The Maximum-Flow problem in undirected graphs

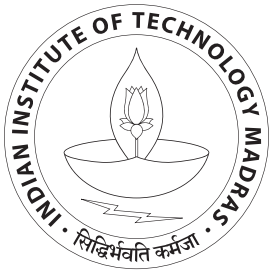
A Project Report

submitted by

KARTHIK ABINAV S

in partial fulfilment of the requirements
for the award of the degree of

BACHELOR OF TECHNOLOGY



DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
INDIAN INSTITUTE OF TECHNOLOGY, MADRAS.

May 2014

THESIS CERTIFICATE

This is to certify that the thesis entitled **The Maximum-Flow problem in undirected graphs**, submitted by **Karthik Abinav S**, to the Indian Institute of Technology, Madras, for the award of the degree of **Bachelor of Technology**, is a bona fide record of the research work carried out by him under my supervision. The contents of this thesis, in full or in parts, have not been submitted to any other Institute or University for the award of any degree or diploma.

Dr. N. S. Narayanaswamy
Research Guide
Associate Professor
Dept. of Computer Science and Engineering
IIT-Madras, 600 036

Place: Chennai

Date:

ACKNOWLEDGEMENTS

I would like to thank Prof. Narayanaswamy for guiding me through the course of this project and giving me pointers to various reading material whenever I needed them. The discussions with him during the course of project enhanced my understanding of this topic and shaped my new ideas. I would also like to convey my sincere gratitude to the other theory professors in the department namely Prof. Jayalal Sarma, Prof. Ragavendra Rao and Prof. Pandu Rangan for the various courses I took with them which helped me enhance my interest towards this subject.

I would also like to thank my parents and brother for their constant support during the ups and downs of my life. I would also like to thank my wingmates and classmates for their constant encouragement and help during my years of stay here. And finally, I would also like to thank the department for providing an opportunity to conduct research as undergraduate, which greatly helped me in making an informed career choice.

ABSTRACT

KEYWORDS: Graph Theory, Maximum Flow, Laplacian solvers

Maximum flow problem has been a very important optimization problem in computer science and mathematics. This problem has a lot of practical relevance. Some of the age-old applications involving maximum flow have been in electrical circuits, water supply networks, etc. With the advent of social media and social networks, this problem has found a newer practical relevance. And since the graphs in these networks are typically very large, researchers are sought after creating faster and more efficient algorithms for this problem.

Some of the classical algorithms to solve this problem are the Ford-Fulkerson's augmenting path algorithm and the Dinic's Algorithm. These algorithms compute the exact value of the maximum flow. It is also fairly straightforward to obtain the optimal flow vector after the termination of the algorithm. The main drawback with this algorithm is that the running time, though polynomial, is very high and is expensive to use in many practical situations. Following these algorithms a series of algorithms based on push-relabel and blocking flow techniques were devised which had a slightly better running time as compared to the classical algorithms. The series of algorithms terminated with the algorithm by Goldberg-Rao which gave a O($\min(n^{\frac{2}{3}}, m^{\frac{1}{2}}) \log(\frac{n^2}{m}) \log U$)-time algorithm for edge capacities in $\{1, 2, \ldots, U\}$. For extremely large graphs, as in the case of social networks, this algorithm is still far from being practical.

In 2008, the breakthrough result by Spielman and Teng gave an algorithm to solve the Symetric Diagonally Dominant(SDD) system of equations in near-linear time. This work was immediately extended by Koutis-Miller-Peng which gave an efficient algorithm to produce an incremental graph

sparsifier. Development of these techniques led to the development of an almost linear time algorithm to the maximum flow problem by Cristiano-Kelner-Madry-Spielman-Teng. This involved looking at the graph as a resistive network, approximating the electrical flow and producing an approximate s-t flow from this approximate electrical flow. This algorithm broke the running time barrier of Golderg-Rao and gave the first almost-linear time algorithm.

In this thesis, we give a survey of the above algorithms for the maximum flow problem. We first start off by giving a brief description of the classical algorithms. We then conclude this by giving a completely recursive specification for the push-relabel algorithm. We then present the required tools from spectral graph theory and linear algebra that is required to understand the almost linear time algorithm. We then present the Cristiano-Kelner-Madry-Spielman-Teng algorithm in detail. Finally, we give some arguments to justify the requirement of the weight updates and also show some steps that can possibly lead to making the algorithm parallel.

TABLE OF CONTENTS

ABBREVIATIONS

IITM	Indian Institute of Technology, Madras
LP	Linear Program
BFS	Breadth First Search
DFS	Depth First Search
SCC	Strongly Connected Components
SDD	Symmetric Diagonally Dominant
CKMST	Cristiano-Kelner-Madry-Spielman-Teng Algorithm
KMP	Koutis-Miller-Peng Solver

NOTATION

m	The number of edges in a graph G
n	The number of vertices in a graph G
u	A $m * 1$ matrix containing the capacities of the edges
r	A $m * 1$ matrix containing the resistances of the links in the network
U	This defines the ratio between the highest and lowest capacity in the graph, i.e. $\frac{\max\limits_{e} u_e}{\min\limits_{e} u_e}$
R	This defines the ratio between the highest and lowest resistance in an electrical network, i.e. $\frac{\max\limits_{e} r_e}{\min\limits_{e} r_e}$
A^{-1}	$A^{-1} = (A^T.A)^{-1}.A^T$ (psuedo inverse) whenever A is a rectangular matrix.

CHAPTER 1

INTRODUCTION

Maximum-Flow problem is an age old optimization problem in mathematics and computer science. Hundreds of researchers have investigated this problem and have given some interesting results. This problem has far-reaching applications in almost every field of engineering and sciences. Hence, this problem is of great importance, not just theoretically, but also in real-world applications. Devising better and more efficient algorithms for this problem and its variants has always been the pursuit. Inspite, of having had so much research into this problem, for many years we were far from getting an algorithm that runs in time that is good for practical purposes. This gives an indication of the hardness of this problem.

1.1 Basics

Graph

A *graph* G(V, E, ρ) is defined as a mathematical structure on the set of vertices V and the set of edges E and an adjacency function $\rho : V \times V \rightarrow E$, such that $\rho(a, b) = e$ for $a, b \in V$ and $e \in E$ tells that there exists an edge e between the vertices a and b.

Undirected Graph

An *undirected graph* is a graph where the function ρ is symmetric, i.e. $\rho(a, b) = \rho(b, a)$.

Directed Graph

An *directed graph* is a graph where the function ρ is not necessarily a symmetric function.

Capacitated Graph

A *capacitated graph* is a graph defined as G(V, E, ρ, ψ), where the first three values in the tuple mean the same as before. The ψ in the definition is a function $\psi : E \to \mathbb{R}$, which assigns a real number corresponding to every edge e in the graph. This real number is called the *capacity* of the graph.

s-t Flow

A *s-t flow* is a vector $f \in \mathbb{R}^{|E|}$ such that the following two criteria hold:

- Flow Conservation:

$$\sum_{e \in E^+(v)} f(e) - \sum_{e \in E^-(v)} f(e) = 0 \quad \forall v \in V \setminus \{s, t\}$$

  where $E^-(v)$ denotes the set $\{a : \rho(a, v) \text{ is defined}\}$ and $E^+(v)$ denotes the set $\{a : \rho(v, a) \text{ is defined}\}$

- Capacity maintenance:

$$f(e) \leq \psi(e) \quad \forall e \in E$$

Additionally, the *value* of the flow f is a real number F such that,

$$| f | = F = \sum_{e \in E^+(s)} f(e) - \sum_{e \in E^-(s)} f(e)$$

s-t Cut

A *s-t cut* is defined as a partition of the vertex set V into U_1 and U_2 such that, $s \in U_1$ and $t \in U_2$ and the vertices in $V \setminus \{U_1, U_2\}$ is present exactly in one of U_1 and U_2.

