

INVESTIGATION INTO MAXIMUM WEIGHT MATCHINGS

Team Number : 18

1. Amith Bhat Nekkare(181IT105)- 971508502105
2. Ankit Gupta(181IT107)- 9079662823
3. Ayush Rahangdale(181IT109)- 9145139726
4. Kumsetty Nikhil Venkat(181IT224)- 9008826999



INTRODUCTION

Some Definitions..

Given a graph $G = (V, E)$, a **matching** $M \subseteq E$ such that no two edges share a common vertex.

A **maximum matching** (also known as maximum-cardinality matching) is a matching that contains the largest possible number of edges. There may be many maximum matchings.



Maximum Weight Matching :

The problem of finding, in a weighted graph, a matching in which the sum of weights is maximized is called the maximum weight matching(MWM) problem.

Maximum Cardinality Matching :

The problem of finding, in a graph, a matching which contains as many edges as possible is called the maximum cardinality matching(MCM) problem.

- The MCM problem is a special case of the MWM problem.



Many real world applications of the MWM problem, require graphs of such large size that the running time of the fastest available weighted matching algorithm is too costly.

Examples of such problems are the refinement of FEM nets, the partitioning problem in VLSI-Design, and the gossiping problem in telecommunications. There also exist applications where the weighted matching problem has to be solved extremely often on only moderately large graphs.

Therefore, there is considerable need for **approximation algorithms** for the weighted matching problem that are very fast, and that nevertheless produce very good results even if these results are not optimal.





REQUIREMENTS ANALYSIS

HARDWARE REQUIREMENTS

- Processors: Will run on Intel Pentium and newer
- Hard Disk : 1 TB
- RAM : 4 GB



SOFTWARE REQUIREMENTS

- OS : Ubuntu 16.04 and upwards
- Requires Python 3.8 interpreter.
- Python libraries such as numpy and networkx(to generate graphs).





PROBLEM STATEMENT

To investigate and implement the following algorithms in the field of MWMs :

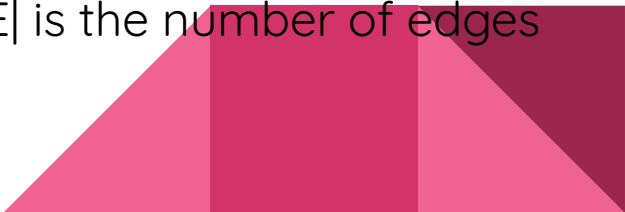
1. Edmonds' MCM Algorithm
2. The Drake-Hougardy MWM Algorithm





EDMONDS' ALGORITHM

Introduction

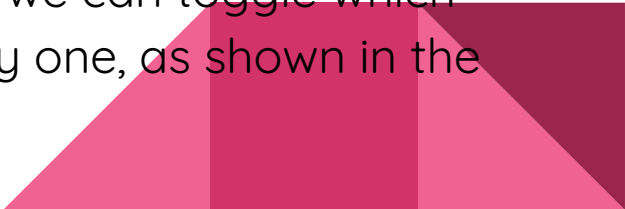
- Also known as the Blossom algorithm, the algorithm was developed by Jack Edmonds in 1961, and published in 1965.
 - As mentioned earlier, the algorithm is a solution to the MCM problem, which is a subset of the MWM problem.
 - Given a general graph $G = (V, E)$, the algorithm finds a matching M such that each vertex in V is incident with at most one edge in M and $|M|$ is maximized. The matching is constructed by iteratively improving an initial empty matching along augmenting paths in the graph.
 - The algorithm runs in time $O(|E| * (|V|^2))$, where $|E|$ is the number of edges of the graph and $|V|$ is its number of vertices.
- 

