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)

https://github.com/MoidHuda/CS-412-Algorithms-Project

Table of Contents

List of Figures

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 $\subseteq A$ x 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.

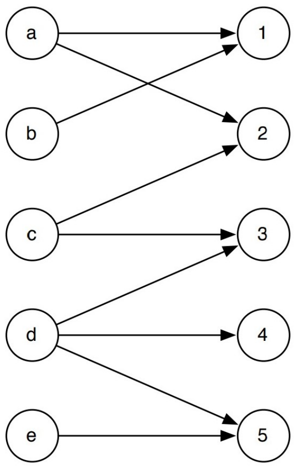
Once the bipartite graph G is converted into network N, the maximum matching for G will be the max flow of N. To find the max flow of the network Ford Fulkerson algorithm is used. Following is the Ford Fulkerson algorithm for finding maximum flow of a network:

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

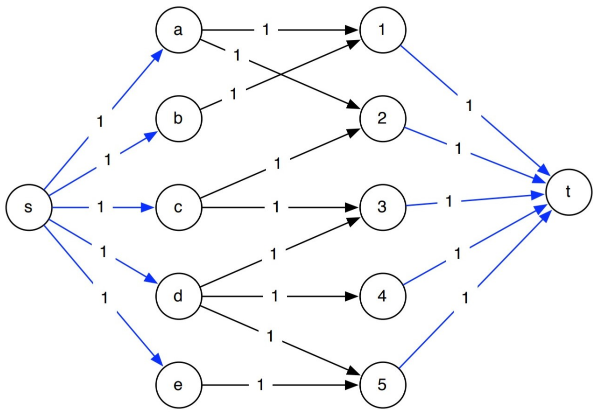
Augmenting path is a path from source to sink with available capacity at every edge.

2 Example

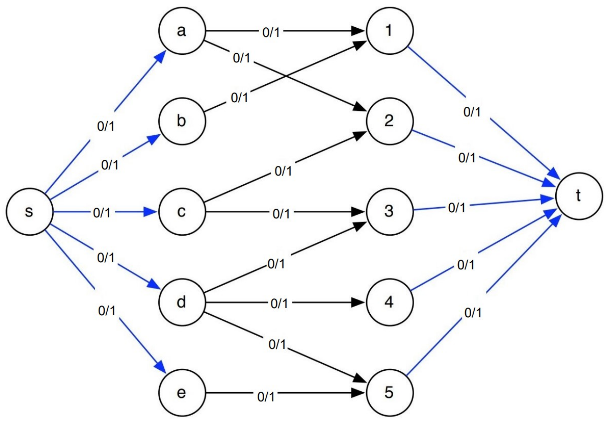
Consider the following graph:



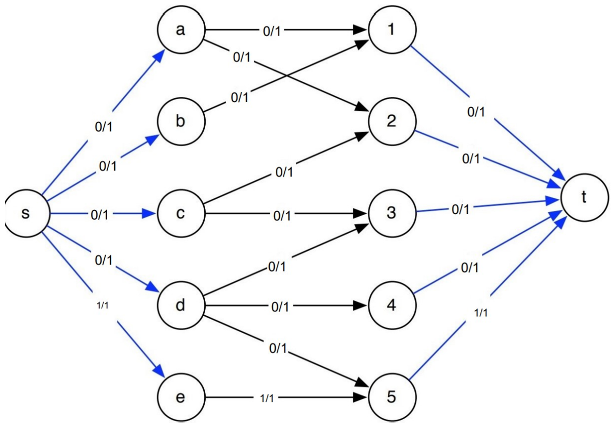
The graph is then converted into following network:



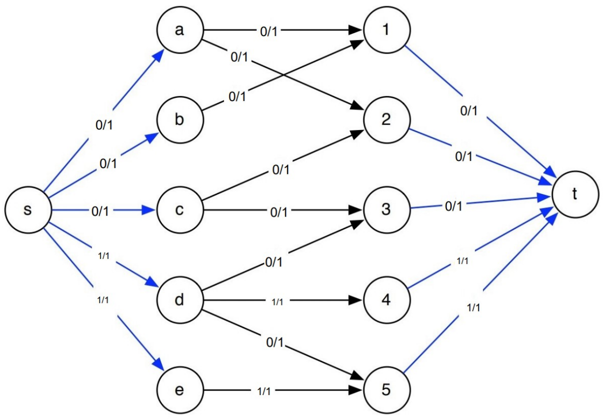
Now set initialize flow to 0 and write each edge as *flow*/*capacity* as shown:



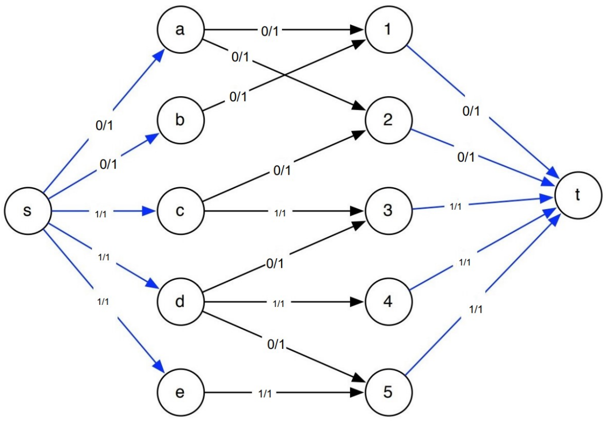
Take the augmented path $s \rightarrow e \rightarrow 5 \rightarrow t$ having bottle neck capacity 1, now flow is 1:



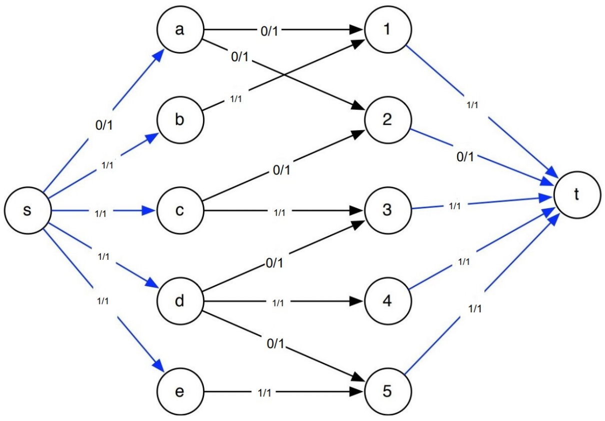
Take the augmented path $s \rightarrow d \rightarrow 4 \rightarrow t$ having bottle neck capacity 1, now flow is 2:



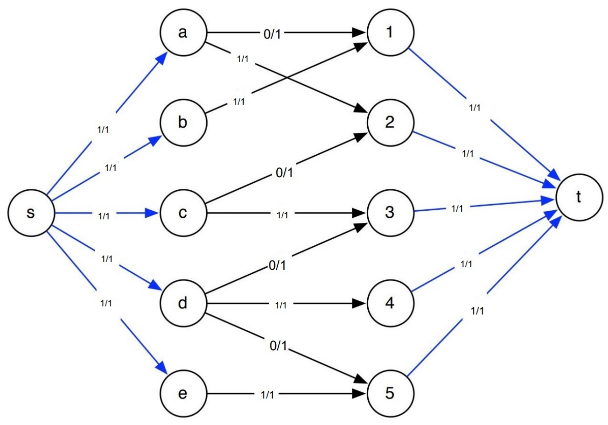
Take the augmented path $s \to c \to 3 \to t$ having bottle neck capacity 1, now flow is 3:



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



Take the augmented path $s \to a \to 2 \to t$ 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

```python
from collections import defaultdict
class Graph:

    def __init__(self,graph):
        self.graph = graph # residual graph
        self. ROW = len(graph)


    def BFS(self,s, t, parent):

        # Mark all the vertices as not visited
        visited =[False]*(self.ROW)
        queue=[]

        queue.append(s)
        visited[s] = True

        while queue:

            u = queue.pop(0)

            for ind, val in enumerate(self.graph[u]):
                if visited[ind] == False and val > 0 :
                    queue.append(ind)
                    visited[ind] = True
                    parent[ind] = u

        return True if visited[t] else False


    # Returns tne maximum flow from s to t in the given graph
    def FordFulkerson(self, source, sink):

        parent = [-1]*(self.ROW)

        max_flow = 0 # There is no flow initially

        while self.BFS(source, sink, parent) :

            path_flow = float("Inf")
            s = sink
            while(s != source):
                path_flow = min (path_flow, self.graph[parent[s]][s])
                s = parent[s]

            max_flow += path_flow

            v = sink
            while(v != source):
                u = parent[v]
                self.graph[u][v] -= path_flow
                self.graph[v][u] += path_flow
                v = parent[v]

        return max_flow,parent


import timeit
import numpy as np
content = np.loadtxt('file1.txt').tolist()
graph = content

g = Graph(graph)

source = 0; sink = 5
start = timeit.default_timer()
flow, parent = g.FordFulkerson(source, sink)
stop = timeit.default_timer()
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 $O(\mid E \mid \sqrt{V})$.
