

Application of Edmonds-Karp Algorithm to Solve Maximum Cardinality Bipartite Matching Problem

Morgen Sudyanto 13518093
 Program Studi Teknik Informatika
 Sekolah Teknik Elektro dan Informatika
 Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia
 13518093@std.stei.itb.ac.id

Abstract—In graph theory, a bipartite graph is a graph that can be divided into two independent sets. A matching in a bipartite graph is a set of edges chosen such that no two edges have a same endpoint. In the maximum matching of a graph, we can find the maximum number of edges in all possible matching sets. Maximum Cardinality Bipartite Matching can be used to solve real world assignment problems. In this paper, I will try to use the Edmonds-Karp algorithm to determine the maximum matching of a bipartite graph.

Keywords—graph, bipartite, maximum matching, edmonds-karp algorithm.

I. INTRODUCTION

Suppose you are a Human Resources Manager in a company. There are many people working in your company and you have some projects that need to be worked on. These people have different sets of project preferences. Your job is to assign these people to each of the projects, giving as many people their preferred projects as possible.

This problem can be modeled as a bipartite graph, with the first set containing all the workers and the second set containing all the projects. The edges are each of the workers' preferences. In a correct matching, each worker will be assigned to one preferred project. Thus, in a maximum matching, there will be a maximum number of workers that are assigned to their preferred projects.

II. THEORIES

A. Graph

A graph is an object consisting of two sets called its vertex (node) set and edge set [1]. The vertex set is a nonempty set. The edge set contains of two element subsets of the vertex set. The edge set can also be empty. Two different vertices are adjacent if and only if both of the vertices are contained in the edge set. The word *vertex* and *node* will be interchangeably used in this paper.

A graph can be undirected or directed. In an undirected graph, the order of elements in the edge set doesn't matter (i.e. $\{A, B\} = \{B, A\}$). In a directed graph, the order of elements in the edge set matter as it shows that there is an edge from one vertex to another, but not the opposite.

A path leads from a vertex to another vertex through the edges

of the graph. A cycle is a path where the first and last vertex is the same. A tree is a graph that does not contain any cycle.

Graph can be represented in many ways, such as adjacency matrix, adjacency list, and graph diagram.

	0	1	2	3	4
0	0	1	0	0	1
1	1	0	1	1	1
2	0	1	0	1	0
3	0	1	1	0	1
4	1	1	0	1	0

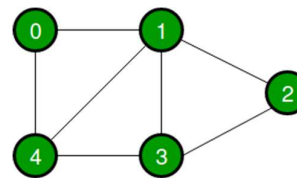
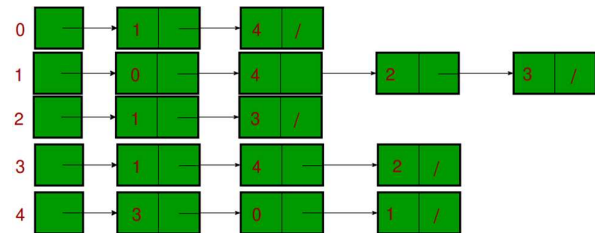


Figure 1. Different representations of a graph [2]

B. Bipartite Graph

A bipartite graph is a graph whose vertices can be partitioned into two disjoint sets V_1 and V_2 and all edges (u, v) in the vertex set has the property that $u \in V_1$ and $v \in V_2$ [3]. A graph is bipartite if it is possible to color its vertices using two colors in such a way that no adjacent vertices have the same color [4]. A bipartite graph also has an interesting property: It could not have a cycle with an odd number of edges.

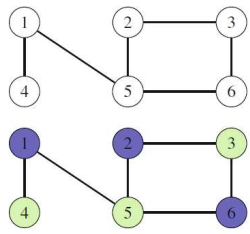


Figure 2. Graph and its coloring [4]

C. Matching

A matching of a graph is a subset of edges such that no two edges share the same vertex [3]. In the Maximum Cardinality matching, we want to know the maximum number of edges that we can take in our subset. There is also another type of matching called Perfect Matching, where we can take all of the edges in our edge set and still achieve a correct matching. Currently, we are interested in Maximum Cardinality of a bipartite graph.