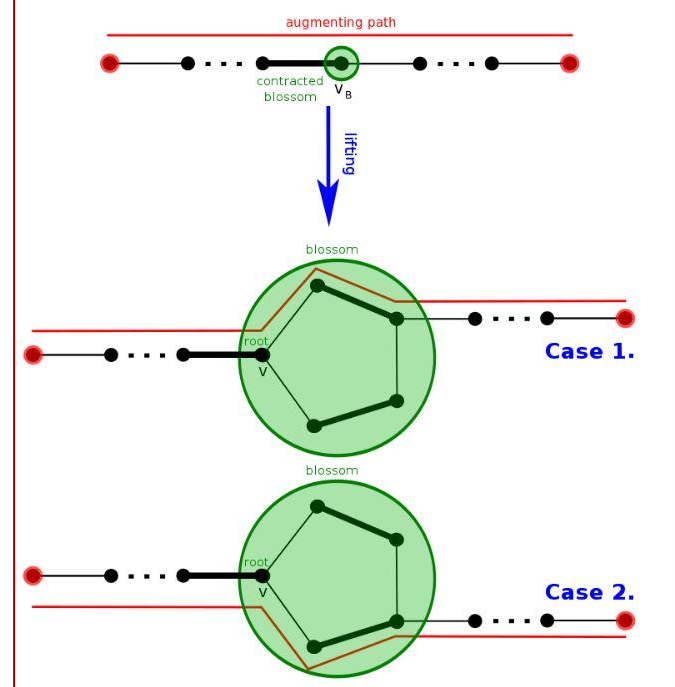
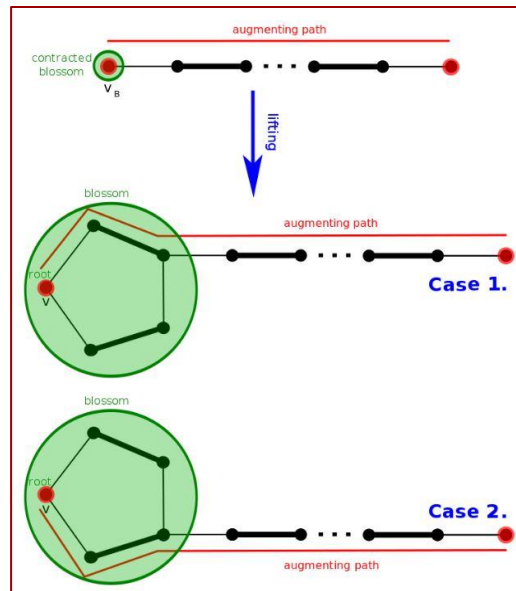
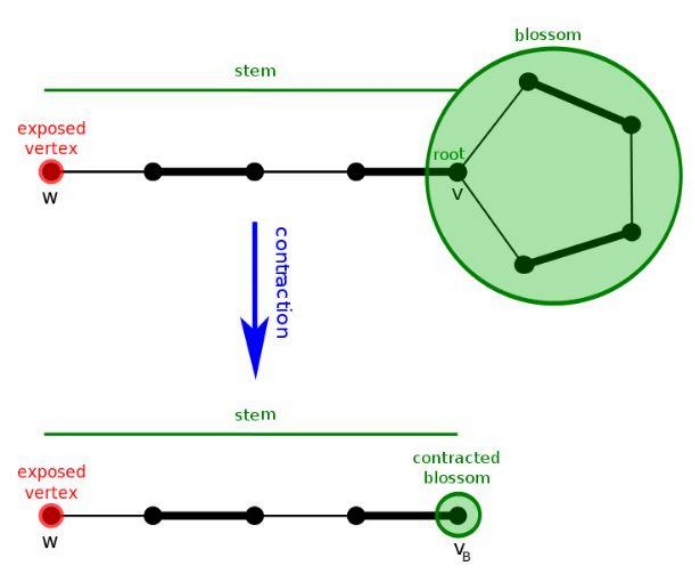
Definitions

For graph G with matching M , an **exposed vertex** v is a vertex that does not belong to M , but is in the graph G . That is $V(G \setminus M)$ are exposed vertices.

