



Finding Maximum Matching in a Bipartite Graph

CS412
Algorithms: Design & Analysis
Fall 2020

Group Members

MUHAMMAD MUNAWWAR ANWAR (ma04289)

SYED MUHAMMAD FASIH HUSSAIN (sh05204)

MOID UL HUDA (mh05205)

SALMAN MUHAMMAD YOUNUS (sy04351)



Table of Contents

List of Figures	2
1 Introduction	3
1. Definition	3
2. Applications	3
2 Ford Fulkerson	4
1. Algorithm	4
2. Example	4
3. Complexity	7
4. Python Code.....	7
3 Hopcroft Karp	9
1. Algorithm	9
2. Example	9
3. Complexity	13
4. Python Code.....	14
4 Hungarian Matching	16
1. Algorithm	16
2. Example	16
3. Complexity	18
4. Python Code.....	18
5 Empirical Analysis	21
1. Results	21
2. Conclusion	21
3. Python Code.....	22



List of Figures

Figure 3.1: The original graph without any assigned matchings	10
Figure 3.2: Match a to 1.....	10
Figure 3.3: Delete all the instances of 1 and a found in the tree.....	10
Figure 3.4: Match b to 2 and delete b from the tree spanning from 3.....	10
Figure 3.5: The matching M at the end of the iteration 1	11
Figure 3.6: The yellow edges represent previously found matchings and the blue edges represent unmatched edges.....	12
Figure 3.7: DFS finds a path from c to 1, to a, and terminates at 3.	12
Figure 3.8: Now 1 is matched with c and 3 is matched with a.....	13
Figure 3.9: Maximal matching and termination of algorithm.....	13
Figure 4.1: Example: Bipartite Graph	17
Figure 4.2: Matching M in the graph	17
Figure 4.3: Alternating Path for the matching M	17
Figure 4.4: New Matching M_1 with more edges than M	18
Figure 5.1: Empirical Analysis of Algorithms for Finding Maximum Matching in a Bipartite Graph .	21



1 Introduction

1 Definition

A matching in a graph is a set of edges that do not have a set of common vertices. In other words, given a bipartite graph $G = (A \cup B, E)$, a matching, $M \subseteq A \times B$, is a subgraph of G where every node has a degree of at most 1.

A maximum matching is a matching of maximum size (maximum number of edges). In a maximum matching, if any edge is added to it, it is no longer a matching. There can be more than one maximum matchings for a given Bipartite Graph. In unweighted bipartite graphs, it is also known as the maximum cardinality matching. This report will discuss three algorithms used to find maximum matching in a bipartite graph and analyze them theoretically as well as empirically. The algorithms discussed are:

- Hopcroft Karp Algorithm
- Ford Fulkerson Algorithm
- Hungarian Algorithm

2 Applications

- **Stable Marriage Problem:** The purpose of the stable marriage problem is to facilitate matchmaking between two sets of people. Given a list of potential matches among an equal number of brides and grooms, the stable marriage problem gives a necessary and sufficient condition on the list for everyone to be married to an agreeable match. This theorem can be applied to any situation where two sets of vertices must be matched together so as to maximize utility.
- **Transportation Theory:** It is used for resource allocation and optimization in travel.
- **Neural Networks in Artificial Intelligence**



2 Ford Fulkerson

1 Algorithm

The problem of finding the maximum matching for a bipartite graph G such that $G = (V, E) = ((A \cup B), E)$, here A and B are disjoint sets of vertices of graph G .

To convert a bipartite graph into a network N we add two new vertices s and t to graph G which are source and sink of network N respectively. We create an edge from s to every vertex in A and similarly we create an edge from every vertex in B to t . Assign the capacity of each edge in the network N to be 1.

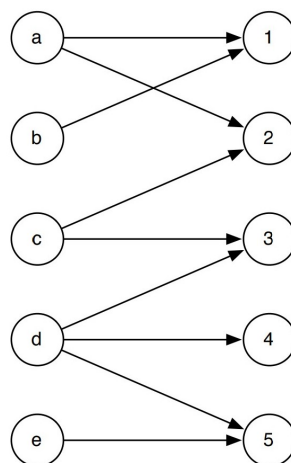
Once the bipartite graph G is converted into network N , the maximum matching for G will be the max flow of N . To find the max flow of the network Ford Fulkerson algorithm is used. Following is the Ford Fulkerson algorithm for finding maximum flow of a network:

1. Set initial flow as 0.
2. While there exists a augmenting path from source to sink, add path-flow to the flow.
3. Return flow

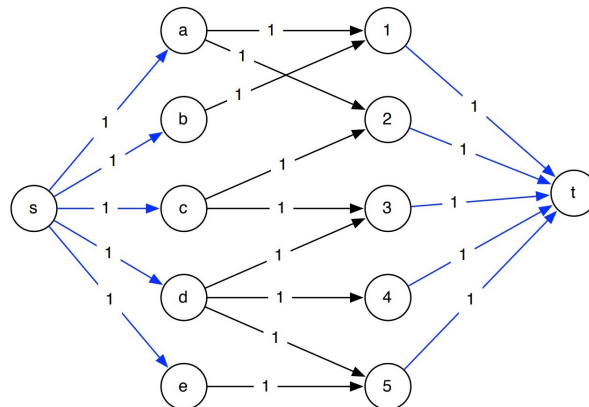
Augmenting path is a path from source to sink with available capacity at every edge.

