Maximum Bipartite Matching

Ralph McDougall

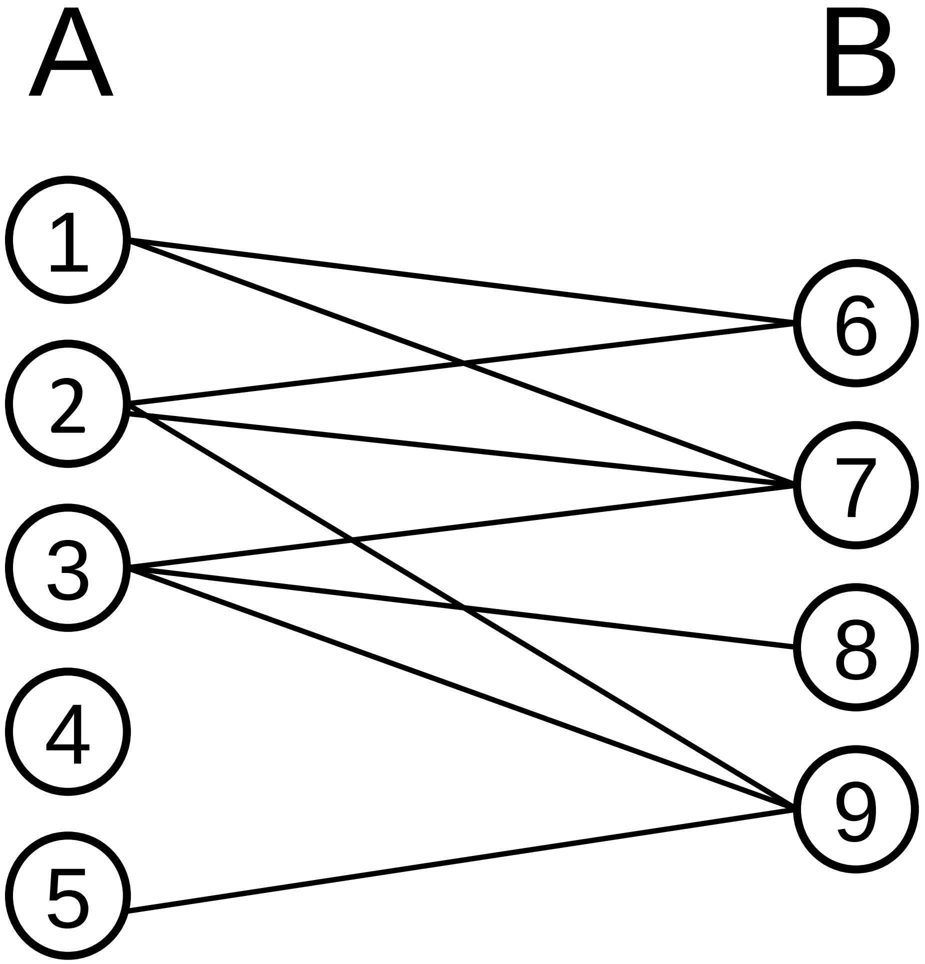
3 February 2018

Comeback tour: 9 March 2019

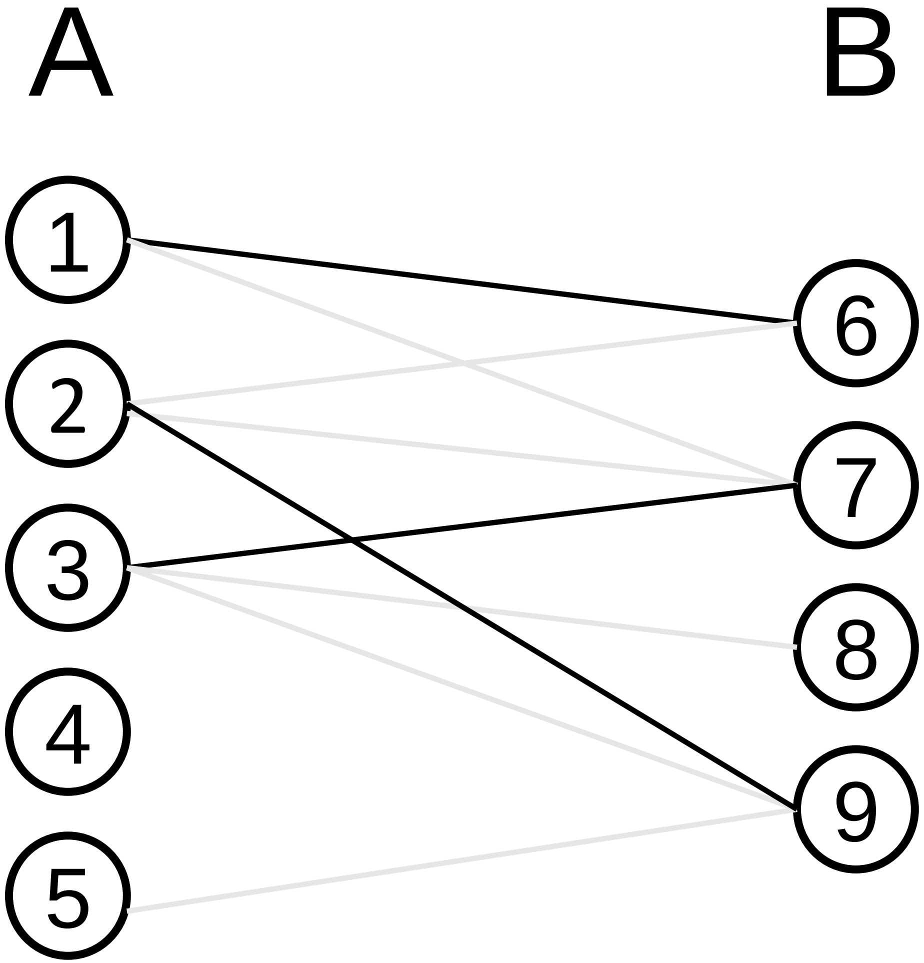
Definitions

- A graph is said to be a bipartite graph if it possible to split the nodes into 2 disjoint sets, A and B, such that nodes in set A only have edges leading to nodes in set B
- A bipartite matching is a bipartite graph such that no 2 edges have a common endpoint
- The maximum bipartite matching of a bipartite graph is the bipartite matching with the most edges

