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.

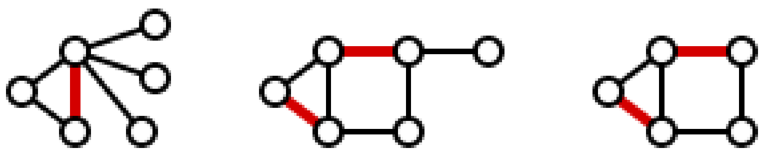


Figure 1.1: Maximal Matching
Here we see three general graphs $G(V, E)$ as well as one possible maximal matching for each. Black lines represent edges, while bold red lines indicate an edges selected as the maximal matching for given graph. The red edges comprise the maximal matchings for each graph 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 a maximal matching for the same graphs seen in Fig. 1.1. Fig. 1.1c 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.

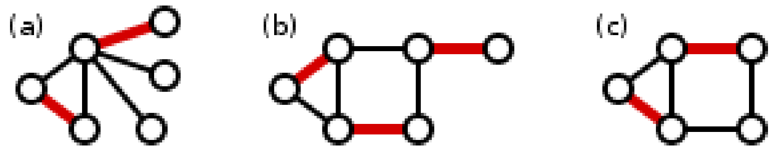


Figure 1.2: Maximum Matching
Here we see three general graphs $G(V, E)$ as well as one possible maximum matching for each. Black lines represent edges, while bold red lines indicate an edges in the maximum matching for given graph. Note that for (a) 3 possible matchings exists which are maximal, (b)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

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].

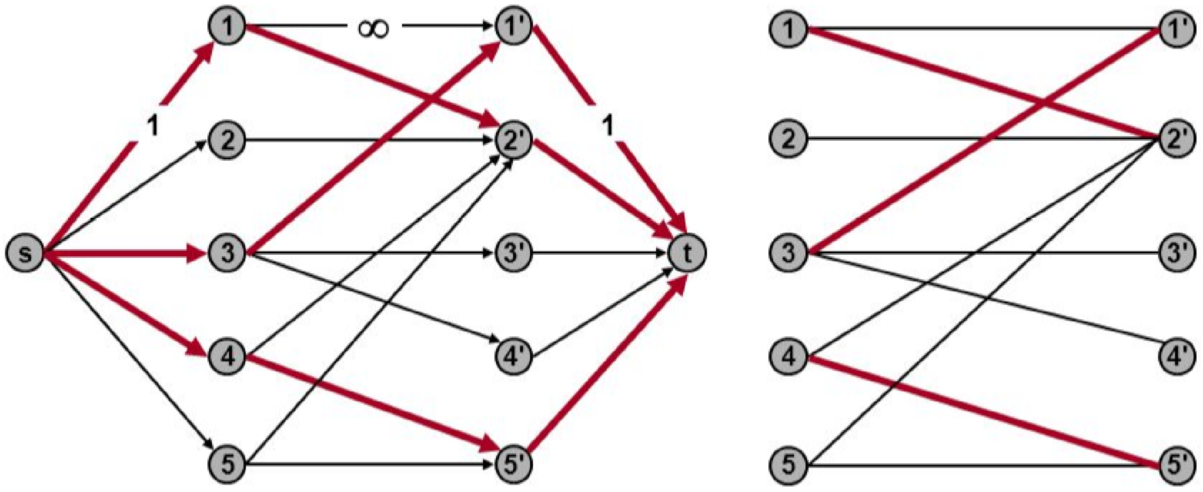


Figure 1.3: Graph conversion to a flow network for the purpose of determining its maximum matching.

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.

Table 1.1: BENCHMARK SUITE

matrix	**u**	**v**	**edges**
divorce	50	9	225
Cities	50	46	1342
World Cities	315	100	7518
Notredame_actors	392400	127823	1470404
12month1	12471	872622	22624727

While Suite Sparse and SNAP contain many graphs that are not bipartite, it is possible to search specifically for bipartite graphs. Table 1.1 lists several bipartite matrices taken from the Suite Sparse Matrix Collection which were utilized in our evaluation.