Hopcroft-Karp repeatedly increases the size of a partial matchching by determing 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 algorithim runs in phases made up of the following steps

- Use breadth-first search to find augmenting paths. It partitons 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 unsused 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 form A, the depth first search moves on to the next statting 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.

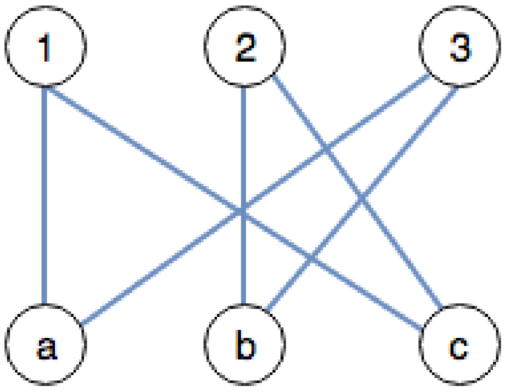


Figure 3.1: The original graph without any assigned matchings

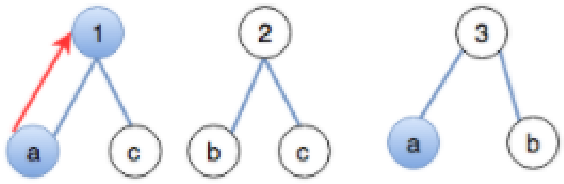


Figure 3.2: Match a to 1

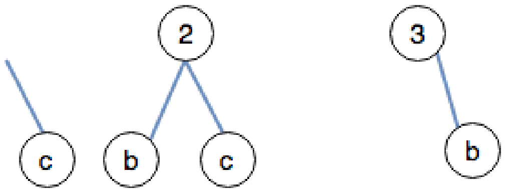


Figure 3.3: Delete all the instances of 1 and a found in the tree

Repeat the process to find the next matching.

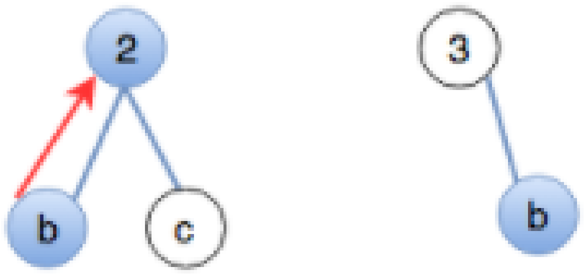


Figure 3.4: Match b to 2 and delete b from the tree spanning from 3.

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

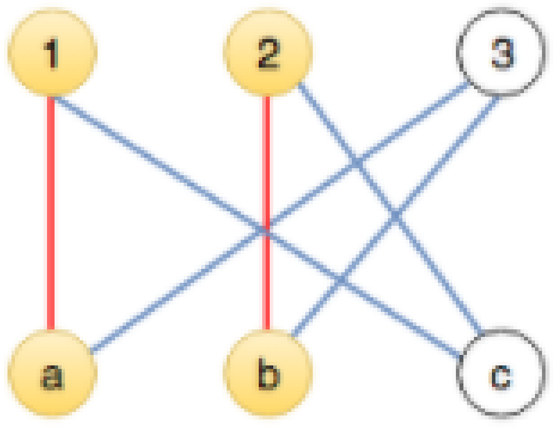


Figure 3.5: The matching M at the end of the iteration 1

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.

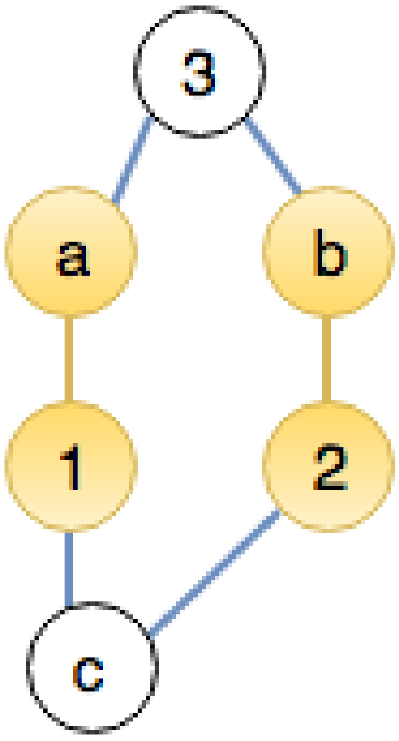


Figure 3.6: The yellow edges represent previously found matchings and the blue edges represent unmatched edges.

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

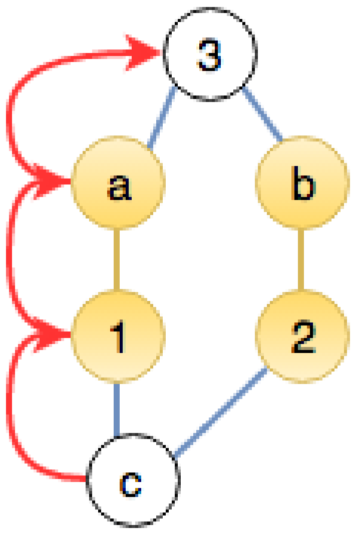


Figure 3.7: DFS finds a path from c to 1, to a, and terminates at 3.

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

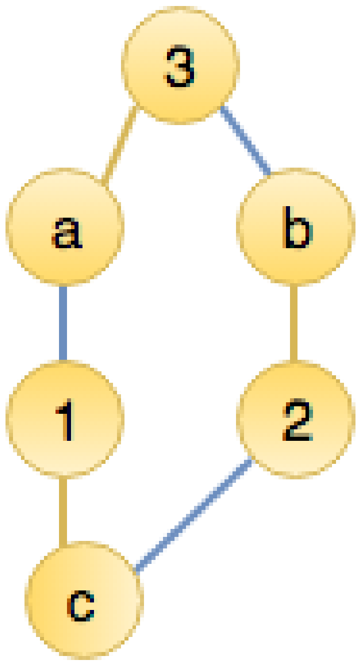


Figure 3.8: Now 1 is matched with c and 3 is matched with a.

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

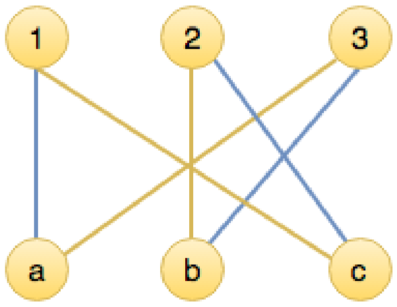


Figure 3.9: Maximal matching and termination of algorithm

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