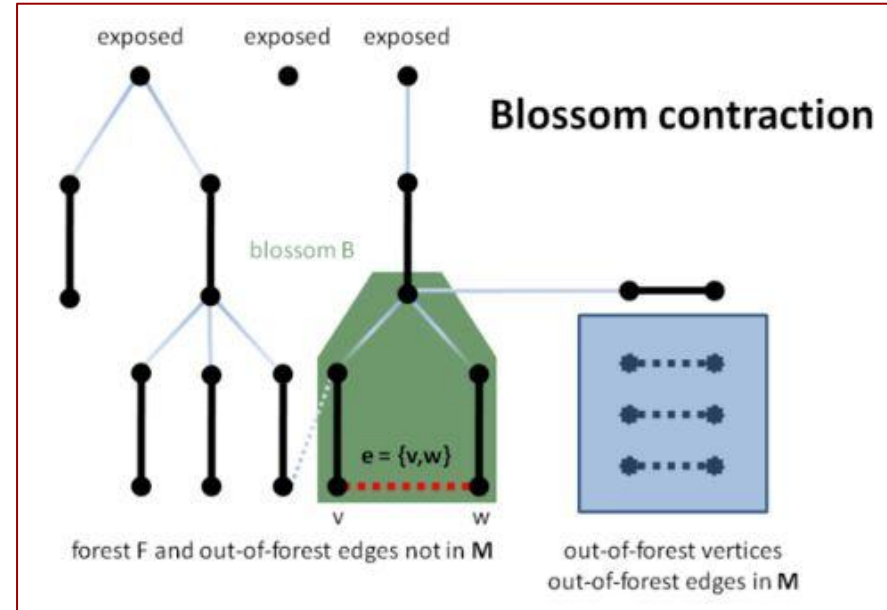
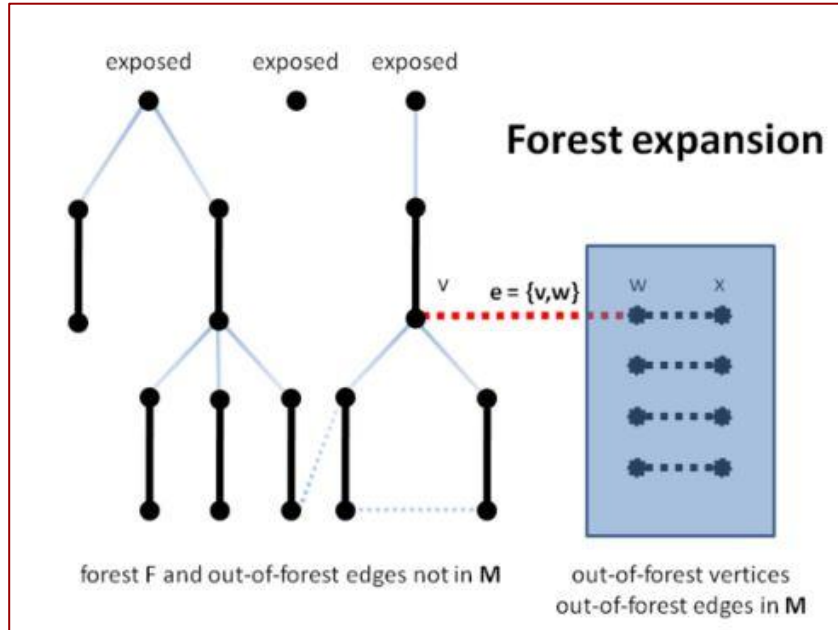
Given a matching M , an **augmenting path** is an odd length path whose endpoints are distinct exposed vertices, and whose edges e_1, \dots, e_{2k+1} alternate such that $e_{2i+1} \notin M$ for all $0 \leq i \leq k$ and $e_{2j} \in M$ for all $1 \leq j \leq k$.