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 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:             $\alpha = M_i$ // 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:                 $\beta[] = E(M_i(u), j)$ // find the new paths
7:                 for $k$ in $\beta$ do // for all edges discovered
8:                     $A = \beta[k]$
9:                     if (M - $\beta[k]$+!$A$) > M then // if swapping paths increases flow
10:                        $M[i]$ =!$A$ // save augmenting path changes
11:                end for
12:            end for
13:    end for
```

Algorithm 1 provides a brief implementation for augmenting paths within a flow network. Starting with an bipartite graph G and the current set of edges within the matching M we evaluate edges not currently apart of 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 i in M, which is associated with a vertex in the set u identified by $M_i(u)$ and find the corresponding v vertex for that edge denoted $M_i(v)$. 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 Hopcroft-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

The initial sequential algorithm 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

For the algorithm choices discussed in section 1.5, 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 Hopcroft-Karp maximum flow solver.

We have developed our sequential implementation using C++. The 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 be the first vertex 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.

As can be seen in Fig. 1.4, the Hopcroft-Karp function calls additional helper functions which perform Breadth First Search (BFS), as well as Depth First Search (DFS) from their respective vertex positions during traversal. Similar to the example shown in Fig. 1.5, Breadth first search on line 43 of the Hopcroft-Karp function is used to determine if an alternating path exists from the current vertex set back to the opposing set. The alternating path, should it exist, would suggest that it may be possible to have a different possible matching. The vertices discovered during BFS represent the vertices we need to investigate for augmenting paths. Augmenting paths are additional paths between the vertex sets that may be selected in an effort to obtain a better matching. A DFS originating from the vertices discovered during BFS evaluate if they are able to reach the opposing vertex set. If this is the case, than the alternating path is an augmenting path and we need to check whether it is a better fit for our matching.

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()) { // breadth first search
        // 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)) { // depth first search
                result++;
            }
        }
    }
    return result;
}
```

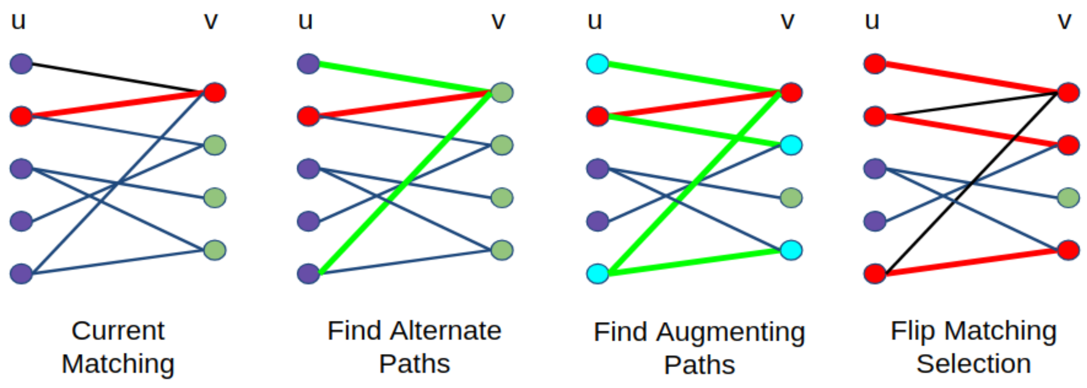


Figure 1.5: Here we can see an example of the discovery and evaluation of alternating and augmenting paths by using BFS and DSF respectively. The initial matching is used to find any alternate paths. The vertices associated with alternate paths are then evaluated for possible augmenting paths back to v. If any exist we perform a swap of the vertices/edges, increasing the matching M accordingly.

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. 1.6 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.