Minimum s-t Cut Problem

Given a weighted graph G, let $\phi(\{U_1, U_2\})$ denote the sum of weights of edges such that one of its endpoints is in U_1 and the other end point is in U_2. The *minimum s-t cut problem* is to select that cut, among all possible s-t cuts, whose value of ϕ is minimized.

1.2 Maximum Flow Problem

Given a capacitated graph G(V, E, ρ, ψ), a source vertex s and a sink vertex t, the goal of the problem is to find a s-t flow such that the value of the flow is maximized among all possible s-t flows.

1.2.1 Simple Linear Program definition

This problem can be posed as a linear programming problem. The first observation is that since LP's are efficiently solvable in polynomial time, this will already give us a polynomial time algorithm. Though, this is helpful, this is not the main goal. The dual to the linear program has a specific interpretation in graph theory. Hence, one can devise potentially more faster algorithms by using some primal-dual methods.

$$\max \sum_{u:(s,u)\in E} f((s,u))$$

subject to

- $\forall v \in V \setminus \{s,t\} \sum_{(u,v)\in E} f((u,v)) = \sum_{(v,w)\in E} f((v,w))$
- $\forall (u,v) \in E\ f((u,v)) \leq c((u,v))$
- $\forall (u,v) \in E\ f((u,v)) \geq 0$

Note that a tuple (a, b) represents an edge e whose end points are a and b.

Clearly, the vector $f \in \mathbb{R}^{|E|}$ forms the set of variables in this LP. The constraints are given to ensure that the set of solutions are the set of valid s-t flows. hence, a constraint on the flow conservation on f and the capacity maintenance on f. Hence, for the given problem instance, the number of variables are polynomial in $\mid E \mid$ and $\mid V \mid$ and the number of constraints are also polynomial in $\mid E \mid$ and $\mid V \mid$.

Sometimes, an alternative formulation of the LP is given for this problem.

$$\max \sum_{i \in P} f(i) - \lambda(\sum_{v \in V} F(v))$$

subject to:

- $\forall (j,k) \in E \; \gamma((j,k)) \leq c((j,k))$
- $\forall i \in P$,
$$f(i) \leq \gamma((j,k)) \quad \forall (j,k) \in i$$
- $\forall v \in V$,
$$F(v) = \sum_{(j,k) \in E_{in}} \gamma((j,k)) - \sum_{(j,k) \in E_{out}} \gamma((j,k))$$
- $f \succeq 0$, $F \succeq 0, \gamma \succeq 0, \lambda \geq 0$

Here, f is a vector from $\mathbb{R}^{|P|}$, where $\mid P \mid$ is the number of s-t paths in the graph. F is a vector from $\mathbb{R}^{|V|}$ and γ is a vector from $\mathbb{R}^{|E|}$. $f(i)$ denotes the flow that goes through the path i. $F(v)$ denotes the flow that is accumulated in the vertex v and $\gamma((i,j))$ denotes the flow that flows through the edge (i,j).

Clearly, the number of variables and the number of constraints are both exponential. Hence, we can't really hope to solve this in polynomial time. The only advantage of this formulation is that a optimal solution to this LP, helps one visualize the amount of flow that goes through each path independently. This formulation helps understanding of some of the path augmenting algorithms easier.

1.2.2 Dual of the Maximum Flow LP

The dual to the maximum flow LP has a very interesting connection to graph theory. In fact, as observed ahead, the dual is a relaxation to the *Minimum Cut Problem*.

$$\min \sum_{(u,v) \in E} c((u,v)) * y((u,v))$$

subject to:

- $\sum_{(u,v)\in p} y((u,v)) \geq 1 \qquad \forall p \in P$
- $y((u,v)) \geq 0 \qquad \forall (u,v) \in E$

Here, y is the dual vector corresponding to the constraints in the primal. Hence, $y \in \mathbb{R}^{|E|}$. Here P is the set of all s-t paths in the graph.

From the constraints, we can now interpret this dual as the relaxation to the s-t minimum cut problem. The vector y can be interpreted as the weight assigned to each edge in the graph. For every s-t path, the sum of edge weights in that path should be atleast one. And the objective is to minimize the weighted sum of these edge weights given by the vector y. Hence, one optimal solution to this above LP will consist of assigning a weight of 1 to the edge with minimum c in every s-t path and 0 to the other edges. This, in fact, gives the value of the minimum s-t cut in the graph and the edge with weights 1 forms the edge across the cut.

Now, consider the following randomized rounding procedure to find a cut (U_1, U_2) such that $\phi((U_1, U_2)) \leq OBJ$ where OBJ is the value of the objective for the dual, for a given vector y.

Let the values given by the vector y be weights of the edge. Now, consider any shortest path from s to v, and let d(v) be that shortest distance for vertex v. Choose a value of h uniformly at random from $[0, 1)$. Consider the set $R = \{v : d(v) \leq h\}$. Clearly, from the constraints above, corresponding to any feasible solution to the dual, t will not be contained in the set R and s will be contained in the set R. Hence, R forms a valid s-t cut.

From linearity of expectation, we get

$$\mathbb{E}[\phi(R)] = \sum_{(i,j)\in E} c((i,j))\mathbb{P}[i \in R \wedge j \notin R]$$

$$\mathbb{P}[i \in R \wedge j \notin R] = \mathbb{P}[d(i) \leq h < d(j)] = d(j) - d(i)$$

Also, from triangle inequality note that

$$d(j) \leq d(i) + y((i,j))$$

Hence, $\mathbb{E}[\phi(R)] \leq OBJ$ and therefore, there exists a cut (U_1, U_2), such that $\phi((U_1, U_2)) \leq OBJ$.

Hence, the dual to the maximum flow problem is a relaxation of the s-t minimum cut problem.

1.3 Application of Maximum Flow problem in Computer Vision

In this section, we consider the problem of Energy Minimization, which is a problem very well studied in the computer vision community. Following the lines of [insert link to paper], we show how this particular problem can be posed as a maximum flow problem. Hence, this will strengthen the importance of this problem outside the theory community.

The energy function considered in the computer vision community [Link to grieg, et al paper] is usually represented as :

$$E(L) = \sum_{p \in P} D_p(L_p) + \sum_{(p,q) \in N} V_{p,q}(L_p, L_q)$$

where $L = \{L_p : p \in P\}$ is a labeling of the image P, D_p is a data penalty function, $V_{p,q}$ is an interaction potential, and N is a set of all pairs of neighboring pixel to the current pixel.

Assume the case of binary coloring. We want to colour each pixel with either a black or a white label. The graph is constructed as follows:

- Each pixel in the image forms a vertex in this graph.

- Additionally, two vertices are added, the vertex corresponding to a black label(s-vertex) and the vertex corresponding to a white vertex(t-vertex).
- The edges are classified into two types - the T-links and the N-links.
  - The N-links are edges among neighboring pixels in the image. The cost of these N-links are the penalty assigned for the discontinuity between the pixels. This can be obtained from the pixel interaction term $V_{p,q}$ in the energy equation.
  - The T-links, are the edges between the pixel and the terminal node which corresponds to either of the two colours. It signifies the penalty of assigning the label corresponding to the terminal to that pixel. This is obtained from the term D_p in the energy equation.