2 Example

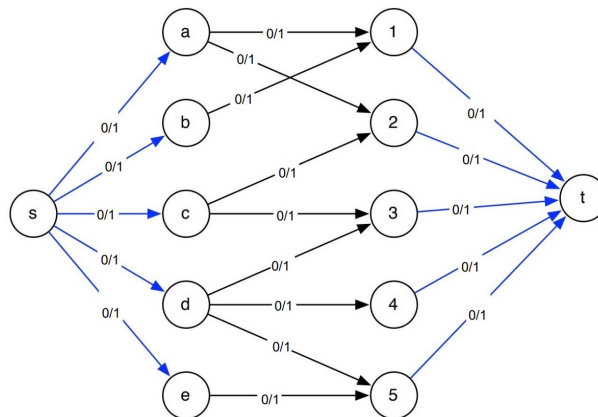
Consider the following graph:



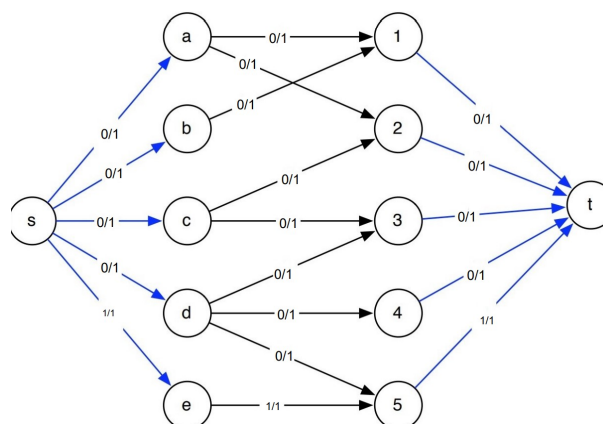
The graph is then converted into following network:



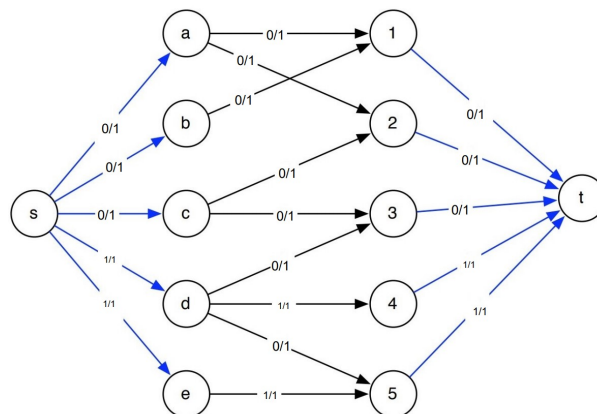
Now set initialize flow to 0 and write each edge as *flow/capacity* as shown:



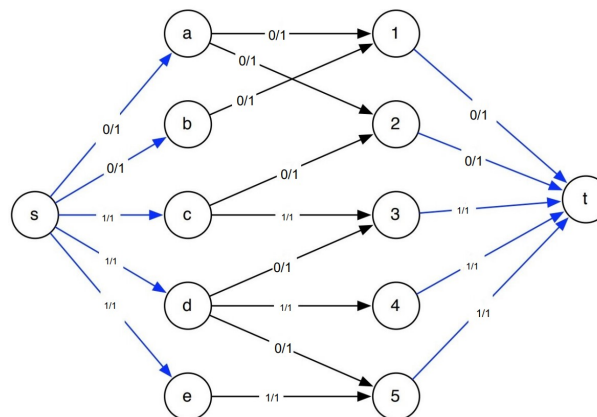
Take the augmented path $s \rightarrow e \rightarrow 5 \rightarrow t$ having bottle neck capacity 1, now flow is 1:



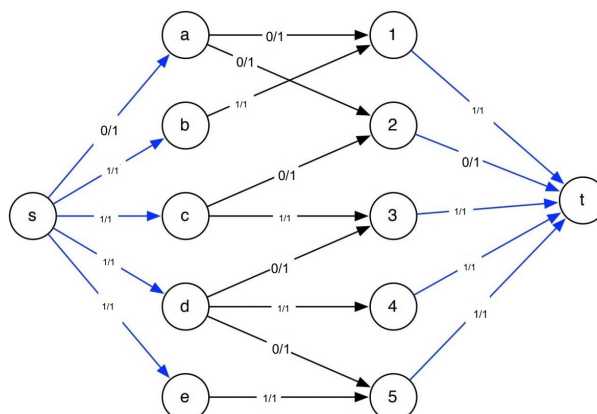
Take the augmented path $s \rightarrow d \rightarrow 4 \rightarrow t$ having bottle neck capacity 1, now flow is 2:



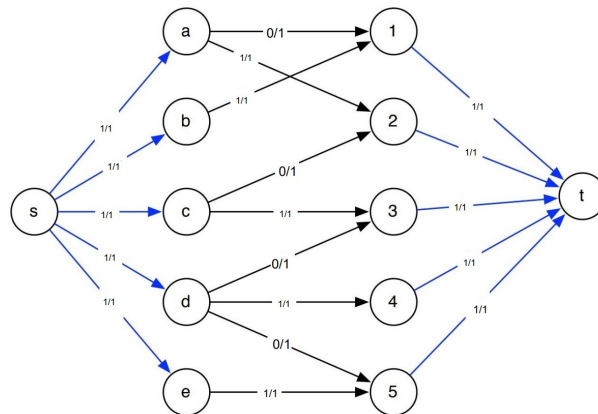
Take the augmented path $s \rightarrow c \rightarrow 3 \rightarrow t$ having bottle neck capacity 1, now flow is 3:



Take the augmented path $s \rightarrow b \rightarrow 1 \rightarrow t$ having bottle neck capacity 1, now flow is 4:



Take the augmented path $s \rightarrow a \rightarrow 2 \rightarrow t$ having bottle neck capacity 1, now flow is 5:



As there is no augmented path in N therefore flow of N is 5 and maximum matching of G is 5.

3 Complexity

The time complexity of Ford Fulkerson algorithm is $O(m'C)$ where m' is the number of edges and C is number of edges leaving s . The number of edges leaving s will always be less than or equal to number of vertices in G so $C = |V|$.

N has all the vertices of G and each vertex in G , N has one additional vertex, so $m' = |E| + |V|$.

$$O(m'C) = O((|V| + |E|) \times |V|) = O(|V| \times |E|)$$

4 Python Code

```

1 from collections import defaultdict
2 class Graph:
3
4     def __init__(self, graph):
5         self.graph = graph # residual graph
6         self.ROW = len(graph)
7
8
9     def BFS(self, s, t, parent):
10
11         # Mark all the vertices as not visited
12         visited = [False] * (self.ROW)
13         queue = []
14
15         queue.append(s)
16         visited[s] = True
17
18         while queue:
19

```




```

20         u = queue.pop(0)
21
22         for ind, val in enumerate(self.graph[u]):
23             if visited[ind] == False and val > 0 :
24                 queue.append(ind)
25                 visited[ind] = True
26                 parent[ind] = u
27
28         return True if visited[t] else False
29
30
31     # Returns the maximum flow from s to t in the given graph
32     def FordFulkerson(self, source, sink):
33
34         parent = [-1]*(self.ROW)
35
36         max_flow = 0 # There is no flow initially
37
38         while self.BFS(source, sink, parent) :
39
40             path_flow = float("Inf")
41             s = sink
42             while(s != source):
43                 path_flow = min (path_flow, self.graph[parent[s]][s])
44                 s = parent[s]
45
46             max_flow += path_flow
47
48             v = sink
49             while(v != source):
50                 u = parent[v]
51                 self.graph[u][v] -= path_flow
52                 self.graph[v][u] += path_flow
53                 v = parent[v]
54
55             return max_flow, parent
56
57
58 import timeit
59 import numpy as np
60 content = np.loadtxt('file1.txt').tolist()
61 graph = content
62
63 g = Graph(graph)
64
65 source = 0; sink = 5
66 start = timeit.default_timer()
67 flow, parent = g.FordFulkerson(source, sink)
68 stop = timeit.default_timer()
69 print('Time: ', stop - start)

```

Code used from <https://www.geeksforgeeks.org/ford-fulkerson-algorithm-for-maximum-flow-problem/>



3 Hopcroft Karp

The Hopcroft-Karp Algorithm is an algorithm that takes a bipartite graph $G(E, V)$ and outputs a maximum matching M . It's worst case Complexity is $O(|E| \sqrt{V})$.

Hopcroft-Karp repeatedly increases the size of a partial matching by determining augmenting paths. The Hopcroft-Karp improves finds a maximal set of shortest augmenting paths per iteration, and increases the augmenting path by the maximum flow rather than one by one.

1 Algorithm

The algorithm runs in phases made up of the following steps

- Use breadth-first search to find augmenting paths. It partitions the vertices of the graph into **layers** of matched and unmatched edges. For the search, start with the free nodes in X . This forms the first layer of partitioning. The search finishes at the first layer k where one or more free nodes in Y are reached.
- The free nodes in Y are added to a set called A . This means that any node added to A will be the ending node of an augmenting path and – a shortest augmenting path since the breadth-first search finds shortest paths.
- Once an augmenting path is found, depth first search is used to add augmenting paths to current matching M . At any given layer, the depth first search will follow edges that lead to an unused node from the previous layer. Paths in the depth-first paths must be alternating paths. Once the algorithm finds an augmenting path that uses a node from A , the depth first search moves on to the next starting vertex.

The algorithm terminates when the algorithm can no more find augmenting paths in the breadth first search step.

2 Example

Observe the graph with edges but no assigned matchings.

Perform BFS starting at all the numeric vertices without a match. Pick any unmatched leaf and go all the way back to a root using DFS. Match the leaf to the root.