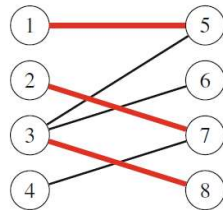


Figure 3. Maximum Cardinality matching of a bipartite graph [4]

D. Flow Network

A flow network is a directed graph that contains two special nodes: a source node with no incoming edges and a sink node with no outgoing edges [4]. Each vertex in the graph are connected using directed edges that has a specified capacity. In each node, the incoming and outgoing flow has to be equal.

In a maximum flow problem, we need to find the maximum number of flow that we can send from the source node to the sink node while not exceeding the flow capacity in any edge.

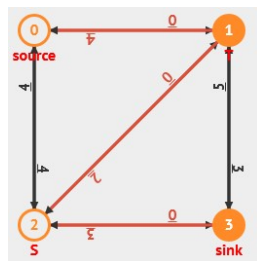


Figure 4. Maximum flow of a flow network [5]

The Maximum Cardinality Bipartite Matching (MCBM) problem can actually be reduced into a maximum flow problem. We can call all the vertices in the first set as the “Left” vertices and all the vertices in the second set as the “Right” vertices. Then, we add two vertices, a source vertex and a sink vertex. The source vertex will then be connected to all the left vertices with a capacity of 1 and the sink vertex connected to all the right vertices with a capacity of 1. All edges that connect the vertices

from the left set and the right set will also be given a capacity of 1. The maximum flow of the network is the maximum cardinality of the graph. In order to see the selected edges, we can “peek” at the flow of each edges. If the flow is 1, then the edge is contained at the maximum cardinality matching set.

E. Ford Fulkerson Method

Ford Fulkerson method is an iterative algorithm that repeatedly finds augmenting path: a path from source vertex to sink vertex that passes through positive weighted edges in the residual graph [3]. A residual graph is a graph that contains the remaining capacity of an edge after some flow pass through it. After finding an augmenting path, Ford Fulkerson method will decrease the capacity of forward edges and increase the capacity of backward edges along path the augmenting path.

This method decreases the capacity of forward edges because by sending a flow through an augmenting path, we will decrease the remaining capacities of the forward edges. The increasing of backward edges capacity allows the algorithm to cancel some of the capacity used in previous iterations (maybe wrong augmenting paths have been chosen before). In finding the augmenting path, we can use the Breadth First Search algorithm or the Depth First Search algorithm.

An implementation of the Ford Fulkerson method that uses Depth First Search to find augmenting paths has a complexity of $O(mf * E)$ where mf is the maximum flow of the flow network and E is the number of edges. This is clearly not a really good complexity, especially if the maximum flow may balloon to increasingly large numbers, even though there is only a small amount of edges.

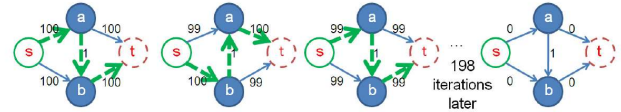


Figure 5. Ford Fulkerson Method in finding maximum flow of a flow network [3]

F. Edmonds-Karp Algorithm

Edmonds-Karp algorithm was discovered by Jack Edmonds and Richard Manning Karp. Edmonds-Karp algorithm uses Breadth First Search to find the shortest path in terms of the number of hops between the source vertex and the sink vertex. Using the flow network in figure 5, Edmonds-Karp algorithm only needs 2 paths, namely s-a-t and s-b-t, each sending 100 flows to obtain the maximum flow. BFS makes sure that the path chosen would not be the longer path (s-a-b-t or s-b-a-t).

With this algorithm, all augmenting paths will be exhausted after $V * E$ iterations, where V is the number of vertices and E is the number of edges in the flow network. As one BFS iteration has a complexity of $O(E)$, then the total complexity of this algorithm is $O(VE^2)$. This means that the total runtime of the algorithm only depends on the network size and not on maximum flow.