  Any s-t cut partitions the pixels into two disjoint groups. Some vertices present in the same partition as the black label vertex and other vertices present in the same partition as the white label vertex. Hence, for an appropriate setting of the weights based on the parameters of an energy, a minimum s-t cut cost will correspond to a labeling with the minimum value of this energy. [link to that paper]

CHAPTER 2

Classical Algorithms

In this chapter, we describe some of the classical algorithms to find the exact value of the maximum flow in an undirected graph. These algorithms have been widely used and give a lot of insight into the nature of the problem.

2.1 Ford-Fulkerson Augmenting Path Algorithm

This is the probably the most widely known algorithm for this problem. This algorithm falls into the class of primal-dual algorithms.

Algorithm 2.1: Ford-Fulkerson algorithm to compute the maximum flow

for *every edge (i,j)* $\in E$ **do**
    Initialize f((i,j)) = 0;
end
while $\exists$ *a path p from s to t in the residual network* G_f **do**
    $c_f(p) = \min\{c_f((i,j)) : (i,j) \in p\}$;
    for *each edge (i,j)* $\in p$ **do**
        if *(i,j)* $\in E$ **then**
            f((i,j)) = f((i,j)) + $c_f(p)$;
        else
            f((i,j)) = f((i,j)) - c_f(p);
        end
    end
end

The running time of this algorithm mostly depends the augmentation method to obtain the residual network. We can show that for a suitably ill-set of edge capacities, the number of iterations taken

by the above algorithm can be very large if the augmenting paths are not chose carefully. Dinic's algorithm precisely solves this part by choosing the shortest augmenting path in each iteration and hence, bounding the overall running time to $O(V^2E)$.

2.2 Dinic's Algorithm

This algorithm is almost identical to the Ford-Fulkerson algorithm, except for the fact that it chooses the shortest augmenting path using a Breadth-First search.

Algorithm 2.2: Dinic's algorithm to compute the maximum flow

for *every edge (i,j)* $\in E$ **do**
    Initialize f((i,j)) = 0 ;
end
while $\exists$ *a* ***shortest unweighted path p*** *from s to t in the residual network* G_f **do**
    $c_f(p) = \min\{c_f((i,j)) : (i,j) \in p\}$;
    for *each edge (i,j)* $\in p$ **do**
        if *(i,j)* $\in E$ **then**
            f((i,j)) = f((i,j)) + c_f(p);
        else
            f((i,j)) = f((i,j)) - c_f(p);
        end
    end
end

The part of the algorithm that is in bold is the only difference between this algorithm and Ford-Fulkerson's algorithm. The advantage however is that now, the number of residual graphs generated is bounded independent of the edge capacities.

The above algorithms were the most efficient known for a long time in the literature until, Goldberg and others subsequently introduced the class of algorithms known as the push-relabel

algorithms. The algorithms above always maintained edge capacities, while violating the reservoir capacity at the vertices. Push-relabel algorithms on the other hand, do the converse. In every iteration, the flow is conserved at every vertex, whereas the edge capacity may be violated. And hence, the algorithm progresses towards "correcting" these violations at the edges.

2.3 Push-Relabel Algorithms

In this section, we will describe the general template of any push-relabel algorithm and present a fully recursive specification of the relabel-to-front algorithm. A completely recursive specification hence, allows programmers to directly to adapt it into languages such as prolog.

Generic Template of the push-relabel algorithm

In this section we will describe a generic template for any push-relabel algorithm. This is adapted from the text given by [bibreference to cormen]

Associate with every vertex in the graph two parameters, namely potential and reservoir.

- The potential is a measure of knowing the direction in which particular flow should be sent. In some sense, it is used to measure the progress of the algorithm between two iterations of the algorithm. The flow is always sent from a vertex of higher potential to a vertex of lower potential.

- The reservoir stores the excessive flow that has been sent to this vertex. Since, in this algorithm the flow conservation is maintained, some flow that has already been sent to this vertex cannot be routed further ahead. Hence, this keeps track of the excessive flow that accumulates at every vertex.

Algorithm 2.3: Initializing the preflow

```
begin INIT_PREFLOW (G,s):
    for every vertex v ∈ V do
        potential(v) := 0 ;
        reservoir(v) := 0 ;
    end
    for each edge (i,j) ∈ E do
        f((i,j)) := 0 ;
    end
    potential(s) = n ;
    for each vertex v ∈ N(s) do
        f((s,v)) = c((s,v)) ;
        reservoir(v) = c(s,v) ;
        reservoir(s) = reservoir(s) - c((s,v)) ;
    end
end
```

The above procedure initializes the required variables and vectors before the start of the algorithm.

Overflowing vertex : We say a vertex v is *overflowing*, if the reservoir at vertex v has positive flow accumulated in it, i.e. reservoir(v) > 0.

As the name suggests, at every iteration we associate two types of operations, a push operation and the relabel operation.

- *Push operation*: This operation is performed on vertices u and v by pushing some of the accumulated flow at u to vertex v. This is applicable only when the vertex u is overflowing, $c_f((u, v)) > 0$ and potential(u) = potential(v) + 1. Here, c_f refers to the capacities in the residual network with respect to the flow f. The operation involves pushing $\min$(reservoir(u), $c_f((u, v))$) units of flow from u to v.

- *Relabel*: This operation involves re-assigning the potential of the vertex u. This is applicable only when u is overflowing, and $\forall v \in V$ such that $(u, v) \in E_f$, we have potential(u) $\leq$ potential(v). Here, $E_f = \{(i, j) \in E : f((i, j)) < c((i, j))\}$. The operation involves re-assigning the potential of this vertex u with one more than the minimum of all such v.

Algorithm 2.4: Push Operation

begin PUSH(u, v):

$\Delta_f((u, v)) = \min($reservoir(u), $c_f((u, v)))$;

if *(u,v)* $\in E_f$ **then**

f((u,v)) = f((u,v)) + $\Delta_f((u, v))$;

else

f((v,u)) = f((v,u)) - $\Delta_f((u, v))$;

end

reservoir(u) = reservoir(u) - $\Delta_f((u, v))$;

reservoir(v) = reservoir(v) + $\Delta_f((u, v))$;

end

Algorithm 2.5: Relabel Operation

begin RELABEL(u):

potential(u) = 1 + $\min\{$potential(v) : $(u, v) \in E_f\}$;

end

With the above two functions available, a generic push-relabel algorithm looks as follows:

Algorithm 2.6: Push Relabel Template

begin MAX-FLOW(G, s, t):

INIT_FLOW(G, s) ;

while $\exists$ *an applicable push or relabel operation* **do**

Call the appropriate push or relabel operation ;

end

end

The proof of correctness of this algorithm can be shown by using the potential function on the vertices. We omit the proof here. The reader is encouraged to read [bib to cormen] for more details.

Based on the specific implementation of the template i.e. the order of push and relabel operations performed, we can bound the running time of the algorithm. In the following sub-section, we give a fully recursive specification of the relabel-to-front algorithm and bound the running time of this algorithm.

2.3.1 Fully Recursive Specification of the Relabel-To-Front Algorithm

Relabel-To-Front is a specific implementation of the generic push-relabel template. In this algorithm, a list of vertices is maintained during the run of the algorithm. At any point, the lis is traversed from left to right, looking for an overflowing vertex and a "discharge" operation is performed on it. A discharge operation either performs a push or a relabel. If the relabel operation is performed on a vertex v, it is then brought to the front of the list. Hence, the name "Relabel-To-Front".

The algoritm contains one key procedure known as the "discharge" procedure. The procedure performs the action as follows:

- If there is no more vertices left in the list of vertex u 's neighbour, i.e. reached the end of the neighbour list, a relabel operation is performed on u.
- If a neighbour v is found such that (u, v) is an admissible edge, then a push operation is performed along that edge.
- If both above doesn't happen, it just moves to the next element in the list of neighbours of u.

