Chapter 1

Bipartite Matching

Contributed by Brian Page

1.1 Introduction

Graph matching seeks to determine a set of edges within the graph such that there are no vertices in common among the edges selected [6]. As its name implies, bipartite matching is a matching performed on a bipartite graph [2] in which the vertices of said graph can be divided into two disjoint sets.

Bipartite matching has many real world applications, many of which resemble some form of assignment or grouping [1]. One such example would be that of job positions vs job applicants. Each applicant has a subset of jobs they have applied for, yet each position can filled by at most one applicant. A matching of this graph would be performed in an attempt to find the maximum number of applicants that can be placed into the job openings. This of course is but one example of bipartite matching.

Bipartite matching while useful in its own right, is often used as an intermediate algorithm to prepare data for subsequent computation. Because of this, efficient computation of bipartite matching has become an interesting topic among High Performance Computing (HPC) researchers as scalability and performance continue to increase in importance.

1.2 The Problem as a Graph

Before we can dive into bipartite matching, we must first understand the different types of graph matching [6]. Considering a general graph $G = (V, E)$ where V is the set of vertices and E the set of all edges a vertex is considered to have been *matched* when an edge has the vertex as one its endpoints.

An notional attempt at matching can quickly generate a *Maximal matching*. A *Maximal matching* is a matching M where the addition of any edge in the bipartite graph G would make M no longer a valid bipartite matching. This occurs when an edge is added, that has had at least one it its vertices matched previously. Fig. 1.1 illustrates three example graphs, and their corresponding maximal matching. Please note that the graphs in Fig. 1.1 are not necessarily bipartite graphs and present maximal matching for generalized graphs.

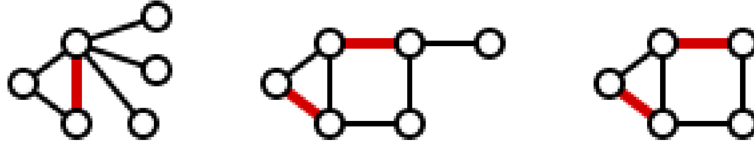

Figure 1.1: Maximal Matching
Here we see three general graphs $G(V, E)$ as well as one possible maximal matching for each. Black lines represent edges, while bold red lines indicate an edges selected as the maximal matching for given graph. The red edges comprise the maximal matchings for each graph since to add any other edge would require adding an edge with a vertex already matched to edge in the matching.

Subsequently the *Maximum Matching*, aslo known as Maximum Cardinality Matching (MCM), of a bipartite graph is a matching consisting of the largest possible independent edge set or total edge weight. It is important to illustrate that all *maximum matchings* are *maximal matching*, however since *maximum matchings* contain the largest cardinality edge set for a matching on the graph G, not all maximal matchings are the maximum for G. This is an important distinction as the calculation of a valid *maximum matching* can and often is much more difficult to obtain.

For comparison Fig. 1.2 illustrates the maximum matching for the same graphs seen in Fig. 1.1. Fig. 1.2c has at least two possible solutions for its maximal matching. Matchings, both maximal and maximum, can have multiple solutions depending on a graph's structure. This does not mean that unique maximum matchings are not obtainable, instead it indicates that either a graphs structure must be such that this situation exists, or an additional constraint must be present to limit edge selection.

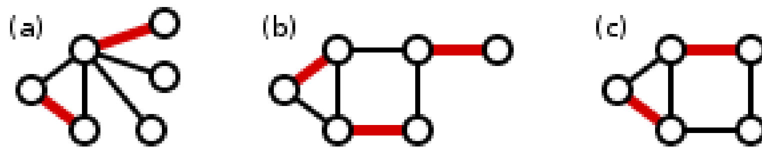

Figure 1.2: Maximum Matching
Here we see three general graphs $G(V, E)$ as well as one possible maximum matching for each. Black lines represent edges, while bold red lines indicate an edges in the maximum matching for given graph. Note that for (a) 3 possible matchings exists which are maximal, (b) has only one maximum matching, while (c) can have 2 maximum matchings.

One important thing to observe is that each *maximum matching* is a *maximal matching*, however not all *maximal matching* are the *maximum matching* for a particular graph. This can be seen in Fig. 1.2 where for graphs (a) and (b) the maximum matching is different from the maximal matching shown in Fig. 1.1. Additionally we see that for graph (c) the previous maximal matching is in fact the maximum matching.