III. SOLVING THE PROJECT ASSIGNMENT PROBLEM

A. The Problem

This will be the same problem as stated in the introduction, but with more details. There are N workers and M projects in a company. Each worker (numbered from 1 to N) can only take one project. Each project (numbered from 1 to M) can be assigned to any number of workers, but with a maximum of A_i workers. Furthermore, each worker has their preferred projects (maybe they are more proficient in some programming languages). This will be represented in a list L of pairs (w, p) where w is the worker ID and p is the project ID. Find the maximum number of workers that can get their preferred project.

Example:

$N = 5$

$M = 3$

$A = [1, 2, 2]$

$LSize = 8$

$L = [[1, 1], [1, 3], [2, 1], [2, 3], [3, 2], [3, 3], [4, 1], [5, 2]]$

Explanation:

There are 5 workers in the company and 3 projects. Project 1 has a maximum of 1 worker, and project 2 and 3 has a maximum of 2 workers. Each worker's preferences:

- Worker 1: Project 1 and 3
- Worker 2: Project 1 and 3
- Worker 3: Project 2 and 3
- Worker 4: Project 1
- Worker 5: Project 2

B. Greedy Solution

There is an obvious (but wrong) greedy solution to this problem. We can greedily take the possible assignment, starting from the projects with lower number. In the above example, we will first assign project 1 to worker 1. Then, as we could not assign project 1 to any more worker, we will assign project 3 to worker 2. Next, we assign project 2 to worker 3. Now, we can see that worker 4 could not be given any projects. Lastly, we give project 2 to worker 5. We end up with four pairs of workers working on a project.

Worker	Project
1	① 3
2	1 ③
3	② 3
4	1
5	②

Figure 6. Greedy project assignment

Why is this not optimal? If we observe closely, we can actually give the first worker project 3. That way, worker 4 would be given a project – namely project 1.

Worker	Project
1	1 ③
2	1 ③
3	② 3
4	①
5	②

Figure 7. Optimal project assignment

C. Modelling the Problem as a Flow Network

This problem can actually be modelled into a Maximum Cardinality Bipartite Matching problem. Each project can be given a label of 6 to 8 (6 = number of workers + 1 and 8 = number of workers + number of projects). Then we add two extra vertices, 0 and 9. Vertex 0 will act as a source and vertex 9 will act as a sink. We will add an edge with a capacity of 1 that goes from the source vertex to all workers. Then, we also add an edge with a capacity of A_i from each of the projects to the sink vertex. Lastly, we connect all the workers with their preferred projects and give a capacity of 1. Clearly, the workers and projects make a bipartite graph. The flow network will be something like this:

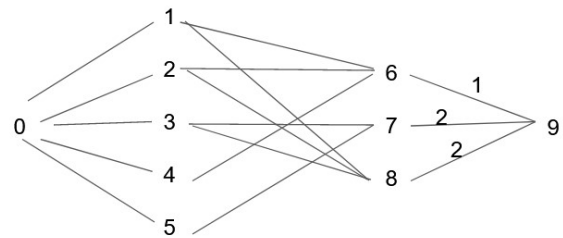


Figure 8. The modelled flow network – the edges without number actually has a capacity of 1.

D. C++ Implementation

To solve this problem, I will use the C++ programming language. The reason for this is because C++ has a lot of builtin data structures and functions that can help in solving this problem.

I. Global Variables

```
int capacity[1005][1005];
int flowPassed[1005][1005];
int path_flow[1005];
int parent[1005];
vector<int> adj[1005];
```

Figure 9. Global variables declared in my implementation

The explanation for each of the global variables are shown below:

- Capacity is a 2-D array that stores the capacity of each edges that are present. The value of capacity[i][j] will be zero if there is no edge that connects vertex i to vertex j. This array will not be modified throughout the maxflow algorithm.
- FlowPassed is a 2-D array that stores the amount of flow that has passed through a particular edge. This array is modified after every BFS iteration.
- Path_flow is a 1-D array that stores the amount of flow that is allowed in a particular BFS iteration. This array will be reinitialized at the start of every BFS iteration.
- Parent is a 1-D array that stores the parent of a particular vertex. This allows the backtracking of an augmenting path. This array will be reinitialized at the start of every BFS iteration.
- Adj is a 1-D vector that stores the graph, represented as an adjacency list. This will not be modified throughout the maxflow algorithm, just like the capacity array.

