- In graph theory, a **matching** is a subset of edges such that none of the selected edges share a common vertex.
- A **maximum cardinality matching** is a matching that contains the largest possible number of edges (or equivalently, the largest possible number of vertices). A perfect matching is a matching which covers all vertices.
- With respect to a weighted graph, a **maximum weight matching** is a matching for which the sum of the weights of the matched edges is as large as possible.

Augmenting path. In each stage of the algorithm we have a matching M which initially is empty.

$$\text{A } \textbf{vertex } i \text{ is } \begin{cases} \text{matched} & \text{if there is a edge } (i,j) \text{ in } M \\ \text{single} & \text{otherwise} \end{cases}$$

$$\text{A } \textbf{edge } i \text{ is } \begin{cases} \text{matched} & \text{if it is in } M \\ \text{unmatched} & \text{otherwise} \end{cases}$$

An **alternating path** is a simple path, such that every other edge on it is matched. An **augmenting path** is an alternating path between two single vertices. It must be of odd length, and in the bipartite case (so our case) its endpoints must be of different part.
Definition. The matching M has maximum cardinality if and only if there is no augmenting path with respect to M. [6]

If we define an indicator variable x_{ij} for each edge between $i \in A$ and $j \in B$ we want to maximize the following objective function:

$$\sum_{i,j} x_{ij} w_{ij}$$

Therefore **maximum weighted matching** has the following formulation of Integer Program Problem [13]:

$$\begin{aligned} \max \quad & \sum_{i,j} x_{ij} w_{ij} \\ \text{s.t.} \quad \forall \quad a_i, \quad & \sum_{j} x_{ij} \leq 1 \\ \forall \quad b_j, \quad & \sum_{i} x_{ij} \leq 1 \\ & x_{ij} \in \{0,1\} \end{aligned}$$

By relaxing the integrality constraint of the previous Integer Programming Problem $x_{ij} \geq 0$ we end up in a Linear Programming Problem.

A weighted vertex cover for graph with weighted edges is a function $y : V \to \mathbb{R}^+$ such that for all edges $e = uv : y_u + y_v \geq w_{uv}$.

Note that for any *weighted matching* M and *vertex cover* Y:

$$\sum_{e \in M} w(e) = w(M) \leq C(Y) = \sum_{y \in V} y_v \tag{3.1}$$

Where $C(Y)$ is the **cost of the vertex cover** Y. So the *vertex cover* is actually **the dual problem** of *maximum matching*. [13]

If we consider the dual program to the linear program relaxation of the maximum matching problem we obtain the formulation of the linear program relaxation of the

weighted vertex cover problem:

$$\begin{aligned} \min \quad & \sum_i y_i \\ \text{s.t.} \quad & \forall e = a_i b_j : \quad y_{a_i} + y_{b_j} \geq w_{ij} \\ & y_i \geq 0 \end{aligned}$$

According to equation 3.1 for any vertex cover Y, $C(Y)$ is an upper bound for $w(M)$ for any matching M. So if a vertex cover is founded and has the same value as a matching both are optimal solution.

Lemma. For a *perfect matching* M and a *weighted vertex cover* y:

$$C(y) \geq W(M)$$

Also $C(y) = W(M)$ iff M consists i of edges $a_i b_j$ such that $y_i + y_j = w_{ij}$. In this case M is optimum. [13]

3.1.2 The Primal / Dual Algorithm

The algorithm starts with $M = \emptyset$ and a trivial feasible solution for the weighted vertex cover which is the following one:

$$\forall a_i \in A; \quad y_{a_i} = \max_j w(a_i b_j)$$

$$\forall b_j \in B; \quad y_{b_j} = 0$$

At any iteration of the algorithm we build an equality graph defined as $G_y = (A \cup B, E_y)$ and it is based on the y values such that it only contains *tight edges*

$$a_i b_j \in E_y \iff y_i + y_j = w_{ij}$$

And let define $y_{a_i} + y_{b_j} - w_{ij}$ the *excess* of $a_i b_j$.

Observation. If M is a perfect matching in G_y then

$$W(M) = \sum_{a_i \in A} y_{a_i} + \sum_{b_j \in B} y_{b_j}$$

And by previous lemma that matching in G is an optimum solution. [13]

Based on this observation, the goal of the algorithm is to find a perfect matching in the equality graph. For this we update y to make more edges tight to be added to E_y (until it contains a perfect matching) while keeping y a vertex cover. Now we present an algorithm to add an edge to equality graph.

Suppose we are at some iteration of the algorithm and M is a maximum matching in G_y but is not perfect. Construct digraph $D = (A \cup B, E')$, where E' is the union of the edges of M directed from B to A and edges of $E - M$ directed from A to B. Let L be a set of nodes accessible from any exposed node in A. Recall that

$C^* = (A - L) \cup (B \cap L)$ is a vertex cover. Therefore there is no edge between $A \cup L$ and $B - L$. However we know that we start with a complete graph G. Thus there are edges in G between $A \cup L$ and $B - L$ but they are not in G_y; which means all those edges have positive excess. We update the y values to make one of these edges go tight. Let:

$$\epsilon = min\{y_{a_i} + y_{b_j} - w_{ij} \quad \text{s.t.} \quad a_i \in A \cap L, \quad b_j \in B - L\}$$

be the minimum excess value of all such edges. Then we update vertex y values to tighten the edges with ϵ excess value by defining:

$$y_{a_i} = \begin{cases} y_{a_i} & a_i \in A - L \\ y_{a_i} - \epsilon & a_i \in A \cap L \end{cases}$$

$$y_{b_j} = \begin{cases} y_{b_j} & b_j \in B - L \\ y_{b_j} + \epsilon & b_j \in B \cap L \end{cases}$$

Note that by this change every edge that was in G_y remains tight. Also by the choice of ϵ, no edge constraint is going to be violated, so y remains a vertex cover. Furthermore, at least one edge between $A \cap L$ and $B - L$ goes tight and therefore is added to G_y. We can repeat this operation until either G_y has a perfect matching, or there is no edges left between $A \cap L$ and $B - L$. At this point the solution y is also an optimum vertex cover. The following pseudo code summarize the Primal/Dual Algorithm:

Algorithm 1 Maximum Weighted Bipartite Matching Algorithm, Primal-Dual Method

```
for each a_i ∈ A do
    y_{a_i} = max_{b_j ∈ B} w(a_i b_j)
    for each b_j ∈ B do
        y_{b_j} = 0
        Build graph G_y and let M be a maximum matching in G_y
        Construct Digraph D
        repeat
            let L be the set of nodes (in D) accessible from any exposed node in A
            ε = min{y_{a_i} + y_{b_j} − w_{ij} : a_i ∈ A ∩ L, b_j ∈ B − L}
            Decrease y_{a_i} for each a_i ∈ A∩L by ε and increase y_{b_j} for each b_j ∈ B−L by ε
            Add the tight edges to G_y and recompute matching M
        until M is a perfect matching
    end for
end for
```

Theorem. Given a complete bipartite graph G with edge weights $w(e)$, the primal/dual method finds a maximum matching and a minimum vertex cover in time $O(n^3)$

3.1.3 The BGL Algorithm

In the Boost Graph Library there are two implementation of Maximum Weighted Matching, the brute force version and the non once, we decide to adopt the non brute force version since it is quicker.

Edmonds proved that for any graph, the maximum number of edges in a matching is equal to the minimum capacity of an odd-set cover (Equation 3.1); this further enable us to prove a max-min duality theorem for weighted matching, as saw in the previous section. [4]

This matching duality theorem gives an indication of how the matching problem should be formulated as a linear programming problem. That is, the theorem suggests a set of linear inequalities which are satisfied by any matching, and it is anticipated that these inequalities describe a convex polyhedron with integer vertices corresponding to feasible matchings. [7]

Algorithm 2 Boost Graph Library Maximum Weighted Matching implementation

0. Start with an empty matching and initialize dual variables as a half of maximum edge weight.
1. **(Labeling)** Root an alternate tree at each exposed node, and proceed to construct alternate trees by labeling, using only edges with zero slack value. If an augmenting path is found, go to step 2. If a blossom is formed, go to step 3. Otherwise, go to step 4.
2. **(Augmentation)** Find the augmenting path, tracing the path through shrunken blossoms. Augment the matching, correct labels on nodes in the augmenting path, expand blossoms with zero dual variables and remove labels from all base nodes. Go to step 1.
3. **(Blossoming)** Determine the membership and base node of the new blossom and supply missing labels for all non-base nodes in the blossom. Return to step 1.
4. **(Revision of Dual Solution)** Adjust the dual variables based on the primal-dual method. Go to step 1 or halt, accordingly.

[7]

Complexity

Let m and n be the number of edges and vertices in the input graph, respectively. The time complexity for *maximum_weighted_matching* is $O(n^3)$. Note that the best known time complexity for maximum weighted matching in general graph is $O(nm+n^2log(n))$ given by [5] but relies on an efficient algorithm for solving nearest ancestor problem on trees, which is not provided in Boost C++ libraries. [7]

3.2 Auction Algorithm

The algorithm operates like an auction whereby unassigned person bid simultaneously for objects, thereby raising their prices. Once all bids are in, object are awarded to the highest bidder. This algorithm makes use of the primal dual problem, in fact the variables of the dual problem may be viewed as the prices of the objects and are adjusted as the algorithm progresses. Like in a real auction, a person's bid is required to be higher than the current price of the object and this provides the mechanism for increasing the object prices. [2]

3.2.1 Assignment by Naive Auction

In the classical *symmetric assignment* problem there are n persons and n objects that we have to match on a one-to-one basis. There is a benefit a_{ij} for matching person i with object j and we want to assign persons to objects so as to maximize the total benefit. We are given a set $\mathcal{A}$ of pairs (i, j) that can be matched.