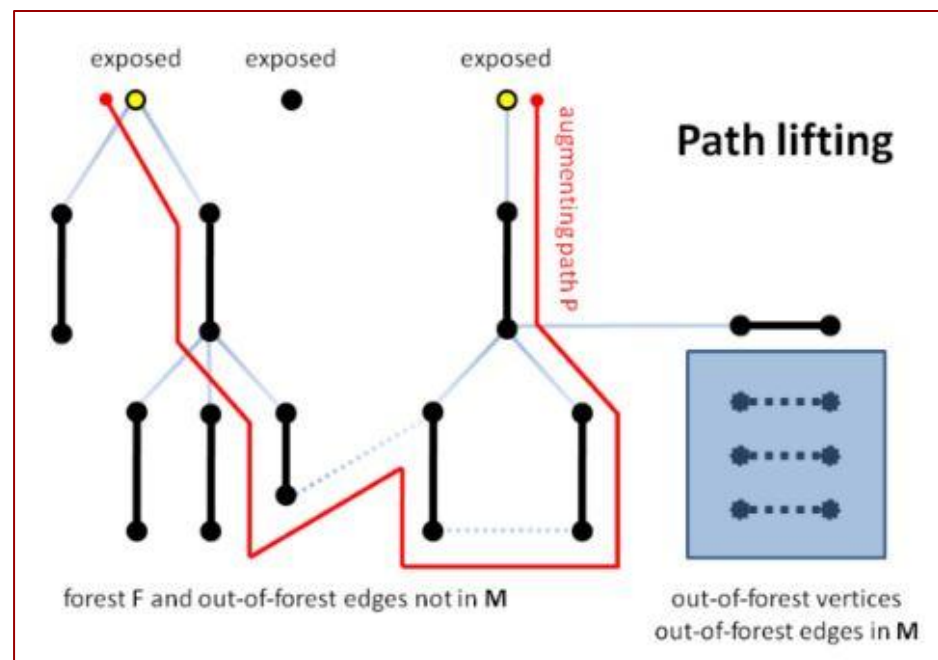
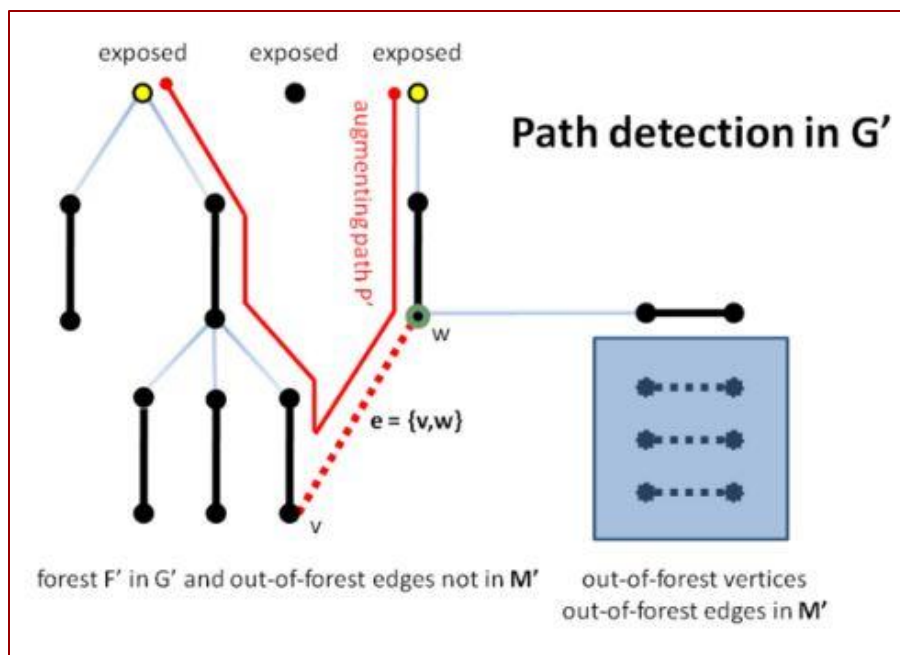
An augmenting path as defined above is an augmenting path of size $2k + 1$ and has k edges belonging to the matching and $k + 1$ which are not part of the matching. Our goal in finding augmenting paths is that we can toggle which edges are included in order to increase the matching by one, as shown in the following theorem.





Example





Pseudo code

Algorithm 1 Blossom Algorithm: Find Maximum Matching

```
1: procedure SEQ_FIND_MAXIMUM_MATCHING( $G, M$ )
2:    $P = \text{SEQ\_FIND\_AUG\_PATH}(G, M)$ 
3:   if  $P == []$  then
4:     return  $M$ 
5:   else
6:     Add alternating edges of  $P$  to  $M$ 
7:   return SEQ_FIND_MAXIMUM_MATCHING( $G, M$ )
```

Algorithm 2 Sequential Blossom Algorithm: Find Augmenting Path

```
1: procedure SEQ_FIND_AUG_PATH( $G, M$ )
2:    $F$  = empty forest
3:   nodes_to_check  $\leftarrow$  exposed vertices in  $G$ 
4:   for  $v$  in nodes_to_check do
5:     | Add  $v$  as single-node tree to  $F$ 
6:     | node_to_root( $v$ ) =  $v$ 
7:   in  $G$ , mark all matched edges (all edges in  $M$ )
8:   for  $v$  in forest_nodes do
9:     | while there exists an unmarked edge  $e = (v, w)$  do
10:      | if  $w \notin F$  then ▷ Vertex  $w$  must be in  $M$ 
11:      | | SEQ_ADD_TO_FOREST( $M, F, v, w$ )
12:      | else
13:      | | if dist( $w$ , node_to_root( $w$ )) % 2 == 0 then
14:      | | | if node_to_root( $v$ )  $\neq$  node_to_root( $w$ ) then
15:      | | | |  $P$  = SEQ_RETURN_AUG_PATH( $F, v, w$ , node_to_root)
16:      | | | | else
17:      | | | | |  $P$  = SEQ_BLOSSOM_RECURSION( $G, M, F, v, w$ )
18:      | | | | return  $P$ 
19:      | | | else
20:      | | | | # Do nothing
21:      | | mark edge  $e$ 
22:   return empty path
```

