CS2500 Project 4
Phase 2

Evan Wilcox

Due May 2, 2019

1 Motivation

Max flow is a metric that is useful across many use cases. Such as, calculating the though put of a highway or rail system or calculating how much water can be sent through a sewer system. Of course being able to calculate this measurement reliably and quickly is very important for many fields. This report will analyse the correctness of the Edmonds-Karp algorithm as well as compare its runtime to two similar max flow algorithms: push relabel and relabel to front.

2 Background

Max flow is calculated between two vertices in a graph. The max flow between those two points is the maximum through-put of the edges. For example, imagine you have a set of pipes with varying diameters. if you were to connect those pipes together the most water you can push through is restricted by the pipe with the smallest diameter. This also applies if you had multiple routes of pipes, the max flow would be the combined flow from all routes.

Edmonds-Karp is an algorithm that, when used on an undirected graph, results in the max flow from one starting vertex to an ending vertex. The algorithm calculates the flow of the shortest path from the start vertex to the end vertex and adds that to the total flow. The algorithm continues to find new paths and adds the flow from each path to the total max flow until there are no more paths.

3 Procedures

1. Express Edmonds-Karp using pseudocode.
2. Implement Edmonds-Karp, push-relabel, and relabel to front in c++.
3. Measure run times of the three algorithms to experimentally determine their run time complexity and compare to their expected run time complexity.
4. List problems encountered during development.
5. Produce a conclusion addressing the efficacy of the methods used.

4 Pseudocode

Edmonds-Karp Pseudocode

```
inputs
    C[n x n] : Capacity Matrix
    E[n x n] : Adjacency Matrix
    s : source
    t : sink

output
    f : maximum flow

Edmonds-Karp:
    f = 0                // Flow is initially 0
    F = [n x n]          // residual capacity array
    while true:
        m, P = Breadth-First-Search(C, E, s, t, F)
        if m = 0:
            break
        f = f + m
        v = t
        while v != s:
            u = P[v]
            F[u, v] = F[u, v] - m
            F[v, u] = F[v, u] + m
            v = u
    return f
```

5 Problems Encountered

I faced trouble when trying to implement the push relabel and the relabel front algorithms, therefore, the code for both algorithms was sourced from the internet. The sources for all code is included below with the code. When implementing the Edmonds-Karp algorithm, I had trouble trying to use only an adjacency list. I eventually settled on a adjacency list and an adjacency matrix for the capacities of the edges.

6 Performance Results

To test and time the three algorithms, each algorithm was tested using the same random set of pairs of vertices. Using this metric the push relabel algorithm performed the best with the Edmonds-Karp algorithm closely behind. The last algorithm, relabel front, performed consistently worse than the other two and was more sporadic in the amount of time it required to calculate.

7 Conclusion

In this report the correctness of the Edmonds-Karp algorithm was analysed and the run time was compared to the relabel front algorithm and the push relabel algorithm. All three algorithms were tested and timed using the Missouri city graph created in the previous lab. Each algorithm determined that the max flow from Joplin to St. Louis, using 25 miles as a cutoff for determining if two cities are connected, was 33. The Edmonds-Karp algorithm performed the best compared to the other algorithms followed by push relabel then relabel front.