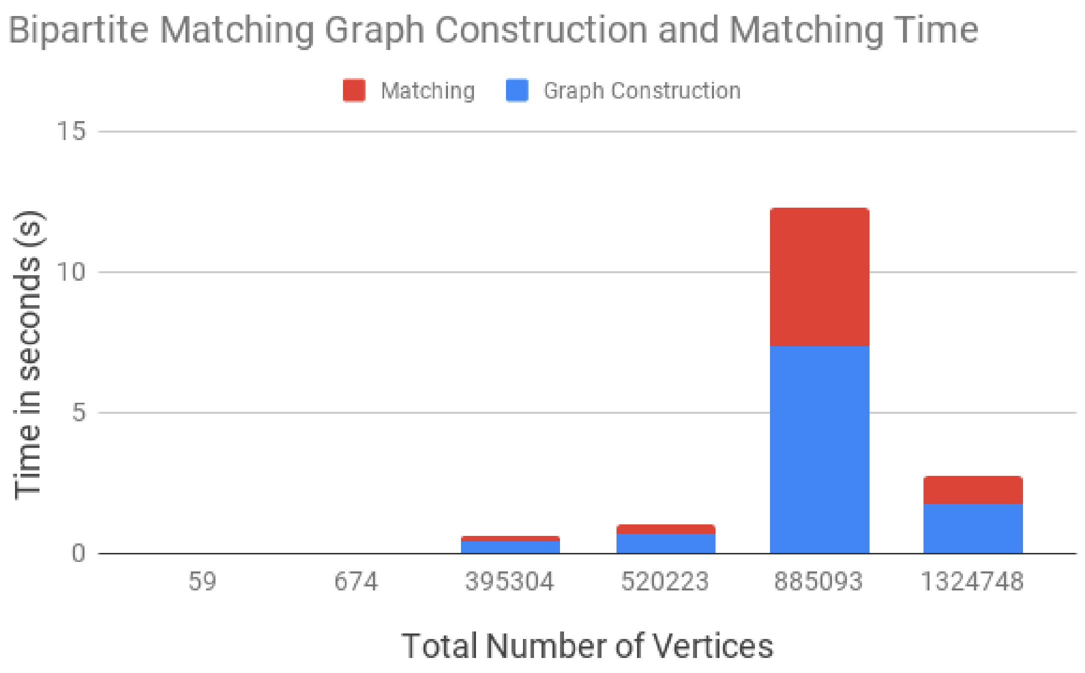


Figure 1.6: This chart shows the graph construction (blue) and maximum cardinality matching (red) time for 7 bipartite graphs, shown in Table 1.1. The graphs were taken from the Suite Sparse Matrix Collection and very in vertex and edge counts.

This is because total the edges of a graph, and not just its vertices, govern how that graph is traversed. able 1.1 lsits 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.