- For each **person** i, we denote by $A(i)$ the set of objects that can be matched with i
$$A(i) = \{j | (i, j) \in \mathcal{A}\}$$
- And for each **object** j, we denote by $B(j)$ the set of persons that can be matched with j
$$B(j) = \{i | (i, j) \in \mathcal{A}\}$$

By an *assignment* we mean a set S of person-object pairs (i, j) such that each **person** i and each **object** j is involved in at most one pair from S. If the number of pairs in S is n, so that every person is assigned to a distinct object, we say that S is **feasible**; otherwise S is said to be **infeasible**. If a feasible assignment exists the problem is said to be feasible, and otherwise it is said to be infeasible. We seek a feasible assignment [a set of person-object pairs $(1, j_1), ..., (n, j_n)$ from A, such that the objects $j_1, ..., j_n$ are all distinct], which is optimal in the sense that it maximizes the total benefit
$$\sum_{i=1}^{n} a_{ij_i}$$

To develop an intuitive understanding of the auction algorithm, it is helpful to introduce an economic equilibrium problem that turns out to be equivalent to the assignment problem.

Consider the possibility of matching the n objects with the n persons through a market mechanism, viewing each person as an economic agent acting in his/her own best interest. Suppose that object j has a price p_j and that the person who receives the object must pay the price p_i. Then, the (net) value of object j for person i is $a_{ij} - p_j$ and each person i would logically want to be assigned to an object j_i with maximal value, that is, with
$$a_{ij_i} - p_{j_i} = \max_{j \in A(i)} \{a_{ij} - p_j\}$$

The economic system would then be at equilibrium, in the sense that no person would have an incentive to act unilaterally, seeking another object, like a Nash Equilibrium. [3]

The Naive Auction Algorithm

Let us consider a natural process for finding an equilibrium assignment and price vector. The naive auction algorithm proceeds in iterations and generates a sequence of price vectors and assignments. At the beginning of each iteration, the Complementary Slackness (CS) condition:

$$a_{ij_i} - p_{j_i} = \max_{j \in A(i)} \{a_{ij} - p_j\}$$

is satisfied for all pairs (i, j_i) of the assignment. If all persons are assigned, the algorithm terminates. Otherwise a nonempty subset I of persons i that are unassigned is selected and the following computations are performed.

Let I an non empty subset of person that are not assigned.
BIDDING PHASE: Each person $i \in I$ finds an object j_i which offers maximal value, that is:

$$j_i \in \arg\max_{j \in A(i)} \{a_{ij} - p_j\}$$

and computes a bidding increment of:

$$\gamma_i = v_i - w_i$$

where v_i is the best object value:

$$v_i = \max_{j \in A(i)} \{a_{ij} - p_j\}$$

and w_i is the second best object value

$$w_i = \max_{j \in A(i), j \neq j_i} \{a_{ij} - p_j\}$$

If j_i is the only object in $A(i)$, we define w_i, to be $-\infty$ or, for computational purpose, a number that is much smaller than v_i. [3]

ASSIGNMENT PHASE: Each object j that is selected as best object by a nonempty subset $P(j)$ of persons in I, determines the highest bidder:

$$i_j = \arg\max_{i \in P(j)} \gamma_i$$

raises its prices by the highest bidding increment $\max_{i \in P(j)} \gamma_i$, and gets assigned to the highest bidder i_j. The person that was assigned to j at the beginning of the iteration (if any) becomes unassigned.
The algorithm continues with a sequence of iterations until all persons have an assigned object. [3]

Note that γ_i cannot be negative since $v_i \geq w_i$ so that the object prices tend to increase. In fact, when i is the only bidder, γ_i is the largest bidding increment for which CS is maintained following the assignment of i to his/her preferred object. Just as in real auction, bidding increments and price increases spur competition by making the bidder's own preferred object less attractive to other potential bidders.

3.2.2 ϵ-Complementary Slackness and Auction Algorithm

Unfortunately, the naive auction algorithm does not always work. The difficulty is that the bidding increment y_i is zero when more than one object offers maximum value for the bidder i. As a result, a situation may be created where several persons contest a smaller number of equally desirable objects without raising their prices, thereby creating a never ending cycle.

To break such cycles, we introduce a perturbation mechanism, motivated by real auctions where each bid for an object must raise the object's price by a minimum positive increment, and bidders must on occasion take risks to win their preferred objects. In particular, let us fix a positive scalar ϵ and say that an assignment and a price vector p satisfy ϵ-complementary slackness (or ϵ-CS for short) if: [3]

$$a_{ij_i} - p_{j_i} \geq \max_{j \in A(i)} \{a_{ij} - p_j\} - \epsilon$$

for all assigned pairs (i, j_i).

In other words, **to satisfy ϵ-CS, all assigned persons must be assigned to objects that are within ϵ of being best.** [3]

The Auction Algorithm

We now reformulate the previous auction process so that the bidding increment is always at least equal to ϵ. The resulting method, the auction algorithm, is the same as the naive auction algorithm, except that the bidding increment γ_i, is:

$$\gamma_i = v_i - w_i + \epsilon \quad \text{rather than} \quad \gamma_i = v_i - w_i$$

With this choice, the ϵ-CS condition is satisfied. Smaller increments γ_i would also work as long as $\gamma_i \geq \epsilon$, but using the largest possible increment accelerates the algorithm. This is consistent with experience from real auctions, which tend to terminate faster when the bidding is aggressive. [3]

It can be shown that this reformulated auction process terminates in a finite number of iterations, necessarily with as feasible assignment and a set of prices that satisfy ϵ-CS. To see this for the case of fully dense problem. Note that if an object receives a bid in k iterations, its price must exceed its initial price by at least of $k\epsilon$. Thus, for sufficiently large k, the object will become *"expensive"* enough to be judged *"inferior"* to some object that has not received a bid so far. It follows an object can receive a bid in a limited number of iterations while some other object still has not yet received any bid. On the other hand, once all objects receive at least one bid, the auction terminates. This the auction algorithm must terminate, and in fact the preceding argument shows that, for the case of zero initial prices, the total number of iterations in which an object receive a bid is more than: [3]

$$\frac{\max_{(i,j)} |a_{ij}|}{\epsilon}$$

If each iteration involves a bid by a single person, the total number of iterations is no more than n times the preceding quantity, and since each bid requires $O(n)$

operations, the running time of the algorithm is:

$$O(n^2 \max_{(ij)} |a_{ij}|/\epsilon)$$

When the auction algorithm terminates, we have an assignment satisfying ϵ-CS, but is this assignment optimal?

The answer here depends strongly on the size of ϵ. In a real auction, a prudent bidder would not place an excessively high bid for fear that he/she might win the object at an unnecessarily high price. Consistent with this intuition, we can show that if ϵ is *small*, then the final assignment will be **"almost optimal."** In particular, the following proposition shows that *the total cost of the final assignment is within $n\epsilon$ of being optimal.* The idea is that when a feasible assignment and a set of prices satisfy $\epsilon - CS$, they also satisfy CS for a slightly perturbed problem where all costs a_{ij} are the same as before, except for the costs of the n assigned pairs, which are modified by an amount no more than ϵ.

Proposition 1: A feasible assignment satisfying ϵ-complementary slackness together with some price vector is within $n\epsilon$ of being optimal. [3]

Suppose now that the costs a_{ij} are all integer, which is the typical practical case (if a_{ij} are rational numbers, they can be scaled up to integer by multiplication with a suitable common number). Then, the total benefit of any assignment is integer, so if $n\epsilon < 1$, any complete assignment that is within $n\epsilon$ of being optimal must be optimal. It follows, that if $\epsilon < 1/n$, and the benefits a_{ij} are all integer, then the assignment obtained upon termination of the auction algorithm is optimal.

Proposition 2: Consider a feasible assignment problem with integer benefits a_{ii}. If

$$\epsilon < \frac{1}{n}$$

the auction algorithm terminates in a finite number of iterations with an **optimal assignment**. [3]

Thus in our first implementation of the Auction Algorithm we adopted

$$\epsilon = \frac{1}{n+1}$$

with n equal to the number of vertices per part.

ϵ-Scaling

The amount of work needed for the auction algorithm to terminate can depend strongly on the value of ϵ and on the maximum absolute object benefit C given by:

$$C = \max_{(i,j)\in A} |a_{ij}|$$

Basically, for many types of problems, the number of iterations up to termination tends to be proportional to C/ϵ as argued earlier for fully dense problems. This can

also be seen from the Figure 3.1, where the number of iterations up to termination is roughly C/ϵ, starting from zero initial prices.

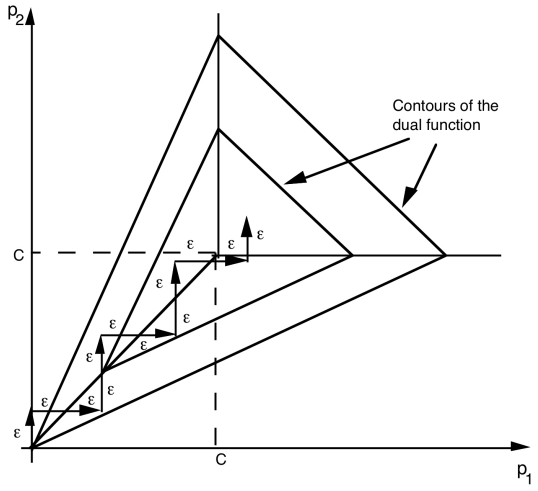

Figure 3.1: A sequence of prices p_1 and p_2 generated by the auction algorithm

For small ϵ, the method is susceptible to **"price wars"**, that is, protracted sequences of small price rises resulting from groups of persons competing for a smaller number of roughly equally desirable objects.