Appendix A - Source Code

```
/*
* Author: Evan Wilcox
* File: main.cpp Date: 5/2/19
* Class: CS 2500 section A
* Instructor : Bruce McMillan
* Brief: program used to test and time three max flow algorithms
*/

#include "EdmondsKarp.h"
#include "PushRelabel.h"
#include "RelabelFront.h"
#include "FlowNetwork.h"
#include "City.h"
#include <math.h>
#include <time.h>
#include <iostream>
#include <fstream>
using namespace std;

// Driver program to test above functions
int main()
{
    ifstream fin;
    fin.open("data.csv");

    vector<City> C;
    string name;
    float lat;
    float lon;

    // Input cites names and coordinates and put them in a vector V of Nodes.
    string line;
    while(getline(fin, line))
    {
        name = line.substr(0, line.find(','));
        lat = stof(line.substr(line.find(',')+1, 10));
        lon = stof(line.substr(line.find_last_of(',')+1, 10));

        C.emplace_back(name, lat, lon);
    };

    int end = 781;      // St. Louis
    int src = 448;      // Joplin
    PushRelabel<int> pr(C.size());

    vector<vector<int>> cap(C.size());
    for(int i = 0; i < C.size(); i++)
    {
        cap[i].resize(C.size());
    }

    EdmondsKarp ek(C.size(), false);

    float dis;

    for(int x = 0; x < C.size(); x++)
    {
        for(int y = 0; y < C.size(); y++)
        {
            dis = sqrt(pow((C[x].m_lat-C[y].m_lat), 2) + pow((C[x].m_lon-C[y].m_lon), 2))

            if(dis < 24)
            {
                pr.AddEdge(x, y, 1);
                ek.addEdge(x, y, 1);
                cap[x][y] = 1;

            }
        }
    }

    clock_t t;
    cout << C[src].m_name << ", " << C[end].m_name << endl;
    t = clock();
    ek.maxflow(src, end);
    cout << "Edmonds-Karp: " << (float)(clock() - t)/CLOCKS_PER_SEC << endl;

    t = clock();
    pr.GetMaxFlow(src, end);
    cout << "Push Relabel: " << (float)(clock() - t)/CLOCKS_PER_SEC << endl;

    t = clock();
    FlowNetwork fw(C.size(), src, end, cap);
    RelabelToFront rf(&fw);
    rf.getMaxFlow();
    cout << "Relabel to Front: " << (float)(clock() - t)/CLOCKS_PER_SEC << endl;

    return 0;
}


/*
* Author: Evan Wilcox
* File: Node.h Date: 4/23/19
* Class: CS 2500 section A
* Instructor : Bruce McMillan
* Brief: Header file for a City struct
*/
```

```
#ifndef CITY_H
#define CITY_H

#include <string>
#include <iostream>
using std::string;
using std::ostream;

/*
* Class: City
* Brief: A struct that pairs a name, latitude and longitude in to a node.
*/
struct City
{
    string m_name;  // name of the node
    float m_lat;    // latitude of the node
    float m_lon;    // longitude of the node

    /*
    * Function  : Node
    * Brief     : Constructor using a initialization list
    * Pre       : None
    * Post      : A Node object is created
    * Param n   : name to be given to m_name
    * Param lat : latitude to be given to m_lat
    * Param lon : longitude to be given to m_lon
    * Return    : None
    */
    City(string n, float lat, float lon): m_name(n), m_lat(lat), m_lon(lon){}
};

#endif


/*
* Author : Evan Wilcox
* File : EdmondsKarp.h
* Date : 5/2/2019
* Class : CS 2500
* Instructor : Bruce McMillan
* Brief : Header file for Edmonds-Karp algorithm
*/

#ifndef EDMONDS_KARP_H
#define EDMONDS_KARP_H

#include <vector>
#include <queue>
#include <iostream>
```

```
#include <assert.h>
#define INF 0x3f3f3f3f

using namespace std;

/*
* Class :   EdmondsKarp
* Brief :   EdmondsKarp algorithm implemented in a class
*/
class EdmondsKarp
{
    private :
        int n;                              // number of verticies
        vector<vector<int>> capacity;       // 2d capacity matrix
        vector<vector<int>> adj;            // adjency list
        bool debug;                         // indicates if debug mode is enabled

    public :
        /*
        * Function : EdmondsKarp
        * Brief : constructor initializes n, capacity, adj
        * Pre : none
        * Post : EdmondsKarp object is created
        * Param n : number of verticies in the graph
        * Return : none
        */
        EdmondsKarp(int n, bool d);

        /*
        * Function : addEdge
        * Brief : adds an edge to adj and capacity
        * Pre : u, v, c are ints
        * Post : a edge is added to adj and capacity
        * Param u : first vertex of edge
        * Param v : second vertex of edge
        * Param c : capacity of edge
        * Return : none
        */
        void addEdge(int u, int v, int c);

        /*
        * Function : bfs
        * Brief : breadth first search to find closest vertex
        * Pre : src and end are positive integers
        * Post : parent stores stores the graph in parent array form
        * Param src : starting vertex
        * Param end : ending vertex
        * Param parent : vector that will store the graph in parent array form
        * Return : flow of closest vertex
        */
        int bfs(int src, int end, vector<int>& parent);

        /*
        * Function : maxFlow
        * Brief : computes the max flow of the graph strored in adj and capacity
        * Pre : src and end are positive integers
        * Post : the maxFlow is returned
        * Param src : starting vertex
        * Param end : ending vertex
        * Return : max flow of the graph stored in adj and capacity
        */
        int maxflow(int src, int end);
};

#endif


/*
* Author: Evan Wilcox
* File: EdmondsKarp.cpp Date: 5/2/2019
* Class: CS 2500
* Instructor : Bruce McMillan
* Brief: Implementation file for Edmonds-Karp algorithm
*/

#include "EdmondsKarp.h"

EdmondsKarp::EdmondsKarp(int n, bool d)
{
    this->n = n;
    this->debug = d;

    capacity.resize(n);
    for(int u = 0; u < capacity.size(); u++)
    {
        capacity[u].resize(n);
    }

    adj.resize(n);
}

void EdmondsKarp::addEdge(int u, int v, int c)
{
    adj[u].push_back(v);
    adj[v].push_back(u);
    capacity[u][v] = c;
}

int EdmondsKarp::bfs(int src, int end, vector<int>& parent)
{
    fill(parent.begin(), parent.end(), -1);

    parent[src] = -2;
    queue<pair<int, int>> q;
    q.push({src, INF});

    while (!q.empty()) {
        int cur = q.front().first;
        int flow = q.front().second;
        q.pop();

        for (int next : adj[cur]) {
            if (parent[next] == -1 && capacity[cur][next]) {
                parent[next] = cur;
                int new_flow = min(flow, capacity[cur][next]);
                if (next == end)
                    return new_flow;
                q.push({next, new_flow});
            }
        }
    }

    return 0;
}

int EdmondsKarp::maxflow(int src, int end)
{
    int flow = 0;
    vector<int> parent(n);
    int new_flow;

    while (new_flow = bfs(src, end, parent))
    {
        flow += new_flow;
        int cur = end;

        while (cur != src)
        {
            int prev = parent[cur];
            capacity[prev][cur] -= new_flow;
            capacity[cur][prev] += new_flow;
            cur = prev;
        }
    }

    return flow;
}
```