```python
def bipartiteMatch(graph):

    # initialize greedy matching
    matching = {}
    for u in range(len(graph)):
        for v in range(len(graph[u])):
            if graph[u][v] != 0 and v not in matching:
                matching[v] = u
                break

    while 1:
        preds = {}
        unmatched = []
        pred = dict([(u,unmatched) for u in range(len(graph))])
        for v in matching:
            del pred[matching[v]]
        layer = list(pred)

        while layer and not unmatched:
            newLayer = {}
            for u in layer:
                for v in range(len(graph[u])):
                    if graph[u][v] != 0 and v not in preds:
                        newLayer.setdefault(v,[]).append(u)
            layer = []
            for v in newLayer:
                preds[v] = newLayer[v]
                if v in matching:
                    layer.append(matching[v])
                    pred[matching[v]] = v
                else:
                    unmatched.append(v)

        # did we finish layering without finding any alternating paths?
        if not unmatched:
            unlayered = {}
            for u in range(len(graph)):
                for v in range(len(graph[u])):
                    if graph[u][v] != 0 and v not in preds:
                        unlayered[v] = None
            return (matching,list(pred),list(unlayered))

        # recursively search backward through layers to find alternating
   paths
        # recursion returns true if found path, false otherwise
        def recurse(v):
            if v in preds:
                L = preds[v]
                del preds[v]
                for u in L:
                    if u in pred:
                        pu = pred[u]
                        del pred[u]
                        if pu is unmatched or recurse(pu):
                            matching[v] = u
                            return 1
            return 0

        for v in unmatched: recurse(v)

if __name__ == '__main__':
    import timeit
    import numpy as np
    from  networkx.algorithms import bipartite as nx
    from operator import itemgetter

    G = nx.random_graph(1000,2000,0.2)
    partition_l =   set(map(itemgetter(0),G.edges()))

    content = nx.biadjacency_matrix(G,partition_l).toarray().tolist()

    start = timeit.default_timer()
    bipartiteMatch(content)
    stop = timeit.default_timer()

    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.