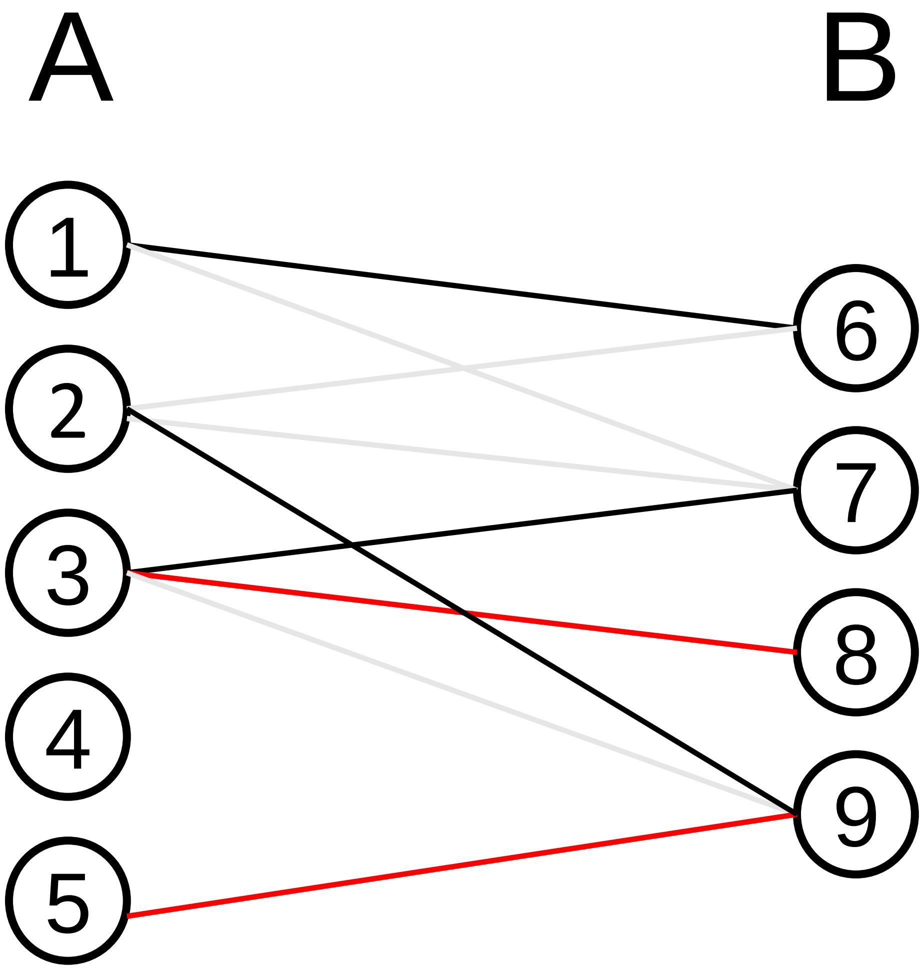
Example: Bipartite Graph



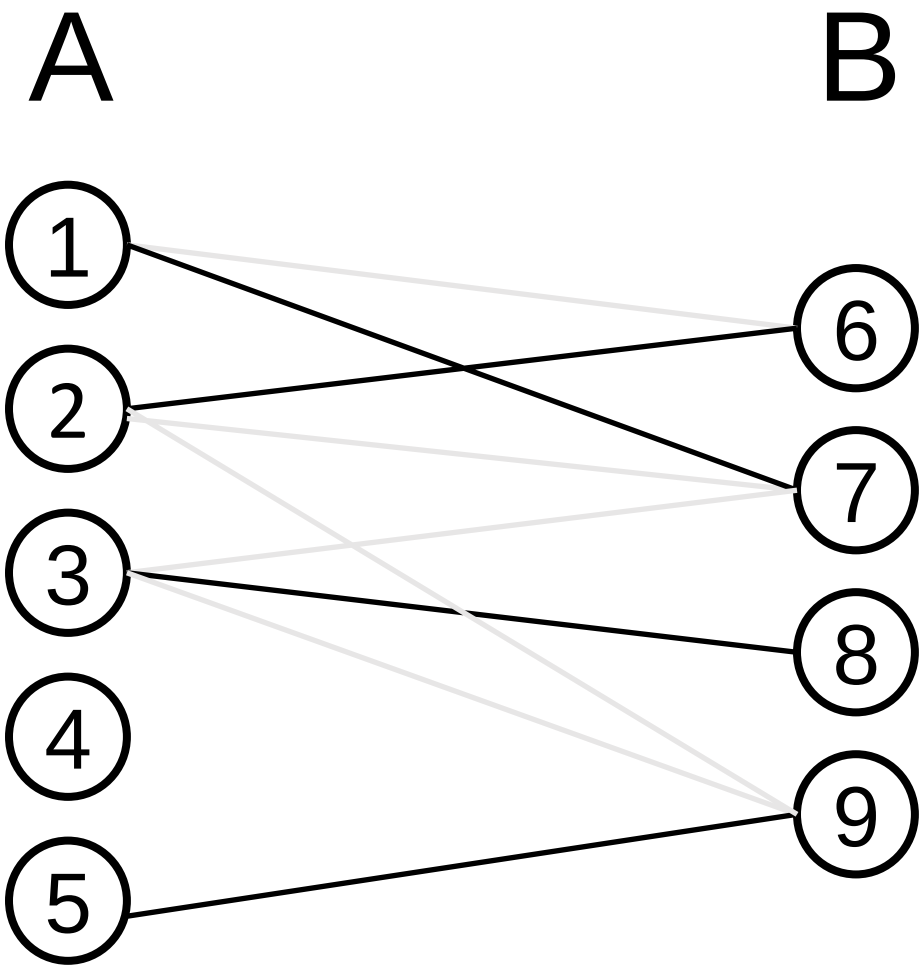
Example: Bipartite Matching



Example: NOT Bipartite Matching



Example: Maximum Bipartite Matching



Things to note