Relabel Front Code
Source - https://github.com/donachys/RelabelToFront-MaxFlow/blob/master/RelabelToFront.java

```
#ifndef FLOW_NETWORK_H
#define FLOW_NETWORK_H

#include <vector>

class FlowNetwork
{

public:
    int src = 0, sink = 0, numVertices = 0;
    //edge capacities, edge flows, vertex neighbors
    std::vector<std::vector<int>> capacities, flows, neighbList;
    std::vector<int> excess, height;

    FlowNetwork(int n, int source, int snk, std::vector<std::vector<int>> &caps);
    void setSrcEnd(int src, int end);
private:
    std::vector<std::vector<int>> buildNeighborList(int n, std::vector<std::vector<int>> &
};


class RectangularVectors
{
public:
    static std::vector<std::vector<int>> RectangularIntVector(int size1, int size2)
    {
        std::vector<std::vector<int>> newVector(size1);
        for (int vector1 = 0; vector1 < size1; vector1++)
        {
            newVector[vector1] = std::vector<int>(size2);
        }

        return newVector;
    }
};


FlowNetwork::FlowNetwork(int n, int source, int snk, std::vector<std::vector<int>> &caps)
{
    numVertices = n;
    src = source;
    sink = snk;
    capacities = caps;
    flows = RectangularVectors::RectangularIntVector(n, n);
    excess = std::vector<int>(n);
    height = std::vector<int>(n);
    neighbList = buildNeighborList(n, caps);
```

```
}

void FlowNetwork::setSrcEnd(int src, int end)
{
    this->src = src;
    this->sink = end;
}

std::vector<std::vector<int>> FlowNetwork::buildNeighborList(int n, std::vector<std::vecto
{
    std::vector<int> numNeighbs(n); //temp array
    std::vector<std::vector<int>> tempNeighbList(n); //create jagged neighbor array from e
    //first count number of neighbors for each vertex
    for (int i = 0;i < caps.size();i++)
    {
        for (int j = 0;j < caps[i].size();j++)
        {
            if (caps[i][j] > 0)
            {
                numNeighbs[i]++;
                numNeighbs[j]++;
            }
        }
    }
    //allocate capacities
    for (int i = 0; i < numNeighbs.size();i++)
    {
        tempNeighbList[i] = std::vector<int>(numNeighbs[i]);
    }
    std::vector<int> counter(n); //temp array
    //store neighbors of each vertex
    for (int i = 0;i < caps.size();i++)
    {
        for (int j = 0;j < caps[i].size();j++)
        {
            if (caps[i][j] > 0)
            {
                tempNeighbList[i][counter[i]] = j;
                tempNeighbList[j][counter[j]] = i;
                counter[i]++;
                counter[j]++;
            }
        }
    }
    return tempNeighbList;
}

#endif
```

```
#ifndef RELABEL_FRONT_H
#define RELABEL_FRONT_H

#include "FlowNetwork.h"
#include <vector>
#define INF 0x3f3f3f3f

class RelabelToFront
{
private:
    FlowNetwork *fn;

public:
    virtual ~RelabelToFront()
    {
        delete fn;
    }

    RelabelToFront(FlowNetwork *flownetwork);
private:
    int push(FlowNetwork *fn, int from, int to);
    void relabel(FlowNetwork *fn, int target);
    std::vector<int> discharge(FlowNetwork *fn, int target, std::vector<int> &currents);
    /*
    *@return the maximum flow of this RelabelToFront object
    */
public:
    virtual int getMaxFlow();
private:
    void initialize();
    std::vector<int> moveIndToFrontOfAry(std::vector<int> &list, int targetInd);
};


RelabelToFront::RelabelToFront(FlowNetwork *flownetwork)
{
    fn = flownetwork;
}

int RelabelToFront::push(FlowNetwork *fn, int from, int to)
{
    int flowAmt = std::min(fn->excess[from], fn->capacities[from][to] - fn->flows[from][to