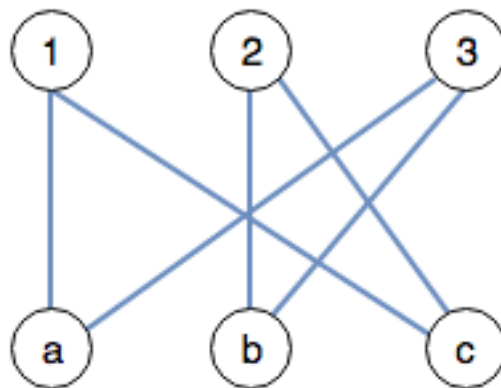


Figure 3.1: The original graph without any assigned matchings

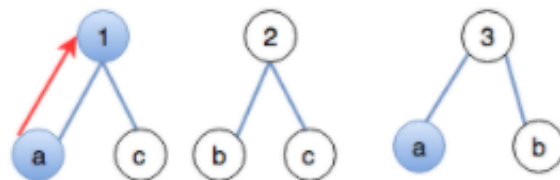


Figure 3.2: Match a to 1

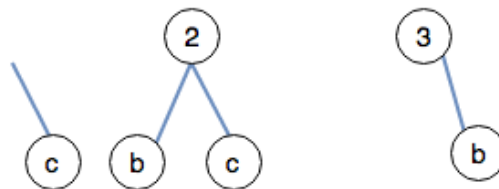


Figure 3.3: Delete all the instances of 1 and a found in the tree

Repeat the process to find the next matching.

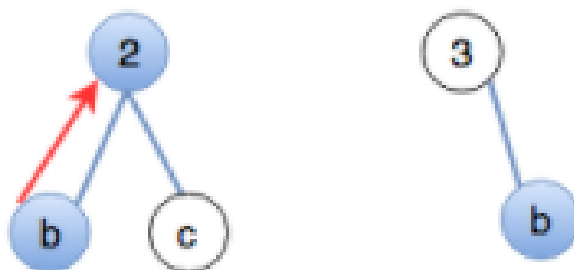


Figure 3.4: Match b to 2 and delete b from the tree spanning from 3.

In this iteration, 1 is matched to a, 2 is matched to b, and 3, along with c, is left without a match.

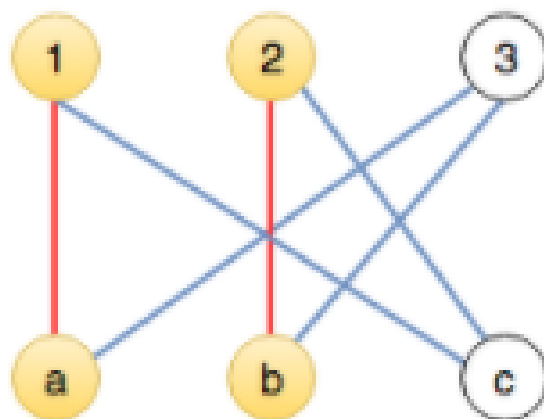


Figure 3.5: The matching M at the end of the iteration 1

Since 3 is left without a match, perform BFS starting from 3 in order to find a matching, or assert that the current matching is already optimal.

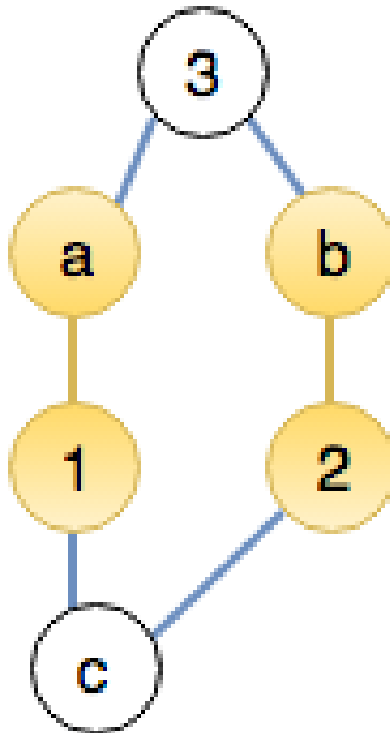


Figure 3.6: The yellow edges represent previously found matchings and the blue edges represent unmatched edges.

Perform DFS once again from the unmatched leaf all the way to the root.

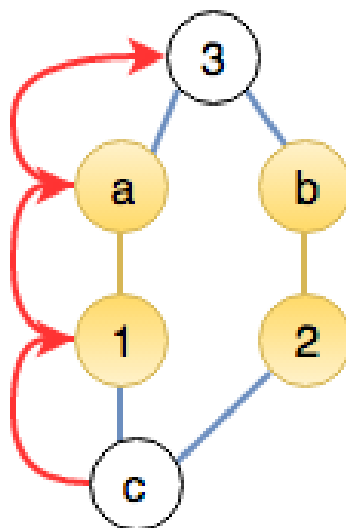


Figure 3.7: DFS finds a path from c to 1, to a, and terminates at 3.

Augment the path by switching the edges that are matched with those that are unmatched.

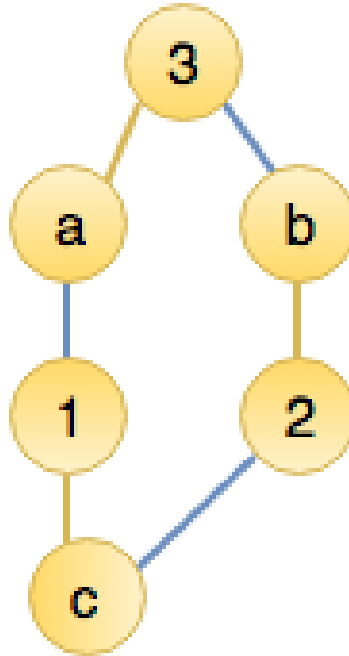


Figure 3.8: Now 1 is matched with c and 3 is matched with a.

