Introduction:

The maximum bipartite matching problem is a problem in graph theory that involves finding the largest possible matching in a bipartite graph. In a bipartite graph, a matching is a set of edges where no two edges share a common vertex, and no two edges are incident on the same side of the bipartition.

The maximum bipartite matching problem involves finding a matching in a bipartite graph that contains the largest number of edges. This problem is NP-hard, which means that there most likely is not an efficient algorithm that can solve it in all cases. However, there are algorithms that can solve the problem in many cases, and these algorithms have applications in a variety of fields, including network design, scheduling, and resource allocation.

A Bipartite graph G is a graph whose vertices can be divided into two disjoint and independent sets L (left) and R (right), that is every edge connects a vertex in L to one in R.

The two sets L and R may be thought of as a coloring of the graph with two colors. (See Figure 1)

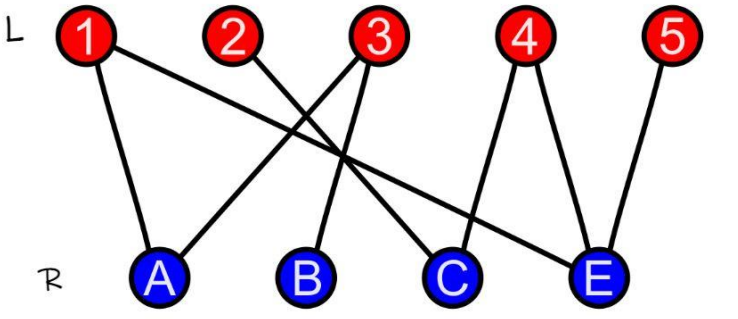


Figure (1)

The algorithm was first proposed by Hungarian mathematician Dénes Kőnig in 1931. In his famous paper, "Theory of Finite and Infinite Graphs," Kőnig showed that the maximum number of matching edges in a bipartite graph is equal to the minimum number of vertices that must be removed to disconnect the graph. This result is known as Kőnig's theorem. (Kőnig)

Problem Statement:

Input: A unweighted, undirected graph G = $< L \cup R, E >$

Output: The set of M tuples (set of edges that make matches)

M = { (L_1, R_1), (L_2, R_2) … (L_n, R_n) }

n = # of edges

Applications:

One common application of the maximum bipartite matching algorithm is in the assignment of jobs to workers or tasks to machines. In this case, the workers or machines are represented by the vertices in one set, and the jobs or tasks are represented by the vertices in the other set. The maximum bipartite matching algorithm can be used to find the maximum number of jobs that can be assigned to workers or the maximum number of tasks that can be assigned to machines.

Another application of the maximum bipartite matching algorithm is in the study of network flows. In this case, the vertices in one set represent the source nodes of the network, and the vertices in the other set represent the destination nodes. The maximum bipartite matching

algorithm can be used to find the maximum flow that can be routed from the source nodes to the destination nodes.

Overall, the maximum bipartite matching algorithm is a useful tool for solving a wide range of optimization problems involving the assignment of elements from two sets to each other.

Recent Work:

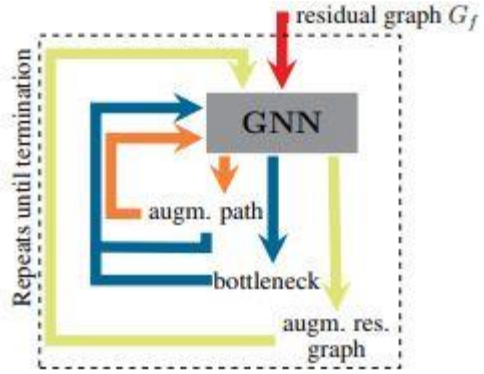
Edmunds and Karp introduced a common approach to solving the Maximum Bipartite Matching problem with weighted edges. They proposed to reformulate the problem as a minimum-cost flow problem. This involves constructing a special network where each edge in the original graph corresponds to a pair of directed edges in the network, one in each direction. The weights of the edges in the original graph are used to assign capacities and costs to the edges in the network. The minimum-cost flow problem is then solved on this network to find the maximum matching in the original graph. (Edmunds)

Fredman and Tarjan made significant contributions to the field of maximum bipartite matching with their paper "Fibonacci Heaps and Their Uses in Improved Network Optimization Algorithms," which was published in the Journal of the ACM in 1987. Together the two proposed a new approach to the problem that used Fibonacci Heaps to speed up the computation of shortest paths in the underlying network. This approach allowed them to develop an algorithm for maximum bipartite matching with a running time of O (n (m + n log n)). This was a significant improvement over previous algorithms, which had running times of O(n^3) for dense graphs and O (nm log n) for sparse graphs. The use of Fibonacci Heaps in maximum bipartite

matching has since become a standard approach and has been widely used in a variety of applications. (Fredman)