    fn->excess[to] += flowAmt;
    fn->flows[from][to] += flowAmt;
    fn->excess[from] -= flowAmt;
    fn->flows[to][from] -= flowAmt;
    return flowAmt;
}

void RelabelToFront::relabel(FlowNetwork *fn, int target)
```

```
{
    int minH = INF;
    for (int i = 0; i < fn->neighbList[target].size(); i++)
    {
        int myI = fn->neighbList[target][i];
        if (fn->capacities[target][myI] - fn->flows[target][myI] > 0 && fn->height[myI] >=
        {
            if (fn->height[myI] < minH)
            {
                minH = fn->height[myI];
            }
        }
    }
    fn->height[target] = minH + 1;
}

std::vector<int> RelabelToFront::discharge(FlowNetwork *fn, int target, std::vector<int> &
{
    //while target vertex of discharge has excess
    while (fn->excess[target] > 0)
    {
        //retrieve the neighbor we are currently inspecting
        int neighbInd = currents[target];
        //ensure that we have not looked through all neighbors
        if (neighbInd >= fn->neighbList[target].size())
        {
            //if we have, then we must relabel and reset our current inspection to the fro
            relabel(fn, target);
            currents[target] = 0;
        }
        else
        {
            //otherwise find the id of the neighbor, check if it is a valid push target an
            int neighb = fn->neighbList[target][neighbInd];
            int residualCap = fn->capacities[target][neighb] - fn->flows[target][neighb];
            if (residualCap > 0 && fn->height[target] == (fn->height[neighb] + 1))
            {
                push(fn, target, neighb);
            }
            else
            {
                currents[target]++;
            }
        }
    }
    return currents;
}

int RelabelToFront::getMaxFlow()
{
    this->initialize();
    //set up the dischargeList, does not include source or sink
    std::vector<int> dischargeList(fn->numVertices - 2);
    int ind = 0;
    for (int i = 0; i < fn->numVertices; i++)
    {
        if (!(i == fn->src || i == fn->sink))
        {
            dischargeList[ind] = i;
            ind++;
        }
    }
    //each vertex has a current neighbor being inspected
    std::vector<int> currentNeighb(fn->numVertices);
    int curVertInd = 0;
    //if we get to the end then we are finished
    while (curVertInd < dischargeList.size())
    {
        int curVert = dischargeList[curVertInd];
        int curVertH = fn->height[curVert];
        currentNeighb = discharge(fn, curVert, currentNeighb);
        if (fn->height[curVert] > curVertH)
        { //curVert was relabeled
            dischargeList = moveIndToFrontOfAry(dischargeList, curVertInd);
            curVertInd = 0;
        }
        else
        {
            curVertInd++;
        }
    }
    //solution equals excess at the sink
    return fn->excess[fn->sink];
}

void RelabelToFront::initialize()
{
    fn->height[fn->src] = fn->numVertices; //height of source = |V|
    fn->excess[fn->src] = INF; //temporarily allow source to have infinite excess to utili
    int sourceOutflowSum = 0;
    for (int i = 0; i < fn->neighbList[fn->src].size(); i++)
    {
        int vertID = fn->neighbList[fn->src][i];
        sourceOutflowSum += push(fn, fn->src, vertID);
    }
    fn->excess[fn->src] = -sourceOutflowSum;
}

std::vector<int> RelabelToFront::moveIndToFrontOfAry(std::vector<int> &list, int targetInd
{
    int vertID = list[targetInd];
    for (int i = targetInd; i > 0; i--)
    {
        list[i] = list[i - 1];
    }
    list[0] = vertID;
    return list;
}

#endif
```