This produces the maximal matching between numbers and letters in the graph.

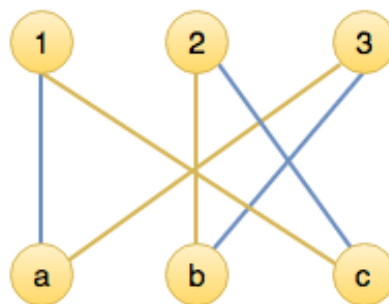


Figure 3.9: Maximal matching and termination of algorithm

3 Complexity

Each iteration of the Hopcroft-Karp algorithm executes breadth-first search and depth-first search once. This takes $O(E)$ time. Since each iteration of the algorithm finds the maximal set of shortest augmenting paths by eliminating a path from the root to an unmatched edge, there are at most $O(\sqrt{V})$ iterations needed.



4 Python Code

```

1 def bipartiteMatch(graph):
2
3     # initialize greedy matching
4     matching = {}
5     for u in range(len(graph)):
6         for v in range(len(graph[u])):
7             if graph[u][v] != 0 and v not in matching:
8                 matching[v] = u
9                 break
10
11     while 1:
12         preds = {}
13         unmatched = []
14         pred = dict([(u, unmatched) for u in range(len(graph))])
15         for v in matching:
16             del pred[matching[v]]
17         layer = list(pred)
18
19         while layer and not unmatched:
20             newLayer = {}
21             for u in layer:
22                 for v in range(len(graph[u])):
23                     if graph[u][v] != 0 and v not in preds:
24                         newLayer.setdefault(v, []).append(u)
25             layer = []
26             for v in newLayer:
27                 preds[v] = newLayer[v]
28                 if v in matching:
29                     layer.append(matching[v])
30                 pred[matching[v]] = v
31             else:
32                 unmatched.append(v)
33
34         # did we finish layering without finding any alternating paths?
35         if not unmatched:
36             unlayered = {}
37             for u in range(len(graph)):
38                 for v in range(len(graph[u])):
39                     if graph[u][v] != 0 and v not in preds:
40                         unlayered[v] = None
41             return (matching, list(pred), list(unlayered))
42
43         # recursively search backward through layers to find alternating
44         # recursion returns true if found path, false otherwise
45         def recurse(v):
46             if v in preds:
47                 L = preds[v]
48                 del preds[v]
49                 for u in L:
50                     if u in pred:
51                         pu = pred[u]
52                         del pred[u]
53                         if pu is unmatched or recurse(pu):

```



```
54         matching[v] = u
55         return 1
56     return 0
57
58     for v in unmatched: recurse(v)
59
60 if __name__ == '__main__':
61     import timeit
62     import numpy as np
63     from networkx.algorithms import bipartite as nx
64     from operator import itemgetter
65
66     G = nx.random_graph(1000,2000,0.2)
67     partition_l = set(map(itemgetter(0),G.edges()))
68
69     content = nx.biadjacency_matrix(G,partition_l).toarray().tolist()
70
71     start = timeit.default_timer()
72     bipartiteMatch(content)
73     stop = timeit.default_timer()
74
75     print('Time: ', stop - start)
```




4 Hungarian Matching

1 Algorithm

The Hungarian Maximum Matching Algorithm is quite an easy approach to finding out the maximum matching in a bipartite graph. The algorithm is based on finding augmenting paths by Breadth First Search. The algorithm starts with finding any matching M in the graph and then finding an augmenting path to that matching which is a path starting from an *unmatched* vertex and connecting to a *matched* vertex and the path alternated with one unmatched and one matched edge and ends at a matched vertex.

This augmenting path is found by doing a Breadth First Search on the bipartite graph from a vertex which is unmatched and is connected to a matched vertex in the graph. After finding an augmenting path P for the matching M , we find the symmetric difference of P with M and we get a matching M_1 which has one edge greater than M . Then we improve the labelling on our edges to make sure there are minimum edge lengths in our matching. This process is repeated until there is no augmenting paths remaining in the graph for the corresponding matching M_k . This matching M_k is the maximum matching in the bipartite graph and hence a maximum matching is achieved.

1. Start with any matching M in the bipartite graph.
2. Find an augmenting path for that matching M
3. Flip the augmenting path to get a new matching which is one more edge than the previous matching.
4. Improve the labelling of the edges.
5. Repeat the process until there is no augmenting path remaining in the graph for the corresponding matching.

2 Example

Let us consider the following example of a bipartite graph. There are 5 vertices in each partition of the graph as can be seen in the figure 4.1.