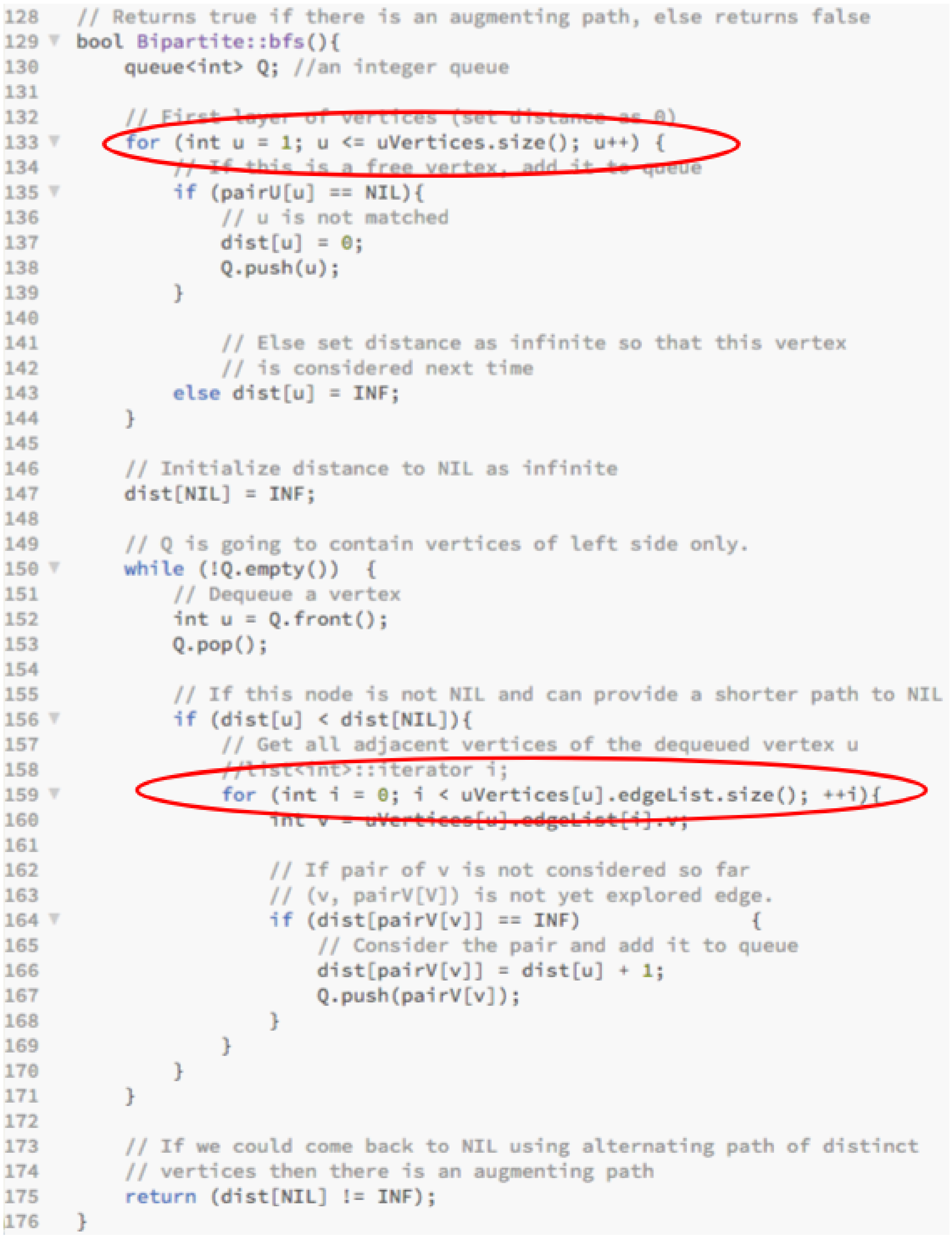
Edge count within a bipartite graph can have a drastic affect on MCM determination, as a greater number of edges can generate a much greater number of augmenting paths. Additional augmenting paths require additional computation, thereby increasing the overall runtime.

1.8 An Enhanced Algorithm

Or first attempt at a multithreaded bipartite matching implementation utilized OpenMP on a single shared memory machine. We opted for the use of parallel for loops with OpenMP for the loops which call BFS and DFS inside the Hopcroft-Karp algorithm. This would create a separate OpenMP threaded for every iteration of the loop, thereby allowing every BFS and DFS to be ran simultaneously on their respective origin vertices.