The number 1005 is my assumption that the number of workers and projects will not exceed 1000. This can be extended for larger cases.

II. Maximum Flow Function

```
int maxflow(int source, int sink) {
    int maxFlow = 0;
    while (true) {
        int flow = BFS(source, sink);
        // all edges exhausted. not giving any more flows.
        if (flow == 0) break;
        int cur = sink;
        // add the flow to our answer
        maxFlow += flow;
        // backtracking the path
        while (cur != source) {
            int par = parent[cur];
            // decrease capacity of backward edges
            flowPassed[par][cur] += flow;
            // increase capacity of forward edges
            flowPassed[cur][par] -= flow;
            cur = par;
        }
    }
    return maxFlow;
}
```

Figure 10. Maximum flow algorithm, the iterations of finding augmenting paths are done by calling the BFS function repeatedly

My maxflow function simply stores the result of all BFS iterations. I made an intentionally “infinite” loop with a while(true) statement. This loop is guaranteed to stop at some point, as it had been proven that Edmonds-Karp algorithm will stop after a maximum of $V \cdot E$ iterations. The parent array stores the augmenting path so that I can backtrack through it from the sink vertex to the source vertex, and modify the residual graph accordingly. The number that is returned by this function is the maximum flow of the flow network.

III. BFS Function

```
int BFS(int source, int sink) {
    // reinitializing the parent and path_flow array
    memset (parent, -1, sizeof(parent));
    memset (path_flow, 0, sizeof(path_flow));
    // initializing the queue (used for BFS)
    queue<int> q;
    // the source has no parent, and should not be visited
    parent[source] = -2;
    // push the source vertex to the queue
    q.push(source);
    // the source has infinite flow
    path_flow[source] = INF;
    // BFS Algorithm
    while (!q.empty()) {
        // get the front vertex, and pop it out of the queue
        int cur = q.front();
        q.pop();
        // iterate through all adjacent vertices
        for (int i=0;i<adj[cur].size();i++) {
            // obtain the adjacent vertex
            int next = adj[cur][i];
            // don't visit a visited vertex
            if (parent[next] == -1) {
                // if flow is still below capacity
                if (capacity[cur][next] > flowPassed[cur][next])
                {
                    // set the parent for backtracking
                    parent[next] = cur;
                    // the allowed flow is the minimum of
                    // current flow in current node and allowed
                    // flow in an edge
                    path_flow[next] = min(path_flow[cur],
                    capacity[cur][next] - flowPassed[cur][next]);
                }
                // if we already arrived at the sink,
                // simply return the flow
                if (next == sink) return path_flow[cur];
                // insert the new vertex into queue
                else q.push(next);
            }
        }
    }
    return 0;
}
```

Figure 11. BFS Function for finding augmenting path of a graph

This function will return the flow of an augmenting path and shows the path itself (through the parent array). At the start of this function, I initialized the parent and path_flow array to -1 and 0 respectively. Then, I used a built-in data structure named queue to simulate the First In First Out nature of the BFS algorithm. I marked the parent of source vertex with -2. Actually, any negative number other than -1 will work. The reason for this is because the source vertex has no parent and should not be revisited by other vertices. I also give the path_flow of the source node an arbitrarily large number, INF to simulate the infinite flow in the source node. INF is actually defined at the start of my program as $10^9 + 7$. Of course, this number is also based on my assumption that the maximum flow of the network will not exceed that number, which is definitely true, as the number of workers will obviously be lower than 1 billion.

Then, I run a modified BFS algorithm. This BFS algorithm will save the parents of every chosen vertex. A vertex will also available to be chosen if the flow is still strictly lower than the

capacity. After choosing a vertex, I gave a flow of that particular vertex the minimum of the current vertex and the allowed flow in that particular edge. I returned the flow of the sink vertex at the end of our iteration.