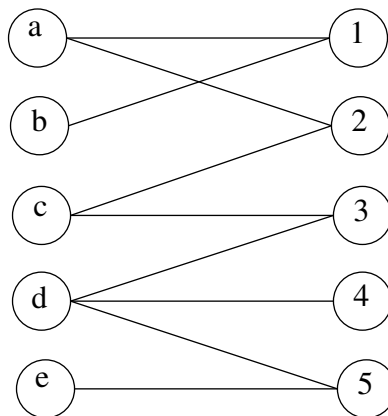


Figure 4.1: Example: Bipartite Graph

Lets look at an example with the matching M as can be seen in the figure 4.2

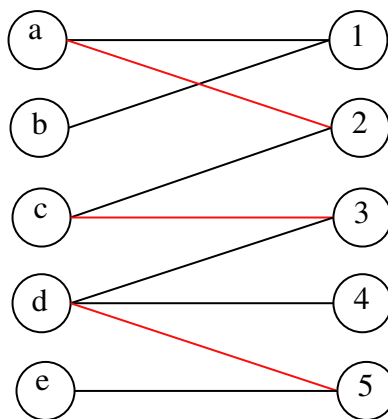


Figure 4.2: Matching M in the graph

Now we find an augmenting path for the matching M in the graph starting from an unmatched edge and ending at an unmatched edge. The augmenting path for matching M can be seen in figure 4.3

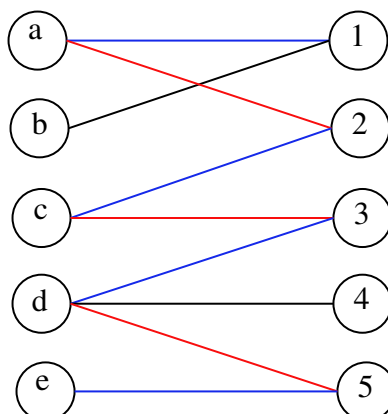


Figure 4.3: Alternating Path for the matching M



After finding the augmenting path P , we take the symmetric difference of P and M to get a matching M_1 which is one more edge than M . As can be seen in figure 4.4

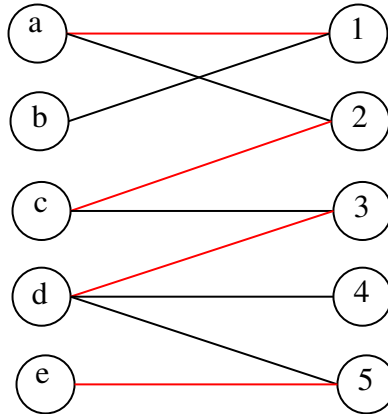


Figure 4.4: New Matching M_1 with more edges than M

3 Complexity

The time complexity of the algorithm can be calculated by dividing it into the steps. The main step of the algorithm is to find a matching in each step which is one edge greater than the previous matching. This has the complexity of $O(V)$ because we are increasing one edge per iteration. The steps performed in the iteration are finding the augmenting path for the matching and then flipping that augmenting path to get the new matching. Each of these steps take up $O(V)$ time and the last step is to improve the labelling of the edges which might take upto $O(V^2)$ time and this whole process hence take the time of $O(V^3)$ time.

4 Python Code

```

1 def improveLabels(val):
2     """ change the labels, and maintain minSlack.
3     """
4     for u in S:
5         lu[u] -= val
6     for v in V:
7         if v in T:
8             lv[v] += val
9         else:
10            minSlack[v][0] -= val
11
12 def improveMatching(v):
13     """ apply the alternating path from v to the root in the tree.
14     """
15     u = T[v]
16     if u in Mu:
17         improveMatching(Mu[u])
18     Mu[u] = v
19     Mv[v] = u

```



```

20
21 def slack(u,v):
22     return lu[u]+lv[v]-w[u][v]
23
24 def augment():
25     """ augment the matching, possibly improving the labels on the way.
26     """
27     while True:
28         # select edge (u,v) with u in S, v not in T and min slack
29         ((val, u), v) = min([(minSlack[v], v) for v in V if v not in T])
30         assert u in S
31         if val>0:
32             improveLabels(val)
33         # now we are sure that (u,v) is saturated
34         assert slack(u,v)==0
35         T[v] = u
36         if v in Mv:
37             u1 = Mv[v]
38             assert not u1 in S
39             S[u1] = True
40             for v in V:
41                 if not v in T and minSlack[v][0] > slack(u1,v):
42                     minSlack[v] = [slack(u1,v), u1]
43         else:
44             improveMatching(v)
45         return
46
47 def maxWeightMatching(weights):
48     """ given w, the weight matrix of a complete bipartite graph,
49     returns the mappings Mu : U->V ,Mv : V->U encoding the matching
50     as well as the value of it.
51     """
52     global U,V,S,T,Mu,Mv,lu,lv, minSlack, w
53     w = weights
54     n = len(w)
55     U = V = range(n)
56     lu = [ max([w[u][v] for v in V]) for u in U]
57     lv = [ 0 for v in V]
58     Mu = {}
59     Mv = {}
60     while len(Mu)<n:
61         free = [u for u in V if u not in Mu]
62         u0 = free[0]
63         S = {u0: True}
64         T = {}
65         minSlack = [[slack(u0,v), u0] for v in V]
66         augment()
67         val = sum(lu)+sum(lv)
68         return (Mu, Mv, val)
69
70
71 if __name__ == '__main__':
72     import timeit
73     import numpy as np
74     content = np.loadtxt('file1.txt').tolist()
75     # print(len(content))

```



```
76     start = timeit.default_timer()
77     maxWeightMatching(content)
78
79     stop = timeit.default_timer()
80
81     print('Time: ', stop - start)
```

5 Empirical Analysis

1 Results

The three algorithms were testing on randomly generated bipartite graphs with different number of vertices in both the partitions and a some defined probability of having an edge between two vertices of the partitions. The algorithms were run for the graph sizes starting from 500 and with a step of 250 all the way to 3250 vertices where it was pretty difficult to let the maximum running algorithm to stay working. This experiment was performed 5 times for the same number of vertices as mentioned before and then the running time for the calculated experiment was taken an average and the results were plotted in the figure 5.1.