Note: pointer(u) represents the position of the pointer in the list currently. And neighbourList(u) represents list that maintains the vertices which neighbour u. And head(list) represents the first element of the list.

Algorithm 2.7: Discharge procedure of Relabel-To-Front algorithm

begin Discharge (u,v) :

if *reservoir(u)* > 0 **then**

return ;

end

if *v is NULL* **then**

RELABEL(u) ;

pointer(u) = head(neighbourList(u)) ;

else if $c_f((u,v)) > 0$ *and* $potential(u) = potential(v) + 1$ **then**

PUSH(u,v) ;

else

pointer(u) = pointer(u) + 1 ;

end

Discharge(u, pointer(u)) ;

end

Now, we need another driver function which calls the discharge function on required set of vertices which will complete the specification of this algorithm.

Algorithm 2.8: Recursive specification for relabel-to-front algorithm

begin Relabel-To-Front (s,t) :

INIT_PREFLOW(G, s) ;

Append $V \setminus \{s,t\}$ to mainList ;

$\forall\ v \in V \setminus \{s,t\}$, Set pointer(v) = head(neighbourList(v)) ;

u = head(mainList) ;

ComputeFlow(u) ;

end

The procedure compute flow recursively calls the discharge function on different set of vertices

and reorganizes the list. More formally, it looks as follows

Note: Follow(list, v) is the vertex that follows vertex v in the list l.

Algorithm 2.9: Recursive computation of flow

```
begin ComputeFlow (u):
    if u is NULL then
        return ;
    end
    Initialize currentPotential as potential(u) ;
    Discharge(u) ;
    if potential(u) > currentPotential then
        Remove u from mainList and add it to the head of the list ;
    end
    Set u as Follow(mainList, u) ;
    computeFlow(u) ;
end
```

This completes the specification of the algorithm. The following proposition proves the bound on the running time.

Proposition 1. *The above Relabel-To-Front runs in time O(n^3)*

Proof: We give a rough idea of the proof. For complete details, the reader is encouraged to refer [link to cormen].

Define "phase" as number of operations between two calls to relabel function. Number of relabel operations is O(n^2). Number of calls to discharge function between two phases is O(n). Next, bound the number of operations taken by discharge function and hence prove overall running time is O(n^3).

2.4 Goldberg-Rao Blocking Flow Algorithm

In this section, we briefly describe the idea behind the Goldberg-Rao's algorithm. The description of this algorithm is adapted from [link to the lecture notes containing the algorithm]

Before describing the main algorithm, let us first have some definitions in place.

Delta indicator: We define a vector L, which is a delta indicator for a particular flow f i.e.

$$L((i,j)) = \begin{cases} 1 & u_f((i,j)) \leq \Delta \\ 0 & \text{otherwise} \end{cases}$$

Admissible Edge: We call an edge $(i,j) \in E_f$ admissible with respect to the flow f, if dist(i) = dist(j) + L_f((i,j)). Here, the dist(v) refers to the shortest distance of vertex v to sink t.

Blocking flow: A flow f is *blocking* if every s-t path in the graph G has atleast one completely saturated edge.

Algorithm 2.10: Goldberg-Rao Algorithm

begin Goldberg-Rao(G, s, t):

Initialize F as mU, where U = $\max_{(i,j)\in E} u((i,j))$;

Initialize f as 0 ;

Define Λ as $\min(m^{\frac{1}{2}}, n^{\frac{2}{3}})$;

while $F \geq 1$ **do**

Assign Δ as $\frac{F}{2*\Lambda}$;

for i:=1 to $5*\Lambda$ **do**

- Compute the Delta indicator vector L ;

- Compute dist(i), using the vector L as the vector of edge lengths ;

- Shrink the strongly-connected components of admissible edges ;

- Find a flow $\tilde{f}$ in the shrunken graph s.t. it is either a blocking flow or value of the flow is $\frac{\Delta}{4}$;

- Assign $\bar{f}$ as $\tilde{f}$ with the flows completely routed within the edges of the shrunken components.;

- $f = f + \bar{f}$;

end

F = $\frac{F}{2}$;

end

end

In the above algorithm, *shrinking* a strongly connected component(SCC) means, replacing the entire SCC by a single vertex, whose incoming edges are the incoming edges of all vertices of the SCC, and whose outgoing edges are outgoing edges of all vertices of the SCC.

As again, we encourage the reader to refer to [link to the notes] for a complete analysis of the correctness of the above algorithm and its running time.

CHAPTER 3

Spectral Graphs

In this chapter, we give some requisite mathematics from spectral graph theory. The definitions and theorems stated in this chapter, will be the key to understanding the breakthrough Cristiano-Kelner-Madry-Spielman-Teng algorithm and our attempts and observations. We only give the required amount of definitions and would avoid most proofs. This section is adapted from the monograph [link to lx=b]. For a detailed explanation of some of these topics, the reader is advised to refer to [link to Lx=b].

3.1 Eigen Values

Throughout this report, we will be dealing with square matrices which are symmetric unless explicitly mentioned otherwise.

Eigen value: A value λ is called the *eigen value* of the matrix A if there exists a vector x such that $A.x = \lambda.x$. This vector x is called the *eigen vector* of the matrix A.

Proposition 2. *Let $\lambda_1 \leq \lambda_2 \leq \ldots \leq \lambda_n$ be the eigen values of a $n * n$ matrix A. Let $v_1, v_2, \ldots, v_n$ be the corresponding eigen vectors. Then, the matrix A can be written as $\sum_{i=1}^{n} \lambda_i v_i v_i^T$*

Alternate characterization of eigen values:

The k^{th} smallest eigen value can be expressed as

$$\lambda_k(A) = \min_{x \in \mathbb{R}^n \setminus \{0\} \,:\, x^T.v_i = 0 \;\; \forall i \in \{1,2,\ldots,k-1\}} \frac{x^T A x}{x^T x}$$

Alternatively, it can also be expressed as:

$$\lambda_k(A) = \max_{x \in \mathbb{R}^n \setminus \{0\} \,:\, x^T.v_i = 0 \;\; \forall i \in \{k+1, k+2, \ldots, n\}} \frac{x^T A x}{x^T x}$$

3.2 Graph Laplacian

Graph Laplacian is a $n * n$ matrix which gives lot of algebraic properties of a graph. This matrix is useful for testing many properties of a graph algebraically. For example, one can test connectivity of a graph using the eigen values of this matrix. In fact, we can also extend this and find the number of connected components in the graph by using the eigen values.

Let A be the adjacency matrix of the graph and let D be a diagonal matrix which contains the degree of the vertex i in the i^{th} entry of the principal diagonal and 0 elsewhere. The *graph laplacian* L is defined as $D - A$.

Proposition 3. *The graph Laplacian L is a positive semi-definite matrix i.e.* $\lambda_1(L) \geq 0$.

In fact, we can also claim that graph laplacian is not positive definite. It can be easily shown that $\lambda_1(L) = 0$.

The above definition also translates to a weighted graph, where every edge of weight w is replaced by w parallel edges and use the same definition as above.

One important advantage of this laplacian is that we can test connectivity properties using the eigen values of this matrix. The following theorem gives the exact relationship.

Theorem 1. *A graph G is connected if and only if the second smallest eigen value of its laplacian is strictly greater than 0.*

In fact, a more generalized version of the above theorem can be shown.

Theorem 2. *Let G be a graph and L be the corresponding laplacian matrix. Let* $\lambda_1(L) \leq \lambda_2(L) \leq \ldots \leq \lambda_n(L)$ *be the eigen values of L. Let* i *be such that* $\lambda_i(L) = 0$ *and* $\lambda_{i+1}(L) > 0$*. Then, the graph G has exactly* i *connected components.*

Plugging $i = 1$ in the above equation gives the theorem on connectivity of the graph.