The preceding observations suggest the idea of ϵ-scaling, which consists of applying the algorithm several times, starting with a large value of ϵ and successively reducing ϵ up to an ultimate value that is less than some critical value. Typical ϵ-reduction factors after each algorithm phase are of the order of 1/4 to 1/10. Each application of the algorithm provides good initial prices for the next application. ϵ-scaling was suggested in the original proposal of the auction algorithm [2], based on extensive experimentation, which established its effectiveness for many types of assignment problems. In particular, e-scaling is typically beneficial for sparse problems. [3]

Pseudo code of Original Auction Algorithm

Algorithm 3 Auction Algorithm implementation

```
1: I set of Items
2: UB set of Unassigned Bidders
3: n the number of vertices per part
4: ϵ = 1/(n+1)
5: while UB ≠ 0 do
6:     for each ub ∈ UB do
7:         Search for the best item j_ub which offers maximal value in depends on
   the first best and the second best item
8:         Compute the bid γ_ub = v_ub − w_ub + ϵ
9:         if γ_ub ≥ than the actual bid for item j then
10:            Save the bidder ID ub and its bid γ_ub
11:        end if
12:    end for
13:    for each i∈ I do
14:        if i obtained a bid then
15:            Update the cost of i p_i = p_i + γ_i
16:            if i has been already assigned to an other bidder s then
17:                Remove the assignment and add s to UB
18:            end if
19:            Assign i to the highest bidder
20:        end if
21:    end for
22: end while
```

Pseudo code of ϵ-Scaling Auction Algorithm

Algorithm 4 ϵ-Scaling Auction Algorithm implementation

```
1: I set of Items
2: UB set of Unassigned Bidders
3: ε = 1.0
4: v the number of vertices per part
5: ε_scaling be the ε scaling factor: 1/4 ≤ ε_scaling ≤ 1/10
6: while ε > 1.0/v do
7:     Reset UB and I except all the price costs
8:     while UB ≠ 0 do
9:         for each ub ∈ UB do
10:            Search for the best item j_ub which offers maximal value in depends on
       the first best and the second best item
11:            Compute the bid γ_ub = v_ub − w_ub + ε
12:            if γ_ub ≥ than the actual bid for item j then
13:                Save the bidder ID ub and its bid γ_ub
14:            end if
15:        end for
16:        for each i∈ I do
17:            if i obtained a bid then
18:                Update the cost of i p_i = p_i + γ_i
19:                if i has been already assigned to an other bidder s then
20:                    Remove the assignment and add s to UB
21:                end if
22:                Assign i to the highest bidder
23:            end if
24:        end for
25:    end while
26:    ε = ε ∗ ε_scaling
27: end while
```

Complexity

For integer data, it can be shown that the worst-case running time of the auction algorithm using scaling and appropriate data structures is $O(nA\log(nC))$. Based on experiments, the running time of the algorithm for randomly generated problems seems to grow proportionally to something like $A\log n$ or $A\log n\log(nC)$. [3]

Chapter 4

Application Benchmark

In this chapter we discuss the application goals and structure, but also the **two phases** of learning process that brings us to the final solution. We inform the reader that the whole project can be viewed and cloned via the following GitHub Repository.

4.1 Application Overview

As mentioned above, the main objective of the project is to compare the performances of the Maximum Weighted Matching and the Auction Algorithm in solving the Assignment Problem in a bipartite graph. To achieve this, we develop a console application to run algorithms for a given number of graphs, which depends on user input.

The structure of the console application is pretty simple, after have been launched we have to specify (Figure A.1):

- The **VERBOSE** option to *ON (1)* or *OFF (0)* to indicate if we want to view in full detail the actual edges weights and the final result of all variables used by the auction algorithm. Like shown in Figure A.2 and A.3 [12].
- Next we have to choose on what **type of bipartite graph** we would like to deal with, *Complete (1)* or *Inomplete (0)*.
- And finally the last choice is the **range of vertices per part** of our graphs. So the number of starting and ending vertices per part.

After having filled these four inputs the application runs the following algorithms on the same graph starting from the initial number of vertices up to the ending number:

- The **Maximum Weighted Matching**.
- The **Original Auction Algorithm** with $\epsilon = \frac{1}{n+1}$.
- The **ϵ-Scaling Auction Algorithms** with ϵ scaling from 1/4 up to 1/10.

At the end of all methods it stores some usual information for further analysis. In particular this information include: the number of vertices, the execution time, the total weight cost of all strategies and the number of iterations of the Auction

Algorithms. Once all the graphs are examined a *.csv* file named like so: **complete / incomplete + starting number of vertices + ending number of vertices + datetime .csv** is stored into the */results* directory in order to save all the useful information.

4.2 Application Timeline

The timeline of the application development can be summarized in two main group choices, the first one describes the initial development and struggles, while the second one is characterized by a clarification of the requirements that move the application to another version.

4.2.1 1° Phase

In the First Phase we have decided to develop the Maximum Weighted Matching and the Original Auction Algorithm to work only with Assignment Problem in Complete Bipartite Graphs. We opted for this choice because at the beginning all the scientific papers analyzed referred to the Assignment Problem using that type of graph.

Knowing the structure of the future graphs we started to develop the general structure of the application. Firstly we focused on the user input, such as the starting and ending number of vertices per part, and then most importantly the actual generation of the randomized complete bipartite graph. This last feature was implemented using the *std::uniform_int_distribution* from the C++ standard library in order to randomly decide the weights of the connection between vertices of different parts.

Then we moved on to the application of a series of graphs to the Boost implementation of Maximum Weighted Matching. Initially, after seeing many different examples on the Internet, we decided to modify the example provided by the Boost Team[1]. In the next step we applied the MWM with integer weights and the overall implemented version was working pretty well. Out of curiosity we decided to switch to double edge weights, and here problems started to emerge. The point was that the MWM Boost implementation after a certain number of vertices per part started an infinite loop. This problem had been already reported by other users in the Stackoverflow Community[2] and also in the Official GitHub Boost repository[3]. More interesting it has been proved that the brute force version of MWM works with the float edge type with the drawback of a enormous amount of time to finish the examination of a single graph. For example the 14-vertices per part case will take 30 days, 15 would run 1.31 years, not the best solution [11]. A little improvement was achieved by changing the edge type from *float*, that is the type with the least possible precision, to *long double* but it still was returning inconsistent performance. The trick that helps us to overcome this curious problem was to use integral weights, with the uniform weight distribution of *(10'000, 500'000)* [4] and scale all

[1] https://www.boost.org/doc/libs/1_79_0/libs/graph/example/weighted_matching_example.cpp

[2] https://stackoverflow.com/questions/65327854/issue-using-cpp-boost-maximum-weighted-matching-algorithm-with-floating-point-ed/65331081#65331081

[3] https://github.com/boostorg/graph/issues?q=is%3Aissue+is%3Aopen+maximum_weighted_matching

[4] https://stackoverflow.com/questions/7524838/fixed-point-vs-floating-point-number/7524916#7524916

the resulting weights operations by *10'000.0*. In other words we initialized all the weights scaled by $\times$ *10'000.0*, further when we finished all the computation and we were to store the data in the *.csv* file, we re-scaled all the numeric information regarding the weights by $\times$ *10'000.0* transforming in a double type number [11]. In addition this shortcut helps us also to overcome the problem of limited choice of edge weight. More deeply, if we consider a small random distribution for the weight edges, like *(1 - 10)*, as the number of vertices increases the possibility to obtain a higher and higher number of the distribution increases as well. Thus we let the edges choose randomly a weight in a higher set in order to introduce some kind of random uncertainty.

The further step was to face the Auction Algorithm, firstly we developed a simplified version that useed *std::vector* structures to store all the variables, like the Pseudocode 3. Next we decided to move the data structure to a different level. After lots of researches we found a strategy from the BGL documentation called Bundle Proprieties[5] that enable the characterization of vertices and edges by specifying their structure. However in our case we have two types of vertices (Items and Bidders) and not a single one of the entire graph. Again we discovered another Boost feature called Variant[6] that allows us to specify the types of vertices. The following snippet of code explains the implementation of these two features.

```
namespace Nodes { // Vertices Bundle Proprieties
    using Weight = int64_t;
    struct Bidder {
        int    id;
        int    best_item = -1;
        Weight val_first_best_item = -1;
        Weight val_second_best_item = -1;
    };
    struct Item {
        int   id;
        Weight cost = 0;
        int    high_bidder = -1;
        Weight high_bid = -1;
    }; // Boost Variant for Vertices types
    using VertexProp = boost::variant<Bidder, Item>;
}
using Nodes::Weight;
using Nodes::Bidder;
using Nodes::Item;
using Nodes::VertexProp;
struct GraphProp { // Graph Bundle Proprieties
    std::vector<int> bidder2item;
    std::vector<int> item2bidder;
}; // Edges Bundle Propriety
using EdgeProp = boost::property<boost::edge_weight_t, Weight>;
using Graph = boost::adjacency_list<boost::vecS, boost::vecS,
```

[5] https://www.boost.org/doc/libs/1_79_0/libs/graph/doc/bundles.html
[6] https://www.boost.org/doc/libs/1_79_0/doc/html/variant.html

```
                boost::undirectedS, VertexProp, EdgeProp, GraphProp>;
// Inserting of Bundle Proprieties: Vertices, Edges and Graph
```

An other important step during the first phase was dealing on what data structure adopt to store the graph. The two choices was *adjacent_list* or *adjacent_matrix*, these two options have the following advantages and disadvantages, (we indicate V as the number vertices and E as the number of edges): [1][8]

Operation	Adjacent Matrix	Adjacent List
Storage Space	$O(V^2)$	$O(V+E)$
Querying Edge	$O(1)$	$O(V)$
Add / Remove vertices	$O(V^2)$, $O(V^2)$	$O(1)$, $O(V+E)$
Add / Remove edges	$O(1)$, $O(1)$	$O(1)$, $O(E)$

Table 4.1: Operations Complexity for Adjacent List and Adjacent Matrix

We decided to adopt *adjacent_list* because in our case since we work with bipartite graph where at most the number of edges is exactly $(V/2)^2$ and there are not any edges within the two parts if we decide to use *adjacent_matrix* some cells would not be used in any occasion and this results in a waste of memory.