The curves obtained were then fitted for the best fit curves and the curve for the Hungarian Algorithm was fitted on a polynomial function of degree 3. The curve for the Ford-Fulkerson Algorithm was fitted on a polynomial function of degree 2 and the Hopcroft-Karp Algorithm's curve was fitted on a curve of polynomial with degree 1.5.

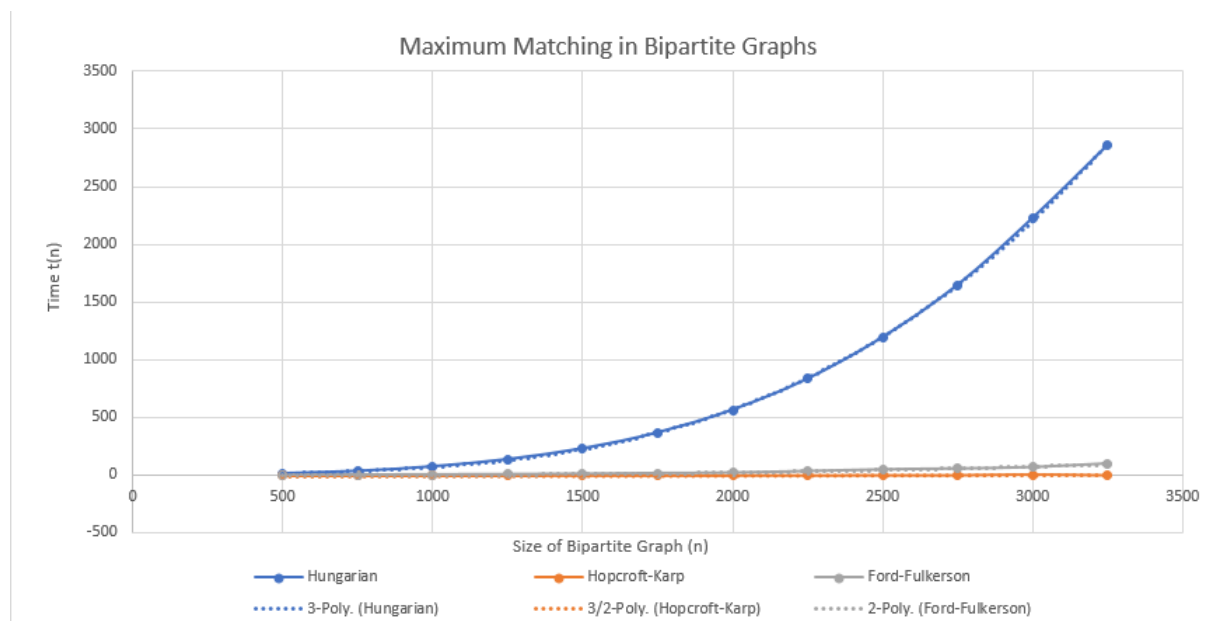


Figure 5.1: Empirical Analysis of Algorithms for Finding Maximum Matching in a Bipartite Graph

2 Conclusion

The results mentioned above are in correspondence with the respective theoretical time complexities mentioned above in the respective algorithms sections. This shows that the theoretical time complexity analysis of the algorithms was on par with the experiments generated with the randomly generated graphs and hence it concludes that the Hopcroft-Karp Algorithm gives us the maximum matching in a bipartite graph in the minimum time out of these algorithms.



3 Python Code

```

1
2 class Edge(object):
3     def __init__(self, u, v, w):
4         self.source = u
5         self.target = v
6         self.capacity = w
7
8     def __repr__(self):
9         return "%s->%s:%s" % (self.source, self.target, self.capacity)
10
11
12 class FlowNetwork(object):
13     def __init__(self):
14         self.adj = {}
15         self.flow = {}
16
17     def AddVertex(self, vertex):
18         self.adj[vertex] = []
19
20     def GetEdges(self, v):
21         return self.adj[v]
22
23     def AddEdge(self, u, v, w = 0):
24         if u == v:
25             raise ValueError("u == v")
26         if u not in self.adj.keys():
27             self.adj[u] = []
28         if v not in self.adj.keys():
29             self.adj[v] = []
30         edge = Edge(u, v, w)
31         redge = Edge(v, u, 0)
32         edge.redge = redge
33         redge.redge = edge
34         self.adj[u].append(edge)
35         self.adj[v].append(redge)
36         # Intialize all flows to zero
37         self.flow[edge] = 0
38         self.flow[redge] = 0
39
40     def FindPath(self, source, target, path):
41         if source == target:
42             return path
43         for edge in self.GetEdges(source):
44             residual = edge.capacity - self.flow[edge]
45             if residual > 0 and not (edge, residual) in path:
46                 result = self.FindPath(edge.target, target, path + [(edge, residual)])
47                 if result != None:
48                     return result
49
50     def MaxFlow(self, source, target):
51         path = self.FindPath(source, target, [])
52         # print('path after enter MaxFlow: %s' % path)
53         # for key in self.flow:

```



```

54 # print('%s:%s' % (key,self.flow[key]))
55 # print('-' * 20)
56 while path != None:
57     flow = min(res for edge, res in path)
58     for edge, res in path:
59         self.flow[edge] += flow
60         self.flow[edge.ledge] -= flow
61     # for key in self.flow:
62     #     print('%s:%s' % (key,self.flow[key]))
63     path = self.FindPath(source, target, [])
64     # print('path inside of while loop: %s' % path)
65     # for key in self.flow:
66     #     print('%s:%s' % (key,self.flow[key]))
67     return sum(self.flow[edge] for edge in self.GetEdges(source))
68
69 def bipartiteMatch(graph):
70     '''Find maximum cardinality matching of a bipartite graph (U,V,E).
71     The input format is a dictionary mapping members of U to a list
72     of their neighbors in V. The output is a triple (M,A,B) where M is a
73     dictionary mapping members of V to their matches in U, A is the part
74     of the maximum independent set in U, and B is the part of the MIS in V.
75     The same object may occur in both U and V, and is treated as two
76     distinct vertices if this happens.'''
77
78     # initialize greedy matching (redundant, but faster than full search)
79     matching = {}
80     for u in range(len(graph)):
81         for v in range(len(graph[u])):
82             if graph[u][v] != 0 and v not in matching:
83                 matching[v] = u
84                 break
85
86     while 1:
87         # structure residual graph into layers
88         # pred[u] gives the neighbor in the previous layer for u in U
89         # preds[v] gives a list of neighbors in the previous layer for v in
90         # V
91         # unmatched gives a list of unmatched vertices in final layer of V,
92         # and is also used as a flag value for pred[u] when u is in the
93         # first layer
94         preds = {}
95         unmatched = []
96         pred = dict([(u,unmatched) for u in range(len(graph))])
97         for v in matching:
98             del pred[matching[v]]
99         layer = list(pred)
100
101         # repeatedly extend layering structure by another pair of layers
102         while layer and not unmatched:
103             newLayer = {}
104             for u in layer:
105                 for v in range(len(graph[u])):
106                     if graph[u][v] != 0 and v not in preds:
107                         newLayer.setdefault(v, []).append(u)
108             layer = []
109             for v in newLayer:

```




```

108         preds[v] = newLayer[v]
109         if v in matching:
110             layer.append(matching[v])
111             pred[matching[v]] = v
112         else:
113             unmatched.append(v)
114
115     # did we finish layering without finding any alternating paths?
116     if not unmatched:
117         unlayered = {}
118         for u in range(len(graph)):
119             for v in range(len(graph[u])):
120                 if graph[u][v] != 0 and v not in preds:
121                     unlayered[v] = None
122             return (matching, list(pred), list(unlayered))
123
124     # recursively search backward through layers to find alternating
125     # recursion returns true if found path, false otherwise
126     def recurse(v):
127         if v in preds:
128             L = preds[v]
129             del preds[v]
130             for u in L:
131                 if u in pred:
132                     pu = pred[u]
133                     del pred[u]
134                     if pu is unmatched or recurse(pu):
135                         matching[v] = u
136                         return 1
137         return 0
138
139     for v in unmatched: recurse(v)
140
141 import sys
142
143 if __name__ == '__main__':
144     import timeit
145     import numpy as np
146     from operator import itemgetter
147     from networkx.algorithms import bipartite as nx
148     import networkx.convert as nc
149     from hungarian_algorithm import algorithm
150     from networkx.algorithms import flow as fl
151     lst = []
152     for k in range(500, 10000, 500):
153         a = int(np.random.uniform(1, k))
154         lst.append((a, k-a))
155     for i, j in lst:
156         G = nx.random_graph(i, j, 1)
157         partition_l = set(map(itemgetter(0), G.edges()))
158         content = nx.bijacency_matrix(G, partition_l).toarray().tolist()
159         # print(len(content))
160
161     start = timeit.default_timer()
162     bipartiteMatch(content)

```



```
163     stop = timeit.default_timer()
164
165     print('Hopcroft-Karp Time: n: ',i+j, stop - start)
166
167     G = nx.random_graph((i+j)//2, (i+j)//2,1)
168     start = timeit.default_timer()
169     algorithm.find_matching(G,matching_type = 'max', return_type = '
list')
170
171     stop = timeit.default_timer()
172
173     print('Hungarian Time: n: ',i+j, stop - start)
```



References

“Matching Algorithms (Graph Theory).” Brilliant Math Science Wiki, <https://brilliant.org/wiki/matching-algorithms/>

Cormen, T. H., Leiserson, C. E., Rivest, R. L., Stein, C. (n.d.). Introduction to algorithms

Kleinberg, J., Tardos, E. (2014). Algorithm design. Harlow: Pearson.

Kingsford, C., (n.d.), Maximum Bipartite Matching

Lecture Notes.