There are other forms of matching that can be discussed, however the most widely used is that of determining the *maximum matching* of a graph and is the focus of this topic.

One of the most common methods for solving bipartite matching is to treat the graph $G = (V, E) = ((u, v), E)$ as a flow network, as seen in Fig. 1.3, in which a connection or edge between vertices u and v may or may not be selected in the final matching. The Ford-Fulkerson algorithm determines the maximum flow through just such a graph/network and in the case of bipartite

matching, is used to determine the maximum matching on G. Ford-Fulkerson [4] works by adding and removing edges while checking the matching with the changed edge state (included or excluded) until it has determined the optimal edge set or *matching*.

There are of course other methods such as Hopcroft-Karp which performs a localized randomization of edge inclusion/exclusion, as well as the well known Bellman-Ford algorithm. This method achieves an improved time complexity of $\mathcal{O}(\|E\| log \|V\|)$ in the average case thanks to the high probability that all non-optimal matching have augmenting paths [5, 9].

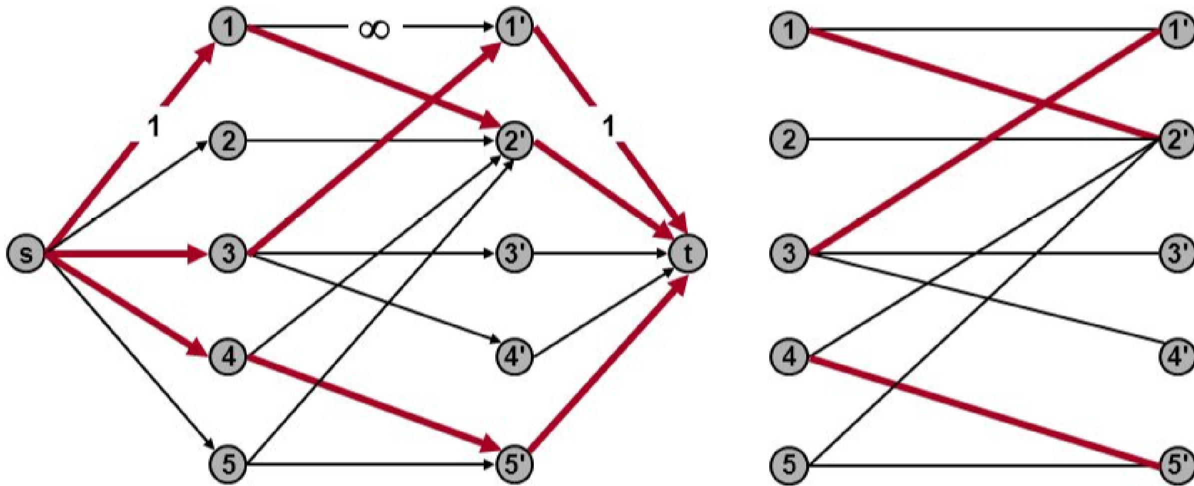

Figure 1.3: Graph conversion to a flow network for the purpose of determining its maximum matching.

1.3 Some Realistic Data Sets

A bipartite graph can and often is represented as sparse matrix, therefore there are many sources of bipartite graphs in existence today. The Suite Sparse Matrix Collection [8] contains many real world data sets for different research areas such as fluid dynamics and circuit problems. The matrices have varying characteristics such as row count and total non-zeros ranging from as few as 20 rows with 90 non-zeros to millions of rows with hundreds of millions of non-zeros.

Another source for real world data is the Stanford Large Network Dataset Collection (SNAP) [7] which hosts many large graphs for social media and web based networks.

Lastly there are many tools which are used for the generation of synthetic bipartite graphs. Using such tools, a researcher can create graphs in which they have control over structural characteristics.

1.4 Bipartite Matching-A Key Graph Kernel

We will discuss a generalized version of the augmenting path algorithm which lies at the heart of many maximum flow algorithms often used for bipartite matching [4, 5, 9]. Augmenting paths ensures the determination of a solution, however in the case of Ford-Fulkerson does not always provide the optimal solution [3].