- If there are m nodes in set A and n nodes in set B, then the number of edges in the Maximum Bipartite Matching has an upper bound of min(m, n)
- There can be multiple maximum bipartite matchings

How is this useful?

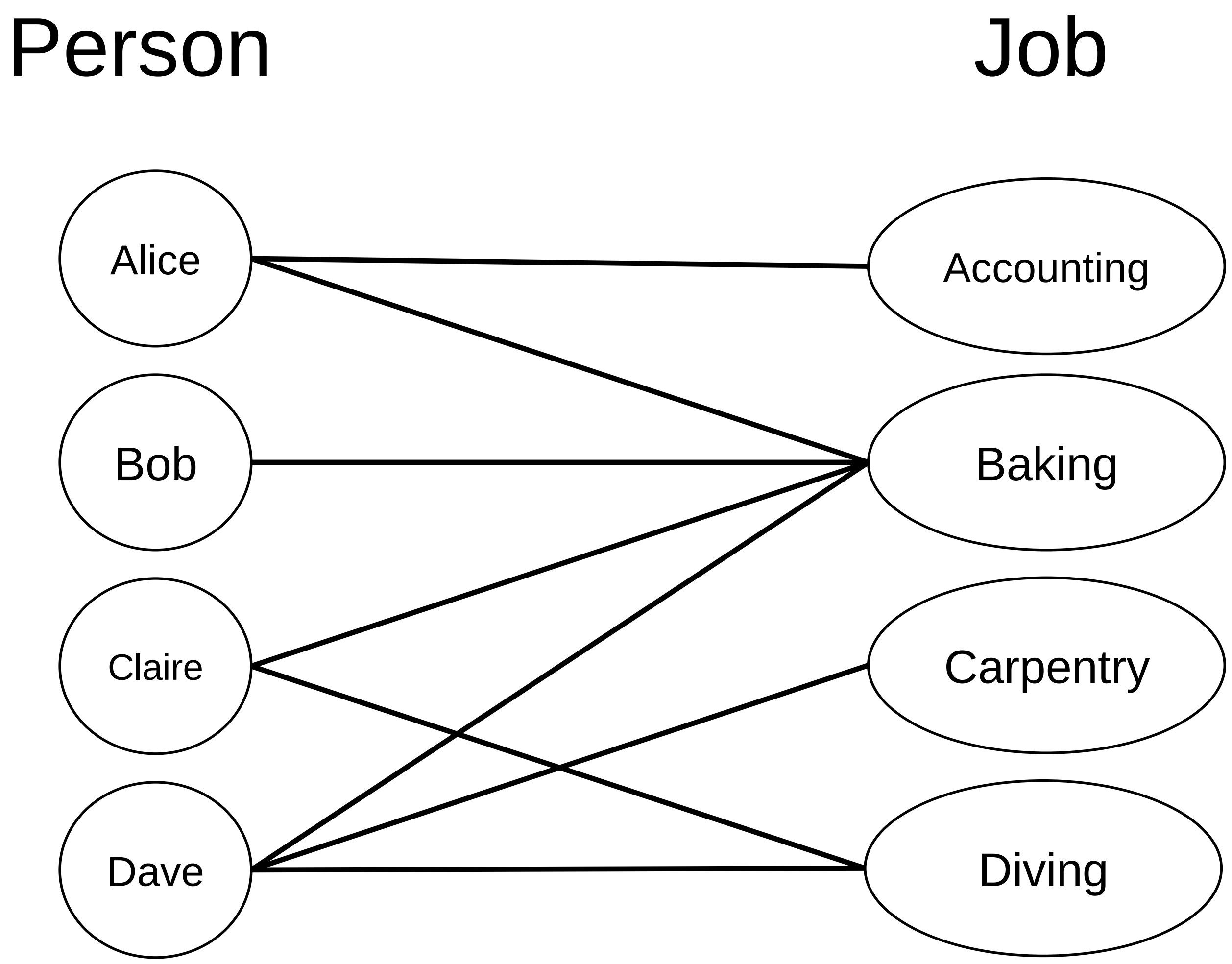
- Sample problem:

n people are applying for m jobs. Each person can only do 1 job and each job can only be done by 1 person. Given a list of jobs that each person is applying for, find the highest number of jobs that can be filled.

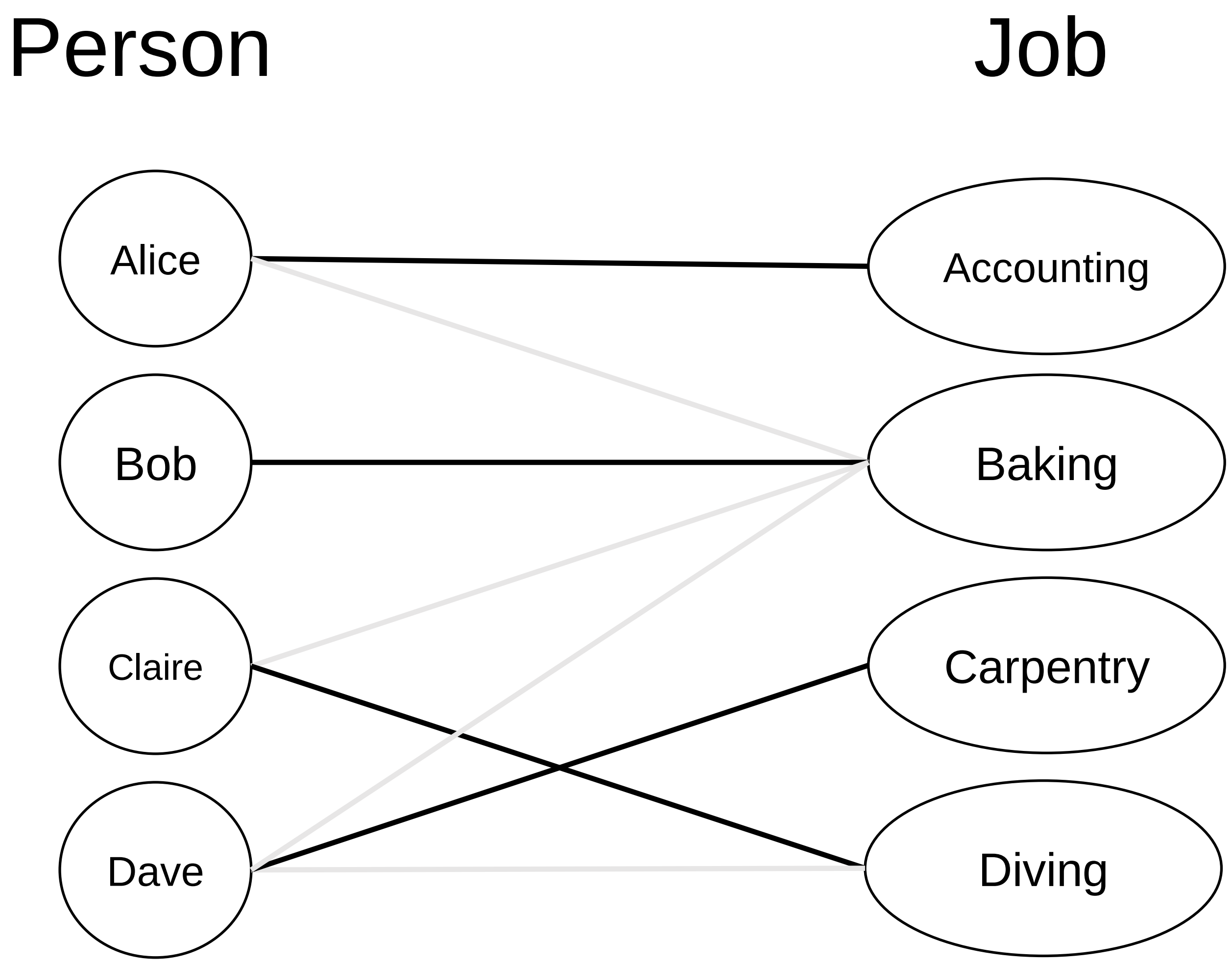
Sample of sample

Person	**Job being applied for**
Alice	Accounting
	Baking
Bob	Baking
Claire	Baking
	Diving
Dave	Baking
	Carpentry
	Diving

Alternate form of sample of sample



Maximum Bipartite Matching of alternate form of sample of sample



Solution to sample

- Alice does Accounting
- Bob does Baking
- Claire does Diving
- Dave does Carpentry

Network Flow

- The problem of finding a Maximum Bipartite Matching is a special case of the Network Flow problem
- A flow network is a weighted directed graph where each edge has a maximum capacity. Flow is sent from a source to a sink. The total flow into a node must be the same as the total flow out except for the source and the sink. The task is to maximise the flow that ends at the sink
- This problem can be solved with the Edmonds-Karp extension of the Ford-Fulkerson algorithm

Edmonds-Karp

- Find the path with shortest length from the source to the sink where none of the edges on the path are full (augmenting path)
- Pass as much flow as possible through that path
- Repeat until there are no more paths from the source to the sink

Code: Initialisation and input

```
#include <bits/stdc++.h>
#define INF 1000000000
using namespace std;

int main()
{
    vector< vector< vector< int > > > neighbours;