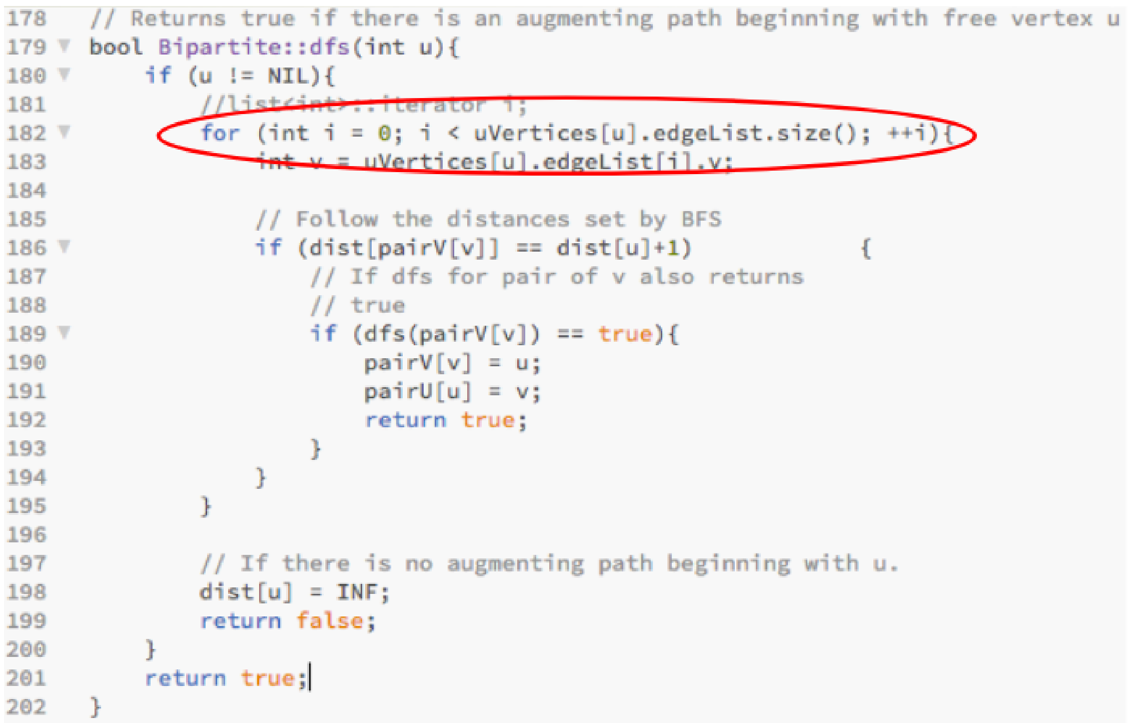
1.9 A Reference Enhanced Implementation

The mutltithreaded implementing of the sequential application share many similarities. We sought to increase concurrency by creating as many threads as possible during execution. Not only was this done with the Hopcroft-Karp algorithm itself, but also with the BFS and DFS functions which Hopcrof-Karp() calls.



```
// Returns true if there is an augmenting path, else returns false
bool Bipartite::bfs(){
    queue<int> Q; //an integer queue

    // First layer of vertices (set distance as 0)
    for (int u = 1; u <= uVertices.size(); u++) {
        // If this is a free vertex, add it to queue
        if (pairU[u] == NIL){
            // u is not matched
            dist[u] = 0;
            Q.push(u);
        }

            // Else set distance as infinite so that this vertex
            // is considered next time
        else dist[u] = INF;
    }

    // Initialize distance to NIL as infinite
    dist[NIL] = INF;

    // Q is going to contain vertices of left side only.
    while (!Q.empty())  {
        // Dequeue a vertex
        int u = Q.front();
        Q.pop();

        // If this node is not NIL and can provide a shorter path to NIL
        if (dist[u] < dist[NIL]){
            // Get all adjacent vertices of the dequeued vertex u
            //list<int>::iterator i;
            for (int i = 0; i < uVertices[u].edgeList.size(); ++i){
                int v = uVertices[u].edgeList[i].v;

                // If pair of v is not considered so far
                // (v, pairV[V]) is not yet explored edge.
                if (dist[pairV[v]] == INF)              {
                    // Consider the pair and add it to queue
                    dist[pairV[v]] = dist[u] + 1;
                    Q.push(pairV[v]);
                }
            }
        }
    }

    // If we could come back to NIL using alternating path of distinct
    // vertices then there is an augmenting path
    return (dist[NIL] != INF);
}
```

Figure 1.7: The primary work loop within the Breadth First Search function, which was placed within an OpenMP pragma for multithreading



```
// Returns true if there is an augmenting path beginning with free vertex u
bool Bipartite::dfs(int u){
    if (u != NIL){
        //list<int>::iterator i;
        for (int i = 0; i < uVertices[u].edgeList.size(); ++i){
            int v = uVertices[u].edgeList[i].v;

            // Follow the distances set by BFS
            if (dist[pairV[v]] == dist[u]+1)          {
                // If dfs for pair of v also returns
                // true
                if (dfs(pairV[v]) == true){
                    pairV[v] = u;
                    pairU[u] = v;
                    return true;
                }
            }
        }

        // If there is no augmenting path beginning with u.
        dist[u] = INF;
        return false;
    }
    return true;
}
```