Algorithm 1 provides a brief implementation for augmenting paths within a flow network. Starting with an bipartite graph G and the current matching M we evaluate edges not currently apart

Algorithm 1 Augmenting Path
P is a path from v to u
α,β are edges (u,v)
A is a augmenting path being evaluated

```
1: procedure GRAPH G((u, v),E), MATCHING M
2:     for i in M do
3:         if M_i(u)−>G(v) then //if another path exists to u
4:             α = M_i // save current u −> v path
5:             for j in E where E(M_i(u), j) do // find any other paths from v to M_i(u)
6:                 β[] = E(M_i(u), j) // find the new paths
7:                 for k in β do // for all edges discovered
8:                     A = β[k]
9:                     if (M - β[k]+!A) > M then // if swapping paths increases flow
10:                        M[i] =!A // save augmenting path changes
11:                end for
12:            end for
13:    end for
```

of the M in order to see if accommodating this new edge and making any required path adjustments will increase M. To do this, we evaluate an edge associated with $M_i(u)$ and find the $M_i(v)$ associated with the edge. From here we need find alternative edges (paths) from v to u such that the rules for bipartite matching are satisfied. This means that we trace back from u back to v, and then forward through an alternative path back to u. If such a set edges exist and adding it will increase M, we invert the edges contained in this "discovery" period are flipped. This means that edges that were contained in M are removed, and edges not previously in M are added. This process is repeated until no more edges can be added which will increase M, leaving the maximum M of G.

Hopcroft-Karp and Ford-Fulkerson both implement augmenting paths to perform a similar portion of their flow discovery. Hopcroft-Karp fior example is based on the push and relabel method for finding maximum flow in which a bipartite graph is given as input. The algorithm then uses breadth first search (BFS) in order to partition the vertices into two sets *matched* and *unmatched*. Edges are then swapped in and out of the matching, with the resulting matching M evaluated against the highest matching achieved thus far.

One key difference between hopcroftt-karp and ford-fulkerson is its use of localized path augmentations where an edge incident to the current vertex can be included or excluded without determining the entire path across G. The results in an overall time complexity of $\mathcal{O}(\|E\|\sqrt{\|V\|})$. Overall, the exact time complexity of determining the maximum matching on a bipartite graph depends on the precise implementation chosen, however The complexity of the Hopcroft-Karp algorithm exhibits good time complexity in the general case [5, 9].

1.5 A Sequential Algorithm

Discuss here the outlines of a sequential algorithm. What programming paradigms might make the most sense? What are the key data structures? Does the computational complexity differ from that in the Section 1.4?

The initial sequential algorithm we will discuss utilizes the Hopcroft-Karp algorithm. Assuming

that an arbitrary bipartite graph has been loaded and stored in the appropriate manner, computation performs Hopcroft-Karp which makes a call to breadth first search. The resulting frontier is used to investigate the initial potential mapping, which is selected based on weight of "flow" of the edges. Once this initial mapping is determined, a check for augmenting paths is performed and the matching process is performed continuously until no augmenting paths exist. The resulting matching is the maximum cardinality matching for the graph used.

1.6 A Reference Sequential Implementation

One possible implementation of maximum cardinality (MCM) bipartite matching utilizes the generation of a flow network and run a common maximal flow algorithm. There are of course much more complex methods to determine the MCM of a graph however flow networks are the traditional and easy to comprehend method of doing go. Because of this, the following sequential implementation is based on the generation of a flow network from our input bipartite graph, and then running the a Hopcroft-Karp maximum flow solver.

We have developed our sequential implementation using C++. Solver loads the bipartite graph from file and populates vertex and edge vectors. The matrices we have used represent graphs in the form of sparse adjacency matrices. This means that each non-zero element within the matrix is an edge between the vertices represented by the non-zero's row and column ids.

Once the matrix is read and stored as a bipartite graph, we begin traversing the graph from an initial vertex. In the case of this implementation the initial vertex is chosen to b vertex 0 in the set of vertices u or v with the largest cardinality. The reason for this is that the set of vertices with the least cardinality is often responsible for limiting the MCM within a bipartite graph. This occurs because the total number of possible matches is only as large as the smallest vertex set within the graph.

The sequential implementation evaluated uses an instance of the Hopcroft-Karp algorithm to evaluate matchings and perform augmenting path updates until an MCM is found for the graph. Fig. 1.4 includes the C++ code which performs the push and relabel operations for vertices and edges as augmenting paths are found and the current matching is updated. It traverses the graph, finding unmatched vertices by performing breadth first searches, then as mentioned evaluated augmenting paths and selects those with the highest weight or flow. Once there are no additional augmenting paths, the resulting flow is returned and represents the maximum matching for the inputed graph.