4.2.2 2° Phase

After a clarification with the professor came out that what we had done up that moment was not the main project requirement. The key requirement was implement the Auction Algorithm in the style of the Boost Graph Library, in the sense that the algorithm must be capable to deal with whatever type of graph, so graph stored via both *adjacent_list* and *adjacent_matrix*. In addition to this, in this time frame, we have brought forward the characteristics of the application by also implementing the possibility of working with incomplete bipartite graphs. Further regarding the Auction Algorithms we managed to implement also the ϵ-Scaling version with scaling factor from 1/4 to 1/10 in order to have another landmark of benchmarks. All of these features characterize the second phase of the development of the application benchmark.

As we were satisfied with what we did, we decided to use the same implementation and structure to meet the need. The key choice was to adopt a *class* to initialize and store all the data structure requests useful for making the auction algorithm work correctly. We were inspired by the classic structure of most Python machine learning algorithms, which first need to be initialized, fitted, applied and finally we can get the model score. We also decided to opt for using *std::unordered_map* instead of classic vectors to optimize Heap memory. With this implementation any graphs type could run our benchmark except those that don not implement an assignment problem.

At this point we have moved on to improve and expand the previous function that generated a random bipartite graph to also create incomplete randomized bipartite graphs. The key point of this feature is that for each bidder we generate a weighted connection to at least one item. So a random distribution between 0 and 1 decides whether the same bidder will have additional connections.

The next step was to update both algorithms to handle incomplete bipartite graphs. For the Maximum Weighted Matching we did not make any changes as the implementation was already done and was working fine. However in the case of small graphs, there was a possibility that the final solutions did not meet the requirements for the Assignment Problem. At that time the main thought was to inform the community of the problem, so we contacted the Boost Team via an issue in the specific GitHub repository [7]. Moreover now we are still working to make a contribution to solve this unusual issue. On the other hand, the Original Auction Algorithm also presented a similar problem where it sometimes got stuck in an infinite loop trying to assign a bidder to his favorite item with the disadvantage that the actual cost was too high. First, to solve this problem, we re-studied the auction algorithm theory from the beginning and at the end we found what has already been reported on Page 18 in the Assignment Phase:

> If j_i is the only object in $A(i)$, we define w_i, to be $-\infty$ or, for computational purpose, a number that is much smaller than v_i.

Thus what we have done was nothing but change the initial value of the temporal variables of the first and second best object from -1 to a very small number like:

```
static_cast<double>(std::numeric_limits<Weight>::min());
```

In this way if a bidder has only a few item connections or even only a single one the bid for those will be much higher respect others bidders.

And in conclusion, as said before in order to have additional information and to make a better comparison we added the run of ϵ-Scaling Auction Algorithms from scaling factor of 1/4 up to 1/10, like the Pseudocode 4. Here since there was the possibility that not all types of scaling factor lead to terminate the computation of the algorithm, we introduced an exit clause in case the best object for a specific bidder was not present in the item list, this meant that his favorite item already cost more than his eventual offer. In that particular case we stop the execution and we return -1 for both execution time and sum of the matching, in order to distinguish the failed runs from the others.

The following snippet of code explain the final implementation of Auction class.

```
template<typename Graph, typename Type>
class Auction {
private:
    struct Bidder {
        int best_item;
        double val_first_best_item;
        double val_second_best_item;
    };
    struct Item {
        double cost = 0;
        int high_bidder = -1;
        double high_bid = -1;
    };
```

[7] https://github.com/boostorg/graph/issues/306

```
    int n_iteration_au = 0;
    int vertices = 0;
    int scaling_factor = 1;
    // Unordered map that will store all the unassigned bidders
    std::unordered_map<int, Bidder> unassigned_bidder;
    // Unordered map that will store all the assigned bidders
    std::unordered_map<int, Bidder> assigned_bidder;
    // Unordered map that will store all the items
    std::unordered_map<int, Item> item_map;
    bool is_assignment_problem(const Graph& graph);
    void auctionRound(const Graph& graph, const double& eps,
        const vertex_idMap<Graph>& V_Map, bool& err);
public:
    void original_auction(const Graph& graph, std::vector<int>& ass);
    void e_scaling_auction(const Graph& graph, std::vector<int>& ass,
        const double& scaling_factor);
    Type getTotalCost(const Graph& graph);
    int getNIterationAu();
    void printProprieties();
    void reset();
    Auction() { }
    Auction(int vertices, int scaling_factor = 1) {
                this->scaling_factor = scaling_factor;
        this->vertices = vertices;
        for (int i : boost::irange(0, vertices)) {
            this->unassigned_bidder.insert(std::make_pair(i, Bidder{}));
            this->item_map.insert(std::make_pair(i, Item{}));
        }
    }
};
```

Chapter 5

Results

In this chapter we discuss the results from the runs of all algorithms in the set of graphs with the number of vertices per part from 2 to 600. Before starting we would like to make clear that all benchmarks were run in a single machine, an HP EliteDesk Tower supplied with 8 cores and 16GB of RAM, with Ubuntu Linux 20.04.04 LTS operating system.

The application was compiled using the *g++* compiler with the *-Ofast* option. Indeed turning on optimization flags makes the compiler attempt to improve the performance and/or code size at the expense of compilation time and possibly the ability to debug the program.

First of all, because of the high amount of data, we have decided to split the benchmark of both **Complete** and **Incomplete** bipartite graphs in three parts in order to analyse and study the phenomena in a better way:

- **Small number of vertices:** from 2 up to 50.
- **Medium number of vertices:** from 51 up to 200.
- **High number of vertices:** from 201 up to 600.

Further, we would like to specify that all the data like graphs and results .csv files are available at the benchmark_graphs folder of our GitHub Project Repository.

5.1 Complete Bipartite Graphs

Firstly, we analyse the benchmark for the Assignment Problem in Complete Bipartite Graphs. Here we use 3 line plots to describe the trends:

- **Execution Time** to compere the time taken of all the strategies to solve the assignment problem.
- **% Difference of Total Cost** to compare the strategies results respect the Maximum Weighted Matching.
- **Auction Iterations** to compare the number of iterations of all Auction Algorithms.

5.1.1 Small number of vertices

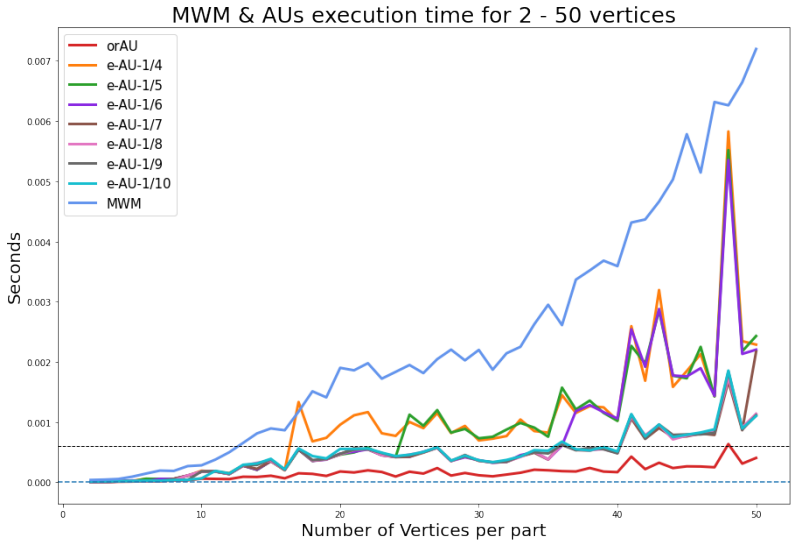

Figure 5.1: Execution Time - 20, 50 - Comp

In Figure 5.1 we note that our implementation of the Original Auction Algorithm takes always less than 0.0006 seconds to solve all the given assignment problems, whereas the Maximum Weighted Matching seems to have an exponential growth directed proportional to the number of vertices per part. On the other hand the ϵ-Scaling Auction Algorithms are divided into two major groups, with execution time always between the Original Auction Algorithm and the MWM.

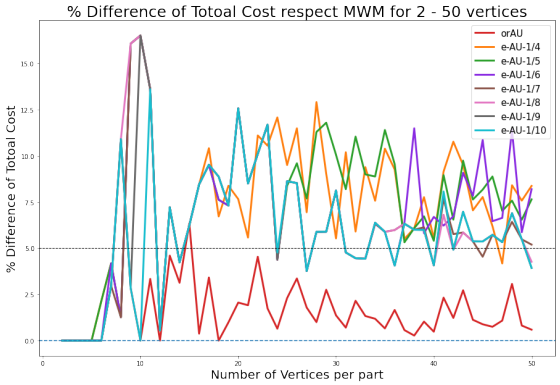

Figure 5.2: % Diff Cost - 20, 50 - Comp

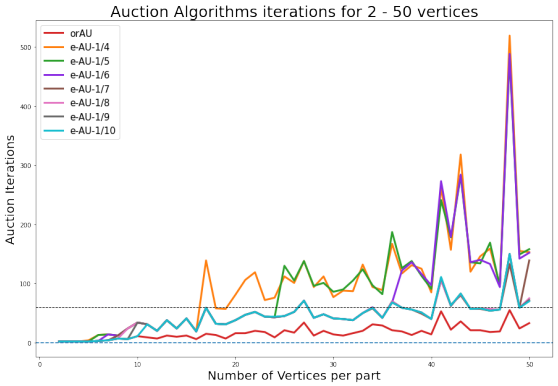

Figure 5.3: AUs iters - 20, 50 - Comp

Continuing in Figure 5.2 we see that the difference of total cost in percentage slowly decreases inversely proportionally to the number of vertices per part. This means that the error made by the Original Auction Algorithm decreases homogeneously and therefore it is more tolerable, indeed in all the runs it returns a difference of cost less than the 5% except for a single one. Whereas the ϵ-Scaling Auction Algorithms have consistently a higher percentage of cost difference with respect to the original version. Moreover, in Figure 5.3 as expected the number of iterations made by the Original Auction Algorithm increases much slowly with respect to the ϵ-Scaling Auction Algorithms. In fact The original version always takes less than 60 iterations.