IV. Main Program

```
int main () {
    //clock_t start = clock();
    int N, M;
    cout << "Number of workers: ";
    cin >> N;
    cout << "Number of projects: ";
    cin >> M;
    int source = 0, sink = N + M + 1;
    memset (capacity, 0, sizeof(capacity));
    for (int i=1;i<=N;i++) {
        adj[0].push_back(i);
        adj[i].push_back(0);
        capacity[0][i] = 1;
    }
    for (int i=1;i<=M;i++) {
        int maxAssigned;
        cout << "Workers required for project " << i << ": ";
        cin >> maxAssigned;
        int projectId = N + i;
        adj[projectId].push_back(sink);
        adj[sink].push_back(projectId);
        capacity[projectId][sink] = maxAssigned;
    }
    int preference;
    cout << "Number of preferences: ";
    cin >> preference;
    cout << "Preference list:" << endl;
    for (int i=1;i<=preference;i++) {
        int x,y;
        cin >> x >> y;
        // convert project into projectId
        y += N;
        adj[x].push_back(y);
        adj[y].push_back(x);
        capacity[x][y]=1;
    }
    cout << "Maximum flow: " << maxflow(source, sink) << endl;
    for (int i=1;i<=N;i++) {
        for (int j=N + 1;j <= N + M;j++) {
            if (flowPassed[i][j]) cout << "Worker " << i << "
                is assigned to project " << j-N << endl;
        }
    }
    //cerr << fixed << setprecision(3) << (clock()-start)*1./
    CLOCKS_PER_SEC << endl;
    return 0;
}
```

Figure 12. Main program

This implementation of the main program is straightforward. First, the program received the number of workers and projects – denoted as N and M respectively. Then, I created a new vertex named source and sink (0 and N+M+1). I connected the source vertex to all the workers, giving each of the edges a capacity of 1. Next up is receiving the array A – the maximum of worker assigned to one specific project. I did not save the array A, but I directly add an edge that connects a project ID (N + i) to the sink vertex, then asked the user to specify the value of A, and that value would be that edge’s capacity. After that, I read the preference list of the workers. I add an edge that connects a worker to a project ID and give it a capacity of 1.

Done! The flow network is now complete. All that’s left was to run Edmonds-Karp algorithm by calling the maxflow function. I then outputted the maxflow of that flow network. In order to obtain the correct assignments, I looked at the flowPassed array (only the edges that connects the workers and project IDs). If the value of that edge is 1, then that edge is selected to be inside the Maximum Cardinality Bipartite Matching set. Note that if the edge is not present in the edge set, its value would be zero – it would be taken.

I also commented the first line and the third last line of my program. Those lines are used for debugging purposes - finding the total time taken used by my program. This program will not be outputting a non-preferred assignment.

E. Running the Program

We will now run the program, using the example in section III A.

```
C:\Users\moondemon68\Downloads\mcbm
λ mcbm
Number of workers: 5
Number of projects: 3
Workers required for project 1: 1
Workers required for project 2: 2
Workers required for project 3: 2
Number of preferences: 8
Preference list:
1 1
1 3
2 1
2 3
3 2
3 3
4 1
5 2
Maximum flow: 5
Worker 1 is assigned to project 3
Worker 2 is assigned to project 3
Worker 3 is assigned to project 2
Worker 4 is assigned to project 1
Worker 5 is assigned to project 2
```

Figure 13. Running the example

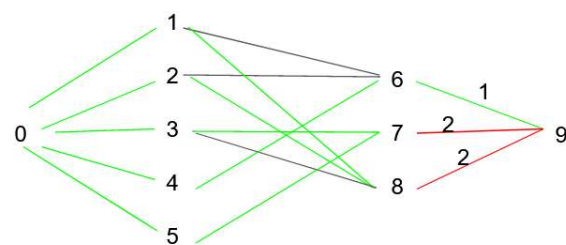


Figure 14. The maximum flow of the example case – edges colored black, green and red has a final flow of 0, 1 and 2 respectively

0	0	1
0	0	1
0	1	0
1	0	0
0	1	0

Figure 15. The resulting flow graph – rows represent workers and columns represent projects

With the greedy solution, we obtained the matching set of 4 elements. With the maximum flow solution, we obtained the

maximum matching of 5 elements. Although this seemed to be unimpressive, this improvement scales – meaning that if there are 10000 employees in a particular company, a 20% increase translates into 2000 more employees getting the projects they prefer, thus increasing the performance of that particular company.