1.7 Sequential Scaling Results

The bipartite graphs that I have tested the reference implementation with have come from the Suite Sparse Matrix Collection [8]. Fig. 1.5 shows the time required to perform graph construction as well as the maximum matching on the constructed graph, for each graph evaluated. As can be seen, total time required, as well as the time required for a particular application phase, increases as vertex count increases. However the *12month1* experiences the largest time requirement even though it does not have the largest number of vertices.

This is because total the edges of a graph, and not just its vertices, govern how that graph is traversed. Fig. 1.6 shows the relation between total vertex count and edge count within each graph. When comparing these two figures, it is clearly visible that the graph with the greatest time requirement has an edge count that is much greater than that of other graphs tested.

Figure 1.4: Sequential Implementation: Hopcroft-Karp Subroutine

```
// Returns size of maximum matching
int biGraph::hopcroftKarp() {
    // pairU[u] stores pair of u in matching where u
    // is a vertex on left side of Bipartite Graph.
    // If u doesn't have any pair, then pairU[u] is NIL
    pairU = new int[m+1];

    // pairV[v] stores pair of v in matching. If v
    // doesn't have any pair, then pairU[v] is NIL
    pairV = new int[n+1];

    // dist[u] stores distance of left side vertices
    // dist[u] is one more than dist[u'] if u is next
    // to u'in augmenting path
    dist = new int[m+1];

    // Initialize NIL as pair of all vertices
    for (int u=0; u<m; u++) {
        pairU[u] = NIL;
    }
    for (int v=0; v<n; v++) {
        pairV[v] = NIL;
    }

    // Initialize result
    int result = 0;

    // Keep updating the result while there is an
    // augmenting path.
    while (bfs()) {
        // Find a free vertex
        for (int u=1; u<=m; u++){
            // If current vertex is free and there is
            // an augmenting path from current vertex
            if (pairU[u]==NIL && dfs(u)) {
                result++;
            }
        }
    }
    return result;
}
```

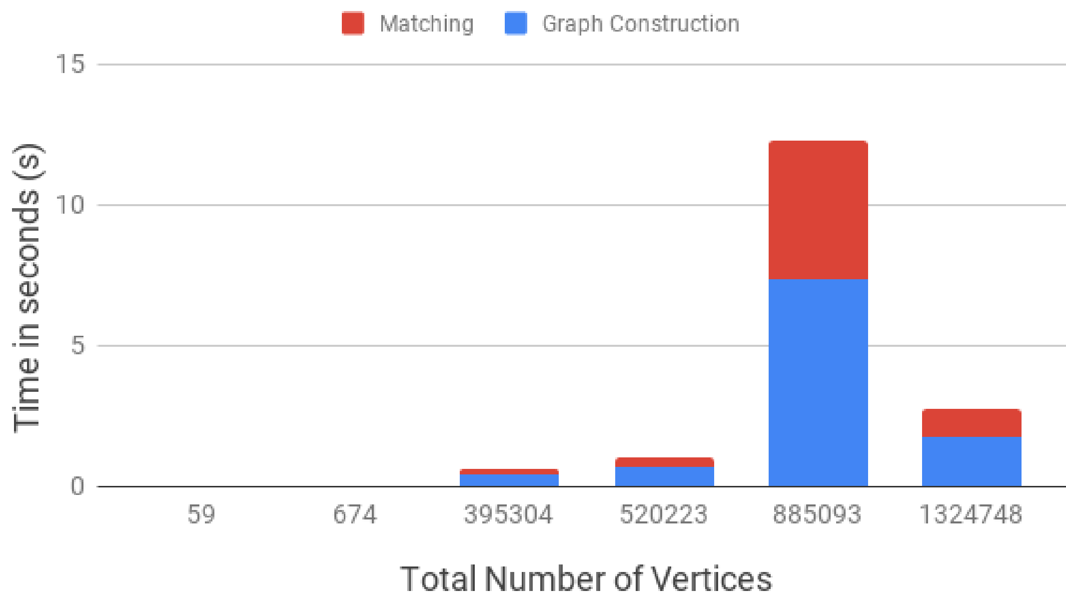

Figure 1.5: This chart shows the graph construction (blue) and maximum cardinality matching (red) time for 7 bipartite graphs. The graphs were taken from the Suite Sparse Matrix Collection and very in vertex and edge counts.