3.3 Laplacian system of equations

A special subset of system of linear equations is a laplacian system of equations. In this case $A.x = b$ system, has the matrix A as a graph laplacian corresponding to some graph G. Hence, these are equations of the form $L.x = b$. It turns out that some optimization problems over graphs can be posed as a laplacian system of equations. In fact, one such procedure is used in the Cristiano-Kelner-Madry-Spielman-Teng algorithm and hence, being able to solve this system effectively is a requirement.

We know that a gaussian procedure to solve $A.x = b$ takes about $O(n^3)$ steps. This is not practically relevant since we are looking for flow algorithms that are close to linear time. But, it turns out that for the case of laplacian systems there exists nice solvers which can solve it efficiently. The following theorem states this more precisely.

Theorem 3. *There exists an algorithm for solving the Laplacian system which takes as input a graph laplacian L, a vector b and an error parameter* ϵ *and returns a solution* x *to the system such that:*

$$\|x - L^{-1}b\|_L \leq \epsilon \|L^{-1}b\|_L$$

Additionally, this algorithm runs in time $\tilde{O}(m \log(\frac{1}{\epsilon}))$ where m is the number of non-zero en-

tries in the laplacian matrix.

We would like to direct the reader [link to olivia simpson's research exam] for a brief survey on the development of various Laplacian solvers.

3.4 Electrical Flows

The key idea leading to the breakthrough of the newer algorithm was to visualize the graph as an electrical network. Every edge corresponds to a resistor of some appropriate value. A voltage of one unit is applied across the vertices s and t. The network thus formed is an electrical network.

In this section, we will formalize the above notion and provide some useful definitions and theorems.

Incidence Matrix:

Consider an arbitrary orientation of the edges in an undirected graph G. We define a $m * n$ matrix B, which is called the *incidence matrix*, as follows:

$$B(i,j) = \begin{cases} -1 & \text{if edge i leaves vertex j} \\ 1 & \text{if edge i enters vertex j} \\ 0 & \text{otherwise} \end{cases}$$

The graph laplacian can now be related to this incidence matrix B by the following theorem.

Theorem 4. *For a graph G having an incidence matrix B and a laplacian L,* $B^T B = L$.

Note in the above theorem, there is no mention of the orientation of the edges for B. Hence, it is true for any arbitrary orientation of the edges.

Current vector: As stated before, we can look at a graph as an electrical network. Replace every undirected edge by a resistor of resistance 1. Now, associate a voltage difference of v across the vertices s and t. The current flowing through each of the resistors(which are edges in the original graph) is given by the *current vector*. It is denoted as c.

Since, the above is an electrical network, we know that the current and voltage satisfies the Kirchoff's current and voltage laws. Using the laws, we can easily show that, $B^T B.v = c$ and hence, $L.v = c$. Therefore, given the current flowing through each of the edges and the laplacian, finding the vertex potentials is essentially solving a laplacian system of equations.

Effective Resistance: Given a particular current vector c, we define the effective resistance of an edge $e = (i, j)$ as the potential difference across e when a current of value 1A is sent across i to j.

Proposition 4. *The effective resistance $R_{eff}(e) = (e_i - e_j)^T L^{-1}(e_i - e_j)$. Here, e_i is a vector of length $\mid V \mid$ which has 1 at the i^{th} location and 0 elsewhere.*

We would like to know what is the voltage across a particular edge, when a current of 1A is put across a fixed set of two vertices s and t. To do that we will now define the following matrix called the Π *matrix*.

Π **matrix**:

This matrix is defined on $\mathbb{R}^{|E|*|E|}$.

$$\Pi(e, f) = b(e)^T L^{-1} b(f)$$

$b(e)$ is the e^{th} row in the matrix B.

Hence, $\Pi = B^T L^{-1} B$. It is easy to check the following properties of this matrix

- Π is symmetric

- $\Pi^2 = \Pi$
- All eigen values of Π are 0 or 1

Electrical flow: Let 1A of current be inducted at vertex s and removed at vertex t. We define f^* as the electrical flow and is given by $BL^{-1}(e_s - e_t)$.

Electrical energy: Let r be the vector of resistances of every edge(Here it is the all 1's vector). Let f be the vector of electrical current through each edge. The electrical energy is

$$\epsilon_r(f) = \sum_e r_e f_e^2$$

Lemma 1. *Among all valid flows of value F, the one that minimizes the energy is the electrical flow.*

Proof: Consider the case of two resistors in series and two resistors in parallel. If we prove the lemma for these two cases, it applies for any general circuit.

Series: In this case, there is just one possible flow, which is the electrical flow. Hence, statement trivially holds true.

Parallel: Let us start with an electrical flow. We will show that any change to the flows will only increase the energy. Let i_1 and i_2 be the electrical flow through resistors r_1 and r_2. From Kirchoff's law we have $i_1 * r_1 = i_2 * r_2$. The energy now is $\epsilon = r_1 * i_1^2 + r_2 * i_2^2$. Let us now redistribute the flow as $i_1 + \delta$ and $i_2 - \delta$. The value of the flow remains the same. The energy now becomes $\epsilon' = \epsilon + \delta^2(r_1 + r_2) \geq \epsilon$. Hence the lemma.

Proposition 5. *For r being the all 1's vector and an electrical flow f^*, the energy of this flow is given by*

$$\epsilon(f^*) = (e_s - e_t)^T L^{-1}(e_s - e_t)$$

CHAPTER 4

Cristiano-Kelner-Madry-Spielman-Teng Algorithm

In this chapter we describe the Cristiano-Kelner-Madry-Spielman-Teng algorithm[CKMST] in detail. This will aid the reader to understand the new tools used to get to a near linear running time for the maximum flow problem in undirected graphs. Later in the section we will also provide some observations and give a method which could possibly lead to a parallel version of the same. This will help in making the algorithm further robust and hence, can compute the maximum flow in even larger graphs.

4.1 Overview

The key idea of the CKMST algorithm is that given an electrical circuit, an approximate set of currents and node potentials can be calculated very efficiently using a fast SDD solver. The research on algorithms for laplacian solvers is a closely related topic. But, in this report we will avoid stating them in detail and just state the necessary theorems whenever needed. The CKMST algorithm in particular, uses the Koutis-Miller-Peng algorithm[Link to the Koutis, Miller, Peng paper] as a subroutine. Given, an approximation to the electrical currents and potentials(in short we call this the electrical flow), the algorithm, then constructs two subroutines.

- (ϵ, ρ) **- oracle**: The authors call this an oracle since, given a flow value F and a graph G whose every edge capacity is 1, this subroutine either returns a flow f whose edge capacities are violated by atmost ρ on each edge and whose average is atmost ϵ or returns a fail which indicates that the required flow value F is too large. Note, that having an upperbound on the average makes the sub-routine do more work.
- **Weights update routine**: This function assigns different weights to each edge in each iteration and then uses the flow returned by the oracle for a suitable ϵ and ρ to finally return a flow which does not violate edge capacity at all edges. The main intuition behind why a flow that has violated edge capacities in some iteration can be modified to give a flow that does not violate any capacity is the fact that the oracle maintains the average capacity as a weighted

average over the weights used by the weights update routine. Hence, by suitably modifying the weights across iterations, one can obtain a required flow.

This algorithm achieves a $(1-\epsilon)$- approximation to maximum flow problem in undirected graphs. The main drawback with this algorithm, as we will see ahead, is that it doesn't actually give a flow vector, but merely gives the value of the flow. This issue was resolved in a follow-up work by Orecchia, Kelner et al[link to paper], which adapts a completely different approach using a variety of new techniques such as non-euclidean gradient descent, construction of oblivious routing and flow sparsifiers.