Figure 1.8: The primary work loop within the Depth First Search function, which was placed within an OpenMP pragma for multithreading

Fig. 1.7 and 1.8 show the for loops which were placed inside OpenMP $\#pragmaompfor$ blocks. It worth nothing that for this initial multithreaded implementation we chose not to explicitly assign the use of shared or private variables. Depending on the graph and vertex/edge selected for exploration this can affect the performance of an arbitrarily selected OpenMP thread.

The multithreaded version of the Hopcroft-Karp function can be seen in Fig. **??**. The important changes are visible on lines 55 and 60. On line 55 we are allowing OpenMP to create a thread for each free vertex that needs to be checked for an augmenting path. Within this for loop we check for augmenting paths by performing DFS from the free vertex. If an augmenting path exists from this free vertex, we add it to the matching and continue looking for augmenting paths. The atomic operation on line 60 is required to prevent race conditions on the matching result by OpenMP threads.

Similar to the sequential implementation, this continues until there are no augmenting paths which require evaluation, and the result is then returned.

1.10 Enhanced Scaling Results

Discuss here results from the enhanced algorithm. Include software and hardware configuration, where the input graph data sets came from, and how input data set characteristics were varied. Ideally plots of performance vs BOTH problem size changes AND hardware resources are desired. Did the performance as a function of size vary as you predicted?

Fig. 1.10 illustrates the comparison between the sequential implementation and the multi-threaded version. Both implementations were run on a single node of the Center for Research Computing cluster at the University of Notre Dame. As can be seen, the sequential implementa-