    int numEdges, source, sink, maxVertex, maxFlow = 0;

    ifstream inFile("edmondsKarp.in");
    inFile >> numEdges >> source >> sink;

    for (int i = 0; i < numEdges; ++i)
    {
        int a, b, f = 0;
        inFile >> a >> b >> f;
        vector<int> p = {b, f};

        maxVertex = max( max(a, b), maxVertex);

        while (((int) neighbours.size() - 1) <= a)
        {
            neighbours.push_back( vector< vector< int > >() );
        }
        neighbours[a].push_back(p);
    }
    inFile.close();
```

Code: BFS looking for path

```
while (true)
{
    vector<int> parent(maxVertex + 1, 0);
    vector<int> cap(maxVertex + 1, 0);

    queue<int> q;
    parent[source] = source;
    q.push(source);

    while (q.size() != 0)
    {
        int curr = q.front();
        if (curr == sink) break;
        q.pop();

        for (int i = 0; i < neighbours[curr].size(); ++i)
        {
            int neigh = neighbours[curr][i][0];
            if (neighbours[curr][i][1] != 0 && parent[neigh] == 0)
            {
                parent[neigh] = curr;
                cap[neigh] = neighbours[curr][i][1];
                q.push(neigh);
            }
        }
    }
```

Code: Finding shortest path and flow

```
if (parent[sink] == 0)
{
    break;
}

vector< int > revPath;
revPath.push_back(sink);

int minCapacity = INF;
int curr = sink;

while (curr != source)
{
    minCapacity = min(cap[curr], minCapacity);
    curr = parent[curr];
    revPath.push_back(curr);
}
maxFlow += minCapacity;
```