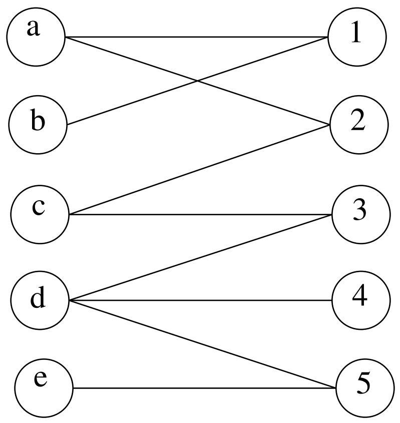


Figure 4.1: Example: Bipartite Graph

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

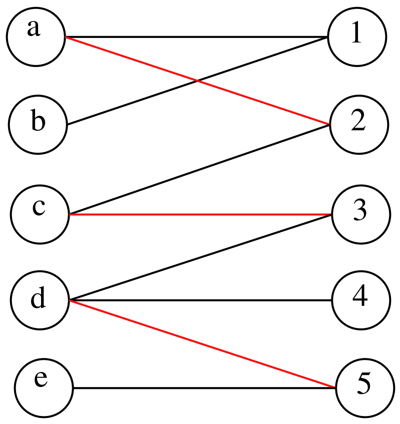


Figure 4.2: Matching M in the graph

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

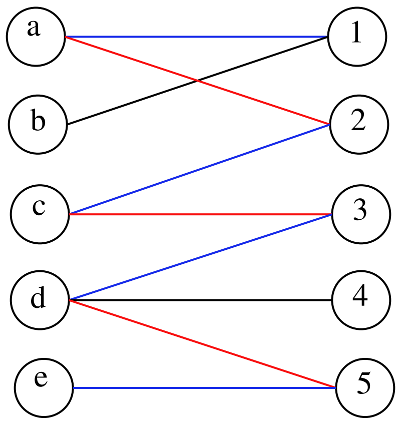


Figure 4.3: Alternating Path for the matching M

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

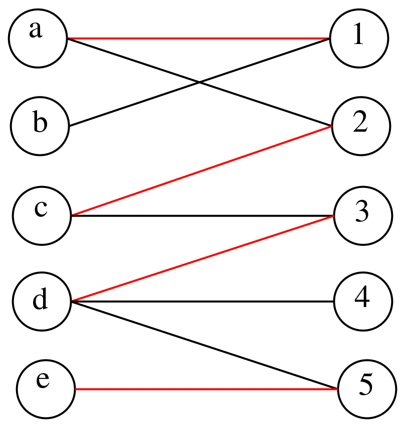


Figure 4.4: New Matching M_1 with more edges than M

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

```python
def improveLabels(val):
    """ change the labels, and maintain minSlack.
    """
    for u in S:
        lu[u] -= val
    for v in V:
        if v in T:
            lv[v] += val
        else:
            minSlack[v][0] -= val

def improveMatching(v):
    """ apply the alternating path from v to the root in the tree.
    """
    u = T[v]
    if u in Mu:
        improveMatching(Mu[u])
    Mu[u] = v
    Mv[v] = u
```

```

def slack(u,v):
    return lu[u]+lv[v]-w[u][v]

def augment():
    """ augment the matching, possibly improving the lablels on the way.
    """
    while True:
        # select edge (u,v) with u in S, v not in T and min slack
        ((val, u), v) = min([(minSlack[v], v) for v in V if v not in T])
        assert u in S
        if val>0:
            improveLabels(val)
        # now we are sure that (u,v) is saturated
        assert slack(u,v)==0
        T[v] = u
        if v in Mv:
            u1 = Mv[v]
            assert not u1 in S
            S[u1] = True
            for v in V:
                if not v in T and minSlack[v][0] > slack(u1,v):
                    minSlack[v] = [slack(u1,v), u1]
        else:
            improveMatching(v)
            return

def maxWeightMatching(weights):
    """ given w, the weight matrix of a complete bipartite graph,
        returns the mappings Mu : U->V ,Mv : V->U encoding the matching
        as well as the value of it.
    """
    global U,V,S,T,Mu,Mv,lu,lv, minSlack, w
    w  = weights
    n  = len(w)
    U  = V = range(n)
    lu = [ max([w[u][v] for v in V]) for u in U]
    lv = [ 0  for v in V]
    Mu = {}
    Mv = {}
    while len(Mu)<n:
        free = [u for u in V if u not in Mu]
        u0 = free[0]
        S = {u0: True}
        T = {}
        minSlack = [[slack(u0,v), u0] for v in V]
        augment()
    val = sum(lu)+sum(lv)
    return (Mu, Mv, val)


if __name__ == '__main__':
    import timeit
    import numpy as np
    content = np.loadtxt('file1.txt').tolist()
    # print(len(content))
    start = timeit.default_timer()
    maxWeightMatching(content)

    stop = timeit.default_timer()

    print('Time: ', stop - start)
```