IV. TESTING

In order to find out the performance of my program, I created several test cases of different N, M and L value, and various A constraints. The test cases are as follows:

1. N = 5, M = 3, L = 8 (Example)
2. N = 50, M = 50, L = 50, $1 \leq A_i \leq 2$
3. N = 50, M = 50, L = 50, $1 \leq A_i \leq 100000$
4. N = 1000, M = 50, L = 10000, $A_i = 1$
5. N = 50, M = 1000, L = 10000, $A_i = 1$
6. N = 500, M = 500, L = 250000, $1 \leq A_i \leq 2$
7. N = 500, M = 500, L = 500, $1 \leq A_i \leq 5$

The testcases are generated using Mike Mirzayanov’s testlib framework [6] with a seed of 682120. All test data, generator, source code and output can be accessed in my GitHub repository: <https://github.com/moondemon68/MCBM>.

The result of the testing phase are as follows:

Testcase	N	M	L	A_i	Time (s)
1	5	3	8	1 – 2	0.000
2	50	50	50	1 – 2	0.000
3	50	50	50	1 – 100000	0.000
4	1000	50	10000	1	0.015
5	50	1000	10000	1	0.015
6	500	500	250000	1 – 5	0.359
7	500	500	500	1 – 5	0.000

Table 1. Runtime comparison for several testcases

As we can see from the table above, the only variable that affects the runtime of the program is the number of preferences. There are two possible causes for this. First, it takes a longer time to read the input if the number of preferences is high. Second, the algorithm’s complexity is $O(VE^2)$, and the value of E increases as L increases.

V. CONCLUSION

This project assignment problem is just one out of many other applications of the Maximum Cardinality Bipartite Matching Problem. Other than Edmond-Karp algorithm, there are some more maximum flow algorithms that can solve this type of problems, such as Dinic, Hopcroft Karp, Edmonds’s Matching. These algorithms can solve a more generalized version of Maximum Cardinality Matching, where the graph may not be bipartite.

VI. ACKNOWLEDGMENT

Firstly, author thanks God for His blessings in helping me finish this paper.

Author thanks Mrs. Fariska Zakhralativa, M. T as the lecturer

of the IF2120 – Discrete Mathematics course for the guidance and knowledge that she shared.

Author also thanks my colleagues for all the support and inspiration that they had given.

Lastly, author apologizes if there are any intentional or unintentional mistakes in this paper.

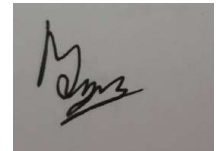
REFERENCES

- [1] Trudeau, Richard J. *Introduction to Graph Theory*. New York: Dover Publications. Accessed on December 3, 2019 from <https://books.google.co.id/books?id=eRLEAgAAQBAJ>.
- [2] Barnwal, Aashish. *Graph and Its Representations*. Accessed on December 3, 2019 from <https://www.geeksforgeeks.org/graph-and-its-representations/>.
- [3] Halim, Steven and Halim, Felix. *Competitive Programming 3*. Singapore: Lulu.
- [4] Laaksonen, Antti. *Guide to Competitive Programming*. Switzerland: Springer.
- [5] Halim, Steven. Visualgo. Accessed on December 4, 2019 from <https://visualgo.net/en/maxflow>.
- [6] Mirzayanov, Mike. Testlib. Accessed on December 4, 2019 from <https://github.com/MikeMirzayanov/testlib>.

PERNYATAAN

Dengan ini saya menyatakan bahwa makalah yang saya tulis ini adalah tulisan saya sendiri, bukan saduran, atau terjemahan dari makalah orang lain, dan bukan plagiasi.

Bandung, 4 Desember 2019



Morgen Sudyanto 13518093