Code: Reducing capacity of path edges

```
        curr = revPath.size();

        while ( (--curr) > 0)
        {
            for (int i = 0; i < neighbours[revPath[curr]].size(); ++i)
            {
                if (neighbours[revPath[curr]][i][0] == revPath[curr - 1])
                {
                    cout << revPath[curr] << " " << revPath[curr - 1] << "\n";
                    neighbours[revPath[curr]][i][1] -= minCapacity;
                    break;
                }
            }
        }
    }
    cout << maxFlow << "\n";
}
```

Analysis

- This algorithm runs in $O(VE^2)$ time
- Dinic's Algorithm can be used to improve this to $O(V^2E)$ when the graph is very dense

Back to Maximum Bipartite Matching

- To turn the Maximum Bipartite Matching into a Network Flow problem we add a source that is connected to all of the nodes in set A and a sink that is connected to all of the nodes in set B
- All edges are given a weight of 1
- From here, the problem of finding the Maximum Bipartite Matching is clearly the same as finding the maximum Network Flow
- Since Maximum Bipartite Matching is such a special case of Network Flow, there does exist an algorithm than runs in O(VE)

Algorithm

- Store the graph as an unweighted directed graph
- Construct a sink and source as before
- Use a BFS or DFS to find a path from the source to the sink
- Reverse the direction of each edge on the path
- Repeat the process of finding a path and reversing until no more paths exist
- The Maximum Bipartite Matching is the collection of all edges that are reversed

Code: Input and initialisation

```
#include <bits/stdc++.h>

using namespace std;

int main()
{
    vector< vector< vector< int > > > neighbours;
    int numEdges, source, sink, maxVertex = 0;

    ifstream inFile("mbm.in");
    inFile >> numEdges >> source >> sink;

    vector< vector< int > > connections;
    for (int i = 0; i < numEdges; ++i)
    {
        int a, b, f = 0;
        inFile >> a >> b;

        maxVertex = max( max(a, b), maxVertex);

        while (((int) connections.size() - 1) <= maxVertex)
        {
            vector<int> v;
            connections.push_back( v );
        } connections[a].push_back(b);
    }
    vector< vector< int > > initial(connections);
    inFile.close();
```

Code: BFS

```
while (true)
{
    bool found = false;
    vector<int> parent(maxVertex + 1, 0);

    queue<int> q;
    q.push(source);
    parent[source] = source;
    while (q.size() != 0)
    {
        int curr = q.front();
        q.pop();
        if (curr == sink)
        {
            found = true;
            break;
        }
        for (int i = 0; i < connections[curr].size(); ++i)
        {
            if (parent[connections[curr][i]] == 0)
            {
                parent[connections[curr][i]] = curr;
                q.push(connections[curr][i]);
            }
        }
    }
```

Code: Path Reversal

```
    if (!found) break;
    int c = sink;
    vector<int> path;
    while (c != source)
    {
        path.push_back(c);
        c = parent[c];
    }
    for (int i = path.size() - 1; i > 0; --i)
    {
        connections[path[i]].erase(find(connections[path[i]].begin(), connections[path[i]].end(), path[i - 1]));
    }
    connections[source].erase(find(connections[source].begin(), connections[source].end(), path[path.size() - 1]));
    for (int i = 0; i < path.size() - 1; ++i)
    {
        connections[path[i]].push_back(path[i + 1]);
    }
}
```

Code: Output

```
    int cnt = 0;
    for (int i = 0; i < initial.size(); ++i)
    {
        if (i == source || i == sink) continue;
        for (int j = 0; j < initial[i].size(); ++j)
        {
            if (initial[i][j] == sink) continue;
            if (find(connections[i].begin(), connections[i].end(), initial[i][j]) == connections[i].end())
            {
                ++cnt;
            }
        }
    }
    cout << cnt << "\n";

    return 0;
}
```

Analysis

- The BFS runs in O(E) time
- The BFS will run at most V times since the number of edges going to the sink is at most V and decreases by 1 every iteration
- Therefore this algorithm runs in O(VE)