Algorithm 3 Sequential Blossom Algorithm: Add to Forest

```
1: procedure SEQ_ADD_TO_FOREST( $M, F, v, w$ )
2:    $x \leftarrow$  vertex adjacent to  $w$  in  $M$ 
3:   add edges  $(v, w), (w, x)$  to  $\text{tree}(v)$  in  $F$ 
4:   add vertex  $x$  to  $\text{nodes\_to\_check}$ 
5:    $\text{node\_to\_root}(w) = \text{node\_to\_root}(v)$ 
6:    $\text{node\_to\_root}(x) = \text{node\_to\_root}(v)$ 
```

Algorithm 4 Sequential Blossom Algorithm: Return Aug Path

```
1: procedure SEQ_RETURN_AUG_PATH( $F, v, w, \text{node\_to\_root}$ )
2:    $\text{root\_v} = \text{node\_to\_root}(v)$ 
3:    $\text{root\_w} = \text{node\_to\_root}(w)$ 
4:    $P1 \leftarrow \text{SHORTEST\_PATH}(F, \text{root\_v}, v)$ 
5:    $P2 \leftarrow \text{SHORTEST\_PATH}(F, w, \text{root\_w})$ 
6:   return  $P1 + P2$ 
```



THE DRAKE- HOUGARDY APPROXIMATION ALGORITHM

Introduction

In this project, we have implemented two different solutions to the MWM problem: Edmonds' Algorithm and the the Drake-Hougardy solution.

The Drake-Hougardy Algorithm, as an approximation algorithm, provides a linear-time approximate solution to the MWM problem with a performance ratio of $\frac{2}{3} - \epsilon$, where ϵ is an arbitrarily small number.

An approximation algorithm has a **performance ratio** of c , if for all graphs it finds a matching with a weight of at least c times the weight of an optimal solution.

The quality of an approximation algorithm for the weighted matching problem is measured by its performance ratio.



Definitions

Let M be a matching in a graph $G = (V, E)$.

If the weight of M is not maximum, then one can replace some set of edges of M by some other set of edges of E such that the new set thus obtained is again a matching and has strictly larger weight than M .


A process which removes the set R and adds a set $S \subset E$ to M is defined as an **augmentation**. The set S is called the augmenting set for this augmentation.

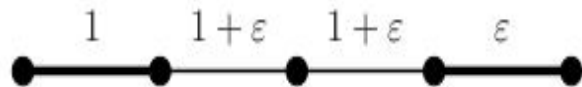


The **gain** of an augmentation with augmenting set S is defined as $w(S) - w(R)$, which is the increase of weight it achieves.

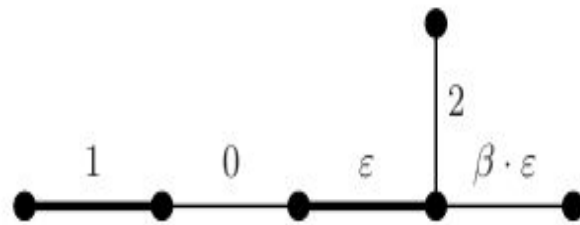
A **short augmentation** is an augmentation such that all edges in the augmenting set are adjacent to some edge $e \in E$. Such an edge e is called a **center** of this short augmentation.

For a constant $\beta > 1$, a **β -augmentation** is a short augmentation such that the augmenting set S has the property that $w(S) \geq \beta \cdot w(R)$.





a)



b)

Fig. 2. Examples motivating the use of β -augmentations. Edges belonging to the matching are shown in bold.

We must also be careful to pick the choices for the beta augmentation carefully, as with the wrong choice, the total weight of the matching may only increase by smaller values, leading to increase in the running time.

Hence, for the purposes of the algorithm, the β - augmentation with the largest possible gain should be selected. But, since selecting the best β - augmentation will take longer time, we strike a balance by selecting a “good” β - augmentation.

A β - augmentation with center e is called **good** if it achieves at least a $(\beta - 1)/(\beta - 1/2)$ fraction of the gain that a best β -augmentation with center e can achieve.