5.1.2 Medium number of vertices

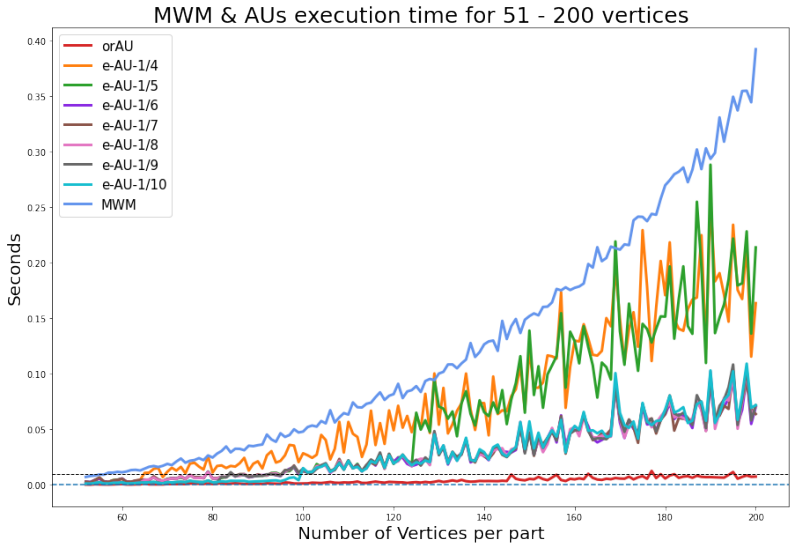

Figure 5.4: Execution Time - 51, 200 - Comp

Like the previous line plots here the trends are almost identical with an exponential growth for the Maximum Weighted Matching and the Original Auction Algorithm that is most of the time close to the 0.01 second of time execution. The others ϵ-Scaling Auction Algorithms still compose two major groups.

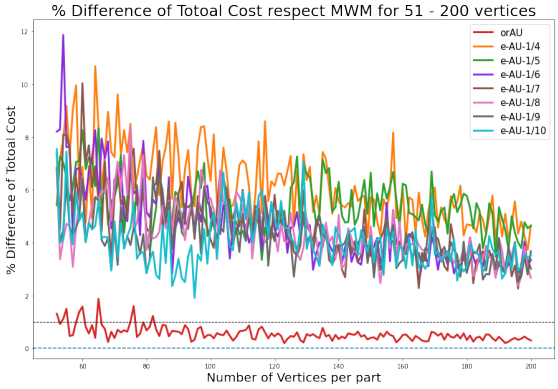

Figure 5.5: % Diff Cost - 51, 200 - Comp

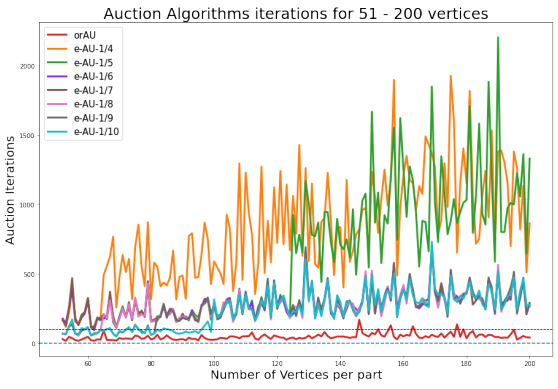

Figure 5.6: AUs iters - 51, 200 - Comp

In Figure 5.5 we note that after the graph with 100 vertices per part every run of the Original Auction Algorithm returns a difference of total cost that is less than 1%. Whereas the ϵ-Scaling versions are always far from the performance of the Original version. Figure 5.6 tells us that almost every assignment problem taken into account by the Original Auction Algorithm has been solved for most of the time with approximately less than 100 iterations. Instead the others Auction Algorithms take much more iteration since they repeat the inner while loop multiple times.

5.1.3 High number of vertices

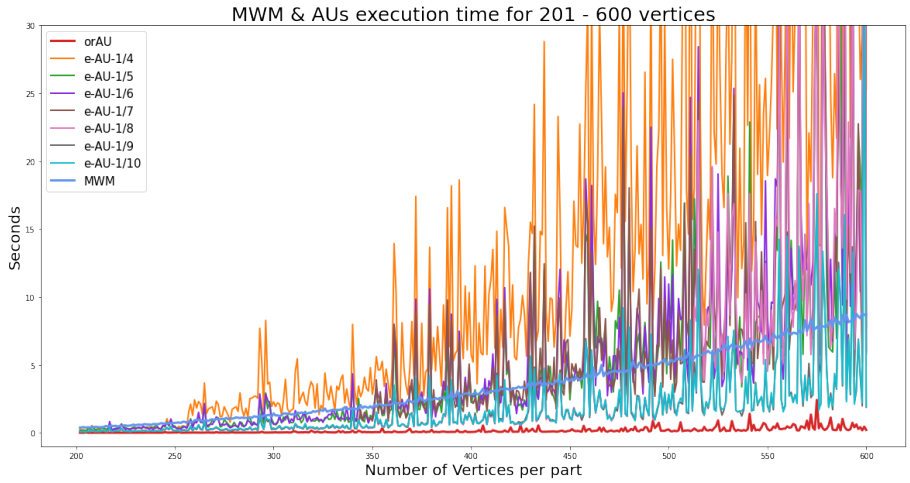

Figure 5.7: Execution Time - 201, 600 - Comp

Here the execution time has the same trend of the previous two, but more interesting we note that the both Original Auction and Maximum Weighted Matching have a smooth increase in execution time, whereas the others versions of Auction Algorithm take much more time to finish the execution (Figure 5.7).

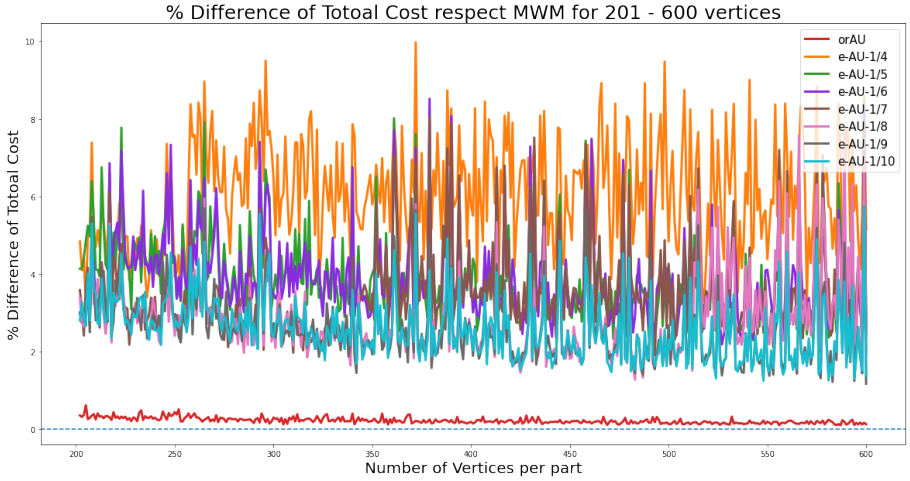

Figure 5.8: % Diff Cost - 201, 600 - Comp

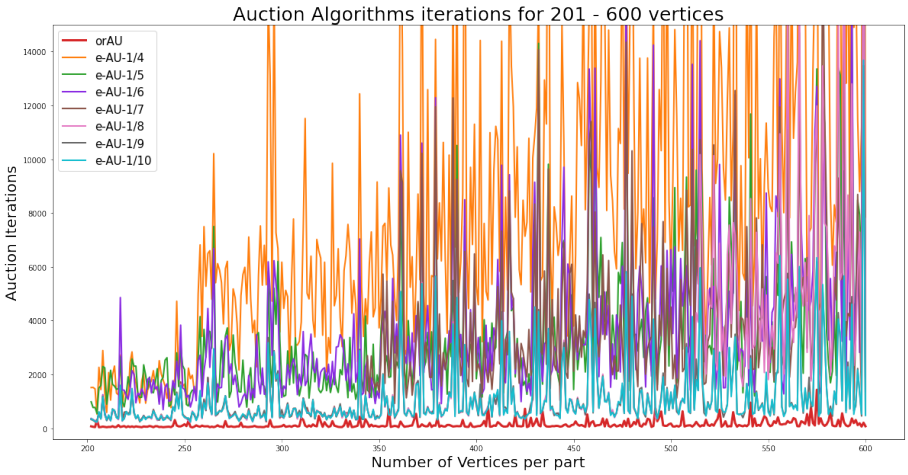

Figure 5.9: AUs iters - 201, 600 - Comp

Regarding Figure 5.8 and more accurately in Figure 5.10 we can see that already from 200 vertices per part the percentage difference of total cost of the Original Auction Algorithm is less than the 0.6% and it reach the 0.35% analysing the graph with 360 vertices per part. The other versions of Auction Algorithm have an increasing trend, but they are again far away from the performance of the original version. Figure 5.11 tell us that the Original Auction Algorithm was capable of solving the assignment problems for almost every graph with a very low number of iterations. In fact in Figure 5.9 we can see that the others versions have an exponential growth of iterations due to the scaling factor.