An important attempt has been to make this algorithm parallel or atleast "less" serial. To do that efficiently, a two step method is required. Firstly, the update routine itself should be modified to ensure that some updates can be done in parallel. In this regard, we make certain observations on the update procedure which will lead to a modification in making every two iterations into two parallel iterations. Although the main crux of the algorithm is in the weights update method, it is also helpful to try and make the subroutine that computes the approximate electrical flow parallel. This requires the SDD solver itself to have a parallel implementation. In this regard some progress has been made recently by [link to the new STOC paper].

4.2 A $\widetilde{O}(m^{\frac{3}{2}}\epsilon^{\frac{-5}{2}})$ time flow algorithm

In this section, first we will have a look at a simpler version of the algorithm. After taking key insights from this algorithm, we can make appropriate observations and modifications to improve the running time.

4.2.1 δ-approximate flow

This subroutine is the one which approximates an electrical flow with another flow, known as the δ-approximate flow, whose value is F. This routine uses the solver by Koutis-Miller-Peng as a black-box.

Algorithm 4.1: δ-approximate flow

Input:

$\delta > 0$

$F > 0$

Vector r of resistances in which the ratio of the largest to smallest resistance is atmost R

Output:

Vector potentials $\widetilde{\phi}$,

An s-t flow $\widetilde{f}$ of value F

begin

- Scale all the resistances between 1 and R.
- Invoke the Koutis-Miller-Peng solver to compute vertex potentials $\widetilde{\phi}$ and $\widetilde{f}$ given F, L , G
- Define $i_{ext} = B\widetilde{f}$.
- Compute flow f' in the spanning tree of the graph, with demands at each vertex as F$\psi_{s,t} - i_{ext}$.
- Return f' + $\widetilde{f}$ as the approximate electrical flow.

end

The above can be computed in time $\widetilde{O}(m \log(\frac{R}{\delta}))$.

Additionally the output satisfies the following :

- $\epsilon_r(\widetilde{f}) \leq \epsilon_r(f) * [1 + \delta]$
- For every edge e,

  $| r_e f_e^2 - r_e \widetilde{f}^2 | \leq \frac{\delta}{2mR} \epsilon_r(f)$
- $\widetilde{\phi}(s) - \widetilde{\phi}(t) \geq (1 - \frac{\delta}{12nmR}) * F * R_{eff}(r)$

The algorithms that follow will use the above algorithm as a sub-routine to obtain a δ-approximate

electrical flow, for appropriate values of δ.

4.2.2 (ϵ, ρ)- oracle

This is the second sub-routine that will be used by the algorithm. The construction of such an algorithm uses the δ-approximate electrical flow algorithm.

Algorithm 4.2: An $(\epsilon, 3\sqrt{\frac{m}{\epsilon}})$ - oracle

Input:

Real number $F > 0$

w - vector of edge weights, where $w_e \geq 1 \ \ \forall e \in E$

Output:

A s-t flow f satisfying the criteria.

begin

- Assign r_e as $\frac{1}{u_e^2}\left(w_e + \frac{\epsilon\|w\|_1}{3m}\right)$ for each edge e in the graph.
- Invoke Algorithm 4.1 using the above resistances and $\frac{\epsilon}{3}$ as the value of δ.
- Let $\overline{f}$ be the returned approximate electrical flow.

if $\epsilon_r(\overline{f}) > (1+\epsilon)\|w\|_1$ **then**
  Return "fail"

else
  Return $\overline{f}$

end

end

A generic oracle has the following properties.

- If $F \leq F^*$, output a flow f s.t.
  - $\mathsf{f} = F$
  - $\frac{\sum_e w_e * cong_f(e)}{\sum_e w_e} \leq (1+\epsilon)$
  - $\max_e cong_f(e) \leq \rho$
- If $F > F^*$, output a flow f satisfying the above or "fail".

Hence, this oracle returns a flow which relaxes the constraint criteria on every edge by a factor of ρ, but at the same time ensures that not too many edge capacities are violated by constraining the average to have an upper bound of ϵ.

4.2.3 Main Algorithm

The main algorithm calls the (ϵ, ρ) subroutine many times with different weights and then takes an average of all the values returned by the oracle as the required value of flow. Note, the flow returned by this algorithm should maintain the congestion criteria for each of the edges. On the other hand, the oracle may return a flow whose congestion at each edge is as bad as ρ. But, since the oracle gives a upper bound on the average congestion, the algorithm uses that critically to, in some sense "correct" the value of flows.

The update of weights in each iteration of the algorithm is the key to the correctness of this algorithm. One severe drawback that can be seen is that, to update the weights in the i^{th} iteration, we need the flow values from the $i - 1^{th}$ iteration. Hence, this induces a serious serial dependency on the algorithm. Hence, to try and attempt to make this algorithm parallel, one of the steps that needs to be taken is to try and modify the weights update procedure. This is where our first observation, which we later state as a theorem, will help.

Note, this algorithm requires a target flow value F as an input. Hence, to get a value of maximum flow, we can perform a recursive doubling and binary search on the value of F to get to the value F^*.

Note: The superscript on the weight vector implies the weights in the i^{th} iteration of the algorithm.

Algorithm 4.3: A $\widetilde{O}(m^{\frac{3}{2}}\epsilon^{\frac{-5}{2}})$ time flow algorithm for maximum flow

Input:

A graph G

A vector u of edge capacities

A target flow value F

(ϵ, ρ) - oracle O

Output:

Either a flow f or "fail" indicating $F > F^*$

begin

- Initialize the weights w^0 as $w_e = 1$ for all the edges e.

- Initialize N as $\frac{2\rho \ln(m)}{\epsilon^2}$

for *i =1 to N* **do**

- Query the oracle O with w^{i-1} and target flow as F

if *O returns a "fail"* **then**

return "fail"

else

- Let f^i be the value of the returned flow.

- Update weights as

$$w^i = w^{i-1} * \left(1 + \frac{\epsilon * cong_{f^i}(e)}{\rho}\right)$$

end

end

- Return f as $\frac{(1-\epsilon)^2}{(1+\epsilon)N}\left(\sum_i f^i\right)$

end

4.3 Our observations and a baby step towards parallelism

As stated earlier, the weights update procedure above inherently makes this algorithm highly sequential. In this section, we try and modify some parameters of the algorithm which gives a first step towards parallelism. We hope that this can be fully extended to make it completely parallel.

Clearly, the two key parameters of the algorithm are the the number of iterations it runs for, which is N, and the weights update mechanism. Once, these two are fixed, the final multiplicative term for the averaging of all the flows is fixed. Here, we fix these as variables and prove bounds on these variables. This is done similar to the technique with which the correctness of the CKMST algorithm was shown [link to paper].

Modification 1. *Replace the number of iterations* $\frac{2\rho ln m}{\epsilon^2}$ *with a variable N. The exact value of this will be fixed after the analysis.*

The main rationale behind changing this value is that, we try to divide the iterations among many parallel processors, such that their total sum of iterations remain the same as that of the sequential version. In some sense, we will perform the first few iterations on one processor, and the next few on another without waiting for the first few to complete and finally merge the two outputs.

Modification 2. *Modify the weights update procedure as*

$$w^i = w^{i-1} * \big(1 + \alpha * cong_{f^i}(e)\big)$$

Again, we just know that the weights should be proportional to the congestion of that edge. If a particular edge has been alloted more flow than its capacity in previous iteration, we should penalize it by having a higher weight for that edge. Hence, its flow in the next iteration will be smaller owing to the upper bound on the average.