In 2004 , M. Wattenhofer and R. Wattenhofer, proposed a new approach to the Maximum Bipartite Matching algorithm that is both fast and simple. Their algorithm is based on a novel use of minimum-cost flow techniques, which allows it to solve the matching problem more efficiently than previous algorithms. In their paper, they show that the maximum bipartite matching problem can be restated as a minimum-cost flow problem, which can then be solved using known algorithms for minimum-cost flow. This approach has several advantages over previous algorithms for maximum bipartite matching. First, it allows the problem to be solved more efficiently, with a running time of O(n^2 log n) for dense graphs and O(n log^2 n) for sparse graphs where n is the number of nodes in the graph. This is a significant improvement over previous algorithms, which had running times of O(n^3) and O(nm log n), respectively. Second, it is relatively simple to implement, as it relies on existing algorithms for minimum-cost flow rather than requiring new, specialized techniques. (Wattenhofer)

In 2020, Dobrik Georgiev and Pietro Lio did some research on faster methods of implementing Maximum Bipartite Matching. They used a single neural network to test the speed and accuracy at which a neural network can solve for the best route of a weighted bipartite graph. The goal was to give the computer a system and have it return the best solution, in regard to edge weight, for the system. By setting the flow of the best solution to one to test the computer and training it with different solutions for other bipartite



Georgiev, 2020

matching, they found that it was 90-95% effective at returning the most accurate answers. The system itself was designed to reduce failure where bad bottlenecks in a system are found. These bottlenecks will slow systems or even allow them to fail when improperly handled. This system seeks to allow neural networks to correct itself to work through information at an optimal speed for longer by avoiding those bottlenecks. (Georgiev)

Algorithmic Solution:

```
1.  Func maxBipartiteMatch(G)
2.      Add source and sink node (with directed edges)
3.      Create G' = <V', E'> as a flow network.
4.      Copy G into G' (residual network)
5.      Initialize flow f to 0
6.      M = set()
7.      While (augmenting path p exists in G'):
8.          augment flow f along p
9.      Drop source and sink from the network
10.     For (u,v) in E'
11.         If (f[u][v] > 0)
12.             M.add((u,v))
13.     Return M
```

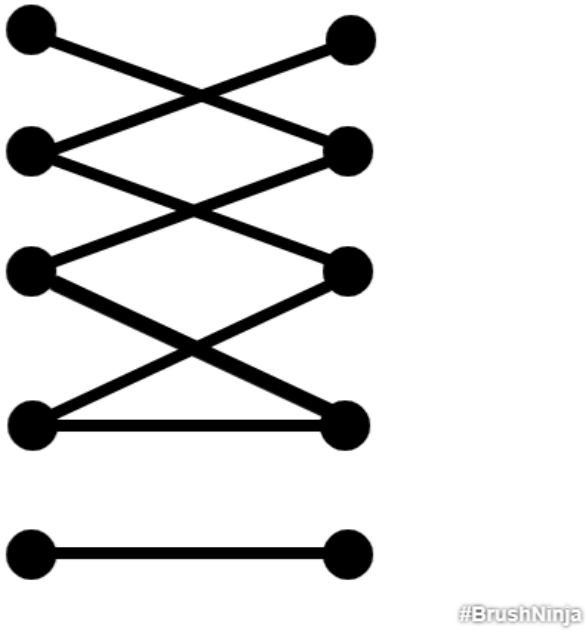
1. Add a source and sink node to the graph **G**, with directed edges connecting the source to every vertex in the left partition and every vertex in the right partition to the sink. This creates a new graph **G'** with an additional layer of nodes
2. Copy the graph **G** into **G'** as the residual network, which will be used to store the flow of the algorithm.
3. Initialize the flow **f** to 0. This will be used to keep track of the total flow in the network.
4. Initialize the matching **M** to the empty set. This will be used to store the final matching in the graph.
5. While there exists an augmenting path **p** in the residual network **G'**: ->
6. -> Augment the flow **f** along the path **p** by adding the minimum capacity of the edges in **p** to **f**. This increases the total flow in the network by the amount of flow that can be sent along **p**.
7. Drop the source and sink nodes from the network, since they are no longer needed. This leaves us with the original graph **G**, with the flow values stored in the residual network **G'**.
8. Iterate through the residual network **G'** and construct the final matching **M** as follows:
9. -> For each edge **(u, v)** in **E'**:
10. ---> If **u** is an element of the left partition and **v** is an element of the right partition, and the flow **f (u, v)** along that edge is greater than 0, add the edge **(u, v)** to the matching **M**.
11. Return the final matching **M**.

This implementation of the maximum bipartite matching algorithm has a runtime complexity of **O (|V| * |E|)**, where **|V|** is the number of vertices in the graph and **|E|** is the number of edges, because it involves repeatedly traversing all the edges in the graph to find augmenting paths.