Main Algorithm

Algorithm `improve_matching` ($G = (V, E), w : E \rightarrow \mathbb{R}^+, M$)

```
1  make  $M$  maximal
2   $M' := M$ 
3  for  $e \in M$  do begin
4    if there exists a  $\beta$ -augmentation in  $M'$  with center  $e$ 
5      then augment  $M'$  by a good  $\beta$ -augmentation with center  $e$ 
6  end
7  return  $M'$ 
```



RESULTS

Edmond's Algorithm

```
with n = 50 , d = 0.7
[33, 32]
[36, 35]
[38, 37]
[41, 34]
[44, 11]
[44, 12]
[44, 43]
[49, 48]
[47, 17]
[47, 21]
[47, 16]
[47, 14]
[47, 11]
[47, 12]
```

Took 0.976536750793457 seconds
with n = 50 , d = 0.9

```
[34, 33]
[3, 35]
[37, 3]
[37, 1]
[37, 36]
[39, 3]
[39, 1]
[39, 38]
[11, 42]
[11, 43]
[45, 11]
[45, 9]
[45, 5]
[45, 1]
[45, 7]
[45, 3]
[45, 44]
[49, 48]
[47, 3]
```

Took 1.4810512065887451 seconds

Drake - Hougardy Approximation Algorithm

```
Drake_Hougardy_Matching.py x
Drake_Hougardy_Matching.py > getGoodBetaAugmentation
285 maxBetaAugmentation = auq2
PROBLEMS OUTPUT TERMINAL DEBUG CONSOLE

After Improve Matching weight: 469.9576935273208
iteration: 88
Ratio: 0.48360965071590406
Before Improve Matching weight: 469.9576935273208
After Improve Matching weight: 539.9588240327057
iteration: 89
Ratio: 0.555644267320089
Before Improve Matching weight: 539.9588240327057
After Improve Matching weight: 543.9209145969999
iteration: 90
Ratio: 0.5597214539696393
Before Improve Matching weight: 543.9209145969999
After Improve Matching weight: 460.1592082892291
iteration: 91
Ratio: 0.47352652602446205
Before Improve Matching weight: 460.1592082892291
After Improve Matching weight: 515.8700203892971
iteration: 92
Ratio: 0.5308557000158375
Before Improve Matching weight: 515.8700203892971
After Improve Matching weight: 579.876496984933
iteration: 93
Ratio: 0.5967215220170505
Before Improve Matching weight: 579.876496984933
After Improve Matching weight: 547.8990567980302
iteration: 94
Ratio: 0.5638151585452549
Before Improve Matching weight: 547.8990567980302
After Improve Matching weight: 501.44936971627646
iteration: 95
Ratio: 0.5160161390699751
Before Improve Matching weight: 501.44936971627646
After Improve Matching weight: 662.1445925618184
iteration: 96
Ratio: 0.6813794508369502

Improved weight : 662.1445925618184
Improved Ratio : 0.6813794508369502
Time taken(sec) : 4.669061
amithbn@amithbn-HP-Laptop-15-bs1xx:~/Desktop/Amith/5semIT/DAA/MiniP/MWM_approximation$
```

Drake - Hougardy Approximation Algorithm

```
PROBLEMS OUTPUT TERMINAL DEBUG CONSOLE 1: bash
amithbn@amithbn-HP-Laptop-15-bs1xx:~/Desktop/Amith/5semIT/DAA/MiniP/MWM_approximation$ python3 Drake_Hougardy_Matching.py
Exact Matching weight : 970.6078945306027

Edges in the final matching

[(16, 17), (18, 28), (30, 24), (19, 46), (43, 33), (40, 45), (32, 9), (29, 23), (27, 31), (21, 47), (5, 44), (49, 4), (42, 20), (0, 41), (10, 36), (25, 7), (13, 12)]

Improved weight : 659.4000106269167
Improved Ratio : 0.6793680685502874
Time taken(sec) : 1.149473
amithbn@amithbn-HP-Laptop-15-bs1xx:~/Desktop/Amith/5semIT/DAA/MiniP/MWM_approximation$
```




CONCLUSION

In this project, we have investigated the MWM problem and implemented two algorithms as its solutions : the Edmonds' Algorithm and the Drake-Hougardy Approximation Algorithm.

In the approximate algorithm, the weight of the matching obtained is 0.67 times the weight of the exact solution, as expected. The program takes around 0.1 seconds to output a matching for an input of 50 nodes on average.

In the blossom algorithm, we have obtained the running times for different number of nodes. We also print the final matching obtained.





THANK YOU