Modification 3. *Modify the final flow as* $\overline{f} = \beta * \frac{\sum_i f^i}{N}$

Again, the old parameter that was included in the algorithm was a consequence of the particular weights update mechanism. Since, we are seeking to modify that, this also will be a new parameter for this generic weights update mechanism.

As the reader would have observed, we have now made the original algorithm less specific by replacing some of the key values with variables. We will now proceed to the analysis of this algorithm with these variables and show these modifications can possibly lead to some parallelism.

Define μ^i as $\sum_{e \in E} w^i$. Now we will use this as a potential function to give upper and lower bounds on the weights during the run of the algorithm.

Clearly, we have $\mu^0 = m$, since initially all the weights are 1.

Theorem 5. *For any iteration i,* $\mu^{i+1} \leq \mu^i e^{\alpha(1+\epsilon)}$

Proof: The proof for this is similar to the one given in [link to the paper].

$$
\begin{aligned}
&\mu^{i+1} \\
&\quad = \sum_{e \in E} w_e^{i+1} && \text{(From definition)} \\
&\quad = \sum_{e \in E} w_e^i (1 + \alpha . cong_{f^i}(e)) && \text{(from the modified weights update procedure)} \\
&\quad = \sum_{e \in E} w_e^i + \alpha \sum_{e \in E} w_e^i \, cong_{f^i}(e) \\
&\quad \leq \mu^i + \alpha(1+\epsilon) \sum_{e \in E} w_e^i && \text{(From the upper bound on average of the flow)} \\
&\quad = \mu^i (1 + \alpha(1+\epsilon)) \\
&\quad \leq \mu^i e^{\alpha(1+\epsilon)} && \text{Because } (1 + x \leq e^x)
\end{aligned}
$$

From the theorem above we can easily conclude that $\sum_{e \in E} w_e^N \leq m e^{N\alpha(1+\epsilon)}$. Also, notice that

so far we haven't placed any restriction on α. Hence, for literally any value of α the above would go through.

Theorem 6. *For any iteration i and any edge e,*

$$w_e^i \geq e^{(1-\rho\alpha).\alpha.\sum_{j=1}^{i} cong_{fj}(e)}$$

Proof: This theorem can again be proved in a similar fashion. We just need to carefully assign some restrictions on the value of α to complete it.

$$\begin{aligned}
& w_e^i \\
& = \prod_{j=1}^{i}(1+\alpha.cong_{fj}(e)) && \text{(From definition)} \\
& \geq \prod_{j=1}^{i}(\frac{cong_{fj}(e)}{\rho}+\alpha.cong_{fi}(e)) && (\frac{cong_{fj}(e)}{\rho} \leq 1, \text{ from oracle promise}) \\
& \geq \prod_{j=1}^{i}\left[\frac{cong_{fj}(e)}{\rho}(1+\rho.\alpha)\right] \\
& \text{We will place the first restriction on } \alpha. \text{ We want that } \rho.\alpha < \frac{1}{2} \\
\text{Therefore, } & \geq \prod_{j=1}^{i} e^{\left[\frac{cong_{fj}(e)}{\rho}.(1-\rho.\alpha).\rho.\alpha\right]} && \text{(Because, } (1+x) \geq e^{x(1-x)} \ \forall\, 0<x<\frac{1}{2}\text{)} \\
& \geq e^{\left[(1-\rho.\alpha).\alpha.\sum_{j=1}^{i} cong_{fj}(e)\right]}
\end{aligned}$$

Theorem 7. *For values of α, β, N, such that*

$$\alpha.\rho \leq \frac{1}{2} \quad and$$

$$\frac{\beta \log m}{(1-\rho\alpha).\alpha.N} + \frac{\beta(1+\epsilon)}{(1-\rho\alpha)} \leq 1$$

The flow returned by the modified algorithm maintains capacity constraints on all edges $e \in E$.

Proof: From theorem 6 we have,

$$\|w_e^N\|_1 \geq w_e^N \geq e^{\left[(1-\rho.\alpha).\alpha.\sum_{j=1}^{N} cong_{fj}(e)\right]}$$

From theorem 5 we have,

$$\sum_{e \in E} w_e^N = \|w^N\|_1 \leq me^{N\alpha(1+\epsilon)}$$

Combining the two, we get

$$e^{\left[(1-\rho.\alpha).\alpha.\sum_{j=1}^{N} cong_{fj}(e)\right]} \leq me^{N\alpha(1+\epsilon)} \qquad \text{.. (1)}$$

Also, we know that $\overline{f} = \beta.\frac{\sum_{i} f^i}{N}$.
Dividing by capacities on both sides of the equation, we get $\sum_{j=1}^{N} cong_{fj}(e) = cong_{\overline{f}}(e)\frac{N}{\beta}$.

Plugging this into (1) and taking log on both sides to get an upperbound on $cong_{\overline{f}}(e)$ as follows

$$cong_{\overline{f}}(e) \leq \frac{\beta \log m}{(1-\rho\alpha).\alpha.N} + \frac{\beta(1+\epsilon)}{(1-\rho\alpha)}$$

Setting an upperbound of 1 to this upperbound will ensure that the edge capacities on every edge will be maintained. This gives the required theorem.

Given the above theorems we will now show a possible approach to get to parallelism. The original algorithm uses α as $\frac{\epsilon}{\rho}$ and runs for N = $\frac{2\rho \log m}{\epsilon^2}$ steps. On the other hand, theorem 7 gives a generalized relationship between the values of α and N. Hence, our approach is to try and run many instances of the above algorithm on different values of (α, N, β) satisfying the above theorem and finally take a some combination of the outputs obtained. The most intuitive method to merge the

solutions is to take the best of the solutions. Since, all of them return an $O(1-\epsilon)$ approximation to the flow, the best solution also will be a O$(1-\epsilon)$ - approximation.

We have attempted to start off with a simplistic method to get the various tuples to be run in parallel. We first assume that we have a constant k number of processors. Hence, we will need k tuples of the form (α_i, N_i, β_i).

We use $N_1 = N_2 = \ldots = N_k = N$, where N is the value used in the original algorithm. Now, fixing N we have to choose k values of α, one for each α_i such that $\alpha_i.\rho \leq \frac{1}{2}$. This will give us corresponding k values of β, one for each β_i. Hence, in some sense, we get a k repeated trials of the same algorithm for different parameters and we choose the best outcome among these.

We call this the baby-step because, asymptotically, the approximation ratio has remained the same. We just have managed to possibly be better off in terms of the constants. We have been pursuing this approach primarily for the fact that, we believe this could be extended further to possibly logarithmic in ρ number of processors and hope to get to a closer approximation ratio. The other advantage is that this approach easily translates to the modification of CKMST algorithm. In their work, they start off with this algorithm and perform a modification to this to get to another algorithm which performs even better in terms of running time. This approach can be directly translated in that setting since, their modification still uses the same weights update procedure.

4.4 A $\widetilde{O}(mn^{\frac{1}{3}}\epsilon^{\frac{-11}{3}})$ time flow algorithm

In this section, we continue to explain the algorithm proposed by Cristiano, et al. They use the previous algorithm as a base and further improve the running time. This improvement from the previous algorithm is achieved in two steps.

- In the first step, by making some observations and hence, modifying the updates procedure and the (ϵ, ρ)-oracle a $\widetilde{(m^{\frac{4}{3}}\epsilon^{-3})}$ algorithm is obtained.
- In the second step, the randomized sparsification method, as introduced by Benczar and

Karger[link to the paper], is used to first sparsify the graph and then apply the algorithm which gives the required $\widetilde{O}(mn^{\frac{1}{3}}\epsilon^{\frac{-11}{3}})$ time flow algorithm.