5 Emperical 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.

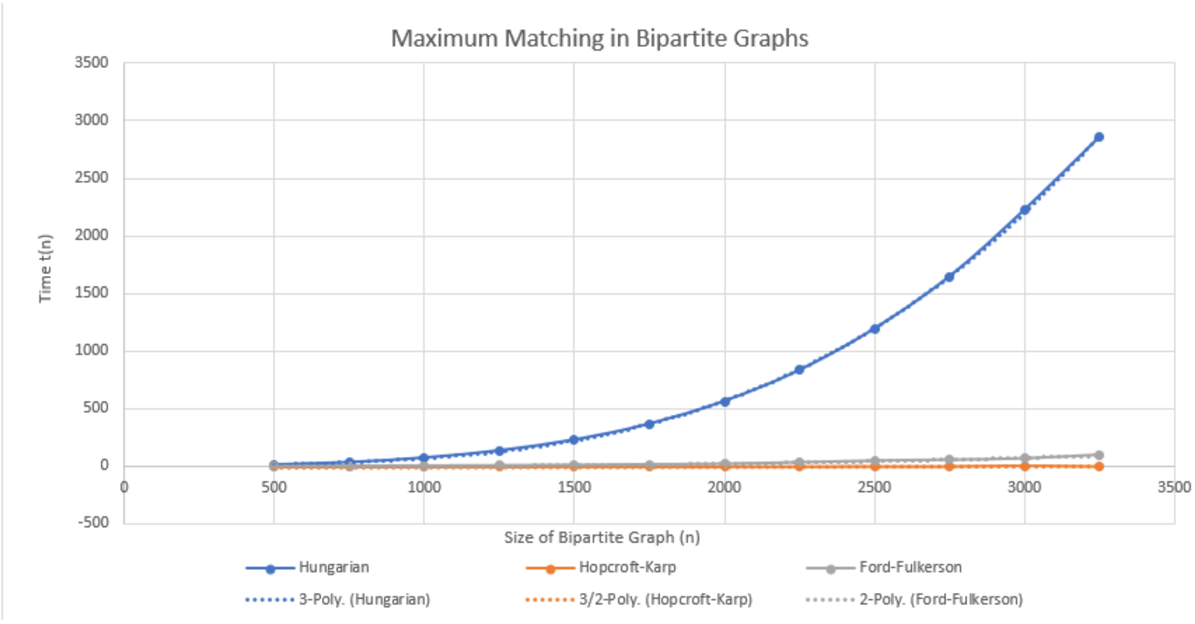


Figure 5.1: Empirical Analysis of Algorithms for Finding Maximum Matching in a Bipartite Graph

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 Hopecroft-Karp Algorithm gives us the maximum matching in a bipartite graph in the minimum time out of these algorithms.