Figure 1.9: 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()) { // breadth first search
        // Find a free vertex
        #pragma omp for
        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)) { // depth first search
                #pragma omp atomic
                result++;
            }
        }
    }
    return result;
}
```

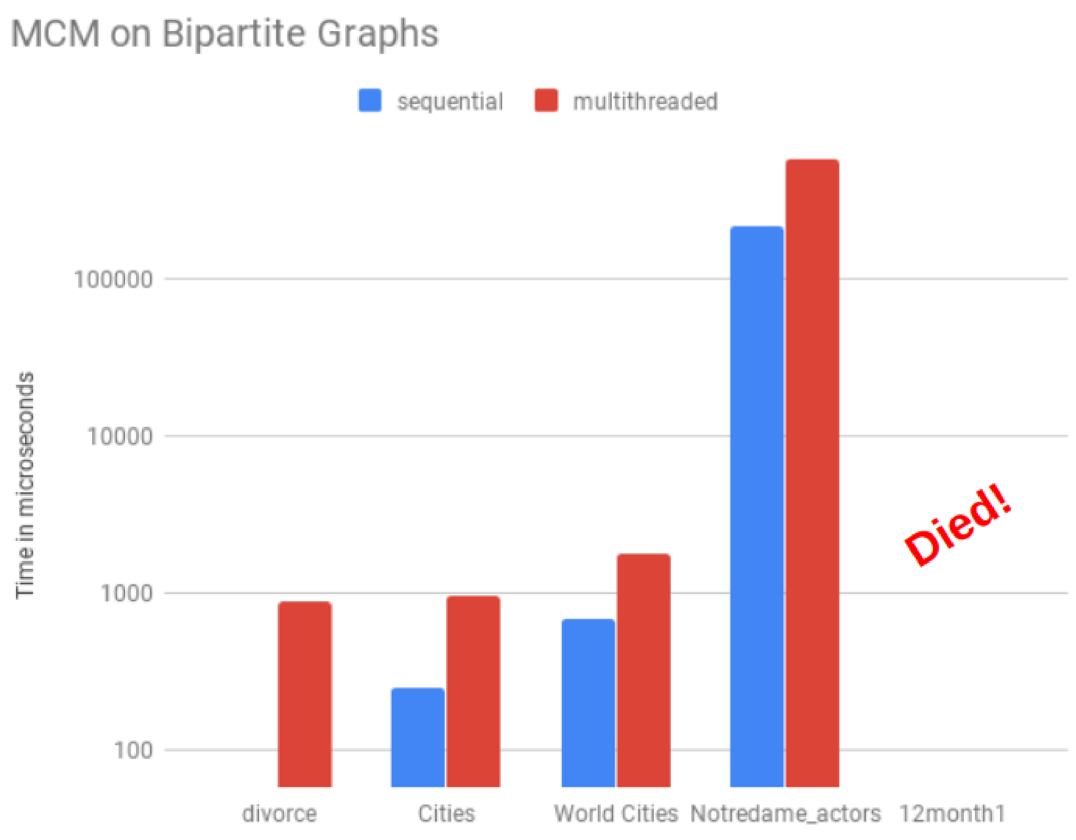


Figure 1.10: This chart shows the graph construction (blue) and maximum cardinality matching (red) time for 7 bipartite graphs, shown in Table 1.1. The graphs were taken from the Suite Sparse Matrix Collection and very in vertex and edge counts.

tion out performs our initial multithreaded version by a large margin. For *divorce* the sequential versions' runtime was so small such that it does not render properly using the log scale present in Fig. 1.10.

The reason for the performance degradation when implementing the sequential algorithm with OpenMP comes from the use of atomic operations. While many portions of the code may be done in parallel, such as BFS or DFS from different origin vertices, the final comparison and assignment to the matching M must be performed atomically. This created a significant barrier preventing threads to continue with augmenting path analysis until they were allowed to access the updated result vector. This behavior is typical for tightly couple algorithms such as is the case with bipartite matching where the selection of one path may prevent the selection of another.

After further investigation it is believed that very careful data partitioning among threads may alleviate a considerable portion of this overhead. However that requires further study into the optimal workload distribution for bipartite graphs on shared memory machines.

1.11 Conclusion

While there are many optimized algorithms for graph traversal, bipartite matching remains an challenging problem. Our attempts to increase performance by leveraging multithreading via OpenMP proved to be unsuccesful. However, this does not indicate that improved perforance using OpenMP is not possible. It may be possible to restructure the problem and or graph element assignment among threads such that there is very little overlap or need for atomic operations. Additionally we believe that an over abundance of parallel fors may have exacerbated the atomic operation barrier behavior.

In the future we hope to evaluate additional data partitioning methods in an effort to achieve increased multithreded performance. Subsequently the development of a multiprocess implementation using MPI is necessary in order to evaluate the scaling potential of hybrid codes for bipartite matching. While an MPI version was attempted, the accompanying errors were far too numerous for its discussion to be included in this paper.

1.12 Response to Reviews

Included a table about the graphs/matrices used in my analysis.

The statements made in this paper regarding maximal and or maximum matchings for the example graphs in section 1.2 are valid.

An indepth discussion of Ford-Fulkerson algorithms performed on bipartite graphs is not given in this paper. This is because the intent was to mention possible methods and then discuss "a" method in greater detail. In face nearly ANY maximum flow algorithm may be used to determine the maximum bipartite matching however the discussion of all possible methods was far beyond the scope of this paper.

Regarding data sets and the lack of a specific data set, this was not discussed as no singular data set exists. Quite literally, a graph in which there are two distinct sets of vertices has the potential to be bipartite. At best someone can look for graphs known to be bipartite or generate one themselves. However no specific bipartite graph collection is known by the author to exist, making this point moot.

Anyone who feels that they can better format my references from wikipedia into bibtex, is more than welcome to do so.

"Pseudocode is not present for the entire algorithm" This is likely referencing the lack of breadth first search or depth first search code. These algorithms are so common that their implementation in this regards is trivial and did not need to be present. It was however added later when discussing the introduction of OpenMP into BFS and DFS.

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.