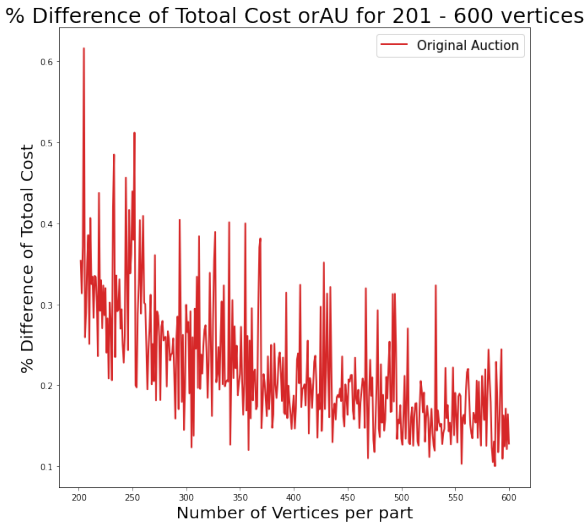

Figure 5.10: % Diff Cost orAU - 201, 600 - Comp

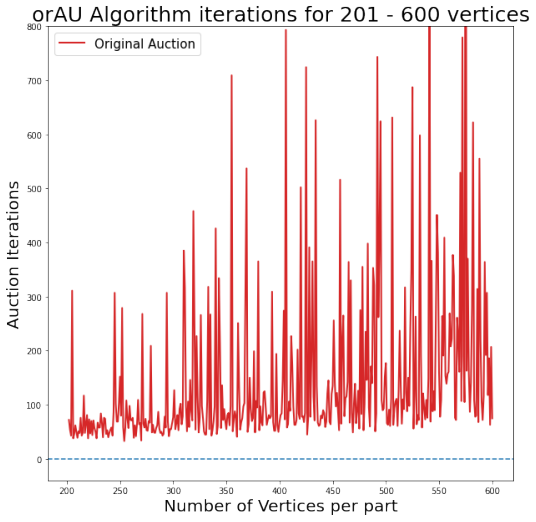

Figure 5.11: orAU iters - 201, 600 - Comp

5.2 Incomplete Bipartite Graphs

Now we discuss the results obtained from the benchmarks of Assignment Problems on Incomplete Bipartite Graphs. In addition to the previous style of graph we have decided to add also 4 tables:

1. **Number of Failure** to explain the number of failures between ϵ-Scaling Auction Algorithms and Maximum Weighted Matching.
2. **Auction Winner** which tell us how many times each Auction Algorithms win. This metric is based on the comparison between the total cost as first parameter and in case of equality the execution time is also contrasted.
3. **Execution Time Winner** to describe the number of wins of Auction Algorithms and Maximum Weighted Matching respect the execution time.
4. **Total Cost Winner** to describe the number of wins of Auction Algorithms and Maximum Weighted Matching respect the matching weighted sum.

5.2.1 Small number of vertices

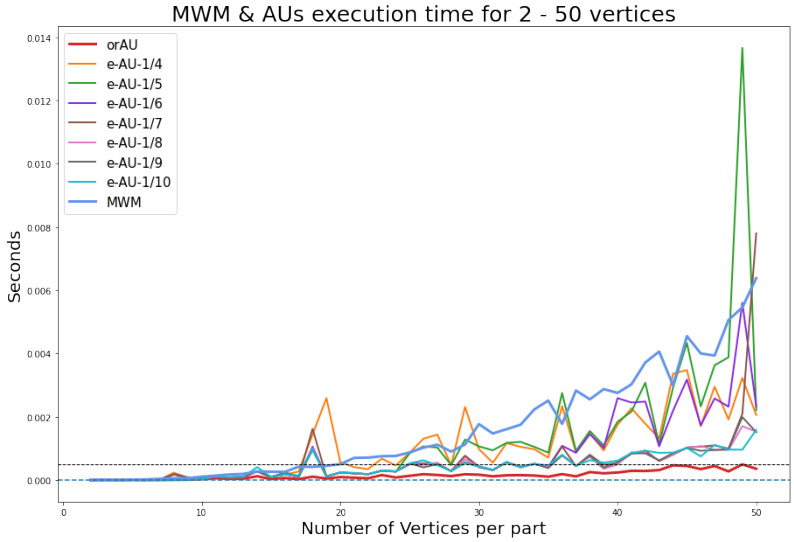

Figure 5.12: Execution Time - 2, 50 - InComp

Looking at Figure 5.12 we can see that now the execution time of the others Auction algorithms is much variable, for example we see that after the execution of graph with 50 vertices per part the versions with scaling factor of 1/8, 1/9 and 1/10 are much near the execution time of the Original Auction Algorithm. Whereas for example the execution of a graph with 18 vertices per part takes an amount of time higher also than the MWM, like also for the graph with 49 vertices.

Continuing in Figure 5.13 we note that 3 executions return a percentage of difference equal to -1 this means that the Maximum Weighted Matching returns an incorrect solution for the assignment problems thus every Auction Algorithms outperform the MWM (Table 5.1). Moreover Figure 5.14 shows a constant increase in iterations in all Auction Algorithms with a more remarkable growth for the scaling versions.

In addition from Table 5.4 we can see that almost every executions are won by the Maximum Weighted Matching except 3 times in which the best Auction Algorithm win and others 7 times where there is a draw.

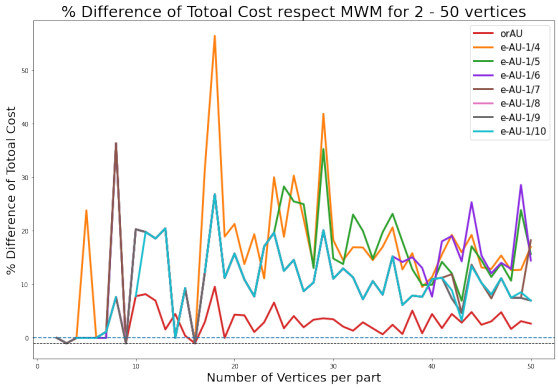

Figure 5.13: % Diff Cost - 2, 50 - InComp

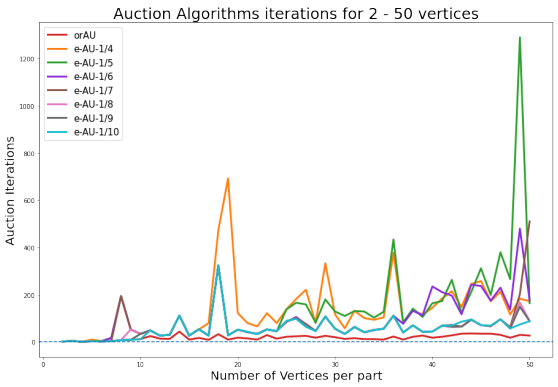

Figure 5.14: AUs iters - 2, 50 - InComp

Algorithm	Number of Failures
MWM	3
eAU_4	0
eAU_5	0
eAU_6	0
eAU_7	0
eAU_8	0
eAU_9	0
eAU_10	0

Table 5.1: Number of Failure (2, 50)

Algorithm	Number of Wins
orAU	45
eAU_4	0
eAU_5	1
eAU_6	0
eAU_7	2
eAU_8	0
eAU_9	0
eAU_10	1

Table 5.2: Auction Winner (2, 50)

Algorithm	Number of Wins
AU	48
MWM	1

Table 5.3: Execution Time Winner (2, 50)

Algorithm	Number of Win
MWM	39
AU	3
None	7

Table 5.4: Total Cost Winner (2, 50)

5.2.2 Medium number of vertices

Now talking about the medium number of vertices we notice by Figure 5.15 that the amount of execution time for the ϵ-Scaling Auction Algorithms overtake more and more the execution time of the Maximum Weighted Matching. Whereas when the number of vertices per part increases gradually the gap between the Original Auction and the others strategies increase as well.

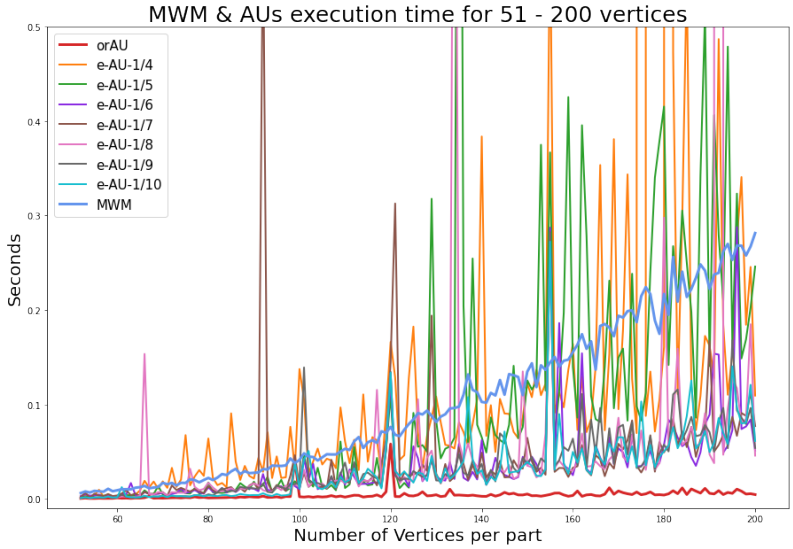

Figure 5.15: Execution Time - 51, 200 - InComp

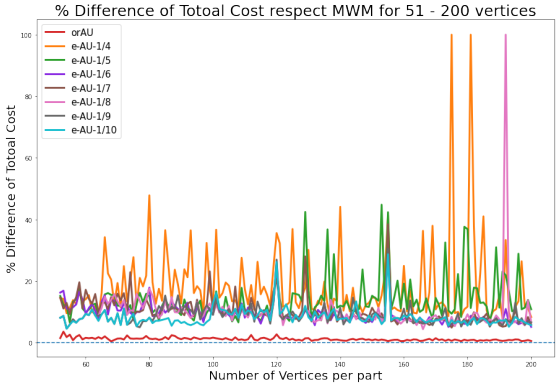

Figure 5.16: % Diff Cost-51,200-InComp

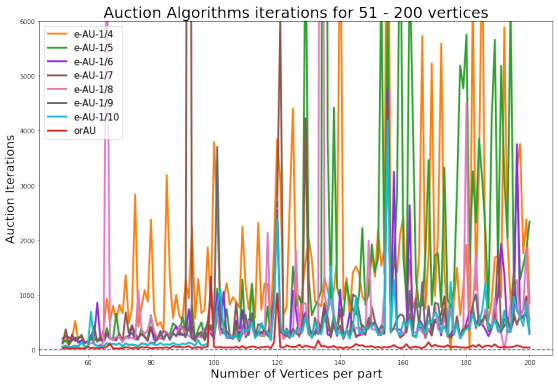

Figure 5.17: AUs iters - 51, 200 - InComp

By Figure 5.16 we notice that 3 runs of ϵ-Scaling Auction Algorithm failed, with respectively scaling factor of 1/4 and 1/8, as mentioned in Table 5.1. Moreover the gap between the Original Auction and the other methods still remain markable in both Figures.