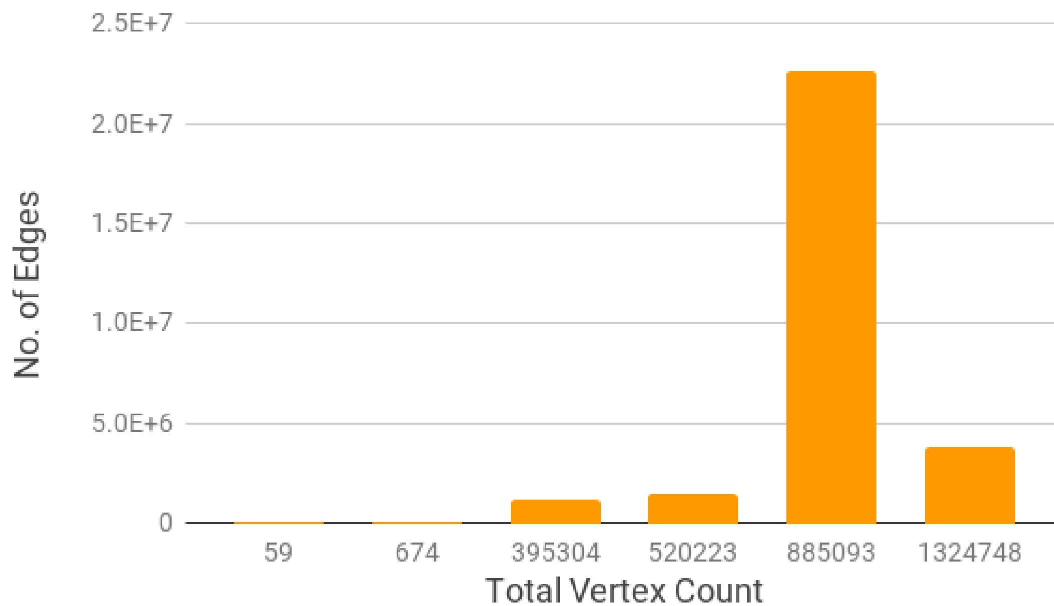

Figure 1.6: This chart shows the total number of edges the for 7 bipartite graphs evaluated.

Edge count within a bipartite graph can have a drastic affect on MCM determination, as a greater number of edges can generate a much greater number of augmenting paths. Additional augmenting paths require additional computation, thereby increasing the overall runtime.

1.8 Response to Reviews

In my initial version of this paper I had made certain assumptions about the level of comprehension for this material. Therefore I needed to go through and provide some additional explanation for particular fundamental concepts used in bipartite matching. Furthermore, some of my material while not necessarily in accurate, failed to convey the concept in a generalized manner. As such the implementation psuedocode which originally gave an extremely high level view of the Hopcorft-Karp algorithm, was replaced with that of Augmenting path on a bipartite graph.

I did my best to maintain a generalize approach to bipartite matching while introducing some of the most common methods used. This meant that some complexity information was omitted since the exact time complexity is highly dependant on the implementation chosen as well as individual graph properties.

Bibliography

[1] Assignment problem. https://en.wikipedia.org/wiki/Assignment_Problem.

[2] Bipartite graph. https://en.wikipedia.org/wiki/Bipartite_graph.

[3] Flow networks. https://en.wikipedia.org/wiki/Flow$_n$*etworkAugmenting_paths*.

[4] Ford-fulkerson algorithm. https://www.geeksforgeeks.org/ford-fulkerson-algorithm-for-maximum-flow-problem/.

[5] Hopcroft-karp algorithm. https://www.geeksforgeeks.org/hopcroft-karp-algorithm-for-maximum-matching-set-1-introduction/.

[6] Matching. https://en.wikipedia.org/wiki/Matching_(graph$_t$*heory*).

[7] Stanford large network dataset collection (snap). https://snap.stanford.edu/data/.

[8] Suite spare matrix collection. https://sparse.tamu.edu.

[9] John E. Hopcroft and Richard M. Karp. An $n^{5/2}$ algorithm for maximum matchings in bipartite graphs. *SIAM Journal on Computing*, 2(4), 1973.