3 Python Code

```python

class Edge(object):
  def __init__(self, u, v, w):
    self.source = u
    self.target = v
    self.capacity = w

  def __repr__(self):
    return "%s->%s:%s" % (self.source, self.target, self.capacity)


class FlowNetwork(object):
  def  __init__(self):
    self.adj = {}
    self.flow = {}

  def AddVertex(self, vertex):
    self.adj[vertex] = []

  def GetEdges(self, v):
    return self.adj[v]

  def AddEdge(self, u, v, w = 0):
    if u == v:
      raise ValueError("u == v")
    if u not in self.adj.keys():
        self.adj[u] = []
    if v not in self.adj.keys():
        self.adj[v] = []
    edge = Edge(u, v, w)
    redge = Edge(v, u, 0)
    edge.redge = redge
    redge.redge = edge
    self.adj[u].append(edge)
    self.adj[v].append(redge)
    # Intialize all flows to zero
    self.flow[edge] = 0
    self.flow[redge] = 0

  def FindPath(self, source, target, path):
    if source == target:
      return path
    for edge in self.GetEdges(source):
      residual = edge.capacity - self.flow[edge]
      if residual > 0 and not (edge, residual) in path:
        result = self.FindPath(edge.target, target, path + [(edge, residual
)])
        if result != None:
          return result

  def MaxFlow(self, source, target):
    path = self.FindPath(source, target, [])
    # print('path after enter MaxFlow: %s' % path)
    # for key in self.flow:
```

```
  #  print('%s:%s' % (key,self.flow[key]))
  # print('-' * 20)
  while path != None:
   flow = min(res for edge, res in path)
   for edge, res in path:
    self.flow[edge] += flow
    self.flow[edge.redge] -= flow
  #  for key in self.flow:
  #   print('%s:%s' % (key,self.flow[key]))
   path = self.FindPath(source, target, [])
  #  print('path inside of while loop: %s' % path)
  # for key in self.flow:
  #  print('%s:%s' % (key,self.flow[key]))
  return sum(self.flow[edge] for edge in self.GetEdges(source))

def bipartiteMatch(graph):
  '''Find maximum cardinality matching of a bipartite graph (U,V,E).
  The input format is a dictionary mapping members of U to a list
  of their neighbors in V.  The output is a triple (M,A,B) where M is a
  dictionary mapping members of V to their matches in U, A is the part
  of the maximum independent set in U, and B is the part of the MIS in V.
  The same object may occur in both U and V, and is treated as two
  distinct vertices if this happens.'''

  # initialize greedy matching (redundant, but faster than full search)
  matching = {}
  for u in range(len(graph)):
    for v in range(len(graph[u])):
      if graph[u][v] != 0 and v not in matching:
        matching[v] = u
        break

  while 1:
    # structure residual graph into layers
    # pred[u] gives the neighbor in the previous layer for u in U
    # preds[v] gives a list of neighbors in the previous layer for v in
  V
    # unmatched gives a list of unmatched vertices in final layer of V,
    # and is also used as a flag value for pred[u] when u is in the
 first layer
    preds = {}
    unmatched = []
    pred = dict([(u,unmatched) for u in range(len(graph))])
    for v in matching:
      del pred[matching[v]]
    layer = list(pred)

    # repeatedly extend layering structure by another pair of layers
    while layer and not unmatched:
      newLayer = {}
      for u in layer:
        for v in range(len(graph[u])):
          if graph[u][v] != 0 and v not in preds:
            newLayer.setdefault(v,[]).append(u)
      layer = []
      for v in newLayer:
```

```
                preds[v] = newLayer[v]
                if v in matching:
                    layer.append(matching[v])
                    pred[matching[v]] = v
                else:
                    unmatched.append(v)

        # did we finish layering without finding any alternating paths?
        if not unmatched:
            unlayered = {}
            for u in range(len(graph)):
                for v in range(len(graph[u])):
                    if graph[u][v] != 0 and v not in preds:
                        unlayered[v] = None
            return (matching,list(pred),list(unlayered))

        # recursively search backward through layers to find alternating paths
        # recursion returns true if found path, false otherwise
        def recurse(v):
            if v in preds:
                L = preds[v]
                del preds[v]
                for u in L:
                    if u in pred:
                        pu = pred[u]
                        del pred[u]
                        if pu is unmatched or recurse(pu):
                            matching[v] = u
                            return 1
            return 0

        for v in unmatched: recurse(v)

import sys

if __name__ == '__main__':
    import timeit
    import numpy as np
    from operator import itemgetter
    from  networkx.algorithms import bipartite as nx
    import networkx.convert as nc
    from hungarian_algorithm import algorithm
    from networkx.algorithms import flow as fl
    lst  = []
    for k in range(500,10000,500):
        a = int(np.random.uniform(1,k))
        lst.append((a,k-a))
    for i,j in lst:
        G = nx.random_graph(i,j, 1)
        partition_l =   set(map(itemgetter(0),G.edges()))
        content = nx.biadjacency_matrix(G,partition_l).toarray().tolist()
        # print(len(content))

        start = timeit.default_timer()
        bipartiteMatch(content)
        stop = timeit.default_timer()

        print('Hopcroft-Karp Time: n: ',i+j, stop - start)

        G = nx.random_graph((i+j)//2,(i+j)//2,1)
        start = timeit.default_timer()
        algorithm.find_matching(G,matching_type = 'max', return_type = '
    list')

        stop = timeit.default_timer()

        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.