Algorithm	Number of Failures
MWM	0
eAU_4	2
eAU_5	0
eAU_6	0
eAU_7	0
eAU_8	1
eAU_9	0
eAU_10	0

Table 5.5: Number of Failure (51, 200)

Algorithm	Distribution of Win
orAU	149

Table 5.6: Auction Winner (51, 200)

Algorithm	Distribution of Win
AU	149
MWM	1

Table 5.7: Execution Time (51, 200)

Algorithm	Distribution of Win
MWM	149

Table 5.8: Total Cost (51, 200)

5.2.3 High number of vertices

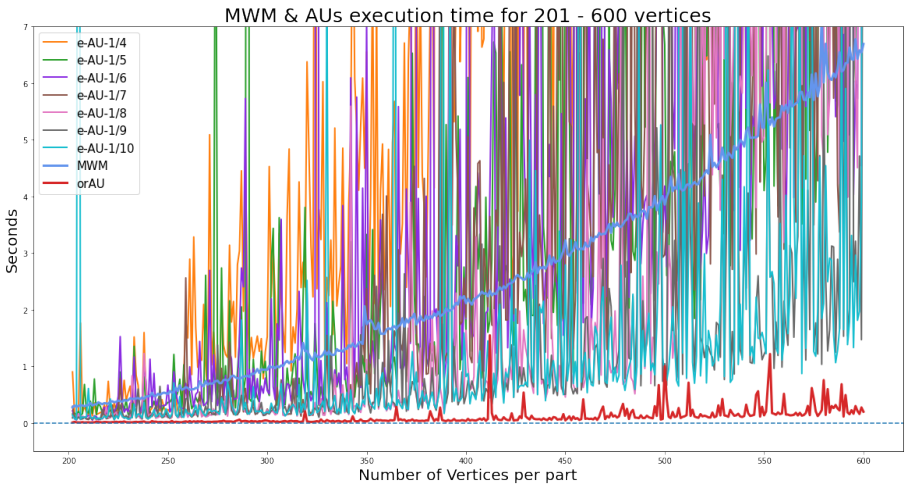

Figure 5.18: Execution Time - 201, 600 - InComp

Finally we arrive at the last analysis, in which we compare the result of graphs with the number of vertices between 201 and 600. Figure 5.18 tell us that the execution time of the Original Auction Algorithm again outperform the others methods. Next, we can mention that most runs of ϵ-Scaling with scaling factor of 1/10 and 1/9 are under the execution time of the Maximum Weighted Matching, but still higher than the once of the Original Auction.

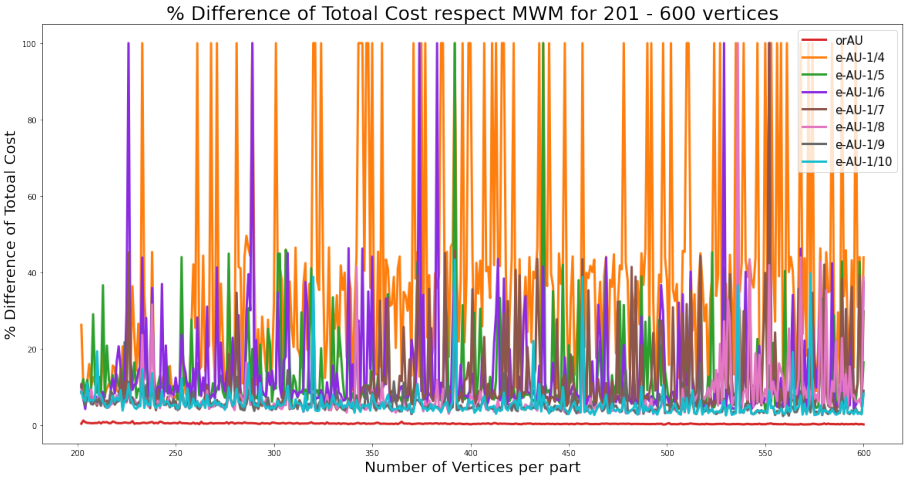

Figure 5.19: % Diff Cost - 200, 600 - InComp

On the other hand in Figure 5.19 we can see that there still is a gap between the Original Auction Algorithm and the other strategies. Moreover many ϵ-Scaling methods failed and this is why they return the difference of total cost to 100% and the number of iterations to -1 (Figure 5.20). All of this is summarized in Table 5.5.

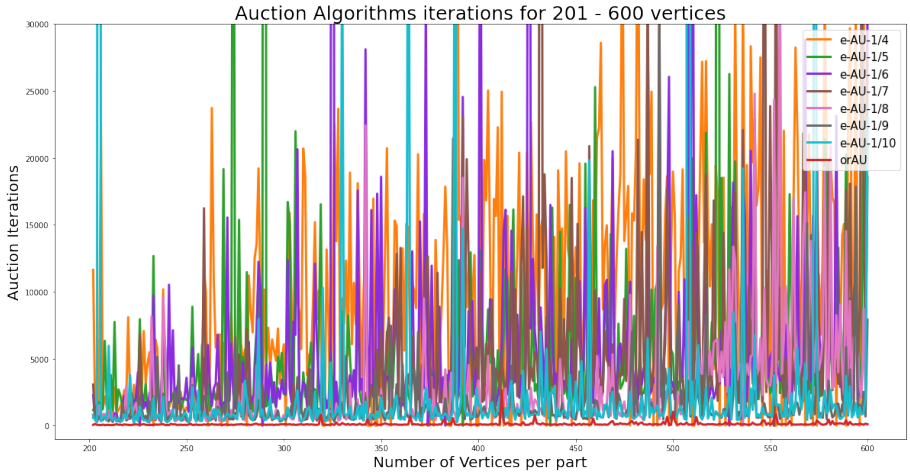

Figure 5.20: AUs iters - 200, 600 - InComp

Algorithm	Number of Failures
MWM	0
eAU_4	61
eAU_5	2
eAU_6	6
eAU_7	1
eAU_8	1
eAU_9	0
eAU_10	0

Table 5.9: Number of Failure (201, 600)

Algorithm	Distribution of Win
orAU	399

Table 5.10: Auction Winner (201, 600)

Algorithm	Distribution of Win
AU	399

Table 5.11: Execution Time (201, 600)

Algorithm	Distribution of Win
MWM	399

Table 5.12: Total Cost Winner (201, 600)

In conclusion to better understand the actual performance of the Original Auction Algorithm we report these three additional graphs.

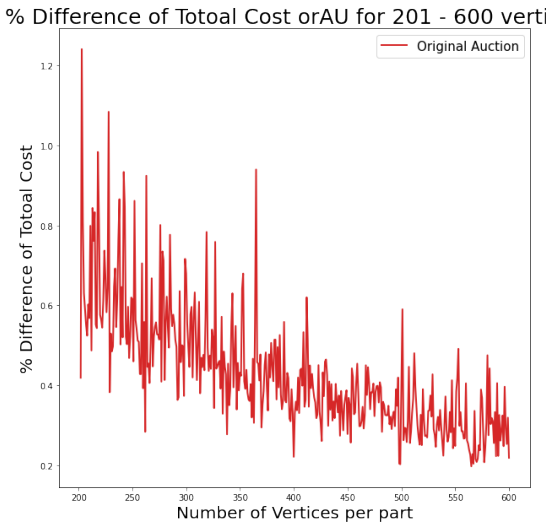

Figure 5.21: % Diff Cost orAU - 201, 600 - InComp

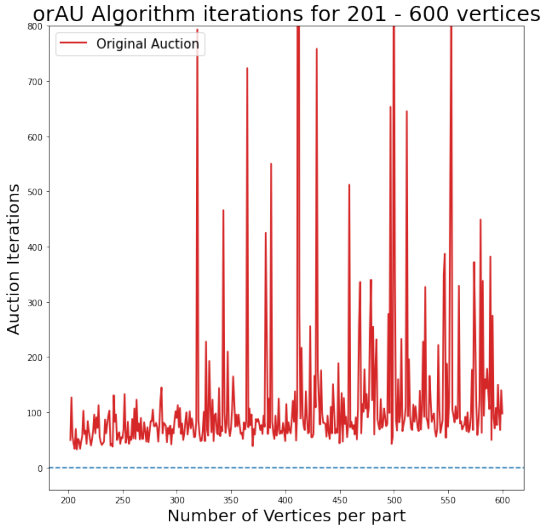

Figure 5.22: orAU iters - 201, 600 - InComp

By Figure 5.21 we can see that more or less the assignment problems for graphs with more than 450 vertices per part are solved with a difference of total cost less than 0.5%. On the other side, from Figure 5.22 we see that the overall number of iterations except a few outliers is not higher than 300.