The runtime of the maximum bipartite matching algorithm depends on the specific implementation, but in general it has a runtime complexity of **O (|V| * |E|)**, where **|V|** is the number of vertices (or nodes) in the graph and **|E|** is the number of edges. This is because the algorithm repeatedly searches for augmenting paths, which involves traversing all the edges in the graph.

The runtime can be improved by using more efficient data structures and search algorithms, such as a priority queue to store and search for the augmenting paths. In this case, the runtime complexity becomes **O (|V| + |E|) * log |V|)**, which is generally faster for large graphs.

It's also worth noting that the maximum bipartite matching algorithm is guaranteed to terminate in finite time, because each time an augmenting path is found and added to the matching, the size of the matching increases by at least one. Since the size of the matching can never exceed the number of vertices in the graph, the algorithm will always terminate after a finite number of steps.

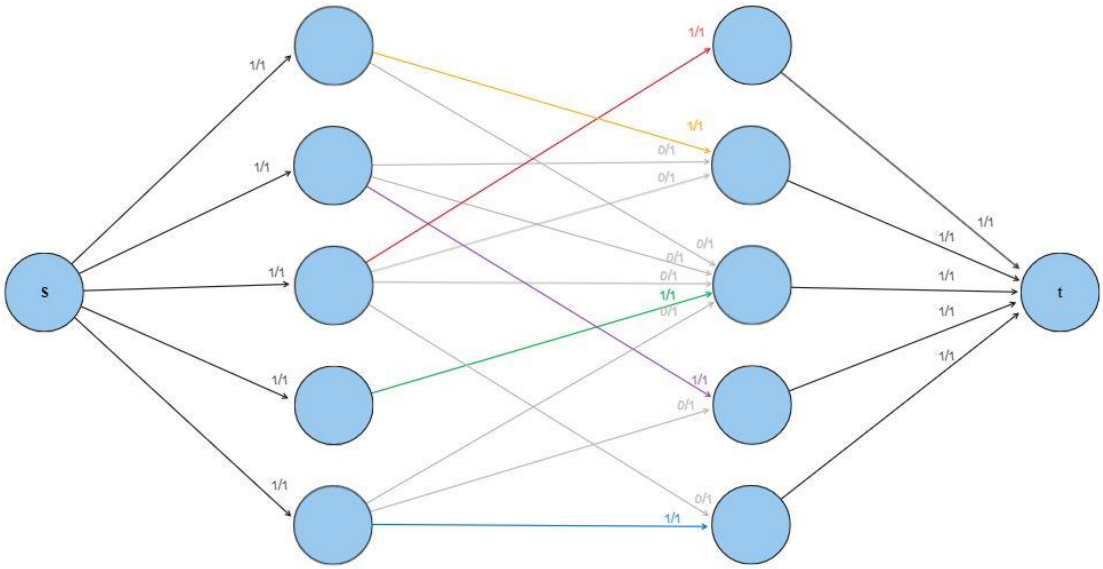


Algorithm Discussion:

The algorithm only applies to bipartite graphs. This means that it cannot be used on graphs that are not bipartite, such as complete graphs or trees, but as we discussed early in this writeup, there are multiple different applications for this algorithm and a definite need to increase its speed and accuracy.

The algorithm only finds a maximum matching, not necessarily a maximum weight matching. In other words, it may not always find the assignment of elements that maximize some objective function, such as the total weight of the matching edges. However, this algorithm can be modified to work with a weighted bipartite graph as well.

The algorithm has a worst-case time complexity of $O(n^3)$, where n is the number of vertices in the bipartite graph. This means that it may be slow for large graphs and may not be suitable for real-time applications or for solving very large problems without a heuristic that could increase speed but reduce accuracy.



Works Cited

Cormen, Thomas H., et al. *Introduction to Algorithms*. Mit Press, 2009.

Edmonds, J., & Karp, R. M. (1972). Theoretical improvements in algorithmic efficiency for network flow problems. Journal of the ACM (JACM), 19(2), 248-264.

Fredman, M. L., & Tarjan, R. E. (1987). Fibonacci heaps and their uses in improved network optimization algorithms. Journal of the ACM (JACM), 34(3), 596-615.

Georgiev, Dobrik, and Pietro Lio. "Neural Bipartite Matching." ArXiv, online, 2 June 2020, https://doi.org/10.48550/ARXIV.2005.11304.

Kőnig, Dénes. "Theory of Finite and Infinite Graphs." American Journal of Mathematics, vol. 53, no. 1, 1931, pp. 46-64. JSTOR, www.jstor.org/stable/2371215.

Wattenhofer, Mirjam, and Roger Wattenhofer. "Fast and simple algorithms for weighted perfect matching." Electronic Notes in Discrete Mathematics 17 (2004): 285-291.