The key idea to execute the first step is to notice that, once an edge has been filled to capacity, it is better to remove it altogether from the graph rather than assigning a large weight to it. This way the graph becomes smaller and smaller in subsequent iterations, hence making the remaining steps faster. They call this set of edges removed as the forbidden edges.

Algorithm 4.4: A modified (ϵ, ρ) - oracle

Input:

Real number $F > 0$

w - vector of edge weights, where $w_e \geq 1 \ \ \forall e \in E$

set H of forbidden edges

Output:

A s-t flow f satisfying the criteria and a set H' of forbidden edges.

begin

- Set ρ as $\frac{8m^{\frac{1}{3}}(\log m)^{\frac{1}{3}}}{\epsilon}$

- Assign r_e as $\frac{1}{u_e^2}\left(w_e + \frac{\epsilon\|w\|_1}{3m}\right)$ for each edge e in the $E \setminus H$.

- Invoke Algorithm 4.1 using the above resistances and $\frac{\epsilon}{3}$ as the value of δ. Invoke it on the graph G(V, $E \setminus H$).

- Let $\overline{f}$ be the returned approximate electrical flow.

if $\epsilon_r(\overline{f}) > (1+\epsilon)\|w\|_1$ *or s,t are disconnected on the set of edges* $E \setminus H$ **then**
  Return "fail"

else if $\exists\ e$ *with* $cong_{\overline{f}}(e) > \rho$ **then**
  - Add all e satisfying the condition $cong_{\overline{f}}(e) > \rho$ to the set H.

  - Restart this procedure.

else Return $\overline{f}$;

end

As the reader would have observed, this just makes a small modification to the original oracle. It just ensures that those edges which have been filled to capacity are eliminated from the graph.

Algorithm 4.5: A $\widetilde{O}(m^{\frac{4}{3}}\epsilon^{-3})$ time flow algorithm for maximum flow

Input:

A graph G

A vector u of edge capacities

A target flow value F

(ϵ, ρ) - oracle O

Output:

Either a flow f or "fail" indicating $F > F^*$

begin

- Initialize the weights w^0 as $w_e = 1$ for all the edges e.

- Initialize H as ϕ

- Initialize ρ as $\frac{8m^{\frac{1}{3}}\log^{\frac{1}{3}} m}{\epsilon}$ and N as $\frac{2\rho \log m}{\epsilon^2}$

- Initialize N as $\frac{2\rho \ln(m)}{\epsilon^2}$

for *i =1 to N* **do**

- Query the oracle O with w^{i-1}, target flow as F and the forbidden edge set as H

if *O returns a "fail"* **then**

return "fail"

else

- Let f^i be the value of the returned flow.

- Replace H with the set H' returned by the oracle.

- Update weights as

$$w^i = w^{i-1} * \left(1 + \frac{\epsilon * cong_{f^i}(e)}{\rho}\right)$$

end

end

- Return f as $\frac{(1-\epsilon)^2}{(1+\epsilon)N}\left(\sum_i f^i\right)$

end

Similarly the main weights update is modified to accommodate for the removal of forbidden

edges. As it is already clear, the weights update mechanism is untouched in this modified algorithm. Hence, our approach towards a parallel version can be easily adopted here.

4.4.1 Correctness and Running Time of the modified algorithm

The strategy to show the correctness is to prove a bound on u(H), which is the total capacity of the edges added to set H when the algorithm terminates. Also, the number of edges in set H is upperbounded at the end of the algorithm. More formally, the following theorem is proved.

Theorem 8. *At the end of the algorithm, the following bounds hold true:*

$$\mid H \mid \leq \frac{30m \log m}{\epsilon^2 \rho^2} \quad and$$

$$u(H) \leq \frac{30mF \log m}{\epsilon^2 \rho^3}$$

We will avoid the detailed proof of this theorem. The reader is advised to take a look at [link to paper] for the proof. We will just give the overall idea here. The key idea is to use the resistances of the network as potential function and note that edges that are added into H set are those that contribute to a significant fraction of the energy. Then, bound the maximum change in effective resistance during the run of the algorithm and hence bound H.

Given this theorem, we will briefly show how the authors prove the running time.

Theorem 9. *For $F \leq F^*$, the algorithm returns a flow $\overline{f}$, which is an (1-ϵ)- approximation to F, in time $\widetilde{O}(m^{\frac{4}{3}}\epsilon^{-3})$.*

Proof: The (ϵ, ρ) - oracle either adds edges to the set H or makes a call to algorithm 4.1. The number of calls made to this algorithm is atmost N + $\mid H \mid$. And from theorem 8 we have an upper bound on $\mid H \mid$. Plugging this value and the value of N used, we get the required running time as

(N + $\mid H \mid$)$\widetilde{O}(m\log(\frac{1}{\epsilon})) \leq \widetilde{O}(m^{\frac{4}{3}}\epsilon^{-3})$.

Now, we need to show the part where it returns a flow $\overline{f}$, whenever $F \leq F^*$.

Substitute the value ρ used in the algorithm in theorem 8. We get,

$$u(H) \leq \frac{\epsilon F}{12}$$

Hence, during the entire algorithm, the graph has a flow atleast $F^* - \frac{\epsilon F}{12} \geq (1 - \frac{\epsilon}{12})F$.

We will now show the step 2 by using the sparsification technique given by Karger et al. More formally, by direct application of the following theorem given by them, we get the required improvement.

Theorem 10. *Given a T(m,n,ϵ) algorithm to find a (1-ϵ)-approximate flow in undirected graph, it can be used to obtain a (1-ϵ)-approximate flow in time $\widetilde{O}(\frac{\epsilon^2 m}{n}T(\widetilde{O}(n\epsilon^{-2}), n, \epsilon))$ using "graph smoothing" as defined by Karger, et al.*

Hence, a direct application of this theorem with T(m,n,ϵ) = $\widetilde{O}(m^{\frac{4}{3}}\epsilon^{-3})$ gives the required bound of $\widetilde{O}(mn^{\frac{1}{3}}\epsilon^{\frac{-11}{3}})$.

REFERENCES

Amarel, S., On representations of problems of reasoning about actions. *In* **D. Michie** (ed.), *Machine Intelligence 3*, volume 3. Elsevier/North-Holland, Amsterdam, London, New York, 1968, 131–171.

Barto, A. G., **S. J. Bradtke**, and **S. P. Singh** (1995). Learning to act using real-time dynamic programming. *Artificial Intelligence*, **72**, 81–138.

Bellman, R. E., *Dynamic Programming*. Princeton University Press, 1957.

Crawford, J. (1992). A theoretical analysis of reasoning by symmetry in first-order logic. URL `citeseer.ist.psu.edu/crawford92theoretical.html`.

Griffiths, D. F. and **D. J. Higham**, *Learning LaTeX*. SIAM, 1997.

Knoblock, C. A., Learning abstraction hierarchies for problem solving. *In* **T. Dietterich** and **W. Swartout** (eds.), *Proceedings of the Eighth National Conference on Artificial Intelligence*. AAAI Press, Menlo Park, California, 1990. URL `citeseer.ist.psu.edu/knoblock90learning.html`.

Kopka, H. and **P. W. Daly**, *Guide to LaTeX (4th Edition)*. Addison-Wesley Professional, 2003.

Lamport, L., *LaTeX: A Document Preparation System (2nd Edition)*. Addison-Wesley Professional, 1994.

Manning, J. B. (1990). *Geometric symmetry in graphs*. Ph.D. thesis, Purdue University.

Ravindran, B. and **A. G. Barto** (2001). Symmetries and model minimization of markov decision processes. Technical report, University of Massachusetts, Amherst.