Push Relabel Code
Source - https://codeforces.com/blog/entry/14378

```
#ifndef PUSH_RELABEL_H
#define PUSH_RELABEL_H

#include <vector>
#include <queue>
using namespace std;


template <class T> struct Edge {
    int from, to, index;
    T cap, flow;

    Edge(int from, int to, T cap, T flow, int index): from(from), to(to), cap(cap), flow(f
};

template <class T> struct PushRelabel {
    int n;
    vector <vector <Edge <T>>> adj;
    vector <T> excess;
    vector <int> dist, count;
    vector <bool> active;
    vector <vector <int>> B;
    int b;
    queue <int> Q;

    PushRelabel (int n): n(n), adj(n) {}

    void AddEdge (int from, int to, int cap) {
        adj[from].push_back(Edge <T>(from, to, cap, 0, adj[to].size()));
        if (from == to)
        {
            adj[from].back().index++;
        }
        adj[to].push_back(Edge <T>(to, from, 0, 0, adj[from].size() - 1));
    }

    void Enqueue (int v) {
        if (!active[v] && excess[v] > 0 && dist[v] < n) {
            active[v] = true;
            B[dist[v]].push_back(v);
            b = max(b, dist[v]);
        }
    }

    void Push (Edge <T> &e) {
        T amt = min(excess[e.from], e.cap - e.flow);
        if (dist[e.from] == dist[e.to] + 1 && amt > T(0)) {
```

```
        e.flow += amt;
        adj[e.to][e.index].flow -= amt;
        excess[e.to] += amt;
        excess[e.from] -= amt;
        Enqueue(e.to);
    }
}

void Gap (int k) {
    for (int v = 0; v < n; v++) if (dist[v] >= k) {
        count[dist[v]]--;
        dist[v] = max(dist[v], n);
        count[dist[v]]++;
        Enqueue(v);
    }
}

void Relabel (int v) {
    count[dist[v]]--;
    dist[v] = n;
    for (auto e: adj[v]) if (e.cap - e.flow > 0) {
        dist[v] = min(dist[v], dist[e.to] + 1);
    }
    count[dist[v]]++;
    Enqueue(v);
}

void Discharge(int v) {
    for (auto &e: adj[v]) {
        if (excess[v] > 0) {
            Push(e);
        } else {
            break;
        }
    }

    if (excess[v] > 0) {
        if (count[dist[v]] == 1) {
            Gap(dist[v]);
        } else {
            Relabel(v);
        }
    }
}

T GetMaxFlow (int s, int t) {
    dist = vector <int>(n, 0), excess = vector<T>(n, 0), count = vector <int>(n + 1, 0

    for (auto &e: adj[s]) {
        excess[s] += e.cap;
        }

        count[0] = n;
        Enqueue(s);
        active[t] = true;

        while (b >= 0) {
            if (!B[b].empty()) {
                int v = B[b].back();
                B[b].pop_back();
                active[v] = false;
                Discharge(v);
            } else {
                b--;
            }
        }
        return excess[t];
    }

    T GetMinCut (int s, int t, vector <int> &cut);
};

#endif
```