Chapter 6

Conclusion

In this report we discussed the comparison between two main algorithms that aim to solve the assignment problem in Complete and Incomplete bipartite graphs. The first one is the Maximum Weighted Matching and use the argumented path in order to find a possible matching. The other is the Auction Algorithm which with all its ϵ-Scaling variants base their behaviour like an classical auction.

Moreover we analysed also a specific benchmark application to compere in detail all the strategies and we obtained the following conclusions:

- **For Assignment Problems in Complete Bipartite Graphs:**
  - In all analysis the implementation of Original Auction Algorithm was much faster than the Boost implementation of Maximum Weighted Matching and than all the ϵ-Scaling variants.
  - The Original Auction Algorithm has the disadvantage that the total cost of matching is always slightly lower than the Maximum Weighted Matching.
  - Al the ϵ-Scaling Auction Algorithms do not perform as well as the Original Auction Algorithm.
  - The choice of the methods depends on the priority that we have. So if we want to have the best possible matching with the maximal cost we have to choose the Maximum Weighted Matching. On the other hand if we care most on the execution time of the algorithm we have to use the Auction Algorithm.
- **For Assignment Problems in Incomplete Bipartite Graphs:**
  - The Maximum Weighted Matching sometimes fail to return a correct solution for an Assignment Problem with few vertices per part, whereas all the Auction Algorithms always return a correct result.
  - The Original Auction Algorithm seems to work much better and faster than its ϵ-Scaling variants.
  - As before the Original Auction Algorithm outperform the Maximum Weighted Matching in execution time, and the Maximum Weighted Matching outperform the Original Auction Algorithm in the matching cost.

In conclusion from my side I mostly like to use the Original Auction Algorithm because:

- It is much faster that the Maximum Weighted Matching and the ϵ-Scaling variants.
- It has a very small difference in total cost, where if the number of vertices per side continues to grow it gradually becomes insignificant.
- And most importantly it does not fail to return a correct solution.

Bibliography

[1] behlanmol. *Comparison between Adjacency List and Adjacency Matrix representation of Graph*. URL: https://www.geeksforgeeks.org/comparison-between-adjacency-list-and-adjacency-matrix-representation-of-graph/.

[2] Dimitri Bertsekas and David Castanon. "The auction algorithm for the transportation problem". In: *Annals of Operations Research* 20 (Dec. 1989), pp. 67–96. DOI: 10.1007/BF02216923.

[3] Dimitri P. Bertsekas. "Auction algorithms for network flow problems: A tutorial introduction." In: *Comput. Optim. Appl.* 1.1 (1992), pp. 7–66. URL: http://dblp.uni-trier.de/db/journals/coap/coap1.html#Bertsekas92.

[4] Jack Edmonds. "Maximum matching and a polyhedron with 0,1-vertices". In: *Journal of Research of the National Bureau of Standards Section B Mathematics and Mathematical Physics* (1965), p. 125.

[5] Harold N. Gabow. "An Efficient Implementation of Edmonds' Algorithm for Maximum Matching on Graphs". In: *J. ACM* 23.2 (Apr. 1976), pp. 221–234. ISSN: 0004-5411. DOI: 10.1145/321941.321942. URL: https://doi.org/10.1145/321941.321942.

[6] Zvi Galil. "Efficient Algorithms for Finding Maximal Matching in Graphs." In: Mar. 1983, pp. 90–113. ISBN: 978-3-540-12727-7. DOI: 10.1007/3-540-12727-5_4.

[7] Yi Ji. *Maximum Weighted Matching*. URL: https://www.boost.org/doc/libs/1_79_0/libs/graph/doc/maximum_weighted_matching.html.

[8] magiix. *What is better, adjacency lists or adjacency matrices for graph problems in C++?* URL: https://stackoverflow.com/questions/2218322/what-is-better-adjacency-lists-or-adjacency-matrices-for-graph-problems-in-c.

[9] Joris van Rantwijk. *Maximum Weighted Matching*. URL: http://jorisvr.nl/article/maximum-matching#ref:1.

[10] James E. Reeb and Scott Leavengood. "Transportation Problem: A Special Case for Linear Programming Problems". In: 2002.

[11] Seth Heeren Riccardo Zuliani. *Boost Maximum Weighted Matching in undirected bipartite random graphs hangs in an infinite loop*. URL: https://stackoverflow.com/questions/73081928/boost-maximum-weighted-matching-in-undirected-bipartite-random-graphs-hangs-in-a.

[12] Seth Heeren Riccardo Zuliani. *Pretty print vertex and edges structs via boost :: dynamic_properties*. URL: https://stackoverflow.com/questions/73053858/pretty-print-vertex-and-edges-structs-via-boostdynamic-properties.

[13] Mohammad R. Salavatipour. *Primal Dual Matching Algorithm and Non-Bipartite Matching*. URL: https://webdocs.cs.ualberta.ca/~mreza/courses/CombOpt09/lecture4.pdf.

[14] Ana Zelaia Jauregi. *The transportation and the assignment problem*. URL: https://ocw.ehu.eus/pluginfile.php/40935/mod_resource/content/1/5_Transportation.pdf.

Appendix A

Application Screenshots

```
-------- MAXIMUM WEIGHTED MATCHING - AUCTION ALGORITHM BECHMARK --------

Do you want to activate VERBOSE mode? (0/1) 0
On what type of bipartite graph would you like to work with, incomplete (0) or complete (1): 0
Please specify the starting number of vertices per part: 5
Please specify the ending number of vertices per part: 5
```

Figure A.1: User Inputs

```
Generation of a Incomplete Bipartite Graph with 5 vertices per part: done

Execution of Maximum Weighted Matching... Finished
The matching is: (0,2,542)(1,1,3555)(2,4,4845)(3,3,9878)(4,0,5666)
It took: 84.528000us, with total cost: 24486

Execution of Auction Algorithms...
Running auction_e_scaling_0.100 with epsilon scaling factor: 0.100000... Finished
Running auction_e_scaling_0.111 with epsilon scaling factor: 0.111111... Finished
Running auction_e_scaling_0.125 with epsilon scaling factor: 0.125000... Finished
Running auction_e_scaling_0.143 with epsilon scaling factor: 0.142857... Finished
Running auction_e_scaling_0.167 with epsilon scaling factor: 0.166667... Finished
Running auction_e_scaling_0.200 with epsilon scaling factor: 0.200000... Finished
Running auction_e_scaling_0.250 with epsilon scaling factor: 0.250000... Finished
Running auction_original... Finished
The matching is: (0,2,542)(1,1,3555)(2,4,4845)(3,3,9878)(4,0,5666)
The best strategy: auction_e_scaling_0.167 took: 6.669000us, with total cost: 24486 and 2 iterations
```

Figure A.2: VERBOSE mode OFF

```
Generation of a Incomplete Bipartite Graph with 5 vertices per part: done

graph G {
0;
1;
2;
3;
4;
5;
6;
7;
8;
9;
0--8  [weight=3271];
0--5  [weight=9745];
0--6  [weight=8617];
1--9  [weight=8708];
1--5  [weight=9717];
1--6  [weight=2829];
1--7  [weight=1349];
2--6  [weight=9611];
2--5  [weight=7484];
2--8  [weight=9774];
2--9  [weight=676];
3--5  [weight=8926];
3--6  [weight=9392];
3--7  [weight=4827];
4--7  [weight=1659];
4--9  [weight=958];
}

Execution of Maximum Weighted Matching... Finished
The matching is: (0,0,9745)(1,4,8708)(2,3,9774)(3,1,9392)(4,2,1659)
It took: 56.826000us, with total cost: 39278

Execution of Auction Algorithms...
Running auction_e_scaling_0.100 with epsilon scaling factor: 0.100000... Finished
Running auction_e_scaling_0.111 with epsilon scaling factor: 0.111111... Finished
Running auction_e_scaling_0.125 with epsilon scaling factor: 0.125000... Finished
Running auction_e_scaling_0.143 with epsilon scaling factor: 0.142857... Finished
Running auction_e_scaling_0.167 with epsilon scaling factor: 0.166667... Finished
Running auction_e_scaling_0.200 with epsilon scaling factor: 0.200000... Finished
Running auction_e_scaling_0.250 with epsilon scaling factor: 0.250000... Finished
Running auction_original... Finished
The matching is: (0,0,9745)(1,4,8708)(2,3,9774)(3,1,9392)(4,2,1659)
|Bidder:0|Best item:0|Value first best item:9745.000000|Value second best item:8617.000000|
|Bidder:3|Best item:1|Value first best item:9392.000000|Value second best item:8926.000000|
|Bidder:4|Best item:2|Value first best item:1659.000000|Value second best item:958.000000|
|Bidder:2|Best item:3|Value first best item:9774.000000|Value second best item:9611.000000|
|Bidder:1|Best item:4|Value first best item:8708.000000|Value second best item:8588.833333|
|Item:4|Cost:119.333333|Higher bidder:1|Higher bid:119.333333|
|Item:3|Cost:326.333333|Higher bidder:2|Higher bid:163.166667|
|Item:2|Cost:1402.333333|Higher bidder:4|Higher bid:701.166667|
|Item:1|Cost:932.333333|Higher bidder:3|Higher bid:466.166667|
|Item:0|Cost:2256.333333|Higher bidder:0|Higher bid:1128.166667|
The best strategy: auction_original took: 7.247000us, with total cost: 39278 and 2 iterations
```

Figure A.3: VERBOSE mode ON

Appendix B

Special Thank

I would like to thank **Seth Heeren** very much for helping me learn the basics of the Boost Graph Library. He answered the following questions I asked in StackOverflow during the development of the project:

- Is it possible to generate multiple custom vertices using the Bundle Properties from Boost Graph Library?
- Pretty print vertex and edges structs via boost::dynamic_properties
- Boost Maximum Weighted Matching in undirected bipartite random graphs hangs in an infinite loop
- Implementation of Auction Algorithm via